## Sample Images

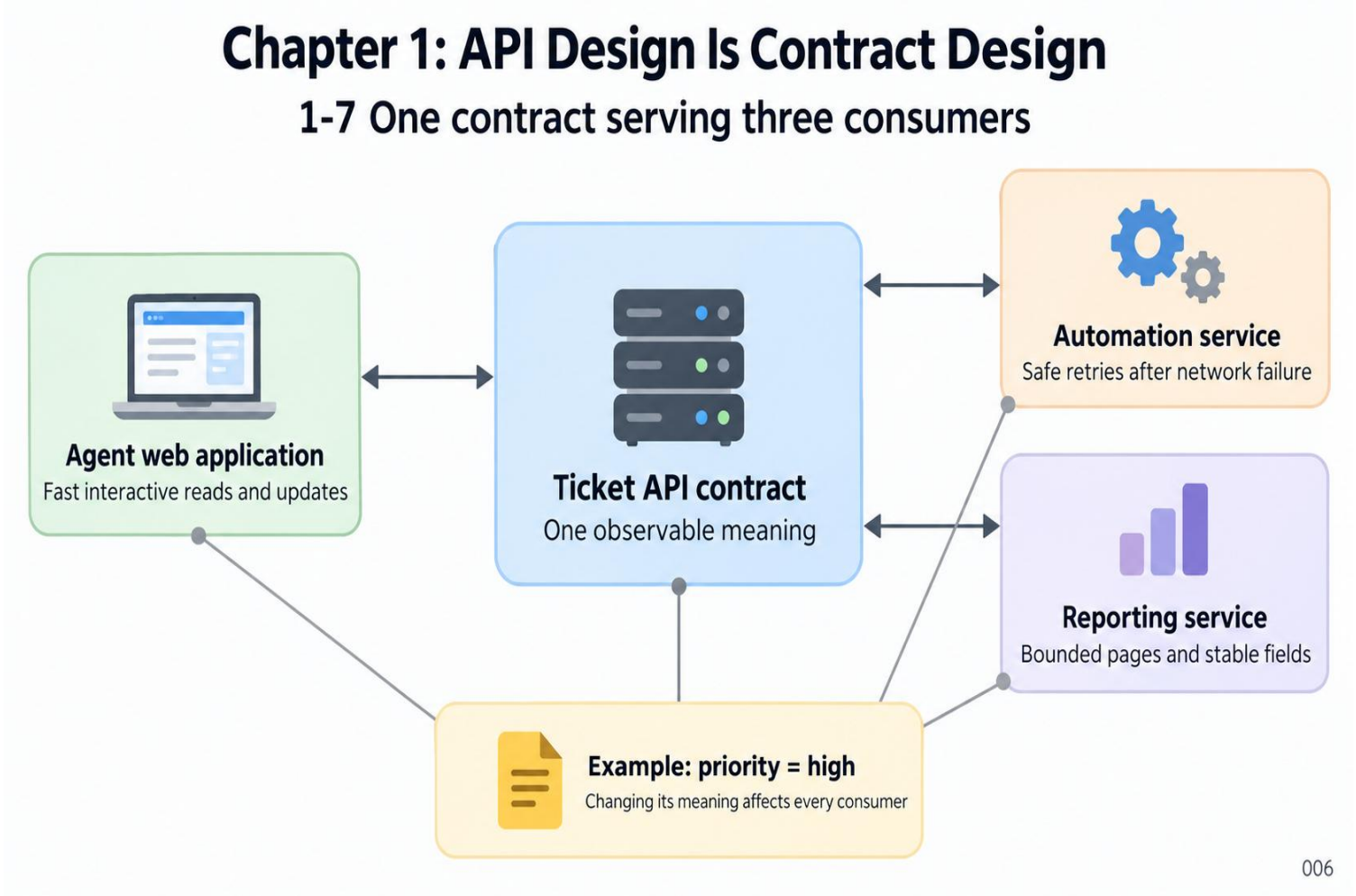



*Figure 0-1: Sample One contract serving three consumers*

## Sample Images (continued)




*Figure 0-2: Sample Test categories as complementary lenses*

# API Design for Systems Engineers

## Foundations, Practical Design, Security, Evolution, and Operations

# Introduction

This book is designed for readers who are taking their first steps with API design, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

## How to Use This Book

Read the chapters in order when studying the subject for the first time. Each chapter builds on terms, mental models, and design decisions introduced earlier, so following the sequence will help you develop the required background step by step.

Before reading a chapter, review its **What You'll Be Able to Do** section. The learning outcomes identify the practical abilities the chapter is intended to develop and help you distinguish the central ideas from supporting detail.

Use the diagrams to clarify relationships, processes, structures, state changes, responsibility boundaries, and data movement that may be difficult to picture from prose alone. The diagrams support the written explanation rather than replace it, so read the surrounding text before drawing conclusions from a visual.

Try the code, HTTP messages, and hands-on examples yourself whenever the necessary environment is available. Reproducing a small example and changing one input at a time is often the fastest way to confirm how a contract behaves.

At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you understand and can retain the fundamentals. Return to the relevant section when an answer is unclear rather than memorizing a phrase without its reasoning.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show why a mechanism exists, how it relates to the foundation, and when you may need to study it in greater depth later.

