O'REILLY®



Early Release

RAW & UNEDITED

Architecture as Code

Specifying Architecture for Humans and Agents

Neal Ford, Mark Richards & Jonathan Johnson

Architecture as Code

Specifying Architecture for Humans and Agents

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Neal Ford, Mark Richards, and Jonathan Johnson



Architecture as Code

by Neal Ford, Mark Richards, and Jonathan Johnson

Copyright © 2026 Mark Richards and Neal Ford. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan

Development Editor: Sarah Grey

Production Editor: Christopher Faucher

Cover Illustrator: Susan Brown

Interior Designer: David Futato

February 2027: First Edition

Revision History for the Early Release

- 2025-12-12: First Release
- 2026-09-04: Second Release

See https://oreilly.com/catalog/errata.csp?isbn=9798341640382 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Architecture as Code*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-64032-0

Brief Table of Contents (*Not Yet Final*)

Preface (unavailable)

Chapter 1: The Intersections of Architecture (available)

Part I: Architecture and AI (unavailable)

Chapter 2: Architecture and Generative AI (unavailable)

Chapter 3: Applying Architecture as Code to Generative AI (unavailable)

Part II: Architecture and Implementation (available)

Chapter 4: Architecture and Source Code (available)

Chapter 5: Describing Architecture Using ADL (available)

Chapter 6: Governing Architecture Using ADL and AI Agents (available)

Chapter 7: Why This Intersection Is Important (available)

Part III: Architecture and Data Topologies (available)

Chapter 8: Data Topologies (available)

Chapter 9: Describing Data Topologies Using ADL (available)

Chapter 10: Governing Data Topologies Using ADL and AI Agents (available)

Chapter 11: Validating Data Topologies Using AI Agents and Fitness Functions (available)

Chapter 12: Why This Intersection Is Important (available)

Part IV: Architecture and Engineering Practices (unavailable)

Chapter 13: Architecture and Engineering Practices (unavailable)

Chapter 14: Architecture and Version Control (unavailable)

Chapter 15: Architecture and Code-Level Metrics (unavailable)

Chapter 16: Applying Architecture as Code to Engineering Practices (unavailable)

Chapter 17: Why This Intersection Is Important (unavailable)

Part V: Architecture and Team Topologies (unavailable)

Chapter 18: Architecture and Team Topologies (unavailable)

Chapter 19: Applying Architecture as Code to Team Topologies (unavailable)

Chapter 20: Why This Intersection Is Important (unavailable)

Part VI: Architecture and Infrastructure (unavailable)

Chapter 21: Architecture and Infrastructure (unavailable)

Chapter 22: Applying Architecture as Code to Infrastructure (unavailable)

Chapter 23: Why This Intersection Is Important (unavailable)

Part VII: Architecture and Systems Integration (unavailable)

Chapter 24: Architecture and Systems Integration (unavailable)

Chapter 25: Architecture Quantum Defined (unavailable)

Chapter 26: The Scope of Agentic Code (Re)generation (unavailable)

Chapter 27: Communication and Dynamic Coupling (unavailable)

Chapter 28: Normalized Distance from the Quantum Main Sequence (unavailable)

Chapter 29: Sagas and Atomic Coupling (unavailable)

Chapter 30: Dependencies and Static Coupling (unavailable)

Chapter 31: Why This Intersection Is Important (unavailable)

Part VIII: Architecture and the Enterprise (unavailable)

Chapter 32: Architecture and the Enterprise (unavailable)

Chapter 33: Applying Architecture as Code to the Enterprise (unavailable)
Chapter 34: Why This Intersection Is Important (unavailable)
Part IX: Architecture and the Business Environment (unavailable)
Chapter 35: Aligning with the Business Environment (unavailable)
Chapter 36: Managing Business Change (unavailable)
Chapter 37: Domain to Architecture Isomorphism (unavailable)
Chapter 38: Applying Architecture as Code to Business (unavailable)
Chapter 39: Why This Intersection Is Important (unavailable)
Part X: Summary (unavailable)
Chapter 40: Lessons and Principles (unavailable)
Chapter 41: Putting It All Together (unavailable)
Chapter 42: Build Your Own Sysops Squad Sandbox (unavailable)

Chapter 1. The Intersections of Architecture

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

As inconvenient as it is, software architecture is highly synergistic with all the other parts of the software development process and ecosystem. In other words, as teams change different facets of the development process, it influences software architecture… and vice versa.

Software architecture doesn't exist in a silo–it is highly interconnected with all the other parts of building software. However, software architects have a unique perspective. We understand the system's overall structure, the development team's capabilities, and the details of infrastructure, data, and lots of other things. We often make decisions in software architecture that affect how software is built, but one constant problem that software architects face is how to make sure that what we design is what teams actually implement. For example, an architect may design a loosely coupled microservices architecture, but the development team could end up tying all the services together with a single coupling point, like a database. While the architect *designed* a microservices architecture, the team *built* a distributed

ball of mud. This is a problem of feedback: we can't tell if the implementation meets the design.

Multiply this problem by all the intersections of architecture: The architect needs cloud-based scalability but the infrastructure can't manage it yet. The architect creates a great design but the organization doesn't have the proper team members to implement it. We could go on and on.

In fact, we've identified nine different areas where software architecture intersects with other key concerns. We call these the *intersections of architecture*:

Implementation

> Is the implementation aligned with architectural concerns surrounding operational characteristics, architectural constraints, and the internal structure of the architecture?

Infrastructure

> Do the infrastructure and the way the architecture is deployed align with the architecture's operational concerns, such as scalability, responsiveness, fault tolerance, and availability?

Data topologies

> One widely ignored intersection is that of architecture with data topologies and data types. The system's data topology (such as monolithic, domain databases, and database-per-service) must align with its architectural style to work.

Engineering practices

> Does the way the development team creates, maintains, and tests software correspond with the architecture? Does the deployment pipeline match the architectural style?

Team topologies

The way teams are organized can significantly impact the architecture and vice versa. If the team structure is not properly aligned with the architecture, development teams struggle, finding even the simplest of changes challenging.

Systems integration

With what other systems or services does the architecture need to communicate? Not paying to this particular intersection can have devastating results for maintenance, reliability, and operational characteristics such as scalability, responsiveness, and availability.

The enterprise

Is the architecture aligned with the frameworks, practices, guiding principles, and standards of the organization and the enterprise?

The business environment

Is the architecture properly aligned with the business environment and the problem domain? Too often, architects ignore this important intersection and design architectures that fail to meet the goals or needs of the business.

Generative AI

How does the increasing use of LLMs impact the architecture? This intersection is quickly becoming important as more companies leverage generative AI within their systems.

The purpose of this book is to describe each of these intersections in a way that provides architects with actionable feedback. In other words, rather than describe these intersections as text, we're going to describe them as *executable code*– that is, *Architecture as Code*.

Core Concepts

How do we describe these intersections in architecture in an executable way? We start with architectural fitness functions, a concept originally defined in the book *Building Evolutionary Architectures* by Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage (O'Reilly, 2022).

The Trade-Off Analysis Tetralogy

Many of our readers are familiar with our previous books, including *Fundamentals of Software Architecture* (now in its 2nd edition), *Software Architecture: The Hard Parts*, and *Software Architecture Patterns, Antipatterns, and Pitfalls*, and for that we thank you. These books all focus on trade-off analysis as a core activity of software architects. Throughout them, we focus on a particular type of analysis that's particularly useful when architects don't have the luxury of building systems before making important decisions: *qualitative analysis*.

From Qualitative to Quantitative Analysis

How can an architect make decisions about their software architecture before they've had a chance to build it? It's important to choose an appropriate architecture style fairly early in the development process, because the style defines the structural scaffolding of the system and dictates the direction of many decisions. For example, in a highly distributed architecture, the team should choose low coupling.

Suppose an architect is tasked with designing a new system. They know that they will need a distributed architecture because of scale. They've narrowed the candidate architecture styles down to eight possibilities. Which should they choose?

One option is to say: "Boss, I have good news and bad news. The good news is that I will implement the objectively best possible workflow for the new system–because we're going to build all eight possible solutions, measure the results, and throw the seven deficient solutions away. The bad news? It will take eight times longer and cost eight times more." This

architect would be measuring working software directly, or performing *quantitative analysis*. But most architects don't live in that world and must rely on the less accurate but more pragmatic option of *qualitative analysis*.

A popular and distinguishing feature of our book _Fundamentals of Software Architecture _ is that we use comparison charts we use to evaluate how each architectural style affects important architectural characteristics. An example of this kind of analysis for the microservices architectural style appears in Figure 1-1.

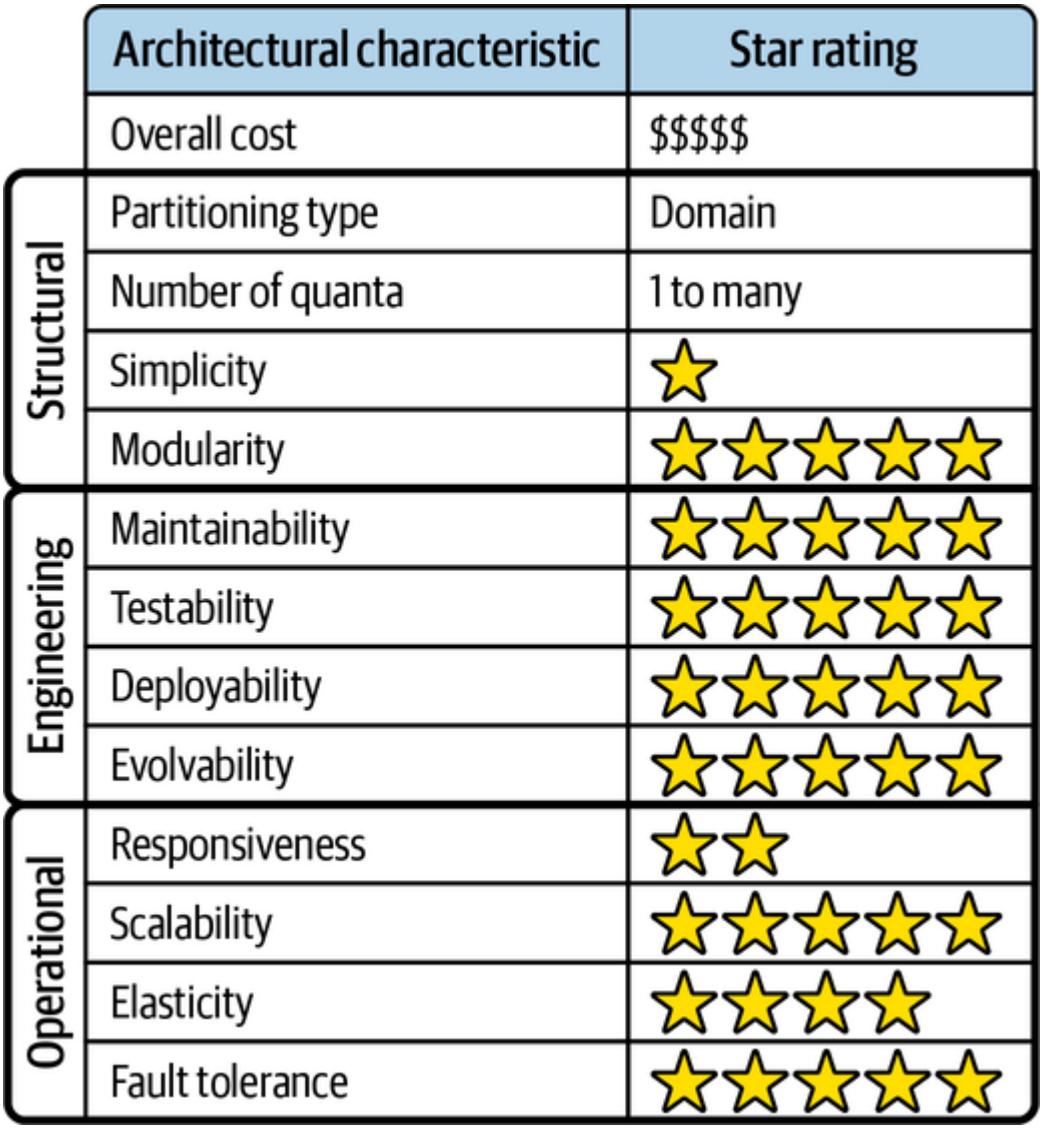


Figure 1-1. The architectural characteristics trade-off charts for microservices from Fundamentals of Software Architecture

In Figure 1-1 we evaluate each typical architectural characteristic for the given architecture, starting with structural elements, on a scale of one to five stars.

We developed the idea of qualitative analysis during our original workshops for new architects. We wanted to be able to compare architecture styles by what really distinguishes them. From the split perspective of *behavior* and *capabilities*, generic architecture styles are generic for a reason—developers can implement virtually any *behavior* in any generic style. What really differentiates styles is how they handle different *architectural characteristics*. Thus, we began comparing styles by their architectural characteristics.

Originally, we had a much cruder scale, using just a thumbs-up and thumbs-down icons for whether the style supported the architectural characteristics or not, as illustrated in Figure 1-2. However, we quickly realized that a binary scale lacked nuance. Some architectural styles support certain capabilities better than others. We briefly added a "sideways thumb" to act as a neutral indicator, but still felt that the scale wasn't broad enough for useful comparison.

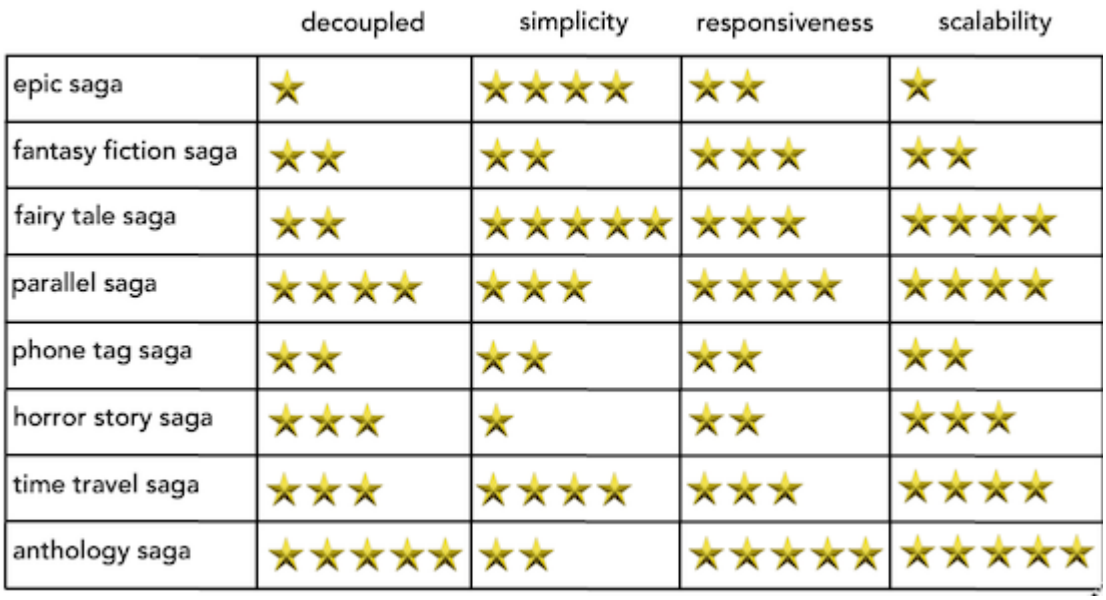


Figure 1-2. The initial pass at qualitative analysis used a binary scale

That's when we settled on the five-star scale we've used since. Five stars provides a 20% difference between each value, which is fine-grained enough to draw useful distinctions but not so fine-grained as to indicate artificial precision. For example, having a 100-point scale isn't useful, because that level of precision isn't really possible when making a judgment call. Finding the correct scale granularity is key.

So how did we come up with the values for the star ratings? We first had to decide what the comparison criteria should be. For architectural styles, we chose the architectural characteristics that are most affected when changing from one style to another. In *Software Architecture: The Hard Parts*, we

built our qualitative analysis by comparing different trade-offs given different combinations of forces, as shown in Figure 1-3.



	decoupled	simplicity	responsiveness	scalability
epic saga	★	★★★★	★★	★
fantasy fiction saga	★★	★★	★★★	★★
fairy tale saga	★★	★★★★★	★★★	★★★★
parallel saga	★★★★	★★★	★★★★	★★★★
phone tag saga	★★	★★	★★	★★
horror story saga	★★★	★	★★	★★★
time travel saga	★★★	★★★★	★★★	★★★★
anthology saga	★★★★★	★★	★★★★★	★★★★★

Figure 1-3. Comparing distributed architecture transactional workflows qualitatively

Figure 1-3 compares the named sagas on the left based on each one's level of coupling (*decoupled*), how simple it is for developers to understand and implement (*simplicity*), its level of perceived performance (*responsiveness*), and how well it handles a large number of concurrent users (*scalability*).

In each case, we built the star ratings qualitatively, based on our inherent knowledge of what that style does and does not support well. For example, in Figure 1-3, we scored scalability for the Fairy Tale Saga at four stars, and for the Epic Saga at one star.

We didn't build all possible solutions to determine this rating. We assessed what characteristics enable scalability in distributed architectures. The Fairy Tale Saga uses asynchronous communication, eventual consistency, and orchestration for coordination. That means it features two of the three settings that allow for maximum scalability: asynchronous communication, eventual consistency, and choreography. Alternatively, the Epic Saga uses synchronous communication, atomic consistency, and orchestration, all of which decrease scalability: synchronous communication ties up requests

with wait states, atomic consistency requires services to coordinate, and orchestration creates bottlenecks.

Thus, even though we haven't implemented each possible solution, understanding what desirable (or undesirable) characteristics it facilitates allows us to perform an approximate analysis.

One of the unique features of this book is its use of qualitative analysis for each architecture pattern. Most books treat patterns and styles in isolation, but architects must often choose between competing solutions. Evaluating these comparison criteria can help them more readily make intelligent, informed decisions. We encourage architects to build such analyses themselves, and will illustrate the effectiveness of this technique throughout the book.

Quantitative Analysis

Architects prefer to rely on *quantitative* analysis—measuring working software. This book illustrates exactly how to react to important changes in the ecosystem by applying architectural feedback opportunities as code.

We will describe ADL as we go, but it is meant to be an extremely lightweight language that describes architecture concerns succinctly. In fact, we don't insist that architects use our descriptive language: if they supervise only a Java codebase, then keeping the fitness functions in Java makes sense. However, nondevelopers can't readily read Java, and it becomes a maintenance headache for architects who work in multiple platforms or technology stacks.

The Purpose of Architecture as Code

Why go through all this trouble and add more overhead to projects? Like any technique, architecture as code should create value that outweighs the effort required to implement it.

A consistent communication medium between intersections

ADL describes each intersection in a consistent format, allowing other parts of the software development ecosystem to consume that description. For example, when architects make important decisions such as the parameters of scalability (how many expected users and the average response time per use), they generally document them in Architecture Decision Records (ADRs). However, the infrastructure team members don't care about the rationale and background of every decision—they just want the facts. ADL provides that, expressing architectural concerns as declaratively as possible.

A *feedback* framework, not a testing framework

While many aspects of Architecture as Code suggest a testing framework, it's not. It is a *feedback* framework. The distinction is important: when a test fails, it indicates an error. When an ADL constraint fails, it is more often a placeholder for a conversation. For example, say an architect constrains a system to a specific set of components, but a developer creates a new component that triggers the structural fitness function. This isn't an error, but a feedback opportunity. Perhaps the architect missed a needed component and agrees that the system needs it, but also wants to reanalyze coupling and other component aspects in light of the updated design.

Protection for the inevitable negative trade-offs

Software architects constantly face decisions that they know have negative trade-offs. After all, the First Law of Software Architecture (defined in *Fundamentals ofSoftware Architecture*)book is that "everything in software architecture is a trade-off." Important decisions almost always come with some negative aspects, which were outweighed by the positive.

Architecture as Code allows architects to not only recognize negative trade-offs but build code that *governs* against those negative trade-offs manifesting. For example, in [Link to Come], we discuss decisions such as whether to use feature branching or feature toggles, the negative trade-offs

associated with each, and how to use code to build feedback loops that raise alarms when something breaks.

How much Architecture as Code do I need?

One important difference between unit and other domain testing and Architecture as Code is their overall coverage goals. A common goal in many software projects is comprehensive test coverage, usually via a combination of testing frameworks and approaches—unit tests, functional tests, user acceptance tests, and so on). However, the goal of architecture as code is less comprehensive and more focused. Let's look at some parallels between domain testing (before and after the advent of Agile engineering) and architectural characteristics verification.

Figure 1-4 illustrates the general state of domain testing before unit and other forms of testing became popular. It was common practice for teams to do ad-hoc testing for possible regressions, mostly based on their experience with the most fragile parts of the codebase.

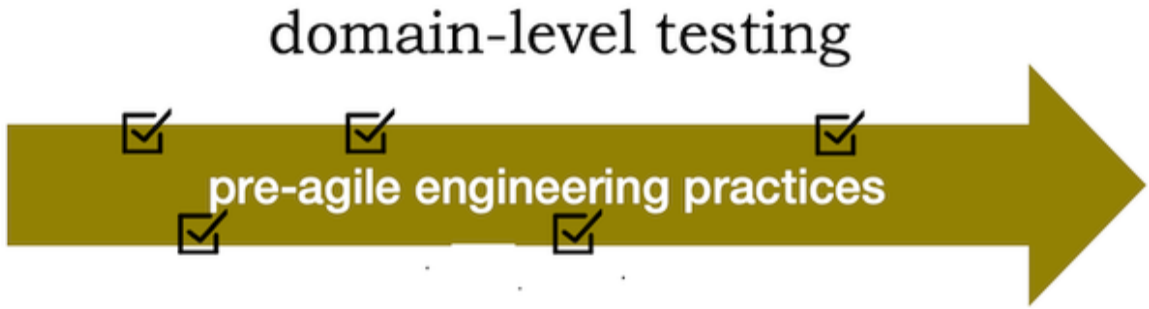


Figure 1-4. Domain-level testing before engineering practices embraced comprehensive testing

After Agile engineering practices were introduced, new tools allowed teams to build automated and comprehensive testing, providing fast feedback for any breaking change, as illustrated in Figure 1-5. Here, testing covers every aspect of the codebase, alerting developers as quickly as possible when something changes and causes a fault. This safety net is a great enabler of aggressive refactoring exercises. For example, a developer can make a sweeping change, then run a suite of unit tests to see how many break, which loosely correlates to the scope of the change. If the scope is too big

to tackle right now, they can easily roll back the changes and add the improvement to a backlog.

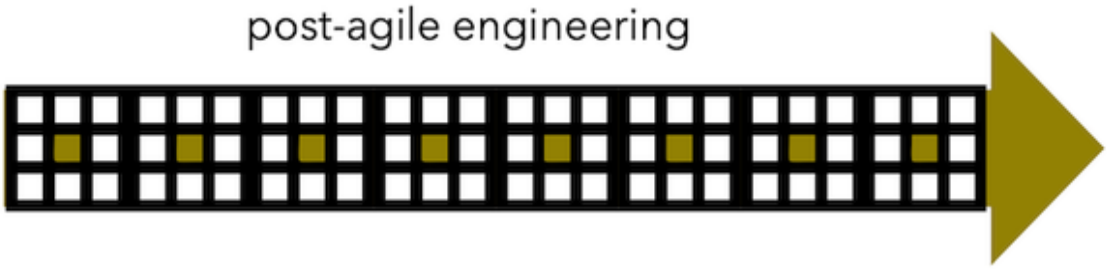


Figure 1-5. Software development over time with comprehensive testing, creating fast feedback for breakages.

Currently, teams on most software projects, even those with a robust testing practice around their domain, do only ad-hoc and spotty verification of architectural characteristics. For example, perhaps the team has a metrics tool wired into the build for code quality checks and a few operational tests, but nothing that runs regularly, as illustrated in Figure 1-6. This way of testing architectural characteristics is opportunistic and not comprehensive.



Figure 1-6. The typical coverage of architectural characteristics verification on most teams before Architecture as Code

The goal of *fitness-function-driven architecture* (FFDA) is *not* to create a comprehensive net around every architectural concern—teams could digressively find reasons to build fitness functions for a long time. In other works, the goal of FFDA is not the version of [Link to Come] shown in Figure 1-7, where the architects have tried to build a governance net around every aspect of the architecture. This is both time-consuming and impractical.

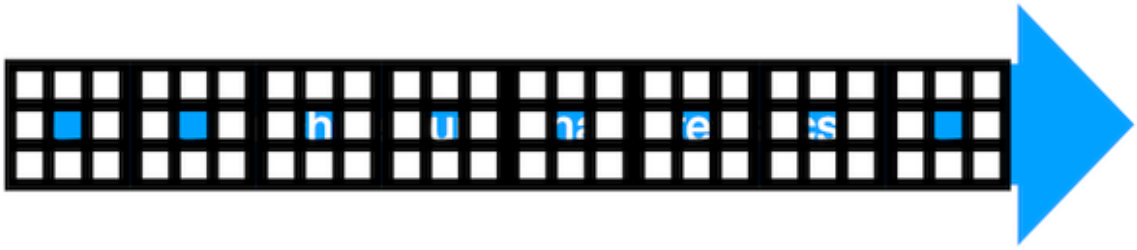


Figure 1-7. Architecture as Code doesn't try to be as comprehensive as domain testing



Figure 1-8. A skyscraper under development.

Instead, we can go back to our building metaphor:; consider the building under construction illustrated in Figure 1-8. The building's architect defines its important structural elements, such as load-bearing walls and number of floors, and the construction crew consults those definitions. The architect doesn't define every aspect of the building, as illustrated in Figure 1-9. They don't try to ensure that every single wall and join point in the building is fixed—only that the load-bearing walls remain in place. Building architects find the *critical breaking points* in their designs and use sophisticated mathematics to analyze them.

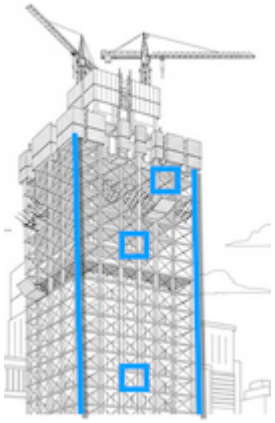


Figure 1-9. A building architect is concerned with the critical parts of the structure.

Similarly, software architects don't try to encompass *every* possibility for governance in a system. Instead, they look for elements of the system where fast feedback would provide a way to address changes.

Architecture versus design

Consider the skyscraper in Figure 1-8. How many floors does it have? Is the roof flat or peaked? Is it a multistory apartment building or offices? All of these things define its overall *structure*--in other words, its architecture. Now take a moment to picture the inside of the building. Do the floors have carpeting or hardwood? What color are the walls of the lobby? Are there lights hanging from the ceiling? All of these things relate to the *design* of the building.

Software architecture is less about a system's appearance and more about its structure, whereas design is more about a system's appearance and less about its structure. For example, the choice to use microservices defines the structure and shape of a system (its architecture), whereas the look and feel of the user interface (UI) screen define its design. But what about decisions such as whether to break apart a service into smaller parts or deciding on a UI framework?

Unfortunately, most decisions like these fall somewhere on a *spectrum* between architecture and design, making it difficult to determine what should be considered architecture. The following criteria can help

determine whether something is more about architecture or more about design:

- Is it more strategic in nature or more tactical?
- How much effort will it take to change or construct?
- How significant are the trade-offs?

These factors are shown in Figure 1-10, illustrating the spectrum between architecture and design to help determine where a decision lies and who should have responsibility for it.

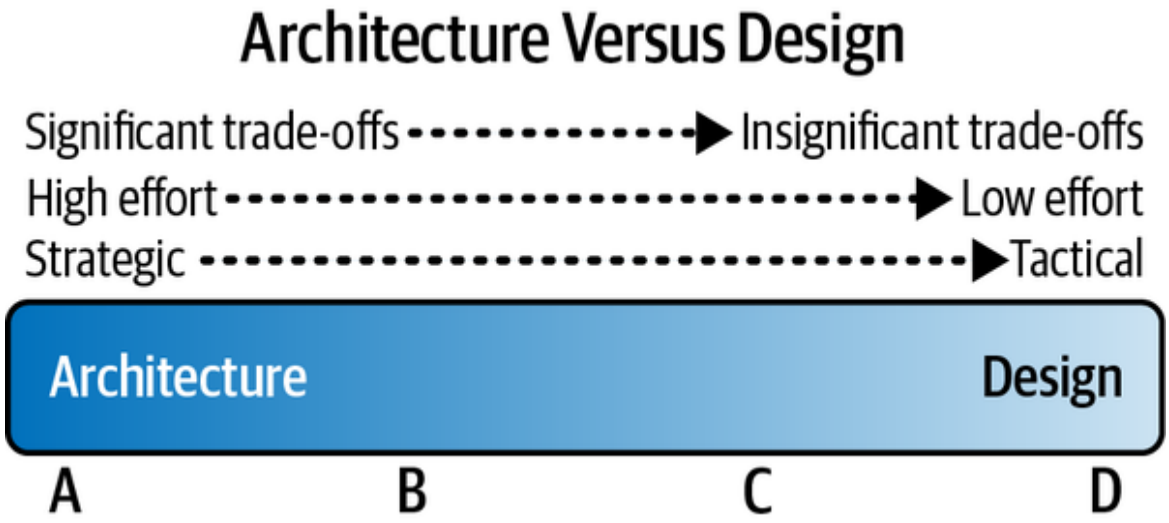


Figure 1-10. The spectrum between architecture and design

Strategic versus tactical decisions

The more strategic a decision, the more architectural it becomes. Conversely, the more tactical a decision, the more it's likely to be about design. *Strategic* decisions are generally long-term, whereas *tactical* decisions are generally short-term and usually independent of other actions or decisions.

One good way to determine whether a decision is more strategic or tactical is to consider the following questions:

How much thought and planning are involved in the decision?

A decision that takes a couple of minutes is more likely to be tactical and hence more about design, whereas a decision requiring weeks of planning is likely to be more strategic and hence more about architecture.

How many people are involved in the decision?

A decision made alone or with a colleague is likely more tactical and on the design side of the spectrum, whereas a decision requiring lots of meetings with many different stakeholders is likely more strategic and on the architectural side of the spectrum.

Is the decision a long-term vision or a short-term action?

A decision that is likely to change soon is usually tactical in nature and thus more about design, whereas one that will last a very long time is typically more strategic and more about architecture.

While these questions are a bit subjective, they nevertheless help determine whether something is strategic or tactical, and thus whether it's more about architecture or design.

Level of effort

In his famous paper "Who Needs an Architect?", software architect Martin Fowler writes that architecture is "the stuff that's hard to change." The harder something is to change, generally, the more effort it requires, placing that decision or activity toward the architectural side of the spectrum. Conversely, something that requires minimal effort to implement or change would fall more on the design side of the spectrum.

For example, moving from a monolithic layered architecture to microservices would require significant effort, so it would be more about architecture. Rearranging fields on a screen would require minimal effort, and would therefore be more about design.

The significance of trade-offs

Analyzing the trade-offs of a particular decision can help a lot in determining whether it is more about architecture or design. The more significant the trade-offs are, the more architectural the decision tends to be. For example, choosing to use the microservices architecture style provides better scalability, agility, elasticity, and fault tolerance. However, this architecture is highly complex and very expensive, has poor data consistency, and doesn't perform well due to service coupling. These are some pretty significant trade-offs. We can conclude that this decision is more on the architectural side of the spectrum than design.

Design decisions have trade-offs, too. For example, breaking apart a class file provides better maintainability and readability—at the cost of managing more classes. However, these trade-offs are not overly significant.

The distinction between architecture and design is important for architecture as code because building and maintaining fitness functions requires effort, and architects want to ensure that their efforts aren't frivolous. In other words, the fitness function's payoff should justify the effort to build and maintain it. Architects don't need crystal balls: they should decide on the value of trade-offs and then reevaluate those decisions on a regular basis.

> **_CAN_ VERSUS _SHOULD_**
>
> Creating a fitness function isn't an exercise of "*Can* I do it?" but "*Should* I do it?"

Logical vs. Physical Architecture

Software architecture represents both the *logical architecture* and *physical architecture* of a software system or product. The logical architecture of a software system identifies relationships between its components and how the source code is structured, whereas the physical architecture represents

the deployed artifacts of the system, such as services, gateways, message brokers, and so on.

The diagram in Figure 1-11 represents an example of a logical architecture. Notice how it only shows components and their interactions with each other and with users, but does not indicate artifacts like user interfaces, services, databases, and so on.

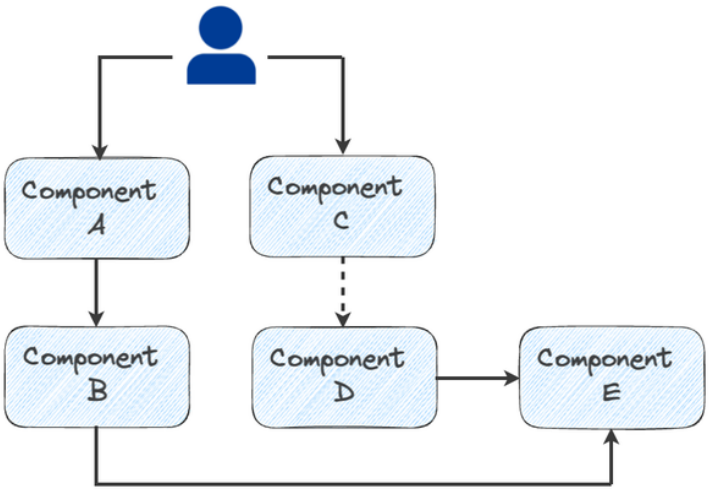


Figure 1-11. A logical architecture contains components and their interactions

A logical architecture is largely independent of the physical architecture of system. For instance, the diagram shown in Figure 1-11 could be implemented as a monolithic architecture, or a highly distributed microservices architecture.

A physical architecture, on the other hand, does specify the physical architectural style (such as microservices) and includes physical artifacts as showin in the diagram in Figure 1-12

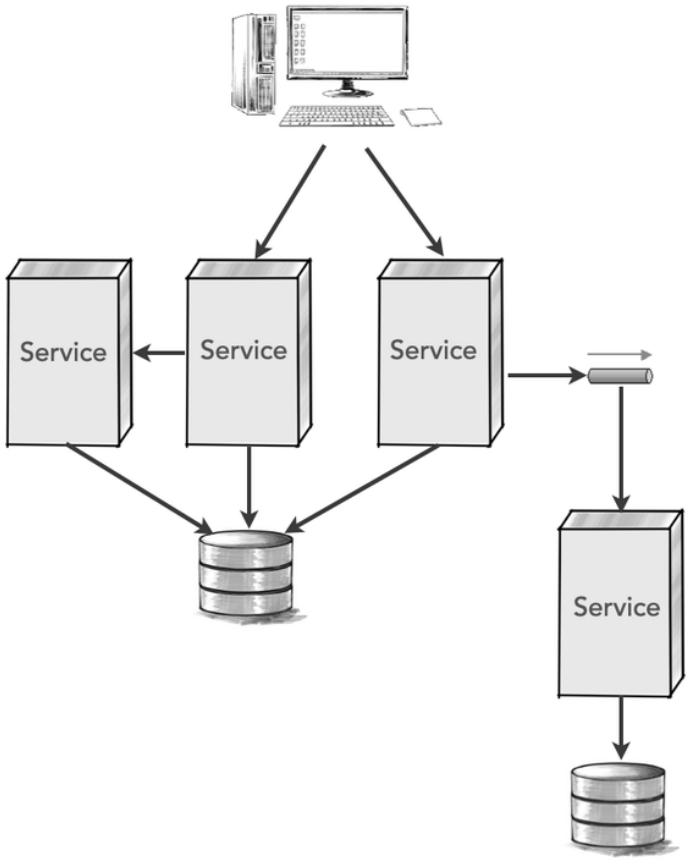


Figure 1-12. A physical architecture contains artifacts such as services, user interfaces, queues, and databases

This distinction is important as we'll referring to both of these architectural representations throughout the book as we demonstrate how to apply architecture as code.

Architecture Definition Language

Use Case: The Sysops Squad

In our book *Software Architecture: The Hard Parts* (O'Reilly, 2021), we used an *architectural kata*—a case study—called the *Sysops Squad*, to concretely illustrate the concepts in that book. We'll use the same architectural kata throughout the book to show concrete examples of the concepts in this book about architecture as code.

The Sysops Squad is a complex customer-based trouble-ticket system for Penultimate Electronics, Inc, a fictional electronics retailer. Here's how their system works: When customers purchase electronic equipment such as computers, home entertainment systems, and stereo systems from the company's stores throughout the country, they can also choose to purchase a support plan. Then, if they run into issues with installation or the equipment, they can create a trouble ticket on the website and an assigned field expert will come to their home or office to fix the problem.

The Sysops Squad ticketing system's behavior includes reporting, ticket maintenance, ticket scheduling, customer sign-in, customer registration, billing, expert knowledge-base search, and maintenance. To limit the scope for this chapter, for now we'll focus just on the core ticketing process (including sending out customer surveys) and customer functionality. The following actions illustrate the overall major workflow of this part of the Sysops Squad system:

1. The customer registers with the Sysops Squad system based on the support contract they purchased
2. The customer can change their profile, including address and contact information
3. The customer creates a trouble ticket using the public-facing Sysops Squad website
4. The system locates a nearby expert and assigns the ticket (this might take up to a couple of days)
5. Once an expert is available and assigned, the system notifies the customer that the expert is on their way
6. The system uploads the ticket to the expert's mobile device
7. The expert fixes the problem and marks the ticket as complete
8. The customer receives a survey link via email

9. The customer fills out and submits the survey using the public-facing Sysops Squad website

This workflow is illustrated in Figure 1-13.

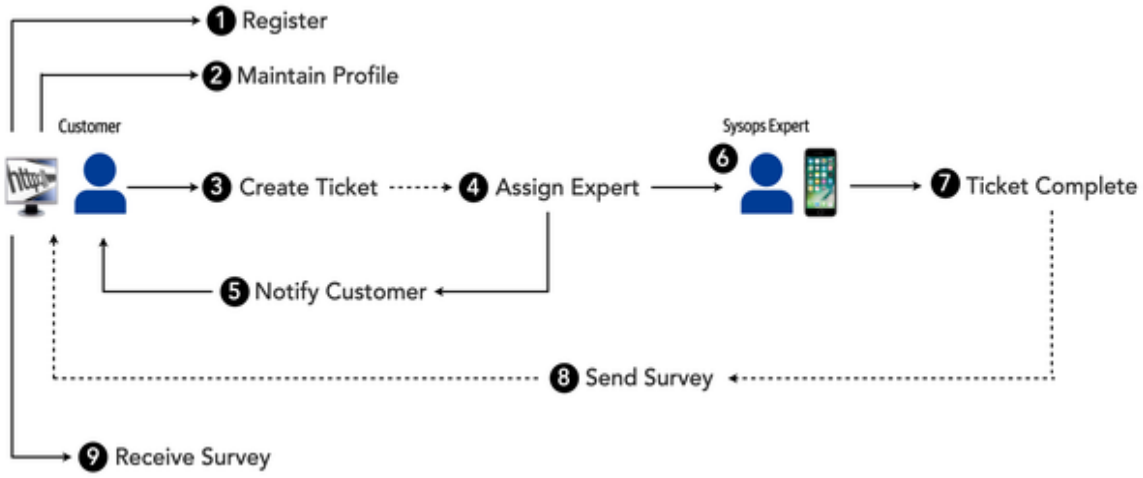


Figure 1-13. The primary ticketing workflow for the Sysops Squad system

The corresponding logical architecture for the Sysops Squad system based on this workflow is illustrated in Figure 1-14. Here, solid lines represent synchronous communication and dotted lines represent asynchronous communication (we use this throughout the book). The logical architecture consists of two domains: `Ticketing` and `Survey`. The `Ticketing` domain contains two components, `Ticket Creation` and `Ticket Completion`, as well as an `Assignment` subdomain with three components: `Ticket Assignment`, `Customer Notification`, and `Ticket Routing`. The `Survey` domain contains three components: `Send Survey`, `Receive Survey`, and `Survey Templates`. Customers interact with the `Survey` domain and the `Ticket Creation` component, and the field experts interact with the `Ticket Routing` and `Ticket Completion` components.

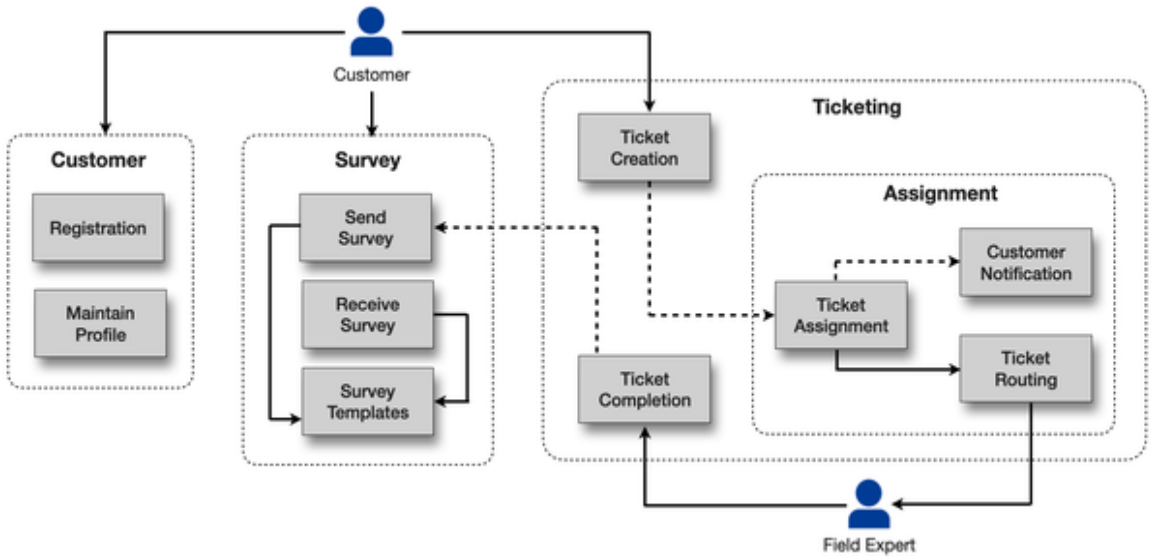


Figure 1-14. The logical architecture for the Sysops Squad system

Notice how the logical architecture defines three domains: `Customer`, `Survey`, and `Ticketing`, with `Ticketing` containing an `Assignment` subdomain. Each of the domains and subdomains contain components—the building blocks of the system. The logical diagram in Figure 1-14 also shows both the synchronous dependencies—those shown as solid lines—and asynchronous dependencies—those shown as dotted lines. chapters Chapter 2 and Chapter 3 discuss these components and their dependencies in more detail.

We'll be using the Sysops Squad architectural kata throughout the book as a way of providing concrete examples as we explain the concepts of architecture-as-code in each chapter.

How This Book Is Structured

The parts of this book correspond to the nine intersections of software architecture. Each part contains multiple chapters. We describe the intersection in question and show how to implement it in ADL and, in many cases, with concrete fitness functions as well.

We also tie the concept of intersections to traditional enterprise architecture (one of our intersections) to build an ecosystem that leverages modern innovations such as generative AI and agentic workflows.

Part I. Architecture and AI

Part II. Architecture and Implementation

Architects spend large amounts of time and effort creating an architecture for a system. They draw lots of lines and boxes (a required skill for an architect), defining the domains, subdomains, and components that make up the architecture. They also define how those components interact with each other and with users of the system. Architects usually represent the resulting architecture through *architecture diagrams*—drawings that are meant to convey a shared understanding of the architecture. All too often, however, development teams use these diagrams only to gain an understanding of the vision of the architecture, and as such the resulting implementation of the architecture rarely matches that depicted by the architecture diagrams.

Architecture as code addresses this problem by specifying the structure of the system through Architecture Definition Language (ADL) and corresponding executable source code. Not only does the ADL describe the architecture, but architects can govern the architecture through executable code to ensure that the implementation is always aligned with it.

We've broken down this part of the book into four chapters:

Chapter 2: Relationship Between Architecture and Source Code

> In this chapter we explain how to "see" the architecture by analyzing the structure of the source code. We also define how to recognize domains, subdomains, and components within the source code structure, and what defines an architectural component.

Chapter 3: Describing Architecture Using ADL

In this second chapter of this part, we explain how to describe a system's logical and physical structure using Architecture Definition Language (ADL). Here, we show how to describe both monolithic and distributed architectures, focusing primarily on defining components and services.

Chapter 4: Governing Architecture Using ADL and AI Agents

One aspect of governing an architecture is ensuring that the system's implementation matches the structure of the architecture. In this chapter, we show how to leverage ADL, AI agents, and external empirical fitness functions to ensure the implementation of the system matches the intended architecture. We also explain how ADL can be used to improve collaboration and communication between architects and development teams, and how to leverage architecture as code when iterating on the architecture.

Chapter 10: Why This Intersection is Important

This final chapter focuses on *why* the intersection of architecture and implementation is so important. Here, we dive deep into how implementation can get misaligned with the architecture, and why it's so critical for them to be aligned.

Throughput this part of the book we'll be leveraging the *Sysops Squad* architectural kata described in "Use Case: The Sysops Squad" as examples for how to define and govern the structure of a system using ADL and architecture as code.

Chapter 2. The Relationship Between Architecture and Source Code

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

Representing a system's architecture through its source code's directory or namespace structure is a critical part of how architects govern systems to align the architecture with the corresponding implementation. A system's *implementation*--its source code—-is usually kept in a version-control system (such as a Git repository) and organized by directories or namespaces, or sometimes both, depending on the programming language. Architects can recognize the architecture of a system by analyzing its directory and namespace structure, then use ADL to ensure that the implementation matches the intended architecture.

Part II of this book, Architecture and Implementation, focuses identifying an architecture's components, subdomains, domains, and services by associating these artifacts with directories or namespace nodes.

Identifying Components

Logical components represent the major functions a system performs. Figure 2-1 shows an example: a logical architecture diagram of the ticketing and survey parts of the Sysops Squad system. Each component in this diagram performs a specific function, such as creating a ticket, assigning an expert, or notifying a customer.

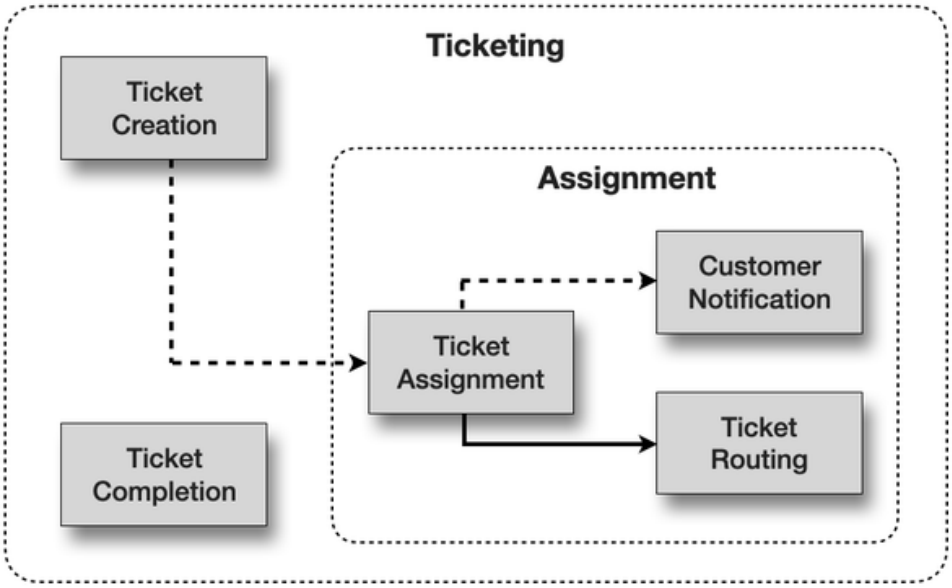


Figure 2-1. Components represent a system's major functions

Each component has a specific role and responsibility. For example, the `Ticket Creation` component is responsible for creating trouble tickets, whereas the `Customer Notification` component is responsible for notifying customers when the Sysops Squad expert is on the way. Each component contains source code, usually classes, that implements its particular business function.

Some programming languages use a directory structure to organize the source code (such as Java). In systems that use these languages, components are represented through the *leaf nodes* of the directory structure, as shown in Figure 2-2.

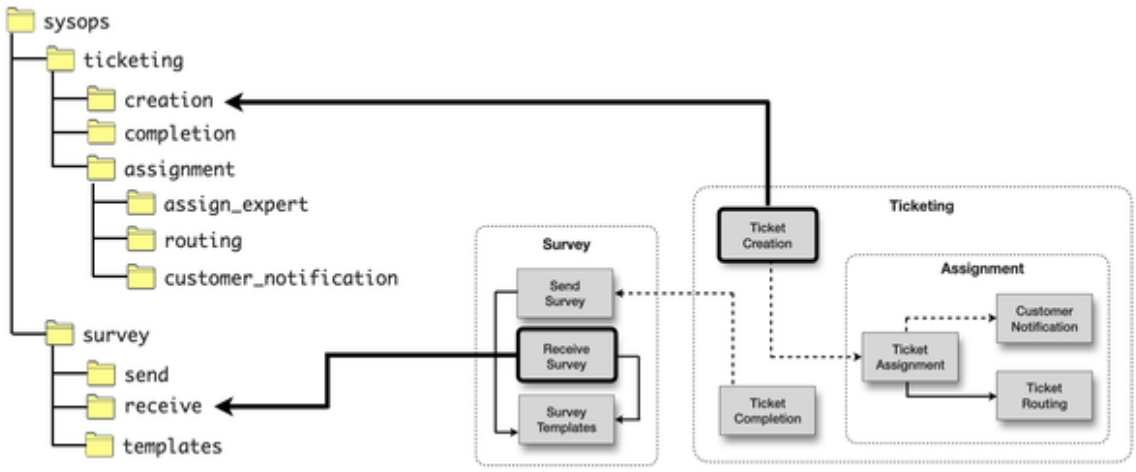


Figure 2-2. Logical components are represented through leaf-node directories or namespaces

Other programming languages, such as C#, primarily represent source-code structure through namespaces. In these cases, components are generally represented through the the leaf nodes of the namespace, as shown in the following example:

```
sysops.ticketing.assignment.notification
                                |_____________|
                                  component
```

These languages sometimes also use a directory structure, although it's not a requirement that the namespace match it.footnote[To learn more about architectural components, refer to Chapter 8 of *Fundamentals of Software Architecture*, 2nd Edition.]

