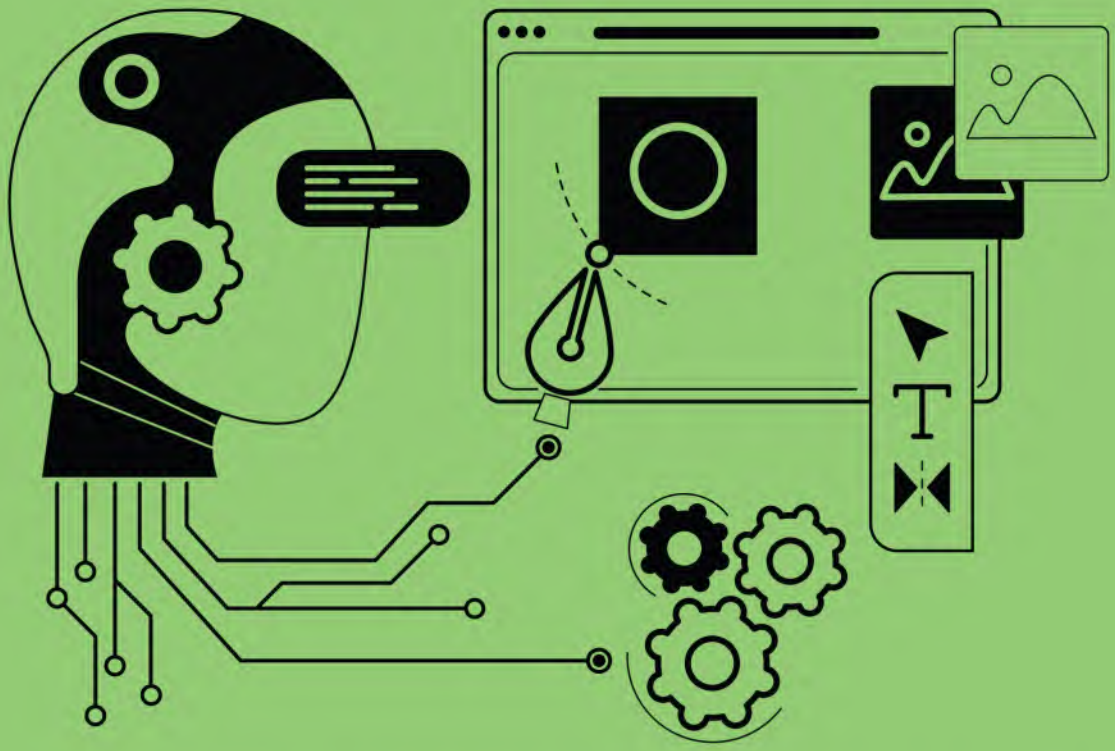
‹packt›

AI-Augmented Domain-Driven Design

Design Better Domains Using LLMs, Structured Prompts, and Multi-agent Workflows



ALBERTO ACERBIS | ALESSANDRO COLLA

Foreword by- Tobias Goeschel | Core Domain Consulting

AI-Augmented Domain-Driven Design

Design Better Domains Using LLMs, Structured Prompts, and Multi-agent Workflows

Alberto Acerbis | Alessandro Colla

packt

AI-Augmented Domain-Driven Design

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Gebin George
Relationship Lead: Dhruv J. Kataria
Project Manager: Ankit Maroli
Content Engineer: Srishty Goyal
Technical Editor: Rahul Limbachiya
Indexer: Hemangini Bari
Proofreader: Srishty Goyal
Production Designer: Ajay Patule
Growth Lead: Rithika Shetty

First published: September 2026
Production reference: 1240926

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80742-105-2
packtpub.com

To Alessandra, my unwavering partner and constant support, and to Marco and Nicolò, two wonderful sides of the same coin. This book is a testament to the love and strength you bring to my world.

– Alberto Acerbis

To my wife, Daniela, my loving partner in this incredible life journey we share. Thank you for your unwavering support, even in the smallest things. You are my refuge and my inspiration.

– Alessandro Colla

Foreword

I first met Alessandro Colla and Alberto Acerbis in the spring of 2024, during a snowstorm at the Explore DDD conference in Denver, Colorado. It was at that very conference that Eric Evans' keynote encouraged attendees to leap into the world of generative AI and explore (sic!) how our unique experience as Domain-Driven Design practitioners would change and influence the way people work with LLMs. "Let's not leave it to others to shape the future" was his urgent message.

Two and a half years later, the DDD community is at the forefront of making agentic software development sustainable, both in terms of effective resource usage and the production of quality products.

And this book, *AI-Augmented Domain-Driven Design*, is direct evidence that the DDD perspective matters.

Instead of looking for ever more ways to replace human engineers with tools of sometimes questionable reliability, Alberto and Alessandro propose a way of integrating coding agents into a team's responsible, natural flow of designing software systems that just work. Systems that work because knowledge and understanding aren't tossed aside in favor of sloppily produced code without quality checks. Systems that work because "knowledge crunching" (Evans), linguistic precision, and methodical rigor are the irreplaceable centerpiece of the development process.

With practical examples designed to work with all popular LLMs, SaaS or open source, we are taken on a journey: from humble beginnings, writing single prompts, to building our own agents, to a smartly automated, augmented production process that much more closely resembles advanced gardening than a mere code factory. Humans cull, nurture, and grow this process, whether by guiding model exploration, making architecture decisions, or observing agents' behavior and improving the harness. Alessandro and Alberto invite us to grow along with it by showing us how to use the tools and also explaining why they work.

We are still only at the beginning of a significant change brought about by AI in our industry and in society as a whole.

How we create and gather knowledge, keep it accessible and malleable for both human and AI collaborators in hybrid teams, and ensure conceptual integrity and clarity within the constraints of limited resources, including human cognitive load and the model's context size, is the key challenge to solve in the foreseeable future.

Alberto and Alessandro have just given us a significant head start.

I hope you enjoy the book!

Tobias Goeschel

Contributors

About the authors

Alberto Acerbis is a hands-on software architect and trainer with 30+ years of experience in IT. A .NET specialist, C# programmer, and Microsoft MVP, he applies domain-driven design in financial and IIoT projects. He actively contributes to open source, speaks at Microsoft and DDD conferences, and co-authored Domain-Driven Refactoring. As a co-founder of DDD Open, he promotes knowledge sharing and collaboration in the software development community.

I would like to first and foremost thank my loving and patient wife for her continued support, patience, and encouragement throughout the long process of writing this book.

Alessandro Colla is a seasoned e-commerce specialist, C# programmer, and domain-driven design expert with over 30 years of experience in the IT industry. He has built management software and ERP systems from the ground up for diverse industries and has guided teams in improving their skills and practices. Alessandro actively contributes to and maintains open source projects, is the co-author of Domain-Driven Refactoring, and co-founded DDD Open, an initiative promoting knowledge sharing and collaboration in the software development community.

I would like to thank my wife for her unrelenting support and patience, not only during the process of writing this book but throughout the life we've shared so far.

About the reviewer

Stephen Wille, with nearly two decades of experience in IT, has progressed from tier-one support to lead software engineer at a Fortune 500 company. His extensive experience provides him with a deep and broad knowledge base, enabling him to elevate engineering standards across diverse projects. A strong advocate for continuous learning, Stephen is passionate about sharing his expertise and empowering teams to achieve higher levels of performance.

My gratitude to those in the field who relentlessly push for excellence.

Table of Contents

Preface xvii

Part 1: A Brief Recap of AI and How an LLM Works 1

Chapter 1: Why AI Matters for Domain-Driven Design Today 3

The rising complexity of modern business systems 4
Architectural distribution • 4
Regulatory and compliance burdens • 5
Integration sprawl and heterogeneous systems • 6
Data volume and real-time analytics demands • 6
Sociotechnical complexity and team scaling • 7
Impact on domain modeling and decision-making • 8
Why traditional domain modeling fails to scale 10
Cognitive overload as the hidden constraint in DDD teams 10
Symptoms of cognitive overload in DDD teams • 12
Strategies to mitigate cognitive load in DDD teams • 13
AI as cognitive amplifier, not automation 14
How AI reshapes the architect's daily workflow 15
Summary 17

Chapter 2: How LLMs Actually Think and Why Precision Unlocks Understanding 19

LLMs as statistical pattern machines, not thinking minds 20
The core mechanism: next-token prediction • 20
Tokens, not words • 21
Prediction, not understanding • 23
A useful mental model for architects • 25
Tokenization, context windows, and predictive chains 26
How the model sees your domain language • 27
The context window: capacity and limits • 29

How the model connects distant tokens • 31
How coherent output emerges token by token • 33
Temperature: the variability dial • 34
Decoding controls in practice • 35
What this means for your domain work • 36
Emergent reasoning and its blind spots 37
Precision in language as the core of domain modeling accuracy 38
Summary 41
References 41

Chapter 3: The Architect's View: Reasoning with AI Systems 43

Architecting conversations, not just systems 44
The limit of the prompt paradigm • 44
Conversation as a system • 45
The AI reasoning loop • 45
Recognizing the DDD parallel • 47
Conversation as a bounded process • 47
Designing information flows for AI 48
The stateless model and the sovereign context window • 48
Information architecture: layering what the model sees • 49
The information flow pattern • 50
Information as probabilistic constraint • 51
The architect as information designer • 51
Creating shared vocabulary between you and the model 52
Introducing constraints as conceptual boundaries 56
Summary 58

Part 2: Apply AI to DDD with Simple Rules and Prompts 61

Chapter 4: From Domain Exploration to AI Collaboration 63

Technical requirements..... 64
Tools that are a bridge between humans and AI 64
EventStorming • 65
Domain Storytelling • 67
Comparing EventStorming and Domain Storytelling • 68

Turning visual chaos into structured context ... **69**
Translation, not transcription • 69
Extracting the structure already present • 70
From discovery artifacts to reusable context • 71
Identifying core events and refining through feedback ... **72**
Candidate events and core events • 73
What the model contributes • 74
Feedback as an architectural input • 75
Corrective feedback • 76
Additive feedback • 76
Structural feedback • 76
A practical refinement loop • 77
Step 1: classify the discovery material • 77
Step 2: inject domain and structural feedback • 79
Step 3: validate the revised set • 80
Refinement as knowledge crunching • 81
From iteration to semantic convergence • 81
What AI changes, and what it does not • 82
The role of language in domain clarity ... **83**
Where instability hides after a workshop • 83
The danger of false coherence • 84
A practical BrewUp vocabulary • 85
Testing and reinforcing the vocabulary • 87
From exploration to deliberate interaction • 88
Summary ... **89**
References ... **90**

Chapter 5: Defining Rules: Teaching the Model to Think in DDD **91**

Technical requirements ... **92**
Why rules matter more than prompts ... **92**
Where rules come from • 94
The three layers of a DDD rulebook ... **95**
Agents, skills, and MCP servers: three containers for DDD guardrails ... **97**
Agents • 98
Skills • 99

MCP servers • 101
Iterating toward a consistent domain vocabulary 103
Summary 106
References 107

Chapter 6: Iterative Refinement: Building the Bounded Context 109

Guiding the model through feedback loops 110
Introducing CQRS and role separation in AI reasoning 117
Refining events, commands, and aggregates 120
Achieving contextual coherence through iteration 124
Coherence emerges from repeated comparison • 125
Stable ownership and stable roles • 126
Open questions are part of coherence • 127
Coherence is reinforced by a context snapshot • 128
Testing coherence with a final review prompt • 130
When the bounded context is coherent enough • 132
Summary 132
Subscribe to Deep Engineering 134

Chapter 7: Prompt as Domain Carrier 135

From asking to encoding: the knowledge shift 136
What a domain carrier is made of • 137
Building a Stock Management domain carrier • 139
Putting the carrier to work • 141
Reusing the carrier across workflows 144
The derive–review–update workflow • 145
Reading AI outputs as reflections of understanding 151
Building shared semantic alignment between human and AI 153
Summary 155
References 156

Part 3: Building Agents to Streamline the Interaction 157

Chapter 8: From Prompts to Agents: Structuring AI Collaboration 159

Technical requirements 161

From a plausible prompt to a fragile implementation 162
Evidence that no authority produced • 163
From findings to durable rules • 165
Turning domain knowledge into a specification 166
What the specification does to the feature request • 167
Clarification and what it makes durable • 169
Reviewing the specification • 170
From specification to plan and tasks 172
What the plan decides • 173
Traceability, and what it does not prove • 174
When a checked task is not a completed task • 175
Adding cognitive boundaries with guard agents 177
What a cognitive boundary constrains • 177
Two kinds of rule: ownership and structure • 178
Where the guards sit in the life cycle • 179
A guard in action: the idempotency requirement • 181
What a guard must not do • 182
Controlled delegation • 183
Running the same feature request through a role-based workflow 183
The trace the repository actually preserves • 184
Where the domain knowledge travels • 184
Summary 185
References 187

Chapter 9: From Guardrails to Specialists: Designing Agents for DDD 189

From defensive guards to productive specialists 190
A specialist transforms; an authority decides • 190
Contracts and handoffs as cognitive boundaries • 192
The BrewUp experiment and its limits • 195
EventStormer: discovering facts without deciding the domain 200
From evidence to candidate facts • 200
When discovery becomes invention • 201
What the run showed, and what the gate accepted • 202
Storyteller: making behavior concrete without completing the story 203

From accepted facts to concrete scenarios • 203
Where the evidence stops, the story must stop • 204
What the run showed, and what the gate accepted • 205
Command and Event Writer: precision without stolen authority 206
From scenarios to behavioral candidates • 206
The pull of the convenient command • 207
What the run showed, and what the gate accepted • 208
Context Mapper: revealing boundaries without designing the implementation 209
From accepted behavior to responsibility boundaries • 209
Run 9.5: mapping responsibilities • 210
Naming boundaries for what they decide • 211
A tidy map can still be wrong • 212
Human gate: accepting boundaries, not implementation structure • 212
Evaluating the complete chain 213
One vocabulary, four transformations • 213
What must survive every handoff • 214
What the chain preserved • 215
The price of specialization • 216
Four agents create four review gates • 217
What the human still had to decide • 217
What the repository makes reproducible 218
From manual gates to orchestration 219
Summary 219

Chapter 10: The Orchestrator: Building the Coordinating Agent 221

Technical requirements 222
The orchestrator as facilitator, not controller 222
Creating the orchestration workspace • 223
Register two bounded specialists • 224
Record the first workflow state • 224
Implementing the eligibility check • 224
Running the transition before the gate exists • 227
Record the human gate and resume • 227
Generalize the boundary from the working example • 228
Designing multi-agent conversations 230

From free-form chat to an explicit protocol • 230
Defining the artifact envelope • 232
Defining the messages and routes • 233
Encode the conversation state • 234
Allowing only legal transitions • 236
Accepting a candidate and advancing • 239
Referring a candidate without advancing • 239
Stopping when authority is missing • 240
Testing the protocol invariants • 240
The BrewUp orchestration loop 241
Building the loop around the protocol • 241
Staging the accepted artifacts • 243
Running the first specialist and pausing for review • 244
Advancing through the four BrewUp stages • 246
Correcting one stage without restarting the chain • 247
Persisting a replayable run trace • 248
Completing only after the final human gate • 249
Testing the orchestration loop end-to-end • 250
Summary 251

Chapter 11: Governed Autonomy: Balancing Autonomy and Alignment Across Agents 253

Technical requirements 254
From coordination to decision rights 254
Turning autonomy into policy • 255
Defining the shared alignment inputs • 256
Separating mechanical violations from semantic divergence 257
Enforce mechanical alignment rules • 258
Surfacing semantic divergence without resolving it • 260
Correcting drift without expanding autonomy 261
Running a controlled drift and correction scenario • 261
Bound correction attempts, time, and tool access • 264
Testing autonomy without surrendering alignment • 266
Operating the multi-agent system as a governed workflow 268
Summary 269

Chapter 12: Unlock Your Exclusive Benefits **271**

Unlock this Book's Free Benefits in 3 Easy Steps **272**

Other Books You May Enjoy **276**

Subscribe to Deep Engineering **278**

Index **279**

Preface

Modern software systems are more complex than ever. Distributed architectures, regulatory pressure, integration sprawl, and sociotechnical fragmentation place enormous cognitive demands on developers, architects, and technical leaders. Domain-Driven Design remains one of the most effective ways to master that complexity, but it depends on sustained reasoning, precise language, and a shared understanding of the business domain.

Large language models introduce a new kind of support for this work. Used only as code generators, they can produce fluent and plausible artifacts without preserving the decisions that make a domain model coherent. This book treats AI as a structured reasoning partner that can amplify human thinking, reduce cognitive overload, and help teams expose ambiguity while people retain decision authority.

The book begins by building an operational mental model of how modern AI and LLMs work. You'll examine tokenization, statistical prediction, context windows, emergent reasoning, and the limits behind apparently intelligent behavior. This foundation explains why precision in language matters and why architectural constraints are essential when an AI system participates in domain reasoning.

From there, you'll apply AI to domain discovery and modeling. EventStorming becomes a bridge between human exploration and structured AI collaboration, while reusable rules and prompt patterns establish vocabulary, output structure, and semantic boundaries. Through iterative feedback and the BrewUp case study, you'll refine events, commands, aggregates, policies, and bounded contexts without hiding unresolved domain questions.

Prompts then evolve from one-off requests into durable carriers of domain knowledge. Examples, constraints, and shared language become reusable assets that preserve the reasoning behind a model. Building on those assets, you'll move from single-prompt interactions to specialized agents for domain discovery, storytelling, command and event writing, and context mapping.

The final part brings these specialists together through an orchestrator with typed messages, legal state transitions, human gates, persistence, replay, and enforceable autonomy policies. Provenance, uncertainty, drift, and human acceptance remain visible throughout the workflow.

By the end of the book, you'll be able to use AI to analyze domains, refine domain events and commands, design bounded contexts, and coordinate multi-agent workflows in a disciplined

and repeatable way. The goal is not autonomous architecture, but better reasoning: domain decisions that remain explicit, inspectable, and under human control.

Who this book is for

This book is for software developers, architects, and technical leaders working on complex business systems who want to use AI to strengthen their Domain-Driven Design practice. You should be comfortable with core software architecture concepts and familiar with DDD fundamentals such as bounded contexts, aggregates, and domain events. You don't need advanced DDD or AI expertise, but practical experience building or maintaining real-world systems will help you connect the modeling techniques to your own work.

What this book covers

Chapter 1, *Why AI Matters for Domain-Driven Design Today*, explains why AI has become a valuable reasoning companion for teams working on complex business systems. It examines growing domain complexity, cognitive overload, and the limits of traditional modeling practices, framing AI as a cognitive amplifier rather than an autonomous system.

Chapter 2, *How LLMs Actually Think and Why Precision Unlocks Understanding*, builds a clear mental model of tokenization, statistical prediction, context windows, apparent reasoning, and their blind spots. This chapter shows how precise language and structured instructions compensate for model limitations and improve domain-modeling accuracy.

Chapter 3, *The Architect's View: Reasoning with AI Systems*, reframes AI as a reasoning collaborator operating within explicit conceptual boundaries. This chapter applies architectural thinking to conversations, information flows, shared vocabulary, constraints, expectations, and responsibilities.

Chapter 4, *From Domain Exploration to AI Collaboration*, uses EventStorming as a bridge between human insight and AI reasoning. This chapter shows how to translate messy exploration into structured prompts, identify and refine core domain events, and build linguistic clarity between participants and the model.

Chapter 5, *Defining Rules: Teaching the Model to Think in DDD*, demonstrates how reusable rules and constraints guide an LLM toward DDD-consistent reasoning. You'll build a prompt rulebook that defines vocabulary, structure, expected outputs, and the criteria used to evaluate semantic and domain accuracy.

Chapter 6, *Iterative Refinement: Building the Bounded Context*, applies feedback loops to events, commands, aggregates, and policies. It introduces CQRS as a way to shape role separation in the model's reasoning and progressively builds a coherent bounded context for the BrewUp Order domain.

Chapter 7, *Prompt as Domain Carrier*, shows how prompts can encode and preserve domain knowledge instead of acting as disposable queries. Examples, rules, and structure become living documentation that supports a shared mental model and makes AI outputs easier to evaluate.

Chapter 8, *From Prompts to Agents: Structuring AI Collaboration*, evolves the workflow from single prompts to a team of specialized agents with clear cognitive boundaries. It introduces role definition, responsibilities, information flow, and collaboration patterns modeled on human teams.

Chapter 9, *From Guardrails to Specialists: Designing Agents for DDD*, guides the creation of agents dedicated to EventStorming, storytelling, command and event writing, and context mapping. This chapter shows how domain rules and shared language keep coordinated specialists focused and consistent.

Chapter 10, *The Orchestrator: Building the Coordinating Agent*, introduces the orchestrator that turns specialist handoffs into an executable control plane. Eligibility checks, typed messages, legal state transitions, human gates, persistence, and replay coordinate the BrewUp loop while preserving provenance, open questions, and decision authority.

Chapter 11, *Governed Autonomy: Balancing Autonomy and Alignment Across Agents*, makes specialist freedom explicit and governable. It distinguishes deterministic boundary violations from semantic divergence, bounds correction without expanding authority, and keeps evidence, uncertainty, and human acceptance visible across the multi-agent workflow.

To get the most out of this book

- You should be comfortable with core software architecture concepts and familiar with DDD fundamentals, including bounded contexts, aggregates, and domain events. You don't need prior AI expertise.
- To follow the hands-on material, you'll need Git, the .NET 10 SDK, access to an LLM-based assistant that supports structured prompts and project-level instructions, and the companion repository containing the BrewUp code, specifications, prepared artifacts, and agent definitions.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781807421052`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and social media handles. For example: "Similarly, defining `StockReserved` as belonging to the warehouse responsibility preserves a boundary that a simple causal list might otherwise hide."

A block of code is set as follows:

```
WriteGate(new GateRecord(
    "eventstormer-review",
    "accepted",
    new EffectiveArtifact(
        "accepted-facts",
        "docs/ch09/runs/9.2-eventstormer/accepted-facts.md"),
    ["ES-17"]));
```