# Table of Contents


**Introduction**
**How to Use This Book**
**Main Text**
**Part I — Understanding the API Contract**
- **Chapter 1 — API Design Is Contract Design**
  - What You'll Be Able to Do
  - The API as an agreement
  - A boundary that hides change
  - Design, implementation, and operation are different responsibilities
  - Internal APIs still need discipline
  - Quality is part of the contract
  - The API lifecycle
  - The support-ticket contract in context
  - Two damaging misconceptions
    - "API design is mainly choosing URL paths"
    - "An internal API can be changed whenever the owning team wants"
  - Before approving an API concept
  - Chapter Summary
  - Review Questions
- **Chapter 2 — How an HTTP API Exchange Works**
  - What You'll Be Able to Do
  - The request-response model
  - The four parts of a request
  - Reading a response
  - A first command-line request
  - Method, URI, headers, and body work together
  - Headers carry message metadata
  - Stateless interactions
  - Intermediaries in the path
  - Following one ticket retrieval
  - Two common misunderstandings
    - "The URL alone defines what an HTTP API call does"
    - "Stateless means that the server cannot store any data"
  - A practical reading checklist
  - Chapter Summary
  - Review Questions
- **Chapter 3 — Turning Use Cases into an API Contract**
  - What You'll Be Able to Do
  - Start with consumers and outcomes
  - Establish a shared domain language
  - Describe behavior before endpoint shape
  - Separate business capability from implementation steps
  - Model workflows and state transitions
  - Capture validation and permission rules early
  - Include nonfunctional requirements
  - Use examples to expose ambiguity
  - Find the smallest coherent first contract
  - Decide where one API ends
  - Why generic CRUD is not enough
  - Two mistakes to prevent
    - "The database schema is the API model"
    - "A generic CRUD interface automatically covers the real workflow"
  - A use-case design worksheet
  - Chapter Summary
  - Review Questions

**Part II — Designing Everyday HTTP APIs**
- **Chapter 4 — Modeling Resources and Naming Endpoints**
  - What You'll Be Able to Do
  - What counts as a resource
  - Give resources stable identities
  - Use nouns and consistent path conventions
  - Model collections and members predictably
  - Represent relationships without over-nesting

- Compare plausible comment designs
- Distinguish resources from actions
- Avoid endpoints with unrelated responsibilities
- A coherent ticket resource structure
- Two recurring modeling errors
  - “REST requires every operation to be pure CRUD”
  - “Every database relationship should appear as a nested URL”
- A naming and modeling review
- Chapter Summary
- Review Questions

**Chapter 5 — Choosing Operations and HTTP Semantics**

- What You’ll Be Able to Do
- Choose methods by observable behavior
- Retrieval belongs on GET
- Creation commonly uses POST
  - Creating a ticket with curl
- Use PUT for full replacement at a known URI
- Use PATCH for defined partial modification
- Use DELETE for removal semantics
- Select success status codes deliberately
- Redirects need explicit meaning
- Do not assume every message supports a body equally
- Applying semantics to ticket operations
- Two high-impact misunderstandings
  - “Idempotent means every repeated response must be identical”
  - “Any successful request should return 200 OK”
- Operation review questions
- Chapter Summary
- Review Questions

**Chapter 6 — Designing Requests and Responses**

- What You’ll Be Able to Do
- A representation is not the resource itself
- JSON provides a small set of value types
- Use one field-naming convention
- Missing, null, empty, and default are different states
- Define field direction and ownership
- Choose types for meaning, not convenience
  - Identifiers
  - Timestamps and dates
  - Durations
  - Money and exact decimals
  - Enumerated values
- Make the creation and retrieval contracts distinct
- Validate at several levels
- Defaults should be observable and stable
- Content negotiation at a practical level
- File transfer is an HTTP-specific design choice
- Two data-contract mistakes
  - “Optional and nullable mean the same thing”
  - “The safest response is to expose every stored field”
- A field-level design check
- Chapter Summary
- Review Questions

**Chapter 7 — Designing Predictable Errors**

- What You’ll Be Able to Do
- HTTP success and business success must agree
- Use status codes for broad categories
- Give errors a stable machine-readable shape
- Stable codes and human messages have different jobs
- Return field-level validation details
- Four practical ticket failures
  - A missing ticket
  - An invalid priority
  - A forbidden assignment
  - A conflicting state transition
- Distinguish retriable and non-retriable failures
- Correlation identifiers support diagnosis

- Do not leak sensitive internals
- Two damaging error patterns
  - “Returning 200 OK with an error flag is simpler for clients”
  - “A stack trace helps consumers diagnose the problem”
- Error-contract review
- Chapter Summary
- Review Questions

**Chapter 8 — Designing Collection Endpoints**

- What You’ll Be Able to Do
- Give collections a stable response shape
- Pagination is a contract decision
- Offset and cursor pagination solve different problems
- Treat cursor tokens as opaque
- Ordering must be deterministic
- Filtering, sorting, and search are distinct
- Define query-parameter conventions
- Set default and maximum page sizes
- Retrieve a filtered page with curl
- Continuation links can reduce client mistakes
- Total counts are not free
- Understand changes between pages
- Sparse field selection is optional
- Two collection-design mistakes
  - “Pagination can be added later without affecting the contract”
  - “Search, filtering, and sorting are interchangeable”
- Collection endpoint review
- Chapter Summary
- Review Questions