Identifying Subdomains

Subdomains contain components and further subdomains. They are represented by lower-level directories in the repository, as illustrated in Figure 2-3.

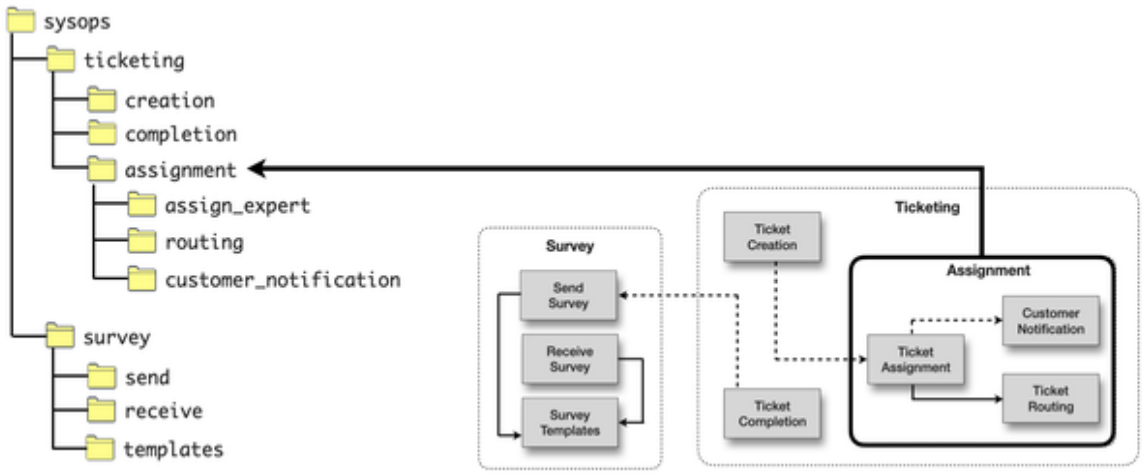


Figure 2-3. Subdomains are represented through lower-level directories or namespaces

In this diagram, `Assignment` is a subdomain with three components. It is represented by the `assignment` subdirectory in the repository.

```
sysops.ticketing.assignment.notification
                  |__________|
                   subdomain
```

When teams define multiple subdomains, the nodes between the highest-level node *after* the system node and the leaf node represent the subdomains. For example, say there are a number of different notifications that could be sent to a customer. If `notification` was a subdomain under `assignment`, the namespace would look as follows:

```
sysops.ticketing.assignment.notification.expert-assigned
                  |__________|_____________|
                   subdomain   subdomain
```

A system can have any number of subdomains, but ultimately the subdomain nodes or directories end with components as the leaf nodes.

Identifying Domains

Domains are the highest-level structure in an architecture and are represented as the highest level after the system directory, as illustrated in Figure 2-4.

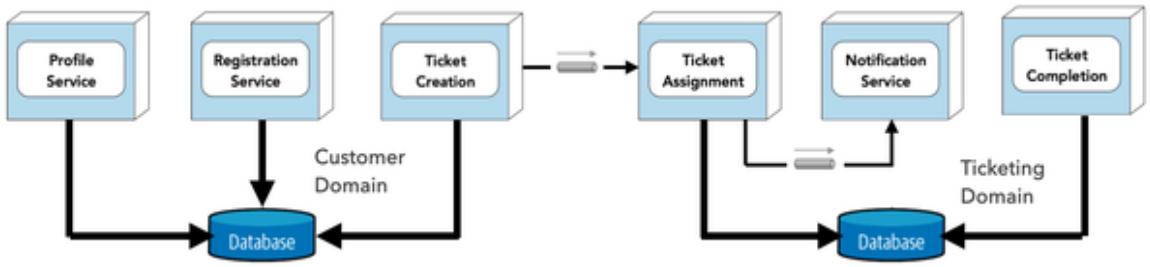


Figure 2-4. Domains are represented through the highest-level directories or namespaces after the system directory

Domains contain components. They may also contain subdomains (such as `assignment`) that contain further subdomains or components. If using namespaces, the domain is represented as the second node in the namespace:

```
sysops.ticketing.assignment.notification
      |_________|
        domain
```

In technically partitioned architectures such as the Layered architecture, the highest-level directory specifies the technical layer, such as `presentation`, `business`, or `persistence`.footnote[To learn more about how to identify subdomains and subdomains and refactor a system to realize these artifacts, refer to Chapter 5 of *Software Architecture: The Hard Parts*. For technically partitioned architectures, see Chapter 10 in *Fundamentals of Software Architecture*.] In this example, the business layer of the architecture, even though it's a technical layer, represents the domain in the system:

```
sysops.business.ticketing.creation
      |________|
```

`domain`

Services

In distributed architectures, the same directory and namespace concepts still apply, since each separately deployed service is defined within a specific domain and has an internal architecture made up of components (and possibly subdomains, depending on the size of the service). In the source code repository, the directory used to define the service is typically represented directly under the domain directory, as illustrated in Figure 2-5.

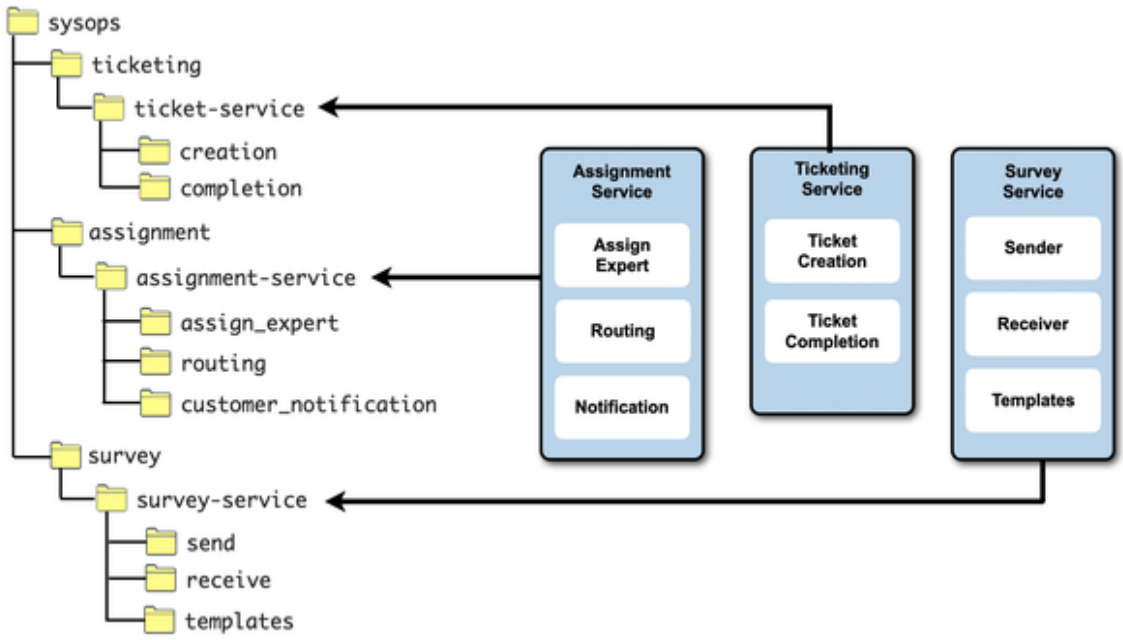


Figure 2-5. Services are represented through higher-level directories or namespaces, usually right below the domain directory

In this example we've identified three separately deployed services: `Ticketing`, `Assignment`, and `Survey`. Here, the higheest-level node of the directory structure *within each domain directory* represents the service, and the leaf nodes represent the corresponding logical components or subdomains contained in those services.

From a namespace perspective, the node after the domain node (in this case `ticketing-service`) represents the physical service:

```
sysops.ticketing.ticketing-service.creation
                 |_________________|
                       service
```

We'll use these concepts in the following chapters to explain how to describe and govern an architecture using the Architecture Definition Language (ADL).

Chapter 3. Describing Architecture Using ADL

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

By describing the structure of a system—its architecture—using Architecture Definition Language (ADL), architects can provide development teams (including AI agents) both a specification and guidance as to how to keep the source code aligned with the intended architecture during the implementation of the system. Architects can then use this ADL to generate executable source code to govern that structure, which we show in the next chapter.

The ADL in this chapter will focus only on the Ticketing and Survey portions of the Sysops Squad. Many other domains exist in the Sysops Squad system, such as `Reporting and Analytics`, `Administration`, and `Expert Knowledge Base`, but we've chosen to limit our example in this chapter for simplicity's sake, especially since these two domains contain all of the necessary architectural artifacts to completely show all the ADL features. Figure 3-1 shows the *logical architecture* we'll use for this chapter (and the next).

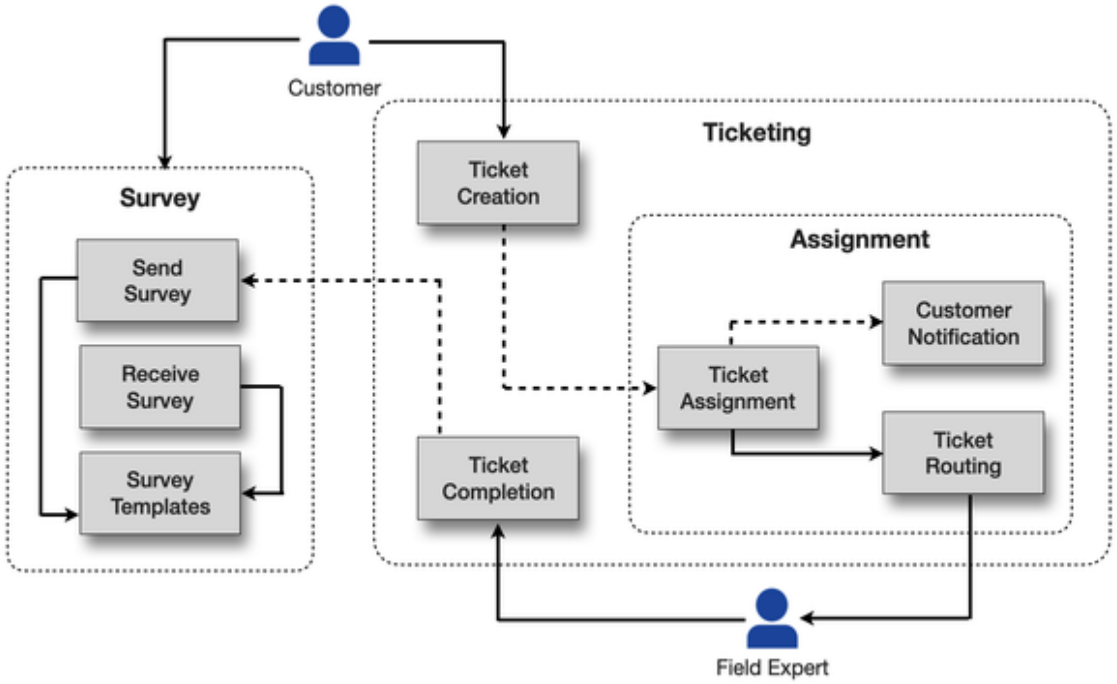


Figure 3-1. The logical architecture for the Ticketing and Survey portions of the Sysops Squad system

Example 3-1 shows the corresponding *directory structure* we'll use for this chapter and the next one.

Example 3-1. The directory structure for the Ticketing and Survey portions of the Sysops Squad system matching the logical architecture

```
sysops
├─ ticketing
│    ├─ creation
│    ├─ completion
│    └─ assignment
│         ├─ assign_expert
│         ├─ routing
│         └─ customer_notification
└─ survey
     ├─ send
     ├─ receive
     └─ templates
```

Describing The Basic Structure of an Architecture

The basic structure of an architecture, regardless whether it's a monolithic or distributed system, involves defining the domains of the architecture, the subdomains and components within each domain, and finally the interactions between components. Each of the following sections demonstrate how to describe each of these basic artifacts using ADL.

Defining Domains

Architects can describe the domains of a monolithic or distributed system by using the Architecture Definition Language (ADL), which we introduced in "Architecture Definition Language" in Chapter 1, to build a representation of the architecture as pseudocode. For example, the following ADL pseudocode in Example 3-2 defines the domains for the ticketing and survey portions of the Sysops Squad system.

Example 3-2. ADL for describing the domains in Sysops Squad

```
DESCRIPTION Describe Sysops Squad Domains
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
  DEFINE DOMAIN Survey AS survey

ASSERT(CLASSES are only CONTAINED within SUBDOMAINS within DOMAINS)
```

The `DESCRIPTION` and `CATEGORY` lines form part of the metadata that describes the ADL's context. The `DEFINE` verb defines a particular artifact, as advertised, by taking a logical name of the artifact and associating it with the name of the corresponding physical directory or namespace using the `AS` verb. For example, in the `DEFINE SYSTEM` line, the `Sysops Squad` part of this directive describes the logical name of the system, and the `sysops` part describes its physical location as either a directory or namespace. It's common for these names to be different, since development teams usually specify a physical directory or namespace node, whereas

architects usually specify the logical name of the corresponding domain, subdomain, or component. The ADL bridges the gap between the component's logical name, which shows up in diagrams and documentation, with its implementation name, which shows up in physical directories or namespaces.

Given the `DEFINE` parts of this ADL, the matching directory structure (or namespace structure) would look as follows, according to the `AS` directives:

```
sysops
├─ ticketing
│   └─ ...
└─ survey
    └─ ...
```

The `ASSERT` part of the ADL specifies the test condition or constraint. In this case, the ADL is asserting that all classes must exist somewhere under these two directory structures (which may include subdirectories representing subdomains or components), but nowhere else. Notice that here the ADL is only defining the domains, not any of the other subdirectories defining the components. Those definitions will come later in the following sections.

Asking an AI agent to interpret this architecture definition yields the results shown in Figure 3-2, demonstrating that both human and agentic development teams can understand the architecture through ADL from a domain perspective.

This reads as an **Architecture-as-Code (AaC) governance rule**, not implementation code — it's declaring structure and then asserting a constraint that can presumably be checked mechanically (e.g., against actual package/namespace layout) rather than just documented.

Line by line:

- `DESCRIPTION` / `CATEGORY Structural` — metadata for whatever tooling consumes this ADL. `Structural` classifies the rule as being about static decomposition/containment (as opposed to, say, a `Communication` or `Dependency` category rule elsewhere in the same language) — so a rule engine could group and report on it separately from behavioral or coupling rules.
- `DEFINE SYSTEM Sysops Squad AS sysops` — establishes the root scope/namespace (`sysops`) that everything else nests under.
- `DEFINE DOMAIN Ticketing AS ticketing` / `DEFINE DOMAIN Survey AS survey` — declares two top-level domains inside the system, aliased for reference elsewhere. This is a DDD-flavored decomposition: the system is being modeled as a set of bounded-context-like domains rather than, say, layers or services.
- `ASSERT(CLASSES are only CONTAINED within SUBDOMAINS within DOMAINS)` — this is the actual fitness function: a structural conformance rule stating that no class may live directly under a `System` or `Domain`; it must be nested one level deeper, inside a `Subdomain`. It enforces a fixed containment depth: `System → Domain → Subdomain → Class`.

Architecturally, the intent is: any class found floating directly inside `Ticketing` or `Survey` (skipping a subdomain) would violate this assertion. That's a common governance pattern for keeping a modular monolith (or a domain-oriented codebase generally) from eroding — it stops developers from bypassing the intended decomposition granularity by dropping classes straight into a domain package instead of organizing them into the proper subdomain.

Figure 3-2. AI interpretation of the ADL defining the Sysops Squad system's domains

Asking an AI agent to create an architecture diagram based on its understanding of the ADL defined in Example 3-2 yields the results shown in Figure 3-3.



Figure 3-3. An AI architecture diagram interpreting the ADL that defines the Sysops Squad system's architecture

Once the architect defines the high-level domains of the system, they can use ADL to describe the details of each of those domains in terms of subdomains and components.

Defining Domain Details

Architects can use ADL to define the details of a particular domain, including any subdomains and components. Each domain typically has its own ADL file describing its subdomains and components. However, architects can also specify the subdomain details of a particular domain in a separate ADL file, which we'll show later in this section.

Consider the Sysops Squad ticketing domain illustrated in Figure 3-4. This domain consists of two components, `Ticket Creation` and `Ticket Completion`. It has one subdomain called `Assignment`, which has three components: `Ticket Assignment`, `Ticket Routing`, and `Customer Notification`.

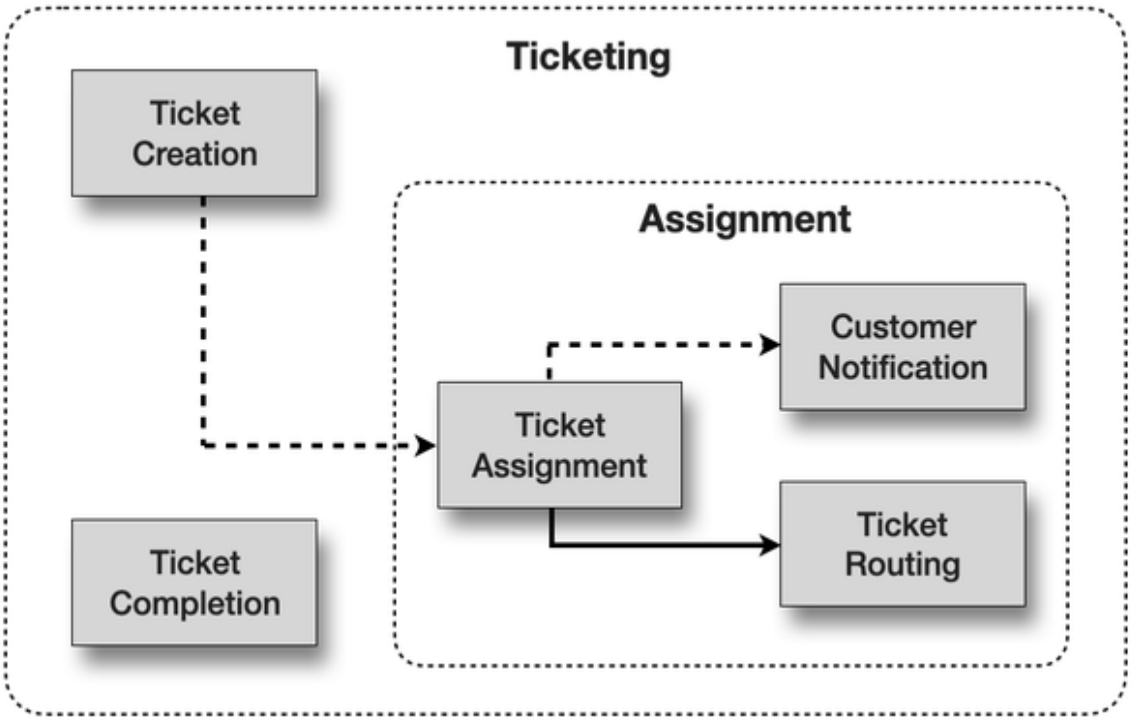


Figure 3-4. The logical architecture of the Ticketing portion of the Sysops Squad system

The corresponding ADL in Example 3-3 describes the `Ticketing` domain for the Sysops Squad system.

Example 3-3. ADL describing the Sysops Squad ticketing domain

```
DESCRIPTION Describe Ticketing Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE COMPONENT Ticket Creation AS creation
    DEFINE COMPONENT Ticket Completion AS completion
    DEFINE SUBDOMAIN Assignment AS assignment
      DEFINE COMPONENT Ticket Assignment AS assign_expert
      DEFINE COMPONENT Ticket Routing AS routing
      DEFINE COMPONENT Customer Notification AS
customer_notification

ASSERT(CLASSES are only CONTAINED within COMPONENTS)
```

The `DEFINE SUBDOMAIN` directive defines the `Assignment` subdomain, indicating that this is a separate subdirectory under the ticketing domain named `assignment`. This ADL further defines the components

within that subdomain by indenting the `DEFINE COMPONENT` directives under that subdomain.

The `ASSERT` directive now states that all source code must reside within a component, whether that component is defined under the domain or within a subdomain. This structure adheres to the rules of the Flattening pattern, which we describe in Chapter 8 of *Fundamentals of Software Architecture* (2nd edition). The pattern effectively pushes all source code to the leaf nodes of the directory structure or namespace so that every class belongs to a particular component. Classes that reside outside of a component (meaning directly under a domain or subdomain directory) are called *orphan classes* because they have no "home" (no component association).

Asking an AI agent to interpret this ADL defining the ticketing domain of the Sysops Squad system yields the diagram in Figure 3-5. This demonstrates that both human and agentic development teams can understand architecture through ADL from a complete domain-definition perspective.

Interpretation

The hierarchy here is `System → Domain → {Component | Subdomain → Component}` — and that "or" is the interesting part. `Ticketing` has two components (`creation`, `completion`) hanging directly off the domain, and a third branch, `Assignment`, that's promoted to a full subdomain before its three components (`assign_expert`, `routing`, `customer_notification`) appear.

That asymmetry is a real architectural statement, not just inconsistent modeling:

- **Creation and Completion** are treated as standalone, single-component concerns — simple enough that they don't need an intermediate grouping layer.
- **Assignment** is modeled as a subdomain because `assign_expert`, `routing`, and `customer_notification` are cohesive as a group — routing a ticket, assigning an expert, and notifying the customer are steps in one workflow, so they're clustered under a shared namespace rather than scattered as siblings of `creation`/`completion`.

Also notable: the `ASSERT` in this version only constrains **components**, not domains/subdomains like your earlier ADL did. Domains and subdomains here are purely organizational — namespacing devices. `COMPONENT` is the actual unit of encapsulation; the rule says a class must live inside *some* component, but is indifferent to whether that component sits at domain depth or subdomain depth. So the architecture explicitly allows variable nesting depth as long as every leaf is a component.

Figure 3-5. AI interpretation of the ADL-defined domains within the Sysops Squad system

Asking an AI agent to generate an architecture diagram using the ADL defined in Example 3-3 yields the results shown in Figure 3-6.

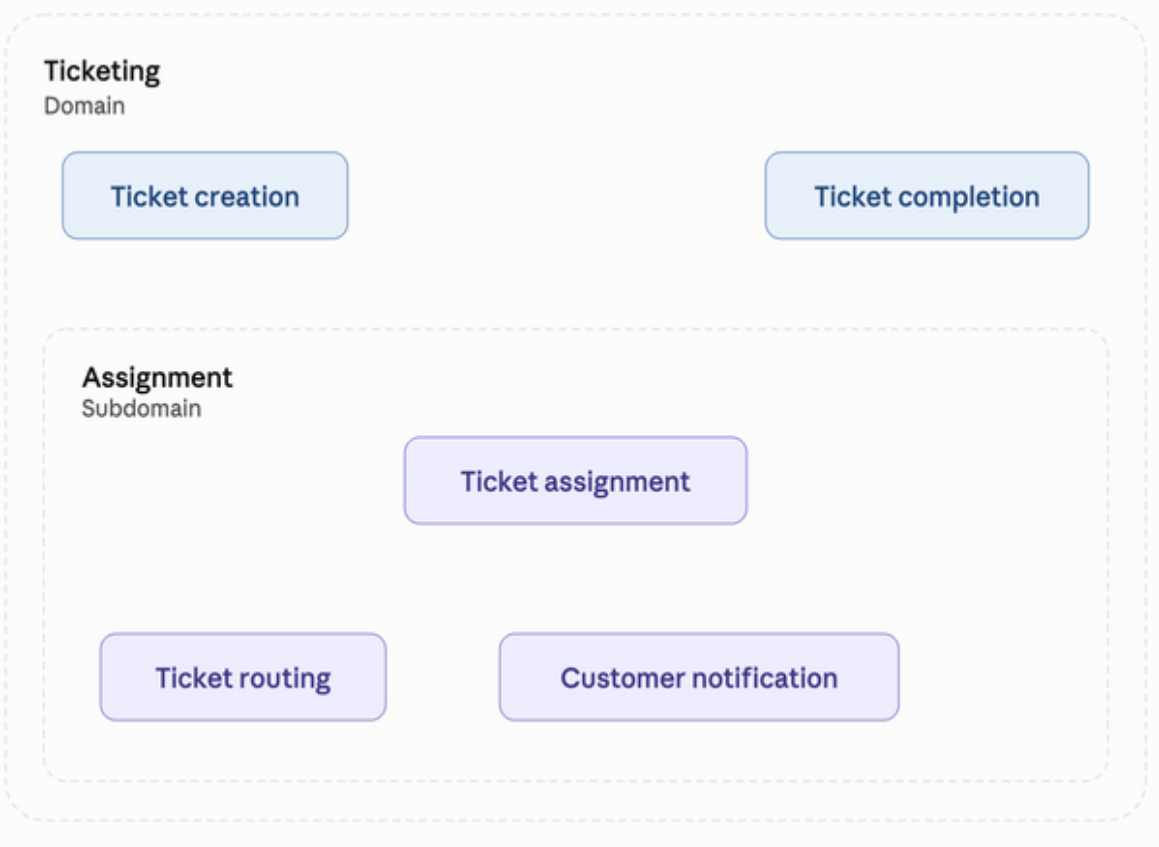


Figure 3-6. AI interpretation of the Sysops Squad system's ADL-defined ticketing domain

So far we've demonstrated how to use ADL to specify all of a system's domain details. However, domains can get quite large and contain many subdomains and components, so architects can also describe domains in terms of the components that reside directly under them, along with any subdomains directly under that domain. This does not include the details of the subdomain, as in Example 3-3; those details are described in separate ADL files that end with the component definitions.

For example, it's possible to break the ADL describing the entire ticketing domain into two separate ADL files: one specifying the `Ticketing` Domain, the other specifying the `Assignment` subdomain. For example, Example 3-4 shows revised ADL specifying only the `Ticketing` domain, without the `Assignment` details.

Example 3-4. ADL describing the ticketing domain in Sysops Squad, without the assignment subdomain details

```
DESCRIPTION Describe Ticketing Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE COMPONENT Ticket Creation AS creation
    DEFINE COMPONENT Ticket Completion AS completion
    DEFINE SUBDOMAIN Assignment AS assignment

ASSERT(CLASSES are only CONTAINED within COMPONENTS and SUBDOMAINS)
```

Notice that the `ASSERT` statement from the prior ADL has changed: it now specifies that only components *and* subdomains can contain classes. This tells an AI agent or a governance-code generator (such as ArchUnit for Java) that the `Assignment` subdomain will eventually contain some source code. (We'll describe this in more detail in the next chapter).

Example 3-5 shows the corresponding ADL specifically describing the `Assignment` subdomain.

Example 3-5. ADL for describing the assignment subdomain under the ticketing domain in Sysops Squad

```
DESCRIPTION Describe Assignment Subdomain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE SUBDOMAIN Assignment AS assignment
      DEFINE COMPONENT Ticket Assignment AS assign_expert
      DEFINE COMPONENT Ticket Routing AS routing
      DEFINE COMPONENT Customer Notification AS
customer_notification

ASSERT(CLASSES are only CONTAINED within COMPONENTS)
```

Although this ADL is only concerned about the `Assignment` subdomain, the architect must still specify the `DEFINE DOMAIN Ticketing AS ticketing` directive and the `DEFINE SYSTEM` directive to provide a context for the subdomain. We don't consider this a significant trade-off, since these artifacts don't change often.

When splitting a domain definition into multiple ADL files, the development team or agent would need all of the ADL files describing a particular domain to understand that entire domain.

Defining Component Interactions

The last part of the basic system structure is defining component interactions. For example, Figure 3-1 at the beginning of the chapter shows the logical architecture of the ticketing and survey domains. (Solid lines between components represent direct communication; dotted lines represent indirect, asynchronous communication.) It includes the following component interactions for the `Ticketing` domain:

- The `Ticket Assignment` component communicates directly with the `Ticket Routing` component to upload the ticket to the expert's mobile device through the Sysops Squad mobile app.
- The `Ticket Assignment` component communicates indirectly (asynchronously) with the `Customer Notification` component to email the customer that the system has assigned an expert and that the expert is on their way.
- The `Ticket Creation` component communicates indirectly (asynchronously) with the `Ticket Assignment` component to let it know the customer just created an unassigned ticket to be assigned to an expert.
- The `Ticket Completion` component communicates indirectly (asynchronously) with the `Send Survey` component to send a survey to the customer.

At this point, all we need to specify in the ADL is which components communicate with each other directly through method calls between classes. (We'll explain remote communication in more detail in [Link to Come], using the `DEFINE SAGA` and `DEFINE COMMUNICATION` directives.)

Coming back to the ADL in Example 3-3 describing the `Ticketing` domain, we can now add the component interactions, using the `ASSERT` directive to describe them as constraints (see Example 3-6).

Example 3-6. ADL describing the Sysops Squad ticketing domain in Sysops Squad and component interactions

```
DESCRIPTION Describe Ticketing Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE COMPONENT Ticket Creation AS creation
    DEFINE COMPONENT Ticket Completion AS completion.
    DEFINE SUBDOMAIN Assignment AS assignment
      DEFINE COMPONENT Ticket Assignment AS assign_expert
      DEFINE COMPONENT Ticket Routing AS routing
      DEFINE COMPONENT Customer Notification AS
customer_notification

# Structural assertions
ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Allowed dependencies
ASSERT(Ticket Assignment IS DEPENDENT ON Ticket Routing)

# Disallowed dependencies
ASSERT(Ticket Assignment HAS NO DEPENDENCY ON
       Ticket Creation, Ticket Completion, Customer Notification)
ASSERT(Ticket Creation HAS NO DEPENDENCY ON other components)
ASSERT(Ticket Completion HAS NO DEPENDENCY ON other components)
ASSERT(Ticket Routing HAS NO DEPENDENCY ON other components)
ASSERT(Customer Notification HAS NO DEPENDENCY ON other components)
```

This ADL doesn't include the indirect (asynchronous) communication between components because it only describes *direct* interactions—method calls—between classes within those components.

Notice that the `HAS NO DEPENDENCY ON` assertion explicitly specifies what this ADL *prohibits*; the `IS DEPENDENT ON` assertion shows what component dependency the ADL *allows*.

Asking an AI agent to interpret this ADL and draw a diagram of the architecture (including the relationships between components) yielded the

results in Figure 3-7.

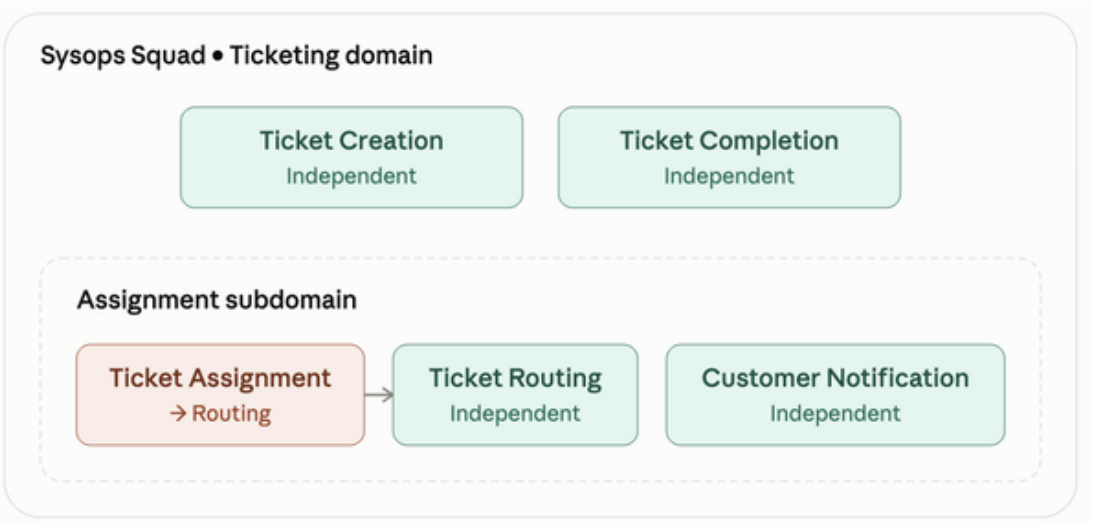


Figure 3-7. AI interpretation of the Sysops Squad ticketing domain and its ADL-defined component dependencies

It might be tempting to shortcut all of the explicit dependency assertions and use an alternative form, such as the following ADL snippet:

```
...
ASSERT(Ticket Assignment IS ONLY DEPENDENT ON Ticket Routing)
...
```

Using the `IS ONLY DEPENDENT ON` directive implies that no other component relationships can exist with respect to the `Ticket Assignment` component, meaning architects could skip the following assertion:

```
...
ASSERT(Ticket Assignment HAS NO DEPENDENCY ON
       Ticket Creation, Ticket Completion, Customer Notification)
...
```

While development teams and AI agents interpret this correctly, using the ADL to generate executable fitness functions to govern this constraint can

cause issues (more details in the next chapter). For example, generating a fitness function with the `IS ONLY DEPENDENT ON` directive using NetArchtest in C# yields the following executable source code:

```
// ASSERT(Ticket Assignment IS ONLY DEPENDENT ON Ticket Routing)
...
var result = Types.InCurrentDomain()
    .That().ResideInNamespace(AssignmentNs)
    .Should().OnlyHaveDependenciesOn(AssignmentNs, RoutingNs)
    .GetResult();

Assert.True(result.IsSuccessful, result.GetFailingTypes());
...
```

This generated code specifies *exclusive* dependencies with the `OnlyHaveDependenciesOn` method call. This code correctly specifies the constraint that the `Ticket Assignment` component can only instantiate classes within itself and the `Ticket Routing` component. The problem is that it also prevents the `Ticket Assignment` component from instantiating *all* other library and framework classes (such as logging libraries and .Net framework utilities).

To address this problem, architects can use the `IS DEPENDENT ON` directive instead and create assertions for all other component-dependency constraints explicitly. Doing so in the ADL, then testing it with AI yields the following source code:

```
// ASSERT(Ticket Assignment IS DEPENDENT ON Ticket Routing)
...
var result = Types.InCurrentDomain()
    .That().ResideInNamespace(AssignmentNs)
    .Should().HaveDependencyOn(RoutingNs)
    .GetResult();

Assert.True(result.IsSuccessful, result.GetFailingTypes());
...
```

No longer using the `HaveDependencyOn` method call exclusively allows the `Ticket Assignment` to instantiate other framework-related classes. Other parts of this generated source code include rules about which other components the `Ticket Assignment` component should *not* communicate with.

Not all LLMs will generate code with this issue, but it's something to watch out for. For this reason, we refrain from using the `IS ONLY DEPENDENT ON` directive, and instead specify all component relationships explicitly.

The `Survey` domain contains more complicated direct component dependencies. The logical architecture diagram in Figure 3-1 depicts the following component interactions in the `Survey` domain:

- The `Send Survey` component communicates directly with the `Survey Templates` component to find out which product the customer had an issue with, so it can send the correct type of survey.
- The `Receive Survey` component communicates directly with the `Survey Templates` component to learn how to parse the survey it receives from the customer.

One way to specify relationships in the `Survey` domain is to prohibit certain relationships explicitly, while implicitly allowing anything outside of those constraints. Example 3-7 shows this in ADL.

Example 3-7. ADL describing the Sysops Squad survey domain by implicitly specifying the allowable component dependencies

```
DESCRIPTION Describe Survey Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
DEFINE DOMAIN Survey AS survey
  DEFINE COMPONENT Send Survey AS send
  DEFINE COMPONENT Receive Survey AS receive
  DEFINE COMPONENT Survey Templates AS templates

# Structural assertions
ASSERT(CLASSES are only CONTAINED within COMPONENTS)
```

```
# Dependency assertions
ASSERT(Send Survey HAS NO DEPENDENCY ON Receive Survey)
ASSERT(Receive Survey HAS NO DEPENDENCY ON Send Survey)
ASSERT(Survey Templates HAS NO DEPENDENCY ON Send Survey or Receive
Survey)
```

While this ADL explains which components should not communicate with other components, it doesn't provide guidance on what the component relationships should actually be. It allows, but does *not* require, dependencies between the `Send Survey` component and the `Survey Templates` component, as the following AI agent interpretation shows:

```
Send Survey ----\
                 --> Survey Templates (allowed, not required)
Receive Survey --/

Send Survey  ↔  Receive Survey (forbidden, both directions)
```

This interpretation is misleading. An AI agent or human developer might not know that the architecture requires this relationship, and could easily miss it during implementation. When we asked an AI agent to diagram this ADL, it produced the architecture in Figure 3-8, which does *not* match the intended logical architecture from Figure 3-1.

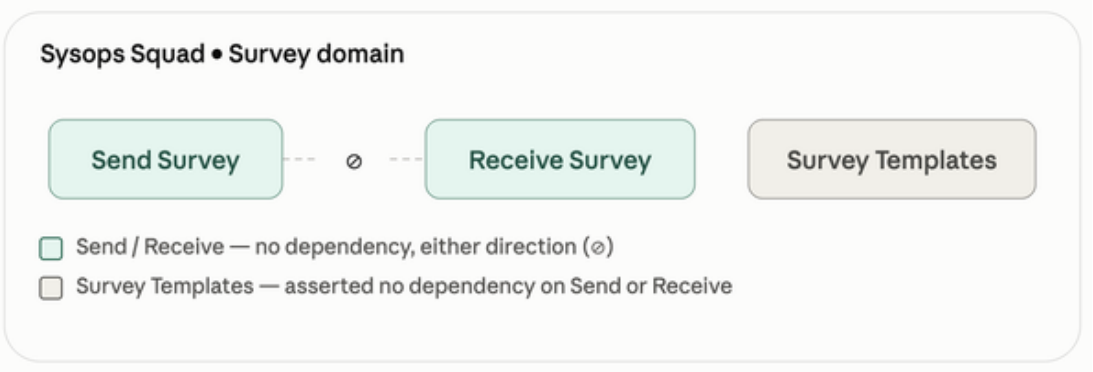


Figure 3-8. AI interpretation of the survey domain and its component dependencies as implied in the ADL

Again, while this ADL works from a governance perspective, it doesn't *explicitly* state what the component interactions should be. This type of implicit ADL description can be useful when an architect isn't what the interactions will be, or when they want to give development teams the flexibility to call other components as necessary. However, we caution against this practice because it doesn't describe what the component interactions actually are, only what is permissible.

The `IS DEPENDENT ON` assertion directive in ADL represents the nature of the component dependencies explicitly, not implicitly, and therefore more accurately. Example 3-8 shows the revised ADL, this time providing an explicit component dependency relationship between the send and receive components and the `Survey Templates` component.

Example 3-8. ADL describing the survey domain by explicitly specifying the component dependencies

```
DESCRIPTION Describe Survey Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
DEFINE DOMAIN Survey AS survey
  DEFINE COMPONENT Send Survey AS send
  DEFINE COMPONENT Receive Survey AS receive
  DEFINE COMPONENT Survey Templates AS templates

# Structural assertions
ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Allowed dependency assertions
ASSERT(Send Survey IS DEPENDENT ON Survey Templates)
ASSERT(Receive Survey IS DEPENDENT ON Survey Templates)

# Disallowed dependency assertions
ASSERT(Send Survey HAS NO DEPENDENCY ON Receive Survey)
ASSERT(Receive Survey HAS NO DEPENDENCY ON Send Survey)
ASSERT(Survey Templates HAS NO DEPENDENCY ON Send Survey or Receive
Survey)
```

This revised ADL specifies the explicit component dependencies, including what isn't allowed: the `Survey Templates` component shouldn't have a reverse dependency on the send or receive components, and that the send

and receive components cannot depend on each other. However, using `IS DEPENDENT ON` explicitly forces a component relationship between the send and receive components and the `Survey Templates` component:

```
Send Survey ----\
                 --> Survey Templates (required, exclusive)
Receive Survey --/

Send Survey  ↔  Receive Survey (forbidden, both directions)
```

Here, the AI agent correctly interpreted that the ADL now requires an exclusive relationship between the send and receive components and the template component. Asking an AI agent to diagram this ADL produces the architecture in Figure 3-9, which now matches the intended logical-architecture diagram.

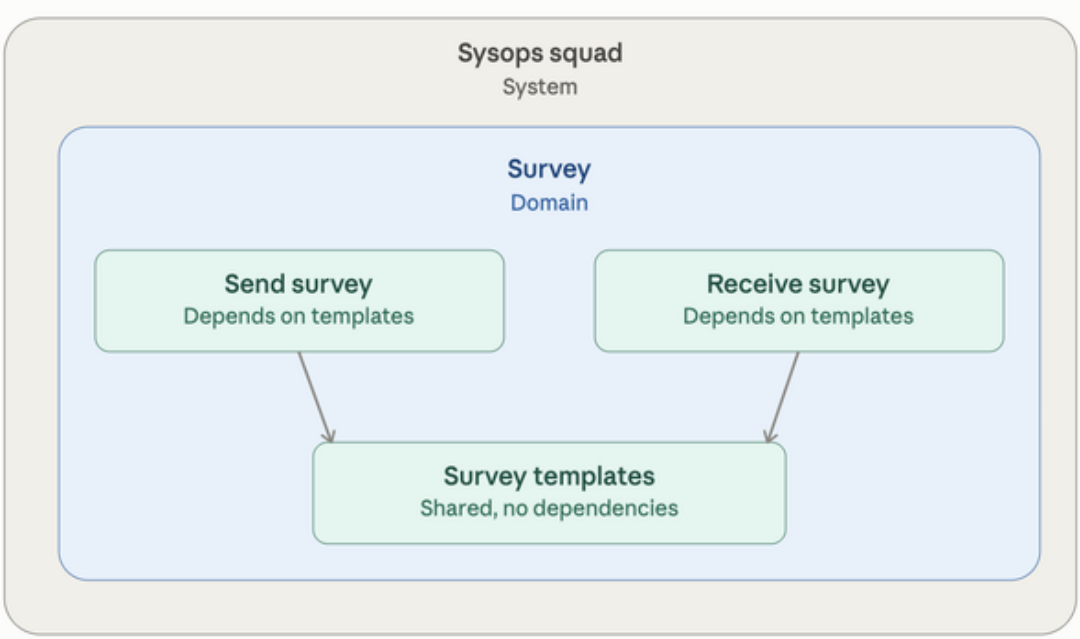


Figure 3-9. AI interpretation of the survey domain and its component dependencies using ADL that explicitly describes the component interactions

Component relationships can get a little verbose in the ADL for larger and more complex systems, but is nevertheless a critical part of describing and governing the architecture. This specificity helps prevent unnecessary

component coupling, which can quickly turn a well-structured architecture into a big ball of mud.

Defining Constraints

To understand how ADL can describe and govern architectural constraints, let's consider a scenario. An architect working on a greenfield software project faces a tight timeline and a small budget. Since this is a new line of business for the company, they know that the database's structure will change constantly. When it does, the business requires the development team to implement and release those changes as fast as possible.

These requirements lead the architect to choose the traditional N-tiered Layered architecture for this system. This architecture is an appropriate choice because it is technically partitioned. (Recall that a technically partitioned architectural style is determined based on the technical usage of the components within its highest level, such as presentation behavior, backend business behavior, and persistence behavior.)

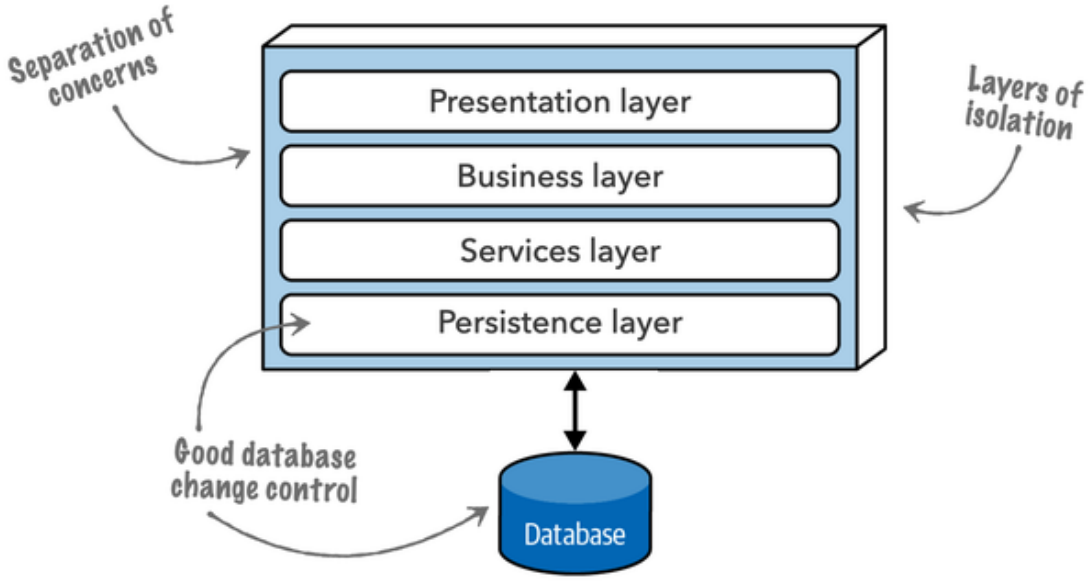


Figure 3-10. The Layered architecture lets teams isolate the database logic, providing better agility for database changes

As [Link to Come] shows, the Layered architectural style supports a good separation of concerns and layers of isolation. The Persistence layer will

form an abstraction layer over the database, isolating the underlying database structure and type from the rest of the layers. This allows teams to change the database structure, or even replace the underlying database type, quickly and easily.

Layers of isolation are an important concept in layered architectures. Isolating layers within the architecture and defining contracts between them allows each layer to change independently from the other layers. That reduces the system's overall brittleness and improves its agility. Architects usually achieve layers of isolation using open and closed layers. If a layer is *closed*, all requests from the layer above it *must* go through the layer below it, even if it's only an unnecessary passthrough. An *open* layer means the layer above can access the layer below it *or* step over it.

[Link to Come] shows open layers. To access the Persistence layer, the Business layer shouldn't have to go through the Services layer; it should be able to bypass the Services layer to access the Persistence layer directly. To accomplish this, the architect would open the Services layer so that the Business layer can bypass it.

The diagram also shows a good example of *closed* layers: The Services layer contains all of the shared service functionality for the Business layer, but *not* for the Presentation layer. Allowing the Presentation layer to access shared business services would couple those layers, meaning that changing a shared service for a business object would also require changing objects in the Presentation layer.To restrict the Presentation layer from accessing the shared Services layer, the architect would close the Business layer. This forces the Presentation layer must go through the Business layer when making a request, preventing it from accessing the underlying Services layer.

The architect will need to specify the following constraints to achieve fast and easy database replacement and preserve layers of isolation:

1. *All* database logic must be contained *only* within the Persistence layer.

2. Only the Persistence layer can access the database directly.
3. The system must respect open and closed layers, meaning the Presentation layer cannot talk to the Services or Persistence layers.
4. No layer can depend on the one above it.

These constraints, as Figure 3-11 illustrates, would achieve the business goal of making it fast and easy to change the database structure or database type, because only the Persistence layer contains the database logic. Reducing coupling between the layers would also decrease brittleness, supporting the notion that *all* layers of the architecture should be isolated.

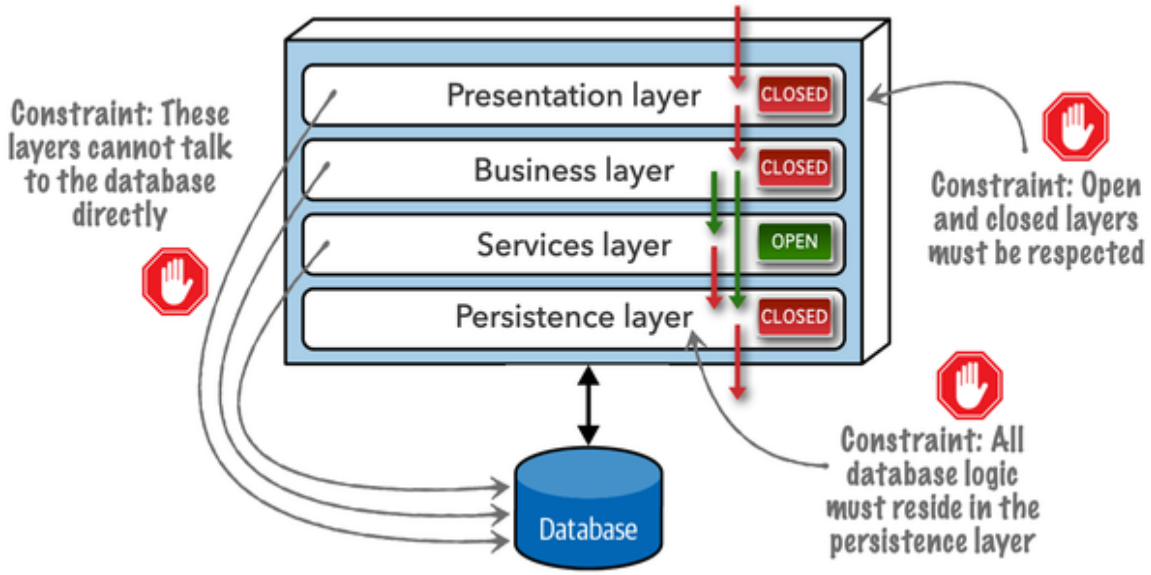


Figure 3-11. Constraints within the layered architecture to support quick and easy database changes and layers of isolation

An architect can specify these constraints using the ADL in Example 3-9.