Any command-line input or output is written as follows:

```
/speckit.tasks
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Ambiguous language does not necessarily produce visibly confused output. More often, it produces **false coherence**."

> **Note**
>
> Warnings or important notes appear like this.

> **Tip**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Share your thoughts

Once you've read *AI-Augmented Domain-Driven Design*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



`https://packt.link/r/1807421058`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Join us on Discord!

Read this book alongside other users, developers, experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the authors via Ask Me Anything sessions, and much more. Scan the QR or visit the link to join the community.



`https://packt.link/deep-engineering`

Part 1

A Brief Recap of AI and How an LLM Works

In this first part of the book, you'll learn how modern AI and LLMs work, build accurate mental models of their capabilities and limits, and reason confidently about how they support architectural work. The chapters move from the relevance of AI in DDD to the mechanics of LLMs and the architect's role in structuring AI-assisted reasoning.

This part of the book includes the following chapters:

- *Chapter 1, Why AI Matters for Domain-Driven Design Today*
- *Chapter 2, How LLMs Actually Think and Why Precision Unlocks Understanding*
- *Chapter 3, The Architect's View: Reasoning with AI Systems*

1

Why AI Matters for Domain-Driven Design Today

Software architects and domain modelers face a different kind of challenge today. The difficulty is no longer limited to technical complexity alone. What weighs just as heavily is the effort required to reason clearly about systems as they grow in scope, interdependence, and constraints. In that environment, even a disciplined approach such as **domain-driven design (DDD)** can come under pressure. Domain knowledge starts to scatter, model coherence weakens, and design clarity becomes harder to sustain. The issue is not a flaw in DDD itself. Applying it consistently at scale now demands more cognitive effort than many teams can realistically sustain.

This chapter introduces the core motivation for the rest of the book: why AI matters to teams who care about modeling domains well.

AI helps most when it strengthens reasoning; acceleration alone changes little.

Used with care, it can support human understanding, ease cognitive pressure, and help teams preserve clarity while systems continue to evolve. In this chapter, you will learn:

- How system complexity is overwhelming traditional modeling and communication practices
- Why the real bottleneck is cognitive overload rather than any flaw in DDD
- What it means to use AI as a reasoning partner instead of a code generator
- How the architect's daily role changes when supported by AI, without losing control

Let's explore how the landscape has changed and why your modeling practices must evolve with it.

> **Note**
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

The rising complexity of modern business systems

The difficulty of modeling modern software systems does not come from a single source. It emerges from forces that interact and reinforce one another: architectural distribution, regulatory constraints, integration sprawl, growing data demands, and organizational scale.

Individually, each of these forces might seem manageable. When they compound, teams begin to lose their footing. Domain knowledge becomes harder to maintain, decisions require more context, and keeping a coherent model increasingly depends on how much complexity people can hold and reason about at once.

To understand why AI can help in this context, it is useful to examine the main drivers behind this growing cognitive load and how they build on one another.

Architectural distribution

One driver of complexity is the widespread shift toward distributed architecture. Microservices and event-driven systems can improve autonomy, scalability, and resilience, but they also move complexity elsewhere. A service might become easier to understand in isolation while the behavior of the system as a whole becomes harder to follow.

Network communication, partial failures, eventual consistency, and asynchronous workflows all require reasoning across boundaries. A business operation that once happened within a single process might now unfold through a sequence of interactions whose effects are distributed across multiple components.

Consider the BrewUp application, a fictional system that we will use as a running case study throughout the book. A seemingly simple action such as placing an order can trigger warehouse checks and updates, payment processing, shipping, and invoicing. Each individual component might have a clear responsibility, yet understanding the business operation requires connecting decisions and state changes that happen in different places and at different times.

The same effect appears in infrastructure and data architectures. Cloud services, distributed data ownership, and domain-specific pipelines can give teams greater autonomy, but they also introduce more boundaries that must be coordinated by teams and understood.

Architectural complexity has therefore not disappeared. Much of it has moved from inside individual applications to the relationships between distributed components. For domain-driven architects, keeping each bounded context coherent is no longer enough. The business model also has to stay understandable while those contexts interact across unreliable and asynchronous boundaries.

Regulatory and compliance burdens

Another driver of complexity is the expanding regulatory and compliance landscape. Modern enterprises face increasing regulatory scrutiny. Privacy laws, financial regulations (for example, anti-money-laundering checks), healthcare rules, and industry-specific compliance requirements continually evolve. The pace of regulatory change has accelerated, and organizations face an increasing cost of noncompliance if they fail to keep up. Each new or changed regulation introduces new rules, auditing needs, and constraints that software must accommodate, often across every layer from data storage to business logic.

The complexity here is twofold. First, systems must implement numerous rules and exception paths to enforce compliance (for example, handling data consent or segregating customer data by region to respect data residency laws). These rules often cut across domain boundaries, introducing tangled concerns that complicate the domain logic. Second, compliance often requires significant traceability and monitoring. These systems need to produce detailed logs, reports, and evidence of controls, which adds infrastructural complexity. In finance, for instance, an online trading platform might need to account for fraud detection, **know your customer** (**KYC**) checks, and audit requirements alongside its core business rules. As you can imagine, every additional concern increases cognitive load on architects and developers who must ensure the system's design remains both correct (satisfying business needs) and compliant with external mandates.

The need for explicitness is why regulatory constraints must be stated explicitly, and their impact on the domain understood. Some become business rules and first-class domain concepts; others remain external constraints or infrastructural concerns. The challenge is knowing which is which. Skip that step, and compliance logic ends up bolted onto the domain as an afterthought, scattered checks and special cases that no one fully trusts and everyone is afraid to touch.

Integration sprawl and heterogeneous systems

A third driver compounds the first two: integration sprawl. Modern business systems rarely live in isolation – they sit in a web of integrations with third-party services, legacy platforms, and partner APIs. The drive for digital transformation and omnichannel customer experiences means that a single business process might span numerous systems: CRM software, payment gateways, analytics services, and IoT devices. The result is integration sprawl: a proliferation of APIs, data formats, and interface protocols that the architecture must accommodate.

You will often find yourself managing a landscape of heterogeneous systems and API endpoints that is complex in itself. Different business units might build or adopt their own APIs and tools, resulting in overlapping or redundant capabilities, and it is not uncommon to end up running several different API management gateways or integration platforms at once. This integration complexity manifests in your day-to-day development too: you and your team must juggle multiple SDKs and data contracts. A change in one external service can have cascading impacts on your internal systems. Ensuring consistency and reliability across heterogeneous components is non-trivial. From a domain perspective, the bigger risk is semantic leakage. External systems rarely use concepts with exactly the same meaning as your domain does. A logistics company integrating routing services, mapping APIs, weather data, and internal fleet management might encounter several representations of concepts such as *delivery* or *route*. Those differences are not necessarily a problem; in DDD, different models can legitimately coexist. The problem begins when those meanings cross boundaries without explicit translation, allowing external assumptions to leak into the domain model.

Architecturally, patterns such as enterprise integration hubs, event brokers, and API gateways can help manage the mechanics of this sprawl. They do not, however, solve the semantic problem. In DDD terms, architects must decide where genuine model boundaries exist and where translation requires the architecture to protect them. Anti-corruption layers and adapters can prevent external models from leaking into the domain, while explicit contracts make those relationships visible. The architect must therefore reason about which meanings the receiving context may accept to cross each boundary, and not only about how the systems connect.

Data volume and real-time analytics demands

A fourth driver adds to the load. The volume, velocity, and variety of data that systems must handle have grown dramatically. In many industries, competitive advantage now comes from how well an organization leverages data (often in real time). This has led to architectures that embed large-scale data processing and machine learning components into the core business workflows. Analytical platforms and models that used to live on the periphery (for example, in a separate data warehouse or offline reports) are increasingly becoming first-class citizens of

software systems (`https://www.infoq.com/articles/architecture-trends-2024`). The implication is that systems are no longer designed only for transactional processing. They must also be capable of streaming events, updating analytics dashboards, and even feeding AI/ML models on the fly.

Handling massive data streams and real-time expectations adds substantial complexity. Systems must support the requirements for high throughput and low latency simultaneously. For example, an e-commerce site today might need to personalize recommendations and prices for a user in real time based on live data, making decisions dependent on continuously updated analytical inputs.

Moreover, once analytics and AI are embedded in user-facing functionality, they must be treated with the same rigor as any other production component. Architects need to reason about aspects such as resilience and scalability, but also about how these components can be observed and understood in production. Data pipelines must be monitored continuously, and models cannot be considered static artifacts. They require periodic retraining to avoid drift, and their life cycle must be managed with the same care as any other critical component. On top of that, data privacy introduces additional constraints that cut across the system. As a result, data-related quality attributes such as throughput and data freshness become first-class concerns, while the accuracy of analytics directly impacts business decisions. These aspects now sit alongside more traditional attributes, such as consistency or availability.

This pressure complicates domain modeling because analytical outputs can begin to influence business decisions directly. A concept such as Customer Segment or Anomaly Score becomes relevant to the domain once the business assigns meaning to it and uses it to make decisions. A model producing the value is not enough on its own. Architects must therefore reason about where those derived concepts belong, how reliable they need to be, and what happens when the underlying data or model changes. Data governance adds another layer to that reasoning, from deciding where domain data might reside to managing feedback loops when predictions influence subsequent business behavior.

Sociotechnical complexity and team scaling

A fifth driver is organizational, not technical. As systems grow, so do the teams and organizations that build them. Modern development practices recognize that software architecture is as much a social design as a technical one. Conway's Law states that "*[O]rganizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.*" (Conway, M. E., "How Do Committees Invent?", Datamation, April 1968: `https://www.melconway.com/Home/Committees_Paper.html`)

This is not just a loose correlation. Communication patterns actively shape system boundaries, responsibilities, and coupling. When teams align around clear domains, architectures tend to reflect coherent bounded contexts. When communication fragments or creates silos, those fractures emerge directly in the software as duplicated concepts, inconsistent models, and unclear ownership. In large organizations, this effect becomes more evident. Teams operate with partial knowledge and localized interpretations of the domain, and the system can evolve as a collection of models whose relationships and boundaries are no longer clearly understood.

A simple example can appear in the BrewUp domain. Order Management, Warehouse, and Shipping may legitimately model the same business process from different perspectives. The problem begins when those differences are accidental or remain implicit. If the teams rarely communicate, they might use the same term with different assumptions about ownership, lifecycle, or responsibility, creating friction when behavior crosses context boundaries.

Domain-Driven Design can be seen as a deliberate response to this dynamic. By introducing bounded contexts, Ubiquitous Language, and explicit relationships between models, DDD helps make both semantic and ownership boundaries visible. But those boundaries are not sustained by diagrams alone. As organizations scale, communication paths change, knowledge fragments, and team responsibilities evolve. The architecture must evolve with them. These sociotechnical forces directly affect domain modeling.

Decisions about model boundaries cannot be separated from questions of ownership, communication, and knowledge flow. A boundary that is conceptually sound but impossible for the organization to sustain will eventually erode. Architecture therefore has to account not only for how the business is structured, but also for how people responsible for understanding and evolving it collaborate.

Impact on domain modeling and decision-making

The five drivers traced in the previous sections were:

- *Architectural distribution*: a single business operation spans across services, so following it means tracing state changes across several boundaries.
- *Regulatory and compliance burdens*: rules that originate outside the business have to be modeled by teams inside it, and they change on someone else's schedule.
- *Integration sprawl and heterogeneous systems*: every external system brings its own vocabulary, and those meanings leak into the domain model.
- *Data volume and real-time analytics demands*: derived values start driving business decisions, so the model has to say where they belong and how far they can be trusted.
- *Sociotechnical complexity and team scaling*: how teams communicate decides where the model's boundaries actually fall, whatever the diagrams say.

Having traced these five drivers individually, it is worth asking what they add up to for the people who model domains day to day. The rising complexity has implications for how you model domains and make design decisions. In a high-complexity environment, DDD is more relevant than ever but also more difficult to apply. One immediate impact is that architects and modelers must constantly deal with change and evolution. In a complex adaptive system, nothing stays static: business processes evolve, teams restructure, regulations update, and new technologies disrupt old patterns.

As Barry O'Reilly notes, "*As soon as we model it, the model starts to drift from reality.*" **Model drift** is an inevitable phenomenon (`https://www.linkedin.com/posts/barry-o-reilly-b924657_model-drift-and-software-attractors-the-activity-7399023297276821504-h4Xm/`). Your software models continuously and often imperceptibly diverge from the living business context. This drift means that a domain model can never be *one and done*. You have to treat it as a living artifact, expecting to refactor and realign it regularly with the business. Decision-making in this context should therefore favor feedback, adaptability, and reversibility. Rather than optimizing for a supposedly final design, architects should preserve the ability to revise boundaries and decisions as understanding of the domain evolves.

From a domain modeling perspective, complex systems make explicit boundaries even more important. Ubiquitous Language does not need to remain uniform across an entire organization. Different bounded contexts might legitimately use different models and assign different meanings to similar terms. The challenge is to keep each model internally coherent and make the translations between them explicit. Context Maps, explicit contracts, Published Languages, and translation mechanisms such as anti-corruption layers help make relationships between models visible and deliberate.

When information crosses a bounded-context boundary, integration contracts and events should express what the receiving context may decide it needs to know without exposing the internal model of the producer.

Finally, continuous modeling creates another challenge: preserving what the team has learned. If assumptions, terminology, boundary decisions, and their rationale remain only in conversations or individual memory, each change forces the team to reconstruct part of the reasoning that came before. Discovering domain knowledge is therefore only half the work. Keeping it accessible as the system evolves is the other half. This is where the problem becomes less architectural and more cognitive.

Why traditional domain modeling fails to scale

Traditional domain modeling becomes difficult to sustain as complexity grows because it depends on teams maintaining an increasing amount of shared context. As modern business systems evolve, teams must simultaneously reason about domain rules, organizational structures, architectural constraints, regulatory requirements, and ongoing changes.

The symptoms are familiar: models multiply, documentation drifts away from both the code and day-to-day decision-making, and earlier decisions become harder to reconstruct. Over time, domain knowledge becomes implicit again, preserved in fragmented conversations, personal experiences, or buried in implementation details.

The limitation, therefore, is one of capacity rather than of principle: what runs short is your ability to sustain the shared understanding that domain modeling requires. DDD gives you powerful ways to partition complexity and make knowledge explicit, but those mechanisms still depend on people continuously maintaining language, context, and design rationale as the system evolves. When that effort exceeds what a team can reliably sustain, the modeling practice begins to degrade.

To understand why this happens, you need to look more closely at cognitive overload itself.

Cognitive overload as the hidden constraint in DDD teams

A team's **cognitive load** reflects how much domain, technical, and organizational context its members must actively process and coordinate to work effectively. When this load remains within reasonable limits, practices such as DDD thrive: the Ubiquitous Language is shared, models are discussed, and decisions are understood in context. However, as the load increases, these same practices begin to become difficult to sustain. DDD still depends on sustained human attention to keep models, language, and decisions connected as the domain evolves. Once **cognitive capacity** is exceeded, even well-constructed models cease to function as reasoning tools and instead become background noise.

In DDD, cognitive capacity matters because modeling is a collective reasoning activity. Teams must continuously connect business terminology, boundaries, dependencies, and past decisions while the domain itself keeps changing.

Team Topologies, by Matthew Skelton and Manuel Pais, treats cognitive load as a constraint on what a team can effectively own and understand. When responsibilities and interactions exceed that capacity, three effects become particularly relevant (`https://teamtopologies.com/news-blogs-newsletters/2024/7/23/newsletter-july-mastering-team-effectiveness-the-power-of-managing-cognitive-load`):

- *Increased context switching*: people constantly jump between tasks and lose focus
- *Lack of mastery*: no time to truly learn any one area, because they juggle too many
- *Diminished ownership*: teams do not feel accountable for any one domain, so the product's quality and evolution suffer

In DDD teams, these issues translate to shallow domain insight, fragmented mental models, and slow progress. The consequence is not merely lower productivity. A team might still deliver features while gradually losing the depth of understanding required to challenge the model, recognize inconsistencies, or notice when a boundary no longer reflects the business.

As cognitive load increases, the mechanisms that preserve shared understanding begin to erode. Language fragments, artifacts lose relevance, and the rationale behind architectural and domain decisions becomes progressively harder to recover. Teams can still appear to be *doing DDD*, while the shared understanding that gives those practices value quietly deteriorates.

You can consider a typical example: a team working on a core business domain that has grown over several years. New members struggle to understand why certain invariants exist, which rules are essential, and which are compromises inherited from the past. Senior developers become bottlenecks, repeatedly explaining the same concepts. Modeling sessions focus more on reconstructing past decisions than exploring new insights. The domain model technically exists, but instead of reducing cognitive effort, it has become another artifact the team must struggle to interpret.

This model is also why bounded contexts can be understood as boundaries of understandability: they do not merely partition software; they limit the set of concepts that must be understood together. Good boundaries reduce the amount of knowledge a team must coordinate at once; poor or eroded boundaries increase it.

Symptoms of cognitive overload in DDD teams

What does cognitive overload look like on the ground, and what does it eventually do to a DDD initiative? It manifests through specific, recognizable symptoms that impede modeling and architecture. Left unaddressed, those symptoms harden into lasting damage to the practice itself. The following are the most common signs:

- *Knowledge silos and heroics*: *Critical domain knowledge becomes concentrated in a few individuals*. Entire releases can experience delays because one person holds all the knowledge for a part of the domain.
- *Repeated rediscovery and lost context*: When cognitive load is high, teams often fail to *preserve or share* what they learn. This loss of shared context leads to a Groundhog Day scenario, where developers repeatedly rediscover domain insights or solutions that colleagues already know.
- *Fragmented Ubiquitous Language*: A foundational DDD practice is establishing a ubiquitous language within a bounded context: a shared vocabulary through which developers and domain experts express that model. Under cognitive pressure, that language begins to fragment. The same concept acquires multiple names, or the same term starts carrying different meanings. Teams skip or rush workshops, assumptions remain implicit, and the shared language gradually stops reflecting a shared model.
- *Domain model drift*: A telltale sign of overload is when the domain model stops evolving with the business. In a healthy DDD process, the model *changes* as the team learns, but that requires time to reflect, discuss, and refactor – time an overloaded team *rarely has*. Features continue to accumulate while the assumptions encoded in the model become progressively less representative of reality.
- *Brittle bounded context boundaries*: Cognitive overload can also erode boundaries that were once clear. Ownership becomes ambiguous; responsibilities leak across contexts, and teams take shortcuts that couple models that the teams intended to evolve independently. What breaks down is the ability to say where one model ends, where another begins, and how the teams may allow them to interact.
- *Friction in collaboration (cognitive drag)*: The clearest symptom is a general increase in friction – everything feels harder than it should. Decisions *take longer*, misunderstandings *multiply*, and *modeling discussions spend more time reconstructing context than exploring the domain*.

These symptoms are often interrelated and reinforcing: knowledge silos encourage fragmented language, fragmented language accelerates model drift, and **model drift** makes boundaries and ownership harder to understand. Cognitive overload therefore does not produce a single

isolated failure; it creates a feedback loop in which the mechanisms DDD relies on to manage complexity become progressively harder to sustain.

The next question is therefore practical: how can teams interrupt that loop?

Strategies to mitigate cognitive load in DDD teams

The symptoms above suggest a practical question: how can teams reduce cognitive pressure without weakening the depth of their domain understanding? No single technique answers it. What helps is a combination of strategies that limit what must be held in mind by teams in mind, externalize knowledge, and remove unnecessary complexity.

- *Limit the cognitive scope*: Bounded contexts help constrain the set of concepts that must be understood together, but organizational capacity also matters. Team responsibilities should be narrow enough for people to develop real mastery without constant context switching. This approach does not imply a rigid one-team-per-context rule, nor should contexts be split merely to fit an organizational chart. Rather, semantic boundaries, ownership, and team capacity should reinforce one another. The same principle applies to day-to-day work: limiting parallel work and unnecessary switching helps preserve attention for the domain currently being explored.
- *Externalize domain knowledge*: Cognitive load decreases when important knowledge no longer depends on individual memory. EventStorming, Context Maps, decision records, and other shared artifacts make assumptions, relationships, and unresolved questions visible to the team. Ubiquitous Language serves a similar purpose within a bounded context: terminology should be reflected consistently in conversations, models, and code rather than reconstructed from memory each time. A glossary can support that work, but it cannot replace the ongoing conversation through which meaning evolves.
- *Make reasoning collaborative*: Shared understanding is easier to sustain when modeling is treated as a collective activity rather than the responsibility of a few experts. Collaborative modeling, pairing, and ensemble techniques expose assumptions while decisions are being made and allow disagreements to surface before they become embedded in the model. The goal is narrower than making everyone know everything: critical domain knowledge and reasoning should simply not depend on a single person.
- *Remove accidental complexity*: Not all cognitive load comes from the domain. Tooling, deployment, infrastructure, inconsistent conventions, and unnecessary technological variation can consume attention without contributing to domain understanding. Standardization and automation are valuable when they remove this accidental complexity, but they should not erase meaningful differences between bounded contexts. Platform capabilities, templates, and sensible defaults can free teams from

repeatedly solving infrastructure problems, leaving more cognitive capacity for the parts of the system where domain-specific reasoning actually matters.

These strategies can reduce cognitive pressure substantially, but they share a limitation: they still depend on people continuously maintaining artifacts, revisiting assumptions, sharing context, and noticing when understanding begins to drift. Ironically, that discipline becomes hardest to sustain precisely when cognitive load is highest. This limitation is the point at which AI introduces a different form of leverage.

AI as cognitive amplifier, not automation

When teams discuss software design, they often frame AI primarily as an automation tool that writes code faster, generates documentation, or takes decisions away from humans. In the context of DDD, this framing is misleading. The central challenge in DDD is not speed of execution, but sustained reasoning about a complex and evolving domain. Automating outcomes does not address this challenge and can amplify confusion when the underlying understanding is weak.

AI becomes valuable in DDD when it serves as a cognitive amplifier, not as a substitute for human judgment. Its main contribution is to reduce the cognitive effort required to explore, question, and maintain domain knowledge, while the decisions themselves stay with the team. AI can help teams externalize what they already know but struggle to keep aligned: assumptions hidden in code, inconsistencies in language, missing invariants, or unclear boundaries between concepts. By helping teams make these elements explicit in shared, inspectable artifacts, AI can reduce how much context depends on individual memory.

From this perspective, AI can extend the role of traditional modeling tools without replacing them. Unlike static artifacts, AI can repeatedly analyze specifications, models, tests, and conversations when the material is made available as context. AI can support exploratory modeling by generating clarifying questions, proposing alternative interpretations, comparing representations, and surfacing potential tensions or ambiguities for the team to examine. Amplification, however, works in both directions: weak assumptions, incomplete context, or ambiguous language can be amplified just as easily as good reasoning. This amplification is why the judgment of the architect and the knowledge of the domain expert remain essential.

This distinction is the book's conceptual foundation: viewing AI primarily as automation encourages teams to delegate thinking prematurely, while viewing it as a cognitive amplifier encourages the opposite, more deliberate reasoning with less cognitive friction. In this role, AI reinforces the fundamental promise of DDD by helping humans sustain the understanding required to work with a complex domain. AI does not make the domain any simpler.

How AI reshapes the architect's daily workflow

If AI is understood as a cognitive amplifier, its impact on the architect's daily workflow is subtle but profound. The architect's role shifts toward making domain reasoning a more frequent part of everyday work rather than simply increasing the pace of production. Activities that were previously episodic, such as domain discovery, model validation, or architectural sense-making, become ongoing and lightweight. Instead of reserving *thinking time* for workshops or design phases, architects can engage in small, frequent cycles of exploration and clarification as part of everyday work.

In practice, this means that architects no longer rely solely on memory, static documents, or occasional meetings to maintain a coherent understanding of the domain. When relevant models, decisions, and domain artifacts are made available as context, AI can help interrogate them, surface assumptions, and compare current reasoning with what was established before. Architects can ask questions like the following and get immediate, contextual support: *What invariants does this change affect?, Where does this concept appear elsewhere in the system?,* and *Which assumptions are no longer valid?* This contextual support reduces the mental overhead of reconstructing relevant context before each decision and allows attention to focus on the decision itself.

AI also reshapes how architects work with teams and stakeholders. Instead of acting as the primary knowledge hub, answering the same questions repeatedly, or translating between business and technical perspectives, the architect can use AI to help create, interrogate, and revisit artifacts that preserve domain language, constraints, and design rationale. Domain language, constraints, and design rationale become easier to revisit, challenge, and refine collaboratively. This reduces dependency on individual memory and gives teams a stronger shared basis from which to reason.

AI also changes the economics of reflection. Tasks that were once considered *too expensive* when considering time and cognitive effort, such as revisiting old decisions, exploring alternative models, or validating consistency across bounded contexts, become less expensive to perform more frequently. The architect's workflow shifts from defending past decisions to repeatedly stress-testing and evolving them. In this way, AI does not replace architectural judgment; it creates conditions for better judgment, applied more often and with less cognitive friction.

These shifts change where the effort goes inside the modeling process rather than replacing the process itself. The DDD Starter Modeling Process maintained by the DDD Crew (`https://github.com/ddd-crew/ddd-starter-modelling-process`) runs through eight steps – Understand, Discover, Decompose, Strategize, Connect, Organize, Define, and Code – and

treats them as a loop rather than a straight line. AI earns its place at the points in that loop where the team has to hold the most context at once: gathering and cross-checking material before a Discover session, testing a candidate decomposition against what the team has already written down, and noticing while coding that the model and the business have drifted apart. It adds less value at points whose purpose is for people to surface and resolve disagreement together.

The following table, Table 1.1, sets the two situations side by side for the activities that recur most often in an architect's week.

Recurring activity	**A week without AI support**	**A week with AI as cognitive amplifier**
Reconstructing context before a decision	Re-reading tickets, diagrams, and chat history, then asking the two people who still remember why.	Asking what the recorded decisions and constraints already say, then checking the answer against the artifacts themselves.
Domain discovery	A workshop every few months, once enough open questions have piled up.	Short clarification cycles between workshops, so the workshop itself is reserved for genuine disagreement.
Model validation	Done when something breaks, or in review, by whoever happens to hold the context.	Run against the invariants the team has written down, every time the model changes.
Answering the same domain question again	The architect is the knowledge hub, and therefore the bottleneck.	The question is answered from shared artifacts; the architect is called in when those artifacts are wrong.
Revisiting an old decision	Too expensive to reopen, so it gets defended instead.	Cheap enough to stress-test, so it can be revised once the domain has moved.
Noticing model drift	Noticed late, usually through a bug or a failed release.	Asked as a routine question, so divergence surfaces while it is still small.

Table 1.1 – The architect's week, before and with AI support

The architect's day-to-day work therefore shifts from reconstructing context from memory toward working with explicit, inspectable knowledge that AI can help explore and challenge. Responsibility for interpreting the domain and making architectural decisions remains human, but more of the surrounding cognitive work can become visible, repeatable, and easier to revisit. The change marks the shift this chapter has been building toward: an architect who can see the domain more clearly.

Summary

In this chapter, you examined why DDD has become harder to sustain as modern business systems grow in complexity. Distributed architectures, regulatory constraints, integration sprawl, growing data demands, and sociotechnical complexity all increase the amount of context that architects and teams must understand and coordinate.

You then identified cognitive overload as a hidden constraint on domain modeling. The principles of DDD keep working as complexity grows. What becomes difficult is maintaining the shared understanding they require. Knowledge concentrates in individuals, language fragments, models drift, and boundaries become harder to sustain. Strategies such as limiting cognitive scope, externalizing domain knowledge, making reasoning collaborative, and removing accidental complexity can reduce that pressure, but they still depend on continuous human attention.

Finally, you reframed AI as a cognitive amplifier rather than primarily as an automation tool. When relevant domain knowledge, models, decisions, and constraints are made available as context, AI can help teams explore assumptions, revisit reasoning, and surface potential inconsistencies with less cognitive friction. AI does not remove the need for architectural judgment or domain expertise. Its value lies in helping humans sustain the understanding required to work with complexity.

This chapter's central shift is: not using AI to think instead of you, but using it to make your own reasoning more explicit, inspectable, and easier to revisit.

In the next chapter, you will examine how modern language models actually work, and why understanding their strengths and limitations is essential if you want to use them effectively as reasoning partners in the design process.

2

How LLMs Actually Think and Why Precision Unlocks Understanding

Large language models (**LLMs**) appear intelligent and capable of reasoning about complex domains. However, beneath that fluency lies a different mechanism. This chapter builds the operational mental model you need to collaborate with LLMs safely and effectively in **domain-driven design** (**DDD**).

This understanding will become the foundation for the prompting and modeling techniques used throughout the rest of the book. In practice, it will help you write better-structured prompts, validate the model's answers more safely, and keep your domain terms consistent across a conversation.

By the end of this chapter, you will be able to read an LLM's output for what it is: probabilistic generation that you can constrain and steer deliberately.

In this chapter, you will learn:

- Why LLMs are statistical pattern machines, not thinking minds, and what that means for architectural responsibility
- How tokenization, context windows, and attention mechanisms shape output, and why context placement matters
- Why fluent output can look like reasoning without being grounded
- Where emergent reasoning breaks down
- Why precision in language, especially through disciplined Ubiquitous Language, is the core lever for domain modeling accuracy

Together, these ideas lead to a practical destination: learning how to steer probabilistic generation with the same linguistic discipline that DDD applies to domain modeling. Before you design prompt patterns and structured workflows, however, you must first understand the engine you are steering.

LLMs as statistical pattern machines, not thinking minds

Before you can collaborate effectively with an LLM, you must first correct a misleading assumption:

LLMs do not think or understand domains in the human sense, nor do they hold intentions, beliefs, or goals. Their operation is more mechanical: they predict. This distinction is operational, not philosophical.

Treating the model as a human mind leads to concrete design mistakes. A colleague would remember a decision taken three turns ago, so you stop restating it; the model can use that decision only while it remains available in the active context or is supplied again by the surrounding application. A colleague who was unsure would say so, so you read fluent phrasing as confidence; the model sounds equally fluent whether its answer is grounded or invented. The workflow breaks at exactly the points where you removed a check, assuming something on the other side was keeping track. Adopting a precise mental model of how LLMs function tells you where those checks belong, and lets you use the model safely as a reasoning amplifier.

The core mechanism: next-token prediction

At its foundation, an LLM is trained to perform one task: *Given a sequence of tokens, predict the most probable next token.* The prediction task is all the model does.

If the input is "*In Domain-Driven Design, a bounded context defines*," the model calculates probabilities across its vocabulary and selects the token that best fits the statistical patterns it learned during training. The model does not *know* what a bounded context is, and it does not retrieve a definition from memory in the human sense. Instead, it generates the continuation that is statistically most coherent. This process repeats token after token, building sentences incrementally. *Figure 2.1* shows the two halves of this process. Panel (a) follows the input tokens through the network up to a probability distribution over the whole vocabulary; panel (b) shows how one token is selected from that distribution, appended to the sequence, and fed back in until the sequence ends.

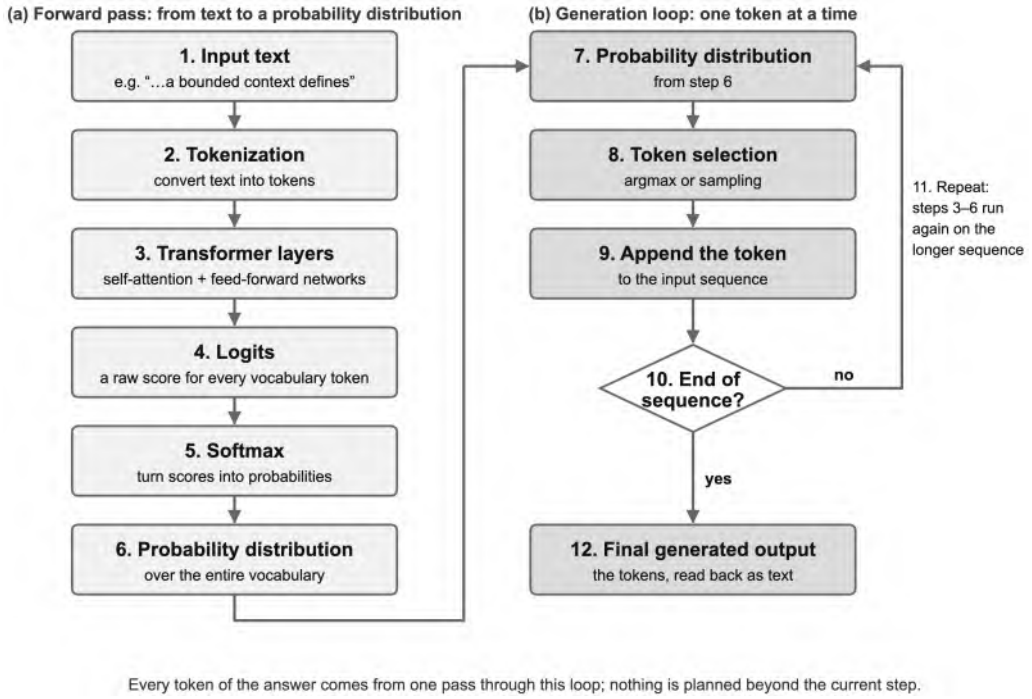


Figure 2.1 – Next-token prediction flow

The key implication is that an LLM never generates an answer as a whole: it produces a sequence, one probability decision at a time.

Concretely, as illustrated in *Figure 2.1*, the model starts with the input tokens and computes a probability distribution over all possible next tokens. It selects one token, appends it to the sequence, and then repeats the process using the updated context. Each step depends entirely on the tokens generated so far. What appears to you as a complete answer is, in reality, the result of many small, sequential probability decisions chained together.

From this simple mechanism, the rest of the chapter follows.

Tokens, not words

To understand how prediction works, and why wording matters in domain modeling, you must understand tokens. An LLM does not receive your Ubiquitous Language as a set of complete domain concepts; it receives a sequence of token fragments. When domain terms are vague, inconsistent, or represented in unfamiliar forms, the model must reconstruct their meaning statistically, increasing the risk of ambiguity and semantic drift.

A token is not necessarily a word. A token may be a full word, part of a word, a punctuation mark, a number fragment, or even whitespace. For example, *bounded context* becomes two tokens, *bounded* and *context*, matching the two words. The event name *OrderPlaced* also becomes two tokens, *Order* and *Placed*, but here the split is invisible in the text; a trailing period is a token of its own. The model operates over tokens, not semantic units.

For architects, this has important implications:

- Domain meaning is reconstructed statistically from fragments.
- The model infers structural relationships from recurring patterns.
- Ambiguous or inconsistent terminology propagates through the predictive chain.

Figure 2.2 illustrates this: even a clean sentence reaches the model as a sequence of tokens, not as a domain concept. The model must infer the concept from the sequence.

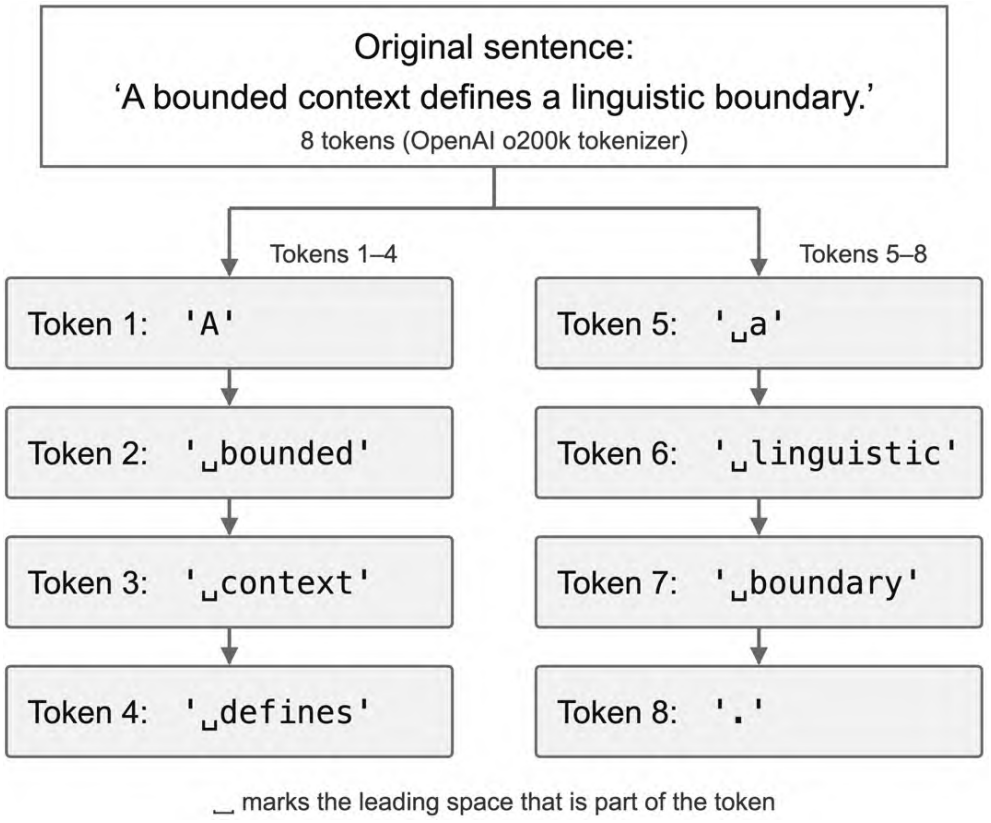


Figure 2.2 – Tokenization example

For domain modeling, this matters more than it first appears. When you provide vague language, ambiguous terms, or inconsistent vocabulary, you are injecting statistical ambiguity into the model's prediction chain. LLMs are sensitive to phrasing because phrasing shapes probability distributions.

Prediction, not understanding

Humans reason through internal models of the world. You maintain causal structures, mental simulations, and conceptual hierarchies, and you can step back from them: reflect, revise a belief, notice a contradiction.

LLMs do none of this. They approximate reasoning by reproducing patterns of reasoning found in training data. Trained on massive corpora of human text, they learn how explanations are formed, how arguments are built, how trade-offs are weighed, and how experts in a field typically write. When a model produces a coherent architectural explanation, *coherent* here means fluent and internally consistent rather than correct. The output reflects statistical patterns in how architectural explanations are structured; it does not demonstrate that the model understands architecture.

The illusion is convincing because the model mirrors the rhetorical shape of expert discourse: it gives definitions before examples, anticipates objections, and balances pros and cons. When it says "*You should separate these concerns into two bounded contexts*," the sentence feels intentional and evaluative. What has happened is that a sentence of that shape frequently follows a context of that shape in the training data, so the model completed the pattern.

The same mechanism that produces elegant explanations also produces elegant fabrications. If texts similar to your prompt often include regulatory constraints, the model might introduce them even when your domain has none. If architectural discussions frequently recommend separation of concerns, it might suggest additional bounded contexts whether or not your system needs them. The fluency of the output masks the absence of grounding: humans evaluate correctness against reality, while LLMs evaluate coherence against prior tokens.

Figure 2.3 contrasts these processes: human reasoning draws on knowledge, an internal mental model, evaluation, and feedback, while the LLM repeatedly predicts and appends tokens to produce coherent text.

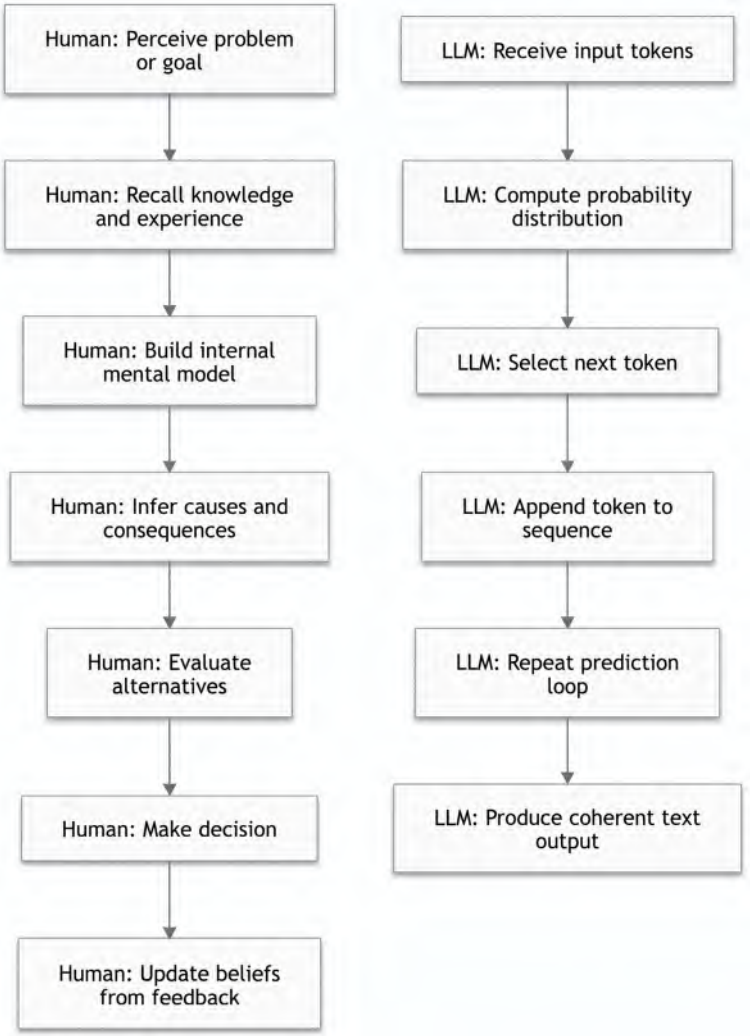


Figure 2.3 – Apparent reasoning versus statistical completion

Despite its mechanical foundation, LLM behavior can appear sophisticated because statistical pattern modeling at scale produces emergent structure. When trained on large corpora, models internalize patterns such as the following:

- Logical argument structures
- Mathematical formatting patterns
- Architectural explanation formats
- Cause–effect linguistic sequences

These patterns create behaviors that resemble reasoning: multi-step answers, self-corrections, and structured explanations. These behaviors are often called emergent reasoning, and you will look at its limits later in the chapter. For now, the point is that these are surface regularities. The model does not maintain an internal representation of truth; it maintains probability distributions over token sequences. For architects, the practical risk is mistaking a coherent explanation for a validated modeling decision. A recommendation to introduce an aggregate, split a bounded context, or define a new invariant may sound convincing while remaining ungrounded in the actual domain. The model can structure thought, but it cannot validate it against your domain's actual constraints, so architects must remain in control of what counts as true.

A useful mental model for architects

Think of an LLM as a probabilistic simulator of linguistic coherence: a high-dimensional statistical engine trained to predict tokens, not a thinker, a domain expert, or a decision-maker.

The model does not understand your domain, form intentions, or maintain beliefs about what is true. Nor does it preserve domain knowledge across interactions unless you reintroduce that knowledge through the current context or an external source of truth. What it can do is compress patterns found across vast amounts of language, reproduce familiar structures of explanation and reasoning, and respond strongly to explicit constraints.

This framing helps prevent two opposite mistakes: over-trusting the model's conclusions and underusing its pattern-synthesis capabilities. When used correctly, it is powerful at:

- Structuring messy information
- Generating candidate interpretations
- Exploring linguistic variations
- Surfacing implicit patterns

These outputs are valuable because they externalize possibilities for human review. They remain unvalidated conclusions.

When the model encounters uncertainty, it does not automatically abstain. Even if no high-confidence continuation exists, it still generates from the available probability distribution and continues. As a result, it might produce invented citations, fabricated APIs, nonexistent domain rules, and other inaccuracies expressed with unwarranted confidence.

The model generates for coherence, not truth. Semantic correctness must therefore be verified externally.

Figure 2.4 illustrates how the model continues selecting tokens even when the probability distribution is relatively flat.

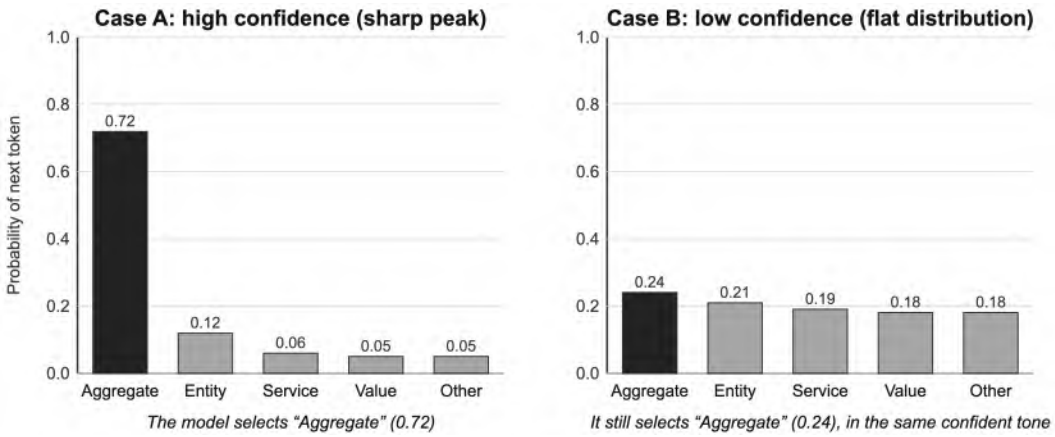


Figure 2.4 – Probability distribution under uncertainty

For DDD, this distinction is critical. If you ask, "*What are the invariants of this aggregate?*" without providing sufficient domain knowledge, the model might invent plausible invariants. There is no deception involved; aggregates typically have invariants, so the pattern the model is completing calls for some, and it supplies them.

For architects, that sets the boundary. An LLM output is a modeling hypothesis: something to inspect and challenge before it is accepted. The model can amplify a constraint you state explicitly, but responsibility for whether the domain is described truthfully stays with architects and domain experts.

This is the mental model to carry through the rest of the book: the model supplies coherent possibilities, and you supply the context and the validation. Within those boundaries, it is a powerful reasoning amplifier. With this mental model established, you can now examine how tokenization, context windows, and predictive chains shape model behavior, and why precision in language becomes the primary lever for high-quality domain modeling.

Tokenization, context windows, and predictive chains

In the previous section, we established that LLMs predict tokens, not meanings. Now, you need to go deeper. Understanding how tokenization works, what the model can see at any given moment, and how output emerges token by token will give you the operational knowledge to work with these models effectively.

The discussion is practical, not academic. Every technique you will apply in Part 2 of this book, from structured prompts to rulebooks to iterative refinement, is designed around the mechanics you are about to explore.

Note

Watch for this

Ask a model to *define the OrderPlaced event*, then ask it to *define the order placed event*, and compare the two answers. Readers typically interpret the first form as a class name, which pulls the output toward code; they interpret the second as a domain concept, which pulls it toward the language of your domain. The rest of this section explains why.

How the model sees your domain language

The model operates on tokens, not words, so the next question is how text is converted into those tokens.

Many modern LLMs use a tokenization algorithm called **byte pair encoding** (**BPE**), a technique originally developed by Philip Gage in 1994 for data compression (Gage, 1994). The principle is straightforward: during training, the algorithm analyzes massive amounts of text and identifies the most frequently occurring character pairs. It merges them into a single token. Then, it repeats the process, building progressively larger tokens from the most common combinations.

Think of the tokenizer as a compression algorithm optimized for general text. The tokenizer creates efficient single tokens for words it encounters frequently, for example, *the*, *and*, *function*, and *return*. This way, common English words and programming keywords become compact, well-represented tokens. Your DDD terms did not get that treatment. Written as two words, *bounded context* consists of common, well-represented tokens; written as the identifier *BoundedContext*, it is not, and the tokenizer has to cut it into pieces.

However, your domain vocabulary is a dialect it learned incidentally, not its native language. Consider what happens when the tokenizer encounters DDD terminology. Words such as *aggregate*, *context*, and *event* exist independently as common English words, so they get clean, single-token representations. However, when you combine them into DDD-specific compound terms, particularly in CamelCase form, things change.

Figure 2.5 illustrates this concretely.

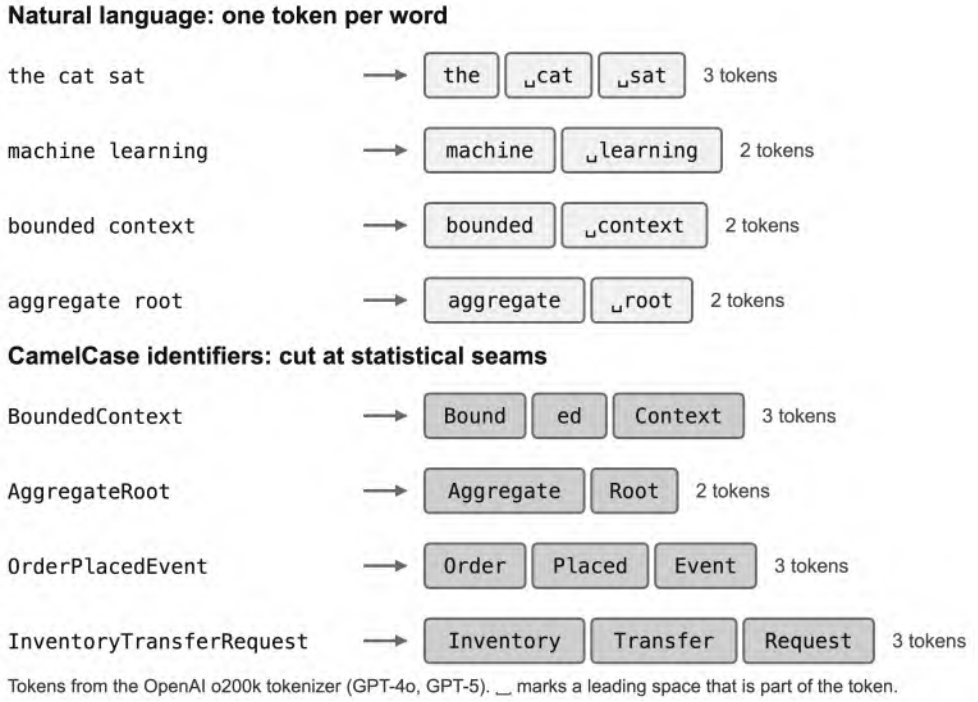


Figure 2.5 – BPE tokenization of DDD terms

When the model processes *BoundedContext* as a single CamelCase string, it does not see one unified concept. It sees three separate fragments: *Bound*, *ed*, *Context*. The capital letters are what break the word: the tokenizer has seen *bounded* countless times in running text, but rarely with a capital B glued to another word, so the familiar unit disappears. The model must reconstruct the semantic unity of *bounded context* as a DDD concept statistically, across those fragments, through the patterns the model learned during training.

This observation has a direct practical implication. When you write prompts for domain modeling, using natural language forms such as *bounded context* is typically more effective than using CamelCase forms such as *BoundedContext*. The reason is purely statistical: the model has encountered *bounded context* as a two-word phrase throughout DDD literature, blog posts, conference talks, and textbooks. The statistical associations between these tokens and the surrounding DDD vocabulary are strong and well-established. The CamelCase variant, by contrast, appears predominantly in source code, where the surrounding context is syntactic rather than semantic.

Vocabulary size varies by model: OpenAI's cl100k encoding, used by GPT-4, has about 100,000 tokens, and its o200k encoding, used by GPT-4o and GPT-5, has about 200,000 (OpenAI, 2024). Anthropic has not published the vocabulary of Claude's tokenizer. Whatever the exact size, the entire specialized vocabulary of DDD represents an infinitesimal fraction of that space. The model *knows* DDD not because it has dedicated tokens for domain concepts, but because it has seen recurring patterns in texts that discuss those concepts. The strength of those patterns depends directly on how frequently and consistently the model encountered them during training.

This analysis explains why, when you build structured prompts in Part 2, you will consistently use natural language rather than code-style naming for domain concepts. You now understand the mechanical reason behind that choice.

Note

Try it yourself

Visit the OpenAI Tokenizer at `https://platform.openai.com/tokenizer` and try entering your own domain terms. Compare how the tokenizer handles *bounded context* (two words) versus *BoundedContext* (CamelCase). Then, try your project-specific terms: your aggregate names, your event names, your service names. The fragmentation patterns will reveal how the model actually perceives your domain vocabulary.

The context window: capacity and limits

The context window is the maximum number of tokens the model can process at once. It includes everything available during an interaction: the system prompt, the conversation history, any additional instructions or documents, and the output generated so far. Every token the model needs to produce a response must fit within this window. Information outside it is unavailable to the model.

At the time of writing (September 2026), context windows range from 200,000 tokens to over one million. The OpenAI model GPT-5 offers 400,000 tokens; Anthropic's Claude Opus 5 and Sonnet 5 offer one million, and the smaller Haiku 4.5 offers 200,000; Google's Gemini models offer one million or more. These figures are a snapshot. Model lineups change every few months, so check the vendor documentation listed in the References for current values; the behavior described in the rest of this section holds whatever the number turns out to be. To put the scale in perspective, 200,000 tokens correspond to roughly 150,000 words, which is approximately 300 pages of text. This amount is more than most novels contain.

However, bigger is not necessarily better. Researchers from Stanford, UC Berkeley, and Samaya AI demonstrated a phenomenon now known as the Lost in the Middle effect (Liu et al., 2024).

They found that LLMs consistently underperform when relevant information appears in the middle of long contexts. Performance follows a U-shaped curve: information at the beginning and end of the context window is used with high fidelity, while information in the central region is often missed. In one of their multi-document question-answering settings, GPT-3.5-Turbo's accuracy dropped by more than 20 percentage points when the relevant document was moved from the beginning of the context to the middle; in some settings, performance was even lower than when no documents were provided. *Figure 2.6* visualizes this.

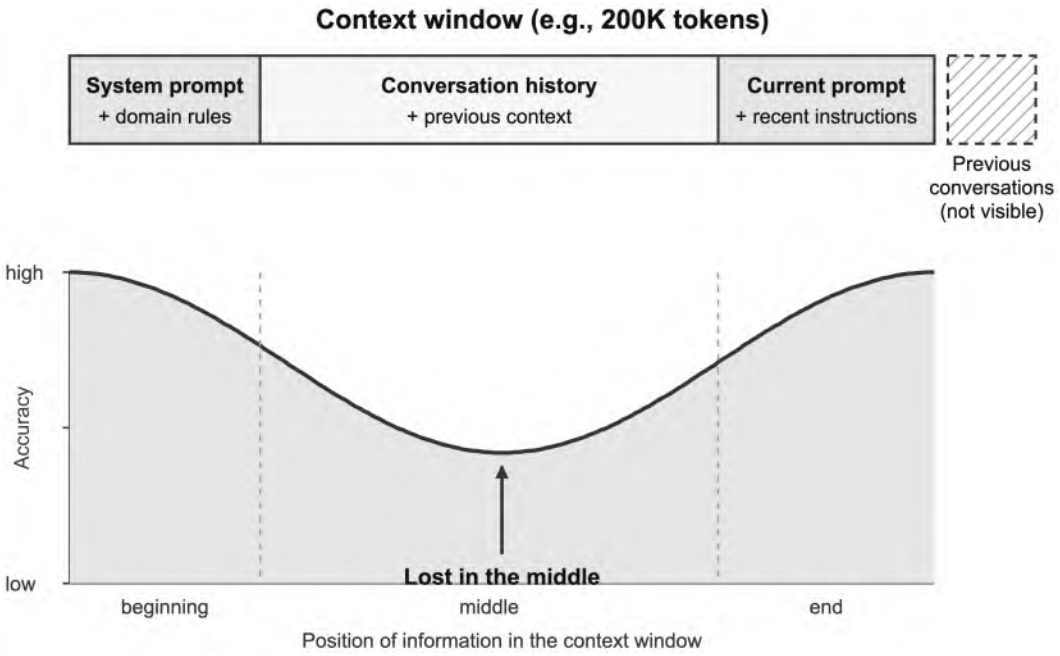


Figure 2.6 – The context window as working memory

The implications are practical and immediate. If you are working on a complex bounded context with the model and you place your domain rules somewhere in the middle of a long prompt, buried between conversation history and background information, the model might effectively ignore those rules. Their presence in the context does not guarantee that the model will use them correctly.

There is another critical point: the context available to the model depends on what the application supplies for each response. Earlier messages can remain part of that context, so a definition of *Order* established earlier in the same conversation might still be available. Starting a new conversation does not automatically carry those decisions across, although applications can retrieve saved information and insert it into the context. Similarly, truncation or summarization can remove or alter details from earlier exchanges. For domain modeling,

the practical requirement is to ensure that the current context contains the relevant definitions and constraints, rather than assume that every previous decision is still available.

This requirement explains why, in Part 2, you will maintain domain vocabulary, constraints, and rules in explicit rulebooks and ensure that the relevant content is available for each modeling task. The workflow can reuse information already present in the conversation and retrieve or restate it when it is missing or outdated. Keeping those rulebooks up to date gives successive interactions a consistent reference for domain decisions.

A common mistake is to assume that a larger context window automatically means better results. In practice, a focused 2,000-token prompt with clear constraints and well-positioned domain rules will often outperform a 50,000-token prompt where important information is scattered across a sprawling conversation history. Effective context is usually far smaller than maximum context. Later in the book, this principle will guide how you design focused prompts, externalize stable domain constraints in rulebooks, and provide agents with only the context required for each step of a workflow.

Note

Try it yourself

Take one domain rule from your project, for example, an order cannot be shipped before payment is confirmed. Paste it into a long prompt in three separate conversations: at the very top, buried in the middle of a few pages of background material, and at the very end. Then, ask the model to list the rules that apply to shipping an order. Compare which version surfaces the rule reliably, and how faithfully it is worded when it does.

How the model connects distant tokens

The model sees tokens in a finite window, but it also needs to connect relationships across that window.

Before the Transformer was introduced in 2017, many neural language models relied on recurrent architectures. These processed tokens sequentially, carrying information from earlier tokens forward in an internal state. They could capture relationships between distant words, but preserving the relevant information across many steps could be difficult. This limitation matters in domain modeling, where a concept and its rules may appear far apart in the text.

Transformer architecture (Vaswani et al., 2017) changed this fundamentally. Its **self-attention** mechanism lets the model *look at* the tokens already in the context, including distant tokens. When generating the next token, the model decides how much weight to give each one.

The mechanism works through three roles that every token plays simultaneously. Each token generates a *Query* (What are you looking for?), a *Key* (What do you represent?), and a *Value* (What information do you carry?). The model computes how well each token's Query matches its own *Key* and the Keys of the tokens before it, and uses those scores to determine how much of each token's Value to incorporate.

You do not need to understand the underlying mathematics. What matters is the operational consequence: the model can connect semantically related tokens regardless of how far apart they are in the text. For AI-assisted domain modeling, this determines whether the model can connect a concept to its rules, events, and boundaries when those elements appear in different parts of a prompt, rulebook, or retrieved context.

Figure 2.7 shows what this looks like for a DDD-specific sentence.

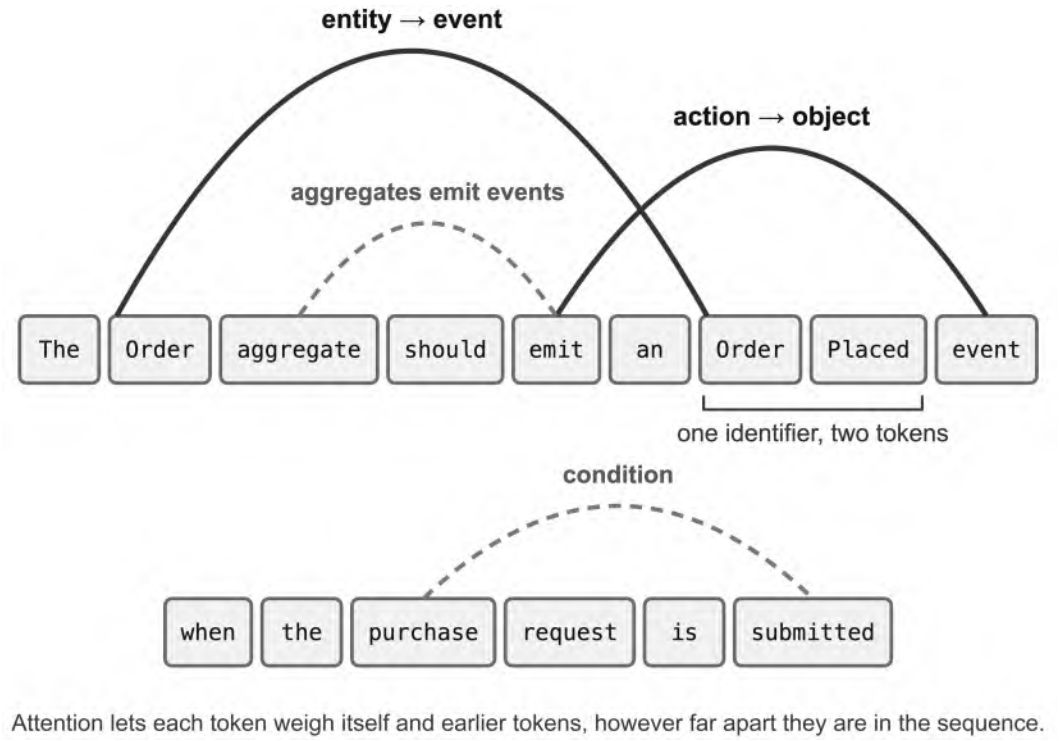


Figure 2.7 – Attention connecting distant tokens

When the model processes "*The Order aggregate should emit an OrderPlaced event when the purchase request is submitted*," attention allows it to do the following:

- Connect *Order* to *OrderPlaced* across four intervening tokens, recognizing the entity-to-event relationship
- Connect *aggregate* to *emit*, recognizing the DDD pattern in which aggregates produce events
- Connect *emit* to *event*, linking the action to its object
- Connect *purchase* to *submitted*, identifying the causal condition

Attention lets the model connect distant tokens directly, helping it track different types of relationships across the context window.

The model actually uses many attention *heads* working in parallel, each specialized in capturing different types of relationships: some focus on syntactic structure, others on semantic meaning, others on positional proximity. This parallel processing helps the model identify relationships within complex, multi-layered domain statements.

There is a cost. For the attention mechanism described here, doubling the number of tokens means roughly four times as many comparisons between them. The model's total computation and response time also depend on its architecture and implementation. This relationship is the fundamental engineering reason why context windows have limits, and why models with long contexts can be noticeably slower. It is also why filling a large context window with unnecessary information is not ineffective but actively wasteful: you are spending computational resources on tokens that dilute rather than strengthen the model's focus. With the mechanics of reading in place, we can turn to how the model writes.

How coherent output emerges token by token

You have established that the model sees tokens, processes them through attention, and operates within a finite context window. Now, we need to understand how it generates output.

The model produces text one token at a time. It examines the entire context, including all tokens generated so far, calculates a probability distribution over its entire vocabulary, selects one token, appends it to the context, and repeats. This sequential process creates what you call a **predictive chain**: each token selection shapes the probability landscape for every subsequent token.

This choice has important implications. Consider a simple prompt: *Define the Order aggregate in BrewUp*. On one path, the model might generate this: "*An Order represents a customer's purchase request...*" The token *purchase* anchors the subsequent generation firmly in e-commerce vocabulary. The statistical patterns associated with purchase, customer, and e-commerce

contexts will influence every token that follows. The chain stays coherent and domain-appropriate.

On another path, possibly because of a slight statistical fluctuation, the model might generate this: "*An Order is a hierarchical directive…*" The token *hierarchical* pulls the generation toward organizational or military vocabulary. Suddenly, *Order* is no longer a DDD aggregate but a command structure. Subsequent tokens can reinforce this deviation because the model conditions on what it has already generated.

This behavior is the cascade effect. An early token choice that seems individually reasonable can systematically steer the entire generation away from your intended domain. Once the chain diverges, the model might fail to notice the error. Asking it to check the response against the prompt can help, but does not guarantee a correction. For architects, the practical concern is that an early interpretation of a domain concept can shape every subsequent modeling suggestion, from aggregate boundaries and event names to invariants and business rules.

The next two subsections examine two decoding controls, temperature and nucleus sampling, that shape token selection and influence how easily a predictive chain can drift.

Temperature: the variability dial

A parameter called **temperature** controls the degree of randomness in token selection. Think of it as a variability dial. Temperature is continuous: most APIs accept any value from 0 to 2, and 1.0 leaves the model's original probability distribution unchanged. The bands below are illustrative points along that range, not discrete modes.

At low temperature (below about 0.3), the model is more likely to choose the tokens it already favors in the current context. Low temperature makes outputs more consistent, which can be useful for DDD terminology. Low temperature does not guarantee that the terms or domain rules are correct.

At mid-range temperature (about 0.3 to 1.0, with 0.7 a common default), the distribution becomes more balanced. The model is willing to consider alternative tokens that are plausible but not dominant. The model produces natural, varied text but introduces some risk of unexpected choices.

Above 1.0, the distribution flattens further, giving less likely tokens a greater chance of being selected. Higher temperature can help explore alternative wording or ideas, but it can also increase the risk of drifting away from the intended domain.

Figure 2.8 shows this visually.

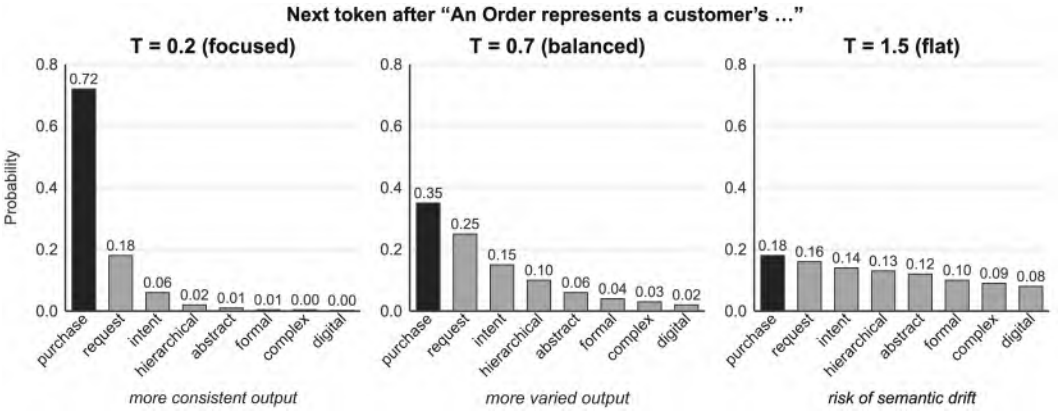


Figure 2.8 – Temperature and output variability (illustrative probabilities)

The figure shows the same generation context – "*An Order represents a customer's…*" – at three different temperatures. At temperature 0.2, the model overwhelmingly selects *purchase*, the statistically dominant continuation in e-commerce contexts. At temperature 0.7, alternatives such as *request* and *intent* gain non-trivial probability. At temperature 1.5, the distribution is nearly flat: *hierarchical*, *abstract*, and *formal* become almost as likely as *purchase*. This wider distribution increases the risk of semantic drift.

Decoding controls in practice

Beyond temperature, you can also control token selection with **nucleus sampling** (also known as **top-p sampling**; Holtzman et al., 2020). The model samples from the smallest set of the most probable tokens whose combined probability reaches a chosen threshold, such as 0.9. What this protects you from in domain work is specific: the long tail of low-probability tokens, which is where a term such as *hierarchical* or *directive* slips into a definition of *Order*. As an illustrative tuning step, lowering top-p to around 0.8 may exclude some low-probability candidates from consideration, while temperature only changes how likely the remaining candidates are to be selected. When temperature flattens the distribution, more tokens are needed to reach the threshold, so the two controls interact. Top-p filters by probability, and it does not check whether the surviving tokens suit your domain.

For domain modeling, start with a low temperature, such as 0.2 or 0.3, when you want consistent names and structured definitions for events, commands, aggregates, or invariants, and leave top-p at its default until you see unrelated vocabulary appearing; lowering it to around 0.8 is the more direct remedy for that. Check the output against your domain vocabulary and rules, because neither control validates the model's suggestions. Use higher temperatures when deliberately exploring alternatives.

Note

Try it yourself

Most API playgrounds expose temperature and top-p directly. Take a prompt such as: *Define the Order aggregate for BrewUp in two sentences* and run it five times at temperature 0.2, then five times at 1.0. Count how many of the 10 answers use the same names for the aggregate's key concepts, and how many introduce a term you never used. The first number is consistency; the second is drift.

What this means for your domain work

Consolidate the mechanics we have explored into five practical principles that will guide everything you do from this point forward.

First, *use natural language for domain concepts in your prompts*. Write *bounded context*, not *BoundedContext*. Write *aggregate root*, not *AggregateRoot*. The natural language forms appear frequently in DDD literature. CamelCase fragments the terms at the tokenizer level and weakens the statistical associations that make the model's output aligned with the intended domain.

Second, *structure your context window strategically*. Place domain rules and constraints at the beginning and end of your prompt. Avoid burying critical information in the middle of long contexts. Liu et al. measured the Lost in the Middle degradation across several models, so it is a useful risk to account for when structuring prompts.

Third, *do not equate context length with quality*. A focused, well-structured prompt with explicit constraints can outperform a sprawling prompt. Extra tokens add material for the model to weigh, and often that material is noise.

Fourth, *choose low temperature for domain definitions*. For event names, aggregate boundaries, or invariant rules, a low temperature, such as 0.2 or 0.3, can reduce variation between outputs. Check those outputs against the domain vocabulary and rules. Use higher temperatures only when you are deliberately exploring alternatives.

Fifth, *keep domain vocabulary and constraints available throughout the workflow*. If an earlier message defined *Order* as an aggregate with specific invariants, that definition can still guide the model while it remains in the current context. Check that the relevant information is present and current, and retrieve or restate it from the rulebook when needed.

These five principles follow directly from how tokenization, context windows, attention, and predictive chains interact: tokenization determines the units through which the model receives language, the context window determines which information is available and its position, and

attention shapes the relationships that follow. Together, they make wording, structure, and context placement active controls over the model's probability space, and every structured prompt technique you will build in Part 2 is designed to work with them. The same mechanics, however, also produce something that looks like reasoning, and the next section looks at where that resemblance ends.

Emergent reasoning and its blind spots

Earlier, you saw that LLMs exhibit behaviors that resemble reasoning: they can follow multi-step chains of inference and solve structured problems, because large-scale pattern compression produces approximations of reasoning structures. This emergent reasoning is useful, but it has boundaries.

First, *reasoning depth can be fragile*. If a prompt lacks structure, the model might collapse into shallow pattern completion. Clear step constraints (analyze assumptions first, identify invariants, evaluate trade-offs) significantly improve performance because they shape the predictive path.

Second, *consistency is not guaranteed across outputs*. Because there is no stable world model, the LLM can generate contradictory outputs across separate interactions, or even within the same interaction if context grows noisy.

Third, *it lacks intrinsic error detection*. Humans experience cognitive dissonance when contradictions appear. LLMs do not. Unless prompted to check for inconsistencies, the model will continue generating locally coherent text even if earlier statements conflict.

Fourth, *abstraction boundaries are statistical, not structural*. When discussing domain models, the LLM may blur boundaries between concepts that frequently co-occur in training data. The model's representations are shaped by statistical relationships among those concepts, not by the explicit boundaries you establish in your domain model.

These blind spots explain many common failure modes:

- Confident hallucinations, such as stating without hedging that an order can be canceled after it has shipped, which BrewUp's invariants forbid
- Subtle logical inconsistencies
- Over-generalized recommendations
- Invented domain rules, such as a minimum order quantity that BrewUp never had
- Architectural over-engineering, such as separate bounded contexts for Order, Payment, and Shipping when BrewUp needs one

Each of these follows from probabilistic generation without grounded validation, so they will keep appearing until something in the workflow does the grounding.

For architects, this reframes how LLMs should be used. Rather than asking only for answers, ask the model to make its assumptions and intermediate conclusions explicit, then constrain, inspect, and validate the result.

Treated as a structured reasoning simulator rather than an oracle, the model becomes useful for exploring design options quickly, simulating objections, testing invariants, and exposing implicit assumptions. All of that depends on precision, which is the subject of the rest of the chapter. If the mechanism is prediction over a probability distribution, then the words you choose are the main control you have over it.

Precision in language as the core of domain modeling accuracy

In DDD, Ubiquitous Language is the structural backbone of the model, not a matter of documentation hygiene. A shared vocabulary between domain experts and developers ensures that every term maps to a specific concept and every concept lives within a clear boundary. When language becomes ambiguous, models fragment; when terms are used inconsistently, invariants weaken.

This principle becomes even more critical when collaborating with LLMs, where language is one of the main controls you have over what the model produces.

An LLM does not understand your domain, validate your aggregate boundaries, or protect your invariants. It predicts the next token based on statistical likelihood within a constrained context window. To obtain accurate domain modeling support, then, you must shape the probability distribution within which the model operates, and language shapes that distribution.

For human teams, Ubiquitous Language creates alignment through shared meaning. For LLMs, disciplined language creates alignment through constrained prediction. Vague phrasing is not just imprecise: it expands the model's probability space. An ambiguous term multiplies the plausible continuations, an unspecified boundary lets the generation drift, and an overloaded concept invites the model to branch into a meaning you never intended.

Consider the difference between these two options: "*Define events for the system*" and "*Define domain events for the Order bounded context in BrewUp. Use past tense naming. Do not introduce new aggregates. Respect the invariants listed in the following.*". The first prompt invites improvisation. The second constrains generation.

Written out in full, the second prompt looks like this:

```
Context: BrewUp, Order bounded context.
Aggregate root: Order.
Task: define the domain events raised by the Order aggregate.
Rules:
- Name events in past tense, as <Aggregate><Verb-ed>, for example OrderPlaced.
- Do not introduce new aggregates or new bounded contexts.
- Every event must be consistent with the invariants below.
Invariants:
- An order with no lines cannot be placed.
- An order cannot be shipped before payment is confirmed.
- A shipped order cannot be cancelled.
Output: one line per event, with the name and a one-sentence description.
```

GPT-5.6 Sol returned the following in September 2026, verbatim, in a fresh conversation with no system prompt:

```
OrderPlaced — Raised when an order containing at least one line is successfully
placed.
OrderPaymentConfirmed — Raised when payment for the order is confirmed, making the
order eligible for shipment.
OrderShipped — Raised when a paid order is shipped, after which it can no longer
be cancelled.
OrderCancelled — Raised when an order that has not been shipped is cancelled.
```

The answer stayed within the supplied constraints: every event is a past-tense fact about *Order*, and each one respects an invariant. Two of the descriptions even restate an invariant in their own words, so the constraints shaped the content and did not merely filter it. The gain came from the prompt closing off the paths that would have led elsewhere.

At the mechanical level, precision narrows the available predictive paths. When you specify the bounded context, the aggregate root, the naming convention, and the invariants, you are narrowing the distribution of valid next tokens.

This is why inconsistent vocabulary is so dangerous. If you alternate between *Order*, *purchase*, and *Request* to describe the same concept, the model interprets them as statistically distinct signals.

Attention mechanisms will connect them only weakly, based on pattern frequency rather than intent. Over time, the predictive chain drifts. The model might introduce distinctions you never designed.

The model mirrors your linguistic discipline, which is why vocabulary drift in a prompt shows up as boundary drift in the output.

There is a deeper implication. In human collaboration, Ubiquitous Language aligns mental models. In AI collaboration, Ubiquitous Language aligns probability distributions. The same discipline that improves conversation between architects and domain experts also improves interaction between architects and LLMs.

Despite this alignment, an important limitation remains: an LLM may continue generating even when something is ambiguous or domain constraints are missing. The model can ask clarifying questions, but it might instead produce a plausible response based on unsupported assumptions. Missing constraints can therefore go unnoticed, and the output can remain fluent while violating the intended meaning of the domain.

This need for precision explains why precision becomes essential. Precision in language performs three critical functions in AI-assisted domain modeling. First, it constrains hallucination. Explicit rules leave fewer degrees of freedom for invented details. Second, it stabilizes predictive chains. Clear early tokens anchor the generation in the intended semantic field. Third, it exposes modeling weaknesses. If you struggle to describe a boundary precisely, inconsistent output often reflects a real conceptual gap.

In this sense, the LLM becomes a linguistic test for your domain model. When the output drifts, look first at where your own language left room for the drift.

This reframes the architect's role: the work is to design linguistic constraints that steer the model's generation and narrow its probability space.

The core insight of this chapter is:

- LLMs are probabilistic engines
- Domain models are linguistic artifacts

The accuracy of AI-assisted domain modeling depends on how disciplined that language is.

For architects, this expands the object of design. You are not only designing software structures; you are also designing linguistic constraints through which AI participates in domain modeling. The prompts, rulebooks, and agent workflows introduced in the chapters ahead are practical expressions of this responsibility.

Summary

LLMs are not domain experts in miniature. They are probabilistic systems that generate one token at a time, shaped by tokenization, context, attention, and the sequence of choices already made. These mechanics explain both their fluency and their fragility: the same process that produces a convincing architectural argument can also produce a coherent fabrication.

For architects, the consequence is practical. Larger context windows can still obscure relevant information, and fluent output might remain ungrounded. Plausible domain concepts therefore still require validation against the actual domain, with context curated and constraints made explicit.

This practical consequence is where Domain-Driven Design becomes especially powerful. Ubiquitous Language, bounded contexts, invariants, and explicit domain rules do more than align people: they narrow the model's probability space. Disciplined language turns AI from an unconstrained generator into a useful reasoning amplifier.

The responsibility remains asymmetrical. LLMs optimize coherence, while architects and domain experts remain responsible for correctness. Once AI participates in domain modeling, the words used to describe the system also help design it.

In the next chapter, you move from model mechanics to architectural principles for structuring conversations, expectations, and responsibilities. These principles should remain stable even as AI technologies change.

References

- Anthropic (2026). models overview. `https://platform.claude.com/docs/en/models/overview` (accessed September 2026).
- Gage, P. (1994). A New Algorithm for data compression. The C Users Journal, 12(2), 23–38.
- Google (2026). Gemini API: Long context. `https://ai.google.dev/gemini-api/docs/long-context` (accessed September 2026).
- Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The Curious Case of Neural Text Degeneration. International Conference on Learning Representations (ICLR 2020). `https://arxiv.org/abs/1904.09751`

- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. Transactions of the Association for Computational Linguistics, 12, 157–173. `https://doi.org/10.1162/tacl_a_00638`
- OpenAI (2024). tiktoken: a fast BPE tokenizer for use with OpenAI's models. `https://github.com/openai/tiktoken`
- OpenAI (2026). models. `https://developers.openai.com/api/docs/models` (accessed September 2026).
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems 30 (NeurIPS 2017). `https://arxiv.org/abs/1706.03762`

3

The Architect's View: Reasoning with AI Systems

Conductors do not play every instrument. They may not understand violin technique better than the violinist in the front row. Yet they determine whether the ensemble produces music or noise. Their competence lies not in execution but in structure. They shape how information flows between sections, decide when to impose constraints and when to leave space, and determine how coherence emerges across the whole ensemble.

This role is also the role of an architect collaborating with a Large Language Model. What matters is not how well you understand the model's internal mechanics, but how well you design the interaction so that it produces coherent, grounded, and domain-accurate reasoning.

The previous chapters explained what an LLM is and why it behaves the way it does. This chapter addresses a different problem entirely: How should an architect think when reasoning with AI?

This chapter is the bridge between theory and practice. Before you write structured prompts, build rulebooks, and design specialized agents in Parts two and three, you need a stable mental framework that remains valid regardless of which model you are using today or will use tomorrow.

Four architectural principles form this framework: treating conversations as designed systems, engineering information flows deliberately, building shared vocabulary as a formal contract, and using constraints as conceptual boundaries. BrewUp will appear briefly in this chapter as a reference point, but the real subject is the architect's way of thinking. The four sections that follow take these principles one at a time, with each building on the last. By the end of the chapter, the four shifts in perspective add up to a different way of working with AI.

In this chapter, you will learn:

- *Architecting conversations, not just systems*: why interacting with an LLM means designing a reasoning workflow.
- *Designing information flows for AI*: how the structure and position of information inside a context window determine output quality.
- *Creating shared vocabulary between you and the model*: why a disciplined Ubiquitous Language is both a DDD asset and a probabilistic steering tool.
- *Introducing constraints as conceptual boundaries*: how rules and guardrails function as the bounded contexts of your AI interaction design.

You can begin by dismantling the most common misconception about working with AI.

Architecting conversations, not just systems

Ask most developers how they use an LLM in their work, and they will describe something like this: they open a chat interface, type a question, read the answer, and ask a follow-up. The mental model is familiar. It resembles a search engine, or a knowledgeable colleague available on demand. The interaction is framed as a *query*: you ask, the model answers.

This mental model creates an architectural risk because a query brings its assumptions with it: queries are transactional, stateless, expect a single complete answer, and require no design. In the context of Domain-Driven Design, where the challenge is sustained, iterative reasoning about complex and evolving domains, a transactional mental model produces exactly the kind of output you should expect from a transaction: shallow, decontextualized, and unreliable.

The limit of the prompt paradigm

The term *prompt engineering* has become widely used, but it carries a subtle misdirection. The word *engineering* suggests rigor, but the word *prompt* still implies a single artifact: you write it, you send it, you receive a result. What that framing leaves out is the process around it: a prompt is the entry point of a reasoning workflow.

When teams use LLMs as glorified search engines, the results are predictable. Prompts are written casually. Context is lost between interactions. Output is inconsistent because each invocation starts without memory of what was established before. Domain rules invented in one session are forgotten in the next. The model feels powerful and unreliable at the same time.

The issue lies in the interaction design: without clear structure, architects ask the model to reason in an unbounded environment.

Conversation as a system

Consider what actually happens during an effective AI-assisted modeling session. You provide domain context, establish rules, and ask the model to reason about specific events or boundaries. Then, you review the output, identify what drifted, refine your instructions, and re-inject corrected language. You iterate. At no point is this a single transaction.

What you are doing is operating a system. Like any well-designed system, it has identifiable components. *Table 3.1* lists the five that matter, and they recur through the rest of the book: the structured prompts, rulebooks, and specialized agents of Parts 2 and 3 are all built from these same components.

Component	Role in the AI interaction system
Prompt	Entry point: the structured request that initiates reasoning
Context	Working memory: the domain knowledge, rules, and history in scope
Rules	Guardrails: constraints that narrow the model's probability space
Feedback loop	Refinement mechanism: human validation and correction reintroduced
Validation	Human judgment: the architect's responsibility for semantic correctness

Table 3.1 – The five components of an AI interaction system and the role each one plays

This table describes the structure of an effective AI collaboration. Architects can design, evaluate, and improve each component. Each component has failure modes, and the quality of the overall output depends on how well these components are composed.

Recognizing this is the first architectural shift this chapter asks of you: stop thinking about prompts and start thinking about *conversation architecture*.

The AI reasoning loop

Once you adopt conversation architecture as the frame, a natural structure emerges. *Figure 3.1* illustrates the AI Reasoning Loop: the cycle that repeats inside a reasoning workflow. The reasoning workflow is the whole design session; the loop is the single turn of intent, prompt, output, and validation that the session repeats.

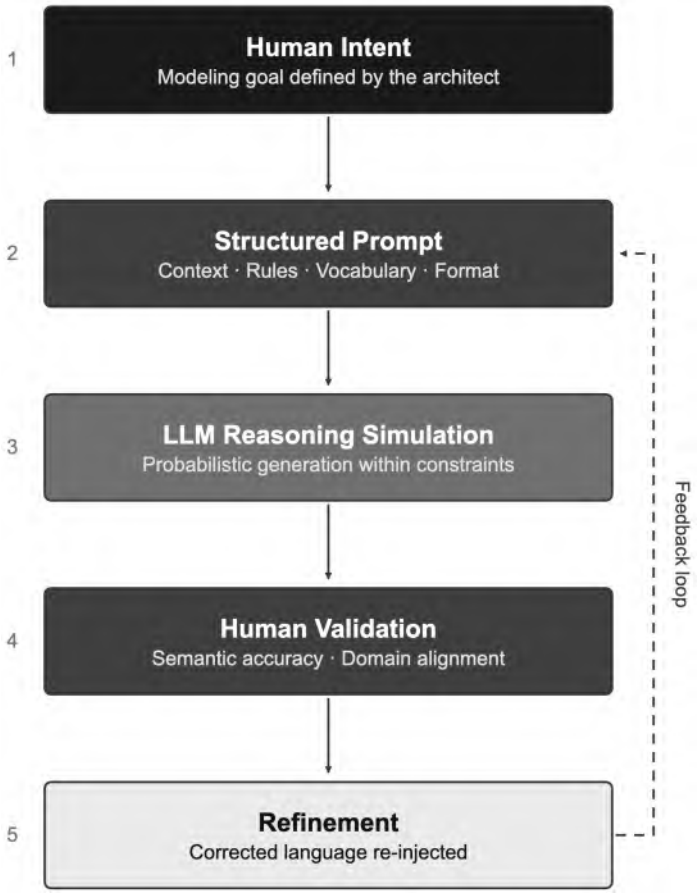


Figure 3.1 – AI reasoning loop

The loop begins with human intent: a specific modeling goal, such as identifying the core events of the Order bounded context in BrewUp. The architect translates that intent into a structured prompt that encodes context, rules, and the expected output format. The model simulates reasoning within that structure and produces output. The architect validates the output against domain knowledge and semantic accuracy, not just surface coherence. Corrections and refinements are reintroduced in the next prompt cycle.

The value of the loop lies in what carries forward between its steps. Each iteration carries forward the vocabulary, the rules, and the corrections established in the previous one, so ambiguity narrows across the session instead of resetting at every prompt.

Recognizing the DDD parallel

If this loop looks familiar, it should. The AI Reasoning Loop is structurally identical to the knowledge refinement cycle underlying the concepts of Domain-Driven Design. *Table 3.2* puts the two side by side: each step of the DDD cycle in the left column, its counterpart in the loop on the right.

DDD Cycle	AI Reasoning Loop
Explore the domain	Formulate human intent
Model the concepts	Structure the prompt
Test against reality	Validate model output
Refine and realign	Inject corrections and iterate

Table 3.2 – The DDD knowledge refinement cycle and its counterpart in the AI Reasoning Loop

The alignment follows from a shared purpose: both processes reduce ambiguity through structured iteration. DDD does it through conversations between architects and domain experts. The AI Reasoning Loop does it through structured conversations between architects and probabilistic reasoning engines. The discipline required is the same: explicit language, clear boundaries, continuous refinement.

A well-structured AI collaboration session can amplify DDD practice rather than shortcut it. The model does not replace the domain expert or the architect. The strategic questions – what matters in this domain, which trade-offs are acceptable, and where the boundaries belong – remain entirely human. What the model changes is the tempo and visibility of the loop. It accelerates refinement, surfaces in minutes ambiguities that might otherwise take a workshop to emerge, and externalizes implicit assumptions that would remain buried in informal conversations. If you have ever watched a modeling discussion stall because two experts were using the same word for different concepts, you know how valuable that externalization is.

Conversation as a bounded process

You can take this architectural thinking one step further. In DDD, a bounded context defines a clear scope: a specific domain, a specific language, a specific set of rules. Nothing crosses the boundary without explicit translation.

An effective AI conversation has exactly the same properties. It has a *purpose* (the modeling goal), *boundaries* (what is in scope for this interaction), and *language* (the domain vocabulary that governs all inputs and outputs). When a conversation lacks any of these, it behaves like a

poorly defined bounded context: concepts bleed across scope, language becomes inconsistent, and the output reflects the ambiguity of the design. The implication is clear:

You are not prompting a model. You are designing a reasoning environment.

Every element you place in that environment – the context, the rules, the vocabulary, the expected output format – is an architectural decision. Like all architectural decisions, it has consequences that compound over time. The components in *Table 3.1*, the loop in *Figure 3.1*, and the boundary described here are the whole of the first shift. The next one is about what goes inside that boundary.

Designing information flows for AI

If a conversation with an LLM is a system, then information is its raw material. As in any system, output quality depends on more than input quality: it depends on *how that input is structured, sequenced, and positioned*. This shift is the second architectural shift this chapter asks of you: from thinking about what you tell the model to thinking about how information flows into the model and what the model can actually perceive within that flow.

Most practitioners focus on content: what facts to include, what terms to define, and what examples to provide. Content matters, but poor placement can make useful information effectively invisible. The mechanics of the context window, which we examined in detail in *Chapter 2*, have direct architectural consequences that require deliberate design.

The stateless model and the sovereign context window

We established in *Chapter 2, How LLMs Actually Think and Why Precision Unlocks Understanding,* that an LLM has no persistent memory. Each interaction begins from zero. The context window is the model's *entire reality* for the duration of that interaction, and it holds only what you put there.

For architects, the consequence is simple:

If information is not available in the current context, the model cannot use it.

The domain rules you established two sessions ago are unavailable to the model unless you reintroduce them. The aggregate boundaries you defined yesterday do not exist unless they appear in today's prompt. The Ubiquitous Language you spent a workshop refining does not exist from the model's perspective unless it is explicitly present in the current context.

The stateless model is a *design constraint to work with*. Like the bounded context principle in DDD, which requires explicit translation at every boundary rather than assuming shared understanding, the stateless model requires explicit reintroduction of knowledge at every interaction rather than assuming continuity. The discipline is the same, even though the

medium differs. Reintroducing knowledge is only half the problem; what goes into the window, and in what order, is the other half.

Information architecture: layering what the model sees

Given that the context window is the model's complete reality, the question becomes: how do you architect what lives in that reality? The answer is information layering, the deliberate separation of different types of knowledge into distinct, ordered layers. Three terms carry this idea through the section. Information architecture is the principle, information layering is the structure it produces, and the information flow pattern is the order in which the layers reach the model.

The order runs from the broadest constraint to the narrowest, so that each layer can only narrow what the layers above it have already left open. *Table 3.3* sets out the five layers, what goes in each, and the architectural function each one performs.

Layer	Contents	Architectural Function
Domain Rules	Invariants, constraints, business policies	Establishes the semantic boundaries the model must not cross
Vocabulary	Ubiquitous Language terms and definitions	Aligns probabilistic prediction with domain-specific meaning
Task Instructions	The specific modeling goal for this interaction	Focuses the reasoning loop on a bounded scope
Examples	Concrete instances of correct output	Anchors generation in established patterns
Output Structure	Expected format, notation, naming conventions	Constrains the final token sequence

Table 3.3 – The five information layers, their contents, and the architectural function of each

These five layers represent different types of information that shape the model's probability distribution at progressively narrower levels of specificity.

When these layers are absent or disordered, the model compensates by inventing plausible content from its training distribution: it infers domain rules that sound reasonable, reaches for vocabulary that is statistically adjacent but semantically incorrect, and produces output in whatever format its training data suggested was typical for similar requests. The result is coherent in appearance and unreliable in substance. Each of those failures traces back to a

layer the architect left out, which puts the fix squarely in the architect's hands: decide what belongs in each layer before deciding what to ask.

The information flow pattern

Three terms are in play here, and they name three different things: *information architecture* is the principle that what the model sees, and where, determines what it can produce; the five *information layers* of *Table 3.3* are the structure that principle produces; and the *information flow pattern* is the order in which those layers reach the model.

In practice, effective AI-assisted domain modeling follows a consistent information flow pattern, shown in *Figure 3.2*:

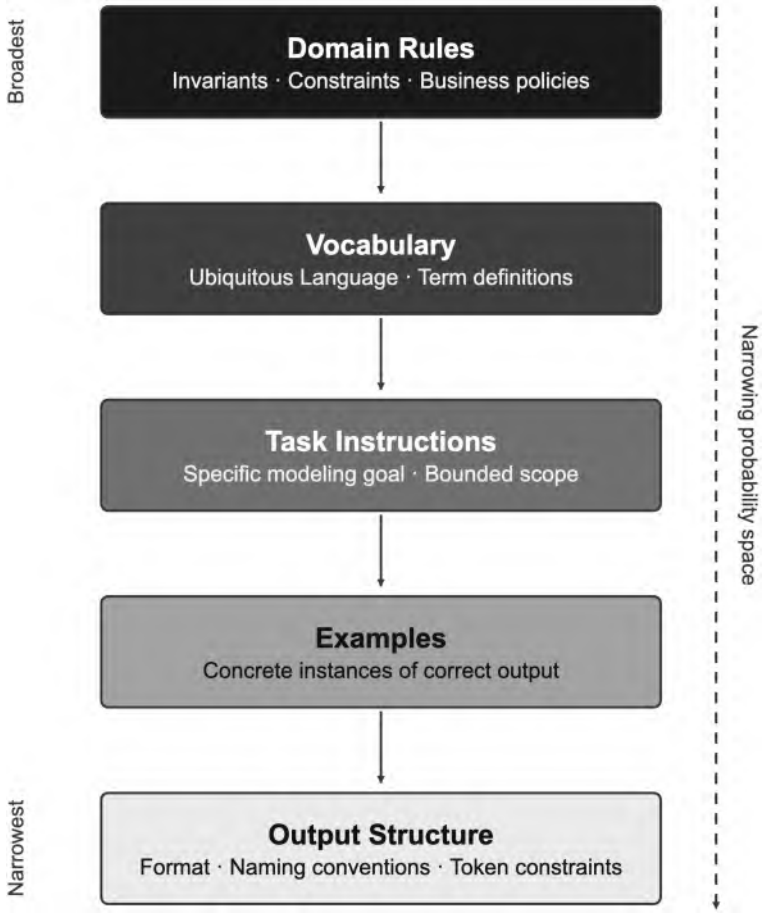


Figure 3.2 – Information layers in AI interaction

The pattern fixes the order in which information reaches the model. The pattern leaves the wording of any particular prompt open. Applied to BrewUp, the five layers look like this:

1. Domain rules declare the invariants of the Order bounded context: an Order cannot be shipped before payment is confirmed.
2. Vocabulary defines Order, `OrderPlaced`, and Customer exactly as the Ubiquitous Language uses them.
3. Task instructions ask the model to identify the domain events raised by the Order aggregate – and nothing else.
4. Examples anchor the expected style: `OrderPlaced` is one of them.
5. Output structure requires each event as a past-tense name with a one-line description.

Notice what this pattern anticipates: the structured prompts, rulebooks, and iterative refinement techniques you will build throughout Part 2. Every technique in the following chapters is an instantiation of this information architecture principle applied to specific DDD modeling tasks.

Information as probabilistic constraint

From the perspective of the mechanics described in *Chapter 2*, information in the context window acts as a *constraint on the probability space*.

Every piece of explicit information you inject narrows the range of statistically plausible continuations. Every domain rule you specify eliminates tokens that would violate it from the high-probability region. Every vocabulary term you define anchors a semantic association that overrides the model's broader statistical defaults. In general, explicit constraints narrow the probability space and can make the resulting reasoning more consistent with the intended domain.

This principle means that information design is *probabilistic engineering*. When you decide what to include in a context window, where to position it, and how to phrase it, you are making engineering decisions about the model's output distribution. The architect's intuition for system constraints, the instinct to make implicit assumptions explicit and to draw clear boundaries, is exactly the intuition required here.

The architect as information designer

That reframing carries a precise redefinition with it. Architects working with AI are *information designers*, responsible for structuring the environment in which reasoning happens.

In traditional software architecture, the architect designs the environment in which components operate: the network topology, the data flows, the service contracts, the boundary conditions. The components then execute within those constraints. In AI-assisted domain

modeling, the same decisions build a reasoning environment: domain rules, vocabulary, task scope, examples, output constraints – and the model generates inside it.

The parallel reflects a continuity in architectural practice. Deciding which invariants the model must see before generating is the same kind of decision as defining a service contract; scoping a modeling session to a single aggregate is the same kind of decision as drawing a context boundary. It reflects a genuine continuity of architectural thinking across two different kinds of systems.

This extends architectural design beyond software systems to the reasoning environment itself, and it prepares you for everything that follows in this book. Part 2 will give you the tools to instantiate this principle in concrete modeling tasks. Part 3 will show you how to scale it across specialized agents and orchestrated workflows. However, the foundation is here: in the recognition that AI collaboration is architecture, applied to a new kind of system.

Creating shared vocabulary between you and the model

Information layering tells you what information belongs in the context window and how it should be structured. However, a deeper question remains: what language should those layers speak?

The third architectural shift is to treat shared vocabulary as a formal contract with the model – a principle Domain-Driven Design practitioners already know well.

In *Chapter 2*, we established that language is a mechanism of probabilistic control: every word placed in the context window reshapes the distribution of possible next tokens. The more disciplined the vocabulary, the narrower the probability space in which the model operates.

Domain-Driven Design addresses this challenge through Ubiquitous Language: a shared vocabulary that aligns domain experts, developers, and architects around the same conceptual model. Each term corresponds to a specific concept, each concept belongs to a particular context, and continuous refinement progressively eliminates ambiguity. When collaborating with an LLM, the same principle applies, but the reason is different.

Human collaborators use Ubiquitous Language to synchronize mental models. For LLMs, it helps stabilize statistical associations.

This practice means that when you establish a disciplined vocabulary in your prompts and interactions, you are anchoring the model's predictions to a specific semantic field rather than teaching it your domain in the human sense. Instead of drifting across all the possible meanings a term might have in natural language, the model stays anchored to a single reading: within this conversation, the word *Order* refers to a specific aggregate, within a specific

business context, governed by specific rules. In other words, vocabulary becomes a contract between you and the model.

Without such a contract, the model falls back on the broad statistical patterns it learned during training. A term, such as *order*, might refer to a purchase request, a command hierarchy, a sequence in mathematics, or a sorting operation in a database query. All of these meanings coexist in the training distribution, and the model selects among them based on surrounding tokens. *Figure 3.3* shows the four meanings competing for the same token.

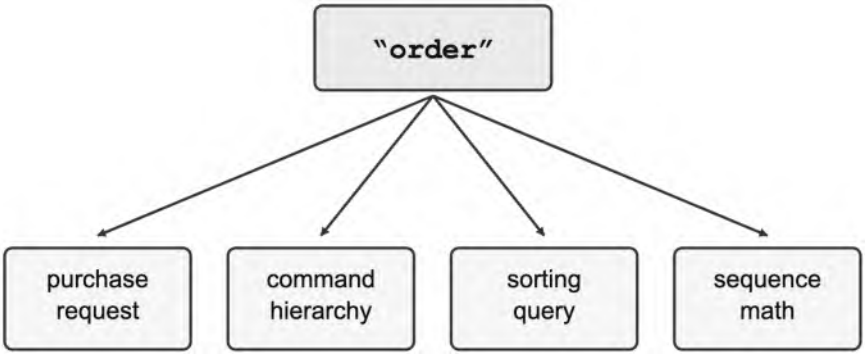


Figure 3.3 – The model statistically associates many meanings with the same word

Disciplined vocabulary narrows those alternatives toward the intended meaning.

Instead of asking "*Define events for the system*," you might establish the following vocabulary section:

Term	Meaning
Order	Aggregate root representing a customer purchase request.
OrderPlaced	Domain event raised when an order containing at least one line is successfully placed.
Customer	External entity initiating an Order.

Table 3.4 – A vocabulary section that fixes the meaning of three BrewUp terms for the model

The difference shows up in a single exchange. Asked only to define events for the system, with no vocabulary in the context window, GPT-5.6 Sol returned this in September 2026, in a fresh

conversation with no system prompt and reasoning effort set to high. The excerpt below uses marked cuts; the meaning is otherwise unchanged:

```
In a system, events are occurrences or changes that trigger the system to perform
an action or update its state.
[…]
User events: login, logout, button click, form submission, purchase request.
[…]
If you tell me which specific system you mean, I can define its complete set of
events.
```

The response landed in the wrong register: interface and infrastructure events rather than domain events, followed by a request to clarify which system was meant. With the vocabulary section placed in front of the same request, in a second fresh conversation, the answer changed:

```
OrderPlaced — Represents the fact that an Order has been successfully placed by a
Customer.
[…]
Raised when: an Order transitions from an unplaced/draft state to placed, and the
order contains at least one line.
[…]
Other events such as OrderLineAdded, OrderCancelled, or OrderShipped should only
be introduced if those transitions are meaningful domain facts required by the
system; they cannot be inferred from the vocabulary given.
```

Three definitions did that work. `OrderPlaced` is described in the terms defined in *Table 3.4*, Customer appears as an identifier rather than as an entity inside the event, and the answer stays inside the three terms it was given. The last line is especially important: when asked about events in general, the model named the events it would not invent and explained why – they cannot be inferred from the vocabulary given. A vocabulary section tells the model what your words mean and, at the same time, where its knowledge of your domain stops.

Once these terms appear consistently in the context window, the model begins generating text that reflects those associations. Attention mechanisms connect repeated tokens across the conversation, reinforcing their semantic relationship. The predictive chain becomes increasingly stable because the same linguistic anchors appear again and again. *Figure 3.4* shows the stable chain that those definitions produce.



Figure 3.4 – A disciplined vocabulary collapses multiple statistical interpretations into a single semantic anchor for the model

This stability has two important consequences. First, semantic drift decreases: when vocabulary remains consistent, the model is far less likely to introduce alternative interpretations or substitute statistically adjacent terms. Without the vocabulary section, a session that starts with *order* might drift into *purchase*, *transaction*, or *request* within a few exchanges – each term pulling the model toward a slightly different statistical neighborhood. With the vocabulary in place, an Order remains an Order, with the same meaning, boundaries, and rules on every iteration.

Second, reasoning becomes cumulative: each pass through the AI Reasoning Loop reinforces the established language, allowing the conversation to evolve toward greater precision rather than resetting into ambiguity.

In practice, effective AI-assisted modeling sessions often begin with a short vocabulary definition. This is not because the model is incapable of handling domain concepts; the explicit vocabulary defines the semantic frame within which prediction occurs.

Curating the language of the interaction is therefore part of the architect's role. Every repeated term strengthens a conceptual anchor, while inconsistent synonyms introduce probabilistic noise, and concepts left undefined invite the model to improvise.

When you maintain a disciplined vocabulary across prompts, corrections, and examples, the model begins to mirror that discipline. Over time, the conversation behaves less like a free-form chat and more like a structured modeling session governed by shared terminology. This stability is why Ubiquitous Language does a different job in AI collaboration than it does within a human team.

Humans tolerate ambiguity and resolve it through dialogue; LLMs resolve ambiguity through probability.

Clearer vocabulary generally makes the resulting reasoning more predictable.

Introducing constraints as conceptual boundaries

If shared vocabulary stabilizes meaning by defining what concepts exist, constraints stabilize reasoning by defining how those concepts are allowed to behave. In Domain-Driven Design, the bounded context performs this role. A bounded context establishes the limits within which a particular model is valid. Inside the boundary, language is consistent, and rules apply. Outside the boundary, translation is required.

This same principle becomes the fourth architectural shift this chapter asks of you: using constraints as conceptual boundaries that keep the model's reasoning inside the domain.

Without explicit constraints, the model's reasoning space expands to the full scope of its training data. The result is often impressive but unreliable: suggestions that sound plausible but violate domain invariants, introduce new aggregates, or mix concepts that belong to different parts of the system. Constraints prevent expansion. They define the boundaries of the reasoning environment in which the model operates.

Consider the following prompt, which echoes the contrast introduced in *Chapter 2, How LLMs Actually Think and Why Precision Unlocks Understanding*:

"*Design a domain model for orders*"

The first prompt is weak, but its weakness has nothing to do with the quality of the answer it produces. Under the same conditions– a fresh conversation, no system prompt, reasoning effort high – GPT-5.6 Sol returned this:

```
Related aggregates
Keep these separate from the Order aggregate when the system becomes non-trivial:
Payment · Shipment · Refund
[…]For a typical e-commerce system, I'd structure the bounded contexts roughly as:
Catalog · Ordering · Payments · Fulfillment
```

It is competent, textbook DDD. It is also not BrewUp. Three aggregates appear that nobody asked for, invariants about refunds and partial shipments that the Order context does not have, and – in the last line – four bounded contexts. Deciding where those boundaries belong is the one judgment this book insists stays with the architect. Given no boundary in the prompt, the model drew them itself.

Now, compare it with the following:

"*Within the Order bounded context of BrewUp, identify domain events raised by the Order aggregate. Do not introduce new aggregates and respect the invariants listed in the following section.*"

The invariants it points to are BrewUp's own, the three that *Chapter 2, How LLMs Actually Think and Why Precision Unlocks Understanding,* attached to the same request: an order with no lines cannot be placed, an order cannot be shipped before payment is confirmed, and a shipped order cannot be canceled.

The second prompt substantially narrows the range of plausible continuations. The model is no longer free to invent surrounding structures. The model must operate inside the conceptual boundaries you defined. *Chapter 2, How LLMs Actually Think and Why Precision Unlocks Understanding,* printed what that second prompt returned: every event in the answer was a past-tense fact about Order, and the answer stayed within the supplied constraints.

In probabilistic terms, constraints narrow the prediction space. Every rule you specify removes potential token sequences from the high-probability region. A scope declaration does this from the other side: naming the Order bounded context places those tokens in the window, where attention connects them to everything generated afterward, so continuations that belong to Payment or Shipment have nothing in the context supporting them, and their probability falls accordingly.

If the model attempts to introduce a new aggregate, the constraint contradicts it. If it attempts to redefine a term, the vocabulary prevents it. Over time, these constraints shape the generation process the same way architectural boundaries shape a software system. Rules, invariants, and scope declarations should therefore appear explicitly in structured prompts as guardrails.

Typical constraint categories include:

Constraint Type	**Purpose**
Scope constraints	Define which bounded context the reasoning applies to.
Structural constraints	Prevent the introduction of new aggregates or entities.
Behavioral constraints	Enforce invariants and domain rules.
Output constraints	Specify the format or structure of the response.

Table 3.5 – The four constraint types that bound the model's reasoning space

Together, the four types in *Table 3.5* create what you call a `bounded reasoning environment`. Inside it, the model explores, proposes, and structures; outside it, it has no license to operate.

This mirrors the design philosophy of Domain-Driven Design itself: just as bounded contexts protect conceptual integrity, architectural constraints protect reasoning integrity. When the boundaries are clear, models remain coherent; when they are vague, conceptual leakage occurs – concepts drift between contexts, rules become inconsistent, and language loses precision.

Constraints focus the model's generative power where it matters. Within well-defined boundaries, the model becomes an effective reasoning amplifier. Outside those boundaries, it becomes a speculative generator. The architect's work is to design the conceptual boundaries inside which generation occurs.

Vocabulary aligns meaning, and constraints align scope; together, they transform an open-ended AI conversation into a structured reasoning environment analogous to a well-designed bounded context.

Summary

In this chapter, we shifted perspective from understanding how LLMs work to understanding how architects should reason with them. The central idea is that effective AI collaboration requires the architect to design the environment in which reasoning happens.

Instead of treating a prompt as a single request, we framed AI interaction as a structured reasoning system, and the architect's role within it as one of information design: what the model sees, how it is layered, and which constraints apply determine the probability space from which every response is drawn.

The chapter's four architectural shifts are:

- *Conversations are systems, not queries* – the unit of design is the reasoning workflow.
- *Information flow determines output quality* – the context window is the model's entire working memory.
- *Shared vocabulary stabilizes meaning* – a disciplined Ubiquitous Language reduces semantic drift.
- *Constraints create reasoning boundaries* – explicit rules and scope declarations work like bounded contexts.

Taken together, they are less four separate techniques than one practice seen from four angles: the conversation gives you a system to design, information flow gives you its raw material, vocabulary makes that material mean what the domain means, and constraints keep the reasoning from leaving the domain. An architect who applies them together is no longer

prompting a model – they are operating a bounded reasoning environment. This is the mental framework the rest of this book builds on.

In the next chapter, we move from architectural principles to the first concrete collaboration technique between humans and AI. We will use EventStorming, one of the most powerful domain discovery practices in Domain-Driven Design, as a bridge between human insight and machine reasoning. You will see how the messy, visual exploration of a domain can be translated into structured prompts that allow an LLM to identify events, surface patterns, and accelerate domain clarity while remaining grounded in disciplined language.

Part 2

Apply AI to DDD with Simple Rules and Prompts

In this part of the book, you'll learn to model domains with AI through rules, prompts, and iterative refinement, producing coherent events, commands, aggregates, and bounded contexts. You'll move from domain exploration and EventStorming to reusable prompt rulebooks and prompts that preserve domain knowledge.

This part of the book includes the following chapters:

- *Chapter 4*, *From Domain Exploration to AI Collaboration*
- *Chapter 5*, *Defining Rules: Teaching the Model to Think in DDD*
- *Chapter 6*, *Iterative Refinement: Building the Bounded Context*
- *Chapter 7*, *Prompt as Domain Carrier*

4

From Domain Exploration to AI Collaboration

In the previous chapters, you established a hard truth: a Large Language Model does not understand a business domain in the human sense. It predicts likely continuations from the context it receives.

This changes the conditions under which AI becomes useful for software architecture. AI can amplify your reasoning only when you make knowledge, constraints, and uncertainty explicit. Without that discipline, the model might produce fluent and plausible answers while silently drifting away from the domain you are trying to understand.

This leads to an immediate practical question:

Where does the material required for disciplined AI collaboration come from?

Before AI can help you reason about a business domain, you must make that domain visible. Its events, actors, decisions, terminology, ambiguities, and causal relationships must be externalized in a form that humans can discuss and that a model can process. This chapter moves from the conceptual foundation of the previous chapters to the first operational step of AI-assisted Domain-Driven Design: domain exploration.

Using BrewUp as a reference, you will examine how exploration artifacts can surface candidate events, reveal causal flows, expose ambiguity, and provide the raw material for more disciplined collaboration with AI. By the end of this chapter, you will understand:

- Why domain exploration is a prerequisite for effective AI collaboration in DDD.
- How EventStorming externalizes domain knowledge and turns workshop output into structured signals.

- How Domain Storytelling complements events with narrative flow, actor perspective, and causality.
- How you can translate exploratory artifacts into structured domain context and refine them into reusable modeling material.

One fragment of the BrewUp ordering flow runs through this chapter. It begins as four sticky notes on a workshop wall – Order submitted, payment authorized, stock reserved, order confirmed – and ends as a file the team can reuse, in which each name carries a definition of what it does and does not mean:

```
## Order confirmed

A domain event raised when BrewUp commits to fulfilling the order after the
required business conditions have been satisfied.
```

Discover makes those notes visible, Refine decides which of them matter, Name fixes what they mean, and Reinforce carries the names into the next conversation. That progression – ***Discover*** → ***Refine*** → ***Name*** → ***Reinforce*** – is the structure of this chapter. You begin where good modeling always begins: by making the domain explicit enough for people to challenge it and for AI systems to operate within it.

Technical requirements

The techniques discussed in this chapter do not depend on a specific AI model or development environment. To follow the examples, you need:

- Access to an LLM interface that supports multi-turn conversations and textual context.
- A text editor capable of working with Markdown files.
- Access to the exploration artifact and working vocabulary used in this chapter: `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design/tree/main/docs/ch04`

The examples and artifacts shown in this chapter are intentionally tool-independent. Their purpose is to illustrate how domain knowledge can transform into a structured context, not to demonstrate the features of a particular product.

Tools that are a bridge between humans and AI

Complex business domains often exceed what a single person can reliably hold in their head. As a system grows, knowledge becomes distributed across domain experts, developers, documents, conversations, and existing software behavior.

The problem is not simply the amount of information. Different people hold different interpretations of that information. Terminology drifts, assumptions remain implicit, and people understand causal relationships differently depending on who is describing the process.

Before AI can assist with reasoning, this fragmented knowledge must be externalized. The knowledge does not need to be complete or perfect. The knowledge must, however, be visible enough to create a shared surface where assumptions can be challenged, terminology can be compared, and missing knowledge can be identified. Domain exploration techniques provide that surface.

EventStorming

EventStorming is a collaborative domain exploration technique designed to externalize business knowledge visually. Domain experts, developers, and other stakeholders work together to map meaningful events across a shared surface. Events are usually expressed in the past tense because they describe the facts that have already occurred in the domain. Commands, actors, policies, external systems, and unresolved questions can then be added by the team, allowing them to explain what caused those events and what happened as a consequence.

The purpose is not merely to document an existing process. The shared surface becomes a temporary memory for the group. It allows participants to inspect the domain together instead of relying on separate mental models. Duplicated events, inconsistent terminology, missing transitions, and conflicting interpretations become visible because they appear next to one another.

Consider a small fragment of the BrewUp ordering domain. During an early workshop, participants might surface expressions such as:

- Sales order created
- Payment authorized
- Stock reserved
- Confirmation sent

At this stage, these notes are not yet reliable: they are candidate statements about the domain. Some may represent meaningful business facts. Others might describe technical reactions, user-interface actions, or assumptions that have not yet been validated.

For humans, the arrangement of those notes provides valuable context. Position, sequence, grouping, annotations, and the discussion that produced them all contribute to their meaning. For an AI system, however, the team must make that meaning explicit.

A photograph or transcription of the wall cannot reliably preserve everything that happened during the workshop. The transcription may capture the words but lose why two notes were separated, why one term was rejected, which boundary a group intended, or which question remained unresolved.

The bridge to AI is therefore the disciplined translation of the wall into structured language. That translation records what the workshop produced – its events, relationships, terminology, boundaries, and open questions – in a form that outlives the session.

Once expressed explicitly, these elements become constraints within the model's context. The model still does not understand the domain as a domain expert does, but it can reason more consistently within the boundaries that the team has provided.

Figure 4.1 shows a simplified fragment derived from the BrewUp ordering flow. The `CreateSalesOrder` command leads to the `SalesOrderCreated` event, which is then used to update the sales read model and support the Sales UI. The external system shown in the diagram represents another participant in the same flow.

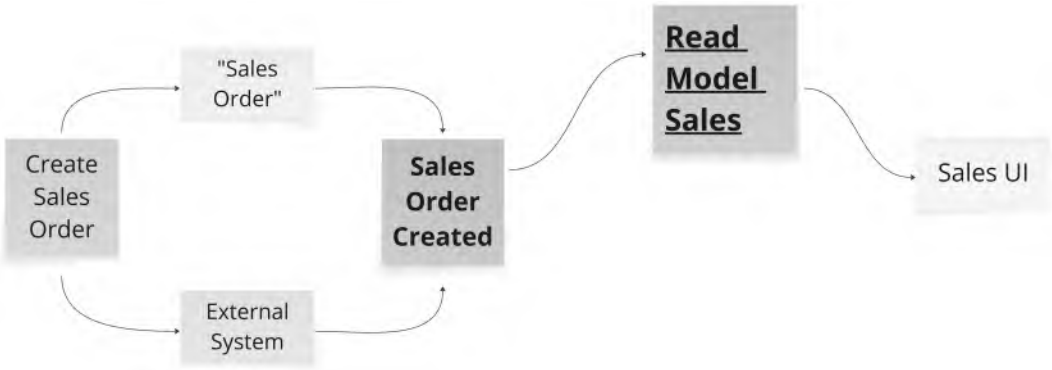


Figure 4.1 – An EventStorming example from the BrewUp ERP

By translating discovery results into explicit events, sequences, and vocabulary, you move part of the cognitive burden from individual memory into a reusable context. The architect can then focus less on recalling every note and more on higher-level concerns:

- Are the boundaries meaningful?
- Are you mixing business facts with technical consequences?
- Which assumptions still need validation?
- Which terms remain stable?
- Which transitions are missing?

In this sense, EventStorming works as a translation layer between human cognition and probabilistic reasoning. It turns implicit knowledge into visible artifacts and those artifacts into structured input that AI can process without allowing it to define the domain on its own.

Domain Storytelling

EventStorming gives you breadth. EventStorming can quickly surface a large number of business facts, questions, and possible relationships.

An event landscape, however, does not always preserve the perspective of the people involved. The event landscape may tell you that something happened without fully explaining who acted, what they were trying to achieve, and why one step led to the next. For that, you need a complementary narrative view.

Domain Storytelling explores a domain through concrete scenarios. Instead of beginning primarily with events, it describes how actors interact with systems and work objects over time. A simple scenario might describe a customer placing a sales order. The sales system validates the request, an external system authorizes the payment, and the order is confirmed once the required business conditions are satisfied.

The scenario is then represented visually using actors, activities, work objects, and a numbered sequence. The goal here is behavioral clarity rather than technical precision:

What actually happens in this domain from the perspective of the people involved?

This narrative approach has an important cognitive effect. People naturally explain work through stories. Stories preserve sequence, reveal intent, and connect individual actions through causality. They also make assumptions easier to challenge.

For example, saying that "*the order is confirmed*" might appear unambiguous in an event list. Placing the same statement inside a story immediately raises additional questions:

- Who confirms the order?
- What conditions must be satisfied first?
- Is payment authorization sufficient?
- Must stock already be reserved?
- Does confirmation create a commitment to the customer?

These questions expose knowledge that an isolated event name might hide. *Figure 4.2* shows the same flow as *Figure 4.1*, now seen from the actor's perspective. The figure shows the participants in a sequence of activities. They include the user, the sales system, an external system, the sales order, the event produced by the interaction, the read model, and the Sales UI.

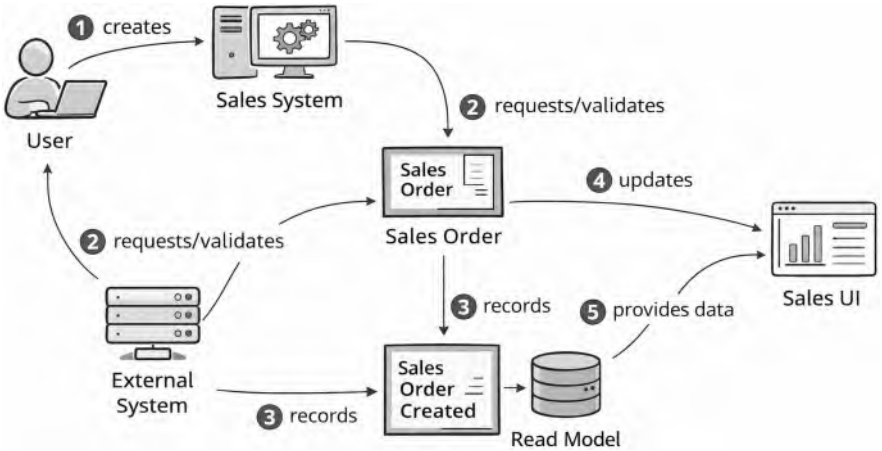


Figure 4.2 – The same process example using Domain Storytelling

Where EventStorming tends to surface many relevant facts quickly, Domain Storytelling slows the exploration down and connects those facts into coherent flows. For AI collaboration, this distinction matters. An event list tells the model what happened. A domain story also identifies who initiated the action, which work object was involved, what the actor intended to achieve, which conditions influenced the flow, and why one action preceded or followed another.

This additional structure reduces the number of plausible interpretations available to the model. An event list that jumps from `OrderSubmitted` to `OrderConfirmed` is compatible with many different business processes: the model can insert a payment step, a credit check, or nothing at all, and each of those completions looks reasonable on its own. A story that records who submitted the order, what they were asking for, and which condition BrewUp checked before confirming rules out many of those completions. The model is doing the same thing it did before, with fewer gaps to fill from its training data.

Comparing EventStorming and Domain Storytelling

EventStorming gives you an event landscape. Domain Storytelling gives you narrative continuity.

Events describe what happened. Stories explain who acted, what they acted on, and how one step led to the next. The two techniques expose different dimensions of the same domain and work together.

EventStorming expands discovery by surfacing business facts, questions, reactions, and possible boundaries. Domain Storytelling organizes part of that material into scenarios that preserve actors, intent, and causality.

When combined, they provide a richer context in which both humans and AI can operate. EventStorming reveals the domain vocabulary and its candidate structural anchors, Domain

Storytelling tests those anchors against realistic behavior, and the comparison between the two exposes inconsistencies, missing transitions, and unstable terminology.

The architect's role is to shape the material from which reasoning begins, while responsibility for interpreting the domain remains with the team. Domain exploration produces raw materials. The next step is to translate that material into a form that can survive outside the workshop and become a reusable context for AI collaboration.

Turning visual chaos into structured context

The central principle is to organize the discovery material before it reaches the model. Asking the model to impose structure on ambiguous material can simply move that ambiguity into the output. The team should therefore make the structure explicit first.

An EventStorming wall or a Domain Storytelling diagram contains far more information than the words written on it. Humans infer meaning from position, color, proximity, sequence, grouping, and from the conversations that took place while the participants created the artifact.

Two events placed far apart may belong to different business capabilities. A cluster might suggest ownership. A question mark might preserve an unresolved business rule. A note that was moved several times might reveal disagreement about sequencing or responsibility.

Even when an AI model can process an image, it cannot reliably recover all the workshop knowledge behind those visual choices. It sees the resulting artifact, but not necessarily the discussions, rejected alternatives, or implicit conventions that gave the artifact its meaning. The goal is therefore not merely to transcribe the wall but to translate it.

Translation, not transcription

A transcription captures what is visible: the text written on each sticky note, the apparent sequence of the notes, the labels attached to actors and systems, and the groups visible on the shared surface.

A translation goes further. A translation makes the meaning behind those elements explicit.

Depending on the exploration material, that translation may include candidate domain events expressed in the past tense, the commands or intentions that might have caused them, the actors involved in the flow, and the causal relationships between actions and outcomes. A translation may also record recurring domain concepts and terminology, candidate boundaries suggested by ownership or responsibility, business rules already confirmed by domain experts, open questions that remain unresolved, and terms that the team rejected because they created ambiguity.

This distinction is important because an LLM tends to complete a missing structure with a statistically plausible structure. If the artifact contains two events but does not explain their relationship, the model may invent a connection between them. If a term is ambiguous, the model might silently normalize it into the most common interpretation found in its training data.

Structured context reduces that freedom. Structured context does not guarantee correctness, but it makes explicit that parts of the domain are known, which distinctions must be preserved, and where the model should stop rather than infer.

Extracting the structure already present

Consider a small fragment produced during exploration of the BrewUp ordering domain:

- Order submitted
- Payment authorized
- Order stored
- Stock reserved
- Confirmation email sent
- Order confirmed

As a raw list, this material is still ambiguous. It mixes business facts, technical actions, and downstream reactions. It also leaves several questions unanswered:

- Does payment authorization imply that the order has been accepted?
- Is stock reserved inside the ordering capability or by the warehouse?
- Is storing the order a business fact or an implementation detail?
- Does confirmation occur before or after stock reservation?
- Is the email itself meaningful to the domain, or only a reaction to confirmation?

The first useful artifact for AI collaboration should preserve this uncertainty rather than hide it. The artifact below is also in the book's repository, as `docs/ch04/brewup-ordering-exploration.md`:

```
# BrewUp Ordering Exploration

## Scope

This artifact describes the early exploration of the BrewUp sales-order flow.
It is not yet a final domain model.

## Candidate events
```

```
- Order submitted
- Payment authorized
- Order stored
- Stock reserved
- Confirmation email sent
- Order confirmed

## Known distinctions

- Payment authorization does not automatically imply order confirmation.
- Ordering and warehouse responsibilities must remain distinguishable.
- UI actions and persistence details are not automatically domain events.

## Open questions

- Which conditions must be satisfied before the order can be confirmed?
- Who owns a stock reservation?
- Is confirmation a business commitment to the customer?
- Which items in the candidate list are technical side effects?
```

This artifact gives the model a bounded object to examine. The candidate events remain visible, but so do the distinctions and unanswered questions. The model can now classify, compare, and challenge the material without being encouraged to treat every note as an equally valid business fact. This transformation is the first practical transformation from domain exploration to AI collaboration. Visual material becomes explicit context before it becomes a modeling prompt.

From discovery artifacts to reusable context

The translated artifact matters because it can survive beyond the workshop. A wall of sticky notes is temporary. Its meaning depends not only on what remains visible, but also on the conversations, disagreements, and decisions that occurred while the team built the wall. Once the session ends, much of that surrounding context begins to fade.

A structured artifact gives the team a way to preserve the current state of discovery. At this stage, its purpose is deliberately limited: the scope being explored, the candidate events and recurring concepts, the meanings the team has already clarified, the distinctions that must remain visible, the questions that are still unresolved, and where the information came from – a workshop, a scenario, or a conversation with a domain expert.

You keep this material in Markdown files. The choice is practical: Markdown is plain text, so the artifact can be read by a person, diffed in version control, and pasted into a model's context without conversion. These artifacts are working artifacts rather than documentation, and their value is continuity – a later conversation can begin from the same domain material instead of reconstructing it from memory.

Without a reusable artifact, each interaction begins with a partial retelling of the domain. Terms change, unresolved questions disappear, and assumptions that were previously rejected might return unnoticed.

With the artifact in place, the team can reopen the exploration from a shared starting point. The model receives the events, definitions, and uncertainties that the team has chosen to preserve, while domain experts retain the authority to revise them.

Some projects also maintain technical instruction files that describe coding conventions or implementation practices. Those files serve a different purpose. They constrain how a solution is implemented, while the artifacts discussed here preserve what the domain currently means. For now, your concern is only the domain material.

These files are the first layer of a broader interaction architecture. They keep the results of discovery from disappearing between one interaction and the next. Defining a complete rulebook, or prescribing how the model should reason, belongs to a later layer – and to the next chapter.

This transformation gives you the first operational outcome of domain exploration. Visual knowledge becomes reusable context before it becomes a reasoning constraint. However, preserving every workshop note is not enough. The translated artifact might still contain UI actions, technical reactions, duplicate expressions, and events with very different modeling significance.

Once visual knowledge has been made explicit, the next task is selection:

Which candidate events are important enough to become structural anchors of the emerging model?

Identifying core events and refining through feedback

Once discovery artifacts have been translated into structured context, the next challenge is no longer broad exploration but selection. An EventStorming workshop may produce dozens of candidate events, but not all of them carry the same modeling weight. Some express meaningful changes in the business. Others describe a screen transition, a persistence operation, an infrastructure reaction, or a local implementation detail.

If you feed them all into an LLM without distinguishing between them, the model has no reliable reason to treat one as more important than another. The model may organize them into a coherent sequence, but that sequence will preserve, and sometimes amplify, the confusion already present in the source material. The result is often a noisy event landscape: plausible, well-written, and conceptually weak.

The first collaboration step with AI is therefore *distillation*. The model can help you group similar formulations, expose missing transitions, challenge weak names, and separate different levels of meaning. However, its first result should not be treated as a model. Treat it as a draft to challenge.

Candidate events and core events

The useful distinction at this stage is between **candidate events** and **core events.** Candidate events include everything that emerges during exploration: facts mentioned by stakeholders, actions that appear to mark a step in a process, technical reactions observed in the existing system, terms that different participants use for apparently similar transitions, and events proposed to explain gaps in a story.

Candidate events are deliberately inclusive. Their purpose is to preserve the breadth of discovery before the team decides what matters.

Core events are narrower. A core event represents a business fact that carries meaningful modeling weight: it may record an important change in domain state, establish a business commitment, trigger a downstream decision or policy, mark the result of a significant business rule, reveal a boundary between responsibilities, or provide an anchor around which commands, policies, aggregates, or bounded contexts might begin to organize.

This distinction does not imply that every core event will eventually appear in the implementation. At this stage, the team identifies events that are important enough to support further modeling.

Consider the following mixed set from an early exploration of the BrewUp sales order flow:

- Order page opened
- Order submitted
- Payment authorized
- Order stored
- Confirmation email sent
- Stock reserved
- Order confirmed

The list captures real observations made during discovery, but it mixes several levels of meaning. `OrderPageOpened` describes a user interface interaction, `OrderStored` a persistence concern, and `ConfirmationEmailSent` is probably a downstream technical reaction. `OrderSubmitted`, `PaymentAuthorized`, `StockReserved`, and `OrderConfirmed` may represent business facts, although their exact meaning and ownership still require validation.

If you ask the model to reason over the list without making these distinctions explicit, it might turn every item into a domain event and produce a polished but misleading model. This distinction is the point where the architect must take control of the refinement loop.

The relevant question is not:

Did the model generate a convincing event list?

The questions that matter are:

- Which events would domain experts recognize as meaningful business facts?
- Which items merely describe how the current system implements the process?
- Which events are duplicates expressed from a different perspective?
- Which names hide multiple business transitions?
- Which conditions separate one event from the next?
- Which events belong to different business responsibilities?
- Which critical transitions are still missing?

The initial list is a hypothesis, and these are the questions that test it.

What the model contributes

At this stage, AI is useful in a specific and limited way: it acts as a fast pattern synthesizer. Given a rough event list, a short domain description, and explicit classification criteria, it can group overlapping formulations, detect likely synonyms, identify events expressed at inconsistent levels of abstraction, and suggest that an item is a UI action or a technical side effect. It can also surface suspicious gaps in a causal chain, propose alternative ways of segmenting the flow, and highlight transitions that require further domain knowledge.

Suppose the list contains `OrderSubmitted` followed immediately by `OrderConfirmed`. The model might notice that the transition appears under-specified and ask whether payment authorization, credit validation, or stock availability must occur between them. That question can be valuable.

The answer proposed by the model is not domain knowledge. A statistically common e-commerce flow might be plausible while still being wrong for BrewUp.

Similarly, if the list contains `ConfirmationEmailSent`, the model might classify it as a downstream reaction rather than a core business event. This classification is a useful hypothesis, but the team must still determine whether sending that communication has a specific contractual or regulatory meaning in the domain.

The model can accelerate comparison and reformulation. Business truth still requires domain validation.

Feedback as an architectural input

Without AI, the team might spend significant time rewriting event lists, comparing terminology, regrouping concepts, and preparing alternative interpretations for the next workshop. A model can perform much of that provisional work between sessions: keeping candidate interpretations visible, generating alternative classifications, applying explicit review criteria consistently, surfacing missing transitions, comparing revised artifacts with previous versions, and preparing focused questions for domain experts.

Meaningful refinement begins when feedback enters the loop. Without feedback, the model might do little more than improve surface coherence. With feedback, it begins to operate inside a progressively sharper semantic boundary.

Each correction removes an invalid interpretation. Each missing rule supplied by a domain expert closes a knowledge gap. Each explicit separation between responsibilities prevents the model from flattening the domain into a single generic process.

Three forms of feedback are particularly useful. They are easier to see against something concrete. In this chapter, you ask a model to classify the BrewUp workshop items; three lines of what it returned are enough to work with here:

```
Order submitted — uncertain. It could mean the customer clicked a submit control,
or it could mean the business has accepted an order submission as a meaningful
fact.

Payment authorized — business event. Records a meaningful business fact:
authorization for payment has been obtained.

Confirmation email sent — technical side effect. Assumes email delivery itself has
no special domain significance.
```

Each classification is plausible given the material the model received, but each requires a different kind of correction: removing an unsupported assumption, supplying missing domain knowledge, or preserving an architectural boundary.

Corrective feedback

Corrective feedback removes assumptions or classifications that do not belong in the model. The line above assumes that email delivery has no special domain significance. It is a candidate because the team, not the model, decides whether that communication carries a contractual or regulatory meaning. In a BrewUp review, it sounds like this:

- This classification is a UI action, not a domain event.
- Storing the order is an implementation detail.
- This event does not exist in your domain.
- The model has introduced a business rule that the team never stated.
- These two terms are not synonyms.

Corrective feedback is an important defense against false completeness. Corrective feedback prevents a plausible output from becoming accepted simply because it looks consistent.

Additive feedback

Additive feedback introduces domain knowledge that the model could not infer reliably from the discovery material. The uncertain line is exactly that case: only a domain expert can say whether Order submitted marks a business fact or a selection on a screen. In BrewUp, it takes the form of statements such as:

- Payment authorization does not mean that the order has been accepted.
- The order cannot be confirmed until stock has been reserved.
- You are missing the event raised when stock reservation fails.
- A customer might submit an order even when payment authorization is deferred.
- Confirmation represents a commitment by BrewUp to fulfill the order.

Additive feedback transforms a generic flow into one grounded in the actual domain.

Structural feedback

Structural feedback preserves architectural distinctions within the material. Payment authorized came back as a plain business event; structural feedback is what records that the fact is reported by an external provider and does not belong to BrewUp's ordering responsibility. It reaches the model as instructions such as:

- Separate sales order events from warehouse events.
- Do not merge stock management into the ordering responsibility.
- Distinguish decisions from reactions.

- Keep events owned by BrewUp separate from facts reported by an external payment provider.
- Do not assume that events in causal order belong to the same aggregate or bounded context.

Structural feedback is essential because an LLM naturally tends to produce one coherent narrative. In domain modeling, however, apparent coherence can hide boundaries that matter. A causal sequence may cross several responsibilities without belonging to a single model.

A practical refinement loop

The three forms of feedback can be organized into a repeatable process:

1. *Generate* an initial classification from the current discovery material.
2. *Critique* the result against explicit domain and architectural criteria.
3. *Refine* it by supplying corrections, missing knowledge, and structural distinctions.
4. *Validate* the revised result against workshop artifacts, scenarios, and domain experts.
5. *Anchor* the stabilized events as part of the working context for later interactions.

The process becomes clearer when applied to the BrewUp example.

Step 1: classify the discovery material

The first prompt asks the model to separate different kinds of signals. It does not yet ask it to create a domain model.

```
Context:

We are exploring the BrewUp sales-order flow.

Candidate items collected during the workshop:

- Order page opened
- Order submitted
- Payment authorized
- Order stored
- Confirmation email sent
- Stock reserved
- Order confirmed

Task:

Classify each item as one of the following:
```

```
- business event
- UI action
- technical side effect
- uncertain

For each classification:

1. Provide a short explanation.
2. State any assumption required by the classification.
3. Use "uncertain" when the available domain information is insufficient.
4. Do not introduce new events.
```

The model returned a classification for each item, together with a short explanation and the assumption the classification required. Abridged:

```
Order page opened — UI action. An interaction with the user interface, not a
business-relevant change in the order.

Order submitted — uncertain. It could mean the customer clicked a submit control,
or it could mean the business has accepted an order submission as a meaningful
fact. More domain information is needed.

Payment authorized — business event. Records a meaningful business fact:
authorization for payment has been obtained. Assumes authorization affects whether
the order can proceed.

Order stored — technical side effect. Persistence of data rather than a business-
domain occurrence.

Confirmation email sent — technical side effect. A communication mechanism
triggered by the process rather than the core business state change. Assumes email
delivery itself has no special domain significance.

Stock reserved — business event. A business-relevant change in inventory
availability associated with the order. Assumes reservation changes the stock
commitment.

Order confirmed — business event. Assumes "confirmed" has an explicit domain
meaning, such as the business committing to fulfill the order.
```

```
The main ambiguity is "Order submitted": its wording does not establish whether it
describes the user's interface action or a durable business fact.
```

The `uncertain` category is important. Without it, the model is forced to classify every item when the context does not support a reliable decision.

The result provides an initial object for review. The team can now reject incorrect classifications, clarify assumptions, and identify where domain knowledge is still missing. Two of the confident classifications warrant closer attention. `PaymentAuthorized` is labeled a business event with no mention that the fact is reported by an external provider, and `StockReserved` is labeled a business event with no owner, although reservation belongs to the warehouse. Neither answer is wrong based on the evidence the model was given. Both flatten a distinction the team has not yet supplied, and that is what the next prompt has to correct. The output reads as finished work. That is the danger at every stage of this loop, not only at the end: a rough draft invites scrutiny, a fluent one invites trust.

Step 2: inject domain and structural feedback

Suppose the first result confirms that the list contains UI and technical concerns but still treats the remaining business events as one uninterrupted process. The next prompt reintroduces the result together with sharper constraints:

```
Refine the previous result using the following feedback:

- Keep only facts that domain experts would recognize as meaningful business
events.
- Exclude UI actions, persistence operations, and downstream technical
notifications.
- Payment authorization does not imply that the order is confirmed.
- Stock reservation belongs to the warehouse responsibility.
- Preserve the distinction between
BrewUp events and facts reported by an external payment provider.
- Causal order does not imply that all events belong to the same aggregate or
bounded context.

Tasks:

1. Return the revised set of core events in causal order.
2. Indicate the responsibility associated with each event.
3. Identify any missing business transition between order submission and order
```

```
confirmation.
4. Mark unresolved points as questions instead of inventing answers.
```

The model returned four core events with their responsibilities:

- `OrderSubmitted` – Sales ordering
- `PaymentAuthorized` – external payment provider
- `StockReserved` – warehouse
- `OrderConfirmed` – Sales ordering

It excluded Order page opened as a UI action, Order stored as a persistence operation, and confirmation email sent as a downstream technical notification. Where the material did not support an answer, it returned a question instead of an event name:

```
What business transition within BrewUp determines that a submitted order may
become confirmed, and which external or business facts must be satisfied for that
transition?
```

This list is smaller and more precise than the original material, but it is still not final. It also reveals that the flow crosses responsibilities. This separation is more useful than a single undifferentiated sequence because it creates concrete questions for the next modeling conversation:

- What causes the warehouse to reserve stock?
- What happens if reservation fails?
- Does Sales wait for both payment authorization and stock reservation?
- Which component decides that the order can be confirmed?
- Are the facts exchanged between these responsibilities domain events, integration events, or both?

The model has helped make these questions visible; answering them is still the team's work.

Step 3: validate the revised set

The revised event set must now be checked against the original exploration material: the EventStorming wall or its structured translation, the Domain Storytelling scenarios, examples, and counterexamples supplied by domain experts, the existing terminology used by the business, the known exceptional and failure paths, and the explicit architectural decisions the team has already made.

The model can assist by comparing artifacts and highlighting inconsistencies, but validation remains external to the model. A clean event list is evidence that the available material has been organized, and nothing more.

Refinement as knowledge crunching

The knowledge crunching process reflects a foundational Domain-Driven Design idea: modeling is a form of **knowledge crunching**. The team begins with fragmented information, competing terminology, partial examples, and different interpretations. The team tries alternative organizing ideas, rejects weak abstractions, and gradually distills what matters into a useful model.

AI changes the tempo of this process. The model can reformulate material quickly, preserve provisional alternatives, compare classifications, and expose inconsistencies between one iteration and the next. Architects and domain experts contribute judgment, business knowledge, and the authority to accept or reject a proposed interpretation.

The model accelerates comparison, reformulation, and the preparation of alternatives, while artifacts and domain experts provide meaning, judgment, and validation. Its contribution is to make those questions concrete enough for the team to examine.

From iteration to semantic convergence

Semantic convergence is a practical sign that the loop is working: the events, their meanings, and their responsibilities stop shifting from one iteration to the next. The phrase describes a state the material reaches, not a technique the team applies. After several iterations, the material begins to stabilize:

- The model stops reintroducing items already rejected as technical details.
- Domain experts stop objecting to the event names.
- The same transition is no longer described through several interchangeable terms.
- Missing steps become less frequent.
- The team increasingly replaces assumptions with explicit rules or open questions.
- Disagreement moves from basic vocabulary toward finer boundary and responsibility decisions.

Semantic convergence means that the team has reached enough stability to support the next modeling step. Stability is more valuable than apparent exhaustiveness. A stable set of core events can be tested, extended, and used as shared context. A long set of plausible events that changes at every interaction cannot. This is why iterative refinement matters more than first-pass brilliance.

In later AI engineering workflows, you will encounter the same posture under different names, including the informal **Ralph Wiggum loop** described by Huntley (2025): a deliberately crude technique that reruns the same prompt until a stop condition is met, letting the files and their history rather than the conversation carry the state between runs. The underlying principle is straightforward: do not design the process around the expectation that the first result will be correct. Design it so that repeated execution, critique, and correction can move the artifact toward convergence.

In domain modeling, however, repetition alone is insufficient. Running the model again without introducing better feedback might simply produce another plausible variation. The quality of the loop depends on the quality of the signals returned to it.

The object being refined in this chapter is the domain material itself: candidate events, meanings, responsibilities, causal relationships, and unresolved questions. The feedback loop helps the team make that material more explicit and stable. The feedback loop does not yet define how the model must reason in future interactions.

In *Chapter 5*, you will apply the same iterative posture to a different artifact. Instead of refining the domain material, you will refine the rules that govern how the model is allowed to reason over it.

What AI changes, and what it does not

AI replaces none of this – not EventStorming, not Domain Storytelling, not the domain expert, and not the architect's judgment. What changes is the speed and density of the refinement cycle that follows discovery. This acceleration makes AI an effective accelerator for the work that happens after the wall is full of sticky notes. The risks, however, remain real.

A clean event list can create a false sense of completeness. A statistically common ordering flow can look convincing while missing the exception that defines the actual business. Generic retail or CRUD concepts can enter the model and replace more specific domain language.

More importantly, linguistic fluency can be mistaken for modeling quality. These risks cannot be eliminated through better prompting alone. They are controlled by disciplined feedback, explicit uncertainty, and validation that remains outside the model.

Identifying core events should therefore be treated as a refinement practice, not as an extraction task. The model can help produce candidates, critique them, rename them, and reorganize them. However, only feedback grounded in the domain turns those candidates into useful design material. Refinement does not make the AI understand the domain. Refinement makes the domain more explicit.

The role of language in domain clarity

Once the event set begins to converge, the next challenge is preserving the meaning attached to it. Discovery and refinement may have given you a smaller set of meaningful events, but those events are not yet stable if their names continue to shift. The same business transition might still appear under several near-synonyms, while business language and technical terminology might continue to bleed into each other.

If that instability enters subsequent prompts, the model has no reliable reason to preserve distinctions that the team has worked to uncover. An event can remain structurally present while its meaning slowly changes through synonyms, generic reformulations, or inconsistent terminology. If events are the structural anchors of the emerging model, language is what prevents those anchors from drifting.

Chapter 3 established why disciplined vocabulary matters when collaborating with an LLM. Here, however, we face a more specific problem: the vocabulary produced during discovery is rarely clean.

EventStorming encourages people to explore freely. Participants speak naturally, mix levels of abstraction, invent shorthand, and use different terms for what may, or might not, be the same concept. Domain Storytelling preserves those expressions inside concrete scenarios, where further variations might emerge.

This fluidity is productive during exploration. It allows ideas to surface before the team has decided how they should be organized. However, the same fluidity becomes a liability when the discovery material is turned into a reusable AI context. The question is:

How do you move from the fluid language of exploration to a vocabulary stable enough to reuse?

Naming matters because it preserves validated distinctions. It shapes the boundaries the model can preserve.

Where instability hides after a workshop

Linguistic instability is not always obvious. It often appears as a collection of expressions that sound sufficiently similar to be treated as interchangeable.

Consider an early EventStorming session for BrewUp in which the following events appear:

- Order submitted
- Order received
- Order accepted
- Order confirmed

In casual conversation, participants may use these expressions to describe roughly the same moment. From a modeling perspective, however, they might represent different transitions.

`OrderSubmitted` may describe the customer sending an order request for evaluation. `OrderReceived` might describe BrewUp acknowledging that the request has entered the system. `OrderAccepted` might indicate that the relevant business checks have succeeded. `OrderConfirmed` might indicate that BrewUp has communicated its acceptance and made a commitment to fulfill the order.

These distinctions matter even before the team begins discussing aggregates, commands, policies, or integration mechanisms. If they remain implicit, an LLM will tend to compress the terms into a familiar ordering pattern. The generated flow might look clean and convincing while silently removing transitions that matter to the business.

For example, a policy that should react only after stock reservation may be triggered too early; payment authorization may be treated as order acceptance; a command may be associated with the wrong event; a commitment to the customer may be represented as a technical notification; and separate business responsibilities may appear to belong to one continuous process.

Domain Storytelling can help expose this instability in context. Suppose one story says "*The customer places an order,*" another says "*The buyer submits a request,*" and a third says "*BrewUp receives the order.*"

These variations may be harmless narrative choices, or they might reveal different actors, intentions, and business moments. The story provides enough context for the team to ask which interpretation is correct. The model cannot preserve a distinction that the team has not yet made explicit.

The danger of false coherence

Ambiguous language does not necessarily produce visibly confused output. More often, it produces **false coherence**.

Suppose the model receives:

- `OrderSubmitted` in an EventStorming transcription.
- `OrderAccepted` in a `revised` event list.
- "*The customer places an order*" in a Domain Storytelling scenario.

The available context does not establish whether these expressions describe one transition, several transitions, or different perspectives on the same business fact. The model will not necessarily stop and request clarification. Unless instructed otherwise, it is likely to select one plausible interpretation and silently normalize the others around it.

The result might look smooth and well-structured. The output feels coherent because the prose no longer shows the ambiguity, even though the domain still contains it. The model has resolved that ambiguity on the team's behalf.

This normalization creates a specific risk: the cleaner the output appears, the easier it becomes to assume that the underlying domain has also become clear. Disciplined language is therefore a defense against prematurely believing that the model has understood something the team itself has not clarified.

When terminology remains unstable, ambiguity must stay visible. Instruct the model to flag competing terms, identify possible distinctions, and ask for missing domain knowledge rather than collapse everything into a polished interpretation.

A practical BrewUp vocabulary

Suppose that the BrewUp ordering flow has converged around the following core events:

- Order submitted
- Payment authorized
- Stock reserved
- Order confirmed

A list of names is useful, but it is not enough. Each name still needs a definition that preserves its business meaning. These events are the same four events the model returned in Step 2, and two of them still carry open questions: whether Order submitted marks a business fact or a selection on a screen, and what condition allows BrewUp to move from submitted to confirmed. The vocabulary is where those answers are recorded.

A working vocabulary might look like this. The complete file is also in the book's repository, as `docs/ch04/brewup-ordering-vocabulary.md`:

```
# BrewUp ordering vocabulary

## Order submitted

A domain event raised when the customer sends an order request to BrewUp for
evaluation.

It does not imply that the order has been accepted or confirmed.

## Payment authorized

A fact reported when the payment provider authorizes the requested amount.
```

```
Authorization does not imply that the payment has been completed or that BrewUp
has committed to fulfilling the order.

## Stock reserved

A domain event raised by the warehouse responsibility when the stock required for
the order has been successfully reserved.

It does not imply that the sales order has already been confirmed.