**Part III — Protecting Correctness and Access**

**Chapter 9 — Making State Changes Safe**

- What You’ll Be Able to Do
- Begin with the resource lifecycle
- Why duplicate requests happen
- Prefer naturally idempotent operations where they fit
- Use idempotency keys for retryable logical operations
  - One key represents one logical attempt
- Define idempotency-key behavior completely
  - Scope
  - Request matching
  - In-progress duplicates
  - Completed duplicates
  - Retention period
  - Failure behavior
- Create safely with an idempotency key
- Idempotency keys do not replace business uniqueness
- Concurrent updates create lost changes
- Optimistic concurrency detects stale writes
- Recover from a concurrency conflict deliberately
- Use the right failure for the right conflict
- Timestamps alone are a weak concurrency mechanism
- Define the transaction boundary
- Avoid exposing multi-call transactions to ordinary clients
- Treat batch operations as a separate design problem
  - All-or-nothing batch
  - Per-item results
- Two correctness mistakes
  - “The client can prevent all duplicates by disabling its submit button”
  - “Last write wins is harmless for ordinary business data”
- State-change design checklist
- Chapter Summary
- Review Questions

**Chapter 10 — Designing Security Boundaries**

- What You’ll Be Able to Do
- Start with assets, callers, and trust boundaries
- HTTPS is the baseline, not the complete security model
- Authentication proves an identity
- OAuth 2.0 and OpenID Connect have different jobs

- Authorization decides what the identity may do
- Scopes, roles, claims, and resource checks serve different purposes
- Object-level authorization prevents cross-resource access
- Least privilege limits the effect of mistakes and compromise
- Walk through four ticket operations
  - Reading a ticket
  - Assigning a ticket
  - Adding a comment
  - Deleting a ticket
- Input validation is not authorization
- Keep secrets and sensitive data out of exposed locations
- Credentials need a lifecycle
- CORS controls browsers, not general API authorization
- Rate limits and abuse controls complement permission checks
- Two security mistakes with large consequences
  - “A valid access token means the caller may access every resource”
  - “CORS protects an API from unauthorized non-browser clients”
- Security boundary review
- Chapter Summary
- Review Questions

**Part IV — Making the Contract Usable and Verifiable**

**Chapter 11 — Describing and Documenting the Contract**

- What You’ll Be Able to Do
- Documentation has two audiences
- OpenAPI describes HTTP API structure
- A minimal ticket description
- A schema is necessary but not sufficient
- Use examples to make abstract rules concrete
- Describe formats, defaults, and constraints precisely
- Choose contract-first or code-first consciously
- Reuse schemas without hiding operation meaning
- Document security where consumers need it
- Make the error contract discoverable
- Generated artifacts accelerate work but do not replace judgment
- Keep description, implementation, and tests aligned
- Changelogs and migration notes answer different questions
- Two documentation misconceptions
  - “Generating an OpenAPI file means the API is well designed”
  - “Examples are optional because the schema is sufficient”
- Documentation readiness check
- Chapter Summary
- Review Questions

**Chapter 12 — Testing and Reviewing an API Contract**

- What You’ll Be Able to Do
- Test behavior, not only endpoint reachability
- Use a compact set of test categories
- Build the test matrix from contract decisions
- Schema validation checks message structure
- Conformance tests compare the provider with the contract
- Integration tests cross the real network boundary
- Consumer-driven contract tests capture real consumer expectations
- Mocks help early work but can preserve a false assumption
- Test permissions as a matrix, not one denial
- Exercise boundaries where mistakes cluster
- Reproduce retries and duplicate delivery
- Test timeouts at uncertain points
- Verify rate limits and partial failures
- Include basic security and robustness checks
- Check compatibility against the previous contract
- Review the design before testing implementation
- Separate contract defects from implementation defects
- Two testing misconceptions
  - “A 200 response proves the API works”
  - “Generated clients eliminate the need for contract testing”
- Release-level verification checklist
- Chapter Summary
- Review Questions

## Part V — Evolving and Operating APIs

### Chapter 13 — Evolving an API Without Breaking Clients

- What You'll Be Able to Do
- Compatibility is judged from the consumer side
- Classify changes before implementing them
- Additive changes can still break consumers
- Enumeration values deserve special care
- Optional response fields need clear absence semantics
- Versioning identifies a contract line; it does not ensure compatibility
- Version only when the compatibility boundary requires it
- Deprecation is a managed period, not a warning label
- Measure usage before removal
- Parallel versions carry continuing operational cost
- Capability negotiation can reduce unnecessary versions
- Contract migration and data migration are related but different
- Example: introducing a new priority
- Example: evolving assignment from one user to a richer model
- Plan rollout as a sequence of reversible steps
- Two compatibility mistakes
  - "Adding a field can never break a client"
  - "Putting /v2 in the URL solves compatibility"
- Compatibility decision checklist
- Chapter Summary
- Review Questions

### Chapter 14 — Designing for Failure, Load, and Performance

- What You'll Be Able to Do
- Bound waiting with timeouts and deadlines
- A timeout creates uncertainty for state-changing requests
- Retry only failures that may improve on another attempt
- Use backoff and jitter to avoid synchronized retries
- Retries multiply across service layers
- Rate limits protect capacity and fairness
- Load shedding preserves essential service
- Caching changes observable behavior
- Decide freshness from the use case
- Control payload size deliberately
- Watch request fan-out
- Partial dependency failure needs an honest response
- Circuit breakers limit repeated calls to an unhealthy dependency
- Bulk and asynchronous work can reduce synchronous pressure
- Service-level objectives shape contract decisions
- Example: ticket search during high traffic
- Example: ticket creation after a client timeout
- Two reliability misconceptions
  - "Every failed request should be retried immediately"
  - "Caching is only an implementation optimization and has no contract implications"
- Failure and load design check
- Chapter Summary
- Review Questions