Example 3-9. ADL describing the Layered architecture's dependency and database-logic constraints

```
DESCRIPTION Describe Layered Architecture Constraints
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE LAYER Presentation Layer AS presentation
  DEFINE LAYER Business Layer AS business
  DEFINE LAYER Services Layer AS services
  DEFINE LAYER Persistence Layer AS persistence
```

```
# Structural assertions
ASSERT(CLASSES are only CONTAINED within COMPONENTS and SUBDOMAINS
within LAYERS)

# Allowed dependencies
ASSERT(Presentation Layer IS DEPENDENT ON Business Layer)
ASSERT(Business Layer IS DEPENDENT ON Services Layer, Persistence
Layer)
ASSERT(Services Layer IS DEPENDENT ON Persistence Layer)

# Disallowed dependencies
ASSERT(Presentation Layer HAS NO DEPENDENCY ON Services Layer,
Persistence Layer)
ASSERT(Business Layer HAS NO DEPENDENCY ON Presentation Layer)
ASSERT(Services Layer HAS NO DEPENDENCY ON Business Layer,
Presentation Layer)
ASSERT(Persistence Layer HAS NO DEPENDENCY ON any other layer)

# Database logic containment
ASSERT(Persistence Layer is the only layer that contains database
logic)
```

The directive `DEFINE LAYER` serves the same purpose as `DEFINE DOMAIN`: describing the system's highest-level structure. Using the `DEFINE DOMAIN` directive to describe the layers also works.

The allowed-dependency assertions in this ADL specify the downward dependencies of the Layered architecture. Its disallowed dependency assertions prevent upward dependencies, as well as preventing a layer from bypassing a closed layer beneath it.

Asking an AI agent to interpret this architecture produces the correct results shown in Figure 3-12.

This ADL describes a **strict/closed layered architecture** for the Sysops Squad system, with four layers and specific rules about how they can depend on each other.

Structure

- `presentation` → `business` → `services` → `persistence`

Allowed dependencies — each layer may only reach "downward":

- Presentation → Business
- Business → Services *and* Persistence
- Services → Persistence

Note the Business Layer gets an explicit exception: it can skip Services and go straight to Persistence. That's a deliberate relaxation of a *strict* layered architecture (where each layer may only talk to the layer immediately below it) into what's often called a **relaxed layered architecture** — some layers are allowed to bypass their neighbor.

Disallowed dependencies — the assertions also rule out:

- Any upward calls (Business → Presentation, Services → Business/Presentation) — this enforces one-directional flow, so a lower layer never knows about or calls back into a higher one.
- Presentation skipping directly to Services (it must go through Business).
- Persistence depending on *anything* — it's a true architectural sink, the bottom of the stack.

Database logic containment

The last assertion is a different kind of constraint — not about dependency direction but about **responsibility isolation**: only the Persistence Layer may contain database logic (SQL, ORM calls, connection handling, etc.). Even though Business and Services can call Persistence, they aren't allowed to embed their own database code — that would violate separation of concerns and let data-access logic leak upward into layers that shouldn't own it.

Figure 3-12. AI agent's interpretation of the ADL describing the Layered architecture

The most interesting part of this ADL is the assertion that the *Persistence layer is the only layer that contains database logic*. This assertion demonstrates the power of ADL as a pseudocode language rather than a strict syntax specification. The interpretation in Figure 3-12 demonstrates that this is a constraint that both humans and AI agents can easily interpret. This means architects don't have to be overly prescriptive, adding every kind of database package or library to the ADL assertion. It also means the ADL remains agnostic about the implementation.

Asking an AI agent to produce a diagram from the prior ADL yields the correct understanding of the dependencies that Figure 3-13 illustrates.

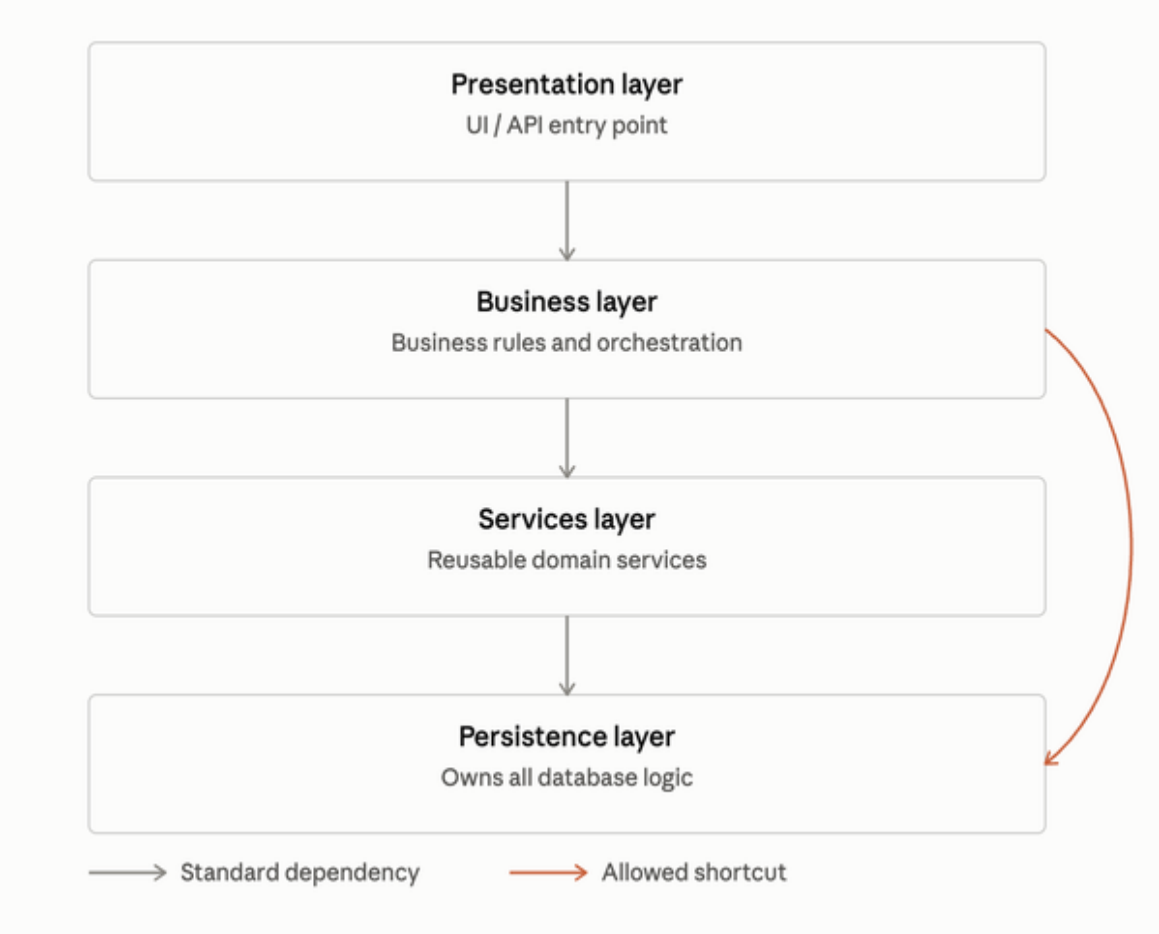


Figure 3-13. An AI interpretation of the ADL describing the Layered architecture

Distributed Architectures

We've shown how to use ADL to describe logical architecture in terms of domains, subdomains, and components. In this section, we show how to use ADL to describe *physical* architecture through the source code structure in terms of separately deployed services. We'll go on to cover the physical database structure in Part III, and the physical architecture with respect to synchronous and asynchronous communication between services in [Link to Come].

Defining Domain Services

Domain services are separately deployed, coarse-grained services typically found in Service-Based architectures.[1] These services typically encompass all the behavior of a particular domain in the system, and rarely communicate among themselves.

Figure 3-14 illustrates two domain services within the Sysops Squad system, for the `Ticketing` and `Survey` domains.

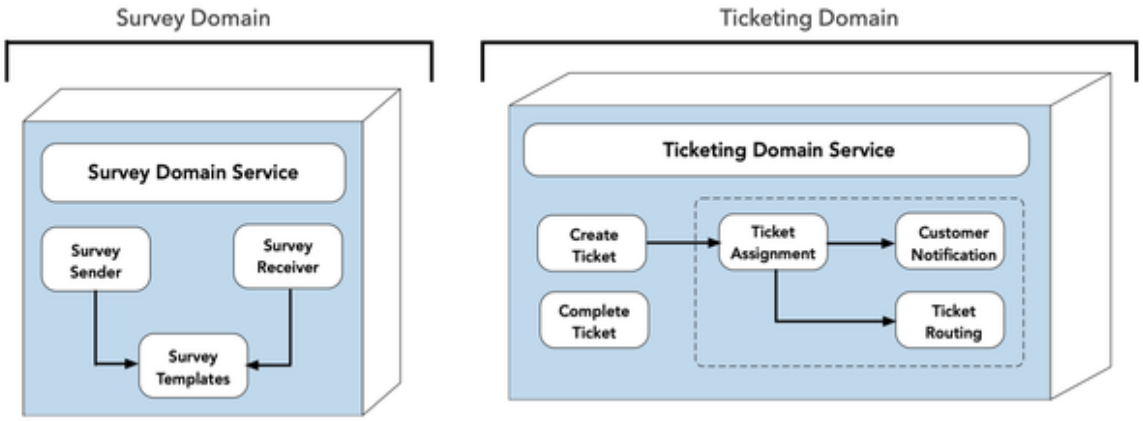


Figure 3-14. Sysops Squad Ticketing and Survey Domain services

Example 3-10 illustrates the corresponding directory structure for these domain services, including the components and subdomains within each service.

Example 3-10. Directory structure for the domain services described in the ADL

```
sysops
├─ ticketing
│    └─ ticketing_service ❶
│         ├─ creation
│         ├─ completion
│         └─ assignment
│              ├─ assign_expert
│              ├─ routing
│              └─ customer_notification
└─ survey
     └─ survey_service ❶
          ├─ send
          ├─ receive
          └─ templates
```

❶ The directory representing the domain service is directly under the corresponding domain directory. This common practice maintains the domain structure of the system, whether there's only a single domain service or multiple under each domain. For example, the architect could have chosen to create two domain services rather than one for the `ticketing_service`: one for ticketing and one for assignment. Without the domain designation, these services would no longer be associated with a domain.

Given the coarse-grained nature of domain services, architects typically define components (and even subdomains like `assignment`) within the service definition, as the directory structure and the architecture diagram in Figure 3-14 show. We recommend first defining the architecture through a high-level ADL file describing all the services and domains.

Example 3-11. ADL describing the high-level domain-service structure

```
DESCRIPTION Describe Domain Services
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE SERVICE Ticketing Service as ticketing_service
  DEFINE DOMAIN Survey AS survey
    DEFINE SERVICE Survey Service as survey_service

# Structural class-to-component assertion
ASSERT(CLASSES are only CONTAINED within COMPONENTS within
SERVICES)

# Dependency assertions
ASSERT(Ticketing Service HAS NO DEPENDENCY ON other services)
ASSERT(Survey Service HAS NO DEPENDENCY ON other services)
```

This ADL contains a new directive, `DEFINE SERVICE`, which associates the source code belonging to the separately deployed artifact with the directory that the `AS` directive defines. The directory identifying the service may or may not be under a particular domain directory. While we recommend defining services under the scope of a domain, architects not

following this practice can simply skip the `DEFINE DOMAIN` directive in the ADL.

The `ASSERT` directives in this ADL only describe the static component dependencies through source code, indicating that no classes within each domain service can instantiate other classes within another domain service. These static assertions in the ADL are critical for defining and protecting the service boundaries, which is particularly useful during deployment when using a monorepo. Architects can describe dynamic remote interservice calls, if any, using the `DEFINE SAGA` directive (see [Link to Come]).

Describing the high-level services in ADL provides a high-level architectural view of the system. This way, development teams and AI agents don't have to look at separate ADL files describing each domain service to determine how many domain services the system contains. Notice that this ADL file doesn't describe the details of the domain services' components and subdomains-—that will come next.

Asking an AI agent to interpret this high-level ADL yields the following statement and the diagram in Figure 3-15:

This ADL describes a domain-driven, service-oriented decomposition of Sysops Squad, with each domain owning exactly one service, and a dependency graph asserted as an explicit constraint set (fitness-function style) rather than left implicit in code.



Figure 3-15. AI agent interpretation of the ADL describing the high-level domain services

Once that high-level description is in place, the architect can describe the details of each domain service, including subdomains, components, and assertions that reflect the code structure and component dependencies. For example, the ADL in Example 3-12 describes the `Ticketing` domain service.

Example 3-12. ADL for describing the ticketing domain service of the Sysops Squad system

```
DESCRIPTION Describe Ticketing Domain Service
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE SERVICE Ticketing Service as ticketing_service
      DEFINE COMPONENT Create Ticket AS creation
      DEFINE COMPONENT Complete Ticket AS completion
      DEFINE SUBDOMAIN Assignment AS assignment
        DEFINE COMPONENT Ticket Assignment AS assign_expert
        DEFINE COMPONENT Ticket Routing AS routing
        DEFINE COMPONENT Customer Notification AS
customer_notification

# Structural class-to-component assertion
ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Component dependency assertions
ASSERT(Create Ticket IS DEPENDENT ON Ticket Assignment)
ASSERT(Ticket Assignment IS DEPENDENT ON Ticket Routing and
Customer Notification)
ASSERT(Complete Ticket HAS NO DEPENDENCY ON other components)
ASSERT(Ticket Routing HAS NO DEPENDENCY ON other components)
ASSERT(Customer Notification HAS NO DEPENDENCY ON other components)
```

This ADL is similar to Example 3-3 in "Defining Domain Details", only this time, it appears within the context of a *service*, not a monolithic system. Asking an AI agent to interpret and diagram this ADL produces the architecture diagram in Figure 3-16.

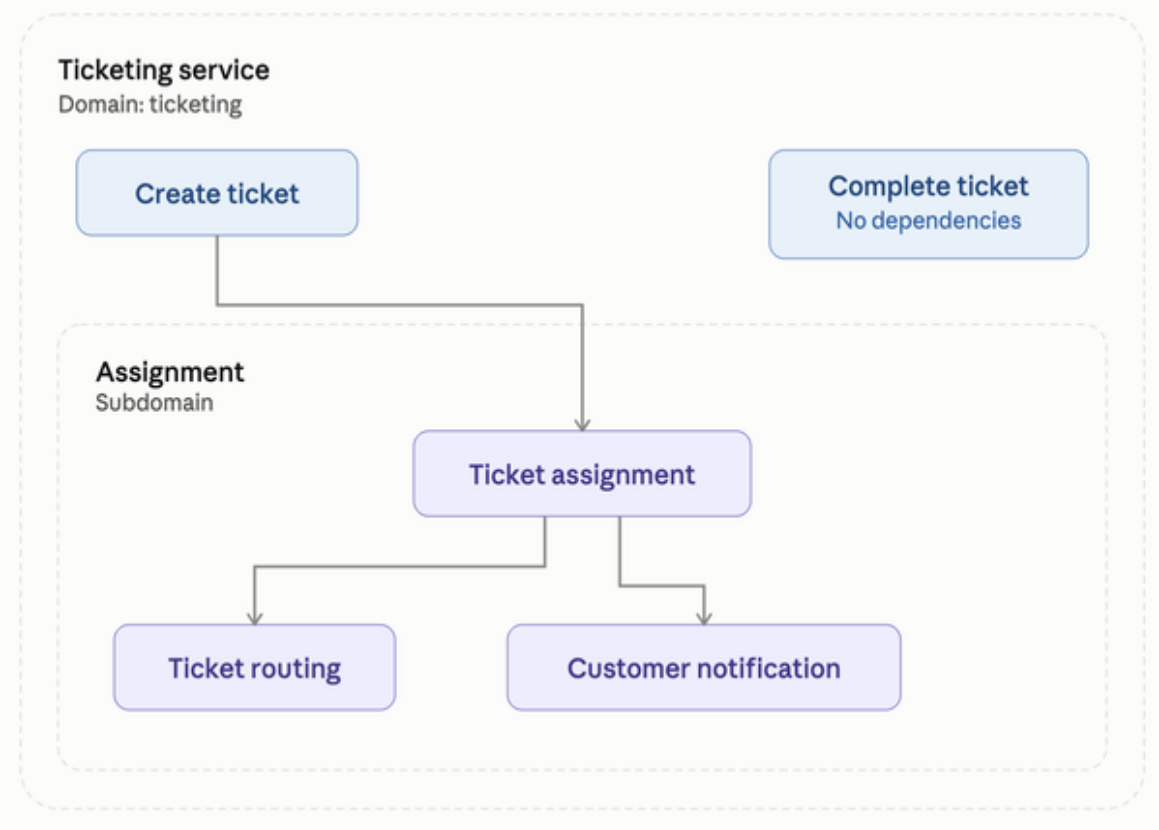


Figure 3-16. AI agent interpretation of the ADL describing the components of the ticketing domain service

Similarly, the ADL in Example 3-13 lends itself well to defining the `Survey` domain service.

Example 3-13. ADL describing the Survey domain service

```
DESCRIPTION Describe Survey Domain Service
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Survey AS survey
    DEFINE SERVICE Survey Service as survey_service
      DEFINE COMPONENT Send Survey AS send
      DEFINE COMPONENT Receive Survey AS receive
      DEFINE COMPONENT Survey Templates AS templates

# Structural class-to-component assertion
ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Dependencies for Survey domain service
ASSERT(Send Survey IS DEPENDENT ON Survey Templates)
ASSERT(Receive Survey IS DEPENDENT ON Survey Templates)
ASSERT(Send Survey HAS NO DEPENDENCY NO Receive Survey)
```

```
ASSERT(Receive Survey HAS NO DEPENDENCY ON Send Survey)
ASSERT(Survey Templates HAS NO DEPENDENCY ON other components)
```

The AI agent that created the diagram in Figure 3-17 has correctly interpreted the `Survey` domain service ADL's pseudocode description.

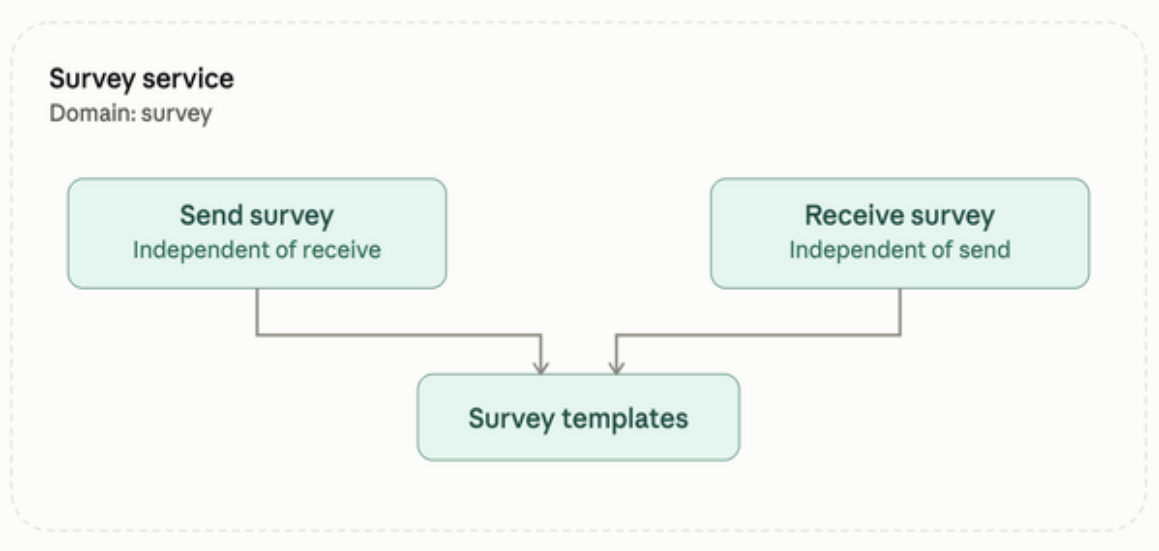


Figure 3-17. AI agent interpretation of the ADL describing the components of the survey domain service

Domain-Based Repositories

When using *domain-based repos* (source-code repositories dedicated to a single domain), it's tempting to skip the higher-level ADL describing all of the domains and their corresponding domain services. The `AS` directive references source code from multiple repositories, which makes it difficult to use the ADL to generate executable fitness functions to govern these services.

Leveraging the ADL `NAMESPACE` directive is the solution here. Assuming that the team is using domain-based repos, the architect can add a `NAMESPACE` reference to each ADL file describing each domain and domain service. For example:

```
DESCRIPTION Ticketing Survey Domain Service
NAMESPACE TicketingNS
```

```
  CATEGORY Structural
  DEFINE SYSTEM Sysops Squad AS sysops
    DEFINE DOMAIN Ticketing AS ticketing
      DEFINE SERVICE Ticketing Service as ticketing_service
  ...



  DESCRIPTION Describe Survey Domain Service
  NAMESPACE SurveyNS
  CATEGORY Structural
  DEFINE SYSTEM Sysops Squad AS sysops
    DEFINE DOMAIN Survey AS survey
      DEFINE SERVICE Survey Service as survey_service
  ...
```

This code uses the `NAMESPACE` directives to describe the high-level architecture by referencing the specific services within their separate ADL files, as Example 3-14 shows.

Example 3-14. ADL describing the high-level architecture using namespaces

```
DESCRIPTION Describe Domain Services
CATEGORY Structural
DEFINE SYSTEM Sysops Squad
  DEFINE DOMAIN Ticketing
    DEFINE SERVICE TicketingNS.Ticketing Service
  DEFINE DOMAIN Survey AS survey
    DEFINE SERVICE SurveyNS.Survey Service
```

Notice the absence of the `AS` directive and the `ASSERT` directives. This is intentional as the ADL files describing each domain service contain the references specific to the domain-based repository.

Defining Microservices

The process for defining microservices through ADL is similar to that for defining domain services, with two exceptions: first, a domain typically contains multiple microservices, and second, microservices commonly use

shared libraries (such as a JAR or DLL). We'll focus on the `Survey` domain as we describe both exceptions in this section.

The architect chose to create two separate microservices within this domain, as Figure 3-18 illustrates: a `Survey Sender` service that emails the survey to the customer, and a `Survey Receiver` service that accepts and records the survey results from the customer. Both microservices use a shared library containing the survey templates.

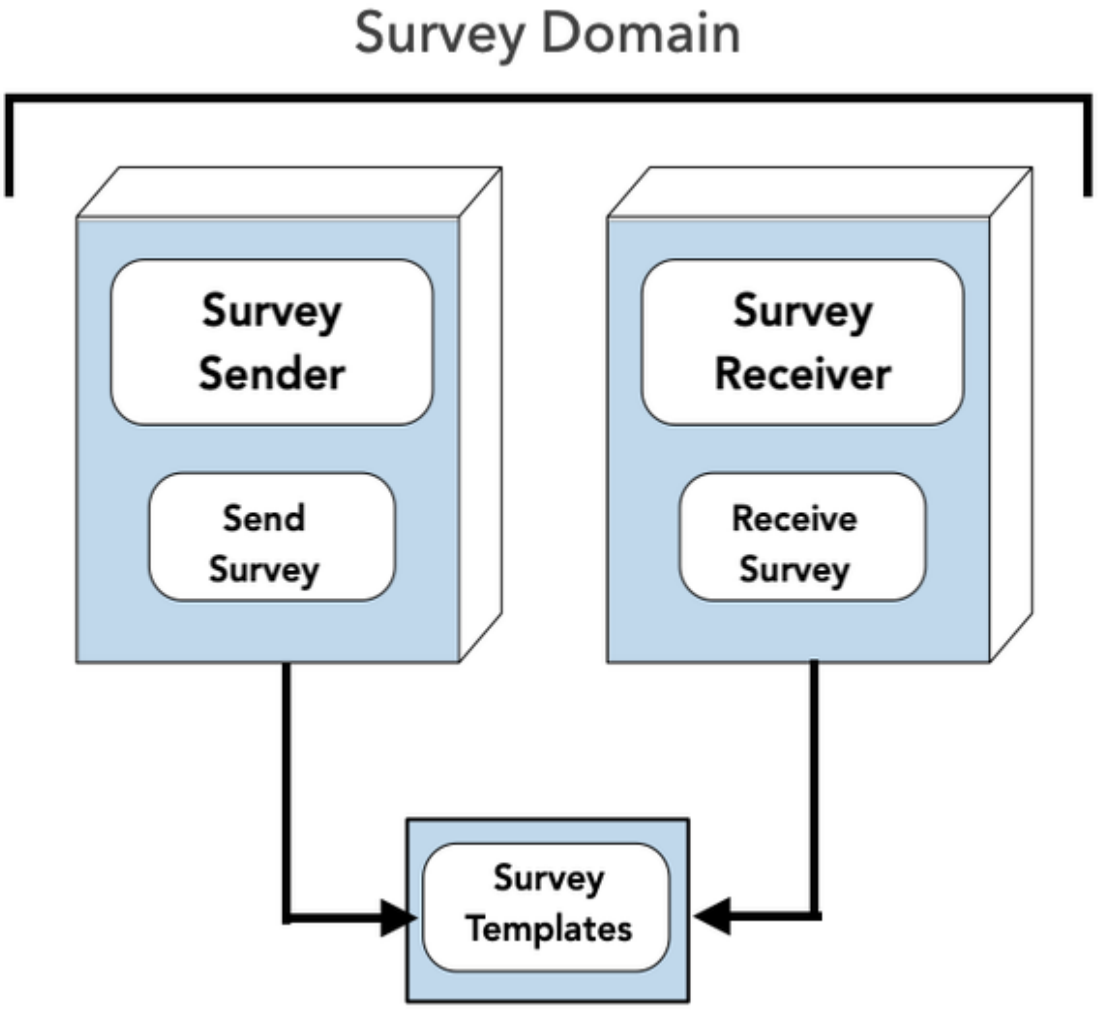


Figure 3-18. Microservices defined within the Survey domain

The directory structure for this architecture would look like Example 3-15, with the `templates` directory representing the shared library outside of

the services directories:

```
sysops
 └─ survey
     ├─ survey_sender_service
     │   └─ send
     ├─ survey_receiver_service
     │   └─ receive
     └─ templates
```

Example 3-15. ADL for describing the survey microservices of the Sysops Squad system

```
DESCRIPTION Describe Survey Microservices
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Survey AS survey
    DEFINE LIBRARY Survey Templates AS templates                    ❶
    DEFINE SERVICE Survey Sender as survey_sender_service
      DEFINE COMPONENT Send Survey AS send
    DEFINE SERVICE Survey Receiver as survey_receiver_service
      DEFINE COMPONENT Receive Survey AS receive

# Structural class-to-component assertion
ASSERT(CLASSES are only CONTAINED within COMPONENTS or LIBRARIES) ❷

# Dependencies between services
ASSERT(Survey Sender has NO DEPENDENCY ON other services)      ❸
ASSERT(Survey Receiver has NO DEPENDENCY ON other services)

# Library dependencies
ASSERT(Send Survey IS DEPENDENT ON Survey Templates)           ❹
ASSERT(Receive Survey IS DEPENDENT ON Survey Templates)
```

❶ The `DEFINE LIBRARY` directive denotes that this code should be used as an external shared library (such as a JAR or DLL file).

❷ This structural assertion here adds the `LIBRARIES` qualification to include the `Survey Templates` shared library, which is not a component.

❸ These two `ASSERT` statements reference the *service names* rather than the *component names*. These assertions define the service boundaries and include all of the components within that service. This means none of the components within a service can communicate with components

within another service. However, the assertions allow communication *outside* the service boundary (such as with the `Survey Templates` shared library).

❹ The `ASSERT` statements at the end of the ADL tell us which services use the shared library (in this case, both) and specify that there is a static source code dependency (such as a method call) between the components in the service and the shared library's source code.

Asking an AI agent to interpret this ADL produces the correct results in Figure 3-19 and the corresponding diagram in Figure 3-20.

Structure:

- **System:** Sysops Squad
  - **Domain:** Survey
    - **Library:** Survey Templates — shared code, not a deployable service
    - **Service:** Survey Sender → contains component *Send Survey*
    - **Service:** Survey Receiver → contains component *Receive Survey*

What the assertions tell us:

1. `CLASSES are only CONTAINED within COMPONENTS` — a structural constraint: no orphan classes floating at the service or domain level; everything must live inside a component.
2. Survey Sender and Survey Receiver have **no dependency on each other** — they are fully decoupled services, communicating (presumably) via some external channel (queue/event), not direct calls.
3. Both `Send Survey` and `Receive Survey` **depend on Survey Templates** — this is shared-library coupling, not service-to-service coupling. Both services need the library at compile/build time, but that doesn't couple them to *each other*.

Figure 3-19. AI agent interpretation of the ADL describing the Survey microservices and shared library

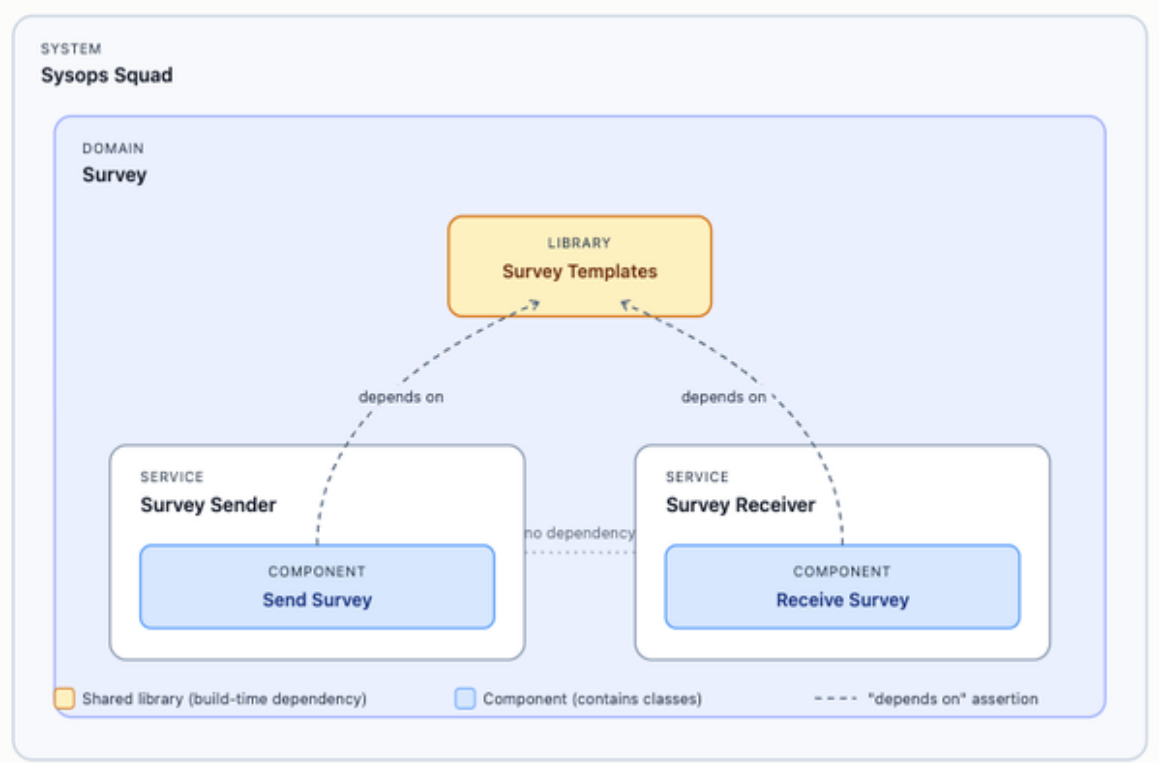


Figure 3-20. AI-generated diagram of the Survey microservices and shared library based on the ADL

WHERE SHOULD TEAMS STORE ADL FILES?

Teams usually place ADL files in the source code repository in a directory named *adl* that sits at a higher level, usually the same level as the system directory. ADL files typically have the suffix *.adl*. For example, the directory structure for the domain services defined in "Defining Domain Services" would look like this:

```
sysops
  ├─ adl
  │    ├─ domains.adl
  │    ├─ ticketing_domain.adl
  │    └─ survey_domain.adl
  └─ survey
       └─ ...
```

Architects can create directories under the `adl` directory to further categorize the ADL files, such as those describing the structure, data topology, service communication, and so on. Architects should make sure to provide a good descriptive name for the ADL files so that it makes it easy to identify what it's defining.

Teaching Agents About Architecture

There are many ways to use the ADL in this chapter, which describes the logical architecture from a domain, subdomain, and component perspective and the physical architecture from a service perspective. Some possible uses are:

- Documenting and describing the architecture through code in a unified, consistent fashion
- Governing the structure of the source code to align the system's implementation of the system with the intended architecture (see the next chapter)

- Teaching AI agents about the architecture

This section focuses on the last point. ADL is concise and consistent and communicates well to both agents and human developers, making it a better option than describing the desired architecture using lengthy prose and thousands of tokens.

To demonstrate how to use ADL to teach an agent about the architectural structure of a system, we'll stay with the Sysops Squad example. This time, however, we'll limit our scope to customer behavior and core ticketing behavior, as the diagram in Figure 3-21 illustrates.

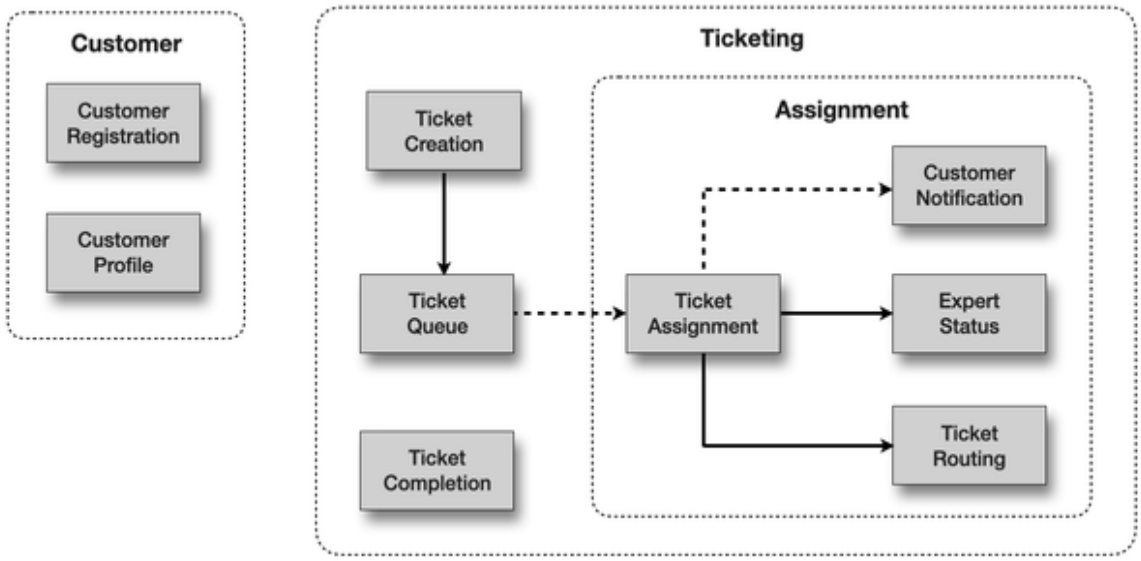


Figure 3-21. The logical architecture for the customer and ticketing domains

Example 3-16 is an excerpt from the Markdown file in the Sysops Squad Monolithic Repository. This prompt is for generating the customer and ticketing behavior *without* using ADL to describe the architecture.[2]

Example 3-16. AI agent prompt for generating the customer and ticketing portions of the Sysops Squad system without using ADL

```
## Functional requirements

1. Customer Registration / Profile Update
   - Fields: Customer ID (short generated UUID), Name, City and
State
     (location), Email.
   - Support many customers and many tickets.
```

```
2. Create Ticket
   - Input: Customer ID, problem description.
   - Place the ticket on a work queue tagged with the customer's
location.
3. Assignment
   - Match an available expert to the ticket by location.
   - Seed one or two experts in the database.
4. Notification
   - Email the customer that the system has assigned an expert.
5. Ticket Routing
   - "Upload" the ticket to the expert's mobile device. Stub this
by emailing
     the expert at expert1@sysopssquad.com (Expert ID 1).
6. Expert Status
   - Update the expert's status in the database: assigned or
available.
7. Ticket Completion
   - Mark the ticket complete in the database.
```

When we instructed the AI agent to generate the Sysops Squad system using this prompt excerpt, it created the directory structure in Example 3-17 (which is also in the Sysops Squad repository).

Example 3-17. Directory structure for the customer and ticketing behavior of the Sysops Squad system generated by an AI agent without using ADL

```
sysops
 ├─ assignment
 ├─ common
 ├─ config
 ├─ customer
 ├─ expert
 ├─ notification
 ├─ routing
 ├─ SysopsApplication.java
 └─ ticket
```

This generated directory structure closely matches an antipattern in our *Fundamentals* book, for which we coined the name Entity Trap. The Entity Trap happens when teams (or agents) create components by considering only the entities of the system. Furthermore, the `SysopsApplication.java` class which is directly under the system root directory, is an orphan class: it doesn't have a corresponding

component home. Not only is this generated structure a mess, it doesn't even come close to the architectural diagram in Figure 3-21.

Using ADL, the architect can teach the agent about the intended architectural structure and have it produce a source-code structure to match that architecture. Example 3-18 shows high-level ADL that describes the domains using the logical architecture diagram for the customer and ticketing portions of the Sysops Squad System.

Example 3-18. ADL for describing the customer and ticketing domains

```
FILENAME: sysops_domains_primary.adl
DESCRIPTION Define Domains
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
  DEFINE DOMAIN Customer AS customer
ASSERT(CLASSES are only CONTAINED within SUBDOMAINS within DOMAINS)
```

Example 3-19 shows ADL for the `Customer` domain that matches the intended logical architecture.

Example 3-19. ADL for describing the Customer domain

```
FILENAME: sysops_customer_domain.adl
DESCRIPTION Define Customer Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Customer AS customer
    DEFINE COMPONENT Customer Registration AS registration
    DEFINE COMPONENT Customer Profile AS profile

ASSERT(CLASSES are only CONTAINED within COMPONENTS)

ASSERT(Customer Registration HAS NO DEPENDENCY ON other components)
ASSERT(Customer Profile HAS NO DEPENDENCY ON other components)
```

Finally, Example 3-20 shows the ADL for the `Ticketing` domain, which also matches the intended logical architecture.

Example 3-20. ADL for describing the Ticketing domain

```
FILENAME: sysops_ticketing_domain.adl
DESCRIPTION Define Ticketing Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
```

```
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE COMPONENT Ticket Creation AS creation
    DEFINE COMPONENT Ticket Queue AS ticket_queue
    DEFINE COMPONENT Ticket Completion AS completion
    DEFINE SUBDOMAIN Assignment AS assignment
      DEFINE COMPONENT Ticket Assignment AS assign_expert
      DEFINE COMPONENT Customer Notification AS
customer_notification
      DEFINE COMPONENT Ticket Routing AS routing
      DEFINE COMPONENT Expert Status AS expert_status

ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Allowed dependencies
ASSERT(Ticket Creation IS DEPENDENT ON Ticket Queue)
ASSERT(Ticket Assignment IS DEPENDENT ON Ticket Routing and Expert
Status)

# Disallowed dependencies
ASSERT(Ticket Completion HAS NO DEPENDENCY ON other components)
ASSERT(Ticket Queue HAS NO DEPENDENCY ON other components)
ASSERT(Customer Notification HAS NO DEPENDENCY ON other components)
ASSERT(Ticket Routing HAS NO DEPENDENCY ON other components)
ASSERT(Expert Status HAS NO DEPENDENCY ON other components)
```

When we provide the same prompt files as before but this time *include* the ADL files, the same AI agent produces the same Sysops Squad system matching the same behavior, but now with the directory structure in Example 3-21. This structure matches the ADL file and the logical architecture from Figure 3-21.

Example 3-21. Directory structure for Customer and Ticketing behavior generated by an AI agent using ADL

```
sysops
 ├─ customer
 │   ├─ profile
 │   └─ registration
 └─ ticketing
     ├─ assignment
     │   ├─ assign_expert
     │   ├─ customer_notification
     │   ├─ expert_status
     │   └─ routing
     ├─ completion
```

```
    ├─ creation
    └─ ticket_queue
```

Specifying Implementation Details in ADL

The ADL files we've shown so far in this chapter haven't included any implementation details, mostly because—by design—ADL describes the architecture of a system independent of its technical implementation details (which we highly encourage). That said, teams may sometimes want to provide some implementation details (such as the desired programming language) in ADL rather than a verbose text-based prompt, such as when using ADL in conjunction with agentic code generation. To specify an implementation detail, they can add a `USING` directive to the `DEFINE` verb in the ADL.

Example 3-22 shows an ADL example of a polyglot system where the `Ticketing` service uses Java and the `Survey` service uses Python. This ADL leverages the `USING` directive to specify the implementation details.

Example 3-22. ADL snippet for describing domain services system with the USING directive

```
DESCRIPTION Describe Domain Services
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE SERVICE Ticketing Service as ticketing_service USING
Java and Spring
        ...
  DEFINE DOMAIN Survey AS survey
    DEFINE SERVICE Survey Service as survey_service USING Python
    ...
```

Architects should be aware that when adding a `USING` verb to the definition of an artifact, they are leaking implementation details into the architectural structure of a system. If those implementation details change, so must the corresponding ADL. For this reason, we caution against prolific use of the `USING` verb unless architects have a good reason for doing so (such as agentic code generation).

Reducing Friction

In this chapter, we've demonstrated that, for a system's implementation to align with its architecture, the *structure* of its source code (directories, package names, namespaces) must align with the domains, subdomains, and components in the system's logical or physical architecture. Predictably, this tends to create friction between architects and development teams.

In the old way of working, software architecture and software development were two separate activities. Software architects identified the system's critical architectural characteristics, created a logical architecture, chose the most appropriate architectural style, and then identified the system's physical architecture and (if appropriate) services. Development teams would then take these artifacts and implement the architecture through class diagrams, UI design, and, of course, source code.

However, to describe and govern the system's implementation and align it with the architecture, the source code's structure must match the structure and the `AS` directives of the corresponding ADL files describing the architecture. This necessary alignment between ADL and the organization of the directory or namespace structure tends to pull software architects into the realm of implementation. Many developers see this as intrusive, and architects as "stealing" the art of programming away from them.

One way to mitigate this friction is for the architect to communicate with the development team, explaining *why* name alignment is necessary and providing concrete goals. Usually, the development teams name the directories or namespaces, providing they match the intended source-code structure. Then the architect writes the ADL to match those names.

We also recommend explaining to development teams that the ADL and corresponding executable tests are't intended to control their actions, just to provide a *placeholder* for fast feedback when the implementation is misaligned with the designated architecture. In other words, as we'll show in the next chapter, just because a particular governance test fails doesn't mean the development team did something wrong.

[1] See Chapter 14 of *Fundamentals of Software Architecture*, 2nd Edition, for more about this architectural style.

[2] See [Link to Come] for how to access and download the Sysops Squad Git repositories.

Chapter 4. Governing Architecture Using ADL and AI Agents

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

In the prior chapter on "Describing Architecture Using ADL" we demonstrated how to describe and define a logical and physical architecture using ADL. In this chapter, we show how to use that same ADL to generate executable source code to validate that the implementation of the system matches the intended architecture.

From a structural standpoint, architects can leverage existing architectural analysis tools and frameworks that can verify architectural constraints based on the structure of the source code. In the past, architects were required to write executable verification tests—architectural fitness functions—by hand to govern certain constraints in the system. However, with the use of ADL and agentic AI, architects can have an agent generate these executable tests.

Architects can use the following tools and frameworks to generate executable tests to verify that the implementation is aligned with the

intended architecture:

ArchUnit (see https://www.archunit.org/)

> ArchUnit is an open source framework written in Java that can govern the structure of a system's architecture. It provides a well-defined and intuitive API that architects can use to write or generate Java tests to govern structural and constraint aspects of an architecture. It's API is perhaps the richest and most mature in the list.

ArchUnitNet (see https://github.com/TNG/ArchUnitNET)

> A C# architecture test library to specify and assert architecture rules in C# for automated testing that is a port from ArchUnit.

ArchUnitTS (see https://github.com/LukasNiessen/ArchUnitTS)

> ArchUnitTS is an architecture testing library used to specify and ensure architecture rules in your TypeScript app. Easy setup and pipeline integration. ArchUnitTS is a port from ArchUnit.

NetArchTest (see https://github.com/BenMorris/NetArchTest)

> A fluent API for .NET that can enforce architectural rules in unit tests for C# applications. NetArchTest is generally more commonly used that ArchUnitNet due to its maturity, plethora of examples, and large corpus of code out in the internet that agents can leverage.

PyTestArch (see https://pypi.org/project/pytestarch/)

> PyTestArch is an open source library that allows users to define architectural rules and test their code against them in Python applications. Although not a direct port from ArchUnit, many of its libraries are inspired by ArchUnit.

GoArchTest (see https://github.com/solrac97gr/goarchtest)

> GoArchTest is a Go library for testing the architectural constraints of your Go applications. It is inspired by the .NET library NetArchTest,

providing a fluent API to define and enforce architectural rules in your Go projects.

As illustrated in Figure 4-1, to leverage these tools and frameworks using ADL, architects first create a logical architecture diagram. From there they create the corresponding ADL files describing the architecture in terms of domains, subdomains, components and their dependencies, and finally services and their dependencies (if creating a distributed architecture). Architects then use an AI agent to generate executable tests in a variety of languages using the tools and frameworks previously listed.

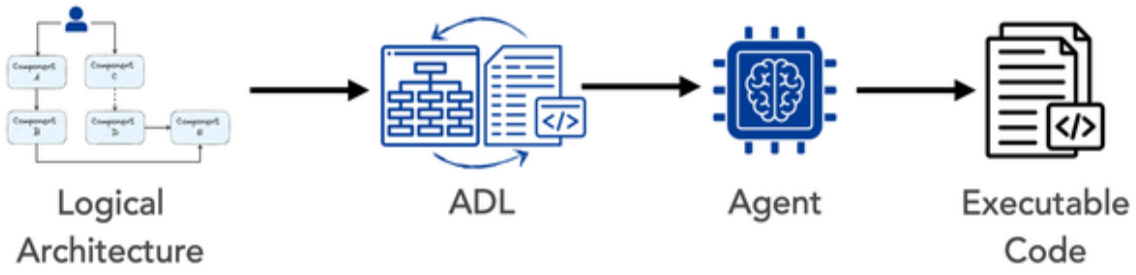


Figure 4-1. The steps an architect takes to generate executable fitness functions to govern the architecture

As we explained in the previous chapter, the first part of the ADL defines the structure of the system in terms domains, subdomains, and components, and the last part uses the `ASSERT` directive to describe constraints regarding the structure. The following sections show how ADL moves from documentation to governance by describing how architects can use ADL along with AI agents and the tools and frameworks previously listed to govern the source code structure and validate that the implementation is continuously aligned with the intended architecture.

Governing Domains

The `ASSERT` directives in the ADL specify certain conditions and constraints based on the artifacts defined in the ADL. For example, using the ADL from the prior chapter, architects can define the domains of the system (in this case the `Ticketing` and `Survey` domains of the Sysops

System), applying the constraint that all code must reside somewhere under these domains as shown in Example 4-1.

Example 4-1. ADL for describing the domains in Sysops Squad

```
DESCRIPTION Define Domains
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
  DEFINE DOMAIN Survey AS survey

ASSERT(CLASSES are only CONTAINED within SUBDOMAINS within DOMAINS)
```

The `ASSERT` part of the ADL specifies the test condition and constraint of the architecture. In this case, the ADL is asserting that all classes must exist somewhere under these two directory structures (which may include subdirectories representing subdomains), but nowhere else.

Architects can pass this ADL into an AI agent and prompt it to generate executable source code using one of the tools and frameworks listed earlier. For example, prompting it to generate an ArchUnit test in Java would yield the following code (abbreviated for brevity):

```
...
String BASE_PACKAGE = "sysops";
String TICKETING_PACKAGE = BASE_PACKAGE + ".ticketing.."; ❶
String SURVEY_PACKAGE = BASE_PACKAGE + ".survey..";

ArchRule domainContainmentRule = ArchRuleDefinition.classes()
   .that().resideInAPackage(BASE_PACKAGE + "..")
   .should().resideInAnyPackage(TICKETING_PACKAGE,
SURVEY_PACKAGE);

domainContainmentRule.check(importedClasses);
...
```

❶ The two dots after `.ticketing` mean that code can reside in this directory *and* any other directories that happen to be below it, therefore including further subdomains and components. It's the `within SUBDOMAINS` part of the `ASSERT` statement in the ADL that does this. Without it, an agent would not include the two dots after the directory

name, meaning the test would fail because it would restrict code from being in any other directory except the domain-level one.

An architect now has executable source code they can use that describes the architecture *and* governs it at the same time. They can execute the code through a variety of means described in "Structural Governance Models" to ensure the implementation always matches the architecture. This way, if a developer or agent happens to create another high-level directory, this test would fail, providing a placeholder for discussion between the team and the architect (see "Iterating on the Architecture" for an example of this scenario).

Governing The Components Within a Domain

In "Defining Domain Details" we showed how to use ADL to describe the entire domain, including all subdomains and components within that domain, using the ADL shown in Example 4-2.