## Order confirmed

A domain event raised when BrewUp commits to fulfilling the order after the
required business conditions have been satisfied.
```

The artifact should also record exclusions:

```
## Vocabulary exclusions

- Do not use "Order received" as a synonym for "Order submitted". Receiving is
BrewUp acknowledging the request; submitting is the customer sending it.
- Do not use "Payment completed" when the domain fact is only authorization.
Authorization reserves the funds; completion moves them.
- Do not use "Inventory allocated" as a synonym for "Stock reserved" unless the
warehouse domain explicitly adopts that term. Two names for one fact hide the
boundary.
- Do not use "Order approved" as a replacement for "Order confirmed". Approval is
an internal check; confirmation is the commitment made to the customer.
- Do not treat payment authorization as sufficient evidence that the order has
been confirmed. Authorization is one input to that decision, not the decision.
```

At this stage, the artifact is still a working vocabulary. It records what the terms mean, which alternatives have been rejected, and which distinctions must remain visible. A complete rulebook would also define the full set of domain invariants, prescribe a reasoning procedure, impose an output schema, and determine where those constraints should be executed. Those additional layers belong to the next chapter.

This artifact also records business decisions about sequence, ownership, conditions, and commitments. The vocabulary becomes a compressed representation of the team's current understanding of the flow.

The distinction between `PaymentAuthorized` and `OrderConfirmed`, for example, prevents the model from collapsing an external provider's decision into BrewUp's business commitment. Similarly, defining `StockReserved` as belonging to the warehouse responsibility preserves a boundary that a simple causal list might otherwise hide.

Injected into later prompts, this vocabulary gives the model a more reliable basis for reasoning than a loose set of event names. It also provides a concrete artifact that domain experts can review and challenge.

Testing and reinforcing the vocabulary

A working vocabulary becomes useful only when its terms survive contact with a concrete domain scenario. Consider the following BrewUp flow:

```
A customer submits an order for 50 bottles of
IPA. The payment provider authorizes the requested amount. The warehouse reserves
the required stock. BrewUp then confirms that it will fulfill the order.
```

The stabilized vocabulary allows the team to describe the flow without introducing new terminology:

- `OrderSubmitted`
- `PaymentAuthorized`
- `StockReserved`
- `OrderConfirmed`

The value of the sequence is that the selected names remain meaningful when placed back in a realistic flow.

The scenario also preserves the distinctions established during refinement. The team does not treat payment authorization as order confirmation. Stock reservation remains associated with the warehouse's responsibility. Order confirmation represents BrewUp's commitment rather than a technical notification.

Now, consider a slightly different scenario:

```
A customer submits an order. The payment provider authorizes the amount, and
BrewUp immediately confirms the order. Stock is reserved afterward.
```

The working vocabulary leaves the validity of this scenario open and exposes a question that the team must answer:

Is successful stock reservation one of the business conditions required before an order can be confirmed?

This is the outcome you need at the end of domain exploration. The artifact makes the unresolved distinction visible and returns it to the team, instead of manufacturing a missing rule or forcing the scenario into a familiar sequence. Beyond normalizing words, a working vocabulary gives the team a stable language in which inconsistencies, missing transitions, and unanswered questions can be discussed.

From exploration to deliberate interaction

This chapter began with a deliberately messy situation. Domain knowledge was distributed across people, workshop conversations, sticky notes, diagrams, and partially aligned terminology. That material was valuable, but it was not yet stable enough to guide repeated interaction with an AI system.

EventStorming made business facts and possible boundaries visible. Domain Storytelling placed those facts inside scenarios that preserve actors, intentions, and causality. Translation carried the meaning of those visual artifacts into a reusable textual form.

From there, refinement separated meaningful business events from UI actions, persistence concerns, and technical reactions. Feedback exposed missing transitions and clarified ownership. Finally, vocabulary stabilization gave the emerging concepts names that could survive across later conversations.

The progression is:

Discover → Refine → Name → Reinforce

Discover makes fragmented knowledge visible. *Refine* separates meaningful business material from noise and exposes what is still missing. *Name* gives stable terms to the distinctions that matter. *Reinforce* carries those terms forward while keeping them open to deliberate revision when new domain knowledge emerges.

The outcome is domain material precise enough to support the next form of collaboration: candidate events have been evaluated, important meaning has been made explicit, and the vocabulary is stable enough to reuse without silently changing at every interaction. That endpoint is where this chapter ends. The next chapter begins with a different question:

Once the language is stable, which rules prevent the model from misusing it?

Summary

In this chapter, you moved from domain exploration to the first operational foundation for AI-assisted Domain-Driven Design. EventStorming helped the team externalize business knowledge as candidate events, questions, reactions, and possible boundaries. Domain Storytelling complemented that event landscape by preserving actors, intentions, work objects, and causal flow.

You then distinguished transcription from translation. Capturing the text written on a workshop wall is not enough. The meaning carried by relationships, grouping, accepted distinctions, unresolved questions, and domain-expert feedback must be preserved in a form that can survive beyond the workshop.

Once the material became explicit, the challenge shifted from discovery to selection. Candidate events preserved the breadth of the exploration, while core events identified the business facts important enough to support further modeling. AI accelerated classification, comparison, and reformulation, but validation remained the responsibility of architects and domain experts.

The refinement process used three forms of feedback:

- *Corrective feedback* removed invalid assumptions, technical details, and incorrect classifications.
- *Additive feedback* introduced business knowledge that the model could not infer reliably.
- *Structural feedback* preserved boundaries, ownership, and architectural distinctions.

Repeated refinement moved the material toward semantic convergence: enough stability in events, meanings, and responsibilities to support the next modeling step. Finally, you examined how language protects that stability. Similar-looking terms might represent different business transitions, while fluent AI output can create false coherence by silently collapsing unresolved ambiguity. A working vocabulary preserves canonical names, current meanings, rejected alternatives, and open questions.

The progression developed in this chapter was:

Discover → Refine → Name → Reinforce

This sequence produces domain material whose meaning is explicit enough to be carried forward. This sequence stops short of a complete design and of any rule about how the model must reason. Domain exploration, the domain expert, and the architect's judgment remain in place. AI accelerates the work of translating, comparing, challenging, and refining the material they produce.

In *Chapter 5*, you will build on that foundation. Stable vocabulary will become the input to a rulebook that constrains how the model reasons inside a bounded context.

References

- Huntley, G. (2025). Ralph Wiggum as a "software engineer." `https://ghuntley.com/ralph/` (accessed September 2026).

5

Defining Rules: Teaching the Model to Think in DDD

In the previous chapter, we used EventStorming and Domain Storytelling to externalize domain knowledge and to stabilize the vocabulary the team will use for the rest of the book. We ended on a working principle: the sequence Discover, Refine, Name, Reinforce produces language stable enough to be trusted across conversations with an AI assistant. Stable vocabulary, however, is not the same as disciplined reasoning. This chapter closes that distance by introducing a rulebook.

A rulebook is a small, explicit set of constraints that govern how the model reasons inside one bounded context. The team writes it once, versions it alongside the code, and reuses it in every session that follows. In practice, it is a file that lives in the repository next to the code: a Markdown instruction file or a short YAML fragment, edited and reviewed like any other source file. Its effectiveness depends on specificity rather than length or cleverness. The rulebook plays the role that bounded contexts play in Domain-Driven Design: it protects meaning, preserves intent, and makes output dependable over time.

Those constraints are necessary because an LLM with a clear glossary in hand can still widen scope, invent aggregates, and blur context boundaries. Such an LLM can also produce fluent output that drifts away from the actual design. This drift follows from how an LLM operates: it fills probabilistic gaps in the direction suggested by the prompt. A rulebook defines in advance how the model should handle those gaps.

You begin with a concrete BrewUp moment, when the team realized that an AI assistant could produce DDD-shaped output and still be silently wrong. From there, the chapter shows how a small number of well-placed rules can prevent that kind of failure, and how those rules can be packaged into reusable containers, namely agents, skills, and MCP servers, so that they keep working session after session.

In this chapter, you will learn:

- Why generic DDD instructions, such as *Apply DDD consistently,* fail to constrain the model, and what a Payment-specific rulebook with vocabulary, invariants, and forbidden synonyms looks like instead.
- Agents, skills, and MCP servers operate as three different containers for rules. Each container shapes a different concern: who the model is supposed to be, how it performs a specific task, and what trusted project sources it can consult.
- How structured output expectations, such as YAML with named fields for commands and events, turn the model's reasoning into artifacts that can be reviewed, diffed, and reused by you.
- How the team refines the rulebook through critique, where each silent failure of the model becomes the seed of a new rule that catches the same mistake the next time.

You start where the team did, with a Payment context that looked correct on the page.

Technical requirements

The ideas in this chapter do not depend on a specific AI model or vendor. The examples assume an assistant that can read project configuration rather than only the current conversation. To follow along, you need:

- An AI coding assistant that reads a project-level instruction file: `CLAUDE.md`, `AGENTS.md`, or `.github/copilot-instructions.md`, depending on the tool.
- Support for named, task-scoped procedures – skills, custom commands, or the equivalent in your assistant – and access to at least one MCP server, if you want to run the examples in the second half of the chapter rather than read them.
- A text editor capable of working with Markdown and YAML files.

The rulebooks, agent excerpts, and skill definitions in this chapter are deliberately short. They show what a DDD constraint looks like in each container, not how to configure a particular product, so they should transfer to whichever assistant your team has standardized on.

Why rules matter more than prompts

When the BrewUp team first asked an AI assistant to model the Payment context, the output looked plausible. The YAML came back with a Payment aggregate, sensible commands such as Authorize, Capture, and Refund, and past-tense events. Then, a senior engineer pointed out that the model had quietly invented a Charge aggregate because the prompt happened to mention '*charge the customer*'. The model had also produced a refund flow that operated on

authorized payments rather than captured ones. The result was fluent and syntactically correct, yet silently wrong about a domain rule the team had never written down.

That moment is when the team stopped writing prompts and started writing rules. The two look similar on the page, but they answer different questions. A prompt asks the model to do something. A rule constrains how it can reason. The distinction matters most when the model is wrong in a way that reads as right. No rephrasing of the prompt would have caught the invented Charge aggregate, because nobody had ever told the assistant that Charge was forbidden.

These rules do not need to be complex to be effective. A small, well-defined set of constraints is often enough to guide the model toward domain-consistent reasoning. The real challenge is one of selection: finding the minimal set of rules that forces the model to think in a Domain-Driven Design way.

A natural first attempt at such a rulebook is to translate textbook DDD into a Copilot instruction file and call it done. The fragment below is a representative slice of one such early attempt:

```
Domain-Driven Design (DDD) Instructions

General Principles
- Apply Domain-Driven Design patterns consistently in all generated code
- Use the Ubiquitous Language of the domain
- Maintain a strict separation between architectural layers

Tactical Design Patterns
- Entities: Must have a unique and persistent identity
- Value Objects: Must be immutable
- Aggregates: Define the Aggregate Root and enforce invariants
- Repositories: Define interfaces in the Domain layer
```

This kind of checklist may look reassuring, but it provides little practical constraint. Each line restates something the assistant has already absorbed from training data. '*Apply DDD consistently*' is not a constraint; it is a benediction. *Entities have a unique identity* is a definition the model can recite without help. The file does not narrow what the assistant can produce in your project, because it does not say anything about your project. The model behaves the same way whether the file is loaded or not. Length is not the problem. A checklist three times longer would constrain the model just as little, because what is missing is specificity: the vocabulary, the invariants, and the boundaries of one bounded context.

A more useful rulebook starts with the specific shape of one bounded context. Below is the same idea applied to BrewUp Payment, where the team encoded the rules they had learned through costly production bugs:

```
context: Payment
vocabulary: Payment, Authorization, Capture, Refund, Settlement
forbidden_synonyms: Charge, Transaction, OrderPayment

invariants:
- Currency at authorization must equal the order currency. No conversion at
payment time.
- An authorization expires 30 minutes after issuance; capture must occur within
that window.
- Refunds are issued only against captured payments, never against authorizations.
- Partial refunds are allowed; partial captures are not.

events:
- Use past-tense names rooted in the Payment vocabulary, e.g. PaymentAuthorized,
PaymentCaptured, PaymentRefunded.
```

Every line in this fragment narrows what the assistant can produce. The file cannot invent a Charge aggregate, because Charge went on the forbidden list the day the model invented it. The file cannot propose currency conversion at payment time, because the invariant forbids it. The file cannot emit a refund against a still-authorized payment, because the refund flow that opened this chapter is now written down as a rule. The file is shorter, but its constraints are more specific. That is the difference between a generic checklist and a domain rulebook.

Where rules come from

A rulebook this specific looks like something that needs a workshop to produce. The team built most of BrewUp's rulebook by recording problems it had already encountered. Most of its rules came from three sources.

The first source is the problems the team has already encountered. `forbidden_synonyms`: Charge, Transaction, `OrderPayment` is a list of names that had already turned up where they did not belong, rather than a guess about what a model might invent. Charge is there because of the modeling session that opens this chapter. The invariants have the same provenance: each one is a rule someone learned through costly experience and wrote down so that nobody has to learn it twice. The list is also the answer to a fair objection – that banning synonyms means proving a negative, and you can never finish. Nobody is enumerating every wrong name

in advance; the list only records the ones that have already cost something, which makes it finite and shrinking.

The second is the glossary the team already maintains. The last chapter ended with a vocabulary exclusions block – *Do not use Order received as a synonym for Order submitted*; do *not use Payment completed when the domain fact is only authorization* – produced by a workshop, not by trial and error. That block is already a forbidden-synonyms list; it needs moving into the rulebook, not inventing. Wherever a glossary takes the trouble to separate two terms that a fluent writer would treat as interchangeable, a rule is waiting to be written.

The third source is the questions the workshops could not close. *Chapter 4* left several open – who owns a Stock reservation, which conditions must be satisfied before an Order can be confirmed – and every one of them is a gap the model will otherwise fill with something plausible. The rule that comes out of an open question is rarely a fact. It is usually an instruction to stop: if ownership is unclear, add an open question instead of inventing a rule.

After that, rules accumulate by running the loop. Give the model the current rulebook, read what comes back with domain judgment, and promote each correction into a rule that catches the same mistake next time. The last section of this chapter follows that loop through two iterations of Order Management. It is where most of a mature rulebook comes from: a team who has run it a few times stops asking which rules to write and starts recognizing them.

Knowing where rules come from is half the job. The other half is knowing what kind of rule you are writing. The Payment fragment does three different things. It says what the model may call things, how those things are allowed to behave, and what shape the answer has to take.

The three layers of a DDD rulebook

A rulebook is built from three layers, and they stack: **language**, **semantics**, and **output structure**. Each one is only as strong as the one beneath it. The first layer is language. The model must use the Ubiquitous Language of the bounded context consistently and avoid drifting into generic technical vocabulary or invented synonyms. The language layer looks like a style rule and works like a guardrail. *Chapters 2 and 3* explained the mechanism: repeated terms reinforce each other through attention, and each reinforcement narrows the probability distribution the model samples from. Stable vocabulary therefore does real work – it shrinks the space the model can wander into. In the Payment rulebook, that layer is a single line: `forbidden_synonyms`: Charge, Transaction, `OrderPayment`.

The second layer consists of semantic rules. These tell the model how domain concepts are allowed to behave. An event is something that has already happened, an aggregate protects its invariants through its root, and repositories belong to the domain contract even if their implementation lives elsewhere. Without them, the model still produces fluent output, but it

flattens or distorts the domain. Payment carries one of these: refunds are issued only against captured payments, never against authorizations. That rule says nothing about naming. It constrains what the system is allowed to do.

The third layer is output structure. Asking the model to express its result in YAML or JSON forces it to organize the answer around meaningful categories instead of free-form prose. Commands, events, aggregates, policies, and boundaries become visible, reviewable, and reusable. The events block in the Payment rulebook is the smallest possible version of this: it does not list the events; it dictates the form they must take.

The chapter presents the Payment rulebook whole, as motivation. A second one is worth reading line by line, with the three layers in hand, to see which rule does which job. Here is the minimal rulebook for BrewUp Order Management, the context you will keep working on for the rest of the chapter:

```
context: BrewUp Order Management
rules:
  - Use only the defined domain vocabulary: Order, Payment, Stock
  - An event must represent something that has already happened
  - Each successful state-changing command should produce at least one domain
event
  - Do not introduce new aggregates
output:
  format: yaml
  fields:
    - command
    - event
    - aggregate
```

Read against the three layers, the file divides cleanly. The vocabulary line is the language layer: three terms, and nothing else is admissible. The rules about events, commands, and aggregates are the semantic layer, because they govern how domain concepts must behave rather than what they are called. The output block is the structural layer, and it exists so that two runs of the same prompt can be compared line by line.

Even this small set of constraints changes the interaction. The model is no longer free to improvise a generic solution: it has to operate inside a defined linguistic and structural boundary. The rulebook constrains where the model may reason instead of asking it to reason better. The file is just as interesting for what it leaves out. The file lists Order, Payment, and Stock as the vocabulary of this context, and never says which of the three Order Management

actually owns. That silence is deliberate, and we will come back to it at the end of the chapter, once we have a durable place to store the rules that emerge from critique.

The three layers tell us *what kind of constraint you are defining*: language, semantics, or output structure. They do not yet tell you where that constraint should live or how it should be reused. That is the next problem. Agents, skills, and MCP servers are not a second rulebook model, but three different mechanisms for making those constraints operational. Some rules belong to the model's persistent posture, some to a specific task, and some depend on trusted knowledge that must be retrieved rather than inferred.

Agents, skills, and MCP servers: three containers for DDD guardrails

Up to this point, we have treated rules as something embedded directly in prompts. That approach becomes difficult to maintain as the workflow grows. The prompt below is the last one the BrewUp team wrote before abandoning this approach:

```
You are a DDD expert. Use only our ubiquitous language (Order, Payment, Stock;
never Charge or Transaction), read the workshop transcript below, extract the
domain events, keep only real business facts, drop UI actions and technical side
effects, do not invent aggregates, check docs/glossary.md before naming anything,
return YAML with command, event and aggregate fields, and mark anything you cannot
confirm as an open question.
```

The prompt is not technically wrong. The problem is that it combines a role, a vocabulary, a procedure, an output shape, and a piece of project knowledge in a single instruction. When the answer comes back subtly wrong, there is no way to tell which of the five concerns failed. It is also not reusable. The next modeling task needs the same vocabulary rules and a different procedure, so the prompt gets copied and edited, and two weeks later the team is maintaining six near-identical prompts that quietly disagree with each other.

A clearer approach is to split those concerns into three containers with different lifecycles. An agent carries the posture the model keeps for an entire session. A skill is narrower: one procedure, invoked when it is needed and dormant otherwise. MCP servers, built on the Model Context Protocol, do something different again: they expose the project sources the model may consult instead of guessing. As a practical rule, a constraint belongs to the agent if it should hold regardless of the task. A constraint belongs to a skill if it applies only while one task is running. A constraint belongs behind MCP if it depends on a fact outside the conversation.

The surface differs between assistants, such as Claude, Codex, or Copilot, but the design intent does not: constrain probabilistic generation so that it behaves more like disciplined modeling

and less like generic completion. What follows is not a complete reference for any of these containers, but representative excerpts showing what a DDD-aware constraint looks like in each.

Agents

Agents operate at the highest level of control. They define the identity, role, and perspective of the model within a given interaction. An agent persists beyond a single prompt: it frames the model's behavior for a whole session rather than one exchange. In a DDD context, an agent can embody the role of a domain architect by enforcing the Ubiquitous Language, respecting bounded contexts, and maintaining architectural discipline. Agents answer a fundamental question: who is the model supposed to be while reasoning?

Figure 5.1 illustrates this idea by framing the model as a DDD architect, not as a generic assistant. That framing is important because it stabilizes perspective before it stabilizes output.



Figure 5.1 – An agent represents a DDD architect

Concretely, this kind of framing usually lives in a project-level instruction file: CLAUDE.md, AGENTS.md, or .github/copilot-instructions.md, depending on the assistant. The container

changes; the design intent does not. A short excerpt is often enough to anchor the assistant's posture for an entire session:

```
## DDD Guardrails

- Act as a domain-focused software architect, not as a generic code generator.
- Use the project's ubiquitous language exactly as defined. Do not introduce
synonyms unless explicitly requested.
- Treat bounded contexts as strict semantic boundaries. Do not merge concepts
across contexts.
- Never invent aggregates, invariants, or business rules that are not supported by
the provided context.
- If domain constraints are missing, ask for clarification instead of filling gaps
with plausible assumptions.
- Prefer domain behaviors, business rules, and events over CRUD-oriented
explanations.
```

Six lines do the work because each one names a move the model is otherwise likely to make. The model is no longer merely encouraged to sound like a domain architect. It is constrained against behaviors such as inventing aggregates, carrying synonyms across bounded contexts, or filling missing information with plausible business rules.

Viewed through the three layers, this excerpt relocates the rulebook rather than replacing it. The lines on ubiquitous language and bounded contexts are the language layer; the lines on inventing aggregates and asking instead of assuming are the semantic layer. The structural layer is missing, and its absence is the point: an agent cannot fix the shape of an answer, because it does not yet know which question is coming. The structural layer belongs to a different container, and to a later step: posture is settled first, and output discipline follows from it.

Skills

Where an agent sets posture, a **skill** performs one job. It is a named procedure the model runs when the conditions for it are met and leaves dormant when it is not needed. In DDD workflows, this is where we encode procedures such as extracting domain events from a scenario, checking aggregate boundaries, reviewing terminology, or separating business facts from technical side effects. An assistant that has been told to think like a domain architect still has to be told how this team extracts events. That gap is what a skill fills.

Figure 5.2 captures this role. A skill acts as a reusable container for rules, instructions, and expected output patterns. The skill says what to do and, more usefully, how.

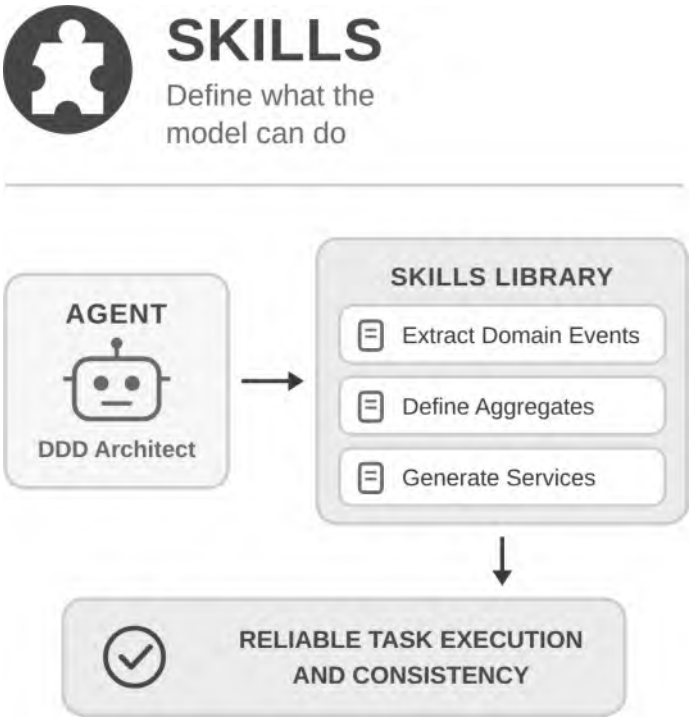


Figure 5.2 – A skill as a container for specific, reusable capabilities

A typical skill is short, named, and scoped to a single modeling task. The following example describes how the team extracts domain events from a workshop transcript, with explicit do/do not steps and a fixed output shape:

```
---
name: extract-domain-events
description: Extract core domain events from a business scenario using DDD rules
---

## Instructions

1. Read the scenario and identify candidate events.
2. Keep only events that represent meaningful business facts.
3. Remove UI actions, technical side effects, and infrastructure reactions.
4. Use past-tense event names only.
5. Do not introduce new aggregates that are not present in the provided context.
```

```
6. If two events appear to be synonyms, normalize them using the canonical
vocabulary.

## Output format

Return YAML with these fields only:
- event
- aggregate
- reason
- open_questions
```

The value here is repeatability. A general instruction such as 'think in DDD' can produce a different answer each time it is invoked. A skill produces answers in the same output schema every time, so the team can compare one run with the last. Repeatability is what turns a one-off prompt into a modeling practice the team can review, version, and refine over months instead of seconds.

Compared with an agent, the division of responsibility is clear. The agent keeps the model honest about the domain, whatever it is asked; the skill ensures that the same procedure and output structure are applied every time. Drop the agent, and the skill drifts in vocabulary. Drop the skill, and the agent still brings good judgment to every run, but in a different shape each time.

MCP servers

Agents and skills both shape how the model reasons. **MCP servers** change the material it reasons over. The protocol, introduced by Anthropic in late 2024, is an open standard for connecting an assistant to sources outside its own conversation through a single uniform interface. An MCP server is one such source, exposed under rules the team writes: tools the assistant may call, documents it may read, project artifacts it may rely on. In DDD work, this matters as soon as the interaction depends on knowledge that should not be guessed from general training data: canonical glossaries, EventStorming exports, context maps, architecture decision records. By anchoring the conversation in real project material, MCP servers turn "*what does the glossary say?*" from a question the model answers by inference into one it answers by lookup.

Figure 5.3 represents this extension. MCP allows the model to reach beyond its immediate prompt and interact with external systems and validated knowledge sources. MCP does not remove the need for rules, but it grounds those rules in actual project material.

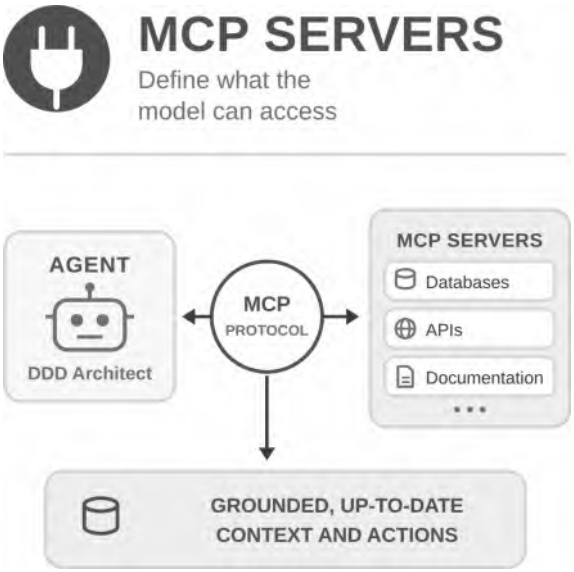


Figure 5.3 – MCP allows communication with external systems

An MCP definition typically declares a small number of tools and the conditions under which the assistant should consult them. MCP is still a moving specification, so read the excerpt below for its design intent rather than as a current schema; the authoritative version is the specification itself. It exposes three sources, namely a glossary, a context map, and validated workshop notes, and constrains how the model uses them:

```
tools:
  - name: glossary.lookup
    description: Returns canonical domain terms and forbidden synonyms
  - name: contextmap.get
    description: Returns bounded contexts and allowed relationships
  - name: eventstorming.search
    description: Searches validated workshop events and notes

usage_rules:
  - Always check glossary.lookup before normalizing domain terminology.
  - Use contextmap.get before suggesting interactions across bounded contexts.
```

```
  - If an event is not found in the validated workshop material, mark it as
proposed, not confirmed.
```

What changes here is the model's relationship with uncertainty. Without MCP, an assistant either commits to a plausible answer or refuses outright. With MCP, it can verify a term against the glossary, check whether a candidate event appears in validated workshop notes, and mark anything unconfirmed as proposed rather than confirmed. Grounding can turn an unsupported inference from a silent failure into a visible signal that the modeling team can review.

MCP still does not decide. A glossary lookup returns the canonical term; it does not decide whether that term belongs in this bounded context. Grounding reduces the need to guess and leaves judgment with the posture enforced by the agent and the review performed by the team.

Three containers, three different jobs: agents stabilize who the model is; skills stabilize how it works; MCP stabilizes what it knows. None of them alone is enough; together they replace the fragile single prompt with a structured environment in which the model can reason inside the boundaries the team chose.

In practice, the borders blur, and a rule can plausibly belong in two places at once. *Never invent aggregates* is a posture, so it belongs in the agent, and yet the event-extraction skill wants it too, because that is where the temptation is strongest. The rule of thumb BrewUp settled on is to write a rule once, in the broadest container that can hold it, and let a narrower container restate it only when it genuinely needs to be stricter there. A skill that repeats the agent verbatim is duplication waiting to drift; a skill that tightens the agent for one task is doing exactly what a skill is for. The three layers remain the conceptual frame of the rulebook: agents, skills, and MCP servers simply provide different ways to apply its language, semantic, and output-structure constraints.

Iterating toward a consistent domain vocabulary

At this point, the rulebook becomes more than an abstract set of instructions. It has become something the team can test against real modeling output. The practical workflow is simple: run the model with the current rules, inspect what it gets right and wrong, then turn the correction into a new rule. Consider a second iteration of the BrewUp Order Management context. Given the rulebook above and nothing else, GPT-5.6 Sol returned this on September 10, 2026, in a fresh conversation with no system prompt and reasoning effort set to high:

```
- command: PlaceOrder
  event: OrderPlaced
  aggregate: Order
- command: CompletePayment
```

```
    event: PaymentCompleted
    aggregate: Payment
  - command: ReserveStock
    event: StockReserved
    aggregate: Stock
  - command: ConfirmOrder
    event: OrderConfirmed
    aggregate: Order
```

This output is already better than a generic answer. The output uses domain language and produces events in past tense, while keeping the result in a reviewable structure. However, it is not finished yet. Plausible output alone is not enough. What the team gains here is a result concrete enough to inspect, and the sharper domain questions that follow from inspecting it.

Read against the context boundary, the draft has the problem the rulebook was supposed to prevent. `PaymentCompleted` is attributed to a Payment aggregate and `StockReserved` to a Stock aggregate, inside a file that is supposed to describe Order Management. The rulebook said not to introduce new aggregates, and two appeared. The model did not invent the names: Payment and Stock are both on the vocabulary line. It promoted terms the context merely refers to into aggregates the context owns. Payment belongs to the Payment context, and Stock to the warehouse responsibility *Chapter 4* identified; Order Management only hears about them. If Order Management treats them as its own, the model has silently collapsed bounded contexts. The language looks correct, but the ownership is wrong. This mistake is exactly the kind of mistake that a fluent LLM can hide unless the rulebook forces the distinction to become visible.

There is a smaller slip in the same output. The workshop in *Chapter 4* settled on Order submitted, and the model returned `OrderPlaced`. The vocabulary line named Order, Payment, and Stock and said nothing about event names, so the model chose one. In a chapter about forbidden synonyms, that omission is the lesson.

So, the next rules are not abstract. They come directly from the critique:

```
- Order Management owns the Order aggregate only.
- Payment and Stock are external concepts for this context.
- Events produced by other bounded contexts must be marked as external inputs.
- Do not treat external events as internal state changes.
- Use the canonical event names from the glossary: OrderSubmitted,
OrderConfirmed, OrderCancelled.
- If ownership is unclear, add an open question instead of inventing a rule.
```

With those rules added, the same request in a fresh conversation returned this:

```
owned_aggregate: Order
command:
  - Submit Order
  - Confirm Order
  - Cancel Order
internal_event:
  - OrderSubmitted
  - OrderConfirmed
  - OrderCancelled
external_input:
  - Payment
  - Stock
policy:
  - After OrderSubmitted, Payment and Stock are external inputs.
  - Confirm Order only after Payment and Stock indicate success.
  - Confirm Order produces OrderConfirmed.
  - Cancel Order produces OrderCancelled.
open_question:
  - What are the canonical external event names from Payment and Stock?
  - Should an unsuccessful Payment or Stock cause Cancel Order?
```

Read against the first draft, three things changed. Order is now the only aggregate the file claims, Payment and Stock have moved to `external_input`, and the event names match the glossary instead of the model's preference. The open questions are particularly important: rather than inventing names for events it does not own, the model asked what Payment and Stock actually call them. The rule about unclear ownership is working.

Two things did not change. The commands came back as Submit Order rather than `SubmitOrder`, because the rulebook pinned canonical event names and said nothing about commands. And `external_input` lists two contexts rather than two events, for the reason the model raised itself. Both become inputs to the next iteration, illustrating how a living rulebook develops.

This is the practical loop the chapter has been building toward. You do not ask the model for the final answer. You ask it for a structured draft, criticize the draft using DDD judgment, and then promote the criticism into a new rule. Over time, the rulebook becomes a living record of domain knowledge: a compact set of constraints that protects vocabulary, ownership, boundaries, and invariants.

When rules are precise, small, and continuously refined, the assistant can operate more consistently within the domain constraints the team has defined.

Summary

In this chapter, we moved from stable domain vocabulary to disciplined reasoning. We saw that even with a clear Ubiquitous Language, an LLM can produce fluent output that violates the underlying design. The defense is not simply a better prompt, but a small set of rules that constrain how the model is allowed to reason inside one bounded context.

The BrewUp Payment moment, where the model silently invented a Charge aggregate, made the cost of generic instructions tangible. A textbook DDD checklist did nothing to prevent that kind of failure because it did not say anything about the project. A short Payment-specific rulebook, by contrast, narrowed the space the model could explore through forbidden synonyms, currency invariants, and refund preconditions. Rules earn their keep when they speak the language of one context.

You learned that:

- A useful rulebook starts with the vocabulary, invariants, and boundaries of one specific bounded context. Textbook definitions might sound rigorous but often constrain little because the model already knows them.
- Every rulebook can be read as three layers: language fixes what the model may call things, semantics fixes how those things are allowed to behave, and structure fixes what shape the answer has to take. The Payment file does all three in a dozen lines.
- Agents, skills, and MCP servers are three different containers for rules. The agents stabilize who the model is supposed to be while reasoning. Skills stabilize how a specific modeling task is performed. MCP servers expose trusted project sources the assistant can consult under explicit rules.
- Structured output acts as a guardrail by making the model's reasoning easier to review and compare. Asking for YAML with named fields for commands, events, and aggregates makes the model's reasoning reviewable, comparable across iterations, and reusable in later modeling work.
- A rulebook develops iteratively through critique. The first rules come from bugs the team has already paid for, from the glossary, and from the questions a workshop could not close. The rest come from running the model, finding where it has silently collapsed bounded contexts, and promoting each correction into a new rule.

In the next chapter, you move from defining rules to using them in the iterative refinement of a bounded context. You will see how feedback loops between the team and the assistant gradually shape events, commands, aggregates, and policies into a coherent design, with **command query responsibility segregation** (**CQRS**) as the lens that organizes the model's reasoning around clear command and query roles.

References

- Anthropic (2024). Model Context Protocol specification. `https://modelcontextprotocol.io/` (accessed September 2026).

6

Iterative Refinement: Building the Bounded Context

In the previous chapters, we turned AI collaboration from a vague interaction into a disciplined modeling practice. *Chapter 4* showed how EventStorming and Domain Storytelling can transform raw domain exploration into structured material. *Chapter 5* showed how to protect that material through rules, constraints, and reusable output structures. However, rules alone do not produce a coherent bounded context.

A bounded context becomes clear through repeated refinement. The model generates a candidate interpretation. The architect critiques it against domain knowledge and promotes corrections into rules. The team runs the model again under sharper constraints. Each cycle makes the boundary more explicit, and the reasoning environment less generic.

This chapter applies that loop to the BrewUp Order Management context. You start from a first AI-generated draft that looks reasonable on the surface but silently collapses ownership across Order, Payment, and Stock. From there, you use feedback loops to separate owned concepts from external inputs, introduce **command-query responsibility segregation** (**CQRS**) as a role-separation lens, refine commands, events, and aggregates, and finally evaluate whether the bounded context is coherent enough to support the next design step.

The goal is not to obtain a perfect model in one prompt. The goal is to make mistakes visible, turn corrections into reusable constraints, and make the same mistakes progressively harder to repeat.

In this chapter, you will learn:

- How **generate–critique–correct loops** turn flawed AI output into reusable modeling constraints.

- How to distinguish naming mistakes from ownership mistakes, and why external facts must not be treated by the model as events emitted by the wrong aggregate.
- How CQRS separates commands from queries, while complementary DDD roles preserve the meaning of events, policies, read models, and external inputs during AI-assisted reasoning.
- How to refine aggregate boundaries by separating collaboration from ownership and confirmed concepts from proposed ones.
- How to recognize **contextual coherence** and capture it in a bounded **context snapshot** that preserves decisions, rejected concepts, and open questions.

Guiding the model through feedback loops

The first AI-generated model is useful because it is wrong in a visible way: its errors give the team something concrete to inspect and challenge. The visible error may sound counterintuitive, especially after the emphasis on rules and constraints, but that is precisely the point. You do not use AI because the first output is reliable; you use it because it can quickly externalize a plausible interpretation of the domain. Once that interpretation is visible, you can inspect it, criticize it, and transform its errors into sharper rules. That inspection is where iterative refinement begins.

To see the loop in action, consider this first draft produced by an AI assistant after the author asked it to model the Order Management context:

```
context: Order Management

aggregates:
  - Order
  - Payment
  - StockReservation

commands:
  - SubmitOrder
  - AuthorizePayment
  - ReserveStock
  - ConfirmOrder

events:
  - OrderSubmitted
  - PaymentAuthorized
```

```
    - StockReserved
    - OrderConfirmed
```

At first glance, this output looks good. The output uses DDD-shaped language and defines aggregates, commands, and events. The names are clear: commands use the imperative form, and events use the past tense. The sequence also resembles a reasonable e-commerce flow: the customer submits an order, the payment service authorizes payment, the stock service reserves stock, and the system confirms the order.

The output reads fluently, which is precisely what makes it difficult to question. The AI has produced something that looks like a domain model while quietly violating the ownership rules established in the previous chapter. The primary problem is semantic ownership rather than naming. More precisely, there are multiple violations:

- Order Management does not own `Payment`.
- Order Management does not own `StockReservation`.
- `AuthorizePayment` is not an Order Management command.
- `ReserveStock` is not an Order Management command.
- `PaymentAuthorized` is not an internal Order Management event.
- `StockReserved` is not an internal Order Management event.

These ownership violations are not cosmetic adjustments; they change the meaning of the bounded context.

If you simply rename Payment to `PaymentInfo` or `StockReservation` to `ReservedStock`, you have not fixed the model. You have only hidden the error behind softer names. The problem is not awkward wording; the model has assigned responsibility to the wrong context.

Order Management owns the Order aggregate. Order Management does not own `Payment`. Order Management does not own Stock. `PaymentAuthorized` and `StockReserved` might be relevant to Order Management, but other bounded contexts produce them. Inside this bounded context, Order Management must treat them as external inputs, not as internal events raised by the Order aggregate.

These two failures differ in kind. A naming error means the model used the wrong word for a concept that does belong here; an ownership error means it placed the concept in the wrong bounded context altogether.

You can usually address the first kind of error through vocabulary stabilization; the second requires an explicit rule. If you do not add that rule, the model will likely repeat the same mistake in a later interaction, even with better names and more convincing prose.

Telling those two kinds of error apart is the first job of the refinement loop shown in *Figure 6.1. The loop* lays out the six steps of a single pass. These steps are to generate, critique, correct, promote the correction into a rule, re-run, and compare.

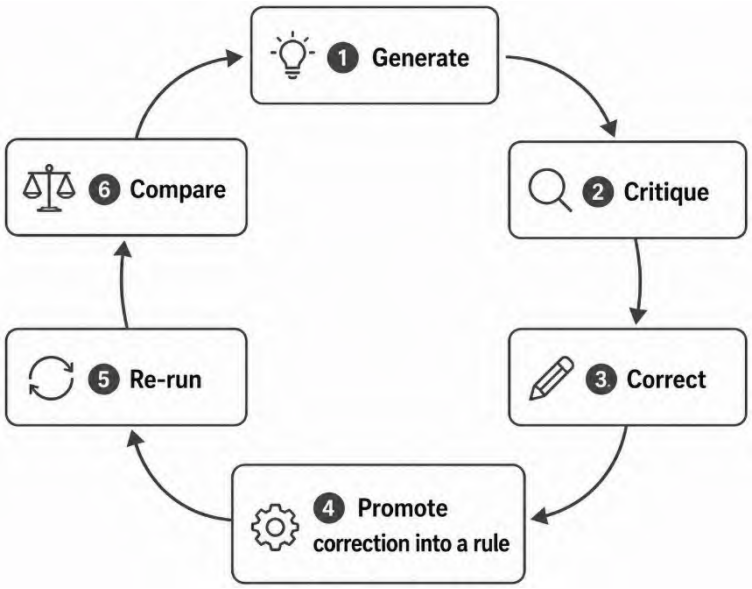


Figure 6.1 – Refinement loop

The loop is a modeling discipline. The rest of this chapter applies it four times. The applications are first to ownership, then to reasoning roles through CQRS, then to aggregate boundaries, and finally to the coherence of the whole bounded context. Each pass leaves behind a rule that the next pass inherits.

The generation step gives you a visible draft. The critique step applies human judgment against domain knowledge, ownership, and boundaries, while the correction step produces a better version of the model. The important move comes next: the correction must become a reusable rule. Only then do you run the model again and compare the new output with the previous one.

Without that promotion step, you will keep correcting the same failure manually. With it, every correction improves the reasoning environment. Here is a practical prompt that applies this discipline to the BrewUp Order Management draft:

```
Context:
We are refining the BrewUp Order Management bounded context.

Known rules:
```

```
- Order Management owns the Order aggregate only.
- Payment and Stock are external concepts.
- Events produced by other bounded contexts must be marked as external inputs.
- If ownership is unclear, add an open question instead of inventing a rule.

Task:
Review the draft model below.

For each item:
1. Keep it if it belongs to Order Management.
2. Move it to external_inputs if it is produced by another bounded context.
3. Reject it if it invents ownership.
4.
Preserve existing open questions and add a new one only if the critique exposes
unresolved ownership.

Return YAML with:
- accepted_items
- moved_to_external_inputs
- rejected_items
- open_questions
- new_rules
```

Notice the shape of the prompt. The prompt does not ask the model to *improve the design* in a generic way. A generic request would invite the model to invent more structure. Instead, it gives the model a classification task under known ownership rules. *Figure 6.2* shows the four verbs that carry the weight here – keep, move, reject, and question – because each one narrows what the model is allowed to do with an item.

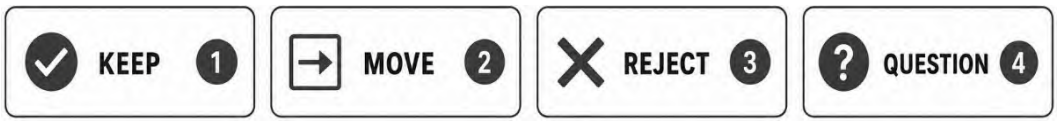


Figure 6.2 – The classification filter used during critique

The assistant is not allowed to absorb everything into Order Management. The assistant must decide whether each item belongs inside the context, crosses into it from outside, should be rejected by the assistant because it invents ownership, or exposes a question that the current domain knowledge cannot answer.

A refined output might look like this:

```
accepted_items:
  aggregates:
    - Order

  commands:
    - SubmitOrder
    - ConfirmOrder

  internal_events:
    - OrderSubmitted
    - OrderConfirmed

moved_to_external_inputs:
  - PaymentAuthorized
  - StockReserved

rejected_items:
  aggregates:
    - Payment
    - StockReservation

  commands:
    - AuthorizePayment
    - ReserveStock

open_questions:
  - What happens when stock reservation fails after payment authorization?
  - How long can an order wait before expiration?
  - Is cancellation always allowed before confirmation?

new_rules:
  - Payment commands are not part of Order Management.
  - Stock reservation commands are not part of Order Management.
  - External inputs may influence Order Management, but they must not be modeled
as events emitted by the Order aggregate.
  - If an external input causes an Order decision, model the resulting Order-owned
command, policy, or event explicitly.
```

This output is not yet the final model, but it is already a stronger modeling artifact. The output keeps Order as the only aggregate owned by Order Management, moves `PaymentAuthorized` and `StockReserved` to `external_inputs`, and rejects `Payment`, `StockReservation`, `AuthorizePayment`, and `ReserveStock` because they imply ownership this bounded context does not have.

The more important result is the new rule set. `PaymentAuthorized` is not merely moved once; its classification confirms an existing rule: events produced by other bounded contexts must remain external inputs. `ReserveStock`, however, exposes a more specific constraint worth preserving: stock reservation commands are not part of Order Management. The rulebook grows this way: concrete failures become reusable modeling constraints.

The refined output exposes what the happy path hides: what happens if stock reservation fails after payment authorization, whether cancellation is always allowed before confirmation, and how long an order can remain unconfirmed.

These questions should not be answered by invention. The following listing shows the resulting open questions: `StockReservationFailed` belongs in external input; what Order Management should do after receiving it remains an open question. Expiration might become part of the Order life cycle, but the timing rule is still missing. At this stage, visible uncertainty is more useful than artificial completeness.

```
context: Order Management
owned_aggregate:
  - Order

commands:
  confirmed:
    user_initiated:
      - SubmitOrder

    policy_triggered:
      - ConfirmOrder

  proposed:
    policy_triggered:
      - ExpireOrder

internal_events:
  confirmed:
    - OrderSubmitted
```

```
    - OrderConfirmed

  proposed:
    - OrderExpired

external_inputs:
  - PaymentAuthorized
  - StockReserved
  - StockReservationFailed

open_questions:
  - Is cancellation always allowed before confirmation?
  - How long can an order remain unconfirmed before it expires?
  - What should Order Management do after StockReservationFailed?
```

Notice that `ExpireOrder` is marked as proposed. The refinement loop has surfaced expiration as a likely Order life cycle concern, but the business rule that governs it is still missing. The model can identify a candidate transition, but it cannot confirm the policy that would make it valid. This draft is therefore intentionally incomplete.

This draft does not decide how the coordination around payment and stock will later be implemented: through a saga, process manager, application service, or another coordination mechanism. That decision belongs to a later architectural step. For now, the important modeling distinction is simpler and more foundational: Order Management can react to external inputs, but it must not claim ownership of the processes that produce them.

`ConfirmOrder` is therefore better understood as a **policy-triggered command**. It is still a command because it requests a state transition in the Order aggregate. What changes is its origin: it is not issued directly by the customer but triggered by an Order Management policy once the required external facts have been observed. `PaymentAuthorized` and `StockReserved` do not become internal Order events merely because they influence that decision. They remain external inputs. If the policy decides that the order can be confirmed, the resulting Order-owned transition is expressed through `ConfirmOrder` and, if successful, through the internal event `OrderConfirmed`.

LLMs often blur exactly this kind of distinction. The model sees a familiar e-commerce sequence and tends to place related concepts together unless ownership has been made explicit. That is why feedback loops must keep applying pressure to the boundary: they prevent a familiar narrative from flattening the domain into a generic retail workflow.

Maintaining that pressure is your job, and in practice it means asking one question each time the model crosses the boundary:

- When the model introduces `Payment` as an aggregate, ask: who owns it?
- When the model proposes `AuthorizePayment`, ask: who can issue that command?
- When the model emits `PaymentAuthorized` internally, ask: which aggregate raised it?
- When the model invents a rule for cancellation, ask: is this known domain knowledge or a plausible assumption?

If the answer is unclear, the correct output is not a fabricated rule. It is an open question. Preserving that gap is one of the most important habits in AI-assisted modeling: open questions prevent plausible assumptions from hardening into structure before the team has made a decision.

After each iteration, the architect should ask two questions:

- What did the model get wrong?
- What rule would prevent this class of error from returning?

That second question turns feedback from a comment into an asset. The assistant becomes less generic not because it has learned in the human sense, but because the reasoning environment has become narrower, clearer, and more domain-specific. Before CQRS can help you separate commands from queries, and before you can classify the surrounding domain roles consistently, you must first establish who owns each concept.

Introducing CQRS and role separation in AI reasoning

Ownership refinement has taken you as far as it can: it settles that context a concept belongs to, but not what type of thing that concept is once it is inside. At this point, in the refinement process, the problem is no longer discovering events or stabilizing vocabulary. The challenge becomes more subtle: preventing the model from collapsing different responsibilities into the same conceptual space.

Command Query Responsibility Segregation (CQRS) gives us the first axis of separation.

In many software discussions, CQRS is introduced as an architectural pattern focused on separating reads from writes. While technically correct, that framing is too narrow for the modeling work you are doing here. In the context of AI-assisted Domain-Driven Design, its value begins with a simpler distinction: separating intentions that might change state from requests that only observe it. Around that distinction, you can make the other domain roles explicit instead of allowing the model to collapse them into a single flow.

When an LLM receives a partially refined domain flow, it naturally tends to compress related concepts into a single narrative. Commands become events, events become process steps, queries become business decisions, and policies disappear entirely inside generic prose. The result is not obviously wrong. In fact, it often looks fluent and coherent. However, the semantic distinctions that make a bounded context stable begin to dissolve. CQRS gives you one axis of separation: commands express intent to change state, while queries request information without changing it. This distinction does not define the rest of the classification used here. Alongside that CQRS axis, you keep four additional domain roles explicit: internal events, external inputs, policies, and read models.

An internal event expresses a fact produced within the bounded context. An external input represents a fact produced elsewhere but relevant to its decisions. A policy expresses the reaction rule that might turn an observed fact into a new command. A read model organizes information for the query side. Keeping these roles distinct prevents the model from collapsing ownership, reaction, and observation into the command/query distinction.

Consider the following BrewUp example:

```
commands:
  - SubmitOrder
  - ConfirmOrder

events:
  - OrderSubmitted
  - OrderConfirmed
  - PaymentAuthorized
  - StockReserved

queries:
  - GetOrderStatus
```

At first glance, the structure appears reasonable. However, semantically, several elements are still mixed together.

Neither the event named `PaymentAuthorized` nor the event named `StockReserved` is an internal Order Management event. They are external facts produced by other bounded contexts. `ConfirmOrder` is a command, although the customer does not initiate it, because it requests a state transition. In this chapter, `policy_triggered` describes the command's origin, not a separate CQRS category. Because the query named `GetOrderStatus` observes state without requesting a state transition, it belongs on the query side of the model.

Without explicit classification, the model tends to flatten these distinctions into a single sequential flow:

```
Customer submits order
   payment is authorized
   stock is reserved
   order is confirmed
   user checks status
```

The sequence is coherent from a linguistic perspective, but it hides ownership, decision boundaries, and responsibility separation. The CQRS command/query split gives you one axis of separation, while the surrounding domain roles keep ownership, reactions, and external facts from collapsing into the same narrative.

Ask the model this question:

```
Design the order flow.
```

You can ask:

```
Classify the following elements according to their role:
- command
- internal event
- external input
- policy
- query
- read model
```

This classification changes the interaction dramatically. The task no longer rewards a smooth narrative. It rewards the preservation of distinctions.

In practice, explicit role classification reduces semantic drift because it narrows the probability space available to the model. CQRS provides the command/query separation; the additional DDD categories preserve ownership, reaction, and event semantics. Once an element has been classified by the model as an external input, the model is less likely to reinterpret it later as an internally owned event. Once a query is separated from commands, the model is less likely to invent side effects around read operations.

CQRS also arrives relatively late in the refinement process for a reason. EventStorming and Domain Storytelling help you discover the domain; iterative refinement helps us critique and stabilize the emerging model. Only once ownership is visible does CQRS add anything as a role-separation lens.

At this stage, CQRS is about protecting conceptual integrity rather than handlers, buses, or distributed architecture. The important shift is subtle but fundamental.

Here, *CQRS is introduced to help you separate meanings, not technologies.*

Once the reasoning environment consistently distinguishes intentions, facts, reactions, and reads, the bounded context stops behaving like a generic workflow description and starts behaving like a structured domain model. What role separation does not tell us is how those elements group together, and that is what the next round of refinement has to settle.

Refining events, commands, and aggregates

Once commands, events, queries, and policies have been separated conceptually, the refinement process can move to a deeper level.

The question is no longer only: *what type of element is this?* The question becomes:

Which elements truly belong together inside the same consistency boundary?

At this stage, however, we must be careful not to approach aggregates as technical containers or persistence structures. In AI-assisted modeling, aggregates are first and foremost semantic boundaries. They define where invariants are protected, where decisions are made, and which events can legitimately emerge from the same transactional consistency boundary.

This distinction matters because LLMs tend to group concepts that frequently appear together in business workflows. If payment, stock reservation, and order confirmation appear together repeatedly, the model might infer that they belong to the same aggregate or bounded context. On the surface, that compression looks coherent. Architecturally, it might be wrong.

Consider this early AI-generated refinement for BrewUp Order Management:

```
aggregate:
  name: Order

properties:
  - orderLines
  - paymentStatus
  - reservedStock
  - shipmentTracking

commands:
  - SubmitOrder
  - AuthorizePayment
  - ReserveStock
```

```
    - ConfirmOrder
    - ShipOrder

  events:
    - OrderSubmitted
    - PaymentAuthorized
    - StockReserved
    - OrderConfirmed
    - ShipmentCreated
```

The output looks structured, fluent, and superficially aligned with DDD terminology. Yet almost every modeling boundary has collapsed.

The aggregate now appears responsible for:

- Accepting orders
- Authorizing payments
- Reserving inventory
- Tracking shipments

These concepts participate in the same collaboration without sharing ownership.

The Order aggregate might depend on external facts produced by Payment or stock contexts, but dependency does not imply containment. An aggregate should not absorb every concept that participates in the same business flow. Separating coordination from ownership is the refinement challenge here.

The most useful way to critique these AI-generated drafts is not to ask whether the model *understood DDD*, but to ask a sequence of ownership-oriented questions:

- Which invariants does the aggregate protect?
- Which decisions require transactional consistency?
- Which events does the aggregate truly emit?
- Which concepts belong to another bounded context?
- Which actions are reactions rather than owned behavior?

When you apply these questions to the previous example, several corrections immediately emerge. `AuthorizePayment` and `ReserveStock` are not Order commands, and `ShipmentCreated` is not an Order event.

`PaymentStatus` and `ReservedStock` should not represent internally owned lifecycles. The Order aggregate might store minimal state derived from external facts, such as whether

payment authorization or stock reservation has been observed. Observing an external fact is legitimate; owning the process that produces it is not.

The refined version becomes much narrower:

```yaml
aggregate:
  name: Order

responsibilities:
  - Track the lifecycle of a customer order
  - Decide whether the order can be confirmed
  - Emit order-owned events

commands:
  confirmed:
    - SubmitOrder
    - ConfirmOrder
  proposed:
    - CancelOrder
    - ExpireOrder

internal_events:
  confirmed:
    - OrderSubmitted
    - OrderConfirmed
  proposed:
    - OrderCancelled
    - OrderExpired

required_external_inputs:
  - PaymentAuthorized
  - StockReserved
  - StockReservationFailed
```

Notice what changed. The business process did not become simpler: payment, stock, shipment, expiration, and cancellation still exist. What became clearer is ownership. Those concerns no longer collapse into the Order aggregate. An oversized aggregate is one of the useful side effects of an AI-generated draft: it makes hidden ambiguity visible.

The same principle applies to events. During early iterations, models frequently produce events that are technically plausible but semantically weak:

```
OrderProcessed
PaymentCompleted
InventoryUpdated
OrderHandled
```

These names sound reasonable because they match generic enterprise software patterns. However, they often hide important domain distinctions.

`PaymentCompleted` may incorrectly collapse authorization, capture, and settlement into a single business transition.

`InventoryUpdated` says almost nothing about the business meaning of the change.

`OrderProcessed` usually describes a workflow phase rather than a meaningful domain fact.

Refinement forces the team to ask:

- What exactly happened?
- Who recognizes this fact as meaningful?
- Which bounded context owns it?
- What downstream decision depends on it?

Meaningful domain facts therefore tend to become more precise over time rather than more generic:

```
PaymentAuthorized
StockReserved
OrderConfirmed
StockReservationFailed
OrderExpired
```

Not all of these facts are emitted by the Order aggregate. Order Management observes the events named `PaymentAuthorized`, `StockReserved`, and `StockReservationFailed` as external facts, while `OrderConfirmed` and `OrderExpired` are emitted by the Order aggregate as order-owned events. What matters here is the naming quality: each name describes a specific business transition rather than a generic processing step.

A refined command vocabulary follows the same direction. The command vocabulary becomes narrower, more intentional, and more aligned with business decisions:

```
SubmitOrder
ConfirmOrder
CancelOrder
ExpireOrder
```

Refinement often removes ambiguity rather than adding concepts.

Every iteration removes interpretations that the model might otherwise consider plausible. Over time, the reasoning space becomes narrower.

Feedback loops become essential here too. The model alone cannot determine whether an aggregate boundary is correct because aggregate boundaries are not statistical truths. They are business decisions shaped by consistency requirements, ownership, organizational structure, and domain semantics.

The AI helps accelerate the refinement cycle by exposing collapsed boundaries, proposing candidate separations, surfacing missing transitions, revealing naming inconsistencies, and externalizing implicit assumptions.

The architect still owns the decision; the model helps expose the choices required to define the aggregate.

Achieving contextual coherence through iteration

At this point, the BrewUp Order Management context has gone through the full refinement loop. Ownership has been corrected, reasoning roles have been separated, and the Order aggregate has been narrowed to the decisions it actually owns.

The next question is practical: when is the bounded context coherent enough for the team to move forward? The rest of this section builds the answer in five steps: comparing outputs across cycles, checking that ownership and roles hold still, treating open questions as part of coherence, freezing the result in a context snapshot, and finally asking the model to audit that snapshot.

A bounded context becomes coherent when its language, responsibilities, rules, and boundaries reinforce each other across repeated iterations. Coherence is a weaker claim than completeness: the answers already available fit together without semantic contradiction, while unanswered questions remain visible.

In early domain modeling, this is the safer target. A coherent model with visible open questions is more useful than a complete-looking model built on unstable ownership.

Consider the refined BrewUp model again. Because of the first pass, it has gained only two things: cancellation as a proposed user-initiated command and its matching event:

```
commands:
  proposed:
    user_initiated:
      - CancelOrder

internal_events:
  proposed:
    - OrderCancelled
```

Everything else remains unchanged from the earlier listing: the single owned aggregate, the confirmed commands, the external inputs, and the three open questions. The stability is the point. The model is incomplete. It still contains open questions, does not define every policy, and leaves implementation concerns such as coordination and message transport undecided. Yet it is already more coherent than the first AI-generated draft.

The reason is the stability of the model's boundaries and roles.

Order Management owns the `Order` aggregate. `Payment` and `Stock` remain outside Order Management. Events raised by other bounded contexts Order Management treats as external inputs. Commands you classify by origin. Internal events the `Order` aggregate emits only. Missing domain knowledge appears as open questions instead of fabricated rules.

This stability is **contextual coherence**: stable ownership, explicit roles, clear external inputs, internal events emitted only by the Order aggregate, and visible uncertainty where domain knowledge is still missing. That level of maturity is what you want before moving into a deeper design or implementation.

Coherence emerges from repeated comparison

A common mistake is to treat each AI output as an isolated artifact. The team asks for a model, reviews it, corrects it, and then moves on. But iterative refinement becomes powerful only when the team compares outputs across cycles.

Instead of asking only whether the current output is correct, compare it with the previous iteration and ask whether its ownership, roles, and unresolved questions are becoming more coherent.

This comparison makes progress visible by showing which ambiguities have disappeared, which boundaries have stabilized, and which uncertainties remain.

The bounded context takes shape that way, not by discovering a perfect boundary in one step but by progressively eliminating contradictions that should not survive inside the same model.

A useful way to read each iteration is to ask what changed:

- Did owned aggregates remain stable?
- Did external inputs stay external?
- Did internal events remain internally produced?
- Did commands preserve their role?
- Did policies become explicit instead of hidden in prose?
- Did unclear decisions become open questions?
- Did the model avoid reintroducing rejected concepts?

If the answer is consistently yes, the bounded context is converging toward a more stable representation.

If the same mistakes keep returning, the reasoning environment is still too weak. The rulebook needs sharper constraints, the vocabulary might need more precision, or the prompt might be asking for too much in one step. Contextual coherence is therefore observed across iterations rather than supplied by a single model output.

Stable ownership and stable roles

A bounded context cannot be coherent if the ownership of concepts keeps shifting. If `Payment` belongs to Order Management in one iteration and somewhere else in the next, the model is not refining; it is drifting.

In BrewUp Order Management, the ownership boundary should remain stable across iterations:

```
owned_by_order_management:
  - Order

not_owned_by_order_management:
  - Payment
  - StockReservation
  - Shipment
  - Invoice
```

What counts is not that these relationships have been defined once by the team, but that later iterations preserve them. Roles need the same stability: commands, internal events, external inputs, and policies should preserve the same semantic role across successive iterations.

For BrewUp Order Management, the classification might stabilize like this:

```yaml
commands:
  confirmed:
    user_initiated:
      - SubmitOrder
    policy_triggered:
      - ConfirmOrder
  proposed:
    user_initiated:
      - CancelOrder
    policy_triggered:
      - ExpireOrder

internal_events:
  confirmed:
    - OrderSubmitted
    - OrderConfirmed
  proposed:
    - OrderCancelled
    - OrderExpired

external_inputs:
  - PaymentAuthorized
  - StockReserved
  - StockReservationFailed

policies:
  confirmed:
    - Confirm an order when payment authorization and stock reservation
      have both been observed.
```

As with ownership, the classification matters less than its persistence. External inputs do not become internal events, proposed transitions do not silently become confirmed, and **policy-triggered commands** do not turn into user actions. These distinctions protect the context from semantic collapse.

Open questions are part of coherence

A coherent model still has open questions, and a model that resolves them too quickly should prompt closer scrutiny. LLMs naturally tend to complete missing structure, even when the

domain knowledge required to complete it has not been provided. This tendency is why open questions are a safety mechanism rather than a weakness.

Compare these two outputs:

```
policy:
  - If stock reservation fails, automatically cancel the order.
```

and:

```
open_questions:
  - What should Order Management do after StockReservationFailed?
  - Should the order be cancelled immediately, kept pending, or retried?
  - What should happen to the existing payment authorization?
```

The first output is complete. The second output is more coherent.

The first invents a business rule. The second preserves uncertainty and exposes the decision that the team still needs to make.

A fabricated rule can easily survive into later iterations simply because it has become part of the context. Open questions prevent that regression by marking the boundary between known domain knowledge and plausible speculation.

This behavior should itself become a reusable constraint:

```
uncertainty_rules:
  - Do not invent domain rules to complete the model.
  - If ownership, policy, or reaction behavior is unclear, preserve it as an open
question.
  - Keep proposed concepts separate from confirmed concepts.
```

Coherence is reinforced by a context snapshot

As iterations accumulate, the team needs a stable artifact that represents the current state of the bounded context. Otherwise, the decision that survived one refinement cycle might disappear or regress in the next. The team therefore maintains a **context snapshot**.

A context snapshot is a compact, reusable representation of the current bounded context rather than a full specification. It captures the decisions that have survived refinement, the distinctions that must remain stable, and the questions that remain open. In practice, the snapshot is kept as a versioned document alongside the bounded context it describes, not in

chat history, and it is the first thing handed to any later AI session about Order Management. *Chapter 7* takes that habit further and turns the snapshot into a maintained domain carrier.

For BrewUp Order Management, a snapshot might look like this:

```
bounded_context: Order Management

purpose:
  - Manage the Order-owned lifecycle of a customer order and the decisions
    that belong to Order Management.

owned_concepts:
  aggregates:
    confirmed:
      - Order

  commands:
    confirmed:
      user_initiated:
        - SubmitOrder
      policy_triggered:
        - ConfirmOrder
    proposed:
      user_initiated:
        - CancelOrder
      policy_triggered:
        - ExpireOrder

  internal_events:
    confirmed:
      - OrderSubmitted
      - OrderConfirmed
    proposed:
      - OrderCancelled
      - OrderExpired

external_inputs:
  - PaymentAuthorized
  - StockReserved
  - StockReservationFailed
```

```
policies:
  confirmed:
    - Confirm an order when payment authorization and stock reservation
      have both been observed.

ownership_constraints:
  - Payment is not owned by Order Management.
  - StockReservation is not owned by Order Management.
  - Shipment is not owned by Order Management.
  - Invoice is not owned by Order Management.
  - External inputs must not be modeled as events emitted by the Order aggregate.

open_questions:
  - Is cancellation always allowed before confirmation?
  - How long can an order remain unconfirmed before it expires?
  - What should Order Management do after StockReservationFailed?
```

This artifact turns iteration into accumulated knowledge. The snapshot gives the model a stable starting point, keeps confirmed decisions separate from proposed ones, preserves open questions, and prevents previously resolved ownership and modeling decisions from regressing in later iterations. Without a context snapshot, the team has a conversation. With one, the team has a durable record that later sessions can inspect and preserve.

Testing coherence with a final review prompt

Once the snapshot exists, you can use the model to review it for internal contradictions rather than to declare it correct.

A useful prompt is deliberately skeptical:

```
Context:
You are reviewing the current BrewUp Order Management bounded context snapshot.