### Chapter 15 — Asynchronous Operations, Webhooks, and Events

- What You'll Be Able to Do
- Choose synchronous interaction only when waiting is reasonable
- 202 Accepted means accepted for processing
- Model long-running work as an operation resource
- Define terminal behavior and retention
- Cancellation is a request, not a guaranteed reversal
- Poll without creating avoidable load
- A webhook lets the provider call the consumer
- Verify endpoint ownership before sending sensitive events
- Sign deliveries and prevent replay
- Acknowledge only after durable acceptance
- Duplicate webhook delivery is normal
- Events use publish-subscribe communication
- Delivery guarantees describe outcomes, not magic settings
- Ordering is limited in scope
- Correlate messages across a workflow
- Commands, events, and resource representations have different purposes

- Evolve event schemas conservatively
- Use events only when the coupling trade-off helps
- Complete export and assignment-notification designs
- Two asynchronous-design misconceptions
  - “Returning 202 Accepted means the operation succeeded”
  - “A webhook will be delivered exactly once and in order”
- Asynchronous contract review
- Chapter Summary
- Review Questions

**Chapter 16 — Designing for Observability and Operations**

- What You’ll Be Able to Do
- Observability turns signals into understanding
- Distinguish the main operational signals
- Give every request a usable identifier
- Correlation connects several requests and messages
- Structured logs are easier to search and govern
- Log decisions, not every piece of data
- Metrics show patterns over time
- Control metric cardinality
- Monitor by operation and consumer where it changes action
- Distributed traces follow work across components
- Audit records support accountability
- Separate client errors from internal evidence
- Health and readiness are operational interfaces
- Health does not prove business correctness
- Runbooks connect signals to action
- Trace a failed assignment end to end
- Two observability mistakes
  - “Logging every request body provides the best diagnostics”
  - “A health endpoint proves every business operation is functioning correctly”
- Operational readiness review
- Chapter Summary
- Review Questions

**Part VI — Choosing and Applying an API Style**

**Chapter 17 — Choosing Among API Styles**

- What You’ll Be Able to Do
- Start with the interaction, not the technology name
- Resource-oriented HTTP APIs fit broad request-response use
- RPC exposes named operations
- gRPC provides strongly typed RPC and streaming
- GraphQL lets clients select fields from a schema
- Event-driven APIs publish facts asynchronously
- SOAP and WSDL remain relevant in some environments
- Compare styles from the same perspectives
- Consider client diversity and control
- Type systems help, but they do not define business truth
- Streaming is a distinct requirement
- Caching differs significantly by style
- Operational complexity belongs in the decision
- Styles can coexist within one system
- Apply the criteria to four ticket needs
  - Public ticket lookup
  - Low-latency service-to-service assignment
  - Flexible analytics queries
  - Ticket-change notifications
- Avoid technology selection by fashion or familiarity alone
- Two style-selection misconceptions
  - “REST, GraphQL, and gRPC are interchangeable transport formats”
  - “A modern API should use one style everywhere”
- API-style selection questions
- Chapter Summary
- Review Questions

**Chapter 18 — Applying the Design Process on a Team**

- What You’ll Be Able to Do
- Keep the work connected from discovery to retirement
- Assign clear ownership
- Define outcomes and constraints before endpoint details

Use example interactions before completing the schema
Produce a small set of connected artifacts
Record significant decisions and alternatives
Review with the people who bear the consequences
Turn review comments into contract consequences
Formalize the contract after the core behavior is understood
Plan release before implementation is complete
Style guides create defaults, not automatic answers
Catalogs make APIs discoverable and accountable
Governance should be proportional to risk
Know when specialist help is necessary
End-to-end case study: the support-ticket API
Design brief
Representative resource and operation set
Core ticket representation
Permission summary
Key design decisions
Representative ticket creation
Representative assignment
Representative collection retrieval
Asynchronous export and webhook
OpenAPI excerpt
Verification plan
Release notes
Operational readiness
Use a practical design-review checklist
Consumer and domain
Contract behavior
Security and privacy
Compatibility and lifecycle
Verification and operation
Two team-process misconceptions
“Governance means a central team must approve every small decision”
“The contract is finished when implementation begins”
The durable working model
Chapter Summary
Review Questions
**Closing Notes**

## Main Text

**An API should be designed as a dependable contract between systems, not as a thin wrapper around application code or database tables.** The contract tells consumers what they may request, what the provider promises to return, which failures can occur, and which behaviors remain stable as the implementation changes.

This guide uses a JSON-based HTTP API over HTTPS as its primary teaching environment. That is one common environment for project work, not a universal rule for all APIs. The recurring example is a support-ticket service used by three consumers:

- A web application used by support agents.
- An automation service that creates and updates tickets.
- A reporting service that reads ticket data for analysis.