Example 4-2. ADL for describing the ticketing domain in Sysops Squad

```
DESCRIPTION Describe Ticketing Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
  DEFINE DOMAIN Ticketing AS ticketing
    DEFINE COMPONENT Ticket Creation AS creation
    DEFINE COMPONENT Ticket Completion AS completion
    DEFINE SUBDOMAIN Assignment AS assignment
      DEFINE COMPONENT Ticket Assignment AS assign_expert
      DEFINE COMPONENT Ticket Routing AS routing
      DEFINE COMPONENT Customer Notification AS
customer_notification

ASSERT(CLASSES are only CONTAINED within COMPONENTS)
```

Notice here the difference between this `ASSERT` statement and the prior ADL describing the domains. Here, it's only necessary to constraint source files to `COMPONENTS` since the domain directory and subdomain directories can't have any source code directly under them. Doing so would

create orphan classes—classes that don't belong to any component (see the *Flatten Components* pattern in "Using ADL for Fitness Function Driven Architecture").

An architect can prompt an AI agent to use this ADL pseudocode to generate executable source code using whatever framework and language the system is implemented in. For example, prompting an agent to generate ArchUnit code in Java using the ADL shown in Example 4-2 would produce following executable source code (for brevity, only the relevant portions of the generated code are shown):

```
...
private static final String BASE_PACKAGE = "sysops.ticketing";

ArchRule rule = ArchRuleDefinition.classes()
    .that().resideInAPackage(BASE_PACKAGE + "..")
    .should().resideInAnyPackage(
        BASE_PACKAGE + ".creation..",
        BASE_PACKAGE + ".ticket_queue..",
        BASE_PACKAGE + ".completion..",
        BASE_PACKAGE + ".assignment.assign_expert..",
        BASE_PACKAGE + ".assignment.routing..",
        BASE_PACKAGE + ".assignment.customer_notification..");

rule.check(importedClasses);
...
```

The same prompt and ADL could be used to generate NetArchtest code in C# as well:

```
...
private const string TicketingDomainNamespace =
    "Sysops.Ticketing";
private const string CreationComponentNamespace =
    "Sysops.Ticketing.Creation";
private const string CompletionComponentNamespace =
    "Sysops.Ticketing.Completion";
private const string AssignExpertComponentNamespace =
    "Sysops.Ticketing.Assignment.AssignExpert";
private const string RoutingComponentNamespace =
    "Sysops.Ticketing.Assignment.Routing";
```

```
private const string CustomerNotificationComponentNamespace =
    "Sysops.Ticketing.Assignment.CustomerNotification";

var result = Types.InAssembly(Assembly).That()
    .ResideInNamespace(TicketingDomainNamespace)
        .Should().ResideInNamespace(CreationComponentNamespace)
        .Or().ResideInNamespace(CompletionComponentNamespace)
    .Or().ResideInNamespace(AssignExpertComponentNamespace)
    .Or().ResideInNamespace(RoutingComponentNamespace)

.Or().ResideInNamespace(CustomerNotificationComponentNamespace)
    .GetResult();
...
```

Regardless of the fitness function testing framework or language you use, the generated code governs the `ASSERT` directive in the ADL, meaning if a developer added another directory or namespace under the `Ticketing` domain, the test would fail, triggering fast feedback that the implementation no longer matches the intended architecture.

Governing Dependencies

Architects can use the same techniques as in the prior section to govern the component and service dependencies described in ADL. These fitness functions help prevent what is known as a "big ball of mud" architecture, where components or services have so many haphazard dependencies that is starts to resemble an unstructured mess, similar to a ball of mud with straw all throughout it.

Referring back to the Sysops Squad logical architecture diagram shown in Figure 3-1 from Chapter 3, it defines the `Survey` domain as having the following component dependencies:

- The `Send Survey` component communicates directly with the `Survey Templates` component to know which type of survey to send based on the product the customer had an issue with

- The `Receive Survey` component communicates directly with the `Survey Templates` component to know how to parse the survey received from the customer
- No other component dependencies are allowed within the `Survey` domain

The corresponding ADL explicitly describing these component dependencies is shown in Example 4-3.

Example 4-3. ADL for describing the survey domain in Sysops Squad by explicitly specifying the component dependencies

```
DESCRIPTION Describe Survey Domain
CATEGORY Structural
DEFINE SYSTEM Sysops Squad AS sysops
DEFINE DOMAIN Survey AS survey
  DEFINE COMPONENT Send Survey AS send
  DEFINE COMPONENT Receive Survey AS receive
  DEFINE COMPONENT Survey Templates AS templates

# Structural assertions
ASSERT(CLASSES are only CONTAINED within COMPONENTS)

# Allowed dependency assertions
ASSERT(Send Survey IS DEPENDENT ON Survey Templates)
ASSERT(Receive Survey IS DEPENDENT ON Survey Templates)

# Disallowed dependency assertions
ASSERT(Send Survey HAS NO DEPENDENCY ON Receive Survey)
ASSERT(Receive Survey HAS NO DEPENDENCY ON Send Survey)
ASSERT(Survey Templates HAS NO DEPENDENCY ON Send Survey or Receive
Survey)
```

The key here is to make sure that developers and agents understand what *is* required, but what also *isn't* allowed, hence the two separate assertion sections in the ADL file. As we demonstrated in the prior chapter, both of these assertion sections are required to correctly interpret the architecture. They're also both required to have an agent generate the correct corresponding executable fitness functions enforcing these dependencies.

Generating (or writing) the fitness functions for component and service dependencies can sometimes get a little tricky depending on the tools and frameworks architects (or agents) use to generate the code. The difficultly lies in the allowed dependencies assertions:

```
# Allowed dependency assertions
ASSERT(Send Survey IS DEPENDENT ON Survey Templates)
ASSERT(Receive Survey IS DEPENDENT ON Survey Templates)
```

For example, using ArchUnit in Java or ArchUnitNet in C#), architects (and agents) are usually temped to generate the following code to describe these component relationships:

```
...
private static final String SEND_COMPONENT_PKG =
"..survey.send..";
private static final String RECEIVE_COMPONENT_PKG =
"..survey.receive..";
private static final String TEMPLATES_COMPONENT_PKG =
"..survey.templates..";

ArchRule sendDependsOnTemplates =
  classes().that().resideInAPackage(SEND_COMPONENT_PKG)
           .should().dependOnClassesThat()
           .resideInAPackage(TEMPLATES_COMPONENT_PKG);

ArchRule receiveDependOnTemplates =
  classes().that().resideInAPackage(RECEIVE_COMPONENT_PKG)
           .should().dependOnClassesThat()
           .resideInAPackage(TEMPLATES_COMPONENT_PKG);

sendDependsOnTemplates.check(imported);
receiveDependOnTemplates.check(imported);
...
```

The problem with this code is that it's evaluated per class, requiring *every class* in the send or receive packages to depend on templates, not just "at least one class". This is okay as long as each component maps to a single

class, or *all* classes in a component genuinely need the templates dependency. However, it will fail if any of the classes within the send or receive packages *don't* access the classes in the template package.

Architects might not catch this, but fortunately some AI agents do (such as Claude Code Sonnet). For example, the AI agent realized this all-or-nothing dependency problem (without additional prompting) and generated the following helper class to ensure *at least one class* semantics:

```
...
// "IS DEPENDENT ON" is an existence claim (at least one
// class depends on the target), so it's expressed as a
// dependency search rather than a per-class ArchRule.
private boolean hasDependencyOn(String fromPackageIdentifier,
                                String toPackageIdentifier) {
    PackageMatcher from =
PackageMatcher.of(fromPackageIdentifier);
    PackageMatcher to = PackageMatcher.of(toPackageIdentifier);

    return classes.stream()
        .filter(javaClass ->
from.matches(javaClass.getPackageName()))
        .flatMap(javaClass ->
            javaClass.getDirectDependenciesFromSelf().stream())
        .anyMatch(dependency ->

to.matches(dependency.getTargetClass().getPackageName()));
}
...
```

With this `hasDependencyOn` helper method, the generated code now works correctly, producing the following dependency methods for the allowed dependency assertions in the prior ADL:

```
...
private static final String SEND_COMPONENT_PKG =
"..survey.send..";
private static final String RECEIVE_COMPONENT_PKG =
"..survey.receive..";
private static final String TEMPLATES_COMPONENT_PKG =
"..survey.templates..";
```

```
void send_survey_is_dependent_on_survey_templates() {
    assertThat(hasDependencyOn(SEND_COMPONENT_PKG,
                                    TEMPLATES_COMPONENT_PKG))
        .isTrue();
}

void receive_survey_is_dependent_on_survey_templates() {
    assertThat(hasDependencyOn(RECEIVE_COMPONENT_PKG,
                                    TEMPLATES_COMPONENT_PKG))
        .isTrue();
}
...
```

For the disallowed dependency assertions, the AI agent produced the following source code:

```
...
void send_survey_has_no_dependency_on_receive_survey() {
    ArchRule rule = noClasses()
        .that().resideInAPackage(SEND_COMPONENT_PKG)
        .should().dependOnClassesThat()
                .resideInAPackage(RECEIVE_COMPONENT_PKG);
    rule.check(classes);
}

void receive_survey_has_no_dependency_on_send_survey() {
    ArchRule rule = noClasses()
        .that().resideInAPackage(RECEIVE_COMPONENT_PKG)
        .should().dependOnClassesThat()
                .resideInAPackage(SEND_COMPONENT_PKG);
    rule.check(classes);
}

void
survey_templates_has_no_dependency_on_send_or_receive_survey() {
    ArchRule rule = noClasses()
        .that().resideInAPackage(TEMPLATES_COMPONENT_PKG)
        .should().dependOnClassesThat()
                .resideInAnyPackage(SEND_COMPONENT_PKG,
                                    RECEIVE_COMPONENT_PKG);
    rule.check(classes);
}
...
```

As always, architects should take the interpolated executable fitness function source code generated from an AI agent and manually test both positive and negative conditions to ensure the code works as intended before making it an empirical fitness function or empirical MCP tool.

Structural Governance Models

One question that an architect will likely encounter is “Where do all of these generated executable empirical fitness functions live?”. Architects can answer this question through three general governance models to ensure the implementation of a system matches the architecture as specified by the ADL.

The executable fitness functions generated by an AI agent can be executed in a variety of ways, including using MCP tools hosted through a custom MCP server with AI agents, executing them through a CI/CD pipeline as standalone triggered fitness functions, or through interactive AI agent conversations. The following sections describe each of these models.

Hosted as MCP Tools Using Agents

A common practice among architects is to create a set of MCP tools from the ADL-based generated source code, and host them on a custom MCP server specific to the system context. Architects can then have an AI agent access those MCP tools to execute the generated fitness functions. Architects can either make a request to the agent as an on-demand query, or have the agent periodically execute the MCP tool fitness functions automatically and alert the architect if the tests fail. This governance model is illustrated in Figure 4-2.

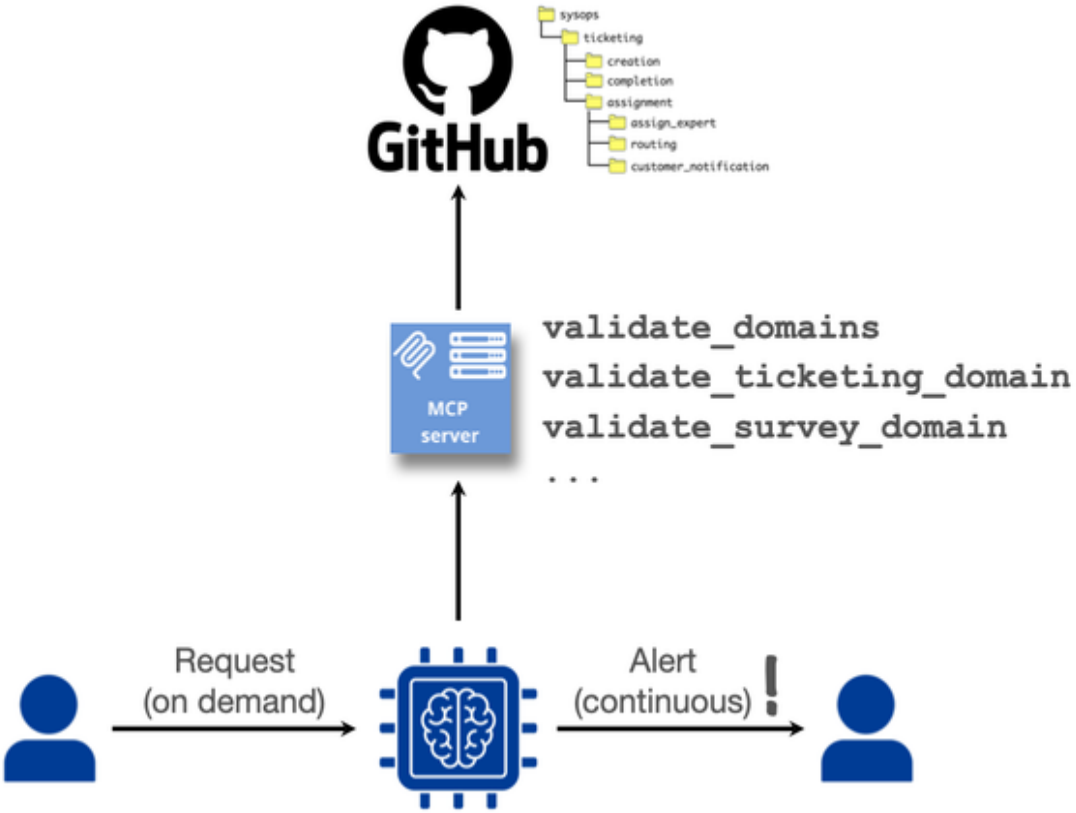


Figure 4-2. Executing empirical fitness functions through an AI agent via MCP tools

When using domain-based repositories in a distributed architecture, each domain would typically contain its own separate MCP server hosting the corresponding executable fitness functions for that domain. Alternatively, architects can use a single consolidated MCP server and set of MCP tools that point to the specific repository to test the entirety of the system as illustrated in Figure 4-3.

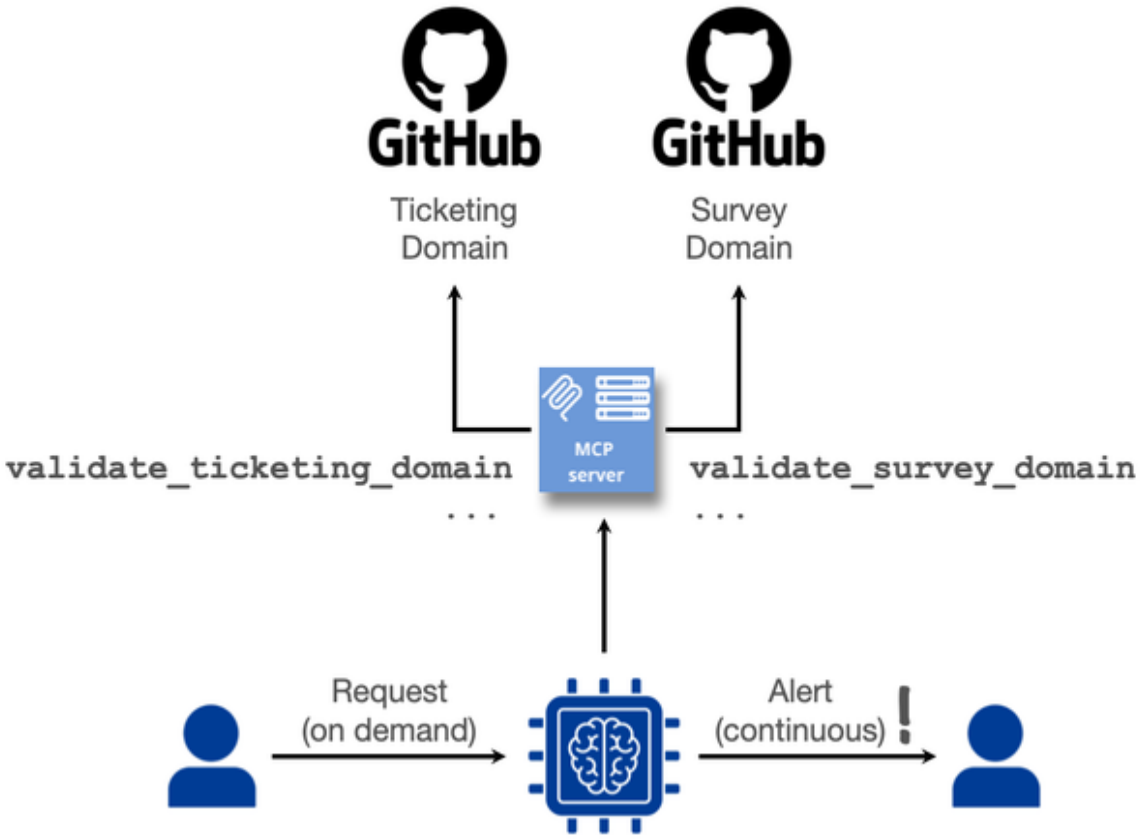


Figure 4-3. Executing empirical fitness functions in separate domains through an AI agent via MCP tools using a consolidated MCP server

Regardless of the approach architects use, the governance model remains the same: an AI agent accesses empirical fitness functions generated from the ADL to validate that the implementation matches the architecture.

Standalone Triggered Fitness Functions

Assuming architects have validated and tested the generated fitness functions by an agent based on the ADL—and thereby creating empirical fitness functions—they can avoid the use of an AI agent altogether and use the CI/CD pipeline as an MCP host to access the same MCP tools described in the prior section. This model utilizes *triggered* fitness functions—architectural tests run against the source code structure that are executed based on some event (such as a software release)—as illustrated in Figure 4-4.

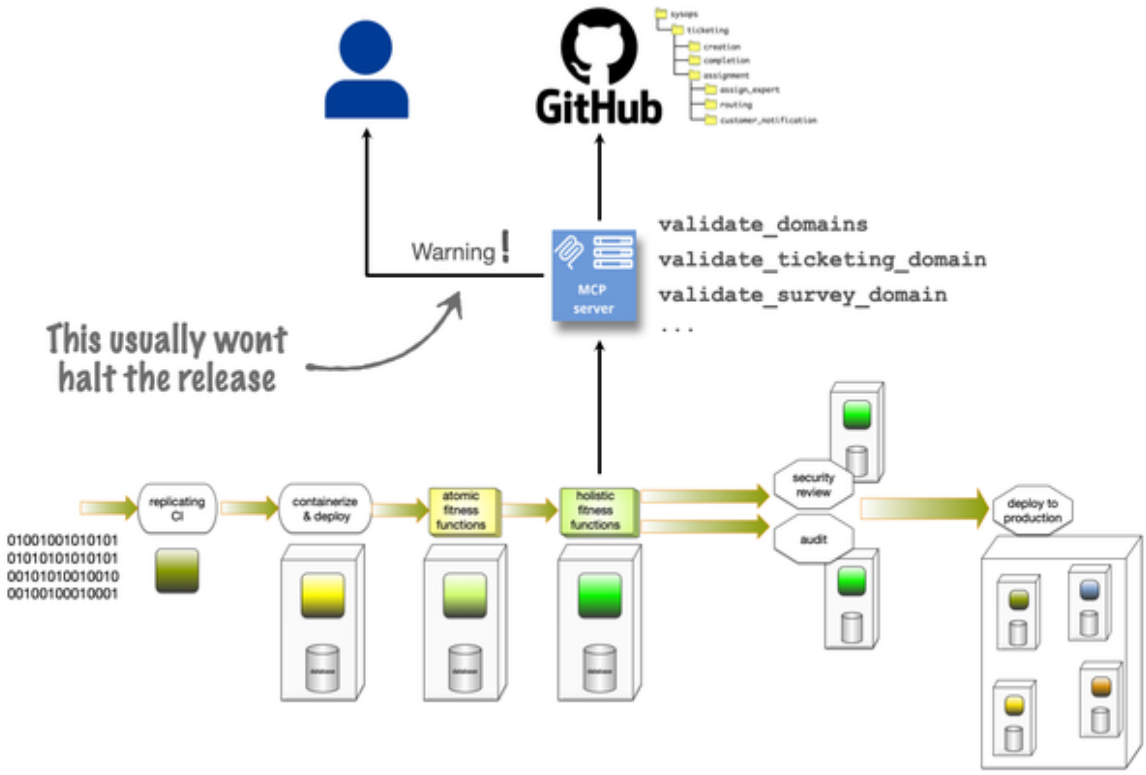


Figure 4-4. Executing triggered empirical fitness functions implemented as MCP tools through a CI/CD pipeline

In this model it's not necessary to host the executable fitness functions in an MCP server as MCP tools. Instead, the individual generated executable tests can be run as standalone executables or executed as part of a deployment pipeline. Depending on the circumstance, we usually recommend hosting them through an MCP server to provide universal access across all the governance models.

Interactive AI Agent Conversations

This governance model is quite different from the others in that it doesn't leverage generated empirical fitness functions, but rather relies on an AI agent to directly read the ADL and perform whatever analysis it chooses to compare the ADL to the structure of the source code in the repository as illustrated in Figure 4-5.

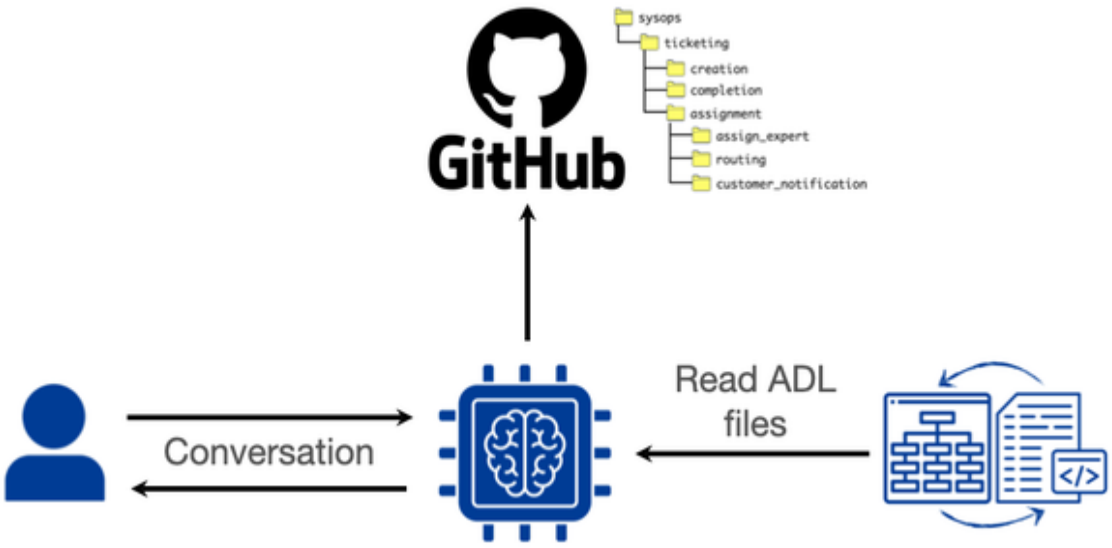


Figure 4-5. Agentic interactive verification of the source code structure based on ADL

We call these type of fitness function tests *interactive* fitness functions as opposed to *empirical* ones because the results may be unpredictable (see [Link to Come] in [Link to Come] for a full definition of these fitness function types). Through practical use and experimentation we have observed that the AI agents we used for this book (Claude Code Sonnet and Fable) both ended up producing dynamic source code using one of the tools and frameworks listed at the start of this chapter based on the programming language the system was implemented in to perform its analysis. While not required, it is a preferred approach.

This model is widely used when using ADL to teach agents about an architecture during source code generation (see "Teaching Agents About Architecture" in Chapter 3). While we have observed good success with this approach, we highly recommend also leveraging one of the other empirical-based governance models to verify the source code structure afterwards, particularly since human developers or agents might change the code afterwards based on new features, consequently misaligning the implementation with the architecture described in the ADL. The next section on "Iterating on the Architecture" describes this real-world scenario.

Iterating on the Architecture

A system's architecture always changes eventually, whether it's a greenfield (new) or brownfield (existing) system undergoing modifications. How many architects can guarantee that the implementation of the system exactly matches the architecture diagrams they worked so hard to produce? Unfortunately, not many. This is where Architecture as Code and ADL can help architects keep the architecture and implementation constantly aligned.

Consider the following existing directory structure for the `Ticketing` domain in the Sysops Squad system, which is currently aligned with the logical architecture, ADL, and generated executable fitness functions:

```
sysops
└─ ticketing
    ├─ creation
    ├─ completion
    └─ assignment
        ├─ assign_expert
        ├─ routing
        └─ customer_notification
```

Suppose the development team needs to add behavior to this system to modify or cancel an existing trouble ticket. The team decides they need a new directory for this additional behavior, so they create one under the `ticket` directory named `modify_ticket` as follows:

```
sysops
└─ ticketing
    ├─ creation
    ├─ completion
    ├─ modify_ticket  ❶
    └─ assignment
        ├─ assign_expert
        ├─ routing
        └─ customer_notification
```

❶ This new directory essentially represents a new component in the architecture.

Whichever governance model the architect chooses to use, at some point (either triggered during release, periodic automated check, or interactive agentic conversation), the architect will be altered that the implementation no longer matches the architecture due to the added `modify_ticket` component. And this is *exactly* what architecture as code is all about—*fast feedback*.

The executable fitness functions derived from the ADL we described in this chapter are not meant to be like traditional unit tests. When a unit test fails, it indicates a problem with the source code—a bug in the system—and developers (hopefully) work to fix the problem before the software can be released.

However, a failed structural fitness function test doesn't necessarily mean something is wrong. On the contrary, all it indicates is that the implementation no longer matches the architecture, prompting the architect to make sure the change is valid and not affecting anything architecturally. If it's a valid change, the architect would then realign the other architectural artifacts, including diagrams, ADL, and executable fitness functions to match the new implementation.

A failed test derived from the ADL can also act as a placeholder when further conversation is needed. For example, if the development team creates a new component, the architect may consult with them to make sure that's the most appropriate action, such as instead adding the new behavior to an existing component. This helps foster a collaborative environment where the architect and development team are working together to iterate on the architecture and communicating more frequently.

Failed fitness functions also prompt the architect to modify the architecture diagrams, corresponding ADL, and associated executable fitness functions, ensuring they all stay in alignment with each another. Architects often delegate the work of modifying the architecture diagrams, ADL, and corresponding executable fitness functions to a lead developer ir tech lead on the team. The architect has to trust the lead developer to make the right choices, and confer with the architect for advice and direction if they have a

concern or are unsure about a particular modification. This can be a win-win approach; not only does it free the architect from having to continually modifying the architectural artifacts, but more importantly, it gives the development team a better perspective on the underlying architecture of the system, allowing them to create a more effective software system.

REGENERATE THE SOURCE CODE OR MODIFY IT?

When modifying an ADL file to add or remove components, the architect can use an AI agent to regenerate the corresponding executable source code or manually modify the previously generated source code themselves. Depending on the tool the architect uses (ArchUnit, PyTestArch, NetArchTest, and so on), the newly generated source code will sometimes need modifications (believe it or not, generative AI is still not perfect at generating source code, albeit it's getting better). Therefore, unless the change is a massive architectural restructuring, we usually recommend manually modifying the existing executable source code when modifying the ADL file so as to reduce unpredictability during fitness function source code generation.

Using ADL for Fitness Function Driven Architecture

Architects that find themselves with an implementation that's significantly misaligned with the architecture can realign it by leverage ADL and corresponding generated tests using a technique called *Fitness-function-driven architecture* (FFDA). With FFDA, architects start by using ADL to define the domains of the system, further defining the details of the most critical domains using ADL as well. Once an architect has the ADL in place, they can use the techniques described in this chapter to create generated, executable fitness functions that describe the desired architecture.

This technique is similar to *Test-Driven Development* (TDD), where teams first write unit tests (which initially fail), then develop source code to make the tests pass. The executable fitness functions generated from the ADL will initially fail, just like unit tests in TDD. As development teams start refactoring the source code to match the intended architecture (for example, by defining the intended domains in the system), the executable fitness functions will start to pass, providing the architect with metrics in terms of how close they are to completing the migration and relaignment.

In our book *Software Architecture: The Hard Parts* (O'Reilly, 2021), we introduced refactoring patterns in Chapter 5 to help architects and developers identify and refactor a system's domains, subdomains, and components. Here's a brief description of each pattern:

Identify and Size Components

> In this pattern, teams identify what the current implementation reveals in terms of components, and the relative sizes of those components. Here, teams work to break up larger components into smaller, more manageable sizes to help facilitate further source code refactorings.

Gather Common Domain Components

> Here teams work to locate and gather common behavior spread throughout the source code. Examples here include shared infrastructure code such as logging, auditing, and metrics gathering, and also shared behavior such as common formatting routines, string and data manipulation routines, and so on. The goal here is to gather and isolate this common behavior in shared components, turning *implicit* class-based coupling into *explicit* component-based coupling.

Flatten Components

> Perhaps the most important of the refactoring patterns, this pattern describes pushing *all* source code down to the leaf nodes of the directory structure or namespace. Doing so creates definable components, with all source code contained within a component—since

components are the building blocks of the architecture. From there, the architect can begin grouping components into subdomains, and then components and subdomains into domains, matching those components with the ones defined in the ADL reflecting the desired architecture.

Determine Component Dependencies

In this pattern teams work to identify dependencies between *components*. A component dependency is formed when a class from one directory or namespace accesses a class in a different directory or namespace. Once teams identify component dependencies, they then begin to untangle those dependencies to match the ADL assertions regarding which components can communicate with others.

Create Component Domains

Here, teams first define the domains as specified in the ADL, and then begin grouping components into those domains. Once grouped, teams can now refactor those components to match the component and subdomain structure defined in the corresponding ADL describing the desired architecture.

For a more complete explanation of these refactoring patterns and how they work, refer to Chapter 5 of our *Software Architecture: The Hard Parts* book.

Summary

The techniques described in this chapter are not meant to be treated like unit tests, where a failed test means something is wrong and needs to be addressed right away. Rather, they're meant to provide the fastest possible feedback loop to ensure that architecture and implementation are correctly aligned. Failed tests are a call to action to modify the ADL, architecture diagrams, and corresponding fitness functions rather than to call out a bug or problem that must be fixes right away.

What architecture as code provides is a way to keep all of the architectural artifacts in sync with the actual implementation of the system as shown in Figure 4-6. These artifacts include architectural diagrams, ADL files, generated executable fitness functions, and the structure of the source code. It's primarily the use of ADL and executable code that accomplishes this amazing feat.

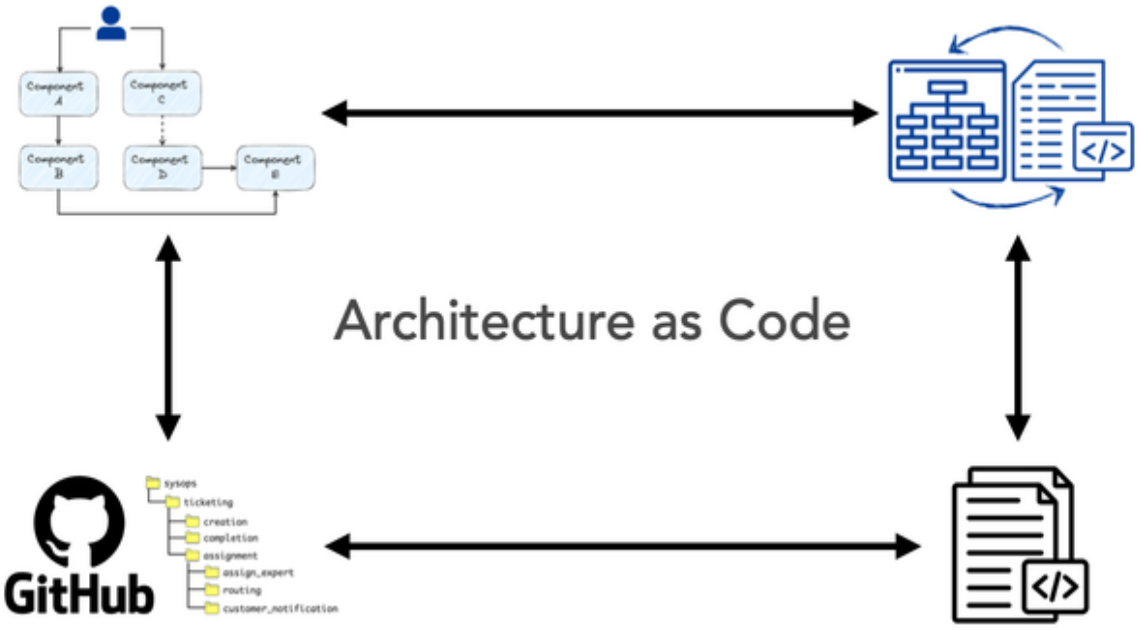


Figure 4-6. Architecture as code is about aligning all of the artifacts that comprise an architecture and the related implementation

So far in this part of the book on implementation we've demonstrated how to associate the structure of the source code with the architecture of a system, how to describe that architecture through ADL, and how to govern the architecture using ADL and AI agents to ensure they are both constantly aligned. In the final chapter of this part of the book (Chapter 5), we describe why this intersection is so critical to making architecture work and ensuring the success of a software system.

Chapter 5. Why This Intersection is Important

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

A building architect designing a 10-story office building is responsible for that building's structural integrity—its architecture. To ensure the building will be structurally sound, the architect, together with the structural engineer, takes into account loads, spans, stresses, weather, ground topology, building materials, and a variety of other factors. They document all of this in the building plans, which the construction contractors use to construct (or *implement*) the architecture. Those who work in the completed office building assume that the building plan was implemented according to the specifications and constraints the architect defined, so that the ceiling won't suddenly collapse or the walls suddenly buckle under extreme load.

A software system or product also has an architect–a software architect–who creates a structure (an architecture) to ensure that the system meets certain critical capabilities, such as scalability, responsiveness, availability, fault tolerance, and so on. The software developers function like a construction crew, implementing the architectural solution through source

code. Like the workers in the office building, the people who use this system assume it was implemented according to the specifications and will meet all of the architectural characteristics and business concerns defined by the business stakeholders. If not, then the system will likely come crashing down like a poorly constructed 10-story office building.

Aligning a system's implementation and its architecture is critical to that system's success. If they become misaligned, the system will fail to achieve its goals, and in some cases simply stop working. Unfortunately, this misalignment is fairly common. Let's look at some of the reasons why:

Communication and collaboration

Software architects and developers don't communicate and collaborate nearly enough. Developers complain that architects are never around and are too busy to answer their questions and issues. Architects complain that developers don't pay enough attention to the architecture and end up developing code that doesn't focus on the critical architectural characteristics. This lack of communication and collaboration usually results in the implementation becoming misaligned with the architecture.

Documentation falling out of sync

Source code is constantly changing as developers fix bugs, add new features, update features, and pay off technical debt. Sometimes those changes affect the structure–the architecture–of the system. When that happens, most of the time the corresponding architecture diagrams and documentation are not updated to reflect those source code changes, so the implementation and architecture become misaligned. Keeping the implementation of a system aligned and in sync with its architecture (and vice versa) is an ongoing challenge in most organizations.

Slow feedback loops

When development teams modify source code, they sometimes modify the corresponding architecture of that system as well. Unfortunately, the

feedback loop for architectural changes made through source code is usually too slow, so the architect of that system often doesn't realize soon enough (if ever) that the underlying architecture has been changed, resulting in a misalignment.

Lack of architectural understanding

Many development teams tasked with implementing or maintaining a system are unaware of its underlying software architecture. This may be due to a lack of communication and collaboration from the architect (the first challenge in this list), high levels of attrition and reassignment within the development team, or even legacy systems that have no architectural documentation (or even no architecture). In these situations it's impossible for the development team to implement or maintain the architecture correctly, causing it to become misaligned and not satisfy the system's business and architecture characteristics goals.

How a system is implemented can affect its architecture, and hence the corresponding architectural capabilities it must support. For example, if a given system requires high levels of scalability and elasticity, its architect must take care to minimize the amount of internal memory the services use, which can affect the system's mean-time-to-start (MTTS) and the ability to create additional instances of that service based on the amount of memory available in the virtual machine. A 500MB internal in-memory cache contained in a service instance might not seem like a lot, but when 10 instances of that service start up, that translates to a total memory consumption of 5GB. It's 10GB after 20 service instances. Imagine a system with 25 services, each containing a 500MB internal in-memory cache, with eight instances of each running. That translates to 25,000GB of required memory, which greatly exceeds the memory of most physical hardware, thus preventing the system from meeting its scalability requirements.

Implementation Misalignment

It's quite common for implementation and architecture to become misaligned. The reasons are plentiful: Architects are generally responsible for creating and maintaining the logical and physical architecture of a system, and development teams are responsible for implementing that architecture. Unfortunately, what happens all too often is that development teams use architecture diagrams to gain an understanding of the architecture's vision rather then treat them as a specification for the structure of the system. As a result, development teams don't take into account the details described in architecture diagrams during implementation and maintenance.

The reverse is also true—architects are most of the time unaware of how ongoing implementation changes impact the logical and physical architecture of the system, so the architecture diagrams and other artifacts quickly get out of sync with the implementation. In today's world of AI, this situation becomes significantly worse as development teams increasingly rely on AI agents to generate some (or all) of the source code for the system. In particular, unless agents fully understand the structural constraints of the system, agents applying continuous maintenance on that source code will likely alter the logical and physical architecture without the architect or the development teams knowing about it.

To illustrate this point, consider the logical architecture for the Sysops Squad illustrated in Figure 5-1, showing the `Customer`, `Ticketing`, and `Survey` domains.

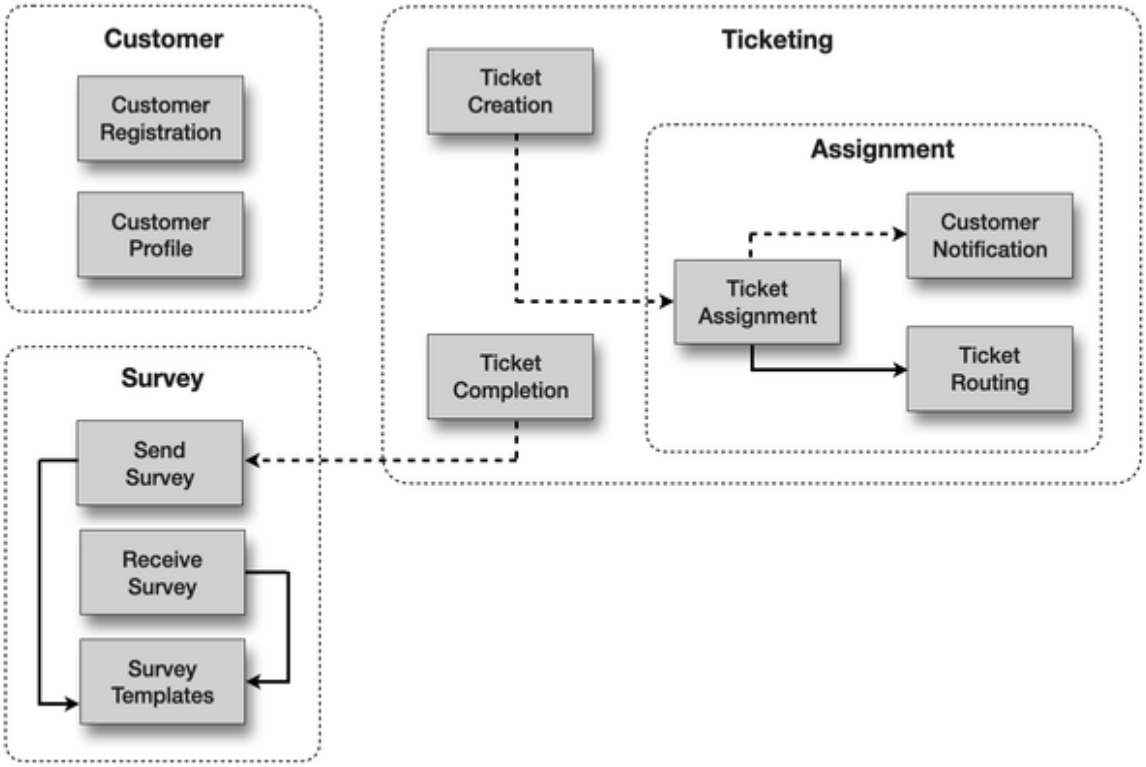


Figure 5-1. The logical architecture for the Sysops Squad system

Coming back to the concepts we described in the first chapter of this part of the book, Example 5-1 represents the corresponding directory structure for this architecture, keeping the implementation fully aligned with the architecture.

Example 5-1. The directory structure for the Sysops Squad system matching the logical architecture

```
sysops
 ├─ ticketing
 │    ├─ creation
 │    ├─ completion
 │    └─ assignment
 │         ├─ assign_expert
 │         ├─ routing
 │         └─ customer_notification
 ├─ customer
 │    ├─ profile
 │    └─ registration
 └─ survey
      ├─ send
      ├─ receive
      └─ templates
```

This source code structure not only matches the intended logical architecture of the system, but it's also easy to understand, easy to follow, and easy to maintain. Adding behavior to send SMS text messages in addition to emails notifying them when the expert will arrive is simply a matter of going to the `customer_notification` component and adding the necessary class files to support that new behavior.

However, the development team (along with some agentic help) created the Sysops Squad system using their own directory structure, resulting in the source code structure shown in Example 5-2.

Example 5-2. The resulting Sysops Squad directory structure created by the development team

```
sysops
 ├─ signup
 ├─ ticket
 │   ├─ expert
 │   ├─ mobile_connection
 │   └─ workflow
 │       ├─ send_ticket
 │       └─ close_ticket
 ├─ routing
 ├─ completion
 │   ├─ send_survey
 │   └─ close_ticket
 │       └─ email_notification
 └─ subscriber
     └─ new_ticket
```

This directory structure doesn't even come close to resembling the logical architecture diagram shown in Figure 5-1. Following the principles from the introduction chapter of this part (Chapter 2) regarding the relationship between architecture and source code, the development team effectively created five domains in this architecture:

- `Signup` (represented by the `signup` directory)
- `Ticket` (represented by the `ticket` directory)

- `Routing` (represented by the `routing` directory)
- `Completion` (represented by the `completion` directory)
- `Subscriber` (represented by the `subscriber` directory)

The `Ticketing` domain has one subdomain called `Workflow`, and the `Completion` domain has a subdomain called `Close Ticket`. Interestingly, the `Signup` and `Routing` domains don't have any subdomains or even components, meaning all of the class files representing those domains are directly under the respective directories.

It's source code structure like this that makes it difficult to understand, follow, and maintain. As a matter of fact, Figure 5-2 represents the corresponding logical architecture represented by this directory structure.

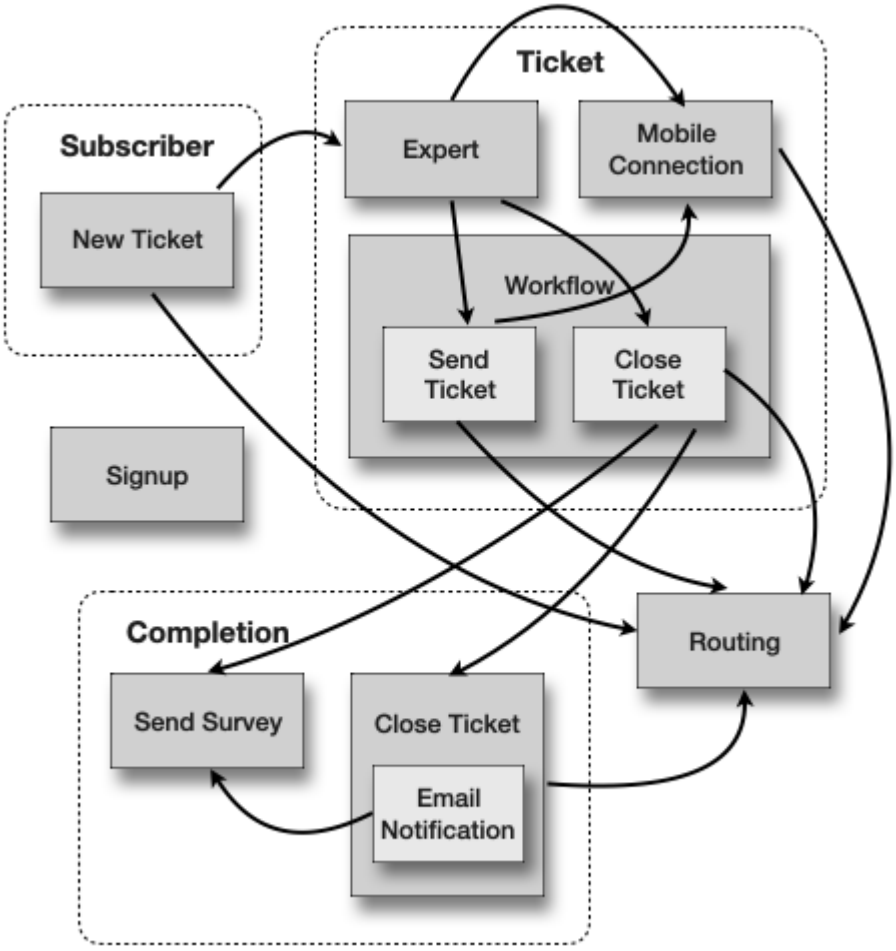


Figure 5-2. The Sysops Squad logical architecture created by the development team

Even though this architecture passes all of the functional unit and scenario tests, it's misaligned with the original intended architecture, which therefore risks not supporting the architectural capabilities the system requires. Because the implemented *behavior* is flawless—that is, it doen't contain any bugs—architects and development teams might ignore this misalignment, questioning the reasons why aligning implementation and architecture is such a big in the first place. After all, the system works, doesn't it?

Comparing the intended architecture illustrated in Figure 5-1 with that of the resulting architecture in Figure 5-2, the negative implications of this kind of misalignment become apparent. These include:

Agility

Agility is the ability to respond quickly to change. It's made up of three primary architectural characteristics: maintainability, testability, and deployability. *Maintainability* describes how easy it is for a developer to locate where in the code to apply a change, fix a bug, or add a new feature. *Testability* describes the ease of and completeness of testing and the risk of introducing bugs as a result of changing or adding code. *Deployability* is about the ceremony, frequency, and risk of releasing the software.

Of the Sysops Squad's two logical architectures, the original intended one created by the software architect is more maintainable, testable, and less risky to deploy–and thus more agile–than the one created from the implemented directory structure. The original architecture is less dynamically decoupled, clearer, and more understandable.

Reliability

Reliability is a *composite* architectural characteristic, or one made up of several other characteristics–in this case, availability, testability (for bugs and errors), data integrity, data consistency, and fault tolerance. Of the two logical architectures for the Sysops Squad, the original intended one created by the software architect is more reliable than the one created through implementation. This is because the original architecture contains less dynamic coupling between components and contains more definable components that the implemented version, making it easier for developers to locate, apply, and test changes, resulting in fewer bugs and unexpected side-effects based on changes to the source code.

Adaptability

We define adaptability here to mean the system's ability to adapt to changes in its environment, platform, and behavior. The original logical architecture is more adaptable than the one created by the development

team because the components and their interactions are more clearly defined, allowing developers to more easily identify areas of the architecture that are impacted by these kind of changes.

Extensibility

Extensibility is the ease with which the architecture can be extended to include additional functionality and features like additional domains, subdomains, and logical components. Of the two architectures, the one created by the development team through the structure of the source code is less extensible than the original, because the more defined the components and their clear interactions, the easier it is to extend the architecture with additional functionality.

Migration

Migration means converting a legacy system (usually a monolithic architecture) to a distributed architecture (such as microservices or event-driven architecture). The original architecture would be much easier to migrate than the one created by the development team because the logical components and their boundaries are better defined, allowing developers to better identify and create services based on those components.

It's important to note that this kind of misalignment doesn't mean the development team was acting irresponsibly when implementing the architecture. Rather, it usually stems from a lack of communication and collaboration between software architects and development teams, which generally results in the development teams not fully understanding the architecture. It can also result from not having enough automated governance in place to detect when the implementation doesn't match the architecture. Architectural diagrams are great for conveying ideas and concepts and for documenting the architecture of a system, but they aren't easy to enforce.

Summary

The intersection between architecture and implementation is an important one, particularly since it's so common and so easy to miss. By using the techniques described in this part, you can describe an architecture using ADL, use that ADL to generate executable code, and create a governance model to ensure the two always stay in alignment, all under the context of *architecture as code*.

Part III. Architecture and Data

Many teams treat a system's physical architecture and its corresponding data architecture as separate. However, a system's data topology can directly influence its architecture (and vice versa). To succeed, the architecture and data must constantly be aligned with the data. Fortunately, this intersection can be described, governed, and validated using architecture as code.

We've broken down this part of the book into five main chapters:

Chapter 6: Introducing data topologies

This introductory chapter provides an overview of the three basic data topologies and their strengths and weaknesses.

Chapter 7: Describing the data topology using ADL

This chapter shows how to describe a system's data architecture using Architecture Definition Language (ADL), including both documenting the architecture and laying the foundation for governing and validating it. Here, we show how each of the data topologies described in Chapter 6 works with various architectural styles and how the ADL relates to that style's physical architecture.

Chapter 8: Governing the data topology using ADL and AI Agents

Governing a data architecture is about ensuring that the system's physical data topology matches what is defined in the corresponding ADL. In this chapter we show how to leverage AI agents and the execution platform to ensure the data topology described in the ADL is implemented correctly. We also discuss how to verify that the bounded contexts defined in the ADL match the system's physical

implementation and ensure that data is consistent when using the domain database and dedicated database topologies.

Chapter 9: Validating the data topology using fitness functions

In this chapter we introduce another form of architecture as code, executable fitness functions, and show how to use them to continuously validate the team's decisions about the database topology while accounting for the system's evolution and the company's growth.

Chapter 10: Why this intersection is important

This final chapter focuses on *why* the intersection of architecture and data is so important. Here, we dive deep into each database topology's strengths, weaknesses, and data-modeling considerations and evaluate which architectural styles align well with which database topologies.