Task:
Evaluate the snapshot for contextual coherence.

Check for:
1. Concepts whose ownership contradicts the bounded context.
2. External inputs incorrectly treated as internal events.
3. Commands that belong to another bounded context.
4. Proposed concepts promoted to confirmed without supporting domain knowledge.
5. Policies that invent missing business rules.
```

```
6. Open questions that have been silently resolved without domain input.
7. Naming or role inconsistencies across commands, events, policies, and external
inputs.

Return:
- assessment
- detected_inconsistencies
- recommended_corrections
- rules_to_add
- open_questions_to_preserve

Do not add new domain rules unless they are directly supported by the snapshot.
Do not resolve open questions.
```

Those last two instructions are what keep the review from turning into another generation step. The prompt asks for `recommended_corrections` and `rules_to_add`, which look like generation, so you must tell the model that both must come out of the snapshot rather than out of its own judgment: it is auditing the snapshot against stated criteria, not completing the design. A possible review might return:

```
assessment: coherent_with_open_questions

detected_inconsistencies: []
recommended_corrections: []
rules_to_add: []

open_questions_to_preserve:
  - Is cancellation always allowed before confirmation?
  - How long can an order remain unconfirmed before it expires?
  - What should Order Management do after StockReservationFailed?
```

This output is useful precisely because it does not attempt to complete the domain model. The review finds no contradiction in the current snapshot, but it preserves the decisions that still require domain input. That distinction matters. Contextual coherence does not mean that every question has been answered. Contextual coherence means that what the team proposes, what is proposed, and what remains uncertain can coexist without contradicting each other.

When the bounded context is coherent enough

The team eventually needs to move forward; iteration cannot continue indefinitely. The purpose of refinement is to reduce ambiguity enough for the next design activity to proceed safely.

A bounded context is coherent enough when repeated iterations stop producing major structural changes, and the remaining uncertainty is explicit rather than hidden inside assumptions.

For BrewUp Order Management, the answer to each of those questions is now yes. Ownership has held: Order is still the only owned aggregate, and Payment, Stock, Shipment, and Invoice have not reappeared as owned concepts. The roles have held too: `PaymentAuthorized`, `StockReserved`, and `StockReservationFailed` are still external inputs, internal events are still emitted only by the Order aggregate, and commands are still classified by origin. Nothing has quietly hardened: confirmed concepts remain separate from proposed ones, open questions remain open rather than converted into invented rules, and no ownership or role decision resolved earlier has regressed.

These criteria are semantic rather than implementation criteria: at this stage the question is not whether the eventual architecture or code is correct, but whether the bounded context says one consistent thing about the domain.

The output of this chapter is therefore a stable bounded context model that can support future decisions, not a final architecture.

Coherent models provide a stronger foundation for later architectural decisions.

The model generates candidates, the architect applies judgment, the rulebook preserves corrections, and the snapshot stabilizes the bounded context. Once the bounded context reaches this level of stability, the snapshot can become the foundation for more durable forms of AI collaboration. The next chapter builds on exactly this point, turning preserved domain understanding into reusable working context.

Summary

In this chapter, you moved from defining rules to applying them through iterative refinement. The first AI-generated model of BrewUp Order Management looked plausible, but it silently collapsed `Order`, `Payment`, and `StockReservation`. This made the central problem visible: the model was not only making naming mistakes, but ownership mistakes.

You used this flawed draft as the starting point for a generate–critique–correct loop. Each correction became more than a local fix: it was promoted into a reusable rule.

`PaymentAuthorized` and `StockReserved` were moved to `external_inputs`, `Payment` and `StockReservation` were removed from the concepts owned by Order Management, and missing decisions became open questions instead of fabricated policies.

CQRS provided an additional role-separation lens. It helped distinguish intentions that might change state from requests that only observe it, while the broader domain classification kept internal events, external inputs, and policies semantically distinct. This made it possible to preserve `ConfirmOrder` as a policy-triggered command, while keeping `PaymentAuthorized` and `StockReserved` as external facts observed by Order Management.

You also refined the `Order` aggregate by separating collaboration from ownership. Concepts can participate in the same business flow without belonging to the same consistency boundary. The Order aggregate can react to external facts, but it must not own the processes that produce them. Finally, you introduced contextual coherence: the point where ownership, roles, confirmed and proposed concepts, and open questions fit together without contradiction, and the context snapshot that holds that state from one cycle to the next.

You learned that:

- The value of the first AI-generated draft is that it makes a flawed interpretation visible.
- Feedback becomes valuable when you promote each correction into a reusable rule.
- Ownership mistakes are more dangerous than naming mistakes because they change the meaning of the bounded context.
- CQRS can be used as a role-separation lens before it becomes an implementation pattern.
- Collaboration does not imply ownership: a bounded context can react to external facts without owning the processes that produce them.
- Open questions are part of coherence: they keep uncertainty visible until the team can decide.
- A context snapshot turns iterative refinement into accumulated domain knowledge by preserving confirmed concepts, proposed concepts, ownership boundaries, and unresolved questions.

In the next chapter, you build on this result and explore how the domain understanding preserved in the context snapshot can become reusable working context for future AI interactions.

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

7

Prompt as Domain Carrier

In the previous chapter, we used iterative feedback loops to refine a bounded context until ownership, events, commands, aggregates, and policies became semantically coherent. The result was a bounded context snapshot: a representation of the team's current understanding of the domain. However, a snapshot alone is not enough. If the knowledge discovered during modeling remains buried inside a conversation, the same ambiguities eventually return. Rejected concepts reappear, open questions become assumptions, and previously resolved decisions must be rediscovered.

This chapter explores how to preserve and reuse that knowledge through a new part of the domain: BrewUp Stock Management. Working in a second bounded context keeps the emphasis on applying the workflow rather than revisiting it, and shows that the technique is not tied to Order Management. The same discipline can preserve meaning across classification, specification, review, change workflows, and later agent-oriented collaboration.

Instead of treating prompts as disposable requests, you begin treating them as **domain carriers**: artifacts that encode what a bounded context owns and what it refuses to assume, so that future interactions can build on a stable foundation.

The chapter also examines a deeper shift. Once domain knowledge becomes part of the prompt itself, an AI output reveals how the model is interpreting the domain. The output also reveals whether the surrounding reasoning environment is strong enough to preserve meaning consistently.

By the end of this chapter, you will understand:

- Why prompts can evolve from temporary instructions into reusable domain carriers
- How to encode domain knowledge so it can be preserved across interactions
- How to use domain carriers to reduce semantic drift and regression

- Why AI outputs should be reviewed as reflections of understanding rather than proof of understanding
- How repeated refinement creates **shared semantic alignment** between humans and AI
- Why Domain-Driven Design provides the boundaries that make long-running AI-assisted modeling possible

From asking to encoding: the knowledge shift

At the end of the previous chapter, the BrewUp Order Management context had reached a useful level of coherence. It was still incomplete, but the main decisions were visible: ownership, external input, internal events, confirmed concepts, proposed concepts, and open questions. That result gave us a bounded context snapshot.

The snapshot captures the current state of domain understanding. It records what the team currently accepts, what it has rejected, what remains uncertain, and which rules should guide future reasoning. Used alone, however, the snapshot is still a reference document. The snapshot describes the model; it does not automatically shape the next interaction with the model.

The next step is to turn that recorded knowledge into reusable working context.

Instead of asking the model about the domain each time, you encode the relevant domain knowledge so the model has to work inside it. The knowledge shift is this: the prompt stops being only a request and starts carrying the rules, language, and constraints the answer must respect.

Early AI interactions usually look like simple requests:

```
Identify the events for this bounded context.
```

or:

```
Generate commands and aggregates for this domain.
```

Those prompts can be useful for a first draft. They are also easy to throw away. Once the answer arrives, the prompt has no further role. The prompt does not preserve the decisions made during review. It does not remember which concepts were rejected. The prompt does not keep confirmed rules separate from open questions. It asks for output, then disappears.

A domain carrier behaves differently. It is a prompt written to hold domain knowledge, not just to request a result.

The domain carrier gives the model a reasoning environment built out of previous modeling work, rather than a question to answer from scratch.

A normal prompt says:

```
Here is what I want you to do.
```

A domain carrier says:

```
Here is the domain knowledge you must preserve while doing it.
```

The change alters the role of the prompt. The prompt becomes a maintained modeling artifact instead of a temporary instruction.

If a prompt contains the rules that protect a bounded context, it deserves the same care as other design materials. The prompt should be reviewed and corrected because rewriting it casually means losing part of the model.

This is why *Chapter 6* matters here. The bounded context snapshot is the raw material from which a domain carrier is built.

The relationship looks like this:

```
context snapshot
    ↓
domain carrier
    ↓
repeatable AI-assisted reasoning
```

The context snapshot records the current state of understanding while the domain carrier turns that understanding into operational guidance.

Once the carrier exists, the model can use it to classify new elements, review proposed changes, generate candidate policies, detect inconsistencies, or prepare specifications for implementation.

What a domain carrier is made of

So far the carrier has been described by what it does. Before building one, it is worth saying what it is, because the rest of this chapter treats it as a concrete file rather than a style of writing prompts.

A domain carrier is a Markdown file, one per bounded context, named after the context it describes. The BrewUp carriers live together in a `domain-carriers` directory, so the one used

throughout this chapter is `domain-carriers/brewup-stock-management.md`, next to `domain-carriers/brewup-sales-order-confirmation.md`. Where that directory sits depends on the harness the team has adopted: in the companion repository, which uses a specification-driven harness, it is `specify/memory/domain-carriers/`. Put the directory wherever your assistant already looks for durable project context and keep the file name tied to the bounded context, because that is the part that must not change when the harness does.

The file has several sections, and the order is deliberate. The file opens with the domain and closes with machinery:

- *Purpose* – what the context is responsible for, stated together with what it is not.
- *Domain authorities* – the neighboring contexts, and which outcomes each of them owns.
- *Ubiquitous Language* – the terms of the context, each with the meaning it carries inside this boundary.
- *Owned concepts* – the aggregates, commands, and internal events the context produces.
- *External inputs* – the facts the context observes but does not emit.
- *Bounded context rules* – numbered rules, BC-001 and onward, stating the boundary in a form a generated artifact can be checked against.
- *Confirmed policies* – rules the team has settled, which generated artifacts might rely on.
- *Proposed policies* – rules still under discussion, which generated artifacts must mark as proposed.
- *Open questions* – decisions not yet taken, which must survive into the output as open questions.
- *Forbidden assumptions* – the completions the model would otherwise reach for, ruled out by name.

Everything after that is machinery rather than domain: what the agent might not invent, what a generated artifact is expected to contain, and a short tail of harness-specific instructions. That tail is the only part that changes when the team changes tools.

How long should it be? In the Stock Management example, the carrier is under four hundred lines: long enough to hold the boundary, but short enough to read before a review. Treat that length as a practical example rather than a threshold. More important is what the carrier contains: if it becomes difficult to review as a single bounded context, consider whether it is carrying more than one boundary. Conversely, shortening it should not come at the cost of losing the decisions and exclusions that give it value.

Who maintains it? The same people who maintain the model: whoever would be in the room for an EventStorming session on that context. The team reviews it in a pull request like any other design material, because a change to the carrier is a change to the model.

When is it updated? After a team decision, not after a model answer. The derive–review–update cycle later in this chapter makes that rule explicit.

How can you tell a strong carrier from a weak one? A weak carrier reads like documentation: it says what the context does. A strong carrier reads like a boundary: it also says what the context refuses to do and what you must not assume. Exclusions and open questions are what separate the two, and they are the first sections a rushed carrier loses.

How does the carrier reach the model? There are four practical ways to supply the carrier with the model as context. The first is by path: an assistant with access to the project is told which file is authoritative, and a harness that keeps durable context in a known directory typically works. The second is an explicit file mention in the prompt, such as `@brewup-stock-management.md` or its equivalent. The explicit file mention is particularly useful because the transcript records which artifact was referenced. The third is to expose the carrier through MCP, which suits knowledge shared across repositories, tools, or teams. The fourth is to paste the carrier directly into the conversation. Pasting the carrier works, but is the least reliable option because the interaction does not preserve a durable reference to the version that was pasted. The prompts printed in this chapter use [Insert domain carrier here] because a page cannot reproduce any of these delivery mechanisms; in a real project, the carrier is supplied through one of them.

Building a Stock Management domain carrier

To make the knowledge shift concrete, move away from Order Management and look at another BrewUp bounded context, Stock Management. Using a second context shows that the technique is not tied to the Order Management example.

In *Chapter 6*, Stock Management appeared from the outside. From the perspective of Order Management, events such as `StockReserved` and `StockReservationFailed` were external inputs, facts observed by Order Management, not events emitted by the Order aggregate.

Now, you reverse the perspective.

Inside Stock Management, those same facts become internal domain events. This boundary is the point of bounded contexts: meaning depends on the boundary. What one context observes from outside, another context might own and produce.

The purpose of Stock Management in BrewUp is to manage product availability and stock reservation for customer orders. Stock Management does not own the customer order lifecycle, authorize payments, or create shipments. Its responsibility is narrower: decide whether

requested stock can reserve, track reservations, release reserved stock when needed, and expose stock-related facts to other contexts.

The excerpt below comes from a single file, not assembled from several. It keeps four sections from that file: purpose, bounded context rules, open questions, and forbidden assumptions. It condenses the first two and reproduces the last two word for word because the rest of this chapter traces generated output back to them. The complete file is in the book's repository, as `domain-carriers/brewup-stock-management.md`:

The **Stock Management carrier (excerpt)**

```
# BrewUp Stock Management Domain Carrier

## Purpose

Stock Management manages product availability and stock reservation for BrewUp
orders.

It does not own the customer order lifecycle.
It does not confirm orders.
It does not authorize payments.
It does not create shipments.

## Bounded Context Rules (condensed)

- Stock Management owns stock availability and stock reservation decisions.
- Stock Management does not own the Order aggregate.
- Stock Management does not decide whether an order is confirmed.
- Stock Management emits stock-related events that other contexts may observe.
- Order lifecycle events are external inputs, not internal Stock Management
events.

## Open Questions

- How long should a stock reservation remain valid?
- Can stock be partially reserved?
- Should a failed reservation distinguish between unavailable stock and invalid
product?
- Which context decides whether to retry a failed reservation?

## Forbidden Assumptions
```

```
- Do not assume that stock reservation confirms the order.
- Do not assume that payment authorization guarantees stock availability.
- Do not introduce shipment concerns inside Stock Management.
- Do not model Order as an aggregate owned by Stock Management.
```

This artifact is intentionally incomplete. Its job is to preserve what the team already knows, keep uncertainty visible, and prevent future interactions from silently changing the meaning of the context.

The carrier is deliberately compact and uses only the sections described earlier.

The practical difference between prompting and encoding: prompting asks the model to continue from the current sentence; encoding shapes the world in which that continuation is allowed to happen.

Putting the carrier to work

Suppose you later ask the model to classify a new list of elements. The test is whether the carrier keeps owned elements distinct from external inputs and preserves unresolved questions rather than resolving them by assumption: **Classification prompt**

```
Context:
Use the following domain carrier as the only source of truth for the BrewUp Stock
Management bounded context.

[Insert domain carrier here]

Task:
Classify the following elements:

- ReserveStock
- OrderConfirmed
- StockReserved
- PaymentAuthorized
- StockReservationExpired
- ShipmentCreated

For each element, classify it as:
- owned command
- internal event
- external input
- rejected concept
- open question
```

```
Return YAML with:
- element
- classification
- reason
- rule_applied
```

Without the carrier, the model might give a fluent but weak answer. The model might treat `OrderConfirmed` as part of the same flow, include `PaymentAuthorized` because payment and stock often appear together in e-commerce examples, or accept `ShipmentCreated` because shipment usually follows order confirmation.

With the carrier in place, the answer should be narrower. `ReserveStock` is an owned command. `StockReserved` and `StockReservationExpired` are internal events. `PaymentAuthorized` and `ShipmentCreated` should be rejected, because the carrier's forbidden assumptions explicitly keep payment and shipment concerns outside Stock Management. `OrderConfirmed` is a different case. It is an order-lifecycle fact, and Stock Management already consumes order life cycle facts: `OrderSubmitted`, `OrderCancelled`, and `OrderExpired` are all listed as external inputs. It is not an alien concept but a plausible, undeclared input. Because the carrier does not list it, the safest classification is an open question. The model might suspect that Stock Management should react to it, but the carrier does not allow that relationship to be invented.

The artifact has not handed the model a verdict; it has constrained the model from inventing one.

The model does not retain the domain across sessions; the artifact reintroduces that domain knowledge into the context window.

The regression matters because AI-assisted modeling often fails by regression. A team corrects the model once, then the correction disappears in the next session. A concept rejected yesterday returns under a different name today. An open question becomes an invented policy because the model tries to complete the structure. A bounded context boundary slowly weakens because the prompt does not preserve the decisions that created it.

The domain carrier reduces this regression by giving the team a concrete place to capture corrections, allowing each one to become part of the working context rather than remaining buried in a past conversation.

Every modeling decision should trigger a simple question: does this belong in the carrier?

A useful workflow is:

```
Generate or review a candidate model
Critique it against domain knowledge
Promote corrections into rules
Update the domain carrier
Reuse the carrier in the next interaction
```

The last step is essential: without updating the carrier, every correction remains local to the conversation; with the carrier, corrections accumulate.

This also changes where prompts should live. If they carry domain knowledge, they should not remain only in chat history. Chat history is a fragile place for maintained domain knowledge because it is difficult to review, diff, share with the team, and preserve. A domain carrier should live where design knowledge can be maintained, in the location given earlier: beside specifications, decision records, tests, or other modeling artifacts.

What matters is:

If the artifact influences reasoning, it should be visible.

If it is reused, it should be versioned.

If it changes, the change should be reviewable.

The same discipline you already apply to source code and architectural decisions. AI does not remove that discipline; it expands the set of artifacts that deserve it.

None of this is tied to a particular tool, for reasons the next section takes up in detail. What matters here is that file names, agent conventions, and specification formats will keep changing, which is exactly why the knowledge has to live somewhere that does not:

```
Domain knowledge must be encoded before it can be reused.
```

A prompt becomes valuable over time when it carries the domain knowledge that future answers must respect. So far the Stock Management carrier has been asked to do one thing: judge whether an element belongs to the context. Classifying elements is the smallest job it can do, and it is not the reason to build one. The rest of the chapter changes the task while leaving the carrier intact – first deriving a feature specification, then reviewing what has been derived, and finally folding a team decision back into the model. A carrier that survives a change of task is what makes the effort of maintaining it worthwhile.

Reusing the carrier across workflows

So far the carrier has done a single job: classifying elements against what the context already owns. Its value, though, is that the job can change while the carrier does not. The task might move from classification to specification, from specification to review, from review to implementation planning, and through all of them the carrier remains the stable input.

A minimal prompt can look harmless: *A prompt without the carrier*

```
Generate a specification for stock reservation in BrewUp.
```

The instructions are short, but too much is missing. The prompt does not define the bounded context. The prompt does not say which concepts Stock Management owns. The prompt does not protect Order Management, Payment, or shipment from being pulled into the same model, and it gives the model no reason to preserve open questions. The assistant might still generate a polished feature specification, and that apparent completeness is precisely the risk.

A better interaction starts with the Stock Management domain carrier built in the previous section. The carrier gives the model a smaller world to operate in. The prompt no longer asks the model to invent the domain; instead, it asks the model to reuse the domain knowledge already captured.

At this point, other files start to live beside the carrier in the repository:

```
/specs/stock-management/reserve-stock.spec.md
/ai/workflows/stock-management.md
```

A framework might generate its own directories, and the layout matters less than the direction of dependency: these files derive from the carrier, never the other way around.

Treat frameworks as adapters rather than the source of truth.

A BMAD-like workflow might use the carrier as shared context for agents that reason about analysis, architecture, or implementation planning. A SpecKit-like workflow might use the same carrier to derive a feature specification before code generates. An OpenSpec-like workflow might use it to keep proposed changes explicit before they become tasks. The surface changes while the domain knowledge should not. All three are real and open source, and the references at the end of this chapter point to them. Here, however, they represent categories of workflow rather than products. The argument applies to any tool that organizes agents, specifications, or change proposals in these shapes.

The useful comparison is between the roles the carrier can play in different workflows, summarized in *Table 7.1*:

Workflow role	Consumes from carrier	Produces
Agent workflow	Purpose, vocabulary, ownership rules, exclusions	Shared agent context
Spec workflow	Policies, events, commands, open questions	Feature specification
Change workflow	Current model, proposed behavior, unresolved decisions	Change proposal and review tasks

Table 7.1 – Roles of the domain carrier in different workflows

Table 7.1 fixes the scope of this chapter - *Figure 7.1*, a few pages further on, does something different, showing a single carrier rule stopping a single answer. You are not designing a multi-agent system yet, and you are not asking a coding assistant to implement a feature; both come later. What you are doing here is preparing the artifact that those workflows will consume, so that the meaning of the domain survives the handover.

The derive–review–update workflow

The artifacts in *Table 7.1* come from the same refinement loop you ran in *Chapter 6*, now made concrete with three prompts: derive, review, and update. The three form one cycle, and each pass through it leaves the carrier more precise than it was. The first prompt derives a specification from the carrier and is deliberately implementation-neutral.

Derive prompt

```
Context:
Use the BrewUp Stock Management domain carrier as the only source of truth.

Task:
Derive an implementation-neutral feature specification for stock reservation.

Rules:
- Do not invent missing policies.
- Preserve open questions.
- Keep Stock Management separate from Order Management, Payment, and Shipment.
- Treat OrderSubmitted, OrderCancelled, and OrderExpired as external inputs.
```

```
- Use confirmed policies and confirmed internal events as specification material.
- Mark proposed behavior separately.

Return Markdown with:
- Feature purpose
- In scope
- Out of scope
- Confirmed rules
- Proposed rules
- Open questions
```

The following shortened output is worth reading for its structure before its content: the sections mirror the carrier, and the two headings that matter most show what the model left out of scope and what it refused to settle.

Model output: the derived specification (shortened)

```
# Stock Reservation Feature Specification

## Purpose

Allow Stock Management to reserve product stock for submitted BrewUp orders.

## In Scope

- Receive OrderSubmitted as an external input.
- Check whether enough stock is available.
- Reserve stock when the requested quantity is available.
- Emit StockReserved after a successful reservation.
- Emit StockReservationFailed when stock cannot be reserved.

## Out of Scope

- Confirming the order.
- Authorizing payment.
- Creating shipments.
- Modeling Order as an aggregate owned by Stock Management.
- Deciding how Order Management reacts to a failed reservation.

## Confirmed Rules
```

```
- Stock can be reserved only if enough quantity is available.
- Reserved stock reduces the quantity available for other reservations.
- Released stock becomes available again.

## Proposed Rules

- A stock reservation may expire after a defined reservation window.

## Open Questions

- How long should a stock reservation remain valid?
- Can stock be partially reserved?
- Should a failed reservation distinguish between unavailable stock and invalid
product?
- Which context decides whether to retry a failed reservation?
```

The value of this output is what the specification refuses to do, and every refusal can be traced back to a named section of the carrier - three of them to lines printed in the excerpt above. Confirming the order, authorizing payment, and creating shipments are out of scope because forbidden assumptions rules them out by name, one line each. Expiration stays marked as proposed because the reservation window appears under Proposed Policies in the full file and under Open Questions in the excerpt, never under a confirmed rule. Partial reservation stays open for the same reason: the second of the four open questions asks whether stock can be partially reserved, and the carrier does not answer it. The reaction to a failed reservation is pushed out of scope because Bounded Context Rules assigns the order life cycle to Order Management. Nothing here is invented; everything is either carried over from the carrier or explicitly deferred by it.

The same discipline holds outside the text the model returns. Where *Table 7.1* listed what different workflows take from the carrier, *Figure 7.1* shows one of them at work inside a spec-driven tool: instead of deciding whether a failed reservation should distinguish insufficient quantity from an invalid product, the tool surfaces the unresolved question and hands the decision back to the team.

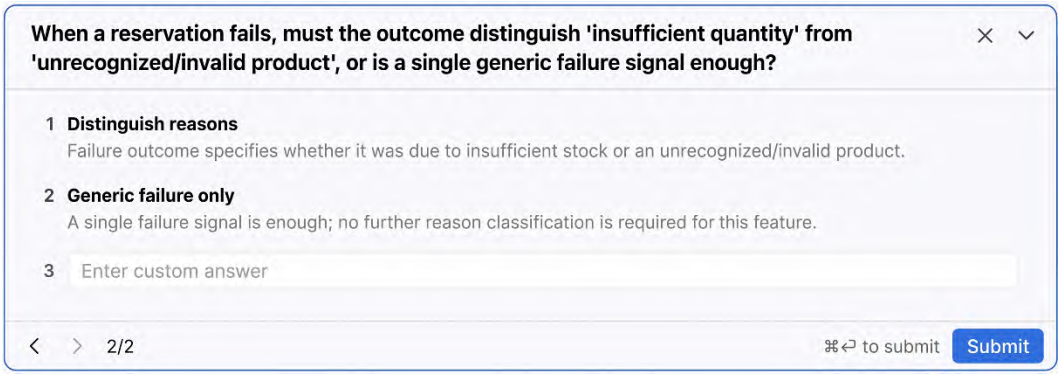


Figure 7.1 – An open question preserved by the domain carrier is surfaced to the team instead of being answered by the model

A domain carrier earns its place here. The model is still generating text, but the carrier decides which text counts as grounded. The team should treat a specification that looks complete but violates the carrier as a failed translation.

The second prompt checks that translation and asks the model to review the specification against the carrier instead of improving it creatively. Like the derive prompt before it, it names the carrier rather than reproducing it: in a real session, the carrier the team supplies through one of the four mechanisms described earlier.

Review prompt

```
Context:
You are reviewing a feature specification derived from the
BrewUp Stock Management domain carrier.

Task:
Check whether the specification preserves the carrier.

Review for:
1. Invented policies
2. Concepts assigned to the wrong bounded context
3. External inputs treated as internal events
4. Proposed rules promoted to confirmed rules
5. Open questions silently answered
6. Forbidden assumptions violated

Return Markdown with:
```

```
- Accepted parts
- Violations
- Corrections required
- Open questions to preserve

Do not improve the specification creatively.
Only review it against the carrier.
```

That last instruction matters. Without it, the assistant might try to help by adding missing behavior. In early design work, that help is expensive. It hides the difference between known domain knowledge and plausible completion. Review prompts should make the model more conservative, not more imaginative.

The third prompt updates the carrier after the team makes a real decision. This closes the loop. The specification must not become a side branch of knowledge. Decisions discovered during review must flow back into the carrier; otherwise, the next workflow starts from stale context.

Update prompt

```
Context:
The team has reviewed the BrewUp Stock Management specification.

New domain decision:
[Insert decision here]

Task:
Update the domain carrier.

Rules:
- Add confirmed decisions only to confirmed sections.
- Move proposed concepts only if the decision validates them.
- Keep unresolved points as open questions.
- Add forbidden assumptions if the decision rejects a recurring mistake.
- Do not remove existing rules unless the new decision explicitly replaces them.

Return:
- Sections to update
- Exact text to add or change
- Reason for each change
```

Suppose the team decides that stock reservations expire after fifteen minutes unless the order is confirmed. That decision should not remain buried in a meeting note or in the last assistant

response. The carrier should be updated: the proposed reservation window becomes a confirmed policy, `StockReservationExpired` remains an internal event, and future specifications can rely on the rule. If the team decides that partial reservation is not allowed, the rejected behavior should also be captured. Rejected behavior is domain knowledge too.

Nothing in that loop required a framework. You ran derive, review, and update as three prompts and one decision, and the carrier absorbed the result. Frameworks enter at this point not to add a step but to automate the three you already have: the cycle can be wrapped in an agent-oriented, specification-driven, or change-driven workflow, and a framework consumes the carrier the same way the three prompts did. That is also the limit of what it can do for you: the carrier is what keeps the workflow from drifting away from the domain.

Tip

OpenSpec and lightweight change discipline

OpenSpec-like workflows are useful when the team wants proposed changes visible before implementation. The tool-specific command or directory is secondary to the discipline: a change should have a proposal, a specification, design notes when needed, and reviewable tasks. The Stock Management carrier feeds that workflow by separating confirmed knowledge, proposed behavior, and open questions, so unresolved domain decisions do not become implementation requirements.

There is a small trap here. Once a framework gives you a clean workflow, it *becomes tempting to trust the workflow more than the domain*. That reverses responsibility. The framework can organize the work, but it cannot validate the business meaning. If the carrier is weak, the workflow will automate weak reasoning. If the carrier is precise, the workflow has something worth preserving.

This also explains why the carrier should stay tool-agnostic. The carrier should not contain Breakthrough Method for Agile AI-Driven Development (BMAD) commands, `SpecKit` syntax, or `OpenSpec` folder conventions unless the team has a strong reason to do so. Those details belong to adapters, examples, or companion repository material, which is why the domain sections listed earlier come first in the file and the harness-specific instructions come last, where they can be swapped without touching the domain. The preceding domain sections are the part the team cannot afford to lose.

Everything so far has pushed in one direction: getting the domain into the prompt. The rest of the chapter turns around and reads what comes back out. Once the carrier has been translated into a specification or workflow artifact, the question changes from whether the model

produced something useful to whether the output holds up when inspected against the domain carrier, and past that, to what the output reveals about the carrier itself.

Reading AI outputs as reflections of understanding

At this stage, the role of the prompt has changed. The model is no longer inside a generic request-response interaction. It is working within a constrained reasoning environment shaped by the carrier.

This also changes how you should read the outputs themselves.

After working with AI for a while, a curious pattern starts to emerge: sometimes, the most valuable part of an answer is not the answer itself.

Suppose you ask the model to generate a specification for the BrewUp Stock Management context. You provide the domain carrier, define the ownership boundaries, and explicitly state that Stock Management does not deal with shipments.

Yet the generated specification contains a rule like this:

```
Create a shipment after stock has been reserved.
```

At first glance, this looks like a simple mistake.

The natural reaction is to correct the specification and move on. However, there is a more interesting question:

Why did the model generate that rule in the first place?

Shipment creation is a common step in many e-commerce workflows. From the model's perspective, stock reservations and shipping frequently appear together. If the bounded context boundary does not express itself strongly enough, the model naturally reconnects concepts that statistically belong together.

The generated rule therefore provides a useful diagnostic signal. The weakness it exposes is in the boundary you drew, not only in the answer.

Read a generated artifact as a diagnostic signal as well as a deliverable. A recurring mistake often reveals a missing constraint, an ambiguous ownership rule, an undefined concept, or an open question that the model is trying to complete automatically. Seen from this perspective, the output becomes a reflection of the current state of understanding.

Correctness is only the first question; the deeper one is what the answer reveals about the model's interpretation of the domain.

This is why fluent AI output can be dangerous. Fluency creates an illusion of understanding. A model can produce grammatically correct, architecturally familiar, and technically plausible structures while still misunderstanding ownership, context, or business meaning.

This is exactly what happened in the first BrewUp Order Management iterations. Notice how payment, stock, and order concepts appear together in a model that initially looks reasonable:

- PaymentAuthorized
- StockReserved
- ConfirmOrder
- Payment
- StockReservation

The sequence resembled a typical e-commerce workflow, and the terminology matched common DDD conventions. Yet the output silently collapsed multiple bounded contexts into the same semantic space. The syntax was not the problem; the output had collapsed the domain's ownership boundaries.

To read outputs this way, it helps to be precise about what is happening underneath.

LLMs do not possess a stable internal domain model that improves through reflection. Each request is reconstructed by the model from the current context window, the prompt, the examples, the rulebook, the domain carrier, and the probability distributions learned during training. The continuity a team perceives comes from the reasoning environment maintained around the interaction, not from persistent domain memory inside the model.

The generated artifact therefore becomes a mirror of the reasoning environment you constructed.

A review should look for this. Suppose the assistant generates this policy:

```
Automatically retry stock reservation three times before failing the order.
```

Whether retrying is a good idea is the second question. The first is:

Where did this assumption come from?

The domain carrier might never have clarified retry behavior; the prompt might have asked for resilience without defining ownership. Previous examples may have contained retry logic, or the model may simply be completing a familiar microservice pattern.

Whatever the cause, the output exposes a gap in the reasoning environment. Read this way, each recurring mistake can point to a gap in the reasoning environment.

Shipment concepts that keep appearing inside Stock Management might point to shipment responsibilities that were never explicitly excluded. Cancellation rules that surface on their own suggest open questions insufficiently protected from completion pressure. An assistant that repeatedly merges external inputs with internal domain events suggests the CQRS role separation is still too weak or inconsistently expressed.

Every recurring mistake tells you something about the quality of the context you have built. That is what makes AI-generated output worth reading closely: not as a verdict on the model, but as a measurement of the environment you put it in. An environment that can be measured can be improved. What happens when a team keeps improving it, session after session, is the subject of the rest of this chapter.

Building shared semantic alignment between human and AI

After enough iterations, the interaction begins to stabilize, and the same concepts start to hold the same meaning across sessions.

It rarely begins that way. In the early stages of a project, interactions with the model often feel unpredictable. The same domain concept might be interpreted in different ways from one session to the next. A bounded context boundary that seemed clear yesterday suddenly becomes blurry again. Concepts that were explicitly rejected reappear under different names.

The model might appear helpful while the interaction remains fragile. After several rounds of refinement, however, the behavior can become more stable.

Suppose you return to the BrewUp Order Management context after days or even weeks of modeling work. You provide the latest domain carrier and ask the assistant to review a proposed change.

The response might now preserve distinctions that previously drifted. The assistant no longer suggests Payment as an aggregate. It no longer treats `PaymentAuthorized` as an internal event. It no longer proposes shipment-related behavior inside Order Management.

When cancellation rules are unclear, it does not invent them. The assistant leaves them as open questions. The underlying model has not changed; the same LLM is still running.

What has changed is the interaction. At this point, many teams describe the experience using expressions such as:

The AI understands your architecture. Or *the assistant knows your domain.*

Those statements are understandable, but they are not literally true. The model does not possess a mental model in the human sense. The model has no enduring conceptual

understanding, no intentionality, and no persistent awareness of the domain across conversations.

What the team is experiencing is something different. The interaction has become stable enough that the same concepts tend to preserve the same meaning across iterations.

Shared semantic alignment is not something the AI holds internally. It emerges through the interaction, sustained by domain understanding, structured context, reusable constraints, stable vocabulary, feedback loops, and iterative refinement.

This distinction matters because the stability comes from the maintained interaction, not from persistent understanding inside the model.

If semantic alignment were something the AI retained internally, the problem would be solved by choosing a smarter model. However, the stability you observe comes from a different source. The stability comes from the discipline used to maintain context over time.

Together, the domain carrier, ownership rules, vocabulary definitions, review prompts, and corrections promoted into reusable constraints create an environment. That environment makes certain interpretations more likely and others increasingly difficult to generate.

In many ways, this resembles how human teams build shared understanding.

A Ubiquitous Language does not appear on the first day of a project. It emerges through conversation, disagreement, clarification, correction, and repeated usage. Over time, the team learns which interpretations belong inside the bounded context and which do not.

AI-assisted modeling follows a similar pattern, but with an important difference: human beings internalize meaning, while the model does not.

Without durable artifacts that carry the agreed language and constraints forward, the alignment can decay between sessions.

A domain carrier acts as an externalized memory for collaboration. Without it, every new session risks partially rebuilding the domain from scratch. Ownership boundaries weaken, rejected concepts return, and open questions slowly transform into assumptions.

With it, the interaction becomes cumulative. Each iteration starts where the previous one ended. This also explains why the assistant can become more useful for specific review tasks over time.

The assistant becomes better at reviewing specifications, identifying inconsistencies, spotting ownership violations, classifying concepts, and proposing refinements that remain aligned with the bounded context.

This improvement comes not from permanent domain learning, but from an interaction that has become constrained enough to preserve meaning consistently.

This marks a practical threshold in AI-assisted software design: without durable context, prompts behave more like fresh conversations; with it, each refinement can strengthen the next interaction.

At that point, the architect's role changes as well. The architect is no longer only reviewing generated artifacts. The architect becomes the curator of the shared semantic environment.

Protecting ownership boundaries, preserving uncertainty, stabilizing vocabulary, capturing rejected concepts, and maintaining the domain carrier become just as important as reviewing the outputs themselves.

This is also why Domain-Driven Design earns a different value in the AI era. Bounded contexts, Ubiquitous Language, ownership, aggregates, and domain vocabulary are no longer only tools for organizing software.

They become a mechanism for shaping machine reasoning. A bounded context no longer aligns only with people. It also aligns interactions. Once that happens, prompts stop being disposable instructions: they become part of the architecture of understanding itself.

Summary

In this chapter, you moved beyond prompting as a way to generate answers and explored how prompts can evolve into artifacts that preserve domain knowledge across interactions.

Starting from the bounded context snapshot created through iterative refinement, you introduced the concept of the domain carrier: a prompt that carries the sections described in this chapter, so the same domain knowledge can be reused consistently over time.

Using the BrewUp Stock Management context, you saw how a domain carrier helps prevent semantic regression, keeps uncertainty visible, and provides a stable foundation for AI-assisted reasoning. Rather than asking the model to rediscover the domain in every conversation, you progressively captured domain knowledge in a form that could be reviewed, refined, versioned, and reused.

You also explored a different way of interpreting AI outputs. Generated artifacts are more than answers to evaluate. They are reflections of how the model interprets the domain under the constraints you provide. Mistakes, omissions, and unexpected suggestions often reveal gaps in the reasoning environment more clearly than correct answers do.

Finally, you examined how repeated refinement gradually creates shared semantic alignment between humans and AI. That alignment does not exist inside the LLM itself. It emerges through the interaction, sustained by domain language, feedback loops, ownership boundaries, and domain carriers that preserve meaning across time.

In this chapter, you learned how to:

- Distinguish between disposable prompts and reusable domain carriers
- Transform bounded context snapshots into artifacts that preserve domain knowledge
- Encode ownership rules, policies, constraints, and open questions into reusable prompts
- Use domain carriers to reduce semantic drift and regression across interactions
- Interpret AI outputs as reflections of understanding rather than proof of understanding
- Recognize how shared semantic alignment emerges through repeated refinement
- Understand why preserved knowledge is the foundation for more advanced AI collaboration models

In the next chapter, you will build on this foundation and move from reusable knowledge to reusable collaborators, exploring how multiple specialized AI agents can work together within clearly defined responsibilities and cognitive boundaries.

References

- BMAD Code (2025). BMAD-METHOD: Breakthrough Method for Agile AI-Driven Development. `https://github.com/bmad-code-org/BMAD-METHOD` (accessed September 2026).
- GitHub (2025). Spec Kit: a toolkit for spec-driven development. `https://github.com/github/spec-kit` (accessed September 2026).
- Fission AI (2025). OpenSpec: spec-driven development for AI coding assistants. `https://github.com/Fission-AI/OpenSpec` (accessed September 2026).

Part 3

Building Agents to Streamline the Interaction

In this final part of the book, you will move from prompts to specialized agents and an orchestrator, creating scalable and consistent AI-assisted domain modeling workflows. You will define agent boundaries, coordinate DDD specialists, and govern autonomy and alignment while keeping evidence, uncertainty, and human decision authority visible.

This part of the book includes the following chapters:

- *Chapter 8*, *From Prompts to Agents: Structuring AI Collaboration*
- *Chapter 9*, *From Guardrails to Specialists: Designing Agents for DDD*
- *Chapter 10*, *The Orchestrator: Building the Coordinating Agent*
- *Chapter 11*, *Governed Autonomy: Balancing Autonomy and Alignment Across Agents*

8

From Prompts to Agents: Structuring AI Collaboration

Chapter 7 ended with a Stock Management domain carrier: an artifact that preserves the purpose of the context, the concepts it owns, the policies already confirmed, the policies still proposed, the open questions, and the assumptions that must remain forbidden. In this chapter, you stop treating that carrier as a standalone prompt and start using it as input to structured AI collaboration.

That carrier applied to a single context. Order confirmation crosses three contexts: Sales makes the commercial commitment, Payment authorizes the transaction, and Warehouse holds the beer. This chapter therefore works from a carrier of the same shape written for the Sales Order Confirmation feature. The discipline is the same one developed in *Chapter 7*; only the boundary it must describe is wider.

To make that shift visible, we follow the same deliberately ambiguous Sales Order Confirmation **feature request** through four increasingly governed environments: prompt-only implementation, **specification-driven development**, the same artifact workflow surrounded by **guard agents**, and a role-based workflow built around explicit responsibilities and handoffs. The comparison asks a practical question: how much domain authority survives as the feature request moves through specifications, plans, tasks, roles, tests, and code?

Later in the chapter, you call the review points that force this distinction cognitive boundaries: places where model inference must yield to explicit domain authority.

Note

Frameworks as experimental lenses

This chapter uses GitHub Spec Kit and the **Breakthrough method for Agile AI-Assisted Development (BMAD)** Method as two concrete ways of organizing AI-assisted development. Spec Kit structures the work as a sequence of durable artifacts – specification, plan, tasks, and implementation – so that each transformation can be inspected before it reaches the code. BMAD distributes planning and delivery across specialized roles, such as analyst, product, architect, developer, and reviewer, making responsibilities and handoffs explicit. The first three runs retain an artifact-centered workflow while progressively adding governance; the fourth changes the collaboration model itself. You do not need either tool installed to follow this chapter; the artifacts they produced are committed to the companion repository, as the Technical requirements section explains.

Neither framework is presented as the source of the DDD principles discussed in this book, and the chapter does not compare them as competing products. You hold the feature request and domain rules constant, then observe where domain knowledge travels, who can challenge an unsupported decision, and what evidence survives each transformation. The principles under examination remain framework-independent: bounded contexts retain authority over their decisions, external outcomes require explicit provenance, completion claims require behavioral evidence, and human approval cannot be delegated to a workflow.

In this chapter, you will learn:

- Why DDD-shaped code can still fabricate evidence when decision provenance is implicit.
- How a domain carrier becomes a set of reviewable constraints in a feature specification.
- How to **trace** one domain promise from that specification into plans, tasks, tests, and implementation.
- How guard agents create cognitive boundaries without replacing domain experts or behavioral proof.
- Why changing frameworks redistributes roles and handoffs but does not remove the need for explicit authority, evidence, and human approval.

Technical requirements

This chapter is a guided reading of a repository. Every artifact and every code excerpt discussed here is preserved at `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design`, and each of the four runs has its own branch. *Table 8.1* maps the branches to the runs and names what to read on each one.

Branch	Run	What to read
`no_sdd`	Run 1 – prompt only	`SalesOrder.cs`, `SalesOrderSaga.cs` and `RequestBeerAvailablityRaisedEventHandler.cs` under `src/`
`sdd_without_custom_agents`	Run 2 – Spec Kit without guards	`spec.md`, `plan.md`, `research.md`, and `tasks.md`, under `specs/001-order-confirmation/`
`sdd-with-harness`	Run 3 – Spec Kit with guard agents	the guard definitions in `.github/agents/speckit.brewup.*.agent.md` and the authoritative sources in `.specify/memory/`
`bmad-solution-with-harness`	Run 4 – BMAD, governed	`project-context.md`, `planning-artifacts/` and `implementation-artifacts/`, under `_bmad-output/`
`bmad-solution-without-harness`	Run 4 – BMAD, overlay removed	the BMAD installation only. This branch preserves no planning artifacts and no implementation, so it cannot support a workflow comparison

Table 8.1 – Where each run is preserved in the companion repository

You can follow the whole chapter by reading these files on GitHub, and nothing needs to be installed to do so

If you prefer to have the code in front of you, clone the repository and check out one branch per run:

```
git clone https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design.git
cd AI-Augmented-Domain-Driven-Design
git checkout no_sdd
```

Building the solution in `src/BrewUp.slnx` requires the .NET 10 SDK, but that is optional: the chapter never asks you to execute the code, only to inspect what it does and what its tests actually prove.

GitHub Spec Kit and the BMAD Method are the two workflows compared in this chapter, and you do not need either one installed. Their outputs – specifications, plans, task lists, product briefs, and review reports – are committed as ordinary Markdown files on the branches listed previously. Install them only if you want to reproduce a run yourself.

From a plausible prompt to a fragile implementation

You began with a deliberately simple baseline run. You gave the AI assistant a short feature request and asked it to implement the change directly, without first creating a specification, identifying decision ownership, or introducing review gates:

```
Implement order confirmation for BrewUp.

An order can be confirmed when:
- the customer payment has been authorized;
- all requested beers are available in the warehouse.

When the order is confirmed, reserve the stock.
```

That short text is the **feature request**. Everywhere in this chapter, *the request* means that text, and never an HTTP request or a message on the bus. The request appears precise because it names two conditions and an action. Yet it leaves the important domain decisions unstated: which bounded context owns payment authorization, whether availability is a durable fact, when stock must be reserved, and even what kind of Order is being confirmed. There is enough information to generate code, but not enough to preserve the domain model.

This baseline run is the first of the four listed in *Table 8.1*. You left those ambiguities unresolved and asked the assistant to go straight from the feature request to working code in the repository. The domain carrier existed, but the assistant received no instruction to consult it, no specification step, and no review gate. The resulting implementation is preserved on the

`no_sdd` branch of the companion repository. You call each preserved branch a **trace**: the artifacts and code a run left behind, readable afterward even though the conversation that produced them cannot be replayed. This first trace records a workflow in which domain knowledge was available but never brought into the decision path.

The generated code initially appears disciplined. It introduces `PaymentAuthorizationId` and `StockReservationId`, adds a `ConfirmSalesOrder` command, raises `SalesOrderConfirmed`, and prevents the Sales Order from becoming Confirmed when either reference is empty:

```
if (string.IsNullOrWhiteSpace(paymentAuthorizationId.Value))
    throw new ArgumentException(
        "Payment authorization id is required.",
        nameof(paymentAuthorizationId));

if (string.IsNullOrWhiteSpace(stockReservationId.Value))
    throw new ArgumentException(
        "Stock reservation id is required.",
        nameof(stockReservationId));

RaiseEvent(new SalesOrderConfirmed(
    new SalesOrderId(Id.Value),
    paymentAuthorizationId,
    stockReservationId,
    correlationId));
```

Viewed in isolation, this method appears aligned with DDD: Sales owns the transition to Confirmed, the invariant is explicit, and a domain event represents the state change. The method proves that two references are present. The method does not prove that the bounded contexts that own those decisions produced them.

Evidence that no authority produced

The decisive problem appears in the Sales Order saga. When the existing workflow reports that the customer's budget has been verified, the generated code constructs a payment authorization reference from the message correlation identifier:

```
internal void MarkCustomerBudgetAsVerified(
    CustomerJson customer,
    Guid correlationId)
{
    var paymentAuthorizationId =
        new PaymentAuthorizationId(correlationId.ToString());
```

```
    RaiseEvent(new SagaCustomerBudgetVerified(
        new CustomerId(_customerId),
        correlationId,
        paymentAuthorizationId,
        customer,
        ...));
}
```

The value satisfies the aggregate guard, but it was not produced by Payment. A correlation identifier connects messages within a technical conversation; it does not represent authorization of an amount. By assigning it the type `PaymentAuthorizationId`, the implementation turns customer budget verification into fabricated payment evidence.

The Warehouse half of the same generated implementation fails in a different form. The `RequestBeerAvailablityRaisedEventHandler` checks a snapshot of availability, queues one reservation command per requested row, and then publishes the outcome:

```
StockReservationId stockReservationId =
    new(Guid.CreateVersion7().ToString());
// [...] one ReserveItemStock command per requested row
if (canReserveAllRows)
{
    foreach (var command in reserveCommands)
        await serviceBus.SendAsync(command, cancellationToken);
}
RequestBeersAvailabilityChecked integrationEvent = new(
    new IntegrationId(@event.AggregateId.Value),
    correlationId,
    canReserveAllRows ? stockReservationId : null,
    beerAvailability);
await eventBus.PublishAsync(integrationEvent, cancellationToken);
```

The identifier is minted inside the handler, before any reservation exists, and `SendAsync` only queues the commands: the event carrying `StockReservationId` is published without waiting for a single `ReserveItemStock` to succeed. Sales therefore receives an identifier before Warehouse has produced the reservation outcome on which confirmation depends. Availability might change; a completed stock reservation is the durable evidence.

This failure makes the baseline run useful. The code contains explicit types, reasonable guards, domain events, tests, and a Warehouse-owned mutation path. Those correct-looking elements

conceal the real defect: the workflow preserved the shape of the evidence while losing the authority that must produce it.

From findings to durable rules

A DDD review of this branch should produce durable domain knowledge rather than comments that disappear after review. If the next interaction receives only the original feature request, the model is free to make the same assumptions again. The review therefore produces candidate rules for the domain carrier:

```
- Payment owns payment authorization outcomes.
- Sales may store PaymentAuthorizationId only as evidence of an outcome produced
by Payment.
- A correlation identifier is not a payment authorization reference.
- Customer budget verification must not be interpreted as payment authorization.
- Warehouse owns stock availability and stock reservation.
- Availability is not durable evidence; confirmation requires StockReservationId.
- The coordinator coordinates external decisions but does not own or reproduce
them.
```

These rules become the cumulative baseline for every run that follows. The specification-driven workflow must carry them from the feature specification into plans and tasks; the guard-agent environment must use them to challenge unsupported transformations between artifacts, and the role-based workflow must preserve them as the work passes between analyst, product, architecture, development, and review roles.

A longer prompt might prevent the assistant from constructing `PaymentAuthorizationId` from a correlation identifier in one interaction. Durable rules serve a different purpose: they keep the decision available for review at every subsequent stage. The next run therefore turns the original request and these rules into a specification whose requirements, open questions, and later projections can be inspected before implementation begins.

To follow this yourself, check out the `no_sdd` branch in the companion repository and read the following three files as a single evidence path rather than as independent implementations. In `SalesOrder.cs`, observe what evidence the aggregate requires before confirmation. In `SalesOrderSaga.cs`, trace `PaymentAuthorizationId` back to the event and value from which it is constructed. In `RequestBeerAvailablityRaisedEventHandler.cs`, compare the moment

`StockReservationId` is published with the moment the reservation commands can actually produce their outcomes:

```
src/Sales/BrewUp.Sales.Domain/Entities/SalesOrder.cs
src/Sagas/BrewUp.Sagas.Domain/Entities/SalesOrderSaga.cs
src/Warehouse/BrewUp.Warehouse.Facade/Acl/
RequestBeerAvailablityRaisedEventHandler.cs
```

Read together, the files reveal what a local code review can miss. The Sales aggregate consumes plausible references, but their provenance is decided earlier in the workflow. One is manufactured outside Payment, while the other is published before Warehouse has completed the reservation. The next run must therefore make ownership, provenance, and ordering reviewable before those assumptions reach the implementation.

Turning domain knowledge into a specification

The prompt-only run left the feature request free to fill its own gaps. For the second run, you kept that feature request unchanged and altered only the environment in which the assistant interpreted it. On the `sdd_without_custom_agents` branch, the workflow could consult the BrewUp constitution and the Sales Order Confirmation domain carrier before producing a specification:

```
/speckit.specify <the feature request above, unchanged>
```

The command is the standard Spec Kit entry point: it turns the quoted request into a `spec.md` that can be reviewed before any code exists. If you had rewritten the request as a detailed solution, you could not tell whether the improvement came from the workflow or from a human performing the modeling in advance. The ambiguity remains; what changes is that it must now pass through durable domain knowledge before it can become code.

Keep the purpose of each artifact in view as the workflow progresses. The constitution defines repository-wide engineering constraints. The domain carrier records the feature's decision authorities, accepted domain meanings, and unresolved policies. The specification transforms the request and those sources into reviewable requirements and open questions. Clarification records human decisions back into the specification; the plan later projects accepted decisions onto the solution; and the task list turns that projection into claims that implementation and tests must demonstrate. Each artifact answers a different question and gives the team another point at which to review the decision.

In this run, the constitution contributed constraints, such as modular bounded contexts, domain purity, test-first development, and architecture fitness functions. The carrier

contributed the feature-specific rules: Sales, Payment, and Warehouse remain separate decision authorities; payment authorization and stock reservation are external outcomes; Sales must not manufacture either one; and availability is not durable evidence for confirmation. The constitution constrains the engineering system, while the carrier constrains the meaning of the feature.

What the specification does to the feature request

Spec Kit did not replace either artifact. It transformed their relevant content into a feature specification that could be reviewed before planning. The generated `spec.md` opened with a section named Domain Constraints (Authoritative). Among its rules were the following.

Read the excerpt below as a provenance map. The first group assigns each decision to the bounded context that owns it; the second distinguishes transient availability from a durable reservation outcome, and the final lines identify the external references that Sales might retain without taking ownership of the decisions they represent.

```
Three separate domain authorities:
Sales - commercial commitment
Payment - payment authorization
Warehouse - physical stock

Availability is not a durable fact.
Only a Stock Reservation is durable enough to support
Sales Order confirmation.

Confirmed requires evidence:
PaymentAuthorizationId and StockReservationId.
```

The specification uses those statements to reinterpret the original wording. *Table 8.2* shows the actual semantic work performed between the prompt and `spec.md`:

Original wording	Specification transformation	DDD effect
Order	Sales Order, owned by Sales	Removes the generic aggregate and fixes lifecycle authority.
Payment has been authorized	`PaymentAuthorizationId` produced by Payment	Turns a vague condition into evidence with explicit provenance.

Original wording	Specification transformation	DDD effect
Beers are available	`StockReservationId` produced by Warehouse	Replaces a transient observation with a durable outcome.
Reserve the stock when confirmed	Reservation outcome is required before confirmation completes	Prevents Sales from committing against stale availability.
Unspecified failure behavior	Open questions and out-of-scope policies	Stops retries, refunds, voids, and notifications from being invented.

Table 8.2 – How the specification transforms the original request into explicit DDD constraints

The most consequential change concerns time. The original sentence makes availability sound like a precondition and reservation, including an action that follows confirmation. The carrier rejects that ordering. Availability can become stale; a reservation is durable evidence. The specification therefore requires the Warehouse outcome before the Sales Order completes its transition to Confirmed. The specification does more than clarify the wording: it corrects the domain model while the correction is still inexpensive.

The specification then converts the invariant into user scenarios, acceptance scenarios, functional requirements, and measurable outcomes. For example, its primary scenario requires both external references before confirmation:

```
Given a Sales Order awaiting confirmation that holds
a valid PaymentAuthorizationId and StockReservationId,
When confirmation is attempted,
Then the Sales Order status becomes Confirmed
and both references remain recorded as evidence.
```

This scenario is useful because it can be challenged without opening the code. Whichever role you hold, you can reach it: a domain expert can ask where the references came from, an architect can check that they cross explicit integration boundaries, a developer can derive aggregate tests from the invariant, and a reviewer can reject any later task that asks Sales to authorize payment or decrement stock. The specification has turned the carrier into a review surface.

Clarification and what it makes durable

The first specification pass also surfaced questions that the prompt had hidden. You then ran the clarification step and supplied three domain decisions:

- If payment is authorized but stock cannot be reserved, the Sales Order remains in its pre-confirmation status.
- The existing Sales Order saga or process manager coordinates the confirmation flow.
- Any release or void of the payment authorization belongs entirely to Payment; Sales takes no compensating action.

Clarification is valuable because the answers become part of a durable artifact rather than remaining in a conversation. The saga is no longer a convenient mechanism selected during coding. It is the explicitly chosen coordinator. The absence of Sales-owned compensation is no longer an omission. It is a boundary decision: Payment retains authority over the authorization it produced.

Other decisions correctly remained open. The specification still asks whether partial reservation is allowed, how long a reservation might remain active, what a payment-provider timeout means, how long Sales waits for Warehouse, who owns customer notification, whether the order of the two external outcomes matters, and whether pre-approved payment terms can provide alternative evidence. Those questions constrain what the plan is allowed to claim, and erasing them so that planning can begin would only hide the constraint.

At this point, the Spec Kit output is substantially safer than the prompt-only implementation. The specification uses the Ubiquitous Language, identifies decision ownership, records evidence instead of manufacturing it, and preserves unresolved policies. A structured workflow improves inspectability rather than authority: the generated text becomes easier to challenge, but the process itself does not make it more valid. The specification still requires a DDD review.

Figure 8.1 summarizes the workflow used in the specification-driven branch. The same feature request enters a governed context, is clarified by a human, and returns to `spec.md` after DDD review whenever the specification requires correction.

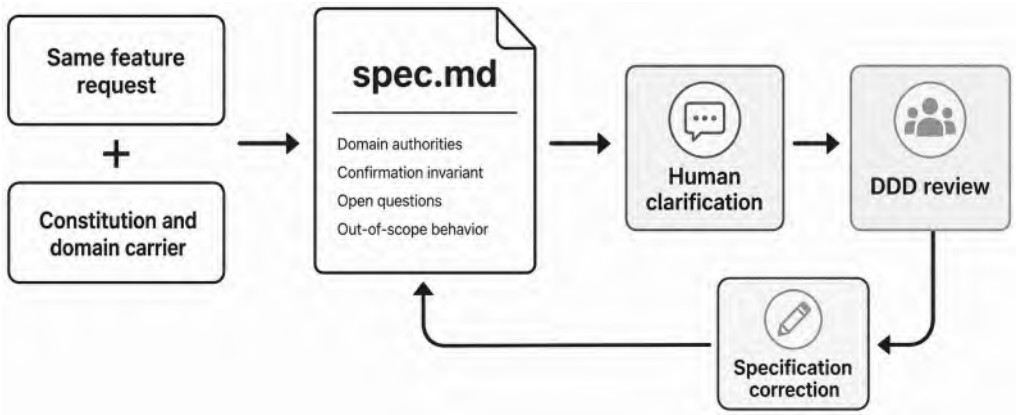


Figure 8.1 – From the same feature request to a reviewable and correctable specification

Reviewing the specification

The specification has made the request reviewable; the DDD review now evaluates the transformation itself. It compares requirements with one another, traces mandatory statements to named sources, and separates accepted domain decisions from plausible policies before planning gives them executable form. This stage therefore contributes more than another ownership check: it can expose contradictions introduced inside an artifact whose individual statements appear reasonable. The first finding is such a contradiction. FR-004 assigns the reservation request to Sales, while FR-012 assigns cross-context coordination to the saga:

```
FR-004: When a Sales Order satisfies the preconditions for
confirmation, the Sales context MUST request a stock
reservation from the Warehouse context through an integration
boundary, rather than reserving, decrementing, or otherwise
mutating warehouse stock itself.