The chapters progress from the basic contract model to everyday HTTP API design, safe state changes, security, documentation, testing, compatibility, reliability, asynchronous communication, observability, API-style selection, and team governance. The most important habit throughout is to ask: **What may a consumer safely rely on?**

# Part I — Understanding the API Contract

# Chapter 1 — API Design Is Contract Design

**The central idea is that an API is a long-lived agreement about observable behavior.** URLs and payloads matter, but they are only parts of that agreement. A useful design also defines meanings, permissions, failures, timing expectations, compatibility rules, and ownership.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Explain why an API is a contract for observable behavior rather than merely a set of URLs.
- Distinguish API design, implementation, and operation responsibilities.
- Identify coupling, quality attributes, lifecycle stages, and ownership concerns that shape a durable boundary.

## The API as an agreement

An **application programming interface (API)** is a defined way for one software component to request capabilities or exchange data with another. The component that offers the API is the **provider**. A component that calls it is a **client** or **caller**. The word **consumer** is broader: it may refer to a client application, another service, a team, or an organization that depends on the API. A **downstream dependency** is a system that the provider itself must call or rely on to complete some requests.

Suppose a support-ticket provider offers an operation for retrieving a ticket. The contract may promise that:

- A ticket has a stable identifier.
- A successful retrieval returns a specific data structure.
- A missing ticket produces a defined error.
- A caller must have permission to view the ticket.
- Existing fields will not silently change meaning.

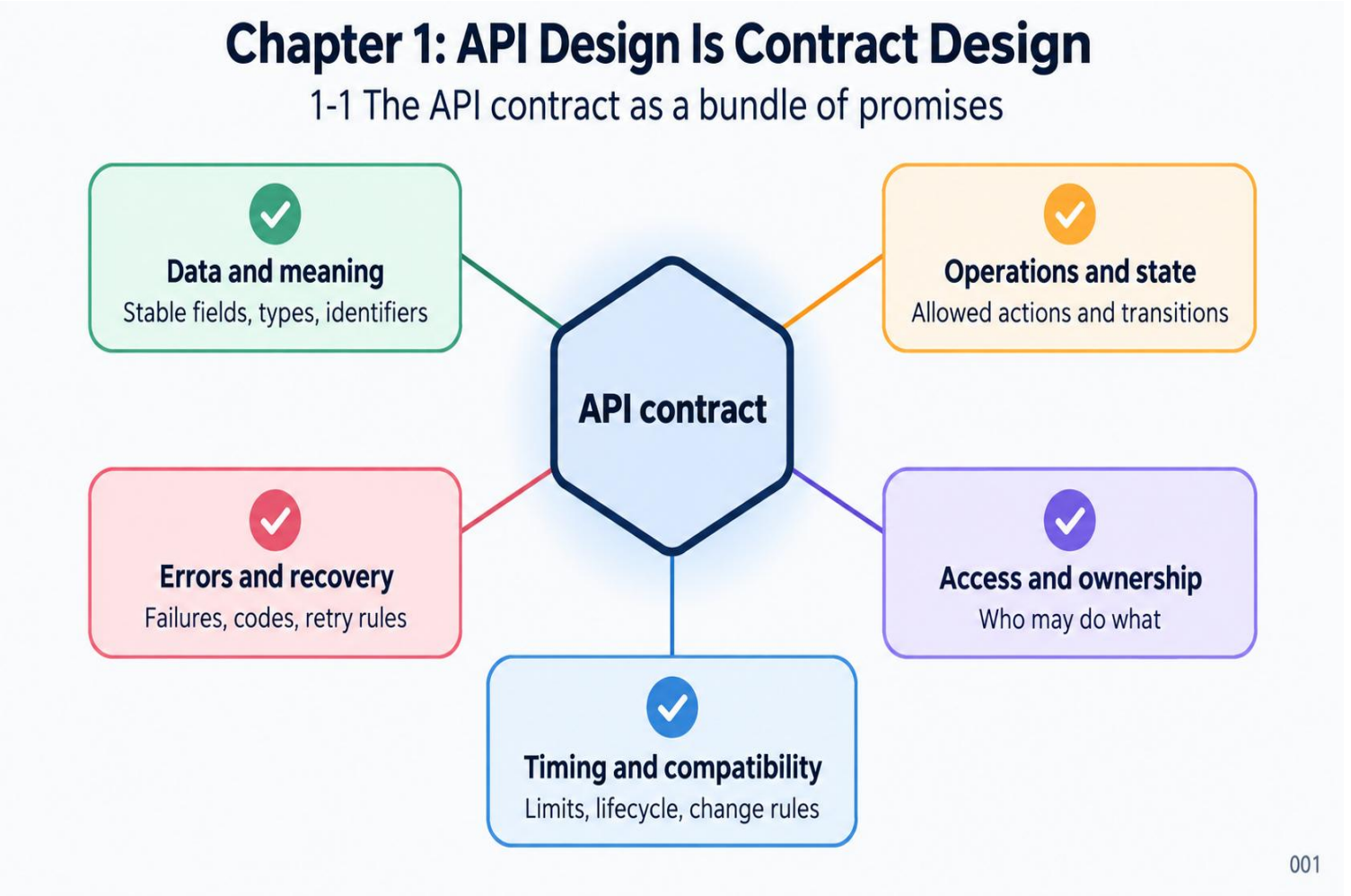



*Figure 1-1: The API contract as a bundle of promises*

Those promises allow the web application, automation service, and reporting service to be developed independently. The provider may rewrite its code or replace its database without forcing those consumers to change, provided that the observable contract remains compatible.

A useful everyday analogy is a restaurant menu. The kitchen is free to reorganize its equipment or change how it prepares a dish, but a customer relies on the menu's name, description, price, and ordering rules. If the kitchen silently changes "vegetable soup" into a meat dish, the problem is not an internal implementation detail; it is a broken contract.