Chapter 6. Introducing Data Topologies

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

A system's *database topology* refers to how the physical databases are partitioned. The three primary data topologies include *monolithic database*, *domain database*, and *dedicated database*. Because each of these topologies has its strengths and weaknesses, misalignment with the system's architecture can cause the system to stop functioning properly, therefore not meeting the needs of the business.

Monolithic Database

The most basic database topology is the *monolithic database*, as illustrated in Figure 6-1. Here, a set of services (or a monolithic architecture) share a single monolithic database.

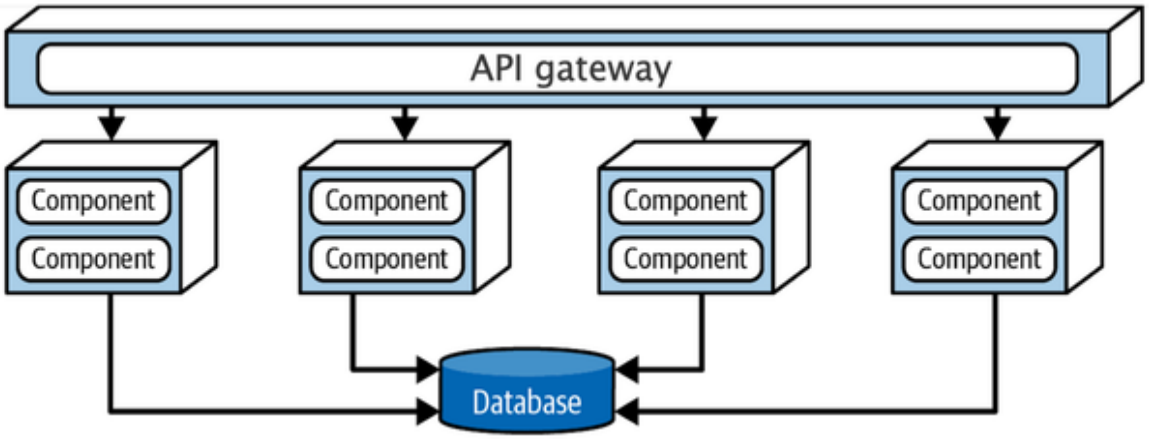


Figure 6-1. All services share a single monolithic database

While this topology allows each service to easily access any data it needs, the system might have challenges from a database perspective with operational capabilities such as scalability, responsiveness, and fault tolerance. While teams can easily scale services by creating additional service instances, they find the database is not as easy to scale. While certain database types such as columnar databases like Cassandra and key-value databases like DynamoDB can help to improve scalability, nevertheless it's the database that usually becomes the ultimate bottleneck for scalable systems. Teams using the monolithic database topology might also find responsiveness an issue due to the workload on the database and limits in terms of the number of connections it supports. From a fault tolerance and overall system availability standpoint, this topology creates a single point of failure (SPOF) in the overall system architecture, something that's hard to address without costly database-specific clustering techniques.

From a process-related capability standpoint, changes to the structure (schema) of the database can impact multiple services, requiring teams to spend more time and effort in locating affected code and coordinating the consolidated testing effort. Because of this, structural database changes in the monolithic database topology are less agile and carry with them more deployment risk than with other database topologies. We discuss these operational and process-related issues in detail in Chapter 9 and Chapter 10.

Table 6-1 below summarizes the database scope, strengths, and weaknesses of the monolithic database topology

Table 6-1. Monolithic database summary

Monolithic Database Topology	
Database Scope	System-Level Behavior
Strengths	Simplicity, Cost, System-Wide Database Access, Data Modeling Ease
Weaknesses	Scalability, Responsiveness, Fault Tolerance, Database Change Ease

Dedicated Database

Now consider the other extreme, the *dedicated database* topology (also referred to as the *database-per-service* pattern), as shown in Figure 6-2. In this topology, which is common in microservices architectures, each service exclusively owns its own database (or schema) in a well-formed bounded context.

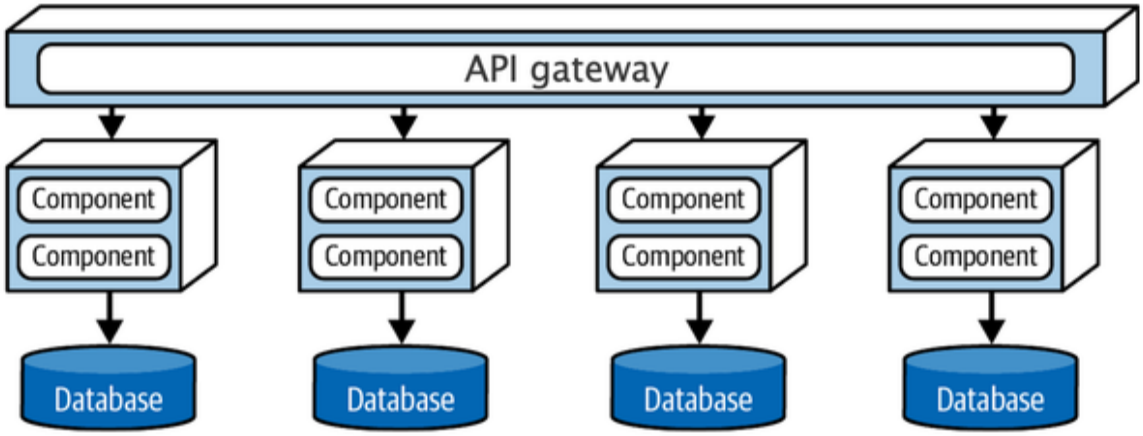


Figure 6-2. Each services owns its own database or schema

The characteristics in this topology are exactly the opposite from those found in the monolithic database topology. Scalability, responsiveness, and fault tolerance are generally not an issue because each service owns its own data (whether it be in a separate physical database or a collection of tables in a schema), meaning the database is only accessed from fine-grained *service-level* behavior. Furthermore, because of the tight bounded contexts formed between the service and its corresponding data, teams will find managing structural database changes—even changing the type of database—becomes easier to change, easier to test, and easier and less risky to deploy than with the monolithic database topology.

However, due to the strict bounded context and exclusive database access found in this database topology, services no longer have easy access to data residing in other bounded contexts. This requires services to make remote calls to other services or use elaborate caching mechanisms to gain access to the data, resulting in tightly coupled and dependent services. If teams aren't careful when using this database topology, they can quickly turn a well-formed architecture into a big ball of distributed mud. We discuss these data access and bounded context issues in more detail in Chapter 8, Chapter 9, and also Chapter 10.

Table 6-2 below summarizes the database scope, strengths, and weaknesses of the dedicated database topology

Table 6-2. Dedicated database summary

Dedicated Database Topology	
Database Scope	Service-Level Behavior
Strengths	Scalability, Responsiveness, Fault Tolerance, Database Change Ease
Weaknesses	Simplicity, Cost, Interservice Communication, Service Coupling, Data Modeling Ease

Domain Database

While these two topologies represent the extremes of a spectrum, the last database topology sits right in the middle—the *domain database* topology. In this topology, as shown in Figure 6-3, groups of services—usually belonging to the same domain—share a common domain database.

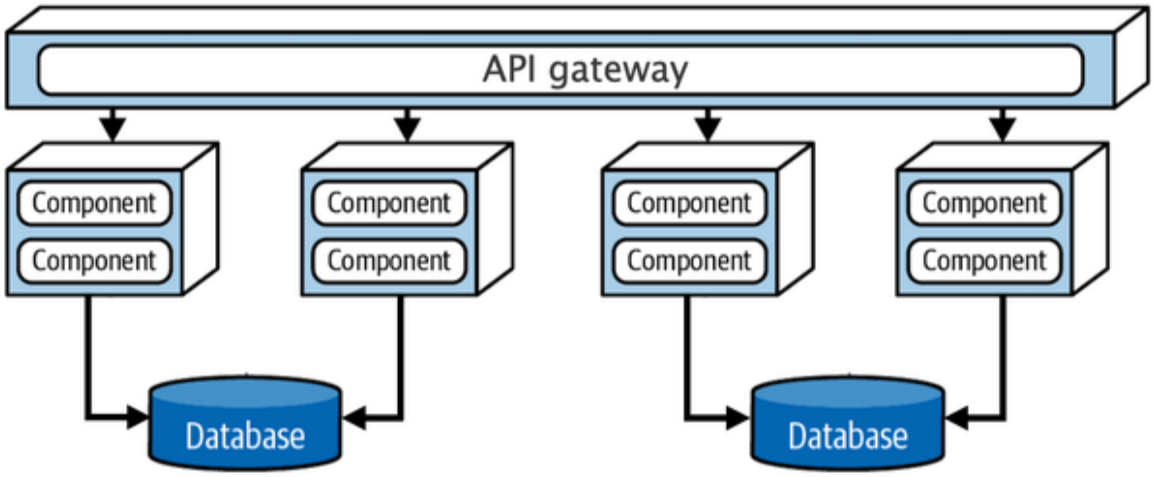


Figure 6-3. Services share a database within a domain

With the domain database topology, teams can have coarse-grained services or fine-grained microservices that logically reside in a particular domain share a common domain database as shown in Figure 6-3. Alternatively, teams can create coarse-grained domain services—those typically found in Service-Based architecture—that are dedicated to a single domain database, mimicking the dedicated database topology illustrated in Figure 6-2. Regardless of the number of services associated with a particular domain, it's the collective *domain behavior* that shares a database in this topology, rather than the more fine-grained single-purpose *service behavior* found in the dedicated database topology.

Operational capabilities such as scalability, responsiveness, and fault tolerance are much better than the monolithic database topology because there's less behavior accessing the database, meaning the operational capabilities reside at the *domain level* rather than at the *system level*. However, this topology is not as good operationally as the dedicated database because the database is accessed by more behavior. As we discuss in Chapter 9 and Chapter 10, certain domains may need to scale or be more fault tolerant than others, requiring validation using fitness functions to properly size and partition the domains in the system.

Table 6-3 below summarizes the database scope, strengths, and weaknesses of the domain database topology

Table 6-3. Domain database summary

Domain Database Topology	
Database Scope	Domain-Level Behavior
Strengths	Scalability, Responsiveness, Fault Tolerance, Database Change Ease
Weaknesses	Simplicity, Cost, Interservice Communication, Service Coupling, Data Modeling Ease

Hybrid Topologies

Using the domain database and dedicated database topologies, teams can alternatively create hybrid topologies such as the one shown in Figure 6-4.

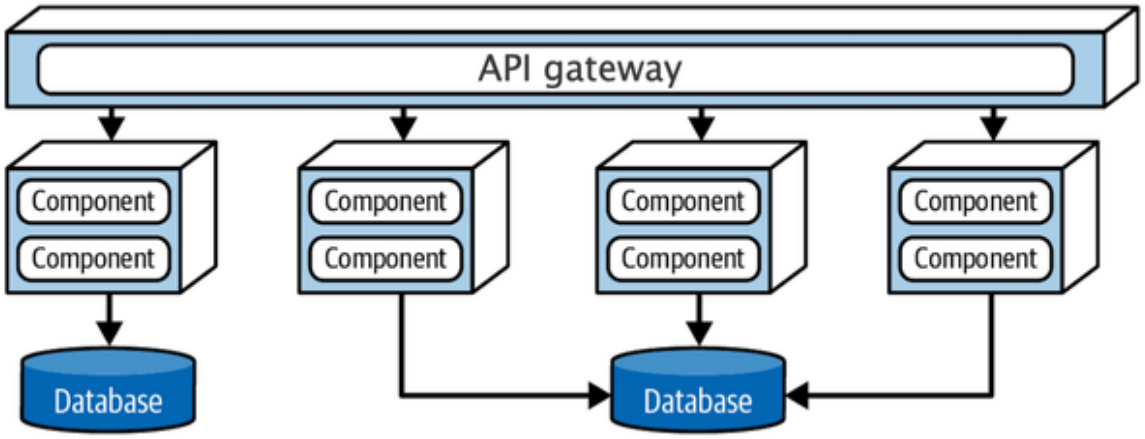


Figure 6-4. Services can have both dedicated and shared databases

Hybrid topologies work well when teams need to break apart a domain to get better scalability, fault tolerance, or database schema change control.

They also work well when teams need to consolidate and share data between fine-grained microservices to avoid interservice communication. However, we urge teams to fully understand the implications of each primary database topology before forming hybrid ones. We describe the specific topology considerations in detail in Chapter 10.

Chapter 7. Describing Data Using ADL

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

Architects typically use diagrams to illustrate and describe the database topology and what services connect to which databases. The problem is, diagrams are not enforceable, rarely reflect the the actual implementation, are not the most effective way of teaching AI agents about the data topology. How can architects solve these problems? The answer, of course, is *architecture as code*.

In Part II of the book on implementation, we demonstrated how to describe and and govern the logical and physical architecture of a system through ADL by identifying domains, subdomains, components, and services. In this chapter, we focus on how to use ADL to describe the data architecture based on the three primary data topologies: *monolithic database*, *domain database*, and *dedicated database*. We'll use this ADL in subsequent chapters to demonstrate how to govern and validate the system's data architecture.

Monolithic Architecture and Monolithic Data

The most basic combination of data topology and architecture style uses the monolithic database topology with a monolithic architecture, as shown in Figure 7-1.

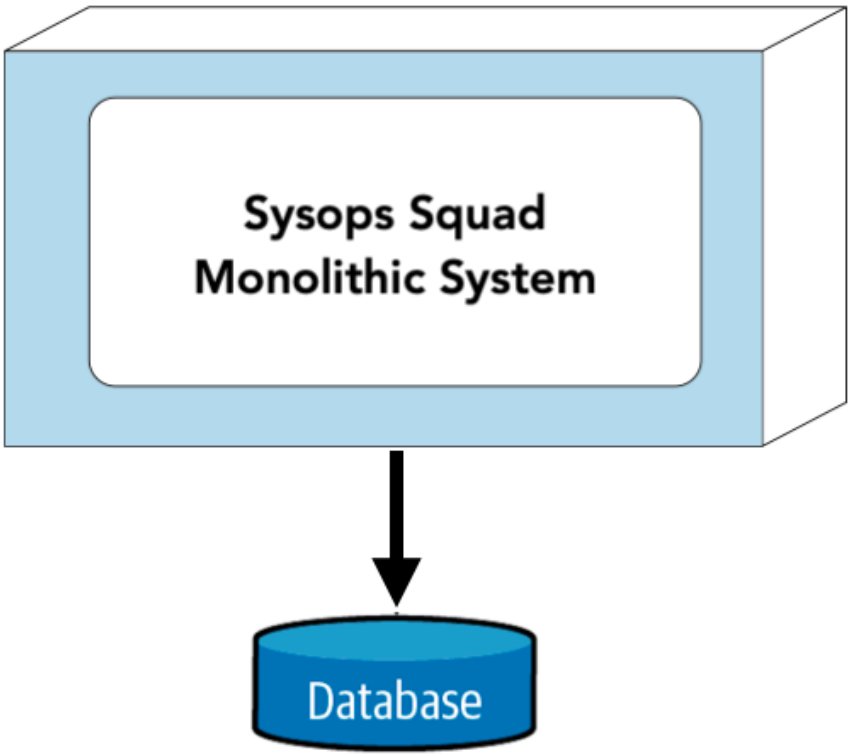


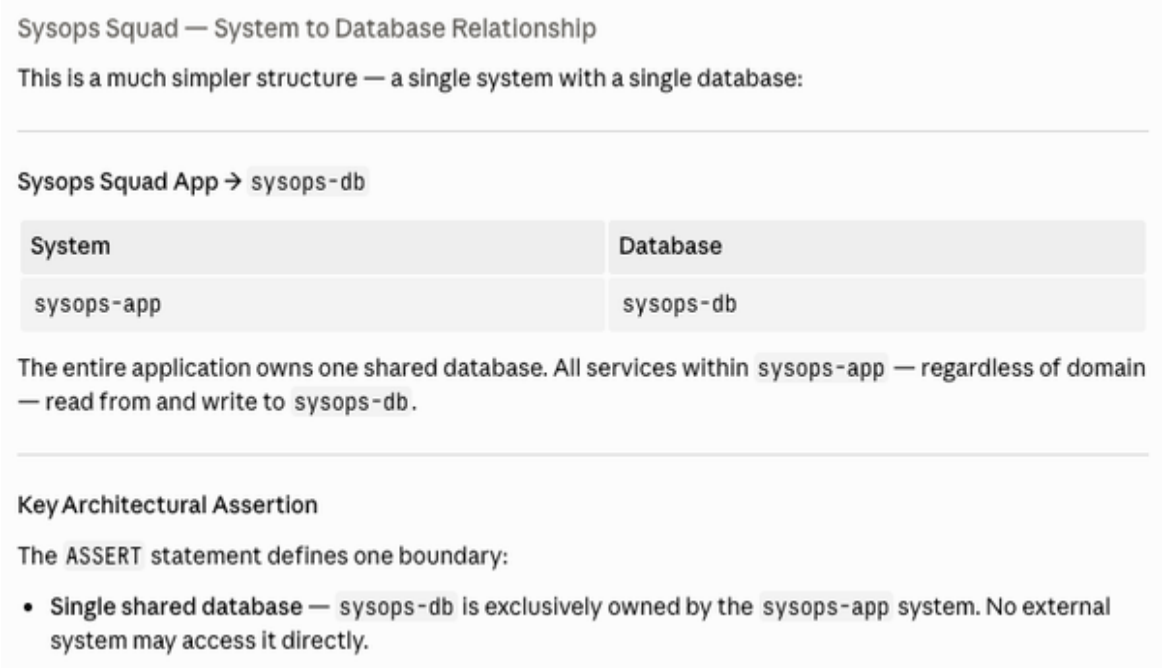
Figure 7-1. A monolithic database topology with a monolithic architecture

Here, the monolithic architecture defines the domains, subdomains, and components of the Sysops Squad system. All components have access to the same database and database schema. We've showed how to define the logical architecture of a system through ADL. Teams can also use ADL to describe the *physical architecture* by defining the monolithic system and its corresponding monolithic database, as shown in Example 7-1.

Example 7-1. ADL describing the monolithic Sysops Squad system and its corresponding database

```
DESCRIPTION Define System and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app
   DEFINE DATABASE Sysops Squad Database AS sysops-db
ASSERT(Sysops Squad Database is only accessed by Sysops Squad App)
```

Teams can pass this ADL to an AI agent to help it understand the overall system topology, database relationships, and database access rules. Figure 7-2 illustrates the results of the ADL shown in Example 7-1.[1]



Sysops Squad — System to Database Relationship

This is a much simpler structure — a single system with a single database:

Sysops Squad App → `sysops-db`

System	Database
`sysops-app`	`sysops-db`

The entire application owns one shared database. All services within `sysops-app` — regardless of domain — read from and write to `sysops-db`.

Key Architectural Assertion

The `ASSERT` statement defines one boundary:

- **Single shared database** — `sysops-db` is exclusively owned by the `sysops-app` system. No external system may access it directly.

Figure 7-2. AI agent interpretation from the ADL describing a monolithic system with a monolithic database topology

The ADL in Example 7-1 defines the physical system (`sysops-app`) and the physical database (`sysops-db`). It defines database access by asserting that the Sysops Squad system is the *only* system that has access to the Sysops Squad database. The system name, `sysops-app`, points to the name of the directory in the repository defining the system, as well as to the name of the physical node or server on which the system is running. Figure 7-3 illustrates these relationships using the Kubernetes platform.

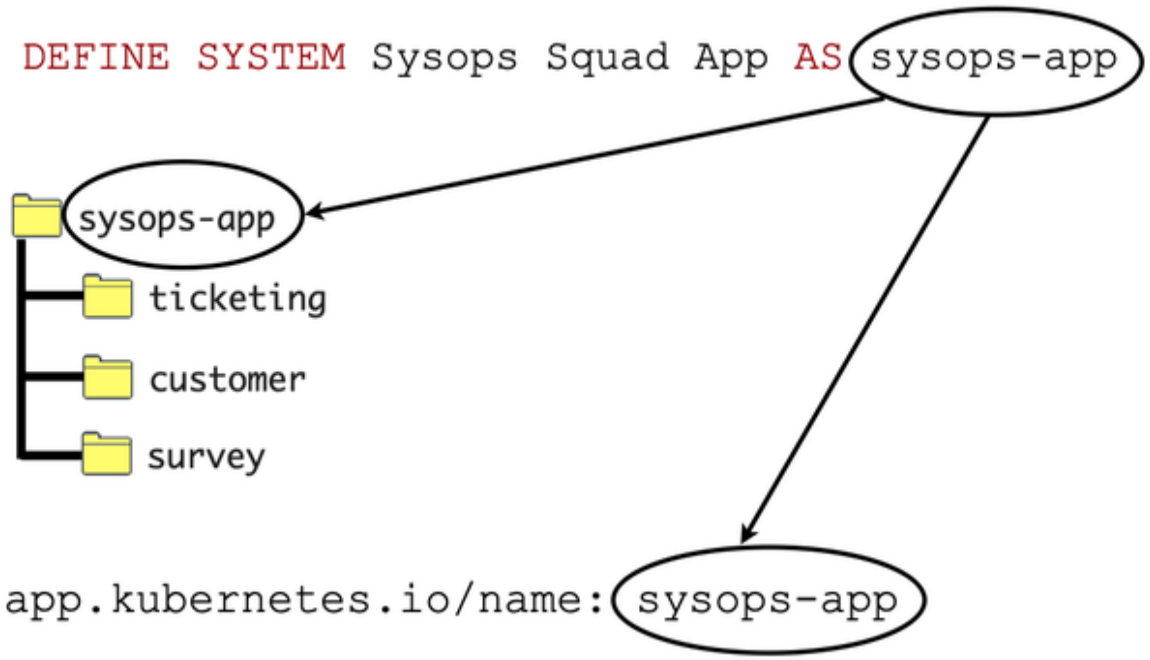


Figure 7-3. The relationship between ADL, the directory structure, and the execution platform for defining the physical system

The physical database name (in this case, `sysops-db`) points to the name of the database as defined in the database connection string, and the physical server or platform on which the database runs. Figure 7-4 illustrates an example of these relationships using a Postgres database running on the Kubernetes platform.[2]

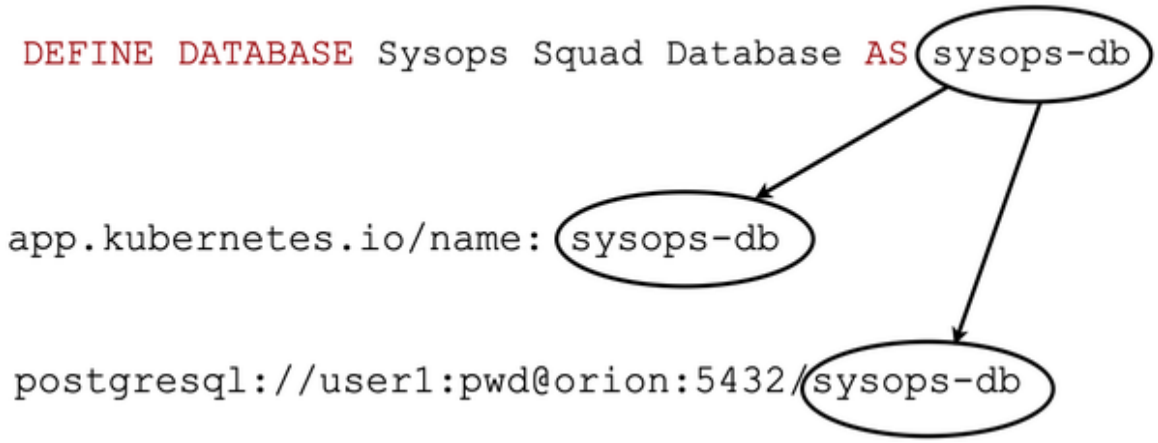


Figure 7-4. The relationship between ADL, the physical database, and the execution platform for defining the physical database

Teams can also specify the actual connection string in the ADL file by adding a `USING` directive in the artifact definition as follows:

```
DEFINE DATABASE Sysops Squad Database AS sysops-db
   USING postgresql://user1:pwd@orion:5432/sysops-db
```

The `USING` directive in ADL is used to specify an *implementation* detail about the architecture. The advantage of this directive is that it provides specific details about the physical architecture of the system and defines the physical nature of the database alongside the logical definition. However, the trade-off is that it introduces specific implementation coupling into the architecture specification. Changes to the implementation would therefore change the architecture specification, which is usually orthogonal to the implementation of the architecture.

To keep implementation details from leaking into the ADL, we suggest utilizing configuration files or infrastructure-as-code to decouple this association as follows:

```
# database implementation
sysops-db = postgresql://user1:pwd@orion:5432/sysops-db
```

The important aspects of these relationships and the alignment of the names will become apparent in the next few chapters as we demonstrate how to govern the physical architecture through ADL.

Domain Services and Monolithic Data

Teams can use the monolithic database topology with distributed architectures, too. This is common in service-based architectures and also some smaller event-driven architectures.[3] In this topology, all services within the system share a single monolithic database, as illustrated in Figure 7-5.

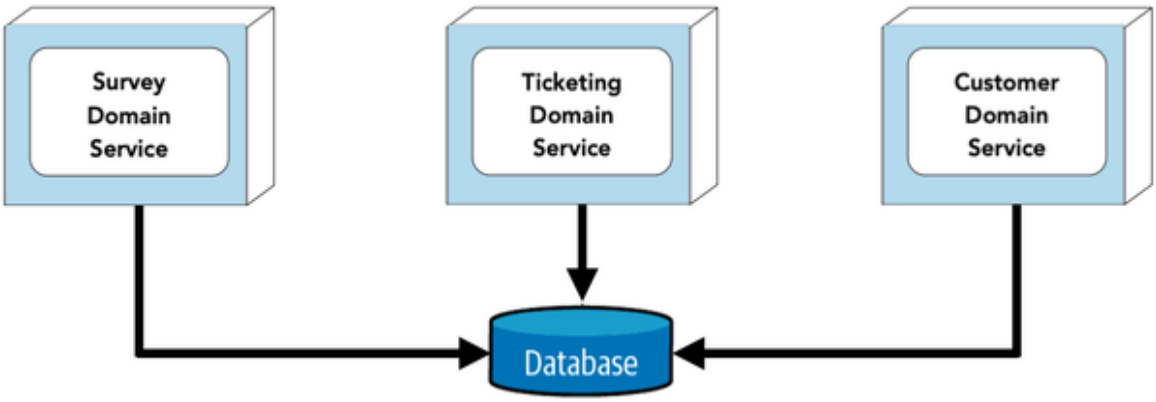


Figure 7-5. A monolithic database topology with a distributed architecture

Assuming they have a monolithic repo, teams can specify both the logical *and* physical architecture of the system by defining the domains, services, and components within each service in the ADL. They can also specify the database at the same time, as shown in Example 7-2.[4]

Example 7-2. ADL describing the distributed Sysops Squad system and its corresponding database

```
DESCRIPTION Define Service and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app
   DEFINE DATABASE Sysops Squad Database AS sysops-db

   DEFINE DOMAIN Ticketing Domain as ticketing
      DEFINE SERVICE Ticket Service AS ticket-service

   DEFINE DOMAIN Customer Domain as customer
      DEFINE SERVICE Customer Service AS customer-service

   DEFINE DOMAIN Survey Domain as survey
      DEFINE SERVICE Survey Service AS survey-service

ASSERT(Sysops Squad Database is only accessed by SERVICES within
Sysops Squad App)
```

This ADL helps AI agents understand the overall system topology from a service and database relationship standpoint, along with database access rules. Figure 7-6 illustrates the results from the ADL shown in Example 7-2.

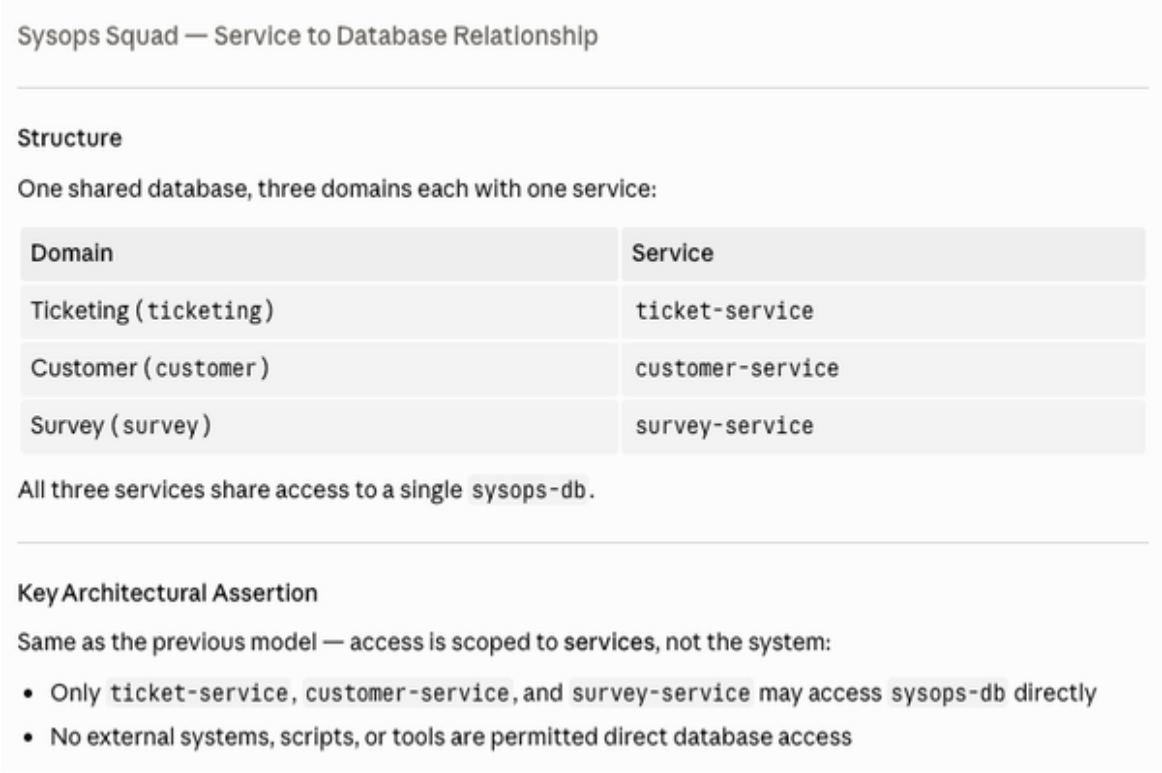


Figure 7-6. AI agent interpretation from the ADL describing a distributed system with a monolithic database topology

Notice, in Example 7-2, that the assertion restricts database access to *only* those services defined in the ADL, meaning no other services or systems can access the database. (We'll demonstrate how to govern this in Chapter 8 using this ADL).

While the logical and physical database name relationships remain the same as they were in the last example, the logical and physical *service* name relationships are different. Here, rather than linking the system name (`sysops-app`) in the ADL to implementation artifacts, teams link the service names instead. This relationship is illustrated in Figure 7-7 using Kubernetes and a monolithic repository as an example.

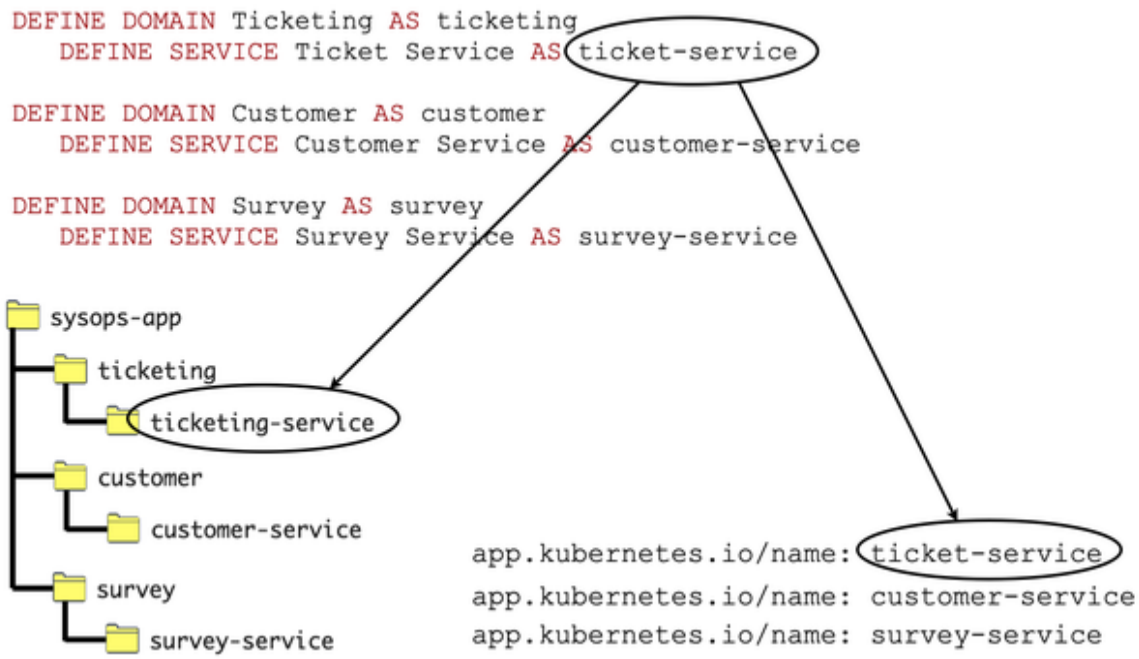


Figure 7-7. The relationship between ADL, the directory structure, and the execution platform for defining the physical services

Since the relationships between the ADL and the physical services and databases are the same for the other topologies, we won't repeat them in the following sections. Instead, we'll focus solely on the ADL definitions.

Domain Services and Domain Data

A common topology in service-based architectures (SBAs) is a distributed system consisting of domain services using the domain-database topology as illustrated in Figure 7-8. In this topology, both the architecture and the database focus on *domain partitioning* within the system. Domain partitioning within an architecture means that the highest-level architectural concept is a domain (such as `com.sysops.ticketing`), as opposed to a *technically partitioned* architecture where the highest-level concept is the technical usage of those components (such as `com.sysops.presentation`).

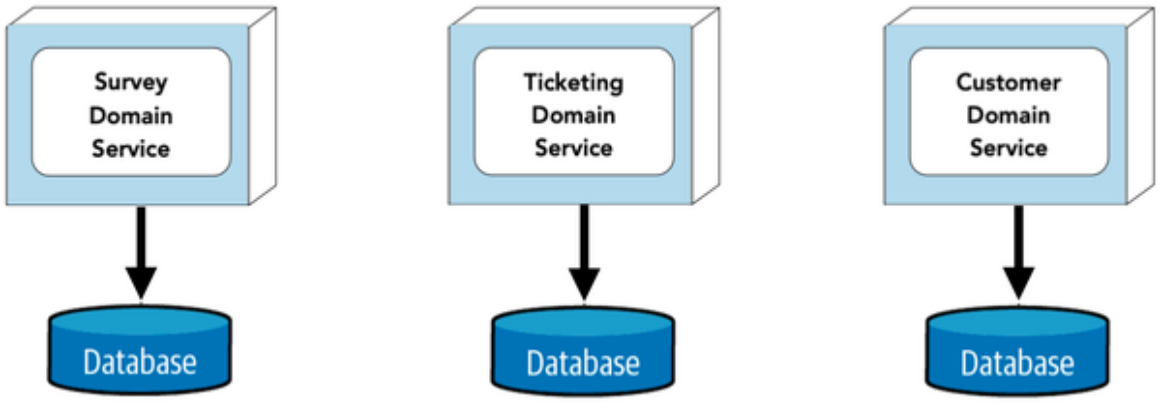


Figure 7-8. A domain database topology with a distributed architecture containing domain services

Notice the strong domain alignment with this overall topology. Although it's not a requirement, each domain service in this example has its own domain database (we'll show a hybrid of this topology a little further down in this section). Like the example in the previous section, assuming they have a monolithic repo, teams can use ADL to specify both the logical *and* physical architecture of the system by defining the domains, services, and components within each service, as well as to define the databases and the relationships between them and the domain services, as shown in Example 7-3.

Example 7-3. ADL describing the distributed Sysops Squad system using a domain database topology

```
DESCRIPTION Define Service and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app
   DEFINE DATABASE Ticketing Database AS ticketing-db
   DEFINE DATABASE Customer Database AS customer-db
   DEFINE DATABASE Survey Database AS survey-db

   DEFINE DOMAIN Ticketing Domain as ticketing
      DEFINE SERVICE Ticket Service AS ticket-service

   DEFINE DOMAIN Customer Domain as customer
      DEFINE SERVICE Customer Service AS customer-service

   DEFINE DOMAIN Survey Domain as survey
      DEFINE SERVICE Survey Service AS survey-service

ASSERT(Ticketing Database is only accessed by Ticketing Service)
ASSERT(Customer Database is only accessed by Customer Service)
ASSERT(Survey Database is only accessed by Survey Service)
```

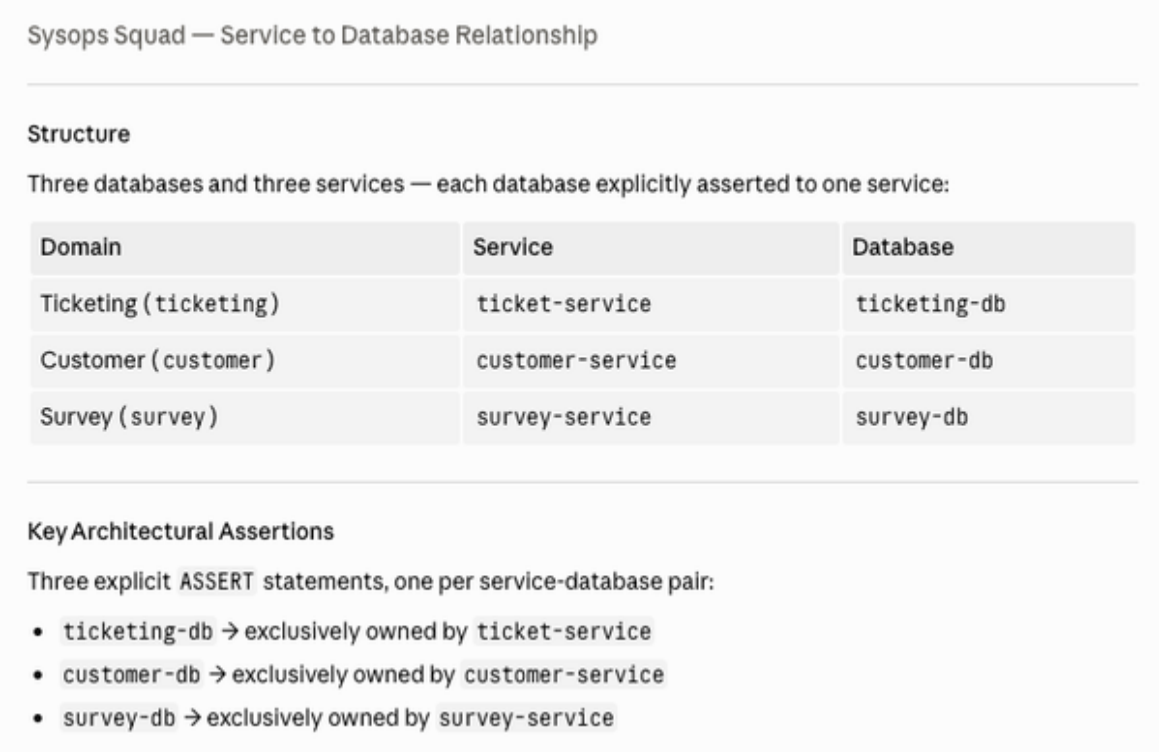


Figure 7-9. AI agent interpretation from the ADL describing a distributed system using a domain database topology

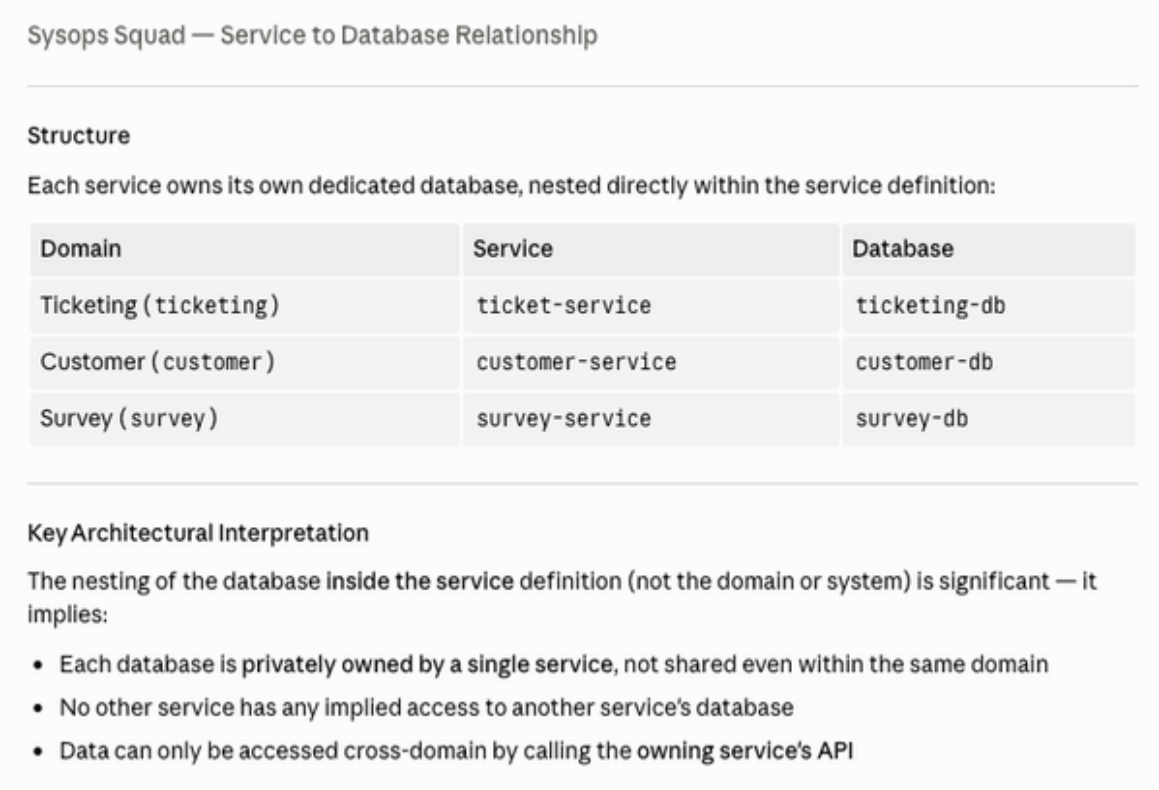
This ADL tells AI agents about the overall system topology, its a service and database relationships, and its database access rules. Figure 7-9 illustrates the results from the ADL shown in Example 7-3. The assertion restricts database access to *only* those domain services matching each corresponding domain database, which essentially defines the physical bounded context between the service and its corresponding data.

Example 7-4. ADL defining the database ownership and access as part of the service definition rather than with assertions

```
DESCRIPTION Define Service and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app

   DEFINE DOMAIN Ticketing Domain as ticketing
      DEFINE SERVICE Ticket Service AS ticket-service
         DEFINE DATABASE Ticketing Database AS ticketing-db

   DEFINE DOMAIN Customer Domain as customer
      DEFINE SERVICE Customer Service AS customer-service
         DEFINE DATABASE Customer Database AS customer-db

   DEFINE DOMAIN Survey Domain as survey
```

```
DEFINE SERVICE Survey Service AS survey-service
   DEFINE DATABASE Survey Database AS survey-db
```

This type of embedded hierarchical structure yields the correct results as interpreted by an AI agent, as shown in Figure 7-10.



Sysops Squad — Service to Database Relationship

Structure

Each service owns its own dedicated database, nested directly within the service definition:

Domain	Service	Database
Ticketing (`ticketing`)	`ticket-service`	`ticketing-db`
Customer (`customer`)	`customer-service`	`customer-db`
Survey (`survey`)	`survey-service`	`survey-db`

Key Architectural Interpretation

The nesting of the database inside the service definition (not the domain or system) is significant — it implies:

- Each database is privately owned by a single service, not shared even within the same domain
- No other service has any implied access to another service's database
- Data can only be accessed cross-domain by calling the owning service's API

Figure 7-10. AI agent interpretation from the ADL describing a distributed system using a domain database topology with a hierarchical structure, rather than assertions

Due to ADL's hierarchical nature, it can be tempting to define the database as part of the service definition when using domain data (as shown in Example 7-4) to avoid the need to provide assertions about data ownership and access. However, we strongly recommend *avoiding* this technique when defining the service-to-database relationship. Here's why: as we mentioned before, using the domain database topology doesn't mean each domain service *must* have a corresponding database. It's perfectly valid in this topology for multiple domains to share a single domain database. For example, suppose the team decides to share data between the survey and ticketing domains, so that other domains don't have to rely on messy interservice communication to gain access to data (see Figure 7-11).

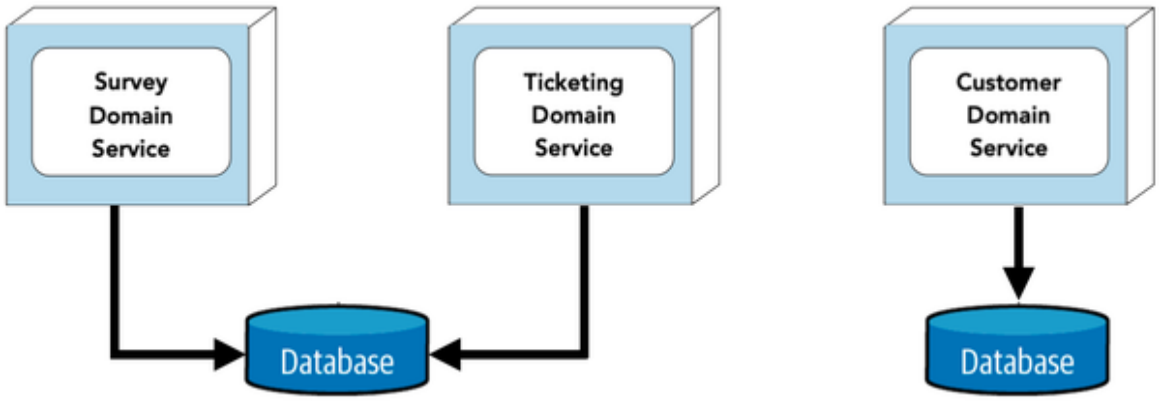


Figure 7-11. A domain database topology with a distributed architecture containing domain services that share domain data

The ADL definition shown in Example 7-4 would no longer work unless the team duplicates the database definition when defining the ticketing and survey services, which might mislead anyone reading or interpreting it (human or agent):

```
DEFINE DOMAIN Ticketing Domain as ticketing
   DEFINE SERVICE Ticket Service AS ticket-service
      DEFINE DATABASE Ticketing Database AS ticketing-db

DEFINE DOMAIN Survey Domain as survey
   DEFINE SERVICE Survey Service AS survey-service
      DEFINE DATABASE Ticketing Database AS ticketing-db
```

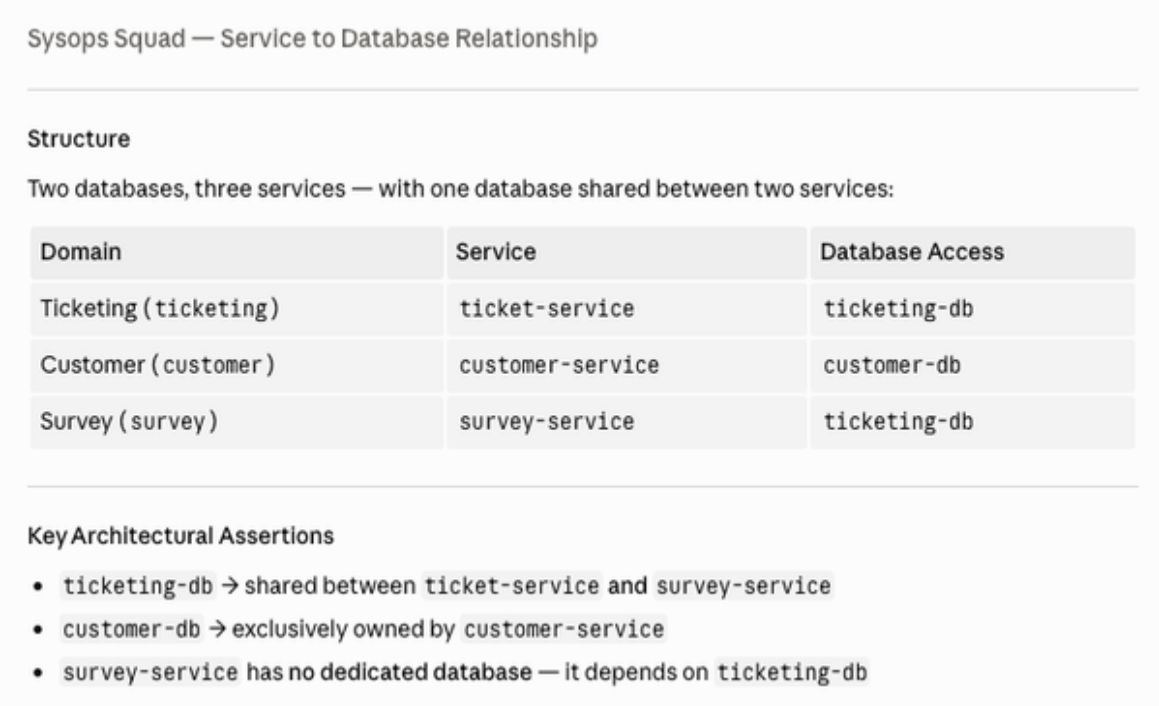
Using assertions to describe the service-to-database relationship when sharing domain data provides an *explicit* relationship definition rather than an *implicit* one, as shown in Example 7-5.