FR-012: The existing Sales Order saga/process manager (Sagas
context) MUST coordinate the confirmation flow across Payment
and Warehouse [...] and MUST NOT own the cross-context
orchestration itself.
```

Each statement is reasonable on its own. Read together, they do not merely leave the sender ambiguous; they disagree. `FR-004` instructs Sales to request the reservation, and `FR-012` forbids Sales from owning the cross-context orchestration in which that request lives. Does Sales send `ReserveStock`, with the saga merely observing outcomes? Or does the saga send it as part of its

coordination responsibility, while Sales receives only the final instruction to confirm? The carrier permits coordination mechanisms but requires their authority to remain explicit. At this stage, the evidence does not establish which participant should send `ReserveStock`. The specification should therefore preserve that ownership question for resolution before planning rather than silently choosing one interpretation.

The second finding concerns idempotency. The generated specification introduced this requirement:

```
FR-009: Confirmation MUST be idempotent: re-processing the
confirmation trigger for an already-Confirmed Sales Order
MUST NOT produce a second confirmation outcome nor a
duplicate stock-reservation request.
```

Idempotency is a sensible property in a message-driven workflow, and the proposal might well be accepted. It is not, however, present in the feature request or domain carrier. The assistant promoted a plausible engineering expectation into a mandatory functional requirement. A DDD review should not delete it merely because the model added it, nor accept it merely because it is a good idea. The DDD review should classify it as a proposed policy, ask which component owns duplicate detection, and confirm what counts as the same confirmation attempt before downstream tasks treat it as settled.

The requirements checklist did not catch either problem. The checklist reports that all items pass, that all requirements are unambiguous, and that the specification is ready for planning. At the same time, `spec.md` remains marked Draft and preserves questions that can change the interaction design. The checklist is still worth running, as long as you read it for what it proves: required sections exist, acceptance scenarios are present, and named carrier constraints appear in the document – not that every requirement has valid provenance, or that two individually plausible requirements agree about decision ownership.

The DDD review therefore produces actions, not code:

- Clarify whether Sales or the saga sends the stock-reservation request, and align every scenario and requirement with that decision.
- Move idempotency into proposed policies until its owner, identity rule, and expected behavior are explicitly accepted.
- Keep unresolved business policies visible and prevent the plan from implementing work that depends on them.
- Treat the generated checklist as evidence of structural completeness, not as domain approval.

The review leaves the specification in a more honest state. Requirements supported by the carrier can move forward, plausible additions such as idempotency remain visible as proposals, and contradictory responsibilities identify the human decision still required. The prompt-only run concealed its boundary violation inside plausible code; the specification-driven run exposes the disagreement before architecture and tasks propagate it. Planning can now begin from explicit decisions and explicit gaps rather than from the checklist's appearance of completeness.

From specification to plan and tasks

With the specification reviewed and its unresolved decisions made visible, the workflow now moves from validated understanding to executable work. That transition raises the cost of every remaining ambiguity. A requirement in `spec.md` can still be challenged while it is only a sentence; the plan assigns it to modules, aggregates, messages, and tests, while the task list turns those assignments into work that a developer or coding agent can execute. Once an unsupported assumption crosses this boundary, it is no longer one questionable line in a specification. The assumption becomes distributed across files, dependencies, test expectations, and completion checkboxes. Planning is therefore more than an administrative handoff: it is the point at which domain meaning begins to acquire an executable shape.

You continued on the `sdd_without_custom_agents` branch with the specification produced in the previous step. The first command asked Spec Kit to plan the feature against the existing BrewUp solution:

```
/speckit.plan
```

The command produced more than `plan.md`. It inspected the repository and created a small design package under `specs/001-order-confirmation`:

- `research.md` records existing-system findings, design decisions, rejected alternatives, and carrier compliance.
- `data-model.md` projects the invariant into aggregate state and behavior.
- contracts/ defines commands, domain events, and integration events at the context boundaries.
- `quickstart.md` translates the scenarios into build and test expectations.
- `plan.md` ties those artifacts to the actual BrewUp modules and file structure.

What the plan decides

Planning maps accepted domain meaning onto concrete parts of the solution. The plan records where Payment and Warehouse evidence enter the workflow and which component coordinates their arrival. It also records where the confirmation invariant is enforced and which producer behavior remains outside the slice. Its Structure Decision captures that projection explicitly:

```
Structure Decision: [...] The confirmation invariant lives in
BrewUp.Sales.Domain (SalesOrder); orchestration lives in
BrewUp.Sagas.Domain (SalesOrderSaga + orchestrator); all
cross-context messages live in BrewUp.Shared. Warehouse-side
production of the reservation outcome and any Payment-side
production are out of scope and are reached only through the
shared contracts.
```

That projection preserves several important DDD choices. Sales owns the transition to Confirmed. Payment and Warehouse remain the authorities that produce the two external outcomes. The saga coordinates their arrival but does not recreate either decision. The Sales Order stores `PaymentAuthorizationId` and `StockReservationId` as evidence rather than embedding foreign models. A developer can now point to a module, a message, and an aggregate method for every part of the invariant.

The design package makes the chain reviewable in both directions. Read the following rows of *Table 8.3* from left to right: a source claim becomes a design decision or a task, and the DDD review says what that projection preserved or lost.

Source claim	Projection into plan and tasks	DDD review
Confirmation requires `PaymentAuthorizationId` and `StockReservationId`.	SalesOrder.Confirm owns the invariant; `T007-T010` create the aggregate test, behavior, and handler.	Preserved. Sales records evidence and owns only its lifecycle transition.
The saga coordinates Payment and Warehouse outcomes.	`D4` assigns coordination to `SalesOrderSaga`; `T018-T020` add state and reactions.	Direction is explicit, but the implemented message path must still match the plan.

Source claim	Projection into plan and tasks	DDD review
Warehouse owns stock reservation.	`T016` creates `ReserveStock` as a contract; the Warehouse handler is declared out of scope.	Boundary ownership is preserved, but the requested reservation behavior is not delivered.
Confirmation should be idempotent.	`D5` makes it a design decision; `T008`, `T009`, and `T019` implement duplicate suppression.	A proposed policy has become mandatory without an accepted owner and identity rule.
Both outcomes should lead to exactly one confirmation.	`T017` promises an orchestrator behavior test and `T024` claims end-to-end validation.	Completion must be proved by behavior, not inferred from checked task boxes.

Table 8.3 – Traceability and DDD review as the specification becomes executable work

Now, read the same rows from right to left, and ask whether each piece of executable work traces back to an accepted requirement, a named authority, and an honest feature scope. The last two rows deserve closer attention: the downstream artifacts look increasingly complete while the underlying policy or behavioral evidence remains unsettled.

Traceability, and what it does not prove

Two rows show the central limitation of **traceability**. The workflow traces idempotency consistently from FR-009 into design decision D5 and the generated tests and guards. Yet the domain never defined what constitutes the same confirmation attempt or who owns that identity rule. The workflow has therefore amplified a plausible proposal before it was accepted. The appropriate response is to return to the specification, classify the policy as proposed, obtain the missing domain decision, and only then amend the plan.

Stock reservation exposes the inverse problem. Here, the carrier is explicit: Warehouse must produce the durable `StockReservationId` required before confirmation. Nevertheless, `research.md` and `plan.md` exclude the Warehouse-side producer, and task T016 stops at a message contract:

```
T016 Create the Warehouse-owned reservation request contract
ReserveStock in [...] carrying WarehouseId, the Sales Order
```

```
reference, requested rows, and correlationId (contract only -
Warehouse-side handler is out of scope per research D2).
```

That seam allows Sales and the saga to consume a synthetic `StockReserved` outcome, but it does not demonstrate that Warehouse actually reserves the requested rows or produces the reference on which Sales depends. The implemented slice therefore demonstrates Sales Order confirmation from external decision evidence, not the complete producer-to-consumer path. Traceability makes the missing Warehouse-side evidence visible; it does not resolve the gap. The next question is whether the checked tasks actually prove the behavior they claim to deliver.

When a checked task is not a completed task

After planning, you generated the executable breakdown:

```
/speckit.tasks
```

The resulting `tasks.md` divides the work into 25 small, ordered tasks with named files, dependencies, parallel opportunities, and test-first expectations. That granularity makes implementation easier to coordinate and review, but it can also make every checked box look like evidence that a specific behavior has been delivered.

On `sdd_without_custom_agents`, the workflow marks every task except T022 as complete. T017, for example, promises tests showing that the two external outcomes cause exactly one `ConfirmSalesOrder`, that either outcome alone causes none, and that failure prevents confirmation. The corresponding test file contains five facts, and they all have this shape:

```
public class SalesOrderSagaConfirmationTests
{
    [Fact]
    public void PaymentAuthorized_CorrelationId_MustBe_Preserved()
    {
        var correlationId = Guid.CreateVersion7();
        SagaPaymentAuthorized @event = new(
            new IntegrationId(correlationId.ToString()),
            correlationId,
            Guid.CreateVersion7().ToString());

        Assert.Equal(correlationId,
            MessageHelpers.GetCorrelationId(@event));
    }
```

```
    // [...] four more facts of the same shape, for
    // PaymentAuthorizationFailed, StockReserved,
    // StockReservationFailed, SalesOrderReadyToConfirm
}
```

To verify the gap directly, inspect `SalesOrderSagaConfirmationTests.cs` and look for a test that constructs or rehydrates the `SalesOrderSaga`, drives it with both external outcomes, and verifies that exactly one `ConfirmSalesOrder` is emitted. There is none. Nor is there a test that drives only one outcome or a failure outcome and asserts that confirmation is not emitted. The file therefore proves message-shape preservation, not the behavior promised by T017. Yet T017 is checked: the presence of a test file and valid assertions has been treated as completion even though the behavior named by the task remains unproved.

T019 exposes the same distinction from another direction. The task describes `ReserveStock` being dispatched and exactly one `ConfirmSalesOrder` being sent, while the implementation retains `ReserveStock` only as a contract and reaches Sales through a different message path. The code might still support a viable architecture, but the checked task no longer describes what the code does.

Treating each checkbox as a behavioral claim makes both mismatches visible. Completion requires an observable result at the boundary named by the task; file creation, compilation, and narrow unit assertions are supporting evidence, not substitutes for that result.

For this trace, the DDD review produces the following concrete actions:

- Decide whether the slice includes the real Warehouse reservation producer; align the feature claim with that decision.
- Make `plan.md`, `T019`, and the implemented saga-to-Sales message path describe the same coordination mechanism.
- Replace the correlation-only saga tests with behavioral tests for both arrival orders, exactly once confirmation, and failure outcomes.

These findings do not automatically update the domain carrier. The reservation gap and the task-test mismatch are workflow and verification defects, not new business rules. They belong first in the specification review and engineering checklist. Only a human decision about idempotency, reservation atomicity, or failure semantics can become durable domain knowledge.

To reproduce this review from the preserved artifacts, inspect `plan.md`, `research.md`, `quickstart.md`, and tasks.md under `specs/001-order-confirmation` on the

`sdd_without_custom_agents` branch. Then, compare T016, T017, and T019 with `ReserveStock.cs`, `SalesOrderSagaConfirmationTests.cs`, `SalesOrderSaga.cs`, and `SagaSalesOrderReadyToConfirmIntegrationEventHandler.cs`. The useful exercise is to follow one domain promise until a real test and production path demonstrate it. Doing that here shows the limit of the artifact chain: it records each decision faithfully, but nothing in it argues back. The next run adds something that does.

Adding cognitive boundaries with guard agents

The specification-driven run made domain decisions traceable across specifications, plans, tasks, tests, and code. It also revealed what structured artifacts cannot accomplish by themselves. Once an unsupported policy enters the chain, each subsequent artifact can reproduce it accurately and make it appear increasingly settled. In the previous run, an unconfirmed idempotency rule became executable work. The Warehouse producer disappeared from scope, while the feature retained its reservation claim and checked tasks overstated the behavior demonstrated by their tests. The workflow made those failures inspectable, but did not ensure that they would be challenged before propagating.

For the third run, you changed only the review environment: a harness of specialized guard agents introduced explicit review points between generative transitions. They inspected whether each new artifact still cited the accepted domain authority, preserved unresolved decisions, and respected the architectural boundaries established by the preceding one. The value of this environment is timely challenge: an unsupported assumption can surface: where it enters the workflow, before later artifacts normalize it as settled work.

What a cognitive boundary constrains

Persistent knowledge is necessary, but persistence alone does not enforce it. A generated specification, plan, or task list can reinterpret, weaken, or quietly ignore the constitution, the carrier, and architecture memory while remaining coherent enough to look correct. The harness brings those passive sources into the review path and uses them as explicit criteria. You call the resulting checkpoints cognitive boundaries.

The term is deliberately parallel to bounded context, and the two must not be confused. A bounded context constrains the meaning and ownership of a domain model: where a term is valid, which model gives it meaning, which authority may make the corresponding decisions. A **cognitive boundary** constrains what an AI participant is allowed to infer, decide, or transform at a particular point in the workflow. A guard does not own the Sales Order life cycle, authorize a payment, or reserve stock. It verifies that generated artifacts do not hand those decisions to the wrong participant or silently resolve policies the domain has deliberately left

open. A guard may stop the workflow; it cannot complete the domain model on behalf of its owners.

Two kinds of rule: ownership and structure

The harness separates two kinds of authoritative rule. Bounded-context rules answer the question of who owns a decision. Architecture rules answer the question of where that ownership has to live in the solution.

The domain carrier contains numbered bounded-context rules such as:

```
BC-003 - Payment owns authorization outcomes.

BC-006 - Stock Reservation is an external decision.
Sales may store StockReservationId as evidence that Warehouse
produced a reservation outcome. [...] Sales must not produce
the stock reservation outcome.

BC-009 - External decision references, not embedded models.
Sales may store external decision references [...] Sales must
not embed Payment or Warehouse domain models.
```

Architecture memory complements those rules with structural constraints:

```
AR-001 - Bounded contexts must be explicit modules.

AR-002 - Standard module project structure.

AR-016 - Business authority must not be collapsed into
another module.
```

Used together, the two rule sets let a guard examine the same artifact along two connected dimensions. Bounded-context rules test whether a decision still belongs to the context that owns it; architecture rules test whether modules, messages, and dependencies preserve that separation in the solution. A requirement might correctly name Payment as the authority while a plan places payment behavior inside Sales. Conversely, a separate Payment module cannot legitimize a requirement that allows Sales to manufacture an authorization outcome. The review succeeds only when domain meaning and structural representation agree.

The guard can therefore report a conflict against named sources rather than an undefined idea of good architecture. A guard can identify a sentence in `spec.md` that violates BC-003, a project layout in `plan.md` that violates AR-001 or AR-016, or a combination in which the requirement

is conceptually sound but its proposed implementation collapses the boundary. Ownership rules protect the decision; architecture rules protect the system shape that keeps its ownership visible.

Where the guards sit in the life cycle

Figure 8.2 shows the harness surrounding the artifact chain. The same feature request first meets the constitution, domain carrier, and architecture memory. Guards then inspect the transitions from specification to plan, from plan to tasks, and from tasks toward implementation. When a guard finds an unsupported decision, the workflow should return to human clarification or artifact correction instead of silently continuing. The BLOCK arrows show the intended response to an accepted finding.

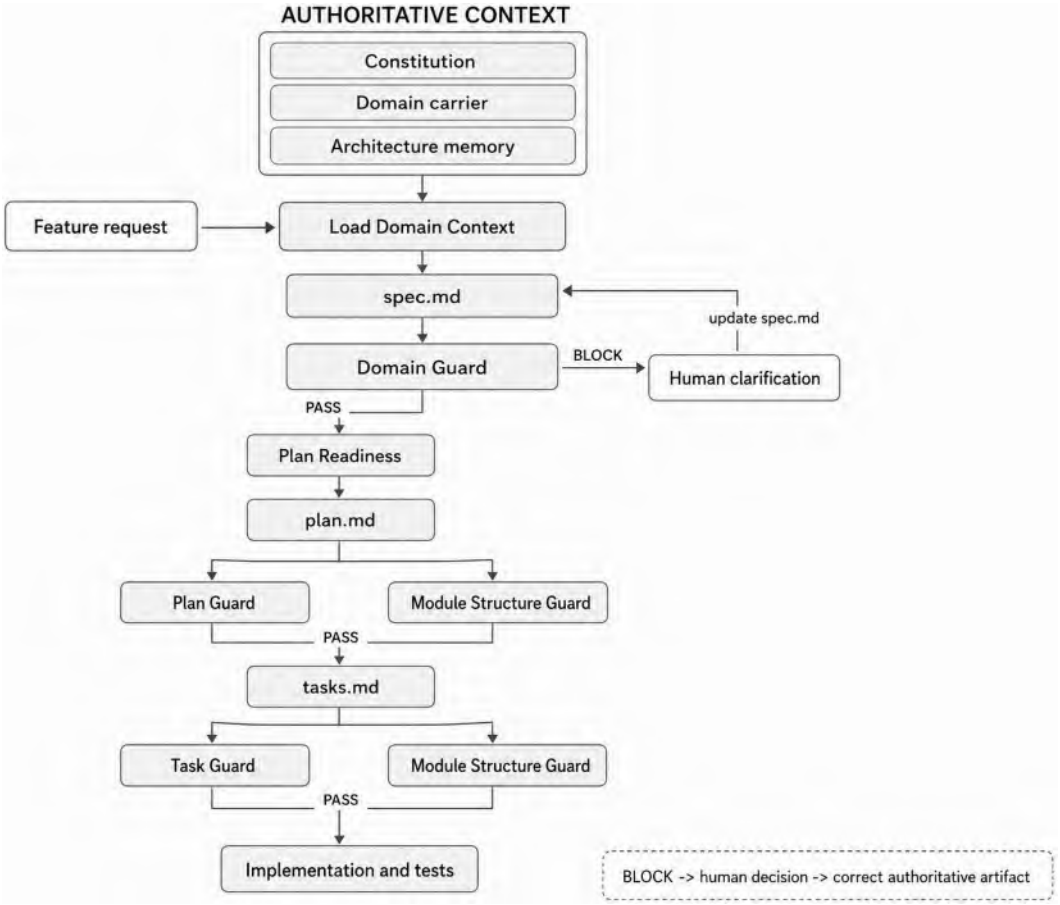


Figure 8.2 – Guard agents create cognitive boundaries between specification-driven artifacts

Each generative transition introduces a new opportunity for drift, so each one deserves a cognitive boundary of its own. The domain context is loaded before the request becomes a specification, and a guard then inspects every artifact the workflow derives from it.

In the BrewUp repository, these checks are wired into the Spec Kit life cycle on the sdd-with-harness branch. One agent loads the authoritative context before specification begins; the remaining guards inspect the artifacts produced at later transitions. Read *Table 8.4* from top to bottom as a map of where domain knowledge can be lost: first when the request becomes a specification, then when requirements become architecture, and finally when the plan becomes executable tasks. The extension configuration offers the hooks automatically at those transitions, but each hook remains optional.

Agent	When it runs	What it checks
`load-domain-context`	Before specification	Loads the carrier, the constitution, and architecture memory into the context of the request
`domain-guard`	After specification	Ownership across Sales, Payment, and Warehouse, and the provenance of every policy
`plan-readiness`	Before planning	Whether an open business question has been silently resolved
`plan-guard`	After planning	Architectural drift in the way responsibilities are projected onto components
`module-structure-guard`	After planning	Module layout against the AR rules held in architecture memory
`task-guard`	After task generation	Ownership and unresolved policy inside the generated tasks

Table 8.4 – The harness agents and the transitions they inspect

The sequence distributes one review question across the life cycle: has the latest transformation preserved the authority and unresolved decisions present in its sources? load-domain-context establishes the baseline; domain-guard checks the first interpretation of the request; the planning guards examine its architectural projection; and task-guard verifies the

final translation into executable claims. Later guards do not repeat the earlier review. They test whether an accepted decision survived a new type of transformation. The idempotency requirement provides a concrete example of how that distributed review changes what the workflow can see.

A guard in action: the idempotency requirement

The idempotency requirement from the previous run shows what changes in practice. The standard workflow introduced the following rule:

```
Confirmation MUST be idempotent: re-processing the confirmation
trigger for an already-Confirmed Sales Order MUST NOT produce
a second confirmation outcome nor a duplicate stock-reservation
request.
```

The requirement is technically reasonable, but that does not give it domain authority. Neither the feature request nor the domain carrier defines the identity of a confirmation attempt, assigns duplicate detection, or determines how repeated external outcomes should be treated. A domain guard must expose that missing provenance and identify the decision required before the workflow continues. The guard must not invent the identity rule or decide that idempotency is unnecessary.

The preserved artifacts contain no blocking report, and FR-009 remains a mandatory requirement. The comparison nevertheless reveals a meaningful change. The specification produced under the harness foregrounds domain authorities, cites authoritative rules beside most requirements, and records unresolved policies in a numbered section with named owners. FR-009 still has no source, but that absence is now visible among requirements that do cite one. The cognitive boundary has not prevented the unsupported rule from entering; it has made that rule distinguishable from accepted domain knowledge.

The task list shows a second effect. The standard run excluded the Warehouse producer while retaining the claim that the feature reserved stock. Under the harness, the reservation behavior becomes explicit work after a human clarification addresses partial reservation. Yet the resulting specification still contains conflicting partial and all-or-nothing scenarios. Provenance records who supplied a decision; it does not automatically reconcile every sentence derived from it. The responsible domain authority must resolve the conflict before planning continues.

Nor do the additional tasks prove that reservation is behaviorally complete. The preserved implementation and tests still leave open whether reservation remains correct across multiple requested beers, repeated attempts, state restoration, and the complete Warehouse-to-Sales

path. These gaps are verification findings. A guard can expose the claims that require proof, but it cannot supply that proof by inspecting artifacts.

The evidence boundary matters here because no run log records a guard rejecting the earlier scope cut. The sequence therefore cannot be replayed, and you cannot claim that the harness caused the change. The narrower comparison is still useful: the environment that made open questions explicit and assigned them to an authority is also the environment in which partial reservation reached a human decision and returned as planned work.

Structural guards apply the same discipline to a different kind of drift. If a plan recognizes Payment as the authorization authority but places its command and handler inside Sales, BC-003 identifies the ownership violation, and AR-016 identifies the structural collapse. At task level, Sales may be asked to store `PaymentAuthorizationId` as external evidence; the task must not ask Sales to authorize payment. Guarding the transition exposes the conflict before that assumption spreads across contracts, tests, and production code.

What a guard must not do

A guard should not silently rewrite an artifact so that it looks compliant; that would make it one more unaccountable decision-maker. Its job is to cite the accepted rule, show the conflicting evidence, and name the artifact or the human decision that must change before the workflow continues. A BLOCKED result means that the next step should not proceed; it does not mean that the guard is necessarily right. A human may reject the finding, and the value is that the rejection becomes an explicit decision instead of disappearing into the next generated output. Every hook is optional, so this is a process rule and not a technical guarantee.

The architect therefore remains essential: someone must decide which sources are authoritative and which findings need a domain expert rather than a technical fix. A guard also enforces only what has been made explicit. If the carrier says nothing about reservation atomicity, no guard knows which partial-reservation behavior is correct. Before you add one, ask which accepted rule it will be able to cite: a guard with nothing to quote is an opinion with a hook. Behavioral proof the team still requires: a checked box is not evidence that the implementation satisfies the condition its task names.

The harness does not close the behavioral proof either. The preserved tests do not demonstrate both arrival orders, exactly-once confirmation, rejection handling, state restoration, or the complete path from Warehouse outcomes through the saga to Sales. Artifact provenance can identify what must be tested; it cannot substitute for observing the promised behavior.

Without a run log, what remains reproducible is the artifact comparison itself. On the `sdd-with-harness` branch, inspect the guard definitions in `.github/agents/speckit.brewup.*.agent.md`, the structural rules in `.specify/memory/architecture/brewup-module-structure.md`, and the wiring in `brewup-spec-kit-harness.md`. Then, take

any requirement in `specs/001-order-confirmation/spec.md` and ask which rule authorizes it. The document either answers or it does not, and that is the reproducible test.

Controlled delegation

The third run makes delegation more controlled and reviewable. The AI might still generate each artifact, while the harness inserts reviews intended to challenge unsupported policy promotion, expose decisions moved across bounded-context boundaries, and flag tasks that collapse an external authority into Sales. That distinction is the practical meaning of a cognitive boundary: it marks where model inference should yield to explicit authority and where a human decision becomes necessary.

Guard agents are the first form of specialization you have used, and their role is deliberately defensive. They inspect transformations, expose unsupported decisions, and give the team a chance to stop architectural drift before it propagates.

The distinction can be summarized in three lines. A prompt describes what the model should attempt; a specification records what subsequent work must preserve; a guard agent verifies whether the next transformation is still authorized. The harness does not make the AI a better architect, and optional hooks cannot prevent it from silently acting as one. What the harness can do is make unsupported decisions visible enough to be challenged before they propagate.

In short, the third run changed when an unsupported decision becomes visible, while leaving the question of who may make it exactly where it was. The next run leaves the artifact chain alone and changes the collaborators instead.

Running the same feature request through a role-based workflow

The first three runs use the same artifact-centered collaboration model: the request became a specification, the specification became a plan, the plan became tasks, and the guards – when present – inspected the transitions between them. The fourth run changes how collaboration is organized and is also the thinnest of the four. Of its two branches, only one preserves usable artifacts, so what follows is a reading of that single trace rather than a fourth result carrying the same weight as the previous three.

In the BMAD run, domain knowledge moves through specialized roles and their handoffs rather than only through a sequence of generated documents. An analyst frames the problem, product roles shape the outcome and scope, an architect projects the decisions onto the system, developers implement the stories, and reviewers challenge completion claims. The comparison therefore asks whether the same domain rules remain visible, contestable, and attributable when responsibility passes between collaborators instead of between artifacts.

The trace the repository actually preserves

The BMAD run has two named branches, but they do not contain equivalent evidence. In the repository state examined for this chapter, the `bmad-solution-without-harness` branch records the BMAD installation and removes the BrewUp governance overlay; it contains no `_bmad-output` planning artifacts, feature implementation, or command transcript. That absence does not show that an ungoverned BMAD workflow fails; it means only that this branch cannot support a workflow comparison.

The usable trace is on `bmad-solution-with-harness`. It starts from the same order-confirmation request and carries the approved BrewUp decisions into a `project-context.md`, then into the product brief, **product requirements document (PRD)**, architecture, epics and stories, implementation-readiness assessment, and the first three implementation stories. The logical sequence preserved by those artifacts is:

```
project context
  -> product brief
  -> PRD
  -> architecture
  -> epics and stories
  -> implementation readiness
  -> Story 1.1: Payment domain and module
  -> Story 1.2: Payment projection and callback
  -> Story 1.3: Warehouse stock reservation
```

The trace is a workflow trace derived from artifacts rather than a replayable command log. No run transcript or tag records the exact commands and conversational handoffs. The files and commit history show which transformations survived; they do not prove every interaction that produced them.

Where the domain knowledge travels

In the BMAD branch, the first durable carrier of domain knowledge is `_bmad-output/project-context.md`. It summarizes the modular-monolith rules, test-first discipline, and the prohibition against inventing unresolved policy, while pointing back to the numbered Sales Order Confirmation and module-structure governance documents. The summary explicitly says that it does not override those sources. This distinction matters: a role receives an operational context, but authority remains with the carrier and the approved domain decisions.

The product brief turns that context into an outcome statement. One short excerpt preserves the boundary that the prompt-only run had violated:

```
Sales stores only PaymentAuthorizationId and StockReservationId
as external evidence. It does not embed Payment or Warehouse models.
```

The PRD expands the statement into requirements. Payment owns authorization, Warehouse owns complete stock reservation, Sales requires both evidence IDs, and the saga coordinates Payment -> Warehouse -> Sales without producing any of those decisions. Architecture decisions bind those requirements to modules and dependency directions. Epics and stories then attach each behavior to named tests, paths, and `BC-###` or `AR-###` rules. Domain knowledge therefore travels through role-shaped artifacts, but it remains reviewable because each handoff leaves a trace to its source.

The missing evidence also matters. The repository does not contain the raw dialogue between analyst, product, architect, and developer roles. The repository therefore cannot prove exactly what each role read or objected to during the run. The repository can prove that the resulting artifacts carry the rules and that the gate reports inspected those artifacts against the committed sources.

Summary

This chapter followed one ambiguous Sales Order Confirmation request through progressively governed environments. The prompt-only run produced plausible DDD-shaped code but fabricated Payment and Warehouse evidence. Spec Kit made carrier rules traceable through specifications, plans, and tasks, while showing that traceability alone cannot prevent unsupported policies or false completion claims. Guard agents added cognitive boundaries that exposed ownership and provenance gaps, but passing checks still required domain judgment and behavioral proof.

The BMAD run redistributed the same knowledge through project context, product brief, PRD, architecture, stories, and review gates. The preserved BMAD trace carries the domain rules into Payment and Warehouse implementation increments, but it stops before the Sales and saga stories needed to demonstrate end-to-end confirmation. Frameworks might redistribute roles and handoffs; authority, evidence, and human approval remain.

Table 8.5 puts the four runs side by side, in the order the chapter presented them.

Run	What it added	What it still could not prove
Prompt only `no_sdd`	nothing beyond the feature request	that either reference had been produced by the context that owns the decision
Spec Kit `sdd_without_custom_agents`	a specification, plan, and task list that cite the carrier and survive the run	that a traced requirement had a legitimate source, or that a checked task had been demonstrated
Spec Kit plus guards `sdd-with-harness`	review points that must name the rule they enforce, and unresolved policies with owners	that reservation works end to end: a guard reads artifacts, not behavior
BMAD `bmad-solution-with-harness`	roles and handoffs that carry the same rules, each traceable to its source	end-to-end confirmation, since the run stopped before the Sales and saga stories

Table 8.5 – The four runs, what each one added, and what each one left unproved

In this chapter, you learned how to:

- Detect evidence that no domain authority produced.
- Carry domain rules from specification to executable work.
- Trace promises through requirements, tests, and production behavior.
- Use guards to expose ownership drift before it propagates.
- Preserve human approval when workflows or frameworks change.

Chapter 9 moves from defensive guards to specialized DDD agents, defining the responsibilities, inputs, outputs, and limits they need to participate in modeling without becoming domain decision-makers.

References

- GitHub (2025). Spec Kit: a toolkit for spec-driven development. `https://github.com/github/spec-kit` (accessed September 2026).
- BMAD Code (2025). BMAD-METHOD: Breakthrough Method for Agile AI Driven Development. `https://github.com/bmad-code-org/BMAD-METHOD` (accessed September 2026).

9

From Guardrails to Specialists: Designing Agents for DDD

The previous chapter showed how guard agents can expose unsupported claims, ownership drift, and missing authority before those defects propagate. However, a guard remains defensive: it can inspect or stop an artifact, not turn messy evidence into candidate facts, scenarios, behavior, or responsibility boundaries.

This chapter makes that productive step explicit. You define four narrowly bounded specialists —**EventStormer**, **Storyteller**, **Command & Event Writer**, and **Context Mapper** —each bounded by a contract and separated from the next by a **human gate**. The governing rule is simple: a specialist transforms; a domain authority decides.

Using a synthetic BrewUp Stock Management walkthrough, you follow the same evidence across the chain and examine the characteristic pressure at each stage: invented policy, narrative closure, convenient commands, and the tendency to produce an overly tidy map. The experiment asks you to judge the specialists by traceability and restraint, while watching whether provenance, status, ownership, uncertainty, and review questions survive every handoff.

In this chapter, you will learn how to:

- Design a specialist around authoritative inputs, one transformation, and explicit prohibitions.
- Separate candidate proposals from modeling material accepted at a human gate.
- Preserve provenance, uncertainty, decision ownership, and review questions across transformations.

- Recognize where fluent facts, stories, behaviors, and maps overstep the evidence.
- Distinguish automatable coordination from domain decisions that still require human authority.

From defensive guards to productive specialists

In *Chapter 8*, you used guard agents to put cognitive boundaries around generated artifacts. A guard inspects material that already exists. In this chapter, you ask an agent to do something harder: to transform evidence into candidate material that did not exist before.

A role title alone does not create that boundary. Specialization requires authoritative inputs, one narrow transformation, inspectable output, explicit non-responsibilities, and a human gate.

A specialist transforms; an authority decides

Transformation adds structure, and the risk lies in that added structure: a sequence implies ordering, a common name can hide disagreement, a command introduces an intention, and a boundary groups decisions. Each move can help a team reason, but it can also make vocabulary, causality, or policy appear to come from the domain.

The output must therefore remain candidate material: concrete enough to challenge, but separate from accepted domain knowledge. An unresolved candidate is often the most accurate result available because it preserves a contradiction rather than smoothing it into policy.

Table 9.1 names the four transformations and the useful work each specialist must refuse to perform.

Specialist	Authoritative input	Transformation	Candidate output	Characteristic prohibition
EventStormer	Raw evidence	Evidence into candidate facts	Facts, triggers, participants	Do not settle ambiguous policy
Storyteller	Accepted facts	Facts into concrete situations	Scenarios and variations	Do not finish beyond the evidence

Specialist	Authoritative input	Transformation	Candidate output	Characteristic prohibition
Command & Event Writer	Accepted scenarios	Scenarios into candidate behavior	Commands, events, rules	Do not invent a convenient instruction
Context Mapper	Accepted material	Behavior into responsibility boundaries	Boundaries and relationships	Do not draw an unsupported connection

Table 9.1 – Four bounded transformations: what each specialist may propose and what it may not decide

Every transformation carries its own pressure to erase uncertainty, and the last column of *Table 9.1* states each one as a prohibition. *Figure 9.1* shows the structural countermeasure: only material a human selects at a gate becomes input to the next stage.

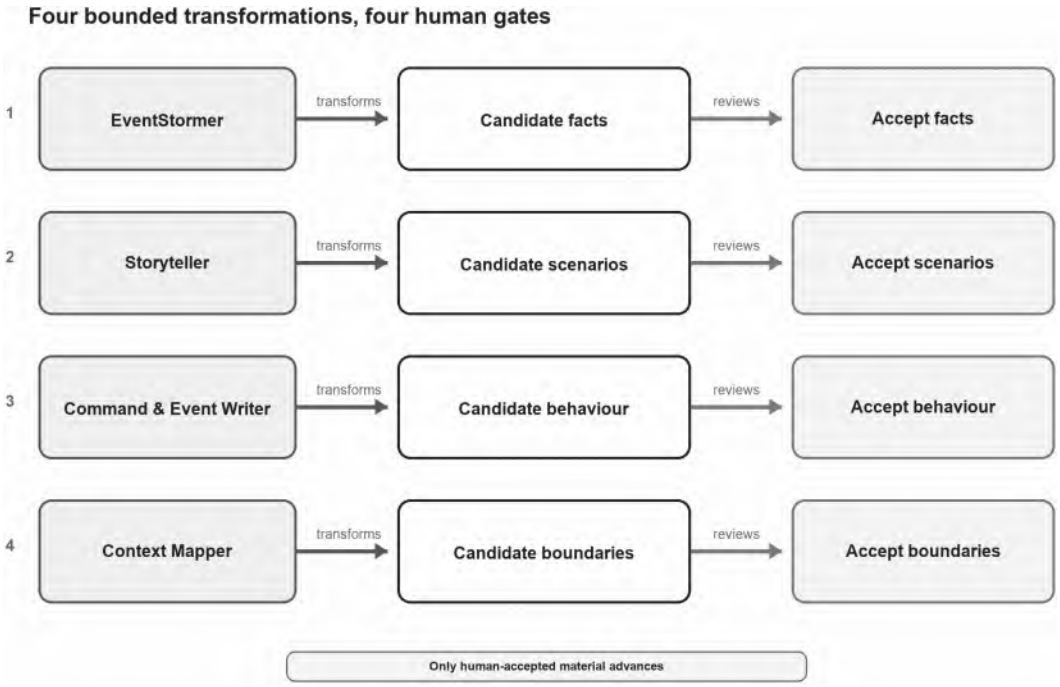


Figure 9.1 – Four bounded transformations separated by human gates

That separation depends on more than assigning each stage to a different specialist; each specialist also needs a clear boundary around what it may receive, produce, and decide.

Contracts and handoffs as cognitive boundaries

Each contract fixes the authoritative input, one transformation, the allowed output kinds, and the decisions the role must not make. That narrowness makes additions attributable: if a policy first appears in a scenario, the team knows exactly where to challenge it.

All four contracts share the same structure, which you can adapt for a specialist of your own:

```
---
description: <one sentence: the transformation, and the decision it must not make>
id_prefix: <two or three letters, unique across specialists>
---

# <Specialist name>

<What this agent is looking at, in the language of the business.>

You perform one transformation and no other: **<input> into <candidate output>.**

## Your inputs
<the authoritative input, plus the evidence, glossary, constitution and schema>

## What to return
<the candidate kinds you allow, one row each>

## <The distinction that matters most>
<the single judgment this specialist gets wrong first>

## Scope
<what must be accounted for, and what goes in unaccounted_sources>

## What you must not do
<the decisions reserved for a human authority>
```

The contract fixes what a specialist may produce. A shared **handoff schema** fixes how the output must be returned, allowing the next reader to judge it without reconstructing the conversation that produced it.

```
candidates:
  - id: <prefix>-01
    kind: <specialist-specific, see your contract>
```

```
    name: <the name you propose, in the language of the business>
    statement: <one sentence saying what this is>
    source: [QUOTE-05, QUOTE-06]
    status: observed | inferred | proposed | unresolved
    # accepted and rejected are unavailable: they record a human decision
    decision_owner: <who must decide this, or "unknown">
    uncertainty: <what would have to be true, and what is missing>
    review_question: <the question a human must answer before this advances>

summary:
  candidates_returned: <n>
  unresolved: <n>
  unaccounted_sources: [<identifiers you could not place, or empty>]
  information_needed: <one or two sentences>
```

Two fields are especially important. The uncertainty field names the evidence or decision still needed, taking the place of a confidence score. The review question turns that gap into work for a named authority. In the summary block, `unaccounted_sources` matters more than any count, because an empty list claims the specialist placed all the evidence, and you check that claim.

One conflict in this schema is worth naming here because it appears twice in the recorded runs. Here, decision_owner names the authority that must decide whether a candidate may advance. Two of the four recorded contracts, the EventStormer's and the Command & Event Writer's, used the same field for causal attribution instead: who brought an outcome about. Both questions matter, and they have different answers, so a single field cannot carry both meanings. You keep the recorded contracts and their output unchanged, because they are the run evidence. A reusable version of either contract should express causal attribution in the statement and the provenance, and reserve decision_owner for the authority. Two contracts out of four reading one field differently is a design finding in its own right, and the following sections record where it surfaced in each run.

ES-17, produced by the EventStormer run, is the record we return to throughout the chapter:

```
- id: ES-17
  kind: trigger
  name: requested quantity was short
  statement: Having less beer available than the requested quantity prompted
either a complete refusal or a partial set-aside, depending on who was working.
  source: [QUOTE-13, QUOTE-15, QUOTE-16, QUOTE-17, QUOTE-18, QUOTE-19]
  status: unresolved
```

```
  decision_owner: unknown
  uncertainty: The short-quantity policy was never written down, and Andrea and
Nadia described different practices without treating either as wrong.
  review_question: Who has authority to decide what must happen
when the available quantity is less than the requested quantity?
```

The evidence does not identify who can decide the short-quantity policy, so `decision_owner` remains unknown. Assigning the nearest plausible participant would close a question the source leaves open.

Notice that neither the contract nor the record uses the terms domain event, aggregate, or bounded context. The omission is deliberate. An agent told to produce bounded contexts is handed the model's shape instead of deriving that shape from the evidence. Every specialist does read constitution-excerpt.md, which states the BrewUp modeling principles plainly: state changes are expressed as domain events, each bounded context speaks its own Ubiquitous Language, and domain ownership must be explicit. The specialist's own instructions stay in the words the brewery used, such as set aside, the board, and the end bay. At the gate, you map the brewery's terms to DDD concepts. In real discovery, domain experts and modelers would then agree on the vocabulary the team will use consistently. *Table 9.2* is the mapping you use when you review.

Candidate kind	**DDD building block a reviewer may recognize**	**Where you met it**
fact	a candidate domain event	*Chapter 4*, EventStorming
trigger	a policy, or an input from outside the context	*Chapter 6*, role separation
participant	an actor or an external system	*Chapter 4*, EventStorming
scenario, variation	a scenario	*Chapter 4*, Domain Storytelling
command	a command	*Chapter 6*
event	a domain event	*Chapter 6*
rule	a policy, or an aggregate invariant	*Chapters 5* and *6*

Table 9.2 – What a specialist proposes, and the DDD building block a reviewer may recognize in it

Tip

Framework lens – BMAD: is a role the same thing as a boundary?

BMAD's named roles help coordinate work, but a brief alone does not limit what a role may decide. A **cognitive boundary** also fixes inputs, one transformation, inspectable outputs, and explicit non-responsibilities. To make a BMAD role cognitively bounded, include that contract in its brief.

The BrewUp experiment and its limits

The experiment returns to Stock Management, the bounded context we wrote a domain carrier for in *Chapter 7*. One naming distinction matters before we begin: Stock Management is the bounded context, while Warehouse, the module you watched produce stock-reservation evidence for order confirmation in *Chapter 8*, is one implementation of it. The evidence here is a synthetic warehouse walkthrough. Nadia and Andrea are fictional composite participants, and the brewery floor is constructed for the experiment.

Every statement in the walkthrough has a stable `QUOTE-nn` or `OBS-nn` identifier. The evidence deliberately leaves several matters open: partial quantities, hold duration, failure reasons, retry ownership, vocabulary variation, and the response to a failed quality check. A citation counts only when the cited passage supports the exact claim.

The visible run pack holds the walkthrough, a reduced glossary, constitution-excerpt.md, the handoff schema, and one specialist contract. The run pack lives under `docs/ch09/run-pack` in the companion repository at `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design`. Four branches, `ch09/9.2-eventstormer` through `ch09/9.5-contextmapper`, carry the frozen protocol and the four specialist checkpoints; each adds one run to the one before it, and ch09/9.5-contextmapper also carries the integrated chain evaluation. The evaluator separately holds the Stock carrier, the evidence inventory, a leak check, the pre-registered hypotheses, and the protocol. *Figure 9.2* shows the separation: the baseline meets the candidate handoff only at evaluation.

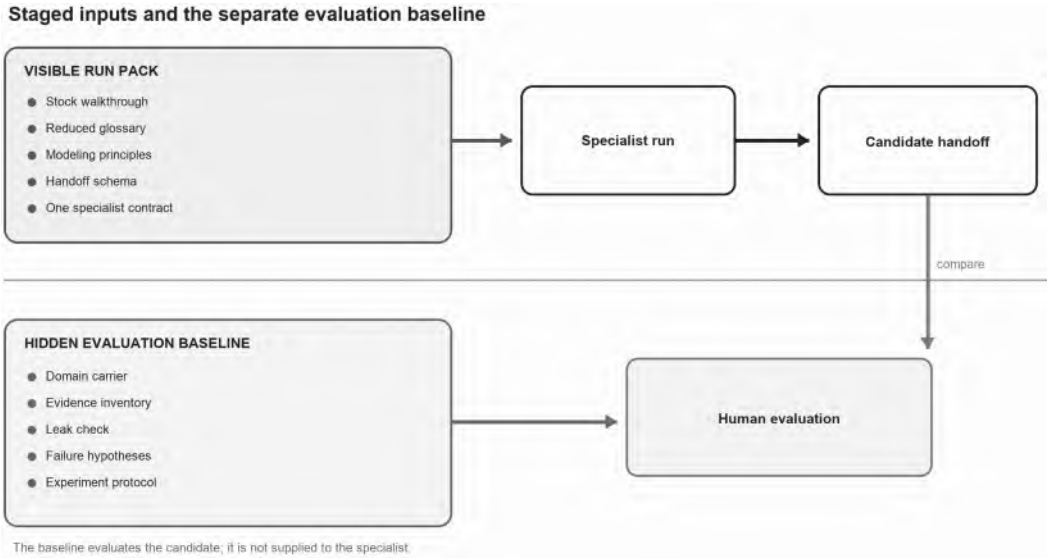


Figure 9.2 – Visible run pack and hidden evaluation baseline

This chapter introduces a working vocabulary and a repository layout at the same time. *Table 9.3* maps each term to what it is and to the file that holds it, so that any claim in the prose can be checked by a reader against the artifact behind it. Paths are relative to the companion repository; the four evaluator-only files are named here but were never staged into a run.

Term	What it is	Where it lives
run pack	The files staged into an empty workspace for one specialist run	`docs/ch09/run-pack/`
evidence (QUOTE-nn, OBS-nn)	The synthetic walkthrough, one identifier per statement or observation	`run-pack/stock-walkthrough.md`
contract	One specialist's instructions: authoritative input, one transformation, prohibitions	`run-pack/contracts/brewup.<specialist>.agent.md`
handoff schema	The YAML shape every specialist must return	`run-pack/handoff-schema.md`

Term	What it is	Where it lives
candidate (ES-, ST-, CEW-, CM-)	One proposed record, prefixed by the specialist that produced it	`each run's raw-output.md`
manifest	The contract, every staged input with its hash, the runtime parameters and the invocation	`each run's input-manifest.yaml`
gate	The recorded human decision: accepted, referred back for correction, or accepted while unresolved	`each run's human-decision.md`
accepted material	The authoritative input to the next specialist	accepted-facts.md, accepted-scenarios.md, accepted-behaviour.md, accepted-context-map.md
evidence inventory (E-nn)	What the walkthrough contains; the baseline for coverage	`docs/ch09/evidence-inventory.md` (evaluator only)
pre-registered hypothesis (FH-nn)	A failure predicted and committed before any run was executed	`docs/ch09/failure-hypotheses.md` (evaluator only)
leak check	The audit that the run pack does not contain the answers the runs measure	`docs/ch09/leak-check.md` (evaluator only)
evaluation baseline	The Stock Management domain carrier, withheld until evaluation	`.specify/memory/domain-carriers/brewup-stock-management.md` (evaluator only)

Term	What it is	Where it lives
checkpoint	A branch holding the frozen protocol and every run up to that point	`ch09/9.2-eventstormer` through `ch09/9.5-contextmapper`

Table 9.3 – The chapter's working vocabulary and where each artifact lives in the companion repository

Table 9.4 records the four specialist runs side by side. The Context Mapper is discussed at greater length under Run 9.5, because its checkpoint also carries the integrated evaluation of the complete chain.

Specialist	Run output	Human gate	What remained open
EventStormer	41 candidates: 24 facts, 8 triggers, 9 participants	39 accepted directly; 2 referred, corrected, then accepted	11 records passed the gate while remaining unresolved; one name had turned an expectation into an occurrence
Storyteller	11 candidates: 7 scenarios, 4 variations	9 accepted directly; 2 referred, corrected, then accepted	10 records passed the gate while remaining unresolved; no missing policy, owner, or ending was supplied
Command & Event Writer	36 candidates: 16 commands, 20 events, 0 rules	All 36 accepted as modeling material	15 records passed the gate while remaining unresolved; no retry owner, hold policy, or invented Stock command

Specialist	Run output	Human gate	What remained open
Context Mapper	18 candidates: 8 boundaries, 10 relationships	15 accepted directly; 3 referred, corrected, then accepted	4 records passed the gate while remaining unresolved; no boundary was named Warehouse, and no delivery, retry, failed-batch, or partial-stock authority was assigned

Table 9.4 – What the four specialist runs produced and what the gates changed

Note

What this experiment can and cannot prove

You built this decomposition as an experiment. Real discovery overlaps and loops, and you keep every handoff manual so a reviewer can see where a proposal entered, who accepted it, and what stayed undecided. Four limits bound what these runs can show:

- *A synthetic fixture.* The walkthrough, its participants, and the brewery floor were written for this experiment. The Stock Management domain carrier is a frozen evaluation baseline known only to the evaluator, so it is neither domain ground truth nor a script for a specialist to reproduce.
- *Staged inputs, not enforced isolation.* Each agent received the files recorded in its manifest. Proving that no other path or network source was reachable would need a recorded enforcement boundary, and you did not build one.
- *An evaluator, not a domain expert.* The authority at every gate is a human evaluator working over a synthetic fixture. No brewery domain expert reviewed these candidates.
- *Reproducible conditions, not reproducible output.* The inputs, contracts, rubric, invocation conditions, and review questions are fixed. Re-execution may well produce different candidates.

Those limits set what the four runs can be asked. Each one can show whether a proposal stayed distinguishable from an accepted domain decision, candidate by candidate, against a

record a reader can challenge. None of them can show that the resulting model is right for a real brewery, because no brewery was asked. With the protocol fixed, you can follow the same evidence through the chain, beginning with the transformation that reads it first.

EventStormer: discovering facts without deciding the domain

The **EventStormer** receives the least processed material in the chain. Its job is to turn statements, observations, vocabulary differences, and contradictions into candidate facts, triggers, participants, and questions that a domain authority can review. It automates the collecting move in EventStorming, the practice we ran by hand in *Chapter 4*, and it inherits that practice's discipline: collect what was said before deciding what it means.

From evidence to candidate facts

Chapter 4 introduced past-tense events and the difference between outcomes and intentions. Here, the contract is narrower: produce candidate facts, triggers, and participants without treating any of them as accepted domain knowledge. Once a human accepts one, a fact becomes a candidate domain event; the reduced glossary the specialist reads is the raw material of a Ubiquitous Language, not the language itself. A past-tense name proves neither who caused the fact nor what should always happen.

The following excerpt from the contract that produced the run shows the transformation, the causal distinction most likely to be lost, and the decisions the specialist must not make:

```
description: Propose the significant things that happen in a business, from raw
interview evidence, without deciding which of them are true of the model.
id_prefix: ES
---

You perform one transformation and no other: **evidence into candidate facts.**

## What to return
| kind        | What it is                                            |
|-------------|-------------------------------------------------------|
| fact        | Something that happened, named in the past tense      |
| trigger     | Something that prompts a fact to happen               |
| participant | A person, group, or thing that takes part             |

## The distinction that matters most
For every `fact`, your `statement` must say whether the people interviewed
described **bringing it about themselves**, or described **being told about it**.
```

```
A group that hears about something, records it, and acts on it has not caused it.
Put the authority that causes a fact in `decision_owner`,
and write `unknown` when the evidence does not say who that is.

## What you must not do
- Do not propose a fact that no cited passage supports.
- Do not name anything the business did not name, unless you say in `uncertainty`
that the name is yours.
- Do not decide, on behalf of the business, anything the evidence leaves open.
- Do not describe how any of this should be built.
- Do not mark anything as accepted or rejected.
```

The EventStormer contract is the first of the two contracts that overload `decision_owner`, the conflict described with the handoff schema above: here the field carries causal attribution for facts rather than the authority that must decide. The excerpt stays unchanged as run evidence.

That distinction gives you a test for every candidate: did Stock bring the outcome about, or was it merely told about it? Nadia moves beer, and Andrea records a quantity. Sales cancels an order, while the lab decides that a batch failed. Stock may react to those external facts without owning them.

When accounts conflict, preserve both. Andrea rejects a short quantity while Nadia separates what is available. The specialist must not average the practices, choose the efficient one, or promote either to policy. It marks the matter as unresolved and asks who may decide it. Each source identifier must point to a passage that supports the statement. An unrelated citation adds no evidence and should be removed.

When discovery becomes invention

The walkthrough supplied 64 addressable pieces of evidence. The challenge was not a lack of evidence, but the mixture of direct observation, second-hand facts, vocabulary variation, and questions that nobody in the interview could settle. The run turned them into the 41 candidates summarized in *Table 9.4*.

ES-17, shown earlier, demonstrates useful restraint: it keeps the two short-quantity practices, cites both accounts, leaves the owner unknown, and asks an authority for a policy. Other records kept Sales cancellations external to Stock and left set aside, hold, and earmark unresolved instead of silently choosing a canonical term.

The run was initially named **ES-24** before more beer arrived, although the evidence established only that replenishment was expected. Its sources and unresolved status were sound, but one past-tense verb had converted an expectation into an occurrence. The run also

over-cited generic interviewer prompts. Provenance has to be judged by what each cited passage actually says.

What the run showed, and what the gate accepted

Before reading any result, you record what was actually run. The manifest names the contract, every staged input with its hash, the runtime, and the invocation, so that a reviewer can challenge the output without trusting your account of it:

```
run_id: 9.2-eventstormer
branch: ch09/9.2-eventstormer

contract:
  source: docs/ch09/run-pack/contracts/brewup.eventstormer.agent.md
  sha256: 08b0f290f116...

inputs:
  # staged into an empty workspace
  - source: docs/ch09/run-pack/stock-walkthrough.md
    sha256: 29fbf6838fe5...
  - source: docs/ch09/run-pack/glossary.md
    sha256: dba095c357ab...
  # constitution-excerpt.md and handoff-schema.md follow, each hashed

runtime:
  harness: codex exec
  parameters: {sandbox: read-only, user_config: ignored, ephemeral: true}

procedure:
  invocation: >-
    Open brewup.eventstormer.agent.md and follow it exactly. Treat the files
    in this workspace as the complete supplied input for this run. Do not
    inspect paths outside this workspace. Return only the YAML document
    required by handoff-schema.md.

output:
  sha256: 0f62f5949c7e...
  edited_after_capture: false
```

Each of the four runs has a manifest of exactly this shape, recorded in its own directory as `input-manifest.yaml` beside the invocation, the raw output, the evaluation and the human decision; only this one is printed here. The hashes carry more weight than the filenames: they

let a reader confirm that the file staged into the run is the file now in the repository, and `edited_after_capture` records whether you touched the output after it was returned.

Run result: The EventStormer surfaced all 19 pre-registered evidence items, with no misses or wholly unsupported additions. Sales, payment, lab, and Shipping facts stayed outside Stock authority. All six declared uncertainties survived, and the two failure hypotheses were not observed. Provenance was only partially met because of the naming overclaim and generic over-citation.

Human gate: Review each candidate, not the agent's overall fluency:

- Does the cited evidence support the exact statement?
- Does the statement distinguish who brought the fact about from the authority who must decide whether it advances?
- Do status and uncertainty keep an open matter open?
- Does the review question name the human authority who must answer it?

Thirty-nine primary candidates passed directly. Reviewers referred **ES-24** and **ES-29** back, renamed them, and then accepted them. The effective accepted-facts.md therefore contains 41 candidates, including all 11 unresolved discovery records. Acceptance makes a record an authoritative input for the next transformation while leaving its unanswered policy open.

That leaves `accepted-facts.md` in a specific state: 41 records, 11 of them still carrying an open question, each citing the walkthrough passage it came from. Nothing in the file decides what the brewery should do when a quantity is short. The next specialist receives it in exactly that state, and meets a different pressure. Where the EventStormer was tempted to complete a policy, the Storyteller is tempted to finish a story.

Storyteller: making behavior concrete without completing the story

The **Storyteller** works one step further from raw evidence. The Storyteller turns accepted facts into particular actors, quantities, actions, and variations, which is the move we made by hand with Domain Storytelling in *Chapter 4*. Specific actors, quantities, and actions let a reviewer test each step against the evidence and stop where a policy, owner, or outcome is missing.

From accepted facts to concrete scenarios

The authoritative input is *accepted-facts.md*, the file a human reviewed and signed off on. A Domain Storytelling scenario must trace through those accepted facts to the original QUOTE and OBS identifiers, and the contract makes that traceability an instruction. The Storyteller receives the walkthrough for the trace, but never the carrier as a script to reproduce.

A request for 12 cases makes an abstract fact concrete: the warehouse checks available quantity, moves beer to the end bay, records the order, and tells Sales. The sequence lets a reviewer ask whether the physical movement, the board entry, or both create the hold. The run produced seven scenarios and four variations; it did not turn them into a workflow or a model.

Where the evidence stops, the story must stop

The contract spends its longest section on the discipline of stopping, because that is the one the specialist will break first:

```
Put the identifiers of the accepted facts a scenario relies on in `source`,
alongside the evidence identifiers.