## A boundary that hides change

A **system boundary** is the point at which one part of a system interacts with another through defined inputs and outputs. A well-designed API exposes supported behavior at that boundary while hiding implementation details.

This hiding is valuable because implementations change frequently. A ticket service might move from one database product to another, split one internal module into several services, or calculate priority differently. Consumers should not need to know unless the public behavior actually changes.

The opposite condition is **coupling**, which means that one component depends on details of another component. Some coupling is unavoidable: a consumer must depend on the API contract. The design goal is to keep that coupling intentional and stable rather than accidental.

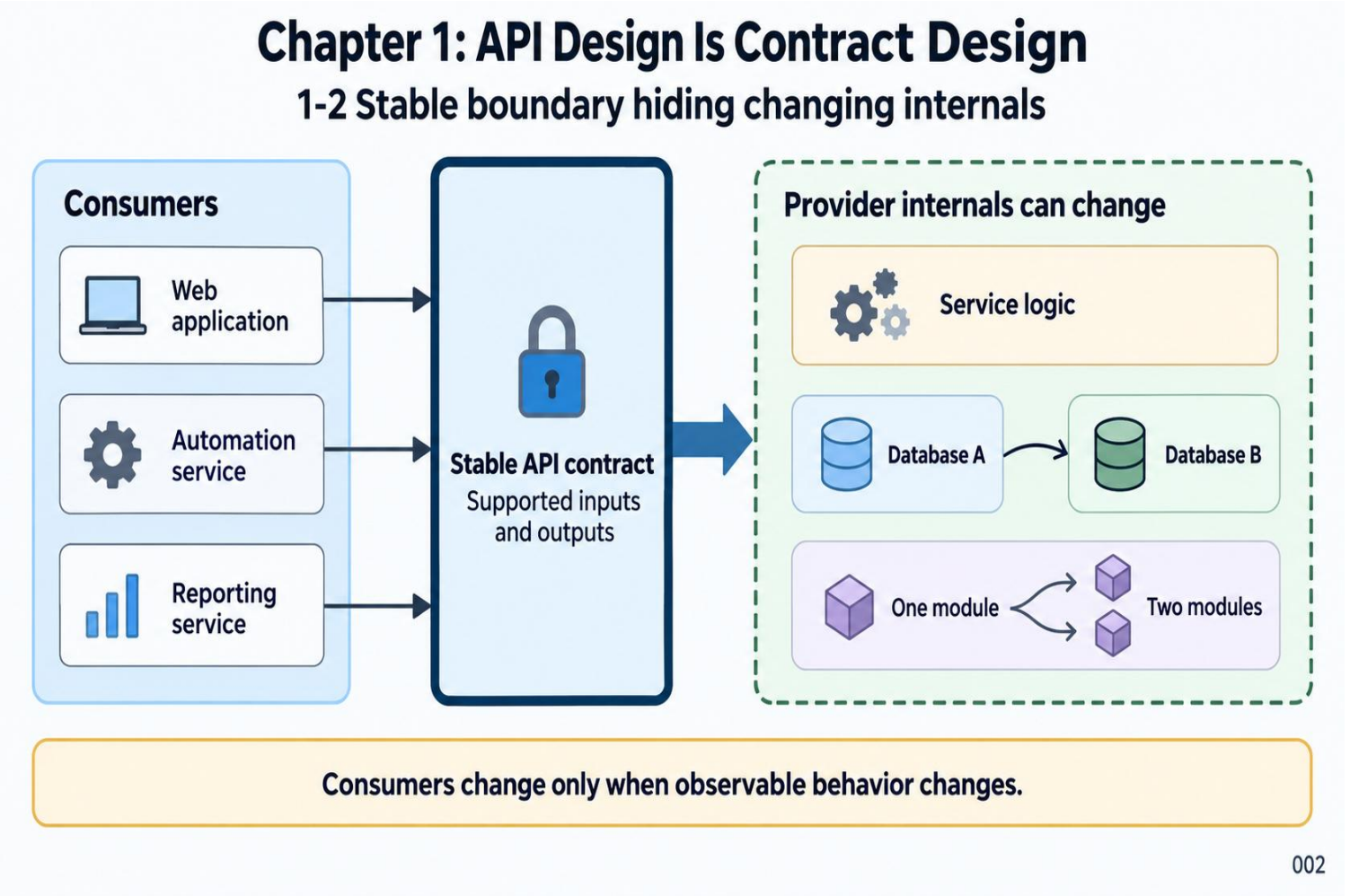



*Figure 1-2: Stable boundary hiding changing internals*

For example, consider an API response copied directly from a database row:

```
{
  "ticket_pk": 48271,
  "status_cd": 3,
  "owner_fk": 991,
  "row_ver": 17
}
```

The field names and numeric codes reveal storage choices but do not explain business meaning. A clearer contract might be:

```
{
  "id": "tkt_48271",
  "status": "in_progress",
  "assigneeId": "usr_991",
  "updatedAt": "2026-09-18T09:42:31Z"
}
```