Example 7-5. ADL defining shared database ownership and access using assertions

```
DESCRIPTION Define Service and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app
   DEFINE DATABASE Ticketing Database AS ticketing-db
   DEFINE DATABASE Customer Database AS customer-db

   DEFINE DOMAIN Ticketing Domain as ticketing
```

```
      DEFINE SERVICE Ticket Service AS ticket-service

   DEFINE DOMAIN Customer Domain as customer
      DEFINE SERVICE Customer Service AS customer-service

   DEFINE DOMAIN Survey Domain as survey
      DEFINE SERVICE Survey Service AS survey-service

ASSERT(Ticketing Database is only accessed by
       Ticketing Service and Survey Service)
ASSERT(Customer Database is only accessed by Customer Service)
```

Not only does an AI agent correctly interpret this topology, as illustrated in Figure 7-12, it clarifies the topology for human readers as well.



Sysops Squad — Service to Database Relationship

Structure

Two databases, three services — with one database shared between two services:

Domain	Service	Database Access
Ticketing (ticketing)	ticket-service	ticketing-db
Customer (customer)	customer-service	customer-db
Survey (survey)	survey-service	ticketing-db

Key Architectural Assertions

- ticketing-db → shared between ticket-service and survey-service
- customer-db → exclusively owned by customer-service
- survey-service has no dedicated database — it depends on ticketing-db

Figure 7-12. AI agent interpretation from the ADL describing a distributed system using a shared domain database topology with assertions

Microservices and Dedicated Data

Another common topology combination is the dedicated database (database-per-service) topology with a microservices architecture. This

topology, illustrated in Figure 7-13, promotes tight, well-formed physical bounded contexts between a service and its corresponding data.

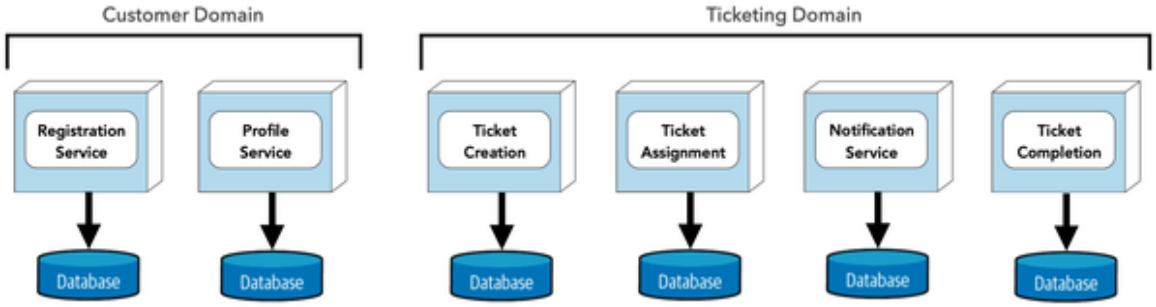


Figure 7-13. A dedicated database topology with a microservices architecture

Here, each service owns its own data, which maximizes operational capabilities like scalability, responsiveness, and fault tolerance. With respect to database structure changes, this combination also maximizes process-related capabilities, including maintainability, testability, and deployability. These three architectural characteristics define the composite architectural characteristic *agility*, or the ability to respond quickly to change.

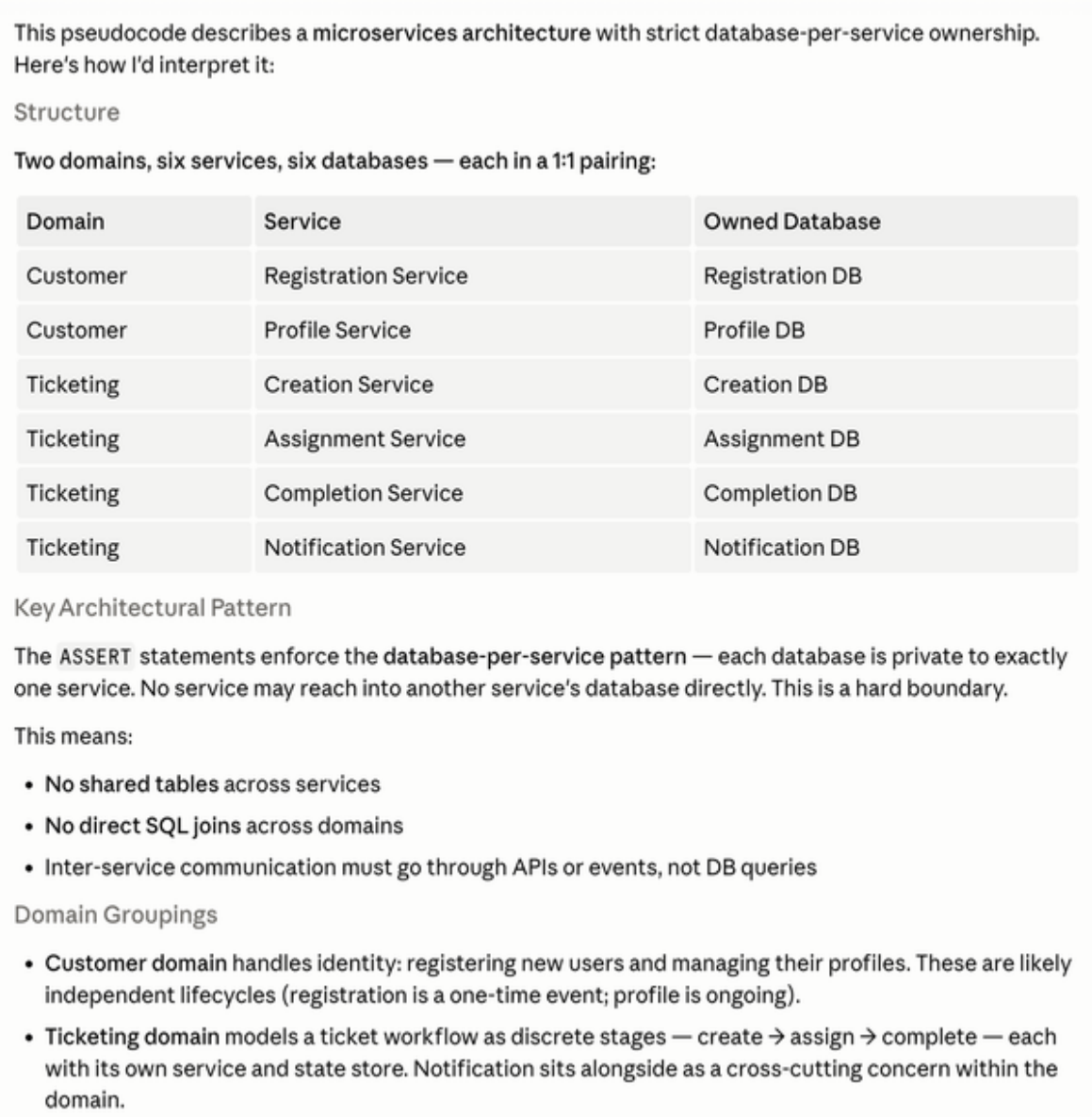
With this topology, teams can use a monolithic repository containing the code for all services within the system, a domain-based repository containing the code for all services within a particular domain, or a service-based repository containing only the code for a particular service. (See [Link to Come] for advice on repository granularity when using microservices or Service-Based architectures) For the purpose of defining the database topology relationships within the ADL, we'll assume a monolithic repository. This lets us show both the customer and ticketing services and their database relationships in the same ADL file (see Example 7-6).

Example 7-6. ADL describing the microservices-based Sysops Squad system for the ticketing and customer domains using the dedicated database topology

```
DESCRIPTION Define Microservices and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app

  DEFINE DATABASE Registration Database AS registration—db
```

```
  DEFINE DATABASE Profile Database AS profile-db

  DEFINE DATABASE Creation Database AS creation—db
  DEFINE DATABASE Assignment Database AS assignment-db
  DEFINE DATABASE Completion Database AS completion-db
  DEFINE DATABASE Notification Database AS notification-db

  DEFINE DOMAIN Customer AS customer
     DEFINE SERVICE Registration Service AS registration-service
     DEFINE SERVICE Profile Service AS profile-service

  DEFINE DOMAIN Ticketing AS ticketing
     DEFINE SERVICE Creation Service AS ticket-create-service
     DEFINE SERVICE Assignment Service AS ticket-assign-service
     DEFINE SERVICE Completion Service AS ticket-complete-service
     DEFINE SERVICE Notification Service AS notification-service

ASSERT(Registration Database is only accessed by Registration
Service)
ASSERT(Profile Database is only accessed by Profile Service)

ASSERT(Creation Database is only accessed by Creation Service)
ASSERT(Assignment Database is only accessed by Assignment Service)
ASSERT(Completion Database is only accessed by Completion Service)
ASSERT(Notification Database is only accessed by Notification
Service)
```

By passing this ADL into an AI agent, that agent can gain an understanding of the defined service-to-database relationships in the overall Sysops Squad architecture as illustrated in Figure 7-14).



This pseudocode describes a **microservices architecture** with strict database-per-service ownership. Here's how I'd interpret it:

Structure

Two domains, six services, six databases — each in a 1:1 pairing:

Domain	Service	Owned Database
Customer	Registration Service	Registration DB
Customer	Profile Service	Profile DB
Ticketing	Creation Service	Creation DB
Ticketing	Assignment Service	Assignment DB
Ticketing	Completion Service	Completion DB
Ticketing	Notification Service	Notification DB

Key Architectural Pattern

The `ASSERT` statements enforce the **database-per-service pattern** — each database is private to exactly one service. No service may reach into another service's database directly. This is a hard boundary.

This means:

- **No shared tables** across services
- **No direct SQL joins** across domains
- Inter-service communication must go through APIs or events, not DB queries

Domain Groupings

- **Customer domain** handles identity: registering new users and managing their profiles. These are likely independent lifecycles (registration is a one-time event; profile is ongoing).
- **Ticketing domain** models a ticket workflow as discrete stages — create → assign → complete — each with its own service and state store. Notification sits alongside as a cross-cutting concern within the domain.

Figure 7-14. AI agent interpretation from the ADL describing a microservices architecture using the dedicated database topology for the Sysops Squad system

As before, teams might be tempted to describe the physical bounded contexts (the one-to-one service-to-database relationship) as part of the service definition. However, because teams also occasionally need to share data between microservices (as described in [Link to Come]), we advise against this practice. Use assertions instead to describe these relationships.

Microservices and Domain Data

Microservice architectures can also use the domain database topology, although teams should have proper metrics in place to validate it (we discuss this in detail in Chapter 9). In this topology, shown in Figure 7-15, small groups of services (usually two to six) share a common database.

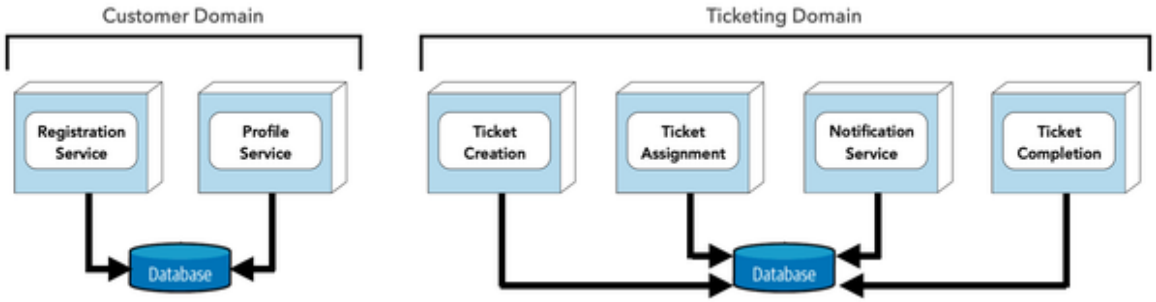


Figure 7-15. A domain database topology with a microservices architecture

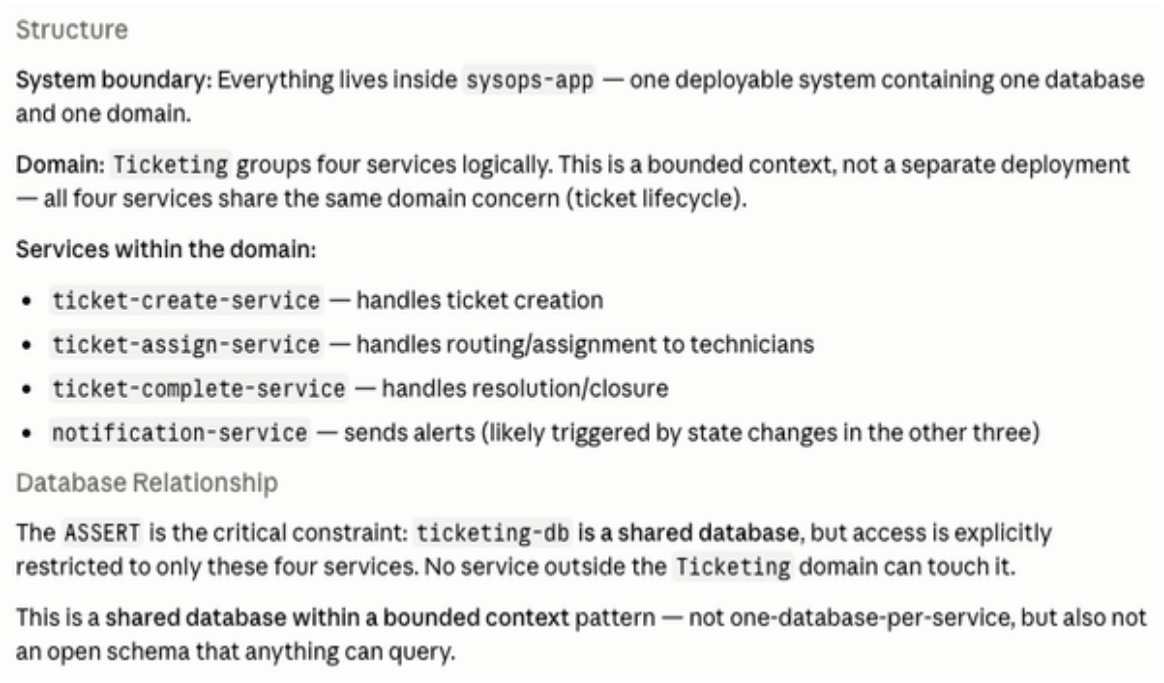
When leveraging this architectural style/database topology combination, teams typically use a *domain-based* repository to associate all the domain-related microservices in a single place. A domain-based repository is a single repository containing all of the source code for services and other artifacts contained within that domain. Not only does this help align teams with domains (see [Link to Come] on team topologies), it also helps with managing database structural changes. Teams can use ADL to specify the system's logical *and* physical architectures by defining the services within the domain and the corresponding domain database in a single ADL file. Example 7-7 shows this, though for brevity only the ticketing domain is described here.

Example 7-7. ADL describing the microservices-based Sysops Squad system for services within the ticketing domain using a domain database topology

```
DESCRIPTION Define Microservices and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app

  DEFINE DATABASE Ticketing Database AS ticketing-db

  DEFINE DOMAIN Ticketing AS ticketing
     DEFINE SERVICE Creation Service AS ticket-create-service
     DEFINE SERVICE Assignment Service AS ticket-assign-service
     DEFINE SERVICE Completion Service AS ticket-complete-service
     DEFINE SERVICE Notification Service AS notification-service
```

```
ASSERT(Ticketing Database is only accessed by {Creation Service,
       Assignment Service, Completion Service, Notification
Service})
```

Passing this ADL into an AI agent yields the correct understanding shown in Figure 7-16 of the service-to-database relationship with the four Sysops Squad ticketing services sharing a domain database.



Structure

System boundary: Everything lives inside `sysops-app` — one deployable system containing one database and one domain.

Domain: `Ticketing` groups four services logically. This is a bounded context, not a separate deployment — all four services share the same domain concern (ticket lifecycle).

Services within the domain:

- `ticket-create-service` — handles ticket creation
- `ticket-assign-service` — handles routing/assignment to technicians
- `ticket-complete-service` — handles resolution/closure
- `notification-service` — sends alerts (likely triggered by state changes in the other three)

Database Relationship

The `ASSERT` is the critical constraint: `ticketing-db` is a shared database, but access is explicitly restricted to only these four services. No service outside the `Ticketing` domain can touch it.

This is a shared database within a bounded context pattern — not one-database-per-service, but also not an open schema that anything can query.

Figure 7-16. AI agent interpretation from the ADL describing a microservices architecture using the domain database topology for the Sysops Squad system

Not all services within a domain have to share the same database. For example, the `Creation` service and the `Completion` service can share a database, whereas the `Assignment` service and `Notification` service each have a dedicated database, as illustrated in Figure 7-17.

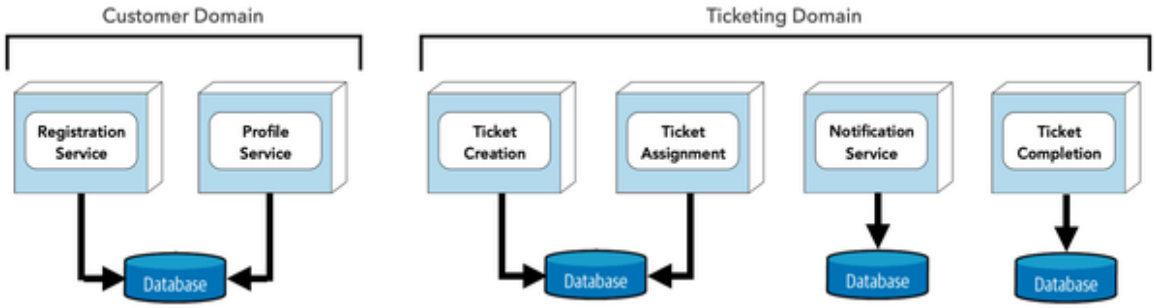


Figure 7-17. A hybrid domain database topology and dedicated database topology with a microservices architecture

The ADL for this service-to-database relationship is shown in Example 7-8 (assuming a domain-based repository).

Example 7-8. ADL describing the microservices-based Sysops Squad system for services within the ticketing domain using a hybrid domain database and dedicated database topology

```
DESCRIPTION Define Microservices and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app

  DEFINE DATABASE Ticketing Database AS ticketing—db
  DEFINE DATABASE Assignment Database AS assignment-db
  DEFINE DATABASE Notification Database AS notification-db

  DEFINE DOMAIN Ticketing AS ticketing
     DEFINE SERVICE Creation Service AS ticket-create-service
     DEFINE SERVICE Assignment Service AS ticket-assign-service
     DEFINE SERVICE Completion Service AS ticket-complete-service
     DEFINE SERVICE Notification Service AS notification-service

ASSERT(Ticketing Database is only accessed by Creation Service
       and Completion Service)
ASSERT(Assignment Database is only accessed by Assignment Service)
ASSERT(Notification Database is only accessed by Notification
Service)
```

Passing this ADL into an AI agent yields a correct understanding by the agent of the service-to-database relationship, with only two of the services sharing a domain database and the remaining services using a dedicated database topology as shown in Figure 7-18 .

Service-Database Relationships

The pseudocode defines a **database-per-service** pattern (a microservices best practice), with one exception: the Ticketing Database is shared between two services.

Ownership Map

Database	Owning Service(s)	Notes
`ticketing-db`	`ticket-create-service`, `ticket-complete-service`	Shared by two services
`assignment-db`	`ticket-assign-service`	Exclusive ownership
`notification-db`	`notification-service`	Exclusive ownership

What the ASSERTs Enforce

The `ASSERT` statements act as **architectural constraints** — they declare that no other service may touch those databases. This is common in C4/ADR-style documentation to prevent accidental coupling.

Notable Design Decision

`ticket-assign-service` has no direct database access to `ticketing-db`, which means it must get ticket data through the `ticket-create-service` API (or an event/message bus) rather than querying the ticketing database directly. Same for `notification-service` — it presumably receives events rather than polling other databases.

The one tension worth flagging: **two services share** `ticketing-db`. This is a pragmatic compromise (ticket lifecycle spans creation → completion), but it creates coupling between those two services at the data layer. If the schema changes, both services are affected. A stricter design would give each service its own store and sync state via events.

Figure 7-18. AI agent interpretation from the ADL describing a microservices architecture using using a hybrid domain database and dedicated database topology

Combining a microservices architecture with a domain database topology might seem strange to some readers, especially since microservice architectures typically use the dedicated database topology. However, there are many reasons to leverage this approach (or the hybrid one shown in Figure 7-17), including avoiding interservice communication for purposes of obtaining data from another bounded context, improving data consistency, improving data integrity, and preserving the semantic entity context within the data model. Chapter 10 describes all of these considerations in detail.

Moving Forward

In this chapter we've demonstrated how to *describe* database topologies and service-to-database relationships using ADL. In the next chapter, we show how to *apply* ADL, in conjunction with agentic AI, to build automated governance that ensures that the implemented topology matches the intended architecture.

---

1 This and all other AI agent examples in this chapter were generated by a Claude Code Sonnet agent.

2 While we are associating the name of the system with the directory name, in some programming languages (such as C#), this will point to the namespace instead.

3 See Fundamentals of Software Architecture, 2nd Edition (O'Reilly, 2024) for more information about these architectural styles.

4 For brevity, we've omitted the components in this and all other ADL code listings in this chapter.

Chapter 8. Governing Data Using ADL and AI Agents

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

In Chapter 7 we demonstrated how to define service and database relationships through ADL. In this chapter, we show how to use that ADL to *govern* the data topology.

Teams can use ADL in conjunction with AI agents and fitness functions to govern both of these areas by leveraging information in the execution platform. Recall from [Link to Come] that teams can create two kinds of fitness functions: *interactive* fitness functions and *empirical* fitness functions. Teams can use AI agents in both types of fitness functions.

Interactive fitness functions are ones that leverage AI agents to ask questions about the environment to validate the architecture based on ADL files. The AI agents will scan the environment and provide the team with an answer, but it might be wrong since it's largely non-deterministic. This type of fitness function is illustrated in Figure 8-1, where the AI agent is accessing the execution platform directly.

Empirical fitness functions, on the other hand, can still leverage AI agents, but these AI agents access deterministic source code—usually implemented in the form of MCP tools—to provide teams with an answer based on ADL files. The deterministic MCP tools that make up the empirical fitness functions can be either written by humans, or generated by AI agents *and verified* through humans. This type of fitness function is illustrated in Figure 8-1, where the AI agent is accessing empirical MCP tools rather than the execution platform directly.

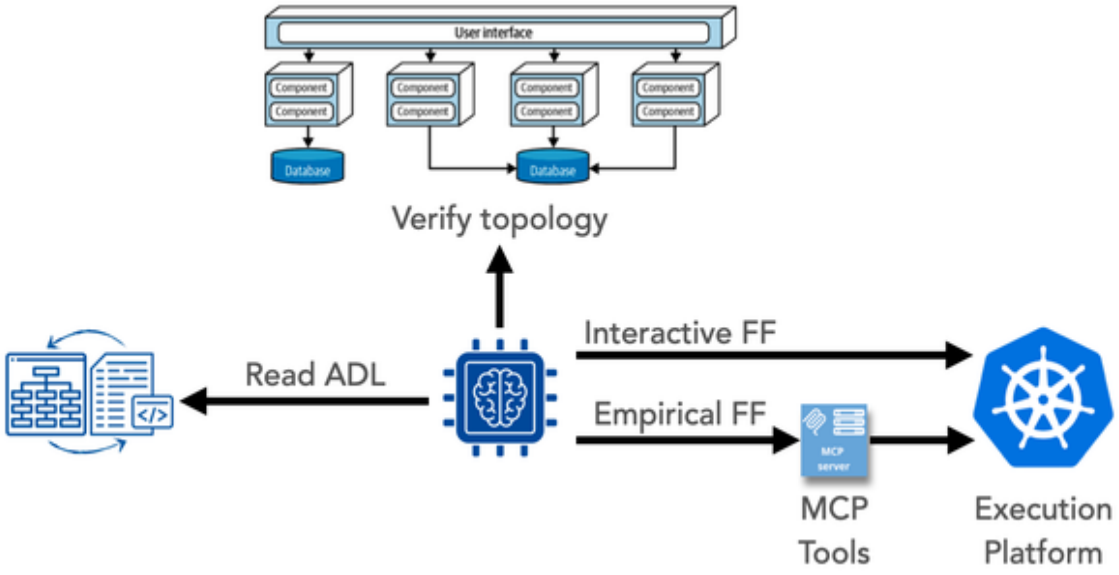


Figure 8-1. AI Agents can compare the architecture defined through ADL with the execution environment to validate the architecture

Governing data topology is about ensuring structural compliance, meaning the implemented data topology matches the defined architecture. For the domain database and dedicated database topologies, governance is also about ensuring that data is consistent between various distributed databases. The following sections describe each of these governance areas.

Structural Compliance

The ADL in the prior chapter used assertions to define the relationship between the services and databases. Those assertions essentially defined the physical bounded context between services and the corresponding data they

are allowed to access. Teams can therefore use the ADL alongside agentic AI to ensure database access constraints.

Without this level of governance, services might connect to databases in another bounded contexts to get data, potentially degrading the scalability and responsiveness of that database. Even worse, when teams make database schema changes to a database, those changes could negatively impact services directly connecting to that database from another bounded context. Preserving the physical bounded context between services and their corresponding data is essential for creating a robust and working architecture, particularly in microservices.

An AI agent can get information about service-to-database access and compare it against the ADL to determine if the data topology is in compliance using one or more of the following techniques:

Static analysis

> Static analysis techniques include leveraging static artifacts such as configuration files, Helm Charts, Kubernetes manifest files, and also basic source code scanning to obtain connection information. Agents may choose to use this technique to match the logical name of the database (such as `sysops-db`) and services (such as `ticket-service`) to various configuration settings as an initial check to look for violations—services or systems connecting to databases they shouldn't be connecting to—according to the assertions in the corresponding ADL.

Dynamic analysis

> In addition to the static analysis, AI agents may also choose to leverage dynamic runtime service-to-database information to determine correct database access. For environments running a service mesh (such as Istio or Cilium), AI agents can interrogate the service-to-database mapping in the proxy logs already provided by the envoys deployed as part of a service mesh. AI agents may also analyze runtime service-to-database connectivity through commands such as `netstat` and other network

observability tools to see which service addresses are communicating with database addresses. This technique would require knowledge of the actual network address of both the service and the database, something ADL doesn't provide, but CALM does. AI agents scanning the network can also likely figure out this information.

Platform analysis

Many database and networking tools expose database access information natively through the platform itself. AI agents can query observability information from database platforms such as Google Cloud SQL and Amazon RDS, secrets managers like Vault, and also DNS service discovery tools.

For interactive fitness functions, an AI agent will scan the execution environment and decide for itself which techniques to use to obtain service-to-database communication. For empirical fitness functions, development teams will usually create a set of MCP tools by writing (or generating) source code using one or more of these techniques for their specific environment that an AI agent can execute.

To demonstrate these techniques, suppose the Sysops Squad development team decided to use a domain database topology using microservices for the system, defining a `Customer` domain and a `Ticketing` domain as illustrated in Figure 8-2.

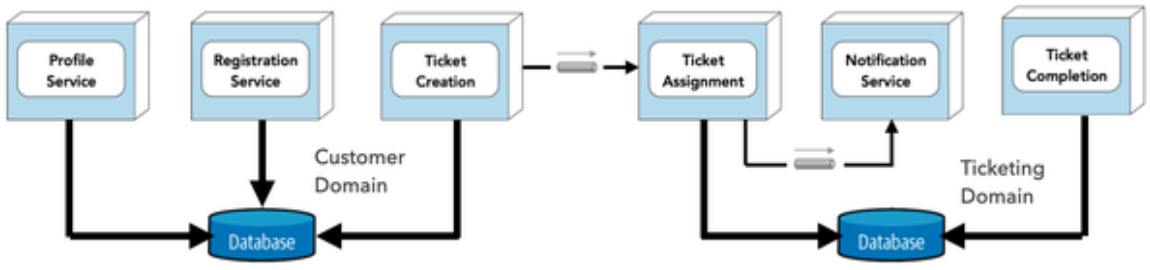


Figure 8-2. Sysops Squad implemented using a domain database topology with microservices

The corresponding ADL for the architecture and data topology illustrated in Figure 8-2 is described in Example 8-1.

Example 8-1. ADL describing the Sysops Squad system using a domain database topology with microservices

```
DESCRIPTION Service and Database Relationships
DEFINE SYSTEM Sysops Squad App AS sysops-app

  DEFINE DATABASE Customer Database AS customer-db
  DEFINE DATABASE Ticketing Database AS ticketing—db

  DEFINE DOMAIN Customer Domain AS customer
     DEFINE SERVICE Registration AS registration-service
     DEFINE SERVICE Profile AS profile-service
     DEFINE SERVICE Ticket Creation AS ticket-creation-service

  DEFINE DOMAIN Ticketing Domain AS ticketing
     DEFINE SERVICE Assignment AS assignment-service
     DEFINE SERVICE Notification AS notification-service
     DEFINE SERVICE Ticket Completion AS ticket-completion-service

ASSERT(Customer Database is only accessed by Customer Domain
Services)
ASSERT(Ticketing Database is only accessed by Ticketing Domain
Services)
```

Teams can now create MCP tools using any of the above analysis techniques that an AI agent can access to obtain the service-to-database mapping, and then compare it against the ADL to verify the implementation matches the intended data topology.

For example, for static analysis we created an MCP tool written in Python named `get_service_db_config()` for the Sysops Squad located in the `mcp_tools` directory of the Sysops Squad Distributed Repository that scans all of the deployment configuration files within the Kubernetes deployment environment, creating a map of service-to-database connections:

```
  # Sysops Squad MCP tool to statically map service-to-database
  # source code located in sysops_mcp_server.py
  get_service_db_config() ->
      the database each service is CONFIGURED to use, from
      its Kubernetes Deployment (static analysis).
  ...
```

We also created a dynamic analysis MCP tool named `get_db_connections()` that uses either Hubble or Istio flow analysis to illustrate the dynamic runtime service-to-database communications mapping in the same repository:

```
# Sysops Squad MCP tool to dynamically map service-to-database
# source code located in sysops_mcp_server.py
get_db_connections() ->
    the service-to-database connections actually OBSERVED
    at runtime (dynamic analysis). Backend selected by
    FLOW_SOURCE: 'hubble' (Cilium eBPF flows, with the
    FORWARDED / DROPPED verdict) or 'istio' (Istio TCP
    telemetry via Prometheus, no drop verdict).
...
```

Regardless of the technique used to get the service-to-database connection information, matching the results against the corresponding ADL is the key. Without it, the human or AI agent would have no guidance or information as to what the data architecture (and corresponding data access) *should* be, and would be unable to properly govern database communication.

Data Consistency

When breaking apart data, teams must also break apart data integrity and consistency constraints such as foreign keys. Because of this, teams can also verify data topology governance by writing fitness functions to ensure data remains consistent between databases, and raising alerts if they are not in sync with each other.

Referring back to the Sysops Squad architecture as illustrated in Figure 8-2, recall that once a ticket is created, the `Ticket Creation` service asynchronously sends it to the `Ticketing` domain via a queue for processing. From a governance perspective, teams can write fitness functions or use AI agents to verify that tickets in the `customer-db` match those in the `ticketing-db`. Tickets not in sync could result from a number of conditions as outlined below:

Asynchronous communication

Using asynchronous communication between domains or services can possibly result in lost messages, particularly if the receiving service crashes or experiences a fatal error after dequeueing a message. In our Sysops Squad example shown in Figure 8-2, because the `Ticket Creation` service in the `Customer` domain asynchronously sends the ticket to the `Ticket Assignment` service located in the `Ticketing` domain, there's a possibility of lost tickets, creating a data consistency issue between the two domain databases.

Teams can leverage the *Asynchronous Chain of Persistence* pattern to guarantee no messages are lost when using asynchronous communication. With this pattern, teams leverage *synchronous send* combined with *client acknowledgement* mode and *last participant support* to ensure that the message is always persisted while it's being processed. This pattern and these techniques are described in detail in our book *Software Architecture Patterns, Antipatterns, and Pitfalls* (O'Reilly, 2026).

Behavioral error conditions

Process-related (behavioral) error conditions not reported or communicated back to the other domain can create data consistency issues. For example, suppose the `Ticket Assignment` service receives a problem ticket and determines that the specific customer location for the ticket isn't supported by the Sysops Squad system. Normally, the `Ticket Assignment` service would raise an exception, which would (hopefully) be addressed by services within the `Ticketing` domain. However, because of this error, the `Ticket Assignment` service never inserted the ticket into the `Ticketing` database, resulting in a data consistency issue. This is only one example of the sort of process-related errors that can occur with regards to ticket processing that can cause databases to get out of sync.

One solution to this consistency problem is for the `Ticket Assignment` service to insert the ticket into the `Ticketing` database with a particular status indicating that it cannot be assigned to an expert, or send a notification back to the `Ticket Creation` service so it can delete the ticket from the `Customer` database and notify the customer that the location is not supported by the system.

Regardless of the examples shown here, when breaking apart data teams should be aware that data consistency issues can occur for a number of reasons, and create measurements using AI agents or fitness functions to verify that the databases are in sync.

Typical business processes

Another way data can get out of sync between databases is with typical business processes such as a customer cancelling a ticket. When a customer cancels a ticket, the `Ticket Creation` service (or another service within the `Customer` domain) would presumably delete the ticket from the `Customer` database. However, if the service performing the ticket cancel process doesn't inform the other domain of the cancellation, the databases would be out of sync with one another.

Teams can leverage fitness functions in the form of backend processes or AI agents that can compare databases based on certain idle conditions and report back if they are not in sync, augmenting the alert with information about which data might be missing from one of the databases. Note that these data-related fitness functions will not *solve* the problem, but rather provide a fast feedback loop that something is wrong based on any of the prior scenarios. Teams can then take action to fix the problem, ensuring the databases remain in sync.

One of the challenge of such governance is preserving the physical bounded context of each domain or service. Having an AI agent or external fitness function directly querying the database would break these physical bounded contexts, and make those fitness functions brittle from a database change

perspective. Using MCP tools and a dedicated MCP server within that bounded context helps address this problem.

Using our Sysops Squad example, teams can create a set of MCP tools to govern and verify data consistency, including tools such as calculating the hash values of all ticketing keys for comparison, checking the queue depth (an indication of an idle state), and also getting the keys when the hash values are different to determine which ones are missing from which databases. Using these MCP tools, teams can create AI agents to ask questions about database consistency in a somewhat non-deterministic (but more flexible) fashion, or write more deterministic fitness functions that perform these queries and calculations. Note that the MCP tools provide a universal way to access this information and can be used by both AI agents as well as deterministic fitness functions.

Figure 8-3 shows an example of how to use MCP tools to execute various data consistency fitness functions.

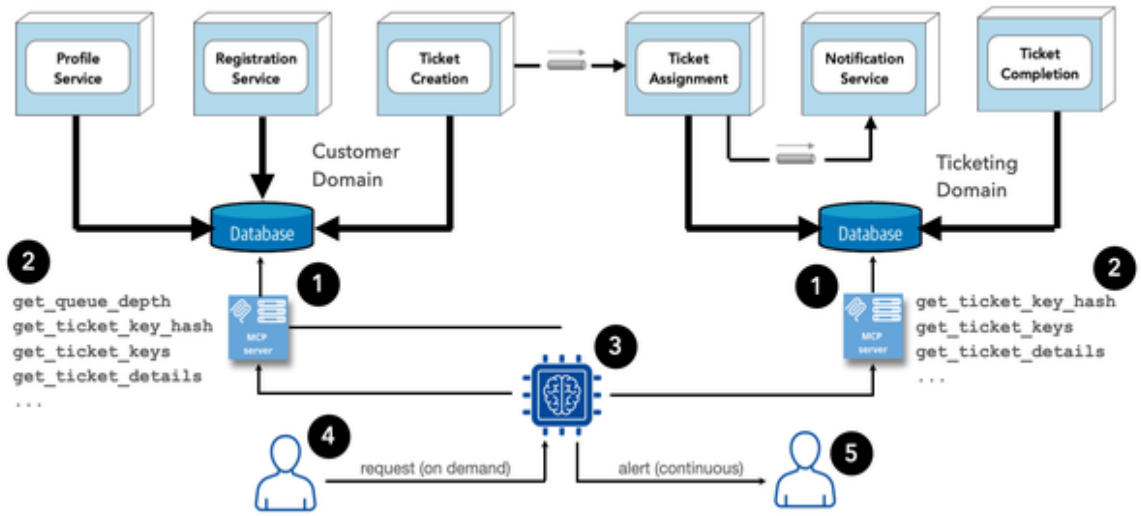


Figure 8-3. Using MCP tools to provide universal access to either AI agents or deterministic fitness functions

The following numbers refer to the diagram illustrated in Figure 8-3:

1. Notice in the diagram that each domain has its own MCP server. This practice preserves the bounded context of each domain with respect to database access and database structure changes. A single

MCP server hosting a single set of Sysops Squad MCP tools would effectively be coupling these domains, meaning a change to a database in either domain would impact the single set of MCP tools. Aligning MCP tools to a particular domain or set of services also avoids messy switching and conditional logic in those tools in terms of which database it's connecting to.

2. Each domain has a set of MCP tools that are specific to that domain, including the database connection strings, the database structure, and even the type of database used. For example, the `Customer` database may use a relational database, whereas the `Ticketing` database may use a document database. Notice that the MCP tool that access the queue depth (`get_queue_depth`) used to determine an idle state is only available through the set of MCP tools within the `Customer` domain. This is because the `Ticket Creation` service must write to the queue, meaning queue is essentially contained within the `Customer` domain (and hence those set of MCP tools).

3. Teams can implement the data consistency fitness function check as an AI agent, or through deterministic fitness functions coded by the team and implemented through a governance mesh. Using AI agents provides a more flexible way of asking questions about database consistency based on the supported MCP tools available, but possibly in a non-deterministic way. On the other hand, writing custom fitness functions provides teams with a more deterministic answer, but are more brittle in terms of the coupling to the specific MCP tools and the algorithms used to determine data consistency. Another factor associated with the use of AI agents for these types of fitness functions is the cost associated with token usage when executing these fitness functions. Either way, the MCP tools abstract the fitness function from the underlying database and message broker. Changes to either of these physical artifacts only impact the MCP tools, not the underlying fitness functions. This is

an important aspect of data consistency governance since those fitness functions are usually cross-domain.

4. Teams can implement data consistency checks as a manual verification via a request, usually through some sort of dashboard or AI chat. For example, a human operator might make a request to the AI agent or fitness function to make sure all databases are in sync. Note that this method of data consistency governance can also be used in conjunction with the autonomous method described in item 5.

5. Most typically, teams will implement this type of data consistency check through a continuous fitness function (one running all the time in the background) or through a triggered fitness function based on some sort of periodic timing via a timer. Either way, with this autonomous method the AI agent or deterministic fitness function would wake up at some point, check to see that the system is in an idle state (via a queue depth check), and verify using the ticket key hash MCP tools that the databases are the same. If not, the AI agent or deterministic fitness function could then get all the keys and compare them to determine which tickets are missing from which databases. The fitness function would then send a human an alert via SMS Text and/or email, notifying them that the databases are not consistent.

To see examples of these MCP tools (written in Python) that an agent can access to validate data consistency, go to the `mcp_tools` directory of the Sysops Squad Distributed Repository and look at the following MCP tools:

```
# Sysops Squad MCP tools check for data consistency
# source code located in sysops_mcp_server.py

get_queue_depth() ->
    messages still sitting in the broker (RabbitMQ)
    queues; 0 = quiescent. Cross-domain handoff is
    asynchronous over the broker, so quiescence is
    "the broker is drained", not a DB queue table.
```

```
get_ticket_key_hash(domain) ->
    md5 of the SET of ticket keys in a domain's
    database ('customer' or 'ticketing'). When the
    broker is drained the two must be EQUAL.
...
```

Next Steps

Although we've shown examples using the domain database topology, these same governance methods apply to the dedicated database topology as well. However, when using the dedicated database topology, teams must apply additional logic to the fitness functions (either AI or deterministic) to know which dedicated databases must be in sync as not all dedicated databases share the same information.

In this chapter we demonstrated how to govern the domain and dedicated database topology in terms of both structure (database access) and correctness (data consistency). In the next chapter, Chapter 9, we'll demonstrate how to validate that the chosen database topology is the right one for the architecture given the business environment, business domain, and required architectural characteristics.

Chapter 9. Validating Data Architecture Using AI Agents and Fitness Functions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

Choosing the most appropriate database topology can be a challenging decision. Architects are faced with a number of questions when making a data topology choice, including:

- "Will our services scale enough to support our current and future customer load?"
- "Are we meeting our responsiveness needs?"
- "Are we struggling to keep the distributed databases in sync?"
- "Are we making too many interservice calls to retrieve data in another bounded context?"
- "Does the database topology match our data model?"
- "Is it too difficult to make structural changes to the data?"

Validating a data topology is about applying measurements to validate that it's the most appropriate choice for the architecture and problem domain, and what changes to the data topology architects might apply to ensure the success of the architecture. In this chapter, we provide architects with tools and techniques using AI agents that leverage both empirical and interactive fitness functions to validate the decisions architects make regarding the data topology.

Validating a data topology involves three main areas of measurements: structural validation, operational capabilities, and process-related capabilities. The following sections address each of these validation measurements.

Structural Validation

In the prior chapter (Chapter 8) on governing data topologies, we showed how to govern the data topology to ensure the implementation matches the physical architecture as described through ADL. While the data topology might be implemented correctly, it might not be the most appropriate one. From a structural perspective, one way architects can validate their data topology choices is to measure and track how many interservice calls are required to retrieve required data from another bounded context. Too many interservice calls between services to retrieve data can be an indication that the data topology might not be the right one for the data model and behavior of the system.

> **WATCH OUT FOR TOO MUCH INTERSERVICE COMMUNICATION**
>
> The more you break apart data, the greater chance other services will need to access that data through synchronous interservice communication.

Consider the simplified Sysops Squad system implemented as microservices using a domain database topology as shown in Figure 9-1.

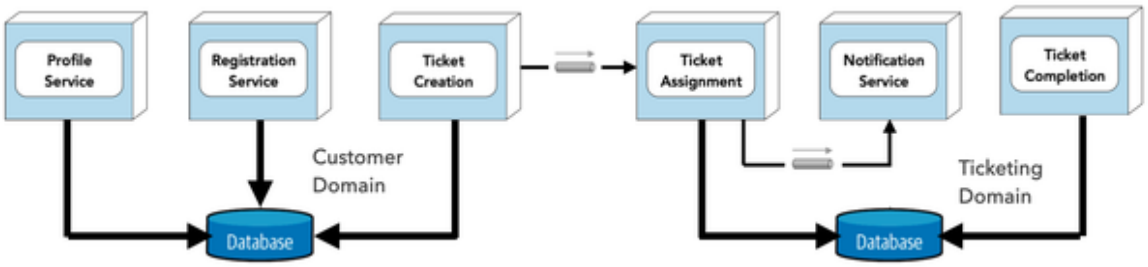


Figure 9-1. Sysops Squad implemented using a domain database topology with microservices

In this topology there's a clean separation between the `Customer` domain and the `Ticketing` domain due to the asynchronous call between the `Ticket Creation` service and the `Assignment` service. In this architecture each domain becomes it's own bounded context, meaning services in one domain cannot access the database in the other domain.

Suppose the `Notification` service needs the customer email address to let the customer know the Sysops Squad expert is on their way as shown in Figure 9-2. The customer contact information is located in the `Customer` database, which the `Notification` service cannot access directly because it's in a different bounded context. Therefore, it must ask a service in that domain—in this case the `Profile` service—for that information. Along the same lines, the `Ticket Assignment` service needs the customer location information so it can locate an expert in that area. Again, that information is located in the `Customer` database, meaning the `Ticket Assignment` service must ask a service in that domain for that information.

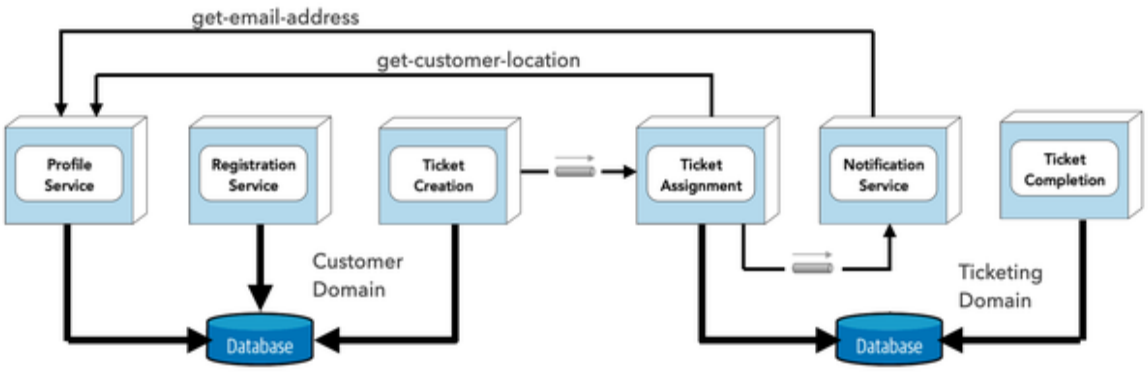


Figure 9-2. Interservice communication for data needed by services outside the bounded context

While these synchronous interservice calls preserve the bounded context of each domain, they can significantly degrade operational characteristics such as scalability, responsiveness, and fault tolerance.

From a scalability standpoint, when one service calls another one synchronously, the service being called must *at least* be able to scale to the level of the calling service. For example, if the `Notification` service must scale to 1000 concurrent users but the `Profile` service can only scale to 200 concurrent users, making a synchronous call to the `Profile` service means the `Notification` service can only scale to 200 concurrent users—far below its scalability requirements.

Responsiveness is also degraded because it's cumulative. The response time of the services being called must be added to the calling service's response time. Latency, in the form of network communication, additional database calls, security, marshaling and unmarshaling of data, and processing time all degrade the overall responsiveness of the original request.

Regarding fault tolerance, if the `Profile` service, located in a separate bounded context, suddenly becomes non-responsive or crashes, the `Notification` service cannot process the request. This type of synchronous dynamic coupling entangles the *architectural quantum* of each domain, thus creating a single architectural quantum. An architectural quantum, described in detail in our books *Software Architecture: The Hard Parts* (O'Reilly, 2022) and also *Software Architecture Patterns, Antipatterns, and Pitfalls* (O'Reilly, 2026), is a separately deployable part of the architecture that contains its own architectural characteristics, bound together through synchronous dynamic coupling. Since each architectural quantum contains it's own set of architectural characteristics, entangling the quanta means the collective architectural characteristics sync to the lowest level of each.

In Chapter 8 we showed how to use AI agents and empirical fitness functions to determine which services are connecting to which databases. Teams can use these same techniques to also determine and track service-

to-service communication for purposes of gaining access to data in another bounded context.

Architects can use ADL to define the *transactional sagas* of a system—how services communicate with each other through workflows. They can then validate through empirical or interactive fitness functions that the dynamic communication specified in the ADL matches the actual service-to-service communication. Any synchronous communication *outside* of the transactional saga definitions identified by this fitness function is likely due to accessing data from a different bounded context, which architects can manually investigate to confirm.

To illustrate this type of fitness function check, consider the architecture illustrated in Figure 9-1. This simplified Sysops Squad architecture contains two transactional sagas: one when tickets are created, and one when they are assigned. When tickets are created, the `Ticket Creation` service communicates asynchronously with the `Assignment` service. When a ticket is assigned, the `Assignment` service asynchronously communicates with the `Notification` service. Example 9-1 shows the corresponding ADL describing the communication flows from the diagram in Figure 9-1.

Example 9-1. ADL describing the Sysops Squad service service communication workflows

```
DESCRIPTION Dynamic Service Communication in the Sysops Squad
System
DEFINE SYSTEM Sysops Squad App as sysops-app

DEFINE SAGA Ticket Creation Saga AS
    DEFINE COMMUNICATION Ticket Creation TO Assignment AS
asynchronous
    DEFINE COORDINATION Choreography
    DEFINE CONSISTENCY Eventual

DEFINE SAGA Ticket Assignment Saga AS
    DEFINE COMMUNICATION Assignment TO Notification AS asynchronous
    DEFINE COORDINATION Choreography
    DEFINE CONSISTENCY Eventual

ASSERT(Communication between services adheres to the
       defined communication declarations)
```

Don't worry about the `DEFINE SAGA` format for now—we describe that in detail in [Link to Come]. Notice the ADL Example 9-1 does not include the additional `Notification` to `Profile` service communication nor the `Assignment` to `Profile` service communication shown in Figure 9-2. Teams can create MCP tools that an AI agent can use to get the existing service-to-service communication in the system, and compare it against this ADL to locate additional synchronous communication points.

We've included an example of an MCP tool agents can use to get service-to-service mappings and compare it against the ADL in the Sysops Squad Distributed Repository accompanying this book. Go to the `mcp-tools` directory in that repository and open the `get_communication_edges()` MCP tool contained in the `sysops_mcp_server.py` file to see an example:

```
# example Sysops Squad MCP tool to govern service communication
# source code located in sysops_mcp_server.py
get_communication_edges() ->
   the inter-service hops observed at runtime: async
   hops as broker channels (each queue+consumer), sync
   hops as direct service->service flows. The agent
   compares them to the declared sync/async hops.
...
```

Given the results of this empirical fitness function compared against the ADL by an AI agent, architects can now determine if the additional interservice communication (if any) is acceptable, and if so, they can add it to the `SAGA` workflow definitions in the ADL. However, if the architect determines this is not acceptable, they can adjust the bounded contexts by modifying the data topology to share data between services, use data schema replication, or leverage caching to hold the data. In the next chapter (Chapter 10), we describe these data access techniques in detail.