## Where the evidence stops, the scenario stops
This is the whole of your discipline, and it will feel wrong every time.

A scenario wants an ending. When you reach a point the evidence does not settle —
how much time passes, what happens when only some of what was asked for can be
supplied, who acts next, how one kind of trouble is told apart from another —
**stop the scenario there.**

Do not finish it. Do not pick the reasonable outcome. Do not write "and then,
presumably". Return the unfinished scenario with `status: unresolved`, and put in
`review_question` the decision that would let it continue.

An unfinished scenario that names the missing decision is the most valuable thing
you can produce. A finished one that quietly supplied the decision is the most
damaging, because it will be read as a description of
the business and nobody will be able to tell which part came from the business and
which part came from you.
```

The decisive case asks for 12 cases when only eight are available. Andrea refuses the request; Nadia puts the eight aside and has someone call Sales. The evidence records both practices and no rule that selects one. Reserving eight, rejecting twelve, waiting for replenishment, or making Sales decide would each supply a policy or owner that the source does not provide.

The run kept both practices as separate variations, with each retaining its evidence trail. ST-02 cites the accepted fact before it cites the evidence, so a reviewer can walk back one gate without leaving the file:

```
- id: ST-02
  kind: variation
```

```
  name: Andrea refuses a short order
  statement: Sales sends the warehouse an order for twelve cases; only eight can
be had; Andrea places none of the eight in the end bay, marks the whole request
not done, and returns to Sales the single response that it could not be done.
  source: [ES-17, ES-18, ES-20, QUOTE-13, QUOTE-14, QUOTE-15, QUOTE-16, QUOTE-18,
QUOTE-19, QUOTE-20]  # accepted facts, then evidence
  status: unresolved
  decision_owner: unknown
  uncertainty: This is Andrea's learned practice, but it conflicts with Nadia's
practice and no short-quantity rule was written down.
  review_question: Who has authority to decide whether a request for twelve must
be refused completely when only eight can be had?
```

ST-03 mirrors it for Nadia's practice. Neither scenario chooses between them, and neither invents the policy that would make the choice unnecessary.

The same restraint applies when beer is out, and more is expected on Thursday. The Storyteller cannot claim that replenishment arrived, that the shortage ended, or that anyone retried the request. The Storyteller may expose those possibilities, but it must stop at the missing decision. Cancellation, payment, failed-batch, and Shipping cases follow the same rule.

A useful unresolved scenario records three things: the last supported step, the missing evidence or authority, and the question a reviewer must answer before the sequence can continue.

What the run showed, and what the gate accepted

Run result: Scenario form covered all 12 declared evidence items and made the open questions operational. It separated shortage from an invalid product code and time-based clearing from release after a Sales cancellation. Neither pre-registered failure hypothesis was observed. Human review did find two names that overstated a replenishment date and an unobserved Sales-to-Shipping route; the reviewers corrected both.

Human gate: Accept a scenario only when:

- Its sequence stays within accepted facts and cited evidence.
- External authorities remain external where Stock only receives or reacts to a decision.
- Its status and uncertainty preserve every open question, and the reviewer can trace the scenario through accepted facts to the walkthrough.

The gate accepted all 11 effective candidates: nine directly and two after correction; 10 retained unresolved status. The gate left short-quantity, retry, hold-duration, payment, failed-batch,

and Shipping decisions open. accepted-scenarios.md carries those limits into the next transformation.

Those open limits are what make the next transformation risky. A scenario that stops mid-sequence is visibly unfinished, and a reader can see the gap. A command with a precise name looks finished whether or not the evidence supports it, and precise names are exactly what the Command & Event Writer produces.

Command and Event Writer: precision without stolen authority

Commands, events, and rules resemble implementation artifacts, so a precise name may be read as a settled design decision. Each candidate must identify an evidenced intention, outcome, or condition and leave ownership unresolved when the source does not establish it.

From scenarios to behavioral candidates

The **Command & Event Writer** receives the 11 accepted scenarios and the accepted facts beneath them. The writer may propose commands for evidenced intentions, events for evidenced outcomes, and rules only for conditions the evidence presents as binding. None is yet assigned to a model or approved for implementation.

Chapter 6 used CQRS to distinguish intentions, facts, and reaction policies. Here, the classification is narrower and remains pre-model. The contract returns candidate commands and events, and permits a rule only when the evidence states a binding condition:

```
## What to return
| kind    | What it is                                                |
|---------|-----------------------------------------------------------|
| command | An instruction to do something, named as an instruction   |
| event   | Something that has happened as a result                   |
| rule    | A condition the evidence states must hold                  |

## Instructions belong to whoever was described carrying them out
The strongest pull in this work is toward tidiness: a scenario has a gap, one
instruction would close it, and there is a group right there who could plausibly
be given it.
Resist it. Propose an instruction for this group only where the evidence shows
them carrying that work out. Two questions are worth asking about every command
you write:

1. Which cited passage shows someone doing this?
```

```
2.
If nobody does it in the evidence, am I proposing it because the business needs
it, or because my scenario would otherwise be incomplete?

The second reason is not a reason. Return the gap as `status: unresolved` with a
`review_question` naming the decision and the authority that must make it.

## Rules
Propose a `rule` only where the evidence states a condition. A practice that two
people described differently is not a rule; it is `status: unresolved`.
A practice that someone said had never been written down is not a rule
either, however consistently they described following it.
```

For commands, ask who has authority to issue the instruction. For events, distinguish the participant that caused the outcome from the bounded context that may own it. The recorded 9.4 contract is the second place where `decision_owner` carries an event's causal actor rather than the deciding authority. The output remains as run evidence, but the field semantics should not be reused.

In this walkthrough, warehouse staff perform the stock work. Sales, the office, the lab, and Shipping remain external sources or actors wherever the evidence assigns the relevant decision or action to them.

The run produced 16 commands and 20 events. Where a scenario depended on an action with no evidenced owner, message, or rule, the candidate stayed unresolved with a review question instead of closing the flow.

The pull of the convenient command

The retry scenario makes the temptation visible. Beer is out, more is expected, and the request might be tried again only if Nadia remembers, Sales calls, or someone else intervenes. Adding `RetryReservation` and assigning it to Stock would make the flow neat, but the evidence identifies neither the owner, the replenishment signal, nor a policy that says the request must be retried.

The run records the retry as CEW-18, with status: unresolved and `decision_owner`: unknown; no Stock-owned confirmation command appears. For Shipping, CEW-30 records the observed instruction to take held beer onto a van while leaving its issuer, dispatch authority, and timing unresolved. Both gaps remain review questions for the relevant authorities.

The same constraint applies to short quantities, hold duration, payment, and failed batches. The writer may record actions described in the walkthrough, but it must leave their governing policy unresolved unless the evidence states that the action is required.

What the run showed, and what the gate accepted

Run result: Fifteen candidates remained unresolved. Neither pre-registered failure hypothesis was observed: confirmation and retry stayed outside Stock authority, and the output did not reconstruct an order aggregate inside Stock. The run proposed no rule candidates. That absence required a gate check: the evaluation baseline identifies confirmed conditions for held and released stock, while unresolved practices must not be promoted to rules.

A provenance limit appears here in the handoff design. The Storyteller contract tells the specialist to cite the accepted facts a scenario relies on, and all eleven scenarios do. The shared schema defines source as identifiers taken from the evidence, while the Command & Event Writer contract does not ask for scenario identifiers. None of the thirty-six behavioral candidates therefore names the scenario it transforms; each cites the walkthrough directly. At the gate, a reviewer can reconstruct that link only by opening both files. A reusable handoff should represent both upstream scenario identifiers and underlying evidence identifiers explicitly, which requires coordinated changes to the shared schema and the Command & Event Writer contract. The recorded 9.4 output remains unchanged so that the observed result stays reproducible.

Human gate: Review behavioral candidates against five tests:

- Is the intention or outcome supported by the accepted scenario and evidence?
- Does ownership match the candidate kind and the authority that decides or acts?
- Has a convenient name made a neighboring authority or a missing instruction look settled?
- Is provenance complete through scenarios and facts to the walkthrough?
- Where the run returned no rule candidates, did the evidence contain a condition that should have been represented as one?

The gate accepted all 36 returned candidates as modeling material, but this candidate-level decision did not certify the run as complete. The absence of rule candidates and the missing scenario links remain separate run-level findings. Fifteen candidates remain unresolved on partial availability, retry, hold duration, payment, failed-batch response, and the Shipping route. accepted-behavior.md is the authoritative staged input for the Context Mapper, while remaining candidate material; it defines neither a command catalog, an aggregate design, nor an automatable process.

One transformation is left. Facts say what happened, scenarios make it concrete, and behavioral candidates name the instructions and outcomes involved. None of the three says which responsibilities belong together, or where one authority stops and another begins. This

is the question the last specialist is given, and it is the one in which a wrong answer is hardest to spot, because the output looks like a diagram.

Context Mapper: revealing boundaries without designing the implementation

By the time material reaches the **Context Mapper**, the chain has already made several kinds of meaning explicit. Accepted facts describe what the evidence supports. Accepted scenarios make those facts concrete. Accepted behavior identifies candidate intentions, outcomes, rules, and the authorities associated with them. The next question is no longer simply what happens, but which responsibilities belong together and where one authority gives way to another.

The Context Mapper performs the final transformation. It reads the accepted material and proposes candidate responsibility boundaries and relationships between them. A boundary groups decisions that appear to belong to the same authority. A relationship records something that crosses from one responsibility to another: a request, a fact, a dependency, or another piece of information whose direction and ownership can be supported by the evidence. As in the previous transformations, the result remains modeling material for review rather than an accepted description of the domain.

This step carries a particular risk because boundaries look architectural very quickly. A name such as Warehouse, Sales, or Shipping resembles a module name. A line between two boxes resembles an integration. A directional arrow can suggest a message, API, or event flow even when the evidence established only that one part of the business knew something about another. Once drawn, these shapes can make a tentative responsibility split feel like an implementation decision.

The goal is to make responsibility boundaries inspectable without allowing the diagram to imply decisions about the organization, domain model, or implementation.

From accepted behavior to responsibility boundaries

Accepted behavior gives the Context Mapper more than a list of commands and events. It carries the decisions that appear to belong together, the external facts Stock reacts to, the responsibilities that remain elsewhere, and the uncertainties that survived the previous gates. The Context Mapper uses that material to ask a strategic question: *where does one responsibility end and another begin?*

Its candidate forms are boundaries and relationships. A boundary groups responsibilities that appear to share a coherent decision authority. A relationship records something that passes between two such responsibilities. Neither is merely a visual container. A boundary makes a

claim about ownership; a relationship makes a claim about interaction. Both therefore require the same provenance discipline that governed facts, scenarios, commands, and events.

The complete accepted chain matters here. A command or event name alone can suggest a boundary too easily. `ReturnBeerToStock`, for example, says little unless you also know which scenario produced it, which facts support that scenario, and which external decisions made the action legitimate. The Context Mapper can follow that trail back through accepted behavior, scenarios, and facts to the original walkthrough.

This also prevents neighboring responsibilities from becoming relationships by default. Sales, the lab, the office, and Shipping all appear in accepted material, but their presence does not prove that Stock has a direct exchange with each of them. A relationship requires evidence that something crosses the boundary and enough information to say in which direction it travels. Where the material establishes only that another responsibility exists, the Context Mapper may preserve it as a neighboring boundary candidate without inventing an edge.

Direction alone is not enough. Every relationship also raises the question of decision ownership. Stock may receive a cancellation from Sales and act on it, but receiving that fact does not make Stock the authority that canceled the order. The lab may report that a batch failed its check, but Stock does not thereby own the quality decision. A useful context map keeps the direction of information separate from the authority that determines its meaning.

For the same reason, an unconnected boundary can be a legitimate result. Diagrams encourage you to connect visible elements, but the absence of evidence for an exchange is itself important information. Leaving a candidate boundary without an edge preserves that gap for review instead of filling it with an architectural assumption.

The next question is whether the names placed around those responsibilities describe what the business actually decides, or merely repeat the labels already familiar from rooms, teams, and software modules.

Run 9.5: mapping responsibilities

Run 9.5 completes the staged chain. The Context Mapper receives the original walkthrough together with accepted-facts.md, accepted-scenarios.md, and accepted-behavior.md. Its contract, brewup.context-mapper.agent.md, permits two candidate kinds – boundary and relationship – and repeats the constraint that governs the prose above: a visible neighboring responsibility is not enough to justify an edge.

The specialist's primary output is a context-map modeling report in the shared handoff schema. In a writable repository workflow it may also create or update a derived context-map artifact, but that artifact remains a projection of the report: repository persistence does not turn a proposed boundary or relationship into accepted domain knowledge. In the recorded

experiment, the report remains the authoritative review surface and the human gate decides what may advance.

The 9.5 checkpoint is also where local and chain-level evaluation meet. The Context Mapper can be reviewed for unsupported names and relationships, while the integrated evaluation asks whether ownership, vocabulary, provenance, and unresolved questions survived all four transformations. Those are separate judgments: a locally disciplined map can still reveal drift introduced earlier in the chain.

Run result: The primary run was executed in a fresh controlled-input workspace and returned 18 candidates: eight boundaries and ten relationships. Its summary reported five unresolved candidates, but the body contained four. The evaluator recorded the mismatch as a run defect rather than silently editing the raw output. Every item in the pre-registered subset for this specialist was surfaced, and of the three pre-registered failure hypotheses, the evaluator observed only FH-08 in CM-17 below.

The gate accepted 15 primary candidates directly and referred CM-10, CM-16, and CM-17. The guided correction removed an unsupported citation from CM-10, kept failed-batch notification authority unknown in CM-16, and replaced CM-17's invented Sales release authority with the observed physical collection by Shipping. The effective handoff therefore contains all 18 candidates, with CM-06, CM-08, CM-12, and CM-14 still unresolved.

Naming boundaries for what they decide

One of the easiest ways to make a candidate boundary appear more certain is to give it a familiar name. Warehouse is an obvious example. The walkthrough takes place in a warehouse, the current implementation contains a Warehouse module, and several participants naturally use the word. None of those facts proves that Warehouse is the right name for a bounded context.

A bounded context should be named for the responsibility and decisions it owns, not for the room in which work happens or the software structure that currently implements it. If the accepted material shows a coherent set of decisions around stock availability, holding beer, and releasing it, the Context Mapper may propose a boundary around that responsibility. However, the name attached to it must remain a proposal unless the business language already supports it.

This distinction matters because names influence later design. Once a box is called Warehouse, it becomes easy to assume that the existing module already represents the right boundary and that future implementation should preserve that shape. The map would then be reflecting the software back into the domain instead of helping you question whether the responsibility has been understood correctly.

CM-01, the run's first candidate, avoided that trap. The candidate groups the decisions around availability, holding, and release. The candidate gives the boundary a responsibility-based name, and the same record states that the business names the implementing group rather than the boundary itself. No candidate in the run is named Warehouse, which is the pre-registered failure this specialist was most likely to produce.

The guard introduced in *Chapter 8* can help expose an unsupported or inconsistent term, but it cannot choose the business name on behalf of the domain. That decision still belongs to the people who own the language and the responsibility the boundary is meant to express.

A tidy map can still be wrong

A context map creates a different kind of pressure from the artifacts that came before it. Facts can remain unresolved, and scenarios can stop where the evidence stops. A diagram, by contrast, tends to look unfinished when a boundary has no connection or when two responsibilities appear close, but no relationship has been drawn between them.

That pressure makes unsupported edges particularly dangerous. Sales, Shipping, the lab, and the office all appear in the accepted material, but their presence does not prove a direct relationship with Stock. A line between two boundaries claims that something crosses between them. An arrow adds a second claim about direction. If the evidence does not support both, the map has become cleaner by becoming less truthful.

The run failed once here, and the gate caught it. The primary output proposed a relationship named held beer released for delivery and made Sales its `decision_owner`, because Sales decides that an order goes through and Shipping later collects the beer. The evidence supports that sequence and says nothing about who releases the beer or who instructs Shipping. The candidate was referred back, and the corrected CM-17 records only the physical collection, with `decision_owner` returned to `unknown`. Sequence had become authority in a single field, which is the pre-registered failure for this specialist and the reason a plausible edge deserves the same scrutiny as a plausible policy.

The quality of the map depends on whether every boundary and relationship can be traced to accepted material while uncertainty remains visible wherever the domain has not yet decided.

Human gate: accepting boundaries, not implementation structure

The final gate reviews the Context Mapper's candidates as domain-modeling material. The gate tests whether each proposed boundary groups responsibilities that the accepted evidence supports. It also tests whether each relationship has a defensible direction and owner and whether unresolved naming or ownership questions remain visible. A plausible diagram is not enough; every box and every line must still be traceable to the material that justified it.

Acceptance at this stage is deliberately limited. Acceptance means that a boundary or relationship is useful enough to become part of the accepted modeling material. Acceptance does not mean that the organization must create a team with the same name, that the codebase should contain a matching module, or that two boundaries must communicate through a particular API, message, or event.

Fifteen candidates were accepted directly and three only in their corrected form, so the effective `accepted-context-map.md` holds all 18. Four of them were accepted while remaining unresolved: failed-batch resolution, product-code correction, retry ownership, and who must be told when only part of a request can be met. Acceptance approved no boundary name, no team, no module, and no integration.

With this final gate, the four specialists have each completed a bounded transformation. The next question is whether the distinctions preserved locally also survived the chain as a whole.

Evaluating the complete chain

Each specialist has so far been evaluated against its own contract. The local evaluation tells you whether a transformation respected its local boundary, but not whether the complete chain preserved the meaning of the original evidence.

The risk changes when several disciplined transformations are composed. A vocabulary choice accepted early may acquire a slightly different meaning later. An unresolved question may disappear because a downstream artifact no longer carries it explicitly. An ownership distinction that was clear in a scenario may become less visible once it is represented as a command, event, or relationship. No single step needs to fail dramatically for the chain to drift.

Human gates reduce this risk by allowing only accepted material to advance, but acceptance alone does not guarantee coherence. Coherence also requires that provenance, uncertainty, decision ownership, and important language distinctions remain traceable as the material changes from facts to scenarios, from scenarios to behavior, and from behavior to responsibility boundaries.

The final evaluation therefore looks across the four transformations rather than at each specialist in isolation. You now ask whether one vocabulary survived the journey, what information had to remain intact at every handoff, and where the resulting chain preserves the evidence without silently adopting or losing meaning.

One vocabulary, four transformations

Shared evidence does not guarantee shared language. Each specialist sees the domain through a different transformation, and every transformation creates an opportunity for a term to shift meaning.

The stock walkthrough provides a useful example. Participants describe `beer as set aside, put on hold, and earmarked`. The EventStormer preserved those expressions without deciding whether they named one fact or several. A Storyteller could then use one of them while constructing a concrete situation, and the Command & Event Writer might turn the resulting behavior into a more precise command or event name. By the time the Context Mapper groups responsibilities, the original ambiguity can easily have disappeared even though no domain authority ever resolved it.

That is what the integrated evaluation looked for, and in this chain it did not happen. The three expressions stay traceable through the accepted artifacts: the EventStormer kept them as separate records without deciding whether they name one action or three, the Storyteller and the Command & Event Writer used each of them where the evidence used it, and the Context Mapper introduced responsibility-level names of its own while stating in every record that the business has not approved them. FH-10, the pre-registered hypothesis for vocabulary drift, is the only one of the 10 that can be scored across runs rather than within one, and it was not observed. This result is narrower than it may sound. No term quietly won; equally, nobody chose between them. The question is still open in accepted-facts.md, which is where it belongs until a domain authority answers it.

Different artifacts may legitimately use different words. Facts, scenarios, commands, events, and boundaries serve different purposes and take different forms. The distinction behind the words is what has to stay stable. If a later specialist introduces a more precise term, the chain should still make clear whether that precision came from accepted domain language or from the transformation itself.

Vocabulary drift is therefore a chain-level problem. A locally reasonable name may become an apparently established concept simply because several later artifacts reuse it. The human gates help by preventing unaccepted candidates from advancing, but they must also preserve unresolved language rather than allowing repetition to turn a proposal into a convention.

Coherence therefore rests on traceability rather than on uniform wording. Each transformation has to preserve enough provenance and uncertainty for a reviewer to see when the meaning changed, why it changed, and whether anyone with domain authority accepted the change.

What must survive every handoff

A handoff is successful only if the next specialist can transform the accepted material without losing the reasons that made it acceptable. The artifact may change form, but its connection to evidence, authority, and unresolved questions must remain visible.

Provenance is the first requirement. A scenario should remain traceable to accepted facts and, ultimately, to the original walkthrough. A behavioral candidate should retain that trace through the scenario, and a proposed boundary should not sever it. Stable identifiers matter

because they let a reviewer follow a chain backward instead of trusting the final artifact in isolation.

Status and uncertainty must survive as well. An unresolved question does not become settled merely because a later specialist needs a more concrete representation. If retry ownership was unknown in a scenario, a command should not silently acquire an owner. If a boundary name remained proposed, repetition across later artifacts should not make it appear accepted.

Decision ownership is equally important. Transformations may change how a responsibility is expressed, but they must not move the authority behind it. A Sales decision can become an incoming event, a scenario condition, or a relationship on a context map without becoming a Stock decision along the way.

Only the human gates may change what is accepted. Specialists can refine, rename, infer, or propose, but acceptance is never inherited from fluency or repetition. Preserving a candidate that was stopped, referred back, or left unresolved is therefore just as important as preserving material that advanced.

The purpose of the handoff is to preserve enough context for a reviewer to reconstruct how meaning changed, which decisions were made by people, and which questions remain open.

What the chain preserved

Evaluating the complete chain requires a different comparison from evaluating each specialist in isolation. The question is no longer whether one transformation covered its evidence subset. It is whether the accepted artifacts together preserved what the walkthrough supported, avoided importing what it did not support, and kept new uncertainty visible.

The comparison is made against the Stock Management domain carrier, which the evaluator held back until this point. Eleven of the carrier's concepts are reachable from the walkthrough, and the accepted chain represents all eleven: the two stock quantities in ES-12 and the availability behavior below it, the physical set-aside and the recorded hold in ES-14, ES-15, and CEW-03 to CEW-06, cancellation-triggered release in CEW-25 to CEW-27, and time-based clearing in CEW-20 to CEW-23. A twelfth concept, an order expiring, is absent, and that is the correct result: nothing in the walkthrough describes Sales expiring an order, and QUOTE-40 says the warehouse is never told what became of such orders. A run that had proposed it would have been adding rather than recalling.

One candidate imported something the walkthrough did not support, and the gate caught it: CM-17, described above. The gate narrowed two others during correction rather than removing them: CM-10 and CM-16. Across the four runs, seven of the 106 returned candidates were referred back, and every correction sits beside the output it corrects instead of replacing it. No specialist output used accepted or rejected as a status, a discipline the shared schema enforces across all four specialists.

The open questions stayed open. Hold duration, partial availability, failure classification and retry ownership entered the chain unresolved and left it unresolved, in CM-01, CM-14, CM-08 with CM-13, and CM-12 respectively. The integrated evaluation records no artifact anywhere in the chain turning repetition into policy. The questions the chain raised for itself – what happens to beer already set aside when a batch fails its check, what to do with an invalid product code, what a notification must contain, who authorizes Shipping – were added to that list rather than answered within it. Authority stayed where the evidence put it: Sales keeps order progression, the lab keeps the batch decision, payment stays with the office, and who authorizes Shipping collection is still unknown.

One finding belongs to the fixture rather than to a run. The leak check is the audit that the run pack does not contain the answers the runs are meant to produce, and the first freeze failed it. The glossary carried a single entry, gathering set aside, put a hold on, and earmark, stating that the speakers treated them as the same action – which settled the vocabulary question before any specialist saw the evidence, and made unifying the three terms cost nothing. Splitting that entry into three, each described as it was heard, is what made the unification measurable at all. The audit earns its place in the protocol precisely because it failed once.

Taken together, the chain recovered everything the evidence supported, imported one thing it did not, and ended with more open questions than it started with. The last of those three is the result worth keeping in mind while reading what follows.

The price of specialization

Specialization reduced the freedom of each agent but did not reduce the amount of judgment required by the modeling process. In some respects, it made that judgment more visible. Every transformation ended with candidate material that someone had to inspect before the next specialist could safely use it.

This friction is the price of the boundaries introduced throughout the chapter. A single unconstrained agent could have moved from the walkthrough to a polished model in one step. The four specialists deliberately could not. Each stopped after one transformation, exposed what it had inferred or left unresolved, and waited for a decision about what was allowed to advance.

That extra friction is deliberate. It is what prevents a plausible interpretation introduced early in the chain from silently becoming an assumption for every step that follows. Specialization trades continuity of execution for locality of responsibility. The result is easier to inspect, but somebody still has to coordinate the inspections.

Four agents create four review gates

The four specialists therefore require four explicit review gates. The EventStormer gate decided which candidate facts could become accepted evidence for the Storyteller. The Storyteller gate decided which scenarios represented the evidence faithfully enough to advance. The Command & Event Writer gate reviewed behavioral candidates and their ownership. The Context Mapper gate decided whether proposed responsibility boundaries and relationships were defensible as modeling material.

The gate decisions were smaller decisions than reviewing one complete model produced in a single pass. Each gate had a narrower question, a limited set of candidate types, and a provenance trail back to the material that justified them. A reviewer did not need to reconstruct the entire modeling process every time an uncertainty appeared.

Well-constrained specialists still leave the gates in place. Someone had to compare candidates with evidence, inspect uncertainty, challenge ownership, decide what could advance, and prepare the accepted material for the next transformation. The contracts narrowed the space of each review and left the review itself intact.

What the human still had to decide

The experiment also makes clear what specialization could not decide on its own. Several questions survived the complete chain precisely because the available evidence did not contain the authority to answer them.

The agents could expose the disagreement around partial availability, but they could not decide whether a short quantity should be held or rejected. They could identify that retry behavior lacked an owner, but they could not assign one. They could preserve uncertainty about how long beer remains on hold and how payment status affects that hold. They could also leave open what should happen after a failed batch and who authorizes or informs Shipping when beer is ready to leave.

These unresolved questions are not failures of the specialists; answering them would have violated the specialists' boundaries.

The human contribution was therefore a matter of judgment rather than editing. It was to decide when the evidence was sufficient, when a proposal could become accepted modeling material, and when a question had to remain open because the relevant domain authority had not yet been consulted.

Authority is the part of the workflow that cannot be replaced by making the prompt more detailed. Better instructions can tell an agent to preserve uncertainty. They cannot manufacture the authority required to remove it.

What the repository makes reproducible

The repository makes this process inspectable beyond the prose of the chapter. It preserves the evidence supplied to each specialist, the contract that constrained the transformation, the invocation, the raw output, the evaluation, and the human decision that determined what advanced.

Concretely, four branches carry the chapter's runs, each adding one run to the one before it. The Git branch named `ch09/9.2-eventstormer` freezes the protocol, the evidence inventory, the pre-registered hypotheses, and the leak check, and adds the EventStormer run under `docs/ch09/runs/9.2-eventstormer/`. The Git branches named `ch09/9.3-storyteller` and `ch09/9.4-command-event-writer` add theirs in the same shape. The Git branch named `ch09/9.5-contextmapper` adds the Context Mapper run under `docs/ch09/runs/9.5-context-mapper/`, together with the chain-level findings above, in `integrated-evaluation.md`. Inside every run directory, the files are the same: `input-manifest.yaml`, `invocation.md`, `raw-output.md`, `evaluation.md`, `human-decision.md`, and the accepted handoff the next specialist received. Where a candidate was referred back, `correction-01/` holds the second invocation and its output beside the first rather than in place of it, which is why the run that produced CM-17 and the correction that narrowed it can both still be read.

Here, reproducibility does not mean expecting an LLM to generate exactly the same text on every run. A rerun may propose different names, split candidates differently, or rephrase an uncertainty in another way. The stable element is the experimental setup rather than the generated wording.

What can be reproduced is the question being asked of the model and the boundary within which it must answer it. The same evidence can be supplied under the same contract, the result can be evaluated against the same rules, and the decision can be compared with the recorded review. Raw output remains available rather than being replaced by polished versions, so a correction or retry does not erase the path that produced it.

The repository therefore preserves more than the final accepted artifacts. It preserves the conditions under which they became acceptable. That distinction makes the modeling chain inspectable even when generation itself remains non-deterministic.

From manual gates to orchestration

The handoffs in this chapter were deliberately manual, and that made it possible to see exactly what each gate produces and what work was required to move accepted material from one specialist to the next. Automating that coordination too early would have hidden the boundary we were trying to examine.

Once those boundaries are explicit, however, much of the coordination becomes mechanical. The correct input package must be assembled. Preconditions must be checked. Output must conform to the handoff schema. Referred or rejected material must stop or branch the chain, while accepted unresolved material must remain visibly unresolved. Provenance must survive every transition.

Those activities are good candidates for orchestration because they coordinate the workflow without making domain decisions.

An orchestrator must not silently turn unresolved into resolved, choose the owner of a missing decision, or accept a proposed bounded-context name. The orchestrator must not merge the four specialists back into one agent that can answer every question. Doing so would automate away the very cognitive boundaries established in this chapter.

The useful boundary for orchestration is therefore coordination versus authority. An orchestrator may enforce the workflow invariants listed previously, and it may record retries and transitions so the path stays inspectable. It still cannot infer the domain decision required to make a stopped transition valid.

The next chapter moves from creating specialized agents to coordinating them without recreating an unbounded agent.

Summary

This chapter moved from defensive guard agents to productive specialists. Across the BrewUp Stock Management chain, each specialist was defined around one bounded transformation and candidate material for a human gate. The four recorded runs showed how additions could remain attributable before becoming downstream premises. Run 9.5 returned 18 Context Mapper candidates, referred three for correction, and completed the integrated chain evaluation; four accepted candidates remain unresolved.

Partial availability, retry ownership, hold duration, payment, failed-batch response, product-code correction, and the Shipping route remain matters for domain authority rather than decisions an agent may settle.

In this chapter, you learned how to:

- Define specialist contracts through authoritative inputs, one transformation, inspectable outputs, and explicit prohibitions.
- Separate proposed material from decisions accepted by a human authority.
- Recognize invented policy, narrative closure, convenient-command, and tidy-map pressures.
- Preserve provenance, uncertainty, ownership, and review questions across handoffs, and evaluate the complete chain for semantic drift.
- Automate coordination without automating the domain decisions that review gates protect.

The next chapter builds on these boundaries by asking how to orchestrate specialists without recreating an unbounded agent.

10

The Orchestrator: Building the Coordinating Agent

Chapter 9 built four bounded specialists around a shared rule: each specialist might transform accepted material into a candidate, but only a human gate can decide what becomes authoritative input for the next stage. The recorded BrewUp chain made those boundaries inspectable, but its handoffs were still coordinated manually.

This chapter turns that manual coordination into an executable control plane. You start with one eligibility check, replace free-form agent conversation with typed messages and legal state transitions, and then run the complete BrewUp specialist chain. The orchestrator routes accepted work, records why progress stops, and preserves unresolved knowledge. Domain interpretation remains with the specialists and human reviewers.

The implementation uses C# and .NET, but the design is independent of a particular model or orchestration framework. The result is a replayable workflow in which every candidate, correction, gate decision, and stop remains attributable to the evidence and authority that produced it.

In this chapter, you will learn how to:

- Define the orchestrator's decision rights without turning it into another modeling specialist.
- Model multi-agent exchanges as typed messages and legal state transitions.
- Preserve causation, provenance, unresolved knowledge, and human authority across handoffs.

- Run, correct, stop, persist, and resume the complete BrewUp specialist chain.
- Verify repository-backed completion without treating workflow completion as proof that every domain question is resolved.

Technical requirements

The orchestration module in this chapter is part of the BrewUp solution used throughout the book. The preserved version is available on the companion repository branch `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design/tree/ch10/orchestrator`.

The code targets .NET 10 and lives under `src/Orchestration`. `BrewUp.Orchestration.Application` holds the coordination types: `EligibilityEvaluator.cs`, `ConversationProtocol.cs`, `BrewUpOrchestrationLoop.cs`, and `WorkflowPersistence.cs`. `BrewUp.Orchestration.Infrastructure` holds the adapters: `JsonWorkflowStore.cs`, `RepositoryCatalog.cs`, `GateVerifier.cs`, `RepositoryScenario.cs`, and the two specialist runners. The accepted *Chapter 9* artifacts that the final section replays are referenced from `docs/ch09`, not copied.

The executable demonstrations are in `BrewUp.Orchestration.Tests`: `EligibilityEvaluatorTests.cs`, `ConversationProtocolTests.cs`, `BrewUpOrchestrationLoopTests.cs`, and `RepositoryCatalogTests.cs`. The listings in this chapter are extracts from that preserved code, not an exercise to type in. You can follow the whole argument from the chapter alone; cloning the branch is needed only to run the tests and inspect the complete implementation.

The orchestrator as facilitator, not controller

You begin with the smallest useful coordinator for BrewUp Stock Management. The coordinator will decide whether Storyteller might run, refuse the transition when a gate is missing, record the human decision, and evaluate the same request again. The example uses EventStormer and Storyteller. These two are specialist names from *Chapter 9*, not C# classes; the C# types introduced in this chapter are the coordinator and its protocol. Later sections extend the mechanism to the full chain.

The coordinator enforces one invariant from *Chapter 9*: candidate facts do not become Storyteller input until a human gate records the accepted-facts artifact that Storyteller might consume. If this transition remains implicit, adding agents hides the failure.

The last chapter ended on the distinction this section makes executable: coordination versus authority. An orchestrator might enforce workflow invariants, such as that accepted artifact is

eligible as input and whether schema, provenance, and a required gate are intact. What it cannot do is infer the domain decision that would make a stopped transition valid.

Creating the orchestration workspace

Create the orchestration workspace under `src/Orchestration`, beside the other source modules, and keep its reviewable evidence under `docs/ch10`:

```
src/
  BrewUp.slnx
  Orchestration/
    BrewUp.Orchestration.Application/
      EligibilityEvaluator.cs
      ConversationProtocol.cs
      BrewUpOrchestrationLoop.cs
      WorkflowPersistence.cs
    BrewUp.Orchestration.Infrastructure/
      JsonWorkflowStore.cs
      RepositoryCatalog.cs
      GateVerifier.cs
      RepositoryScenario.cs
    BrewUp.Orchestration.Tests/
      EligibilityEvaluatorTests.cs
      ConversationProtocolTests.cs
      BrewUpOrchestrationLoopTests.cs
      RepositoryCatalogTests.cs
docs/ch10/
```

The workspace already exists on the companion branch; it contains two class libraries and one executable test project. The infrastructure library references the application library, and the Tests project references both of them. All three projects are registered in `src/BrewUp.slnx`, while `src/BrewUp.Rest/Program.cs` remains the application composition root:

```
Application      class library: protocol, state, orchestration loop
Infrastructure   class library: JSON store, repository catalog, runners
Tests            xUnit project: executable demonstrations and invariants
```

Application owns the protocol and workflow types, Infrastructure owns persistence and repository adapters, and the Tests project owns the executable demonstrations. The first fixture answers one procedural question: is the requested specialist eligible to run? The fixture does not assess the quality of the accepted facts.

Register two bounded specialists

The `EligibilityEvaluatorTests.cs` fixture starts with the first two transformations from *Chapter 9*. EventStormer consumes raw evidence and has no predecessor gate. Storyteller consumes accepted facts and therefore requires the EventStormer review gate:

```
private readonly IReadOnlyDictionary<string, Specialist> _registry =
    new Dictionary<string, Specialist>(StringComparer.Ordinal)
    {
        ["eventstormer"] = new(
            "raw-evidence", "candidate-facts", RequiresGate: null),
        ["storyteller"] = new(
            "accepted-facts", "candidate-scenarios", "eventstormer-review")
    };
```

The dictionary functions as a capability registry rather than a persona catalog. Each entry declares the input kind, candidate output kind, and predecessor gate required for routing. The specialist's behavioral contract remains elsewhere.

Record the first workflow state

Create the initial `WorkflowState` in the same test fixture. At the beginning of the run, only the evidence from the Stock walkthrough is authoritative input:

```
private readonly WorkflowState _state = new(
    "brewup-stock-01",
    new Dictionary<string, string>(StringComparer.Ordinal)
    {
        ["raw-evidence"] =
            "docs/ch09/run-pack/stock-walkthrough.md"
    });
```

The fixture intentionally contains no accepted facts. Candidate output might already exist in the repository, but only a recorded gate can make it authoritative input for the next transformation.

Implementing the eligibility check

The first coordinator is implemented as an application service rather than a second console application. `EligibilityEvaluator` receives a specialist name, the state, the registry, and a

gate directory. The service checks the required gate, makes the gate's effective artifact available, and verifies that its kind matches the specialist's declared input.

```
public static EvaluationResult Evaluate(
    string specialistName,
    WorkflowState state,
    IReadOnlyDictionary<string, Specialist> registry,
    string gateDirectory)
{
    if (!registry.TryGetValue(specialistName, out var specialist))
        return Blocked(
            state.Artifacts,
            $"unknown specialist {specialistName}");

    var available = new Dictionary<string, string>(
        state.Artifacts,
        StringComparer.Ordinal);
    var unresolved = Array.Empty<string>();

    if (specialist.RequiresGate is { } requiredGate)
    {
        var path = Path.Combine(gateDirectory, $"{requiredGate}.json");
        if (!File.Exists(path))
            return Blocked(
                available,
                $"required gate {requiredGate} is missing");

        GateRecord? gate;
        try
        {
            gate = JsonSerializer.Deserialize<GateRecord>(
                File.ReadAllText(path),
                JsonDefaults.Options);
        }
        catch (JsonException exception)
        {
            return Blocked(
                available,
                $"gate {requiredGate} is invalid: {exception.Message}");
        }
        if (gate is null)
```

```
            return Blocked(available, $"gate {requiredGate} is empty");

        if (!string.Equals(
                gate.GateId,
                requiredGate,
                StringComparison.Ordinal))
            return Blocked(
                available,
                $"gate id {gate.GateId} does not match {requiredGate}");

        if (!string.Equals(
                gate.Decision,
                "accepted",
                StringComparison.OrdinalIgnoreCase))
            return Blocked(
                available,
                $"gate {requiredGate} is {gate.Decision}");

        available[gate.EffectiveArtifact.Kind] =
            gate.EffectiveArtifact.Path;
        unresolved = gate.PreservesUnresolved ?? [];
    }

    if (!available.TryGetValue(specialist.Consumes, out var artifactPath))
        return Blocked(
            available,
            $"required artifact {specialist.Consumes} is missing",
            unresolved);

    var message = $"READY: {specialistName} can consume " +
                  $"{specialist.Consumes} from {artifactPath}";
    if (unresolved.Length > 0)
        message += $"; unresolved preserved: {string.Join(", ", unresolved)}";

    return new EvaluationResult(true, message, available, unresolved);
}
```

`EligibilityEvaluator` checks gate presence and disposition, exposes the gate's effective artifact, and verifies the required input kind. It never interprets the domain material.

Running the transition before the gate exists

Run the Storyteller eligibility test before writing any gate. The fixture creates an empty temporary gate directory, evaluates the request, and expects the precise blocked result:

```
var result = EligibilityEvaluator.Evaluate(
    "storyteller", _state, _registry, _gateDirectory);
Assert.False(result.Ready);
Assert.Equal(
    "BLOCKED: required gate eventstormer-review is missing",
    result.Message);
```

The refusal prevents Storyteller from consuming an unaccepted artifact. A coordinator that reads the raw EventStormer output and selects plausible facts would make the gate decision itself. Returning the missing gate identifier preserves the *Chapter 9* boundary and tells the operator what must happen next.

A focused `xUnit` test protects both the blocked outcome and the exact missing gate identifier.

Record the human gate and resume

The EventStormer gate creates the authoritative handoff. The test writes a `GateRecord` into its temporary gate directory with the decision, the effective artifact, and the unresolved identifiers that must survive the transition:

```
WriteGate(new GateRecord(
    "eventstormer-review",
    "accepted",
    new EffectiveArtifact(
        "accepted-facts",
        "docs/ch09/runs/9.2-eventstormer/accepted-facts.md"),
    ["ES-17"]));
```

The gate does not resolve ES-17. It accepts the record as a usable handoff while preserving the disagreement as an open premise for Storyteller.

Evaluate the same request again and assert that the accepted artifact becomes available without losing ES-17:

```
var result = EligibilityEvaluator.Evaluate(
    "storyteller", _state, _registry, _gateDirectory);

Assert.True(result.Ready);
Assert.Equal(["ES-17"], result.Unresolved);
Assert.Contains("unresolved preserved: ES-17", result.Message);
```

The trace also preserves ES-17, so later stages cannot mistake the open question for accepted policy.

Run the focused eligibility suite from the repository root:

```
$ dotnet test src/Orchestration/BrewUp.Orchestration.Tests/
BrewUp.Orchestration.Tests.csproj --nologo \
    --filter FullyQualifiedName~EligibilityEvaluatorTests
Passed! - Failed: 0, Passed: 4, Skipped: 0, Total: 4
```

Generalize the boundary from the working example

The working example makes this boundary concrete. The orchestrator can select an eligible transition, assemble accepted input, validate declared conditions, record a refusal, and carry unresolved identifiers forward. The orchestrator cannot create a gate, change Referred to accepted, or drop ES-17.

Table 10.1 generalizes the decision rights exercised by the small program, giving *Chapter 9*'s coordination-versus-authority boundary a row-by-row operational form. Read each row as one action and three answers: what the orchestrator may do mechanically, what only the specialist may do, and what only a human might decide. The rows are intentionally concrete: each one can later become a function, validation rule, persisted record, or human action in the complete workflow.

Action	Orchestrator	Specialist	Human authority
Select the next eligible step	Routes from recorded state	Does not select itself	Not required
Prepare the input package	Assembles accepted artifacts	Consumes the staged package	Establishes what is accepted

Action	Orchestrator	Specialist	Human authority
Transform domain material	Must not transform it	Produces candidates	Reviews the proposal
Check schema and provenance	Validates declared rules	Supplies required fields	May challenge the evidence
Accept, refer, or reject	Records and applies the decision	Cannot decide	Decides at the gate
Resolve policy or ownership	Stops and exposes the gap	Preserves uncertainty	Decides when authorized
Record a run or transition	Maintains the trace	Provides raw output	Signs the gate record

Table 10.1 – Decision rights across the orchestration boundary

Note

When stopping the workflow is the correct result

When evidence, an artifact, or a required decision is missing, continuation is the defect. The orchestrator's reliability depends on refusing invalid transitions as well as performing valid ones.

The result is a thin control plane. Deterministic code checks eligibility, artifact kinds, gate records, identifiers, and provenance. Bounded specialists transform domain material, and human reviewers accept, refer, or reject candidates. Any new synthesis must be assigned to another contracted specialist, not hidden inside the coordinator.

The two-role coordinator makes unauthorized transitions mechanically impossible and explains each refusal. So far it has done so for a single handoff. Scaling that discipline to the whole chain means giving the exchanges themselves a form the orchestrator can check, and that is where the next section starts.

Designing multi-agent conversations

The eligibility check decides whether a specialist might start. The eligibility check does not govern what follows a candidate: review, correction, acceptance, rejection, duplicate delivery, or stopping when authority is missing. `ConversationProtocol` replaces transcript interpretation with typed messages, sender authority, evidence requirements, and legal state transitions. Specialists retain the generative work, and human gates remain authoritative.

From free-form chat to an explicit protocol

Consider the following plausible exchange. It is readable, but it is not safe to execute:

```
EventStormer: I found the short-quantity exception.
Orchestrator: Looks good. Storyteller, continue from it.
Storyteller: I will write the scenario.
```

The phrase "*Looks good*" cannot authorize a transition. The phrase might describe a valid file, a plausible candidate, or human acceptance. The exchange also omits the workflow identifier, exact artifact, evidence references, ES-17, and the message that made Storyteller eligible, so a replay cannot prove what authorized the handoff.

For each exchange, the protocol must answer four operational questions:

- Identity: which workflow and message does this exchange belong to?
- Authority: is the sender allowed to make this statement?
- Causation: which previously applied message does this one follow?
- Effect: which state transition is legal if the message is valid?

Figure 10.1 answers those four questions for one complete stage. It is the same handoff the free-form exchange attempted, redrawn as an ordered exchange through the orchestrator, with the human gate shown explicitly.

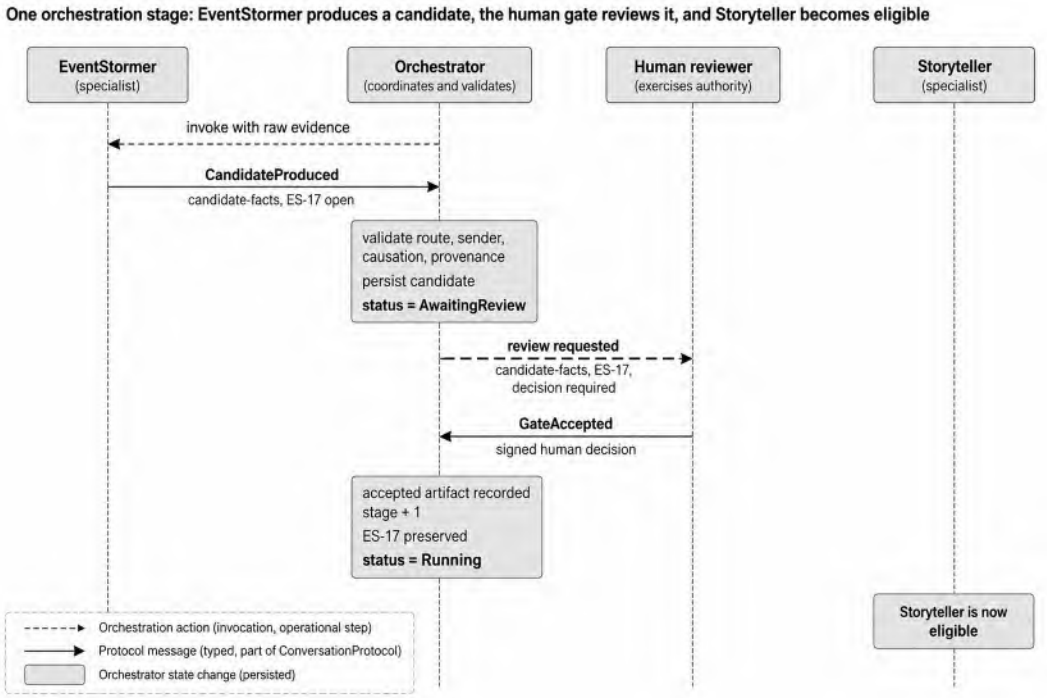


Figure 10.1 – One orchestration stage: candidate, human gate, and the next specialist made eligible

A typed message now becomes the unit of coordination. The orchestrator applies, blocks, or ignores it according to the protocol. The human-readable explanation reports that result instead of replacing it.

Note

Framework lens – BMAD: what authorizes a handoff between roles?

BMAD moves work between named roles, and its handoffs read much like the exchange above: one role reports, another approves, the next begins. A role brief says what a role might do, not which message might change the state of the work. Record the sender, the artifact, and the decision that authorized a handoff, not the assent.

Defining the artifact envelope

The protocol lives in `ConversationProtocol.cs` in the application project. The protocol starts with an envelope for every candidate or accepted artifact:

```
public sealed record ArtifactEnvelope(
    string WorkflowId,
    string Producer,
    string Kind,
    string Path,
    string[] EvidenceRefs,
    string[] Unresolved,
    int Attempt,
    string ContentSha256,
    string[]? SourceArtifactSha256 = null,
    string? HumanDecisionSha256 = null);
```

The artifact payload remains outside the message. Path points to the reviewable artifact, while the envelope carries coordination data. Producer and kind bind it to the sender and route, EvidenceRefs exposes its source set, and ContentSha256 identifies the exact bytes that were reviewed.

Unresolved makes open questions enforceable: acceptance might advance them, but it might not erase them. Attempt distinguishes a correction from a replay. Accepted envelopes also carry source-artifact and human-decision hashes, linking each handoff to the candidate and authority record that created it.

Tip

Keeping the envelope small

The full candidate does not need to be copied by the system into orchestration state. Persist the artifact once and exchange a typed reference. The protocol should validate coordination facts, not become another domain model.

Defining the messages and routes

The protocol begins with five message types. `CandidateProduced` submits work for review. `GateAccepted` creates an authoritative handoff. `GateReferred` requests a correction from the same producer. `GateRejected` terminates the candidate path. `AuthorityRequired` stops when the current specialist cannot continue without a decision it is not allowed to make:

```
public enum ProtocolMessageKind
{
    CandidateProduced,
    GateAccepted,
    GateReferred,
    GateRejected,
    AuthorityRequired
}

public sealed record ProtocolMessage(
    string MessageId,
    string WorkflowId,
    string CausationId,
    ProtocolMessageKind Kind,
    string Sender,
    string Recipient,
    ArtifactEnvelope? Artifact = null,
    string? Reason = null);
```

Every inbound message targets the orchestrator because peer-to-peer handoffs would bypass state validation and the central trace. `CausationId` must equal the last applied message, connecting each proposal to the accepted knowledge that made it eligible. A retry might reuse the workflow and artifact path, but not the message identity.

Now, declare the four-stage contracts. The list describes the protocol shape without running the BrewUp loop yet:

```
ProtocolRoute[] routes =
[
    new("eventstormer",
        "candidate-facts", "accepted-facts"),
    new("storyteller",
        "candidate-scenarios", "accepted-scenarios"),
    new("command-event-writer",
```

```
        "candidate-behavior", "accepted-behavior"),
    new("context-mapper",
        "candidate-context-map", "accepted-context-map")
];
```

A route identifies the specialist together with its candidate and accepted artifact kinds. Prompts, models, and domain decisions remain outside the protocol. Changing a route does not change what might cross the gate.

Encode the conversation state

The conversation also needs a serializable state that is independent of the transcript. Use a small `enum` and a record. The loop will persist that state together with its current input, processed run commands, and trace:

```
public enum ProtocolStatus
{
    Running,
    AwaitingReview,
    CorrectionRequired,
    Stopped,
    Completed
}

public sealed record ProtocolState(
    string WorkflowId,
    int StageIndex,
    ProtocolStatus Status,
    string LastMessageId,
    ArtifactEnvelope? Candidate,
    ArtifactEnvelope[] AcceptedArtifacts,
    string[] PreservedUnresolved,
    string[] ProcessedMessageIds,
    string? StopReason = null);
```

`StageIndex` identifies the active route. `Candidate` exists during review or correction, and `AwaitingReview` makes proposal and acceptance visibly different authorities. `AcceptedArtifacts` records authoritative handoffs. `LastMessageId` and `ProcessedMessageIds` provide causal ordering and idempotent delivery, while the enclosing snapshot makes those guarantees survive restart.

Starting a run creates an explicit causal root rather than leaving the first message unanchored:

```
public ProtocolState Start(string workflowId) => new(
    workflowId,
    StageIndex: 0,
    ProtocolStatus.Running,
    LastMessageId: "workflow-start",
    Candidate: null,
    AcceptedArtifacts: [],
    PreservedUnresolved: [],
    ProcessedMessageIds: []);
```

Table 10.2 makes the state machine reviewable before you implement it. Read a row as a rule: a message of this kind is legal only in the listed statuses, only from the listed sender, and it moves the workflow to the status in the last column. The sender column is a decision-right boundary. The transition column is the only state change the orchestrator may apply for that message. The last row is the idempotency rule: a message that has already been applied is ignored rather than blocked, so a retried delivery changes nothing.

Message	Valid status	Authorized sender	Resulting status
`CandidateProduced`	Running or `CorrectionRequired`	Current specialist	`AwaitingReview`
`GateAccepted`	`AwaitingReview`	Human reviewer	Running or Completed
`GateReferred`	`AwaitingReview`	Human reviewer	`CorrectionRequired`
`GateRejected`	`AwaitingReview`	Human reviewer	Stopped
`AuthorityRequired`	Running or `CorrectionRequired`	Current specialist	Stopped
`Duplicate` message ID	Any	Any	Unchanged; ignored

Table 10.2 – Legal messages in the conversation protocol

Figure 10.2 shows the same machine as a diagram. Solid arrows are human decisions and dashed arrows are specialist messages, so the two kinds of authority stay visible at a glance.

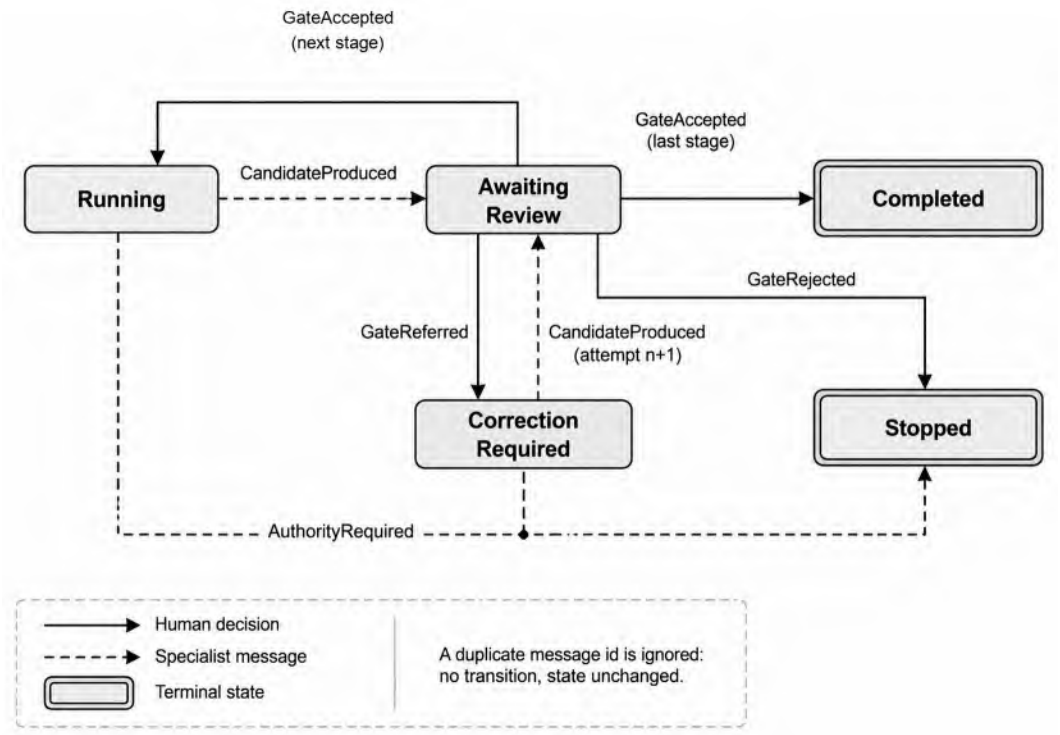


Figure 10.2 – Legal transitions of ProtocolStatus

Allowing only legal transitions

`ConversationProtocol.Handle` is the single entry point. Duplicate detection comes first so a retried delivery is harmless. Workflow, recipient, and causal-order checks follow before any transition-specific rule:

```
public ProtocolResult Handle(
    ProtocolState state,
    ProtocolMessage message)
{
    if (state.ProcessedMessageIds.Contains(message.MessageId))
        return Ignored(state,
            $"message {message.MessageId} was already applied");

    if (message.WorkflowId != state.WorkflowId)
```

```
            return Blocked(state, "workflow id does not match");

        if (message.Recipient != "orchestrator")
            return Blocked(state,
                "protocol messages must target the orchestrator");

        if (message.CausationId != state.LastMessageId)
            return Blocked(state,
                $"expected causation {state.LastMessageId}");
        if (state.Status is ProtocolStatus.Stopped
            or ProtocolStatus.Completed)
            return Blocked(state, $"workflow is {state.Status}");

        return message.Kind switch
        {
            ProtocolMessageKind.CandidateProduced
                => RecordCandidate(state, message),
            ProtocolMessageKind.GateAccepted
                => AcceptCandidate(state, message),
            ProtocolMessageKind.GateReferred
                => ReferCandidate(state, message),
            ProtocolMessageKind.GateRejected
                => StopFromGate(state, message),
            ProtocolMessageKind.AuthorityRequired
                => StopForAuthority(state, message),
            _ => Blocked(state, "unsupported message kind")
        };
    }
```

A blocked message leaves state unchanged. A blocked message is different from Stopped. BLOCKED means that the attempted message was not legal and therefore never became part of the conversation. The STOPPED state is an applied protocol outcome that records why the workflow cannot continue. Operators need that distinction when deciding whether to fix a delivery or obtain a domain decision. *Table 10.3* states it in two rows.

Outcome	What happened	Workflow state	What the operator does
BLOCKED	The message was not legal, so it never became part of the conversation and nothing was applied.	Unchanged	Fix the delivery and send a legal message.
STOPPED	An applied outcome that records why the workflow cannot continue.	Stopped, and persisted with the reason	Obtain the missing domain decision, then start a causally linked workflow.

Table 10.3 – A blocked message and a stopped workflow

`RecordCandidate` validates the active route, sender, provenance, artifact kind, evidence, and correction attempt before storing the candidate and moving to `AwaitingReview`. It never invokes the next specialist:

```
if (state.Status is not (ProtocolStatus.Running
    or ProtocolStatus.CorrectionRequired))
    return Blocked(state,
        $"CandidateProduced is not valid while {state.Status}");
if (message.Sender != route.Specialist)
    return Blocked(state, $"expected sender {route.Specialist}");
if (artifact.Kind != route.CandidateKind)
    return Blocked(state,
        $"expected artifact kind {route.CandidateKind}");
if (artifact.EvidenceRefs.Length == 0)
    return Blocked(state, "candidate has no evidence references");

var next = AppliedState(
    state with
    {
        Status = ProtocolStatus.AwaitingReview,
        Candidate = artifact
```

```
    },
    message);
return new(next, true,
    $"APPLIED: awaiting review of {artifact.Kind}");
```

Accepting a candidate and advancing

Acceptance is the only message that advances `StageIndex`. Acceptance requires the human reviewer, the expected accepted artifact kind, verified content and decision hashes, provenance that cites the candidate under review, a matching attempt, and preserved unresolved identifiers:

```
if (!IsSha256(accepted.HumanDecisionSha256))
    return Blocked(state,
        "accepted artifact human decision hash is missing or invalid");
if (accepted.SourceArtifactSha256 is not { Length: > 0 } sources ||
    !sources.Contains(state.Candidate.ContentSha256))
    return Blocked(state,
        "accepted artifact does not cite the candidate under review");
if (accepted.Attempt != state.Candidate.Attempt)
    return Blocked(state, "accepted artifact attempt does not match");

var lost = state.Candidate.Unresolved
    .Except(accepted.Unresolved)
    .ToArray();
if (lost.Length > 0)
    return Blocked(state,
        $"accepted artifact drops unresolved: {string.Join(", ", lost)}");
```

ES-17 need not remain unresolved forever, but resolving it requires a recorded human decision and a new accepted artifact. Omitting the identifier during acceptance would not constitute that transition.

Referring a candidate without advancing

A referral keeps the same stage and accepted input, records the human reason, and changes the status to `CorrectionRequired`. The specialist must answer with a causally linked message and a higher Attempt. Its corrected candidate returns to `AwaitingReview`, so the stage can loop without granting self-acceptance.

Stopping when authority is missing

Some obstacles the team cannot resolve through correction. Suppose `EventStormer` can prove that the Stock walkthrough contains inconsistent short-quantity behavior, but no authorized owner can decide which policy should govern future scenarios. Asking `EventStormer` to choose would convert an evidence specialist into a policy authority. Retrying would produce more text around the same missing decision.

Represent that condition with `AuthorityRequired`. It is an applied protocol outcome: the state becomes `Stopped`, and `StopReason` becomes part of the persisted snapshot and trace.

STOPPED is terminal for this protocol instance. Continuing the work requires an explicit operational action, such as recording the missing decision and starting a causally linked workflow. Mutating Stopped back to Running would erase the boundary that made the stop trustworthy.

Testing the protocol invariants

The suite focuses on failures that can corrupt orchestration state: duplicate delivery, stale causation, lossy handoffs, broken candidate linkage, and fabricated provenance. Redelivering the same work must not create a second interpretation.

Duplicates are ignored because their effect is already present; messages with the wrong cause are blocked because they do not belong at the current point in the conversation. Neither condition is a domain failure, so neither stops the workflow.

Finally, prove that an accepted envelope cannot make uncertainty disappear:

```
[Fact]
public void AcceptedArtifactCannotDropAnUnresolvedItem()
{
    var awaitingReview = AwaitingReview();

    var result = _protocol.Handle(
        awaitingReview,
        Accepted("gate-1", "candidate-1", []));

    Assert.False(result.Applied);
    Assert.Contains("drops unresolved: ES-17",
        result.Message);
    Assert.Equal(0, result.State.StageIndex);
}
```

Run only the protocol suite while iterating, then run the complete *Chapter 10* test project. On the final branch, the full project also includes loop and repository-replay tests:

```
$ dotnet test src/Orchestration/BrewUp.Orchestration.Tests/
BrewUp.Orchestration.Tests.csproj --nologo \
    --filter FullyQualifiedName~ConversationProtocolTests
Passed! - Failed: 0, Passed: 9, Skipped: 0, Total: 9

$ dotnet test src/Orchestration/BrewUp.Orchestration.Tests/
BrewUp.Orchestration.Tests.csproj --nologo
Passed! - Failed: 0, Passed: 25, Skipped: 0, Total: 25
```

The nine protocol tests cover legal transitions, sender authority, causation, idempotency, correction attempts, unresolved preservation, candidate linkage, and human-decision provenance. With the four eligibility tests, they protect the boundary before and during each exchange.

The protocol now defines the legal vocabulary and state machine around each specialist transformation. The takeaway is that authority has become a property of the message rather than of the tone of a reply. A transition happens because a typed, causally linked message from an authorized sender says so, or it does not happen at all. The next section connects that machinery to the accepted *Chapter 9* artifacts and runs the complete chain.

The BrewUp orchestration loop

The protocol from the previous section can validate a candidate, apply a human gate, and advance one route, but it cannot invoke a specialist or assemble its workspace. `BrewUpOrchestrationLoop` adds those operational steps without moving generative judgment into the protocol.

The complete Stock Management chain runs first through deterministic test fixtures and then through the repository-replay test against the verified *Chapter 9* records. The focused tests make one correction visible and verify the recorded authority and provenance behind the chain.

Building the loop around the protocol

Separate three responsibilities. `ConversationProtocol` decides whether a message might change state. `ISpecialistRunner` invokes a specialist and returns a candidate envelope. `BrewUpOrchestrationLoop` assembles input, invokes the runner, submits the result to the protocol, and records the transition.

Add `BrewUpOrchestrationLoop.cs`. A stage declares the artifact kinds that cross its two boundaries, and `SpecialistInput` distinguishes the primary handoff from the accepted context that supports it:

```
public sealed record LoopStage(
    string Specialist,
    string InputKind,
    string CandidateKind,
    string AcceptedKind);

public sealed record SpecialistInput(
    ArtifactEnvelope Primary,
    ArtifactEnvelope[] Context);

public interface ISpecialistRunner
{
    Task<ArtifactEnvelope> RunAsync(
        LoopStage stage,
        SpecialistInput input,
        int attempt,
        CancellationToken cancellationToken = default);
}
```

The interface returns a candidate envelope. A concrete runner might prepare files, render a prompt, invoke a model, validate output, and persist the result. `CandidateProduced` and the human gate still control whether that result advances.

Tip

Keeping model invocation outside the protocol

A state machine should remain deterministic. The loop owns model invocation; `ConversationProtocol` validates the resulting envelope. Replaying protocol state therefore does not trigger fresh model output.

The loop projects the stage definitions into the `ProtocolRoute` contracts already used by `ConversationProtocol`; it does not introduce a second state machine.

Staging the accepted artifacts

Chapter 9 already produced the handoff vocabulary. Do not ask the orchestrator to recreate it from prose; register the four stages explicitly:

```
LoopStage[] stages =
[
    new("eventstormer",
        "raw-evidence",
        "candidate-facts", "accepted-facts"),
    new("storyteller",
        "accepted-facts",
        "candidate-scenarios", "accepted-scenarios"),
    new("command-event-writer",
        "accepted-scenarios",
        "candidate-behavior", "accepted-behavior"),
    new("context-mapper",
        "accepted-behavior",
        "candidate-context-map", "accepted-context-map")
];
```

Each stage names the accepted input required to run, the candidate it might propose, and the accepted handoff that can unlock the next stage.

The initial envelope stages the original Stock walkthrough and its evidence inventory:

```
var rawEvidence = new ArtifactEnvelope(
    WorkflowId: "brewup-stock-01",
    Producer: "evidence-store",
    Kind: "raw-evidence",
    Path: "docs/ch09/run-pack/stock-walkthrough.md",
    EvidenceRefs: ["OBS-01", "OBS-02", "OBS-03"],
    Unresolved: ["ES-17"],
    Attempt: 0,
    ContentSha256: ArtifactHash.Sha256(
        repositoryRoot,
        "docs/ch09/run-pack/stock-walkthrough.md"));
```

The loop retains the original envelope and every accepted handoff. Each specialist receives the latest accepted artifact as Primary and earlier accepted material as Context. Before repository

replay, `RepositoryCatalog` verifies the artifact's recorded completion status and every declared source and decision hash, then extracts the evidence and unresolved identifiers.

Running the first specialist and pausing for review

`RunCurrentAsync` is the side-effect boundary. The method rejects duplicate command IDs, illegal states, and wrong input kinds before invoking `ISpecialistRunner`. Checking `CandidateProduced` after the call would be too late because the model could already have run twice.

```
public async Task<ProtocolResult> RunCurrentAsync(
    string commandId,
    CancellationToken cancellationToken = default)
{
    if (_processedRunCommands.Contains(commandId))
        return Ignored(
            $"run command {commandId} was already applied");
    if (State.Status is not (ProtocolStatus.Running
        or ProtocolStatus.CorrectionRequired))
        return Blocked(
            $"cannot run a specialist while {State.Status}");
    var stage = CurrentStage;
    if (_currentInput.Kind != stage.InputKind)
        return Blocked(
            $"{stage.Specialist} requires {stage.InputKind}, " +
            $"not {_currentInput.Kind}");
```

After those checks, the loop calculates the attempt, assembles context, and invokes the runner:

```
    var attempt = State.Status == ProtocolStatus.CorrectionRequired
        ? (State.Candidate?.Attempt ?? 0) + 1
        : 1;
    var context = State.StageIndex == 0
        ? []
        : new[] { _initialInput }
            .Concat(State.AcceptedArtifacts[..^1])
            .ToArray();
    var candidate = await _runner.RunAsync(
        stage,
        new SpecialistInput(_currentInput, context),
```

```
            attempt,
            cancellationToken);
```

The loop wraps the returned candidate in `CandidateProduced` and sends it through `ConversationProtocol`. Only an applied protocol result marks the run command as processed and enters the trace:

```
        var message = new ProtocolMessage(
            MessageId: $"{commandId}-candidate",
            WorkflowId: State.WorkflowId,
            CausationId: State.LastMessageId,
            Kind: ProtocolMessageKind.CandidateProduced,
            Sender: stage.Specialist,
            Recipient: "orchestrator",
            Artifact: candidate);
        var result = _protocol.Handle(State, message);
        if (!result.Applied)
            return result;
        _processedRunCommands.Add(commandId);
        Apply(result, message, stage.Specialist, "candidate-produced");
        return result;
    }
```

Calling `RunCurrentAsync("run-1")` invokes `EventStormer` once and leaves the workflow in `AwaitingReview`. Storyteller does not start; the candidate and its provenance are now available to the reviewer.

The pause is what everything else in the chapter depends on, so it is worth seeing from the outside. Rendering the state after that single call shows the reviewer what the decision is about, and nothing else:

```
workflow brewup-stock-01    stage 1 of 4    status AwaitingReview

candidate      artifacts/candidate-facts-attempt-1.md
produced by    eventstormer, attempt 1
kind           candidate-facts
evidence       OBS-01, OBS-02, OBS-03
unresolved     ES-17
content hash   aaaaaaaa...  (deterministic fixture value)
caused by      run-1-candidate
```

```
decision required: accept, refer, or reject.
storyteller cannot start until this gate records one.
```

Nothing in that view is a domain opinion. The orchestrator reports which artifact is waiting, who produced it, on which evidence, which question is still open, and which decision the reviewer must make. ES-17 travels with the candidate, so accepting the candidate cannot quietly close it.

Note

Making command claims durable before distribution

Persisted state prevents an applied command from running again after restart. In production, claim the command ID durably before a remote call and persist the candidate through an outbox or equivalent. Protocol idempotency prevents duplicate state transitions, not exactly-once execution of an external model.

Advancing through the four BrewUp stages

A human-review adapter reads the recorded decision and accepted artifact, then calls `AcceptCurrent`. That method records and applies the external decision; `ConversationProtocol` still verifies reviewer identity, accepted kind, provenance, evidence, and unresolved preservation.

In the repository replay, those accepted artifacts come from the verified *Chapter 9* handoffs with their exact content, source-output, and human-decision hashes.

Each stage follows the same rhythm: produce a candidate, pause for review, and advance only after `GateAccepted`. `JsonWorkflowStore` persists state, current input, processed commands, and trace atomically so another process can resume without interpreting a transcript.

Note

Framework lens – Spec Kit: what represents a stage going backward?

In the *Chapter 8* Spec Kit run, the sequence from specification to plan and tasks was inspectable in one direction: each artifact could be reviewed before the next command ran. That recorded workflow did not preserve a first-class referral, correction attempt, or stop naming the missing authority. The orchestration states introduced here make that backward movement explicit without rerunning accepted stages.

Correcting one stage without restarting the chain

The first Storyteller candidate closes the short-quantity story too neatly. The reviewer refers it with the reason "*Separate the observed shortage from the replenishment policy.*" This call records the referral that separates the observed shortage from the replenishment policy: `GateReferred` changes the status to `CorrectionRequired` but leaves `StageIndex` at Storyteller and preserves the accepted facts that fed attempt 1.

```
var referred = loop.ReferCurrent(
    "refer-storyteller",
    "Separate the observed shortage from the " +
    "replenishment policy.");

var corrected = await loop.RunCurrentAsync(
    "run-storyteller-2");

Assert.True(referred.Applied);
Assert.True(corrected.Applied);
Assert.Equal(
    ["eventstormer", "storyteller", "storyteller"],
    runner.Invocations);
Assert.Equal(1, loop.State.StageIndex);
Assert.Equal(2, loop.State.Candidate!.Attempt);
```

The invocation assertion proves that `EventStormer` did not repeat, Storyteller ran exactly twice, and Command & Event Writer did not start early. The corrected candidate still returns to AwaitingReview; the correction does not grant self-acceptance.

Note

Using a new command ID for a deliberate retry

Redelivering run-Storyteller-1 is a duplicate, and the protocol must ignore it. The referral creates new work, so the correction uses a new command ID and Attempt 2. Idempotency and retry are different operations.

Persisting a replayable run trace

A final status alone cannot explain how the workflow reached that state. Record every applied transition with enough information to order it, connect it to its cause, locate its artifact, and inspect the uncertainty that crossed the boundary. Persist that trace together with the state required to resume:

```
public sealed record WorkflowSnapshot(
    ProtocolState State,
    ArtifactEnvelope CurrentInput,
    string[] ProcessedRunCommands,
    RunTraceEntry[] Trace);

private void Persist() => _store.Save(new WorkflowSnapshot(
    State,
    _currentInput,
    [.. _processedRunCommands],
    [.. _trace]));
```

`JsonWorkflowStore` atomically replaces the previous snapshot. The persisted-resume test reconstructs the loop from that store and redelivers the same command:

```
var resumed = new BrewUpOrchestrationLoop(
    "brewup-stock-01",
    TestFixtures.Stages,
    TestFixtures.RawEvidence,
    resumedRunner,
    new SteppingClock(),
    store);
var replay = await resumed.RunCurrentAsync("run-1");

Assert.Equal(ProtocolStatus.AwaitingReview, resumed.State.Status);
Assert.Single(resumed.Trace);
Assert.False(replay.Applied);
Assert.StartsWith("IGNORED", replay.Message);
Assert.Empty(resumedRunner.Invocations);
```

The assertions show that the restarted loop reloads its `AwaitingReview` state and trace, ignores the duplicate command, and invokes no specialist. The separate referral test proves that a correction reruns only Storyteller while accepted `EventStormer` work remains intact.

Completing only after the final human gate

Context Mapper's candidate still leaves the workflow in `AwaitingReview`. The happy-path test proves that `ConversationProtocol` reaches Completed only after the final `GateAccepted` message is applied:

```
Assert.Equal(ProtocolStatus.Completed, loop.State.Status);
Assert.Equal(4, loop.State.AcceptedArtifacts.Length);
Assert.Equal(["ES-17"], loop.State.PreservedUnresolved);
```

The value `accepted=4` means that every stage has an authoritative handoff; `unresolved=ES-17` means that completion did not manufacture a policy. Completed says the planned chain and its gates finished, not that every domain question was answered.

Now, replay the Packt records through `RepositoryCatalog`, `GateVerifier`, and `RepositoryScenario`. The repository test requires exact complete status, recomputes every declared source and human-decision hash, and reproduces the recorded correction paths. These hashes establish artifact and gate-decision provenance, but they do not retroactively fill the candidate-level provenance gap identified in *Chapter 9* between Storyteller and Command & Event Writer:

```
var store = new JsonWorkflowStore(directory);
var loop = await RepositoryScenario.RunAsync(
    FindRepositoryRoot(),
    store);
var resumed = await RepositoryScenario.RunAsync(
    FindRepositoryRoot(),
    store);

Assert.Equal(ProtocolStatus.Completed, loop.State.Status);
Assert.Equal(4, loop.State.AcceptedArtifacts.Length);
Assert.Equal(40, loop.State.PreservedUnresolved.Length);
Assert.Equal(14, loop.Trace.Count);
Assert.Equal(
    loop.Trace.Select(entry => entry.MessageId),
    resumed.Trace.Select(entry => entry.MessageId));
Assert.Equal(14, resumed.Trace.Count);
```

The 40 identifiers are the cumulative set of unresolved identifiers recorded across the four accepted *Chapter 9* handoffs, not newly discovered issues. They should not be read as forty

candidate-to-candidate lineage links. Loading the persisted fourteen-transition snapshot again performs no specialist work.

Rejection is the other terminal path: `GateRejected` records the human reason, leaves the loop Stopped, and blocks later run commands before specialist invocation.

Testing the orchestration loop end-to-end

The final suite covers terminal decisions, invalid inputs, duplicate commands, candidate leakage, trace causation, repository provenance, and persisted resume. The following assertion verifies the accepted context supplied to Context Mapper:

```
var input = runner.Inputs[^1];

Assert.Equal("accepted-behavior",
    input.Primary.Kind);
Assert.Equal(
    ["raw-evidence",
     "accepted-facts",
     "accepted-scenarios"],
    input.Context.Select(artifact => artifact.Kind));
```

Run the complete test project:

```
$ dotnet test src/Orchestration/BrewUp.Orchestration.Tests/
BrewUp.Orchestration.Tests.csproj --nologo
Passed! - Failed: 0, Passed: 25, Skipped: 0, Total: 25
```

These tests verify orchestration rather than scenario quality. They establish that a specialist ran, which bytes it received, which message caused the transition, and which unresolved identifiers survived.

The BrewUp chain is now executable without turning the orchestrator into a fifth modeling specialist. It routes accepted inputs, pauses for human decisions, preserves correction history, and stops when authority is missing. If you take one thing from this chapter, take the line the orchestrator never crosses: it can automate coordination because coordination is mechanical. The orchestrator must leave authority to people because authority is a judgment about the domain. The remaining design question is how much freedom a specialist can exercise without drifting from accepted evidence, shared language, or explicit decision rights.

Summary

This chapter turned the manual handoffs from *Chapter 9* into an executable coordination layer. `EligibilityEvaluator` blocked unauthorized input. `ConversationProtocol` enforced typed messages, legal states, causation, idempotency, and human gates. `BrewUpOrchestrationLoop` ran the four stages, referred the affected stage, persisted a replayable trace, and completed after the final accepted handoff. The repository replay finished with four accepted artifacts and all 40 unresolved identifiers preserved – the cumulative set recorded across the four accepted *Chapter 9* handoffs, not forty newly discovered issues.

In this chapter, you learned how to:

- Separate orchestration decisions from specialist transformations and human domain authority.
- Use typed messages, legal states, and causation to govern multi-agent collaboration.
- Block duplicate, stale, or lossy handoffs from changing authoritative state.
- Correct one stage without repeating accepted work or starting the next stage.
- Persist and replay a run while preserving unresolved questions.

Chapter 11 builds on this control plane by defining governed autonomy: what specialists might explore, how semantic drift becomes visible, and which decisions still require human authority.

11

Governed Autonomy: Balancing Autonomy and Alignment Across Agents

Chapter 10 established who may act, what accepted material each specialist may consume, and when a human gate must intervene. It left one question open: how far may a specialist interpret accepted evidence before it changes the domain?

This chapter makes that interpretive freedom explicit. The question now moves inside the transformation itself: how much may a specialist interpret, rename, structure, or generalize before a useful proposal becomes an unsupported domain decision? We encode that freedom as policy, separate mechanical violations from semantic divergence, and run a controlled drift-and-correction scenario through the BrewUp loop while keeping evidence, uncertainty, decision rights, and human acceptance traceable.

In this chapter, you will learn how to:

- Define enforceable autonomy policies for specialist agents without prescribing their domain conclusions.
- Assemble shared alignment context from accepted evidence, approved language, unresolved questions, and explicit decision rights.
- Distinguish deterministic boundary violations from semantic divergence that requires authorized review.
- Bound correction attempts, time, and tool access while keeping correction local, traceable, and subject to human acceptance.

Technical requirements

The implementation in this chapter extends the orchestration workspace introduced in *Chapter 10*. The preserved version is available on the companion repository branch `https://github.com/PacktPublishing/AI-Augmented-Domain-Driven-Design/tree/ch11/governance`.

The implementation targets .NET 10. The governance types are in `src/Orchestration/BrewUp.Orchestration.Application. Governance.cs contains AutonomyPolicy and AlignmentContext, AlignmentGuard.cs` contains the deterministic alignment checks, and `SemanticAlignment.cs` defines `ISemanticAlignmentReviewer`. The controlled semantic-review implementation is `MarkdownSemanticAlignmentReviewer.cs` in `BrewUp.Orchestration.Infrastructure`.

The corresponding executable examples and tests are in `src/Orchestration/BrewUp.Orchestration.Tests`, primarily `AlignmentGuardTests.cs, SemanticAlignmentTests.cs`, and `GovernedOrchestrationLoopTests.cs`. You can follow the argument from the listings in this chapter; you need to clone the repository only to inspect or run the preserved implementation and its complete test suite.

From coordination to decision rights

The orchestration loop from *Chapter 10* made coordination explicit: it determines which specialist may run, which accepted material that specialist may consume, when a **human gate** is required, and which transitions are legal. It still does not define how much freedom a specialist has inside its transformation.

A Storyteller must be free to turn accepted facts into concrete situations. A Command & Event Writer must choose a precise form for intent or outcomes supported by evidence. A Context Mapper must be able to propose boundaries not already drawn in the input. Without such freedom, a generative specialist would add little value.

That freedom also creates risk. A specialist may introduce a cleaner term, infer an owner, close an unresolved scenario, or make evidence look more consistent than it is. The artifact may satisfy its schema, and the conversation may follow every legal transition while the meaning has already drifted.

The practical question, then, is which choices belong to a specialist, and which would change accepted domain knowledge.

The orchestrator cannot answer by interpreting the domain without becoming another modeling specialist. The orchestrator can encode the permitted freedom, reject mechanical violations, and return semantic disagreements to the people authorized to resolve them.

Turning autonomy into policy

Autonomy is easier to govern when treated as a contract rather than as a property of the model. Each specialist already has a bounded transformation. The policy describes the operational freedom inside that boundary. It specifies which artifact the specialist may produce, which accepted material it may inspect, whether it may introduce terminology, which tools it may use, and how many correction attempts the workflow permits.

```
public sealed record AutonomyPolicy(
    string Specialist,
    string OutputKind,
    string[] AllowedEvidenceKinds,
    bool MayIntroduceNewTerms,
    int MaxCorrectionAttempts,
    TimeSpan Timeout,
    string[] AllowedTools);
```

The Storyteller policy makes these limits concrete for one specialist:

```
var storytellerPolicy = new AutonomyPolicy(
    Specialist: "storyteller",
    OutputKind: "candidate-scenarios",
    AllowedEvidenceKinds: ["accepted-facts", "walkthrough"],
    MayIntroduceNewTerms: true,
    MaxCorrectionAttempts: 2,
    Timeout: TimeSpan.FromMinutes(2),
    AllowedTools: ["read-accepted-artifact"]);
```

A Context Mapper may inspect the complete accepted chain because its transformation depends on decisions expressed across several artifacts.

The policy bounds exploration without dictating the conclusion. `MayIntroduceNewTerms` controls whether proposing unfamiliar terminology is part of the specialist's permitted transformation. When it is `false`, introducing a term outside `ApprovedTerms` is a mechanical policy violation, and the guard can reject it without interpreting its meaning. When it is true, the specialist may propose the term, but the proposal does not become part of the accepted

ubiquitous language. Its relationship to existing language remains subject to semantic review and, ultimately, authorized human acceptance.

Likewise, access to an accepted artifact grants the specialist permission to transform it, not permission to revise the decision that made it accepted.

Defining the shared alignment inputs

Local autonomy still depends on a shared frame of accepted knowledge. If each stage reconstructs that frame differently, the orchestrator may enforce a procedurally valid workflow while the agents gradually diverge in meaning.

Shared alignment inputs define the constraints every transformation inherits: accepted artifacts, the current goal, approved language, unresolved questions that must remain visible, and decisions the current specialist is not authorized to make.

The following record represents that shared frame:

```
public sealed record AlignmentContext(
    ArtifactEnvelope[] AcceptedArtifacts,
    string WorkflowGoal,
    string[] ApprovedTerms,
    string[] Unresolved,
    string[] ProhibitedDecisionTypes);
```

`AutonomyPolicy` defines what a specialist may do while producing a candidate; `AlignmentContext` defines what the transformation must remain consistent with.

For the Storyteller, for example, the context might contain accepted-facts, the Stock walkthrough goal, the currently approved domain terms, the unresolved short-quantity question, and a prohibition against introducing a replenishment or allocation policy.

```
var storytellerAlignment = new AlignmentContext(
    AcceptedArtifacts: state.AcceptedArtifacts,
    WorkflowGoal:
        "Turn accepted Stock facts into concrete scenarios " +
        "without resolving unsupported policy.",
    ApprovedTerms:
    [
        "Stock",
        "Sales",
        "set aside",
        "hold",
```

```
        "earmark"
    ],
    Unresolved:
    [
        "ES-17"
    ],
    ProhibitedDecisionTypes:
    [
        "short-quantity-policy",
        "retry-owner",
        "hold-duration-policy"
    ]);
```

These fields do not tell Storyteller how to write a scenario. The Storyteller may choose actors, sequence supported actions, select wording, and expose variations, but it cannot turn ES-17 into a rule, replace unresolved language with a canonical term, or assign unsupported ownership.

The same context changes as the workflow advances. Command & Event Writer receives accepted scenarios in addition to the earlier material. Context Mapper receives the complete accepted chain. The goal becomes more specific to each transformation, while accepted knowledge and unresolved decisions accumulate rather than being recreated from memory.

`GovernanceCatalog` replaces `AcceptedArtifacts` with the protocol state's accepted records. `WorkflowGoal`, `ApprovedTerms`, unresolved, and `ProhibitedDecisionTypes` remain part of the specialist governance configuration; they might be populated from accepted language and handoff records in a production integration, but the orchestrator does not infer them from prose.

With these inputs explicit, the next boundary becomes enforceable. Code can reject a specialist that exceeds a declared permission, but it must stop short of deciding whether a permitted interpretation is semantically sound. That distinction separates mechanical alignment from **semantic divergence**.

Separating mechanical violations from semantic divergence

The alignment inputs are now explicit, so part of the frame can be checked by code. Artifact kind, provenance, unresolved identifiers, and declared decision types are facts a program can compare. Whether a new term collapses a distinction the domain has not resolved is not such a

fact. The two failures therefore need different machinery, because the first can be rejected by a declared rule and the second can only be reported to an authority.

Enforce mechanical alignment rules

Some alignment failures do not require interpretation. If Storyteller returns the wrong artifact kind, cites evidence that was never supplied, drops an unresolved identifier, or claims a decision type that its contract explicitly forbids, the violation does not require model interpretation. These breaches violate declared boundaries, and deterministic code should detect them.

The alignment check remains separate from the conversation protocol. `ConversationProtocol` decides whether a message is legal; an **alignment guard** decides whether the candidate carried by that message respects the mechanical constraints of the current transformation. The two overlap once, deliberately: *Chapter 10* already refuses an accepted artifact that drops an unresolved identifier, but it refuses it at the gate, after a reviewer has read the candidate. Checking it again in production means the specialist never presents a candidate whose open questions have quietly closed.

The candidate therefore exposes a small amount of machine-readable alignment metadata alongside its normal artifact:

```
public sealed record CandidateAlignmentMetadata(
    string[] Terms,
    string[] DecisionTypes);

public sealed record AlignmentIssue(
    string Code,
    string Detail);
```

`AlignmentGuard` compares declared facts using explicit ordinal comparisons. The following excerpt shows three of the five checks; the complete guard also checks artifact kind and unfamiliar terminology:

```
var knownEvidence = context.AcceptedArtifacts
    .SelectMany(artifact => artifact.EvidenceRefs)
    .ToHashSet(StringComparer.Ordinal);

var unknownEvidence = candidate.EvidenceRefs
    .Where(id => !knownEvidence.Contains(id))
    .ToArray();
```

```
var lostUnresolved = context.Unresolved
    .Except(
        candidate.Unresolved,
        StringComparer.Ordinal)
    .ToArray();

var forbiddenDecisions = metadata.DecisionTypes
    .Intersect(
        context.ProhibitedDecisionTypes,
        StringComparer.Ordinal)
    .ToArray();
```

Table 11.1 states each check as the boundary it protects. Every row compares declared facts, so none of them requires the guard to interpret domain prose.

Issue code	Comparison	Deterministic basis
wrong-artifact-kind	`candidate.Kind` against `stage.CandidateKind`	The stage declares one expected kind.
unknown-evidence	`candidate.EvidenceRefs` against accepted evidence	Evidence identifiers either exist in the supplied set or do not.
dropped-unresolved	`context.Unresolved` against `candidate.Unresolved`	The guard computes a set difference.
unapproved-term	`metadata.Terms` against `context.ApprovedTerms` when `policy.MayIntroduceNewTerms` is false	The policy explicitly determines whether unfamiliar terminology may be proposed.
prohibited-decision	`metadata.DecisionTypes` against `context.ProhibitedDecisionTypes`	The contract names the decision types withheld from the specialist.

Table 11.1 – The five mechanical alignment checks and the boundary each one protects

The last check deserves one qualification. The guard does not inspect the historical `decision_owner` field from the *Chapter 9* artifacts and try to infer what that field means.

Chapter 9 recorded why: the field is defined as decision authority, but the 9.4 contract used it to identify an event's causal actor, and that output was kept as run evidence rather than reused. Mechanical alignment therefore works from the explicit decision rights declared for the current transformation, rather than silently normalizing historical evidence.

The guard can prove that a candidate changed artifact type, cited unknown evidence, dropped ES-17, or declared a forbidden decision. Questions of meaning – whether two phrases are equivalent or an observation has become policy – remain review material for an authorized reviewer.

Surfacing semantic divergence without resolving it

Suppose Storyteller receives accepted material in which set aside, hold, and earmark coexist without being established as synonyms and then produces a coherent scenario using only reserve. The artifact kind and evidence references are valid, and no unresolved identifier has disappeared, yet the new term might collapse a distinction the domain has not resolved. The orchestrator cannot decide whether that abstraction is useful or lossy.

Rejecting every unfamiliar term would prevent specialists from contributing to the language. Automatically replacing one term with another would silently decide that the concepts are equivalent.

The safer alternative is to represent the difference itself as a reviewable outcome:

```
public sealed record SemanticDivergence(
    string Kind,
    string CandidateValue,
    string[] RelatedAcceptedValues,
    string Reason);
```

A language-aware check can therefore report a candidate term without deciding whether it is wrong:

```
var divergence = new SemanticDivergence(
    Kind: "candidate-term",
    CandidateValue: "reserve",
    RelatedAcceptedValues:
    [
        "set aside",
        "hold",
        "earmark"
    ],
    Reason:
```

```
        "The candidate introduces a term that is not " +
        "part of the accepted language.");
```

The term remains provisional until an authorized review accepts it. A Context Mapper, for example, might need to name a responsibility before the business has agreed on its best name.

The workflow attaches semantic divergence to the candidate. The human gate can accept the interpretation, refer it for correction, reject it, or update the accepted vocabulary or decision record.

In the controlled run, Storyteller crosses this boundary, the orchestrator records the divergence without repairing it, and the correction returns to Storyteller.

Correcting drift without expanding autonomy

Declared rules and reported divergences are only useful if the workflow can act on them without acquiring authority it does not have. This section runs one deliberate drift through the BrewUp loop, keeps the correction local to the specialist that produced it, bounds what a retry may change, and tests the conditions that keep the outcome reviewable.

Running a controlled drift and correction scenario

Chapter 10 already referred a Storyteller candidate on a reviewer's own reading of it; the referral mechanism here is the same. What is new is what produces the referral. The accepted facts still contain ES-17: Andrea rejects a request when the full quantity is unavailable, while Nadia sets aside what is available. No accepted policy chooses between those practices.

For this run, let Storyteller produce a candidate that is structurally valid but semantically too certain:

```
Sales requests twelve cases but only eight are available.
Stock reserves the eight available cases and informs Sales
that the remaining quantity cannot be supplied.
```

Mechanically, the candidate is valid. The candidate has the expected artifact kind, cites only evidence supplied to Storyteller, preserves ES-17, declares no prohibited decision type, and introduces `reserve` under a policy that explicitly permits the specialist to propose new terminology. Nothing in those checks requires the guard to interpret what `reserve` means. *Chapter 9* had already named this particular invention: reserving eight was one of four moves that would supply a policy the source does not provide.

Despite passing those checks, the candidate has changed the meaning of the evidence. One observed behavior has become the behavior of Stock, and the new term reserve, borrowed from the *Chapter 8* implementation, makes reserve sound more established than it is.

That semantic change in meaning is not visible to the schema. In this run, a bounded semantic reviewer reports possible divergence:

```
public interface ISemanticAlignmentReviewer
{
    Task<SemanticDivergence[]> ReviewAsync(
        ArtifactEnvelope candidate,
        AlignmentContext context,
        CancellationToken cancellationToken = default);
}
```

The interface is observational: it cannot mutate the candidate, update accepted language, or advance the workflow.

For the deliberately drifting Storyteller attempt, the reviewer returns two findings:

```
DIVERGENCE candidate-term
value: reserve
related accepted terms: set aside, hold, earmark
reason: the candidate introduces a more precise term
        that has not been accepted

DIVERGENCE possible-policy-inference
unresolved: ES-17
reason: one observed short-quantity practice is expressed
        as the general Stock behavior
```

The human gate receives these findings as review evidence, not as new domain facts. A production semantic reviewer might itself be generative; the controlled run uses a deterministic Markdown adapter. In either case, the reviewer reports divergence but cannot repair another specialist's candidate.

The orchestration loop therefore records the divergences beside the candidate and still moves to `AwaitingReview`.

```
var assessment = await semanticReviewer.ReviewAsync(
    loop.State.Candidate!,
    storytellerAlignment);
```

```
loop.RecordAlignmentAssessment(
    "alignment-storyteller-1",
    assessment);
```

The human reviewer can now inspect the candidate, the accepted evidence, and the reported divergence together. In this case, the reviewer refers the candidate:

```
loop.ReferCurrent(
    "refer-storyteller-alignment",
    "Do not turn Nadia's observed short-quantity practice " +
    "into the general Stock policy. Preserve both practices " +
    "and keep ES-17 unresolved.");
```

The trace retains the reserve candidate, its alignment findings, and the referral that prevented it from advancing.

Storyteller then runs again from the same accepted input:

```
await loop.RunCurrentAsync(
    "run-storyteller-alignment-2");
```

The corrected scenario now states the disagreement rather than resolving it:

```
Sales requests twelve cases but only eight are available.

In one observed case, Andrea refuses the request because the
full quantity cannot be supplied.

In another, Nadia sets aside the eight available cases and
has someone call Sales.

The evidence does not establish which practice should govern
future short-quantity requests.

ES-17 is still unresolved. No new owner has been introduced. The candidate does
not claim that set aside, hold, earmark, and reserve are interchangeable terms.
```

The correction therefore remained local to the transformation that produced the candidate. Storyteller revised its scenario; the orchestrator, semantic reviewer, and human reviewer preserved their own responsibility boundaries.

A focused test confirms that boundary:

```
Assert.Equal(
    ["eventstormer", "storyteller", "storyteller"],
    runner.Invocations);

Assert.Equal(2, loop.State.Candidate!.Attempt);

Assert.Contains(
    "ES-17",
    loop.State.Candidate.Unresolved);

Assert.Contains(
    loop.AlignmentTrace,
    entry => entry.MessageId ==
        "alignment-storyteller-1");

Assert.Contains(
    "Do not turn Nadia's observed short-quantity practice",
    loop.Trace.Single(
        entry => entry.MessageId ==
            "refer-storyteller-alignment").Reason);
```

The invocation sequence and attempt count preserve the *Chapter 10* guarantees: a semantic finding does not widen the specialist's authority. The last three assertions are the new ones – ES-17 survives the correction, the finding stays attached to the attempt that caused it, and the reviewer's reason stays in the trace.

Only after the corrected candidate passes through the human gate can the workflow advance.

The retry changes only Storyteller's candidate; it does not alter the evidence, governance contract, tools, or review responsibility. That separation makes the next requirement possible: retries can be bounded operationally without expanding the specialist's authority.

Bound correction attempts, time, and tool access

Interpretive freedom also needs operational limits. A specialist might explore several formulations, but it cannot retry indefinitely, expand its tool access after failure, or leave the workflow suspended by an unbounded model call.

These limits constrain execution without deciding domain meaning. The autonomy policy already carries them: the correction budget, timeout and cancellation, tool access, and a terminal state when the correction budget is exhausted.

The loop checks those boundaries before invoking the specialist again.

```
if (attempt > policy.MaxCorrectionAttempts)
{
    return SpecialistExecutionResult.Stop(
        "correction-budget-exhausted",
        $"Correction budget exhausted for {policy.Specialist}.");
}

using var timeout =
    CancellationTokenSource.CreateLinkedTokenSource(
        cancellationToken);

timeout.CancelAfter(policy.Timeout);

var candidate = await runner.RunAsync(
    stage,
    input,
    attempt,
    timeout.Token)
    .ConfigureAwait(false);
```

A failed attempt does not widen the boundary; the correction budget, prompt constraints, tool set, and specialist contract remain unchanged.

`SpecialistToolRegistry` applies the same rule before invocation:

```
public string[] Resolve(AutonomyPolicy policy)
{
    if (_availableTools is null)
        return [.. policy.AllowedTools];

    var available = _availableTools
        .ToHashSet(StringComparer.Ordinal);

    return policy.AllowedTools
        .Where(available.Contains)
        .ToArray();
}
```

If Storyteller may inspect accepted evidence but not operational systems, every attempt receives the same tool boundary. A referral changes the reason for rerunning, not the specialist's permissions.

A timeout records the end of the execution window, not a judgment on domain meaning. The workflow then stops or retries according to policy.

An exhausted correction budget produces an explicit terminal state:

```
STOPPED
specialist: storyteller
reason: correction-budget-exhausted
attempts: 2 of 2, third refused
last-candidate: preserved
last-review-reason: preserved
```

The trace preserves failed attempts, findings, and referral reasons. An exhausted budget stops the workflow without changing its decision rights.

Testing autonomy without surrendering alignment

The autonomy tests do not judge whether Storyteller produced the best scenario or Context Mapper found the best boundary; those are modeling judgments. The tests instead establish whether the system preserves the conditions under which those judgments remain reviewable.

A focused guard test verifies that **unresolved knowledge** cannot disappear:

```
[Fact]
public void UnresolvedQuestionCannotDisappear()
{
    var candidate = Candidate(
        unresolved: []);

    var issues = AlignmentGuard.Check(
        candidate,
        candidate.AlignmentMetadata!,
        StorytellerStage,
        Governance.Policy,
        Alignment);

    Assert.Contains(
        issues,
        issue =>
```

```
            issue.Code ==
            "dropped-unresolved");
}
```

The suite also covers accepted terminology, wrong artifact kinds, unknown evidence, and prohibited decisions.

Semantic divergence requires a different assertion. With the candidate fixture already loaded, the focused assertions prove that the reviewer reports reserve without normalizing or rewriting the candidate:

```
var divergences = await reviewer.ReviewAsync(
    candidate.Artifact,
    Alignment());

var divergence = Assert.Single(
    divergences,
    item => item.Kind == "candidate-term");

Assert.Equal("reserve", divergence.CandidateValue);
Assert.Contains(
    "hold",
    divergence.RelatedAcceptedValues);
Assert.Equal(before, after);
Assert.Contains("Stock reserves", candidate.Text);
```

The final assertions compare the file contents before and after review and confirm that the original candidate wording is retained.

The policy-inference test applies the same boundary to Nadia's observed behavior. The semantic reviewer might report that one observed practice has been generalized into Stock policy, but it cannot rewrite the candidate or decide which practice is correct. Only after a human referral does Storyteller produce a different interpretation. Together, these tests establish the three authority boundaries used throughout the chapter: deterministic code enforces declared constraints, specialists produce interpretations, and human reviewers decide what may become accepted knowledge.

Operating the multi-agent system as a governed workflow

The mechanisms are now complete. *Figures 11.1* and *11.2* bring them together as a synthesis of the workflow developed across *Chapters 9*, *10*, and *11.* Specialists own transformations, deterministic code owns enforceable workflow and alignment rules, and human reviewers own acceptance and domain decisions. The second figure adds the correction path, showing that a rejected interpretation returns to the specialist that produced it rather than expanding the authority of the orchestrator or reviewer.

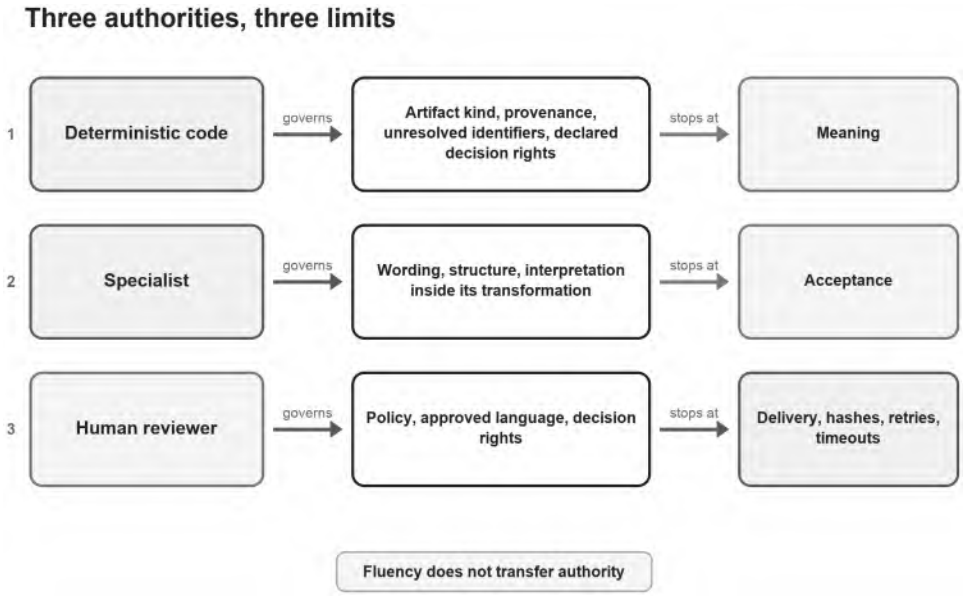


Figure 11.1 – Three authorities and the limit of each one

The value of orchestration here lies in keeping responsibility explicit. *Figure 11.2* shows the separation the trace records.

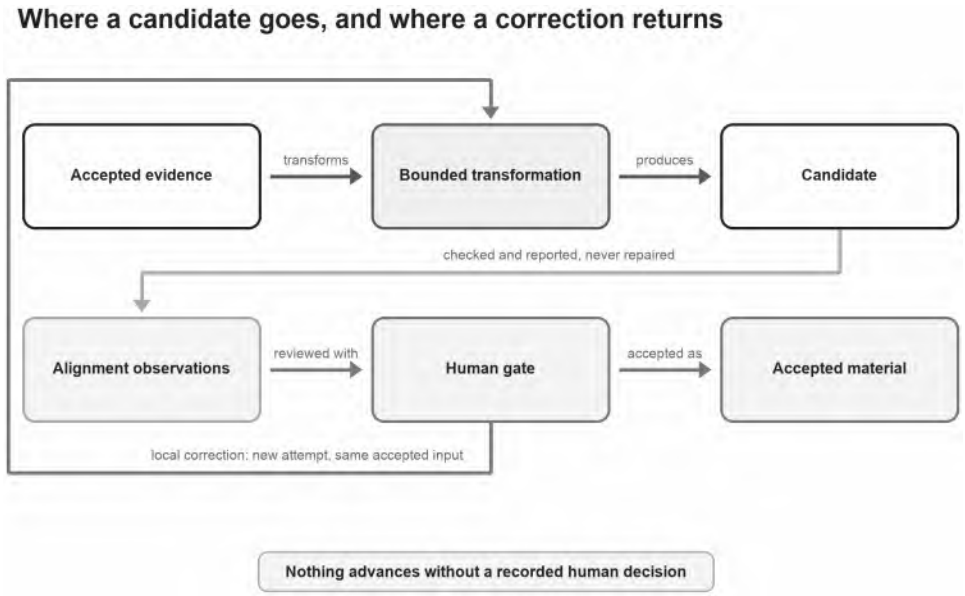


Figure 11.2 – Where a candidate goes, and where a correction returns

`Completed` describes the agreed modeling process, not a domain in which every question has been answered. The specialists have transformed accepted material, required reviews have taken place, accepted artifacts are traceable, and remaining uncertainty is visible.

That distinction also closes the technical journey of this book. We began by using AI to explore domain material and generate candidate interpretations. We then made those interpretations increasingly accountable: first to evidence and shared language, then to bounded specialist responsibilities, explicit handoffs, orchestration rules, and finally governed interpretive freedom. The goal was never to make the model authoritative. It was to make its contribution useful without losing sight of where knowledge came from, which questions remain open, who made each decision, and why an artifact was allowed to become accepted domain knowledge.

Summary

This chapter completed the orchestration model by turning specialist freedom into a reviewable contract. `AutonomyPolicy` constrains evidence access, outputs, terminology, correction attempts, time, and tools; `AlignmentContext` carries accepted artifacts and configured governance constraints. In the Storyteller run, deterministic guards preserved declared invariants, semantic review exposed a plausible policy inference, and the workflow corrected the candidate locally before human acceptance.

In this chapter, you learned how to:

- Grant specialists bounded freedom without granting domain authority.
- Use deterministic guards for declared invariants and semantic review for changes in meaning.
- Keep correction attempts, time, and tools inside explicit operational policy.
- Preserve evidence, uncertainty, attribution, and human acceptance across the workflow.

Across the book, the focus has shifted from producing AI-generated outputs to preserving the chain that makes those outputs trustworthy. Raw observations became candidate facts. Candidates crossed human gates before becoming accepted inputs; unresolved questions such as ES-17 survived multiple transformations; corrections returned to the specialist who introduced the drift; and every accepted decision remained attributable to evidence and human authority. Agents accelerated discovery, transformation, and coordination, but they never acquired the authority to turn a plausible interpretation into domain knowledge on their own.

12

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:







Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 12.1* on desktop), search for this book by name and select the correct edition.

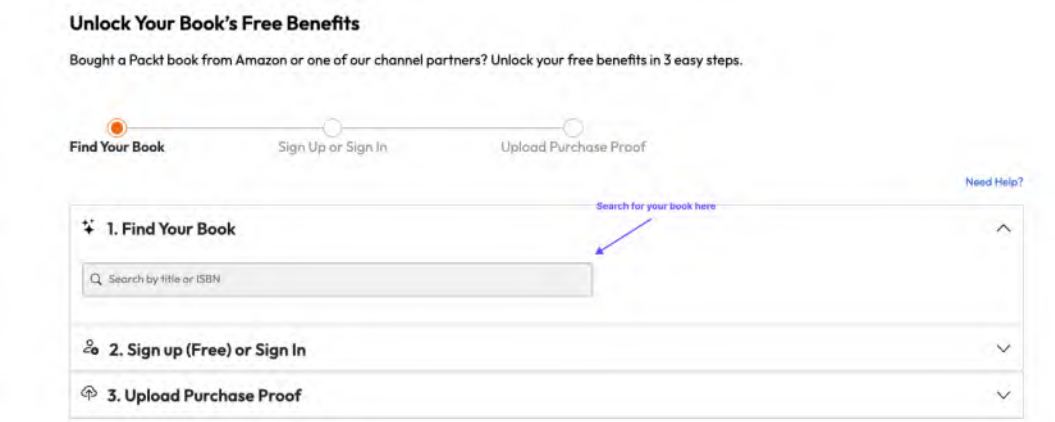


Figure 12.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

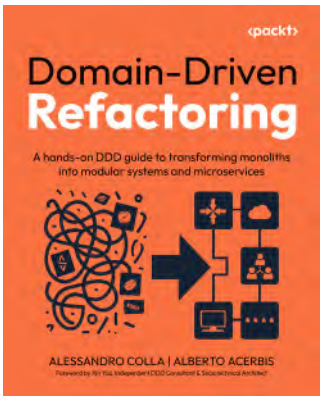
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

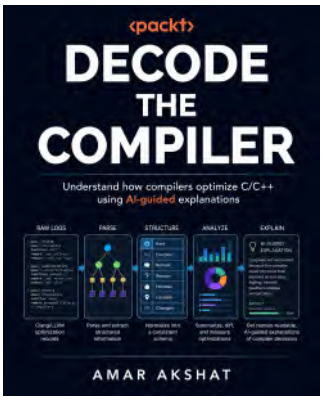


Domain-Driven Refactoring

Alessandro Colla, Alberto Acerbis

ISBN: 978-1-83588-910-7

- Find out how to recognize the boundaries of your system's components
- Apply strategic patterns such as bounded contexts and ubiquitous language
- Master tactical patterns for building aggregates and entities
- Discover principal refactoring patterns and learn how to implement them
- Identify pain points in a complex code base and address them
- Explore event-driven architectures for component decoupling
- Get skilled at writing tests that validate and maintain architectural integrity



Decode the Compiler

Amar Akshat

ISBN: 978-1-80602-233-5

- Read and interpret LLVM IR, CFGs, and optimization remarks
- Use explncc and LLMs to make sense of opaque optimization logs
- Compare Clang's output with GCC and MSVC for real-world workloads
- Apply AI guardrails to prevent hallucinations in diagnostics
- Trace vectorization, inlining, and loop unrolling decisions
- Craft reusable prompt libraries for performance insights
- Evaluate cost models for target-specific tuning
- Integrate human-in-the-loop workflows for safer AI tooling

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

Index

A

AI reasoning loop **45, 46**

aggregates
- refining 120 – 124

artificial intelligence (AI)
- architect's daily workflow, reshaping 15 – 17
- as cognitive amplifier 14

attention mechanism **33**

autonomy
- testing 266, 267

B

BMAD commands **150**

BMAD-like workflow **144**

BrewUp experiment **195**
- limits 195 – 198

BrewUp orchestration loop **241**
- accepted artifacts, staging 243
- building, around protocol 241, 242
- completing, after final human gate 249
- one stage, correcting 247
- replayable run trace, persisting 248
- review, pausing 244, 245
- specialist, running 244, 245
- stages, advancing through 246
- testing 250

BrewUp vocabulary **85 – 87**
- reinforcing 87
- testing 87

bounded-context rules **178**

byte pair encoding (BPE) **27**

C

CamelCase forms **28**

Classification prompt **141**

Command Query Responsibility Segregation (CQRS) **117**
- role separation 117 – 119

Command and Event Writer **206**
- convenient command 207
- results 208
- scenarios 206, 207

Context Mapper **209**
- boundaries, naming 211
- context map 212
- human gate 212
- responsibilities, mapping 210
- responsibility boundaries 209, 210

candidate events **73**

chain evaluation **213**
- handoff 214, 215
- manual gates to orchestration 219
- preservation 215, 216
- repository 218
- review gates 217
- specialization 216
- transformations 213, 214

checked task **175, 176**

clarification **169**

cognitive boundary **177, 195**
- constrains 177

cognitive capacity **10**

cognitive load **10**
- mitigation strategies, in DDD teams 13

cognitive overload **11**
- symptoms in DDD teams 12

commands
- refining 120 – 124

consistent domain vocabulary
- iterating 103 – 105

constraints
as conceptual boundaries 56 – 58
context snapshot **128**
contextual coherence
achieving, through iteration 124, 125
bounded context 132
from repeated comparison 125, 126
open questions 127
reinforced, by context snapshot 128 – 130
stable ownership 126, 127
stable roles 126, 127
testing, with final review prompt 130, 131
contracts **192**
controlled delegation **183**
conversations
AI reasoning loop 45, 46
architecting 44
as bounded process 47
as system 45
DDD parallel, recognizing 47
prompt paradigm limit 44
core events **73**
additive feedback 76
corrective feedback 76
feedback, as architectural input 75
identifying 72
knowledge crunching process 81
model contribution 74
refining 72, 73
semantic convergence 81, 82
structural feedback 76
correction attempts
limits 264 – 266

D

DDD guardrails
agents 98, 99
containers 97
MCP servers 101 – 103
skill 99 – 101
DDD review **171, 176**
DDD rulebook
language 95
output structure 96
semantic rules 95
Domain Storytelling **67, 68**
versus EventStorming 68
decision rights **254**
autonomy, turning into policy 255
shared alignment inputs, defining 256, 257
derive-review-update workflow **145 – 150**
domain carriers **135 – 139**
AI outputs, reading as reflections of understanding 151, 152
derive-review-update workflow 145 – 150
implementing 141 – 143
reusing, across workflows 144, 145
shared semantic alignment, building 153, 154
Stock Management domain carrier, building 139 – 141
domain knowledge **184**
domain-driven design (DDD) **3**
drift
controlled drift, running 261 – 263
correcting 261
correction scenario 261 – 263

E

EventStormer **200**
candidate facts 200, 201
invention 201
results 202
EventStorming **65 – 67**
versus Domain Storytelling 68
emergent reasoning **37**
blind spots 37
events
refining 120 – 124

F

feature request **162**
clarification 169

running, through role-based workflow 183
specification 167, 168

feedback practical refinement loop **77**
discovery material, classifying 77 – 79
domain and structural feedback, injecting 79, 80
revised set, validating 80

G

guards **179, 180**
implementing 181, 182
responsibilities 182

H

handoff schema **192, 193**

human gate **254**

I

idempotency requirement **181**

information flows, for AI
architect, as information designer 51
designing 48
information architecture 49
information, as probabilistic constraint 51
pattern 50, 51
sovereign context window 48
stateless model 48

K

knowledge crunching process **81**

knowledge shift **136, 137**

L

LLMs, as statistical pattern machines **20**
core mechanism 20, 21
mental model, for architects 25, 26
prediction 23, 24
tokens 21 – 23

language, in domain clarity **83**
false coherence 84
instability 83, 84
practical BrewUp 85 – 87
vocabulary

large language models (LLMs) **19**

M

mechanical alignment rules **258, 259**

mechanical violations
separating, from semantic divergence 257

model
guiding, through feedback loops 110 – 117

model drift **9**

modern business systems complexity **4**
architectural distribution 4
data volume 6
heterogeneous systems 6
impact on decision-making 8, 9
impact on domain modeling 8, 9
integration sprawl 6
real-time analytics demands 6, 7
regulatory and compliance burdens 5
sociotechnical complexity 7, 8
team scaling 7, 8

multi-agent conversations
acceptance 239
artifact envelope, defining 232
conversation state, encoding 234 – 236
designing 230
explicit protocol 230, 231
legal transitions 236 – 238
messages and routes, defining 233
protocol invariants, testing 240, 241
referral 239
STOPPED 240

multi-agent system

operating, as governed workflow 268

N

nucleus sampling 35

O

OpenAI Tokenizer 29
Order aggregate 121, 125
orchestrator, as facilitator 222
boundary, generalizing from working example 228, 229
bounded specialists, registering 224
eligibility check, implementing 224 – 226
human gate and resume, recording 227, 228
orchestration workspace, creating 223
transition, running 227
workflow state, recording 224
ownership rules 179

P

plan 173
policy-triggered command 116
predictive chain 33
product requirements document (PRD) 184
productive specialists 190
prompt rules
significance 92 – 94
sources 94, 95

R

Ralph Wiggum loop 82
rulebook 91
rules 165

S

Stock Management domain carrier
building 139 – 141
Storyteller 203
concrete scenarios 203
discipline of stopping 204, 205
results 205
self-attention mechanism 31
semantic convergence 81, 82
semantic divergence 257
surfacing 260, 261
shared alignment inputs
defining 256, 257
shared semantic alignment
building 153, 154
shared vocabulary
creating 52 – 55
specification 166
on feature request 167, 168
reviewing 170 – 172

T

Team Topologies 11
time limits 264
tokenization 26
coherent output 33, 34
context window 29 – 31
distant tokens, connecting 31 – 33
domain language 27 – 29
temperature parameter 34
token selection control 35
tool access
restrictions 264
top-p sampling 35
trace 163, 184
traceability 174
traditional domain modeling 10
limitations 10
transformations 190, 191
transformer architecture 31

U

Ubiquitous Language 38 – 40

V

visual chaos into structured context 69

reusable context 71, 72
structure extraction 70, 71
translation 69, 70