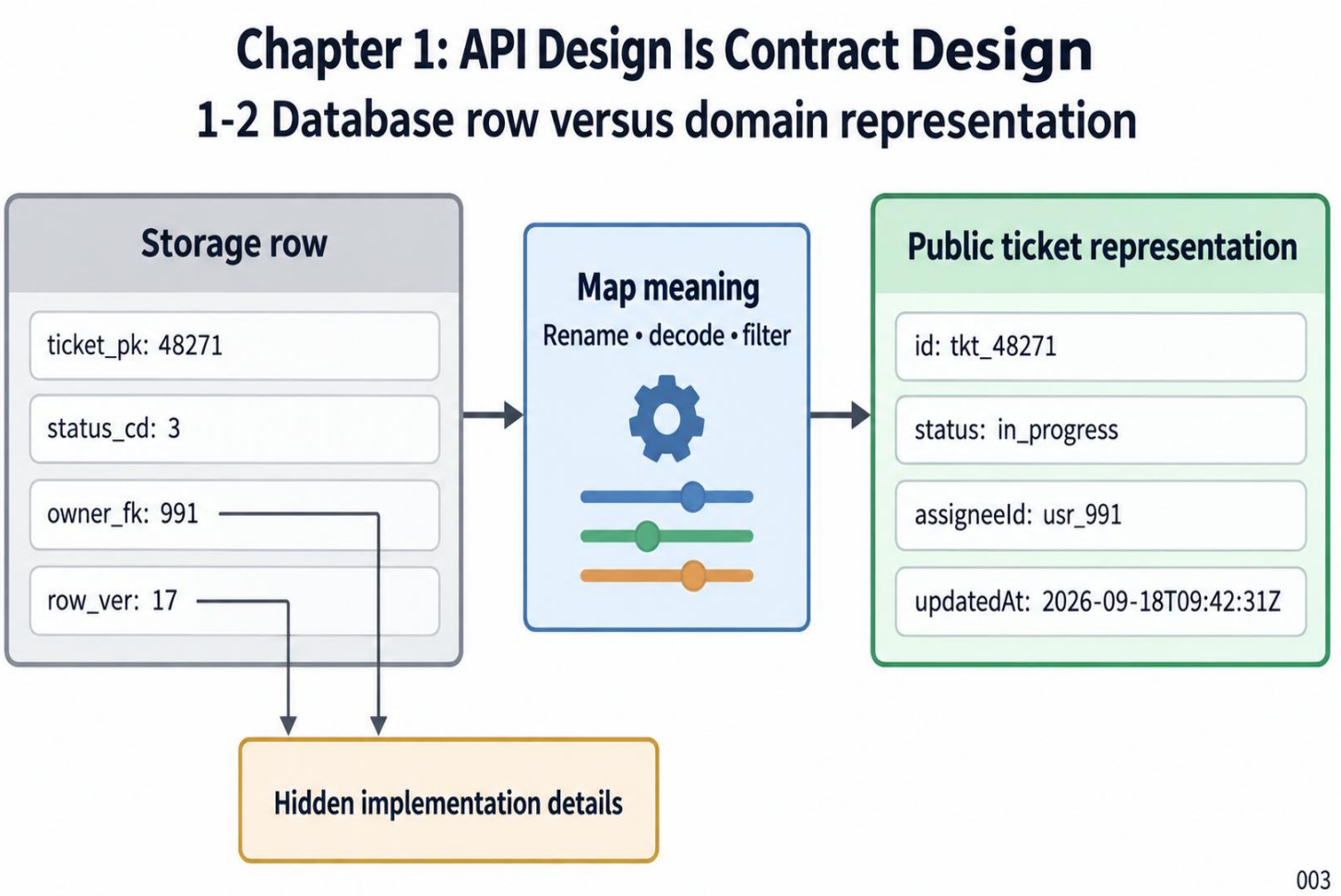



*Figure 1-3: Database row versus domain representation*

The second representation still has implementation behind it, but the contract uses domain concepts that consumers can understand. The provider can later change primary-key types, table names, or versioning columns without exposing those changes.

## Design, implementation, and operation are different responsibilities

These three areas influence one another, but they answer different questions.

| Area | Main question | Ticket-service example |
|---|---|---|
| API design | What behavior and data may consumers rely on? | Which fields identify a ticket, how assignment works, and which failures are defined. |
| API implementation | How does the provider produce that behavior? | Which framework, database, and internal classes are used. |
| API operation | How is the running API kept available and diagnosable? | Deployment, monitoring, capacity, incident response, and runtime configuration. |

Confusing these areas leads to weak decisions. “Our framework automatically maps this class to JSON” is an implementation fact, not evidence that the resulting data contract is appropriate. “The database can update this column” does not prove that consumers should be allowed to perform that update.

**Design begins with consumer-visible behavior. Implementation choices must support that behavior, not define it accidentally.**

## Internal APIs still need discipline

APIs are often described as public, partner, or internal.

- A **public API** is offered to a broad external audience.

- A **partner API** is shared with selected external organizations under an agreed relationship.
- An **internal API** is used within one organization, often between teams or services.

The labels affect documentation, onboarding, support, and risk, but none makes contract design optional. An internal API can be especially difficult to change because its consumers may be numerous, poorly recorded, or deployed on different schedules. The fact that two teams work for the same organization does not make a breaking change inexpensive.

A practical rule is: **Treat every API as independently consumed unless deployment and ownership arrangements prove otherwise.** Even when provider and client are currently released together, that condition may change as the system grows.

## Quality is part of the contract

Functional requirements describe what an API must do. For the ticket service, examples include creating a ticket, assigning an agent, adding a comment, and retrieving open tickets.