Operational Capabilities

Many systems have distinct operational capability requirements, such as meeting specific response time, scalability, and availability objectives. Service granularity, service communication, workflows, and the way teams implement behavior can all have an impact on operational characteristics. The database topology is no exception. Choosing the wrong database topology can impact operational characteristics by having too many services and too many requests accessing the same database, creating an operational bottleneck.

Operational measurements provide architects with valuable information to validate the scope of a shared domain database in terms of the number of services accessing it. One of the ways architects can determine whether the database is being overloaded with requests is by measuring *connection waits* on the database. A connection wait occurs when the database gets too overloaded and cannot accept any further requests. Services wishing to perform an action on the database must therefore wait until other connections free up.

Another way architects can validate the database topology from an operational standpoint is to track query and update response times to see if they degrade as the concurrent user load increases. Like connection wait metrics, this metric provides information as to whether the system can scale based on the load on the database.

Teams can write empirical fitness functions in the form of MCP tools that can obtain these metrics, and then use agents to execute those MCP tools and compare the results against assertions in the ADL to determine if the database connection waits and query times are within a specified tolerance range. Teams can access these metrics through one of two ways: through *platform-based observability*—basic observability within the execution platform through database tools and observability products, or through *service-based observability*—services that gather expose metrics information.

The following sections describe each of these approaches using the Sysops Squad example, focusing only on the `Customer` domain since it's

customer-facing and requires both responsiveness and scalability. Since the `Ticketing` domain doesn't have any responsiveness and scalability requirements, it's not necessary to gather metrics and measure them.

> **ONLY MEASURE WHAT'S IMPORTANT**
>
> Only measure those operational aspects of the environment that are important for the system and have specific quantified service-level objectives. For example, if responsiveness and scalability are not important capabilities of one area of the system (such as administrative behavior), then there's no need to spend the effort and time to create fitness functions to measure and analyze it.

Platform-Based Observability

With the platform-based observability approach, the execution platform provides the AI agents with database observability metrics, usually through a set of empirical MCP tools. AI agents can then compare the results from the MCP tools with the directives specified in the ADL to determine whether the connection waits or database query times are within the specified limits as shown in Figure 9-3

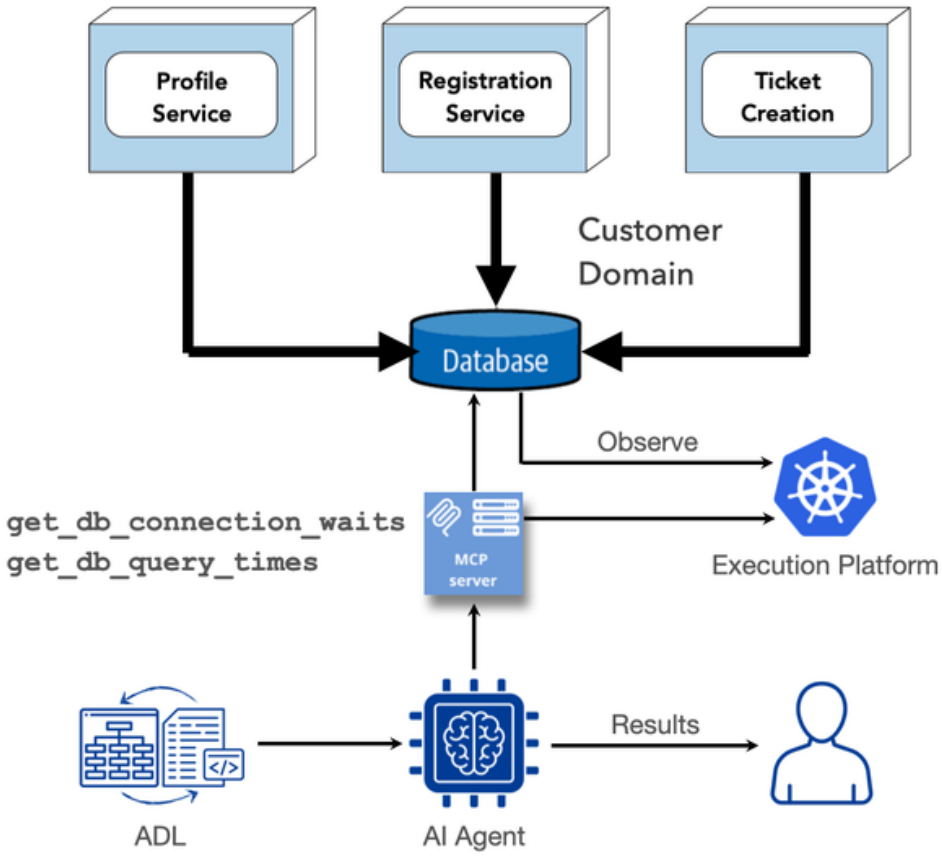


Figure 9-3. MCP tools access the platform to get database observability metrics

Notice in the diagram that the MCP tools can gather database connection wait and query time metrics either from the database itself, or through observability tools within the execution platform. Some database vendors provide an API to get this information, such as Google Cloud SQL and Amazon RDS, or the execution platform can monitor the database to provide this information.

Providing the AI agent with directives as to whether the metrics satisfy the requirements is the real key. Without these directives, the AI agent can observe these metrics but has no way of knowing whether those metrics show a potential problem. Teams can use ADL to specify the database connection waits and query times using ADL as shown in Example 9-2.

Example 9-2. ADL describing the Sysops Squad database capability requirements

```
DESCRIPTION Define Database Capabilities For the Customer Database
DEFINE SYSTEM Sysops Squad App as sysops-app
```

```
ASSERT(Customer Database has fewer than 50 connection waits over a
24 hour period)
ASSERT(Query time in the Customer Database is less than 25ms)
```

Notice the assertions in the ADL specify the tolerance levels for both connection waits and database queries. The architect could easily have indicated the following assertion if no connection waits are tolerated:

```
...
ASSERT(Customer Database has no connection waits over a 24 hour
period)
```

Architects can specify this capability requirement in a separate ADL file as we show in Example 9-2, or when the database is defined:

```
...
DEFINE DATABASE Customer Database AS customer-db
...
ASSERT(Customer Database is only accessed by SERVICES in the
Customer Domain)
ASSERT(Customer Database has no connection waits over a 24 hour
period)
ASSERT(Query time in the Customer Database is less than 25ms)
```

We prefer to separate the *static coupling* definitions—those specifying the service-to-database relationships—from the *capabilities* definitions—the quantified architectural characteristics and constraints—and put these in separate ADL files as they represent different aspects of the architecture. Either way, these directives in the ADL not only specify the quantified architectural characteristics of the architecture, but also provide the AI agent executing the empirical fitness function with guidance as to whether something is wrong or not.

DON'T FORGET THE PURPOSE OF ADL

While this chapter focuses on the physical architecture and corresponding operational aspects of the system, don't forget that the purpose of ADL is to describe—through code—the entire architecture, including the logical architecture (domains, components, and their interactions) and the physical architecture (services, databases, and capabilities).

Leveraging ADL and AI agents, architects can use these techniques to gain visibility into the system to determine of the database is adhering to the quantified capabilities of the system, and make modifications to the database topology if it's not meeting these requirements.

Service-Based Observability

Another approach architects can use to gather database observability metrics is through *service-based observability*. This platform-agnostic approach places the responsibility of metrics gathering on the service rather than the platform. With service-based observability, specific services (such as the `Ticket Creation` service shown in Figure 9-4) gather their own metrics, and make those metrics available through some external means, usually through streaming.

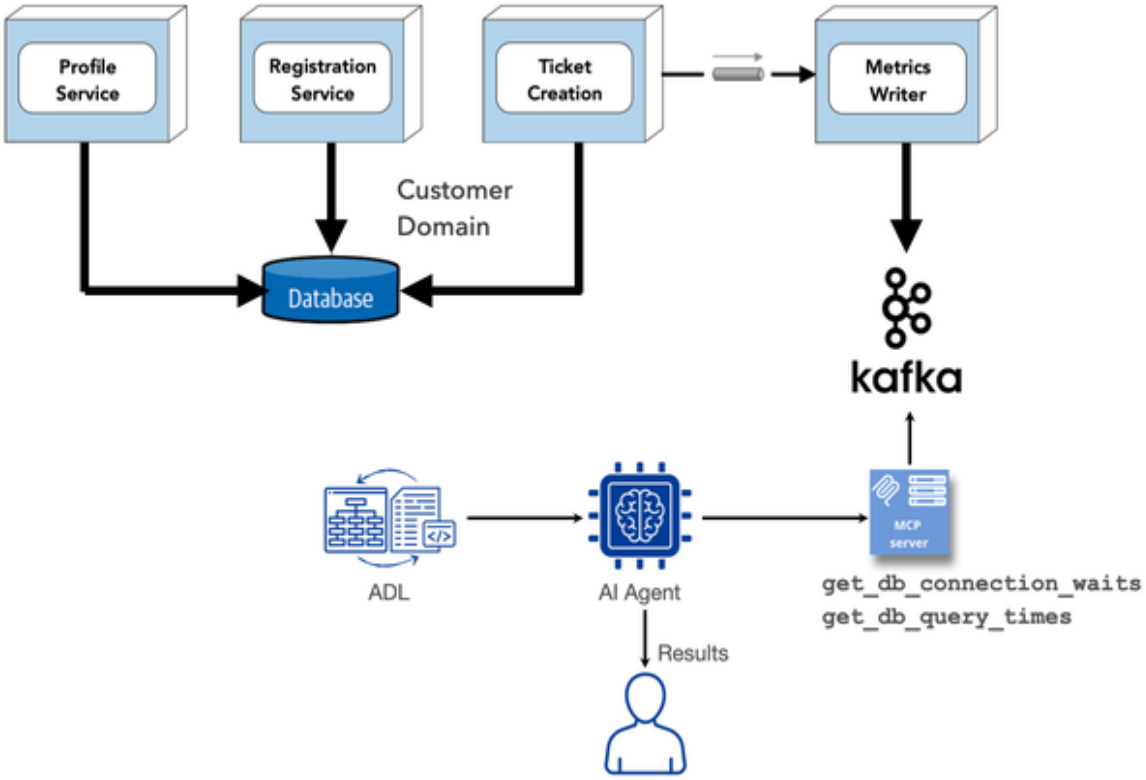


Figure 9-4. Services provide database observability metrics through streaming

Notice here we chose to use a `Metrics Writer` to take responsibility for streaming metrics to the Kafka cluster. This practice maintains the architectural quantum for each domain streaming metrics information. For example, without an asynchronous writer service, services in the `Ticketing` domain also streaming information to the same Kafka cluster would effectively be entangling the `Customer` and `Ticketing` architectural quanta. This is because the Kafka cluster isn't owned by either quantum, and services are synchronously coupled to it. By leveraging a separate writer service and having services asynchronously send metrics information to it, the architectural quanta are preserved in each domain.

Using this approach, the services exposing observability metrics would require additional logic to create timers surrounding connections and queries, and, if over a specific value, asynchronously send those to a queue for streaming as shown in the following pseudocode:

```
var start_time = get_time()
get_connection()
```

```
var duration = get_time() - start_time
if duration > 25ms
   send_metrics(duration)
endif
```

The astute reader will quickly observe the problem with this approach: the tolerance range for connection waits is hardcoded in the service (in this case 25ms), meaning the ADL directives, if specified, can quickly get out of sync with the fitness function check. One approach for addressing this hardcoding issue is to have the service stream all of the metrics, regardless of the connection time, and have the AI agent compare those values against the ADL to look for violations:

```
var start_time = get_time()
get_connection()
send_metrics(get_time() - start_time)
```

The trade-off of this approach is it produces a massive amount of streamed data. Agents would therefore need to process and analyze many more data points to determine if the connection wait times or query times have been exceeded.

Regardless of which technique teams use, notice that the AI agent in Figure 9-4 still uses MCP tools to get the observability metrics (in this case from topics in the Kafka cluster). This approach decouples the agent from having to know *where* the metrics are, only that it must access the MCP server and execute the correct tool (in this case `get_db_connection_waits`). In this manner teams can alter the empirical MCP tools to point back to the execution platform to gather metrics without changing the agent or actual fitness function.

Choosing an Observability Approach

Which is the right approach: using platform-based observability to gather metrics from the execution platform, or using service-based observability to have services gather and expose the metrics?

The first consideration an architect should take into account is whether the execution platform has visibility to the observability metrics they are interested in. Most execution platforms have observability tools to measure basic operational characteristics such as responsiveness, scalability, service uptime, and availability. However, some observability metrics are domain-specific and behavioral, thus requiring services to expose that information.

For example, suppose an architect needs observability metrics to capture the number of Sysops Squad tickets that weren't able to be assigned within 48 hours of creation. This domain-specific information isn't available in an execution platform, and necessarily a service (such as the `Assignment` service in our Sysops Squad example) would need to make that information available through service-based observability instead.

Another consideration an architect should take into account is the scoping of observability metrics. Take, for example, the connection wait metrics described in "Service-Based Observability". The execution platform might know that the `Customer` database had experienced 300 connection waits over the past 24 hours, but has no knowledge of which service experienced those connection waits. Observability metrics targeting specific services (like the `Ticket Creation` service) sometimes requires service-based observability to obtain those fine-grained targeted metrics.

Process-Related Capabilities

Process-related capabilities are those that relate to developing and releasing software, and include architectural characteristics such as maintainability, testability, and deployability, all of which combine to form a composite characteristic known as *agility*—the ability to respond quickly to change. Making an incorrect choice about the data topology can significantly degrade agility, and as such is an important observability metric architects can use to validate their data topology choices.

One observability metric architects can use is how often *database structural changes* occur. Database structural changes are those changes impacting the

schema or structure of the database. Examples include changes like dropping an attribute from a key-value database, or dropping a column from a table in a relational database. The frequency of database structural changes isn't the only metric that matters, but rather that metric in combination with the *scope* of the database change—in other words, the number of services impacted by the database change and how difficult it is for teams to apply those changes.

Consider two of database topologies we introduced in Chapter 6, the monolithic database and dedicate database. With the monolithic database topology, all services share the same database, meaning a structural change in that database will likely impact multiple services since the data is shared. The effort in changing the database schema requires the team to determine which services are impacted by the database change, and applying changes to those services at the same time. This, of course, assumes that teams aren't doing other maintenance on those services—quite an undertaking to say the least.

Now consider the other extreme—the dedicated database topology. Here, because of the tight bounded context formed between the service and its corresponding data, changing the structure of the database only impacts the service (or services) within that bounded context. This requires much less effort than the monolithic database topology, and hence teams can be much more agile when applying database changes.

It's the combination of the frequency of database changes and the number of services impacted that forms the fitness function to validate the database topology. Here are the possible combinations architects should be aware of, and the corresponding action they should take for each:

Occasional database change and few impacted services

> This combination of metrics is the most optimal. The database structure is fairly stable, doesn't change often, but when it does, not many services need to change. Given this combination, architects can relax, knowing the data topology is sound from a process-related standpoint.

Occasional database change and many impacted services

The database may only occasionally change, but when it does, lots of services are impacted. This should be a warning sign for architects. Because the database only occasionally changes, it might be okay, but architects should take note of the shared database topology and keep an eye on the frequency of database changes.

Frequent database change and few impacted services

Some databases undergo constant change due to new features and existing feature maintenance. If the number of services impacted is low, this is a good indication that the database topology is likely a good choice. However, architects should also keep an eye on this combination to make sure the number of services impacted doesn't increase over time.

Frequent database change and many impacted services

This combination should raise a red flag to the architect, indicating that this is not the right database topology for this combination. Here, a frequently changing database with many service changes indicates that the shared database topology is likely too tightly coupled, and agility will be at it's lowest. Breaking apart the data to isolate the changes and having fewer services accessing the database is a likely response for this combination.

This type of fitness function typically does not have corresponding ADL to describe these constraints. Rather, teams can leverage the ADL, specifically the ADL describing the service-to-database relationships, to better understand the data topology and make modifications to improve it based on operational and process-related capabilities. Also, many of these metrics are typically manually gathered and stored, unless teams can utilize tools within their CI/CD pipeline to capture this information through deployment scripts or tools containing knowledge of what artifacts are deployed for a particular feature.

Summary

Architects can use the techniques described in this chapter to validate their choices regarding the database topology by answering three important questions:

1. Structural validation: “How many interservice calls are needed based on my data topology choices?”
2. Operational validation: “Am I stressing the database based on the number of services sharing the data?
3. Process validation: “What impact do database schema changes have on the system, and how easy it is to make those changes?”

By focusing on these questions and applying corresponding empirical or interactive fitness functions, architects can validate whether they have the most appropriate data topology in place given the business environment, architectural style, and capability constraints.

In the final chapter of this part (Chapter 10), we provide additional guidance for architects in terms of which data topologies are best aligned with which architectural styles, and how to address many of the messy situations we described in this chapter, such as avoiding interservice communication when using distributed databases and optimizing process-related capabilities.

Chapter 10. Why This Intersection is Important

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *sgrey@oreilly.com*.

For an architecture to work and achieve its desired goals it must be properly aligned with the right data topology. Architects can utilize the techniques we demonstrated in the prior chapters to describe their data topology and validate it using AI agents, ADL, and fitness functions, However, those fitness functions won't tell an architect what the most appropriate data topology *should* be—only that there's a problem.

In this chapter we take a deep dive into the considerations architects should take into account when deciding on a data topology to make sure it's properly aligned with the architecture in the first place—then use ADL to describe, govern, and validate their choices.

Architects should take into account four main architectural capability areas when deciding on the most appropriate data topology for their system: process-related capabilities, structural capabilities, data-related capabilities, and finally operational capabilities. Each of these capability areas are described in detail in the following sections.

Process-Related Considerations

Process-related considerations include those architectural capabilities that have to do with the creation and ongoing maintenance of the underlying databases and overall data topology, and include such things as the overall cost of database creation and maintenance, the overall simplicity of the database topology, and the ease in which teams can apply database structural changes.

These architectural capabilities are unfortunately often overlooked as architects get immersed in the operational and technical aspects of the system. However, making the wrong choice of a database topology based on the business environment, problem domain, and architectural style can have devastating effects on the success of the overall system.

Cost

Cost includes the initial time and effort involved with implementing the database topology, time and effort involved with ongoing maintenance in terms of the adaptability and evolvability of the system, and also the database licencing fees incurred through the database topology.

Not surprisingly, the monolithic database topology generally offers the lowest cost option out of the three topologies due to the use of a single consolidated database. Licensing and maintenance costs tend to be lower that with the domain database topology and the dedicated (database-per-service) topology due to only having one central physical database. Modeling costs also tend to be lower than the other topologies because data doesn't need to be physically separated, which is often a daunting task (see "Preserving Semantic Data Coupling").

Overall costs tend to rise significantly as teams move towards a dedicated database topology, mostly due to the effort to model fine-grained data (see "Preserving Semantic Data Coupling") and database licencing costs.

Table 10-1 shows the qualitative costs associated with each of the three primary database topologies.

Table 10-1. Qualitative comparisons for cost

Database Topology	Cost
Monolithic Database	$
Domain Database	$$$
Dedicated Database	$$$$$

Teams can somewhat lower the costs associated with the dedicated database topology by leveraging databases that support multiple schemas—isolated groups of tables that restrict access control—so that a single database contains multiple schemas. Teams can also lower the high costs associated with using dedicated databases by leveraging cloud-based on-demand databases such as Amazon's DynamoDB.

Simplicity

Having a single centralized database is the least complex configuration for most monolithic and distributed systems. Configuration, optimization, maintenance, backup, disaster recovery, and modeling are all easier with a single database. For systems with clear, well-defined independent domains, the domain database adds a little bit of complexity, but not by much (only one star). However, when using the domain database topology we caution architects to pay close attention to the domain boundaries to ensure the data is well delineated within the domain. Cross-domain data modeling and having too many domains in the system tends to increase the complexity of the domain database topology, thus degrading the qualitative stars even further.

As expected the dedicated database is the most complex of the three primary topologies, earning it only one star (not good) for simplicity. Must

of this complexity is associated with the data modeling efforts required to break apart data, combined with having to balance dedicated data with service granularity.

The dedicated database topology is most affected when teams consolidate services to make them more coarse-grained or split them apart into more finer-grained services. When teams adjust service granularity, they typically also have to adjust the database topology as well. The domain database and monolithic database topologies are generally not affected as teams adjust service granularity.

Table 10-2 shows the qualitative ratings for each of the three primary database topologies with respect to overall simplicity.

Table 10-2. Qualitative comparisons for overall simplicity

Database Topology	Simplicity
Monolithic Database	★★★★★
Domain Database	★★★★
Dedicated Database	★

The fact that the domain database topology doesn't increase complexity much makes it a popular choice among the database topology choices.

Ease of Database Changes

Database tables and schemas are rarely static. New features in a system usually require a corresponding change to the physical database model, which in turn may impact services using the changed data structure.

Database change control refers to the level of difficulty in making a change to the structure of the data, whether it be changing a node or relationship in a graph database, or dropping a column in a relational one. It also refers to the impact on services as well.

When teams change the structure of a database, they must also apply corresponding changes to the source code in the services or monolithic system using that database. The level of effort for a database structural change is not only changing the database itself, but also changing the services impacted by database change. Teams must researching the impact of a database change as well as take into account the effort to apply source code changes, test those changes, and collectively deploy all of the changes in concert with each other.

The monolithic database topology is by far the most difficult to change with respect to the impact on other services or behavior since the entire system (monolithic or distributed) shares the data. For example, the simple task of dropping a column in a `Customer` table requires the team to locate all of the services and places in the source code that use that table, and modify the source code accordingly. This, of course, assumes those services are not undergoing any sort of other maintenance. Also, imagine trying to deploy dozens of services in concert with the database changes in a production environment—a high-risk effort since the team might have missed a service or two in their impact analysis as illustrated in Figure 10-1. The problem is, those missed services are now non-operational since it would be too difficult to back out all of the other changes.

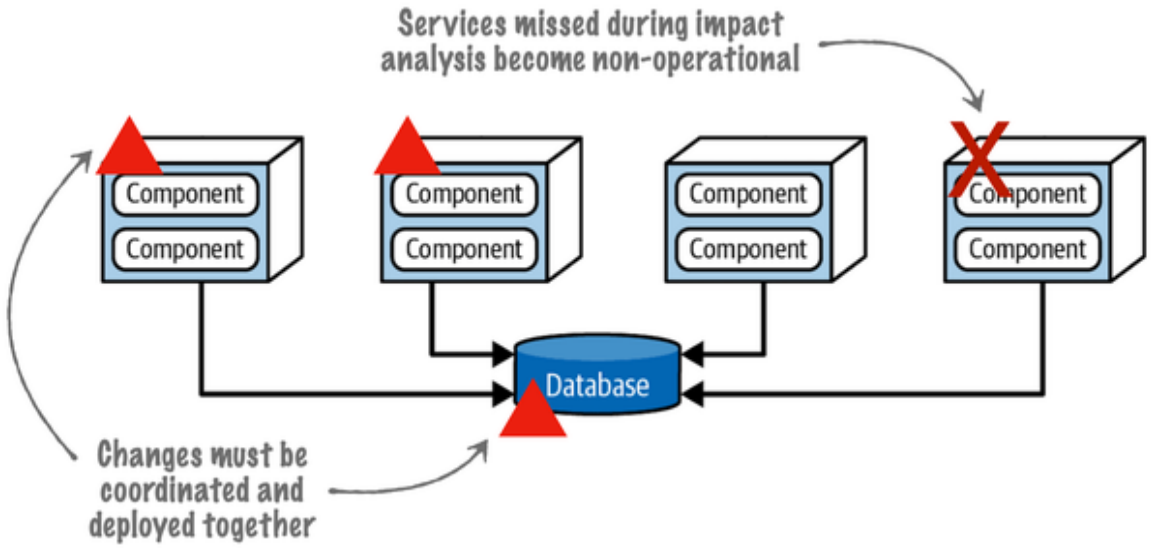


Figure 10-1. Database changes are risky with the monolithic database topology because teams might miss some services in their impact analysis

The domain database topology is a little better with regards to the ease of database changes, but only by two stars as indicated in Table 10-3. This is because database structural changes are at the *domain level* rather than the system level, thus reducing the scope of services that might be impacted by the database change. However, teams will still incur some deployment risk when using this topology because the domain database is shared my multiple services within that domain.

The clear winner with regards to the ease of database changes is the dedicated database topology. Teams can more easily change the database structure due to the tight bounded context formed between the service and its corresponding data as shown in Figure 10-2. Changing the database structure means only changing the corresponding service within that bounded context—no other services are impacted by the change.

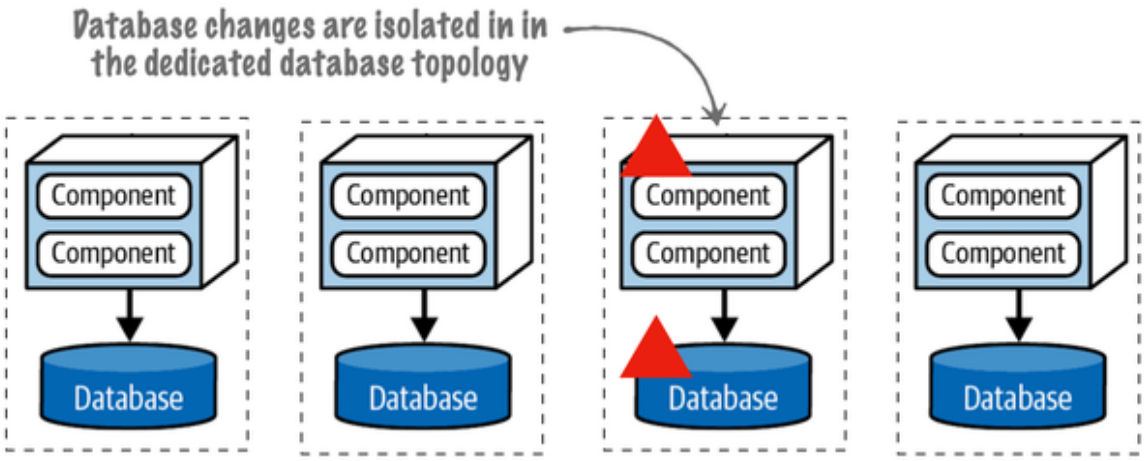


Figure 10-2. Database changes are much easier with the dedicated database due to the impact being at the service level within a bounded context

That said, there are times when teams will need to change the contract between services based on a database change in one bounded context. For example, if the `Profile` service added a 5th address line to the customer's address in the database, the corresponding contract containing the customer's address would *eventually* have to change as well, meaning the database change and corresponding service change could be deployed before other services know about the additional data field. Teams can also leverage contract versioning when using strict contracts to help mitigate the initial impact of database structural changes.

Table 10-3 shows the qualitative ratings for each of the three primary database topologies with respect to the ease of database structural changes.

Table 10-3. Qualitative comparisons for ease of database changes

Database Topology	Ease of Database Changes
Monolithic Database	★
Domain Database	★★★
Dedicated Database	★★★★★

We recommend architects track the metrics we showed in Chapter 9 to determine the importance of this capability with respect to the database topology choice.

Structural Considerations

Another consideration architects should take into account when choosing a database topology is the impact on the overall structure of the system. Making distributed data too-fine grained (such as choosing the dedicated database topology) can create lots of synchronous interservice calls to retrieve data, thus coupling services together and significantly degrading operational characteristics. However, making data too coarse-grained (such as using the monolithic and domain database topologies) can entangle the architectural quanta, thus increasing the overall dynamic coupling in the system and further degrading operational characteristics. The following sections discuss these structural considerations and how to create a balance between them.

Service Decoupling

Whenever an architect uses the domain or dedicated database topology, they run into the possibility that a service might need access to data in another

another bounded context. For example, in the diagram illustrated in Figure 10-3 using the dedicated database topology, both the `Profile` service and the `Ticket Assignment` service own their own database in separate bounded contexts. This means services outside of the bounded context cannot directly access the database within that bounded context. However, the `Ticket Assignment` service needs to know the customer location information, which is stored in the `Profile` database.

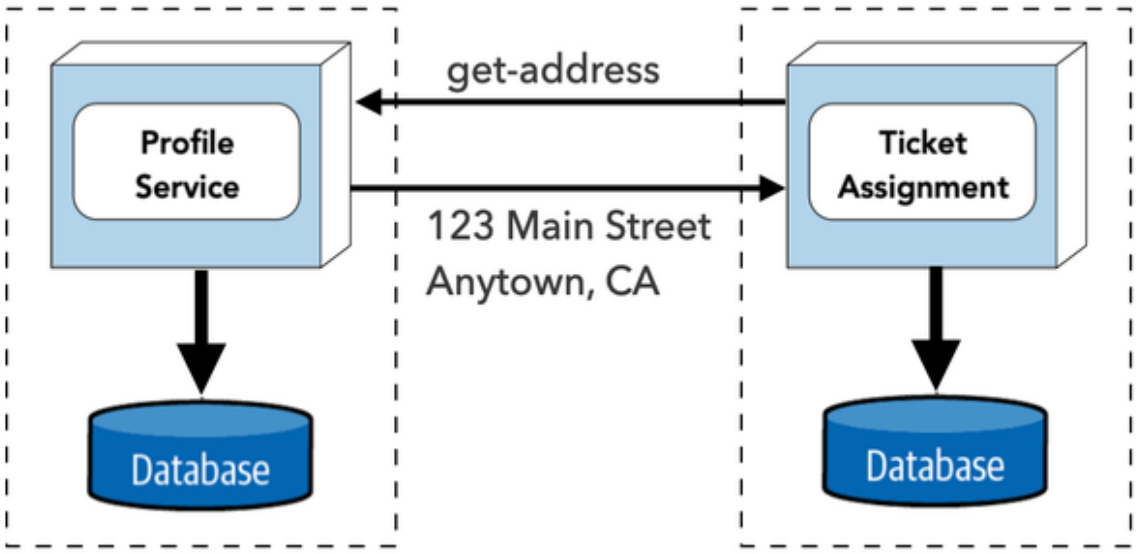


Figure 10-3. A synchronous interservice call to retrieve data from another bounded context

Since the `Ticket Assignment` service cannot access the `Profile` database directly, it must make a synchronous interservice call to get that information[1]. This creates service dependencies, thus degrading scalability, responsiveness, and fault tolerance. Furthermore, when the source service synchronously communicates with another service, the source service's architectural characteristics sync to the lowest level of the connected services's architectural characteristics.

For example, in Figure 10-3, suppose the `Ticket Assignment` service has a responsiveness requirement of 800ms. While the `Ticket Assignment` service currently responds within an average of only 500ms, well below the requirement, the `Profile` service takes about 700ms to retrieve and return the customer location information. This brings the

`Ticket Assignment` service's total response time to 1200ms (500ms + 700ms), well *over* the requirement.

Another issue with synchronous interservice communication to get data is scalability. Suppose the `Ticket Assignment` service can scale to 5,000 concurrent customers, but the `Profile` service can only scale to 400 concurrent customers. If the `Ticket Assignment` service needs to make a synchronous interservice call to the `Profile` service, the `Ticket Assignment` service syncs to the lowest scalability level, which is 400 concurrent customers—well below the required 5,000 concurrent customers.

Table 10-4 shows the qualitative ratings for each of the three primary database topologies with respect service decoupling. Here, five stars means service decoupling is good—in other words, services don't need to communicate to obtain data because they are all sharing the same monolithic database. Domain databases is rated at three stars, meaning while services *within* a domain don't have to communicate to obtain domain data, they do have to communicate to obtain data *outside* the domain. Dedicated databases have the highest probability that services will need to communicate to get data, which is why we rated it only one star (bad) for service decoupling.

Table 10-4. Qualitative comparisons for service decoupling

Database Topology	Service Decoupling
Monolithic Database	★★★★★
Domain Database	★★★
Dedicated Database	★

Synchronous interservice communication is only one technique a service can use to obtain data outside of its bounded context. Teams can use several other techniques, such as leveraging data-based payload contracts, data schema replication, in-memory replicated caching, data sidecar with distributed caching, and changing the database topology to share data. It's important for teams to understand the trade-offs involved with each of these options, which we summarize in the following sections.

Data-based payload contracts

One way teams can make sure services have the data they need for processing a request or event is to make sure they have all the information they need in the payload of the request. This pattern is called the *Data-Based Event Contract* pattern.

While this may seem like an obvious choice to avoid synchronous interservice communication, it has its downfalls as well. The first downfall is that all of the services in a workflow chain would need the data as well so they can pass it along to the next service. This results in a significant increase in bandwidth. Furthermore, the data-based event contract pattern must have a contract to define the data in the payload (such as the complete data for a ticket), thus statically coupling the services together. A change to that contract (such as adding a customer's alternative work address) would impact all services in the system, similar to the issues we described earlier with changing the database structure in the monolithic database topology.

Table 10-5 shows the trade-offs for the data-based event contract pattern.

Table 10-5. Trade-offs for the data-based event contract pattern

Advantages	Disadvantages
Service Decoupling	High Bandwidth Usage
Elasticity	Data Consistency
Responsiveness	Data Integrity
Scalability	Contract Management
Throughput	Contract Versioning
Fault Tolerance	

To learn more about the the data-based event contract pattern and its trade-offs, see our book Software Architecture Patterns, Antipatterns, and Pitfalls (O'Reilly, 2026).

Data schema replication

Another way teams can avoid synchronous interservice communication is through replicating the data in another database. For example, using the same `Profile` and `Ticket Assignment` service scenario from the prior section, teams can replicate the customer's location information in the `Ticket Assignment` database by adding columns for that data as illustrated in Figure 10-4.

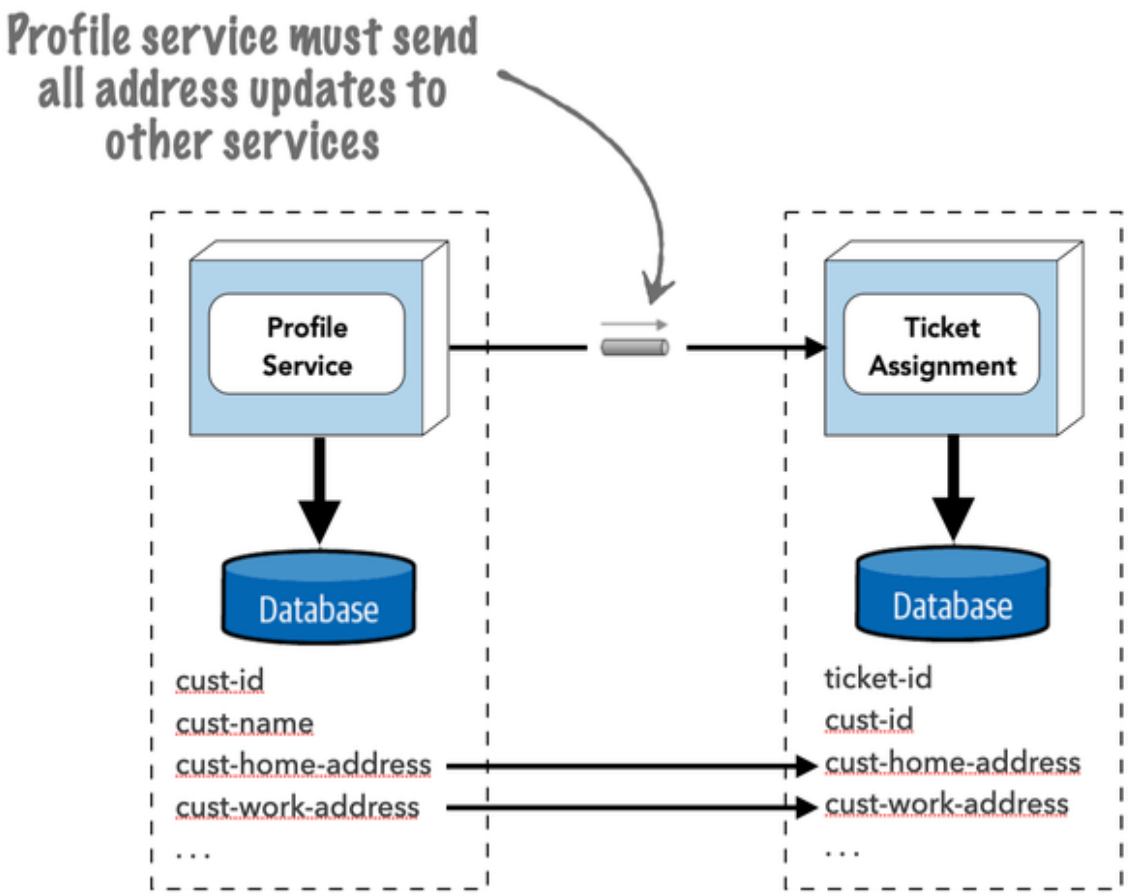


Figure 10-4. Customer location data is replicated in the Ticket Assignment database

While this technique removes the need for a service to make a synchronous interservice call to get the data it needs, it does require the team to create some sort of custom data synchronization process so that when a customer is added, removed, or the address information changes, the `Profile` service can make sure the `Ticket Assignment` service has the latest information. In Figure 10-4 we are showing the use of asynchronous messaging using a queue, but even that has trade-offs as opposed to using synchronous communication, including the data being eventually consistent, and not knowing when—or if—the data is in sync.

Table 10-6 shows the trade-offs for the data schema replication technique.

Table 10-6. Trade-offs for the data schema replication technique

Advantages	Disadvantages
Service Decoupling	Bandwidth Usage
Low Bandwidth Usage	Data Consistency
Responsiveness	Data Synchronization
Scalability	Data Ownership
Throughput	
Fault Tolerance	

To learn more about the data schema replication technique, refer to Chapter 10 in our book Software Architecture: The Hard Parts (O'Reilly, 2021).

In-memory replicated caching

Caching is another option teams can leverage to not have to ask the data. The first caching technique is *in-memory replicated caching*, where the data needed by a service in another bounded context is located in the service's memory via an in-memory data grid. With this technique, the `Profile` service would contain an updatable in-memory cache (usually consisting of simple name-value pairs) containing the customer's home and work addresses, and the `Ticket Assignment` service would contain a corresponding read-only in-memory cache containing that same information as shown in Figure 10-5.

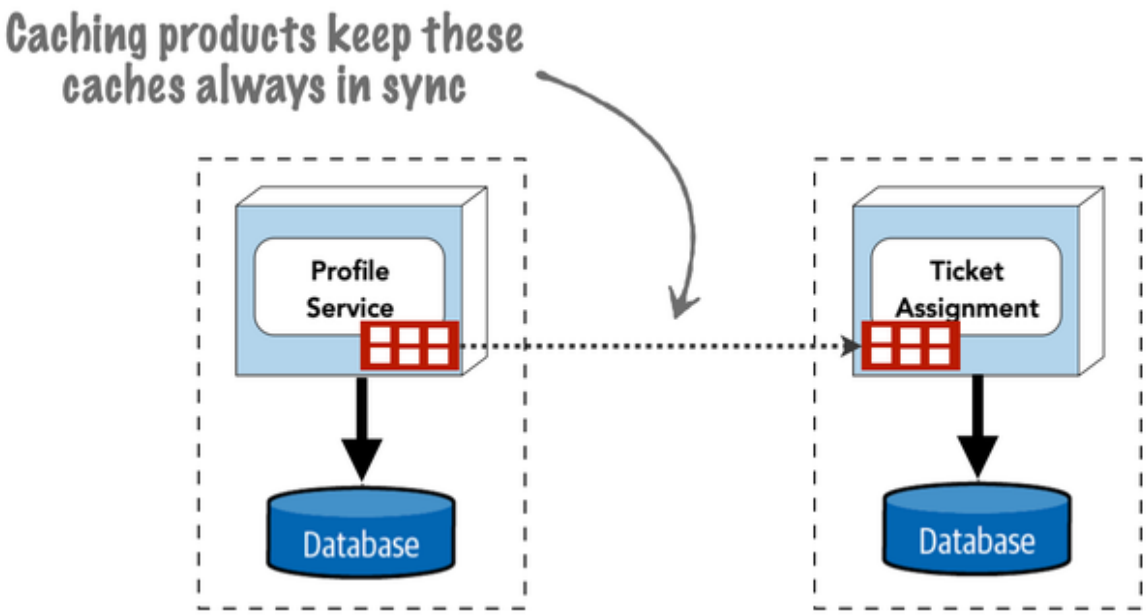


Figure 10-5. Customer location data is located in an in-memory replicated cache within the Ticket Assignment service

Caching products such as Hazelcast, Apache Ignite, and Oracle Coherence (to name a few) work asynchronously to always keep these caches in sync with each other. If the customer location information changes in the `Profile` service, the service first updates the database, then updates its in-memory cache. Caching products notice the update in the cache and update the in-memory caches in all other services using that same-named cache.

While this technique avoids the need for interservice communication and data schema replication, it does have a few problems related to data volumes and data update rates. Because the cache is stored in the service's memory, this technique is best suited for smaller (<100MB) caches. Also, if the update rate is too high and exceeds the replication latency of the caching products, the data will never be in sync.

Table 10-7 shows the trade-offs for the in-memory replicated caching technique.

Table 10-7. Trade-offs for the in-memory replicated caching technique

Advantages	Disadvantages
Service Decoupling	Data Volumes
Low Bandwidth Usage	Data Consistency
Responsiveness	High Data Update Rates
Scalability	Cold Start Dependency
Throughput	
Fault Tolerance	
Data Synchronization	
Data Ownership	

To learn more about the in-memory replicated caching technique, refer to Chapter 10 in our book Software Architecture: The Hard Parts (O'Reilly, 2021).

Data sidecar with distributed caching

An alternative caching technique teams can use to avoid synchronous interservice communication is to use a *data sidecar* with a distributed cache. A data sidecar is an external distributed cache associated with a service that other services can directly access to obtain data.

Using this technique, teams create an external distributed cache belonging to the owning service's bounded context as shown in Figure 10-6. Services needing data from that bounded context access the external cache rather than making an interservice call to the owning service, thus not disrupting the owning service from doing its primary job.

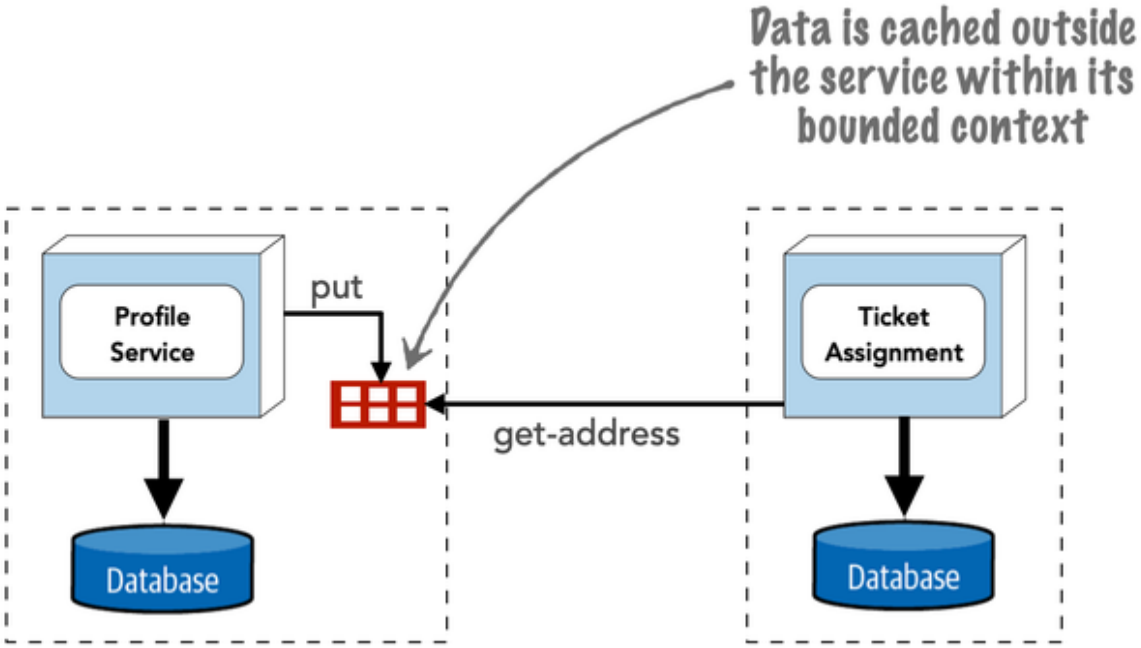


Figure 10-6. Customer location data is located in an external data sidecar, allowing other services to access the data via the cache

In this example, the `Profile` service is the only service that can write to the cache. All other services (for example, the `Ticket Assignment` service) can now access the cache directly in a read-only fashion. Essentially, the data sidecar cache acts as an external contract for the data, isolating the internal database structure from the outside world. Teams are free to make changes to the `Profile` database without having to change the external cache representation.

This is a popular technique among architects due to its simplicity and data consistency. Like the in-memory replicated caching technique, the `Profile` service can go down and not impact any other processing because the data is in a cache. Also, data volumes and update rates aren't usually an issue with this technique because the cache is external to the

service. While Redis remains a popular caching product choice for the data sidecar distributed caching technique, any distributed cache would work.

Table 10-8 shows the trade-offs for the data sidecar with distributed caching technique.

Table 10-8. Trade-offs for the data sidecar with distributed caching technique

Advantages	Disadvantages
Service Decoupling	Fault Tolerance (Cache Dependency)
Low Bandwidth Usage	Cold Start Dependency
Responsiveness	
Scalability	
Throughput	
Data Synchronization	
Data Ownership	
Data Volumes	
Data Update Rates	
Data Consistency	

To learn more about the data sidecar with distributed caching technique, refer to Chapter 10 in our book Software Architecture: The Hard Parts (O'Reilly, 2021).

Sharing data between services

One last approach teams teams can use to avoid synchronous interservice communication is to simply share the data between the services, thus forming a broader bounded context as illustrated in Figure 10-7.

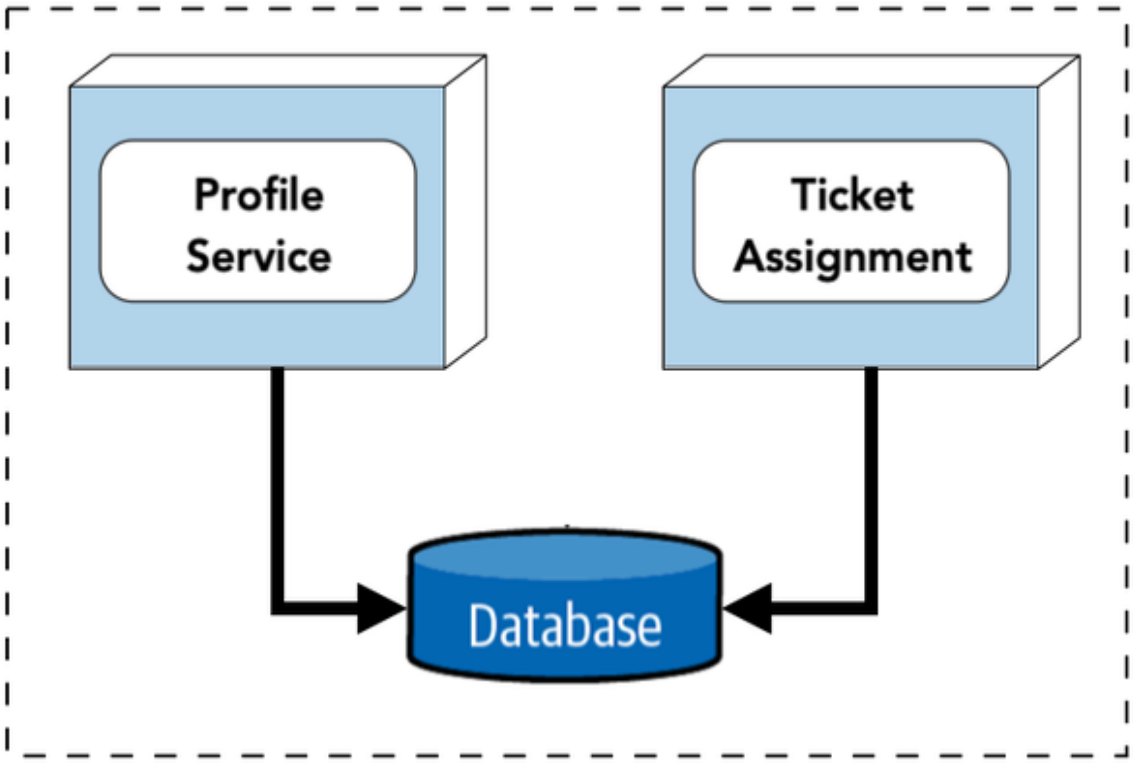


Figure 10-7. Sharing the database between services

Changing the database topology is always an option, but architects should be aware that doing so might negatively impact other process-related, data-related, and operational-related considerations. Many times architects find that behavior can easily be broken apart into separate services, but breaking apart the underlying data is either challenging or in some cases simply not possible.

CHECK SERVICE GRANULARITY WHEN SHARING DATA IN MICROSERVICES

Although a common practice, whenever sharing data between microservices always ask if the microservices sharing the data should still remain as separate services. If teams don't have good justifiable reasons, they can consolidate the microservices to simplify the architecture.

Table 10-9 shows the trade-offs for the data sharing technique.

Table 10-9. Trade-offs for the data sidecar distributed caching technique

Advantages	Disadvantages
Service Decoupling	Ease of Database Changes
Low Bandwidth Usage	Data Ownership
Responsiveness	Possible Security Access Issues
Scalability	Domain Alignment
Throughput	
Data Synchronization	
Data Volumes	
Data Update Rates	
Data Consistency	
Fault Tolerance	
No Cold Start Dependency	

To learn more about this technique, refer to Chapter 10 in our book Software Architecture: The Hard Parts (O'Reilly, 2021).

Architectural Quantum

An *architectural quantum*, defined in our books *Software Architecture: The Hard Parts* (O'Reilly, 2021) and *Software Architecture Patterns, Antipatterns, and Pitfalls* (O'Reilly, 2026), is a separately deployable part of the architecture that is bound together through synchronous dynamic coupling. Since each architectural quantum contains it's own set of architectural characteristics, entangling the quanta means the collective architectural characteristics sync to the lowest level of each.

The type of database topology an architect choses to use can significantly influence the architectural quantum of the system. For example, using the monolithic database topology always forms as single architectural quantum for the entire system as shown in Figure 10-8, irrespective of the asynchronous communication between the `Ticket Creation` and `Ticket Assignment` services. This is because all of the services are synchronously coupled to the database, meaning if the database were to go down, all services would cease to function (that's fault tolerance). If the monolithic database can only scale to 200 concurrent users, then *all* services in the system can only scale to that level.

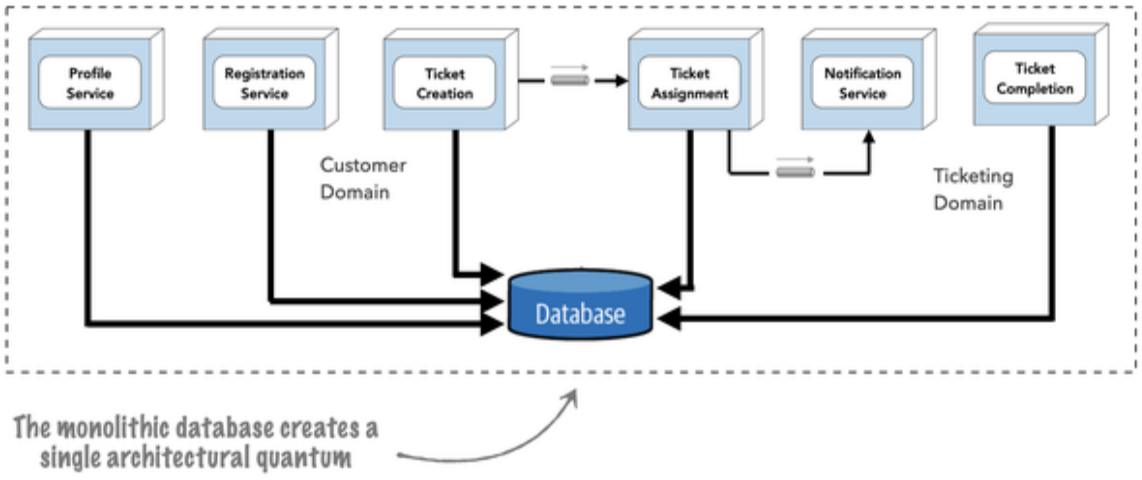


Figure 10-8. The monolithic database topology forms a single architectural quantum

However, if an architect chooses to break apart the database using either the domain or dedicated database topology, they can create separate architectural quanta from a database perspective as illustrated in Figure 10-9.

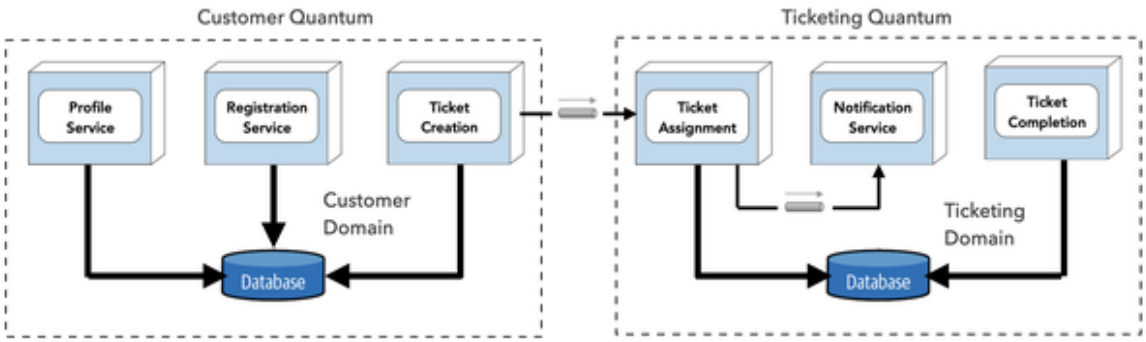


Figure 10-9. Distributing the database can form separate architectural quanta

Forming separate quanta means each can have its own set of architectural characteristics and is a completely and independently deployable separate part of the system. For example, in Figure 10-9 the `Customer` quantum, containing its own services and database, is totally independent of the `Ticketing` quantum due to the asynchronous communication between the `Ticket Creation` service and the `Ticket Assignment` service. As such, each can have a completely different set of architectural characteristics, such as more scalability, availability, and fault tolerance.

Note that although changing the *database topology* can create separate architectural quanta, synchronous service communication can entangle them, which brings us back to the issues we discussed in "Service Decoupling". For example, suppose an architect decides to use a domain database topology for Sysops Squad system to create separate quanta, but decides to have the `Ticket Creation` service *synchronously* send a new problem ticket to the `Ticket Assignment` service as illustrated in Figure 10-10. Doing so would effectively entangle those quanta into a single architectural quantum, irrespective of the fact that the architect is using a domain database topology.

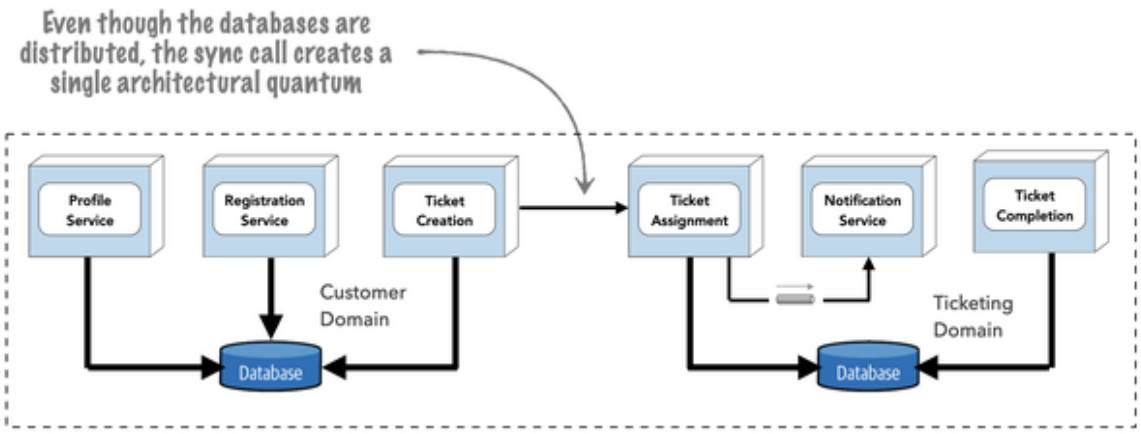


Figure 10-10. Synchronous communication between quanta entangles them, regardless of the database topology

Table 10-10 shows the trade-offs for creating separate architectural quanta within a system.

Table 10-10. Qualitative comparisons for creating separate architectural quanta

Database Topology	Architectural Quantum
Monolithic Database	★
Domain Database	★★★
Dedicated Database	★★★★★

The important point to remember is that the choice of database topology can impact the boundaries of the architectural quanta in a system, but it's not the only factor that determines that boundary.

Data-Related Considerations

Data-related considerations are those architectural capabilities associated with specific aspects about the data side of architecture. They include capabilities such as data consistency, preserving semantic data coupling, and data integrity contraints. The following sections describe the considerations and trade-offs of each of these capabilities for the three primary database topologies.

Data Consistency

Data consistency is about keeping data in sync when breaking apart databases. For example, in the Sysops Squad architecture illustrated in Figure 10-9 in the prior section, tickets are created in the `Customer` domain, and then processed in the `Ticketing` domain. Data consistency is about ensuring that each domain database contains the same tickets, and that customer keys in the `Ticketing` domain database are also contained in the `Customer` domain database.

Clearly, the monolithic database topology has the highest rating (five stars) for data consistency since all transactional workflows use the same shared database. However, data consistency starts to degrade as architects break apart the database because the data can easily get out of sync.

When using the domain database topology, processing within the domain has good levels of data consistency. But when workflows span multiple domains, as illustrated in Figure 10-9, data consistency degrades, but only by two stars as shown in Table 10-11. The dedicated database topology has the lowest data consistency rating (only one star) due to the fine-grained nature of the data and the number of databases that need to be kept in sync.

Processing workflows using the domain database topology or dedicated database topology must rely on distributed transactions when performing updates across multiple data sources. Distributed transactions can be error prone and hard to reconcile when services encounter unrecoverable errors, thus also degrading data consistency.

For example, imagine if a customer decided to unsubscribe from the Sysops Squad system while a ticket from that customer was still waiting to be

processed. Most of the dedicated databases belonging to Sysops Squad services likely contain that pending ticket and corresponding customer's information (at least the customer ID at a minimum). This means each of those services would need to be involved in the processing workflow when a customer unsubscribes so they can remove the pending ticket and corresponding customer information from each of their databases. However, if any errors occur during this workflow, the databases can get out of sync with each other.

Table 10-11 shows the trade-offs for data consistency for the three primary database topologies.

Table 10-11. Qualitative comparisons for data consistency

Database Topology	Data Consistency
Monolithic Database	★★★★★
Domain Database	★★★
Dedicated Database	★

Architects should always verify the importance of data consistency in their system prior to making a database topology choice. As always, data consistency produces trade-offs for an architect to think about in conjunction with the other considerations presented in this chapter.

Preserving Semantic Data Coupling

Semantic data coupling describes the logical relationships between database tables or other table-related constructs based on a particular domain entity concept, independent from how they are physically coupled. Physical coupling includes database artifacts such as foreign key relationships,

views, triggers, and stored procedures. We discuss physical coupling in "Data Integrity Constraints".

To better understand semantic data coupling, suppose a Sysops Squad data architect decideed to model the domain entity concept of a `Ticket` as four separate database tables: `ticket_core`, `ticket_status`, `ticket_expert`, and `ticket_location` as illustrated in Figure 10-11.

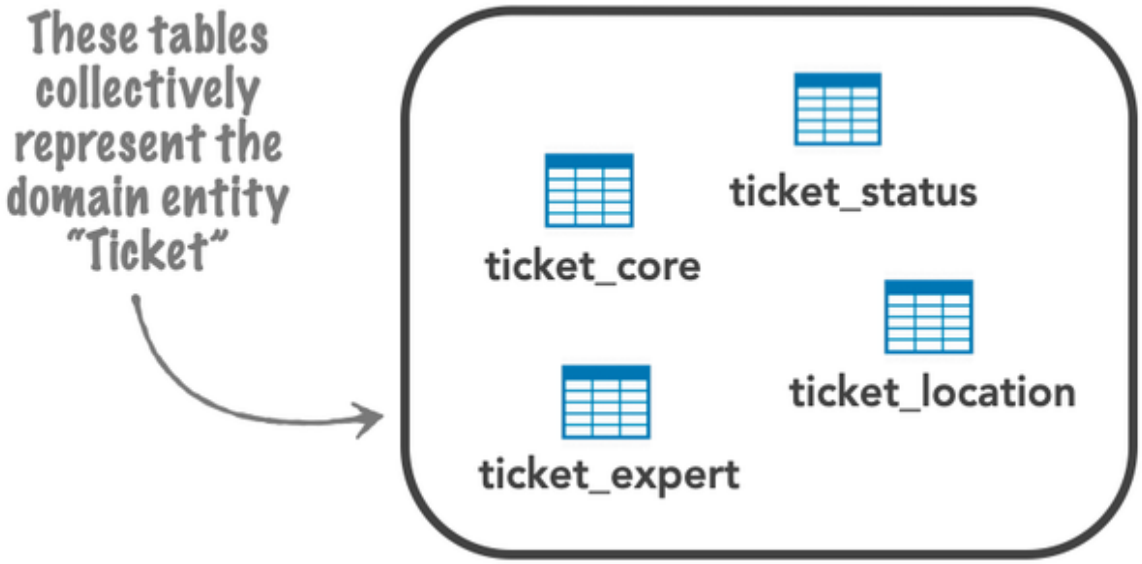


Figure 10-11. The Ticket domain entity modeled as four database tables

Together, these four tables represent a ticket in the database, and as such are considered to be semantically coupled. Most architects think of coupling as bad and tend to avoid it at all costs. However, strong semantic data coupling for tables representing a single domain entity is not only good, but also necessary for teams to understand and maintain the domain entity and data model. The more the semantic nature of the data model is broken apart, the less the team will be able to understand the domain entity, making it more challenging for them to understand and maintain the data model.

Preserving semantic data coupling essentially decouples the data model from the architecture, allowing teams more flexibility in terms of creating and maintaining the data model and simplifying the processing of the data within the services or monolithic system.

The monolithic database topology allows architects to group data that is semantically coupled in the same shared database, therefore preserving the semantic coupling between the data, regardless of any unusual or rare cross-domain semantic coupling. The same is true for the domain database topology, as illustrated in Figure 10-12.

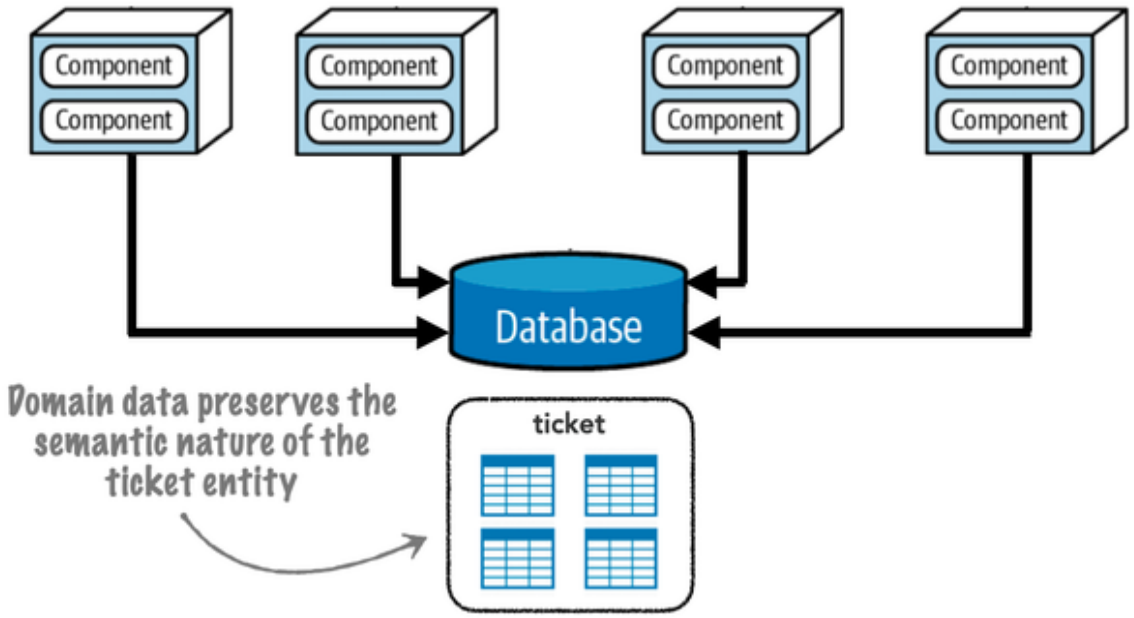


Figure 10-12. The semantic nature of the ticket tables is preserved when using the monolithic or domain database topology

Because cross-domain semantic coupling is rare, we've given both the monolithic and domain database topologies high ratings (five stars) in Table 10-12 with respect to preserving the semantically coupled nature of the data entities.

However, when using the dedicated database topology, teams necessarily need to break up data into fairly fine-grained databases or schemas that match the service's specific needs. Any team that has undergone this effort knows this is a challenge—one that tends to often fail. One of the reasons for these challenges has to do with how semantically coupled the data is within the data model.

The dedicated database topology tends to break apart the semantic nature of the data model, as illustrated in Figure 10-13.

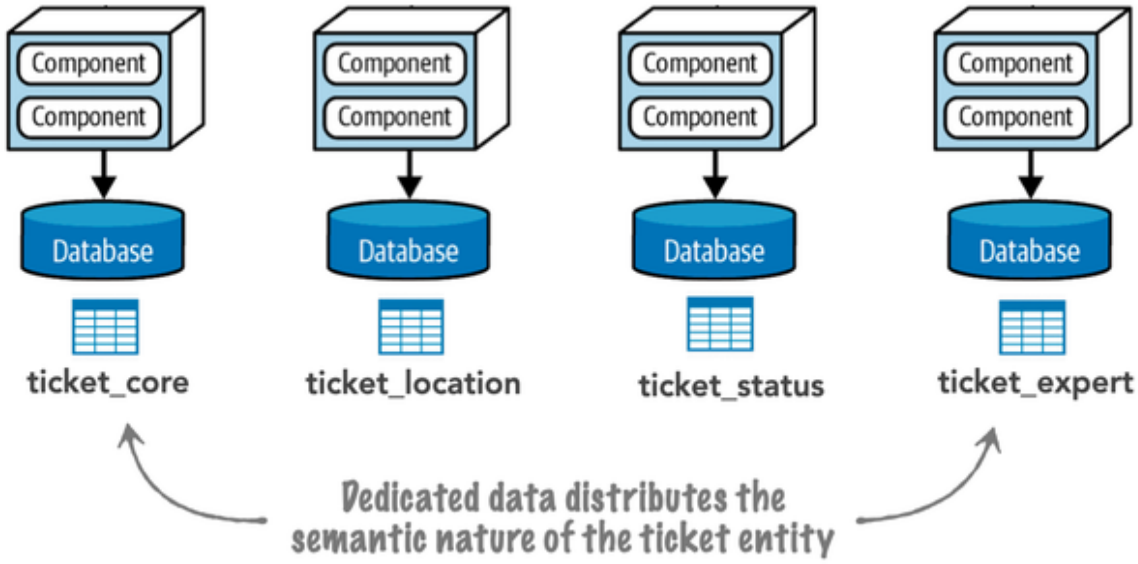


Figure 10-13. The ticket tables are spread across multiple databases when using the dedicated data topology

Notice here the semantic nature of the concept of a `Ticket` in the data model is lost because the tables are spread across multiple databases. Not only will teams find it challenging to understand what tables (or in this case databases) make up the concept of a `Ticket`, but they will struggle when trying to maintain or modify the data model.

Table 10-12 shows the trade-offs for preserving semantic data coupling for the three primary database topologies.

Table 10-12. Qualitative comparisons for preserving semantic data coupling

Database Topology	Preserving Semantic Data Coupling
Monolithic Database	★★★★★
Domain Database	★★★★★
Dedicated Database	★★

The monolithic and domain database topologies have the same five star rating because when data is usually semantically coupled, its at a domain-level and rarely cross-domain.

Data Integrity Constraints

Data can be physically related in many ways. In relational databases, data is physically related through such things as primary and foreign key constraints, views, triggers, and stored procedures. In graph databases, data is physically related through nodes and relationships. These kind of artifacts physically bind data together, creating what's called *implementation coupling*. Unlike semantic coupling, implementation coupling means the data has a physical dependency that forms constraints to ensure the integrity of the data.

For example, as illustrated in Figure 10-14, the `ticket_core` table has a reference to the customer ID for a ticket. This relationship is represented through a primary key (PK) and foreign key (FK) relationship. in this case the `customer_profile` table owns the primary key, and the `ticket_core` contains the customer ID as a corresponding foreign key. The same is true for the `ticket_core` table and the other ticketing tables. The `ticket_core` owns the primary key (in this case the ticket ID) and the other ticketing tables contain the ticket ID as a foreign key reference.

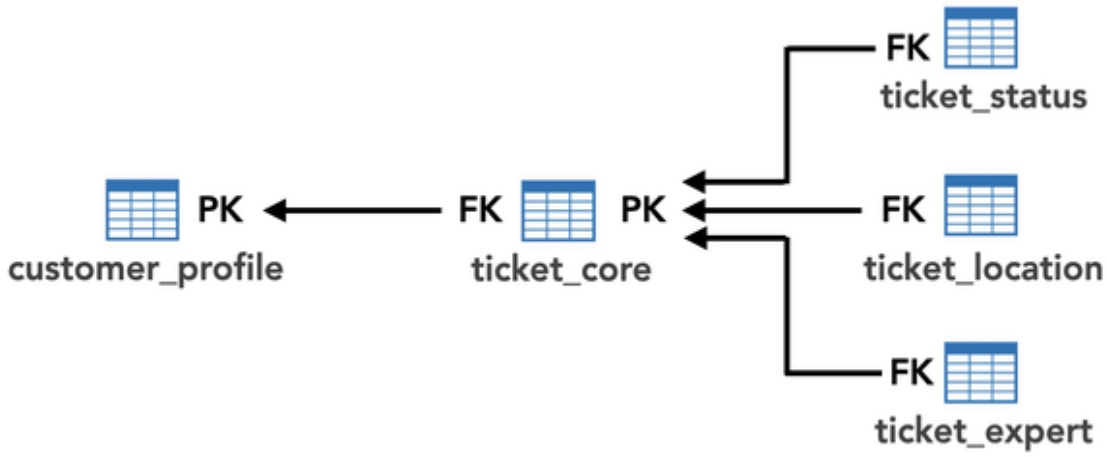


Figure 10-14. Artifacts that physically couple data are referred to as implementation coupling and help support data integrity

With these constraints in place, it's not possible to have an entry in the `ticket_core` table for a ticket without a corresponding customer ID in the `customer_profile` table. Similarly, a ticket ID must exist in the `ticket_core` table for any rows to exist in the other ticketing tables. These constraints are what help support the integrity of the data across these tables.

Since implementation coupling defines a physical connection between data, the monolithic database topology works well here because of the use of a single physical database. Architects don't need to worry about breaking apart foreign key constraints, materialized views, triggers, stored procedures, or graph nodes and graph relationships. Those physical relationships are preserved when using a single database.

However, When breaking data apart, teams may have to also break apart the physical implementation coupling such as foreign key constraints, views, triggers, and stored procedures. For example, when using the domain database topology, teams need to remove physical cross-domain relationships. For the dedicated database topology, they have to remove physical relationship within a domain.

The domain database topology preserves the physical relationships within the domain, but not ones that are cross-domain. For example, given the physical relationships shown in Figure 10-14, if the `customer_profile`

table existing in the `Customer` domain database and the `ticket_core` table in the `Ticketing` domain database, then the primary key to foreign key relationship between these tables would have to be removed. The `ticket_core` table would still contain the customer ID, it's just that the physical relationship wouldn't be enforced by the database. However, the primary key to foreign key relationships between the `ticket_core` table and the other ticketing tables would be preserved.

We gave the domain database three stars for data integrity in Table 10-13 due to possibly having to remove cross-domain data integrity constraints while still being able to keep the inter-domain ones intact.

The dedicated database topology, however, does not generally preserving the physical implementation coupling due to the fine-grained nature of this database topology. As shown in Figure 10-13 in "Preserving Semantic Data Coupling", breaking apart data into fine-grained databases matching the specific service functionality not only removes the semantic coupling, but the implementation coupling as well. For this reason we gave the dedicated database topology one star in Table 10-13.

Table 10-13 shows the trade-offs for data integrity and the ability to preserve the physical coupling constraints (like primary key and foreign key relationships between tables) for the three primary database topologies.

Table 10-13. Qualitative comparisons for data integrity constraints

Database Topology	Data Integrity Constraints
Monolithic Database	★★★★★
Domain Database	★★★
Dedicated Database	★

If data integrity is a critical architectural characteristic for the system, the dedicated database is likely not a good choice. The same is true for the domain database topology—if the data model has lots of critical cross-domain physical coupling, teams might find it a struggle to maintain data integrity across the domain databases.

Operational Considerations

Operational considerations include architectural capabilities that influence the runtime aspects of the system such as scalability, elasticity, responsiveness, fault tolerance, availability, and so on. Because scalability, elasticity, and responsiveness are closely related within the context of database topologies, we've chosen to consolidate these three capabilities into a single section in "Scalability". Similarly, since fault tolerance and availability are also closely related within the context of database topologies, we've consolidated these into a single section in "Fault Tolerance".

Scalability

This section includes scalability, elasticity, and responsiveness (which we collectively refer to as *scalability* in this section), and how the database topology can influence them. This is an important section for architects to consider since the ultimate bottleneck in any scalable system is the database.

The monolithic database topology generally doesn't scale as well as the other distributed data topologies because all requests are funned to a single database, which typically forms a scalability bottleneck. That said, there are some database vendors that overcome many of the bottleneck issues found with monolithic databases. For example, Oracle offers both vertical and horizontal scalability features in its popular relational database by allowing a single database to run on multiple servers. However, note that while

scalability is improved, leveraging these features quickly degrades cost and simplicity.

The dedicated database topology is exactly the opposite, offering architects the best possible scalability available from a database topology perspective. Databases only need to scale at a service-level, meaning most systems leveraging the dedicated database topology do not have issues with scalability.

The domain database topology fits right in the middle of these two extremes. Domain databases must be able to scale at a domain-level, which is much better than a monolithic database topology, but not as good as using dedicated databases. Scalability is sometimes what drives domain scoping from a database perspective. If the domain is too large and contains too many services, the system will likely not meet its scalability needs.

Table 10-14 shows the trade-offs for scalability for the three primary database topologies.

Table 10-14. Qualitative comparisons for scalability

Database Topology	Scalability
Monolithic Database	★
Domain Database	★★★
Dedicated Database	★★★★★

A note of caution with regards to scalability: Architects should take into account the architectural style they are using when choosing a database topology where scalability is a concern. For example, the Space-based architectural style (see "Space-Based Architecture") is an architectural style

that offers the highest level of scalability out of any other architectural style because it leverages caching rather than a database for transactional processing. However, because of it's unique architecture and the use of tuple-space, architects might be able to get extreme levels of scalability using a domain database topology.

Fault Tolerance

Fault tolerance and availability are closely related with respect to the database topology, and as such we combine them here, referring to each collectively as *fault tolerance*. Fault tolerance is the ability for parts of the system to remain functional when other parts of the system fail. For databases, if the database goes down, all services connected to that database fail, which basically describes the overall availably of that part of the system.

A big disadvantage of the monolithic database topology is the lack of fault tolerance. Because all services are dependent on a single database, the database forms an architectural single point of failure. Like with scalability, some database vendors deal with availability and fault tolerance through horizontal scalability and allowing the database to run on multiple servers, but again at the sacrifice of cost and simplicity.

Like scalability, the dedicated database offers the highest level of fault tolerance. If a database goes down, only the service accessing that database fails. Architects might think this database topology would solve all of their fault tolerance needs, but unfortunately this capability is a little more complicated than that.

Recall from "Service Decoupling" that the dedicated database topology can sometime require services to synchronously communicate with each other to get data from another bounded context. If this happens, then those services become coupled to each other and create synchronous dynamic dependencies. Even through the dedicated database topology provides excellent fault tolerance, it starts to significantly degrade if one service needs data from another service and that service is unavailable.

Table 10-15 shows the trade-offs for fault-tolerance for the three primary database topologies.

Table 10-15. Qualitative comparisons for fault tolerance

Database Topology	Fault Tolerance
Monolithic Database	★
Domain Database	★★★
Dedicated Database	★★★★★

> **THE CAPABILITIES FOR DATA TOPOLOGIES ARE INTERTWINED**
>
> Just because a database topology provides excellent capabilities in one area (such as fault tolerance) doesn't mean the data topology will ultimately have those capabilities. Process-related capabilities, structural-related capabilities, data-related capabilities, and operational-related capabilities are all intertwined, requiring architects to collectively take all of these considerations into account, striking a balance between them to arrive at the most appropriate database topology choice.

Qualitative Ratings Summary

Table 10-16 summarizes the qualitative star ratings for the monolithic, domain, and database topologies based on the considerations discussed in the prior sections.

Table 10-16. Qualitative comparison summary for database topologies

Capability	Monolithic	Domain	Dedicated
Process-Related Considerations			
Cost	$	$$$	$$$$$
Simplicity	★★★★★	★★★★	★
East of Database Changes	★	★★★	★★★★★
Data-Related Considerations			
Data Consistency	★★★★★	★★★	★
Semantic Entity Context	★★★★★	★★★★★	★★
Data Integrity Constraints	★★★★★	★★★	★
Structural Considerations			
Service Decoupling	★★★★★	★★★	★
Architectural Quantum	★	★★★	★★★★★
Operational Considerations			

Capability	Monolithic	Domain	Dedicated
Scalability	★	★★★	★★★★★
Fault Tolerance	★	★★★	★★★★★

Architects can use this table to help determine the most appropriate database topology choice based on what capabilities are important for the system to support. For example, for high levels of scalability, fault tolerance, and ease of database changes, the dedicated database would be a good choice. For those systems with significant cost and time constraints, the monolithic database would fit well.

While Table 10-16 is useful for helping an architect make the right database topology choice, it's not the only consideration they should take into account. The type of architectural style used also matters, which we cover in the next section.

Architectural Style Alignment

In the prior sections we demonstrated how architects can take various database-related capabilities into account when choosing the most appropriate database topology for their system. In this section, we provide additional context in terms of the alignment between a database topology and an architectural style.

Layered and Modular Monolith

Both the Layered and Modular Monolith architectural styles, described in Chapters 10 and 11 of our book Fundamentals of Software Architecture, 2nd Edition, are considered traditional *monolithic* architectures, meaning they're deployed as a single unit. For example, the system might be deployed as a single EAR file in Java or a single assembly in C#.

The strengths of these architectural styles are cost and simplicity, lending themselves to a more simple database topology—namely the monolithic database. Also, because the system is deployed as a single unit, the number of architectural quanta is always 1. As such, these architectural styles would not benefit from having distributed database topologies to help break part the architectural quanta. Rather, these topologies would only serve to increase cost and complexity.

Table 10-17 shows the data topology alignment for the layered and modular monolith architectural styles.

Table 10-17. Data topology alignment summary for the layered and modular monolith architectural style

Database Topology	Alignment	Considerations
Monolithic Database	✓	None
Domain Database	**X**	Not aligned
Dedicated Database	**X**	Not aligned

Notice only the monolithic database topology is suitable, with no further considerations (such as scalability, fault tolerance, and so on) to address.

Microkernel

The Microkernel architectural style, described in Chapter 13 of our book Fundamentals of Software Architecture, 2nd Edition, is an interesting one when it comes to database topology alignment as *all* of the primary topologies are suitable for this architectural style.

The Microkernel architectural style leverages *plug-ins* to provide additional behavior. Plug-ins are components that are separate from the core system, deployed either as libraries (like JAR files in Java or DDLs in C#) or

delineated through package structures or namespaces such as `app.sysops.plugin.*`).

Given that the Microkernel architecture is generally deployed as a monolithic architecture, one might assume that the monolithic database topology would be most suited for this architectural style—and it is. Like the Layered and Modular Monolith architectural styles, the strengths of this architecture are cost and simplicity, aligning well with the monolithic database topology as shown in Figure 10-15.

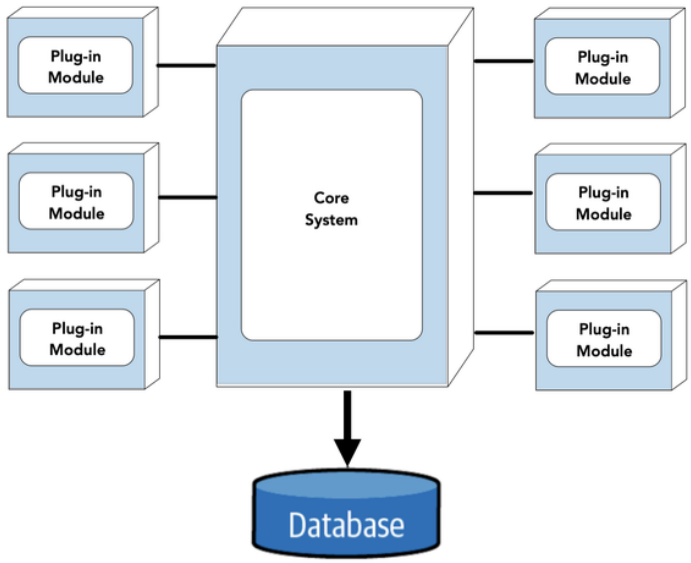


Figure 10-15. The entire architecture, including the plug-ins, share the same database

However, this architectural style has some unique aspects with regards to database topologies. Take, for example, a typical electronics recycling application where users can recycle their old electronic devices. This system, implemented as a Microkernel architecture, can have a group of plug-in components for quoting—the amount the company will give a user for their electronic device—with each plug-in representing a different electronic device (like an iPhone 13 or a Samsung Galaxy 5). Another group of plug-ins could perform an assessment of the device, again with each plug-in containing the assessment rules for a specific electronic device. Each of these plug-in domains can use a domain database topology as illustrated in Figure 10-16.

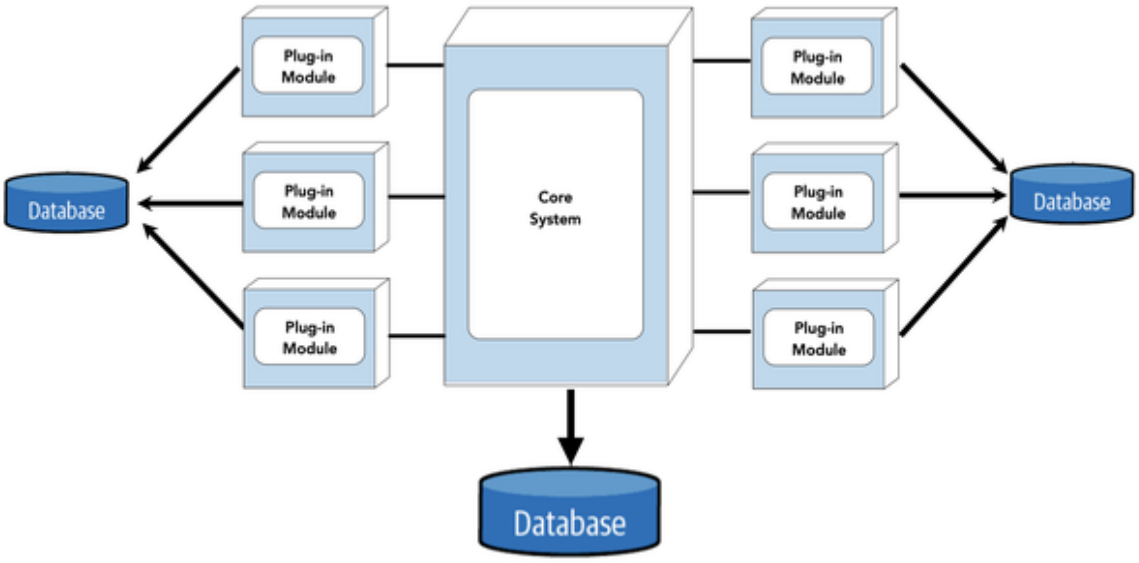


Figure 10-16. Each grouping of plug-ins can have their own domain database

Alternatively, each plug-in that calculates the quote for how much the company will pay for the device and assesses the device can have its own database using the dedicated database topology as illustrated in Figure 10-17.

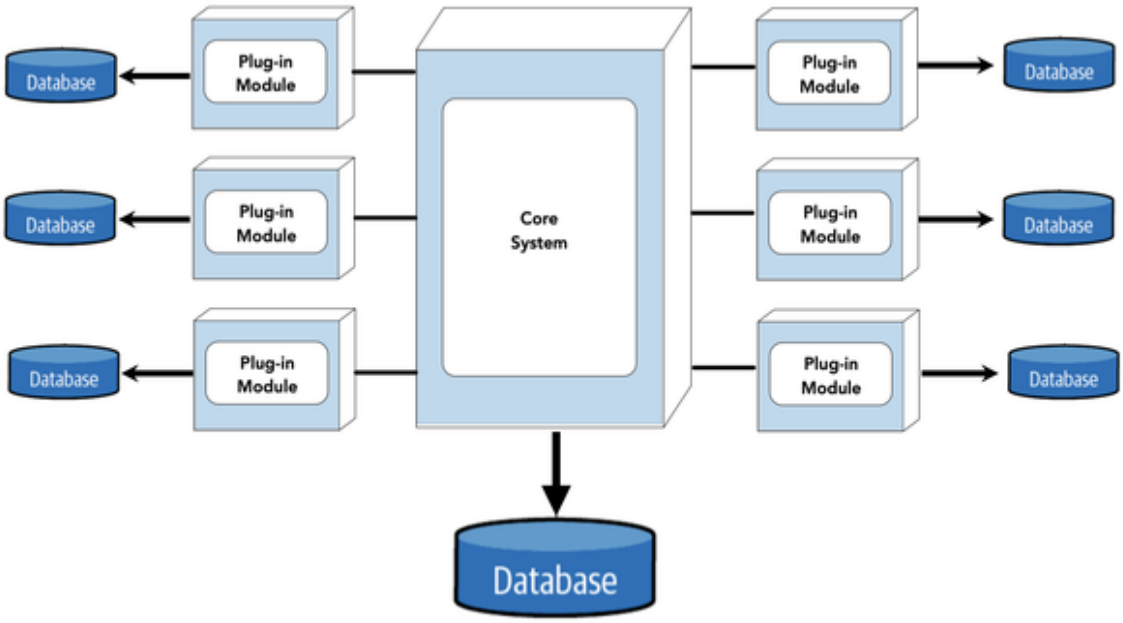


Figure 10-17. Each plug-in can have its own dedicated database

Table 10-18 shows the data topology alignment for the microkernel architectural style.

Table 10-18. Data topology alignment summary for the microkernel architectural style

Database Topology	Alignment	Considerations
Monolithic Database	✓	None
Domain Database	✓	None
Dedicated Database	✓	None

Since there are no operational considerations to take into account, Microkernel architecture has the most database topology flexibility than any other architectural style.

Service-Based Architecture

Service-based architecture, described in Chapter 13 of our book Fundamentals of Software Architecture, 2nd Edition, is an architectural style that focuses on domain partitioning—so much so that each service in this architecture typically represents a domain in the system.

Because of the domain-based focus on this architectural style, the domain database topology aligns nicely, although architects should be aware of cross-domain interservice communication when services in one domain needs data from another domain (see "Service Decoupling"). For this reason we strongly suggest implementing the fitness functions we described in Chapter 9 to track and measure this kind of interservice communication.

One of the unique aspects of Service-based architecture is its ability to use a monolithic database topology due to the small number of separately deployed services (usually two to twelve). That said, we urge architects to

consider the system operational characteristics needs such as scalability and fault tolerance when using the monolithic database topology with this architectural style.

Because Service-based architecture is already domain-centric, there's not much difference between the domain database topology and the dedicated database topology with this architectural style as they are both domain-aligned. However, that's not to say that several domain services can't share a single domain database.

Table 10-19 shows the data topology alignment for the service-based architectural style.

Table 10-19. Data topology alignment summary for the service-based architectural style

Database Topology	Alignment	Considerations
Monolithic Database	✓	Scalability, Fault Tolerance
Domain Database	✓	Scalability, Fault Tolerance, Service Coupling
Dedicated Database	✓	Service Coupling

Although all three database topologies are a fit for the service-based architectural style, architects should take into account the scalability and fault tolerance needs of the system to make sure they would be met by the database topology choice. We also encourage architects to watch out for service coupling when using the domain or dedicated database topologies.

Microservices

Microservices architecture, described in Chapter 18 of our book Fundamentals of Software Architecture, 2nd Edition, is a popular architectural style that contains fine-grained, single purposes services, each owning their own data in a tight, well-formed bounded context.

It's not uncommon to have hundreds or even thousands of microservices in a system due to the fine-grained nature of services within this architectural style. Because of this plus its high ratings for scalability and fault tolerance, the dedicated database is the only database topology that's suited for the microservices architectural style.

That said, teams typically use a hybrid approach of the dedicated database, occasionally sharing data between microservices, but no more than six services in a shared database context to avoid degrading scalability and fault tolerance.

Table 10-20 shows the data topology alignment for the microservices architectural style.

Table 10-20. Data topology alignment summary for the microservices architectural style

Database Topology	Alignment	Considerations
Monolithic Database	**X**	Not aligned
Domain Database	**X**	Not aligned
Dedicated Database	✓	Service Coupling

Since the dedicated database topology is the only topology suitable for microservices, architects should pay close attention to service coupling, and use the techniques described in "Service Decoupling" to avoid synchronous

interservice communication for purposes of getting data from another bounded context.

Event-Driven Architecture

Event-driven architecture, described in Chapter 15 of our book Fundamentals of Software Architecture, 2nd Edition, is another popular architectural style that uses asynchronous communication between services to react to events that are triggered in the system.

Like microservices, this architectural style's strengths are in operational capabilities such as scalability, elasticity, fault tolerance, and responsiveness. As such, one might assume that the dedicated database would align well with this architecture. While it does, the other database topologies align with this architectural style as well.

The dedicated database topology aligns well with event-driven architecture because the operational capabilities supported by the database topology are at the same high level of operational characteristics of the architectural style (such as scalability, elasticity, and fault tolerance). However, as noted in "Service Decoupling", the dedicated database topology tends to require services to make synchronous calls to other services to get data. This presents quite a problem for event-driven architecture because the services are highly decoupled from each other through asynchronous communication. Having to make synchronous calls to other services in a highly asynchronous architecture couples them together, degrading most of the architectural characteristics supported by event-driven architecture.

As a response to this problem, many architects choose to use the domain database topology instead. While the operational characteristics of the domain database topology are not as good as compared to with the dedicated database topology, it does limit the number of interservice calls needed to get data to only cross-domain data needs. However, architects should pay close attention to the scalability and fault tolerance needs when using event-driven architecture to make sure the domain databases can scale enough to support the requirements of the system.

In cases where systems using an event-driven architecture don't have high scalability or fault tolerance needs, architects can choose to use the monolithic database topology. This may come as a surprise to most readers, but using a monolithic database topology within event-driven architecture provides the best possible service decoupling out of all three topology choices, something that is very desirable in a highly asynchronous architecture. As with the domain database topology, architects should pay very close attention to their scalability, responsiveness, and fault tolerance needs when choosing to use the monolithic database topology.

Table 10-21 shows the data topology alignment for the event-driven architectural style.

Table 10-21. Data topology alignment summary for the event-driven architectural style

Database Topology	Alignment	Considerations
Monolithic Database	✓	Scalability, Fault Tolerance
Domain Database	✓	Scalability, Fault Tolerance, Service Coupling
Dedicated Database	✓	Service Coupling

Notice that although all three database topologies align with event-driven architecture, architects must take considerations into account for each of these topologies.

Space-Based Architecture

Space-based architecture, described in Chapter 16 of our book Fundamentals of Software Architecture, 2nd Edition, is a highly scalable, responsive, and elastic architecture that is build on the *tuple-space* pattern, where all transactional data is stored in memory, usually though in-memory replicated caching. This means services never access the database directly, giving this architecture a tremendous amount of flexibility when it comes to the database topology.

Figure 10-18 shows the basic topology for the space-based architectural style.

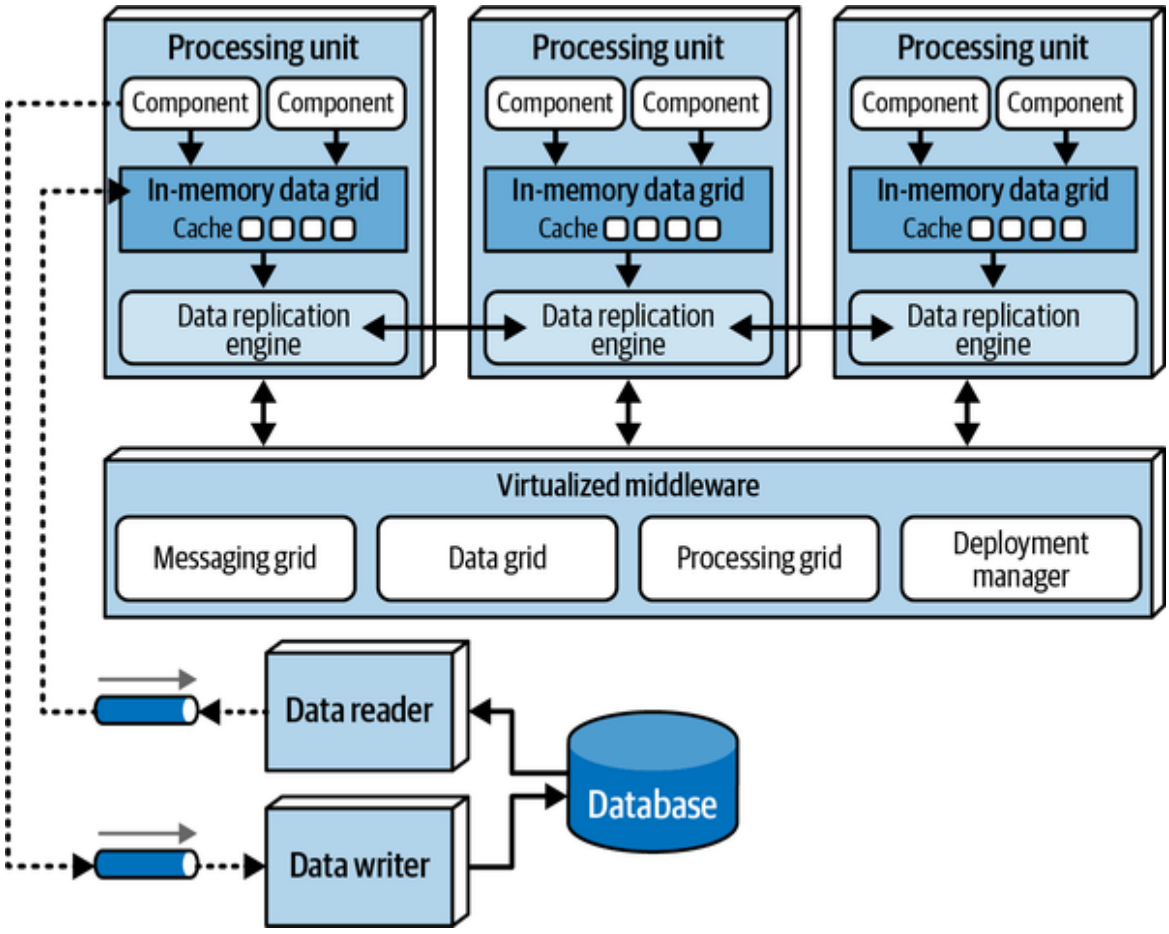


Figure 10-18. The basic topology for the space-based architectural style

Services in this architecture style use a data pump—usually implemented through a queue or stream—to asynchronously pass data to a `Data Writer` service, which communicates directly with the database to keep the database in sync with the in-memory caches in the services.

Because services don't access the database directly, all three database topologies are aligned with this architectural style. However, as noted in Table 10-22, architects should verify their scalability needs as the monolithic or domain database topologies might not be able to keep up with the high user loads typically found in this architectural style.

Table 10-22 shows the data topology alignment for the space-based architectural style.

Table 10-22. Data topology alignment summary for the space-based architectural style

Database Topology	Alignment	Considerations
Monolithic Database	✓	Scalability
Domain Database	✓	Scalability
Dedicated Database	✓	Backend database usage

Although systems implemented in the Space-based architectural style can use the dedicated database topology, we often don't see this in practice because the backend database where all the data is stored is often used for reporting and data analytics. Because these types of systems usually require domain or monolithic data, architects should verify how the backend database is used before choosing the dedicated database topology.

Alignment Summary

Table 10-23 shows a summary of which database topologies align well with the different architectural styles. Notice that some of the checkmarks showing alignment have notes attached to them—these are things architects should consider before choosing that database topology.

Table 10-23. Database topology and architectural style alignment summary

Architectural Style	Monolithic	Domain	Dedicated
Layered and modular monolith	✓	X	X
Service-based architecture	✓(1)	✓(3)	✓(2)
Microservices	X	X	✓(2)
Event-driven architecture	✓(1)	✓(3)	✓(2)
Space-based architecture	✓(4)	✓(4)	✓(5)

Notes:

(1) Pay attention to scalability and fault tolerance needs when choosing this topology

(2) Pay attention to service coupling when choosing this topology

(3) Pay attention to scalability, fault tolerance, and service coupling when choosing this topology

(4) Pay attention to scalability needs when choosing this topology

(5) Pay attention to how the backend database is used before choosing this topology

[1] We show other techniques of getting data from another bounded context in the upcoming sections

About the Authors

Neal Ford is an independent software consultant, best-selling author, and international speaker. He is an internationally recognized expert on software development and delivery, especially in the intersection of agile engineering techniques and software architecture. Neal has authored magazine articles, nine books (and counting), dozens of video presentations, and spoken at hundreds of developers conferences worldwide. His topics include software architecture, continuous delivery, functional programming, cutting edge software engineering, and includes a business-focused book and video on improving technical presentations. He is an innovator in building concrete verification mechanisms for software architecture, for humans or agents. Check out his website at *nealford.com*.

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service-oriented architectures, and distributed systems. He's been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. He's the author of numerous O'Reilly technical books and videos, including *Fundamentals of Software Architecture*, *Software Architecture: The Hard Parts* (both with Neal Ford) several books on microservices, the Software Architecture Fundamentals video series, the Enterprise Messaging video series and was a contributing author to *97 Things Every Software Architect Should Know*. A speaker and trainer, he's given talks on a variety of enterprise-related technical topics at hundreds of conferences and user groups around the world.

Jonathan Johnson is a software architect, presenter, and trainer with four decades of experience in software development and architecture. An independent consultant, he helps organizations adopt and master cloud native technologies, with expertise spanning containers, Java, microservices, Kubernetes, continuous delivery, and distributed computing. Jonathan has developed software across more than 20 product life cycles and a wide range of industries. He's passionate about helping teams design

and build scalable, resilient, and secure systems while sharing his expertise through workshops, presentations, and technical training worldwide.