A design also needs **quality attributes**, which are properties that describe how well the API must work. Common API quality goals include:

- **Clarity:** Consumers can understand the contract without reverse-engineering server behavior.
- **Consistency:** Similar operations use similar names, structures, and rules.
- **Security:** Only authorized callers can access or change protected data.
- **Reliability:** The API behaves predictably during normal failures and retries.
- **Performance:** Response time and payload size are suitable for the expected use.
- **Evolvability:** The provider can add capabilities without routinely breaking existing consumers.

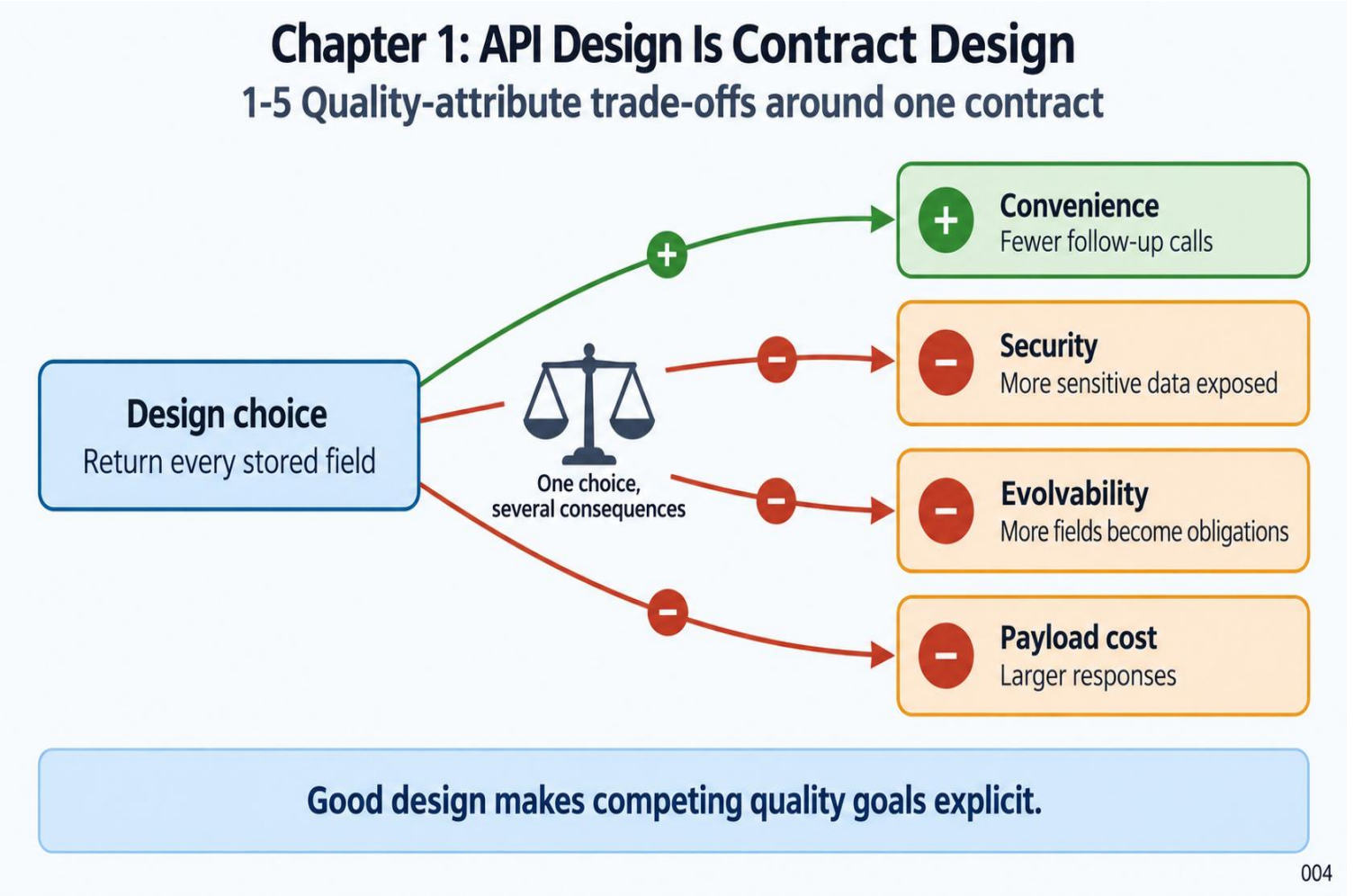



*Figure 1-4: Quality-attribute trade-offs around one contract*

These goals sometimes compete. Returning every ticket field in one response may appear convenient, but it can increase payload size, expose sensitive information, and create more compatibility obligations. Good design makes such trade-offs explicit.

## The API lifecycle

An API passes through more stages than implementation and release. A practical lifecycle is:

1. **Discovery:** Identify consumers, business goals, constraints, and existing systems.
2. **Design:** Define resources, operations, data, errors, permissions, and behavior.
3. **Review:** Check the contract with consumer, domain, security, and operations representatives.
4. **Implementation:** Build the provider according to the agreed contract.
5. **Testing:** Verify successful behavior, failures, permissions, and compatibility.
6. **Release:** Publish the contract, documentation, and migration information.
7. **Operation:** Observe usage, performance, errors, and abuse.
8. **Evolution:** Add or change capabilities while protecting consumers.
9. **Retirement:** Deprecate and remove an API through a managed process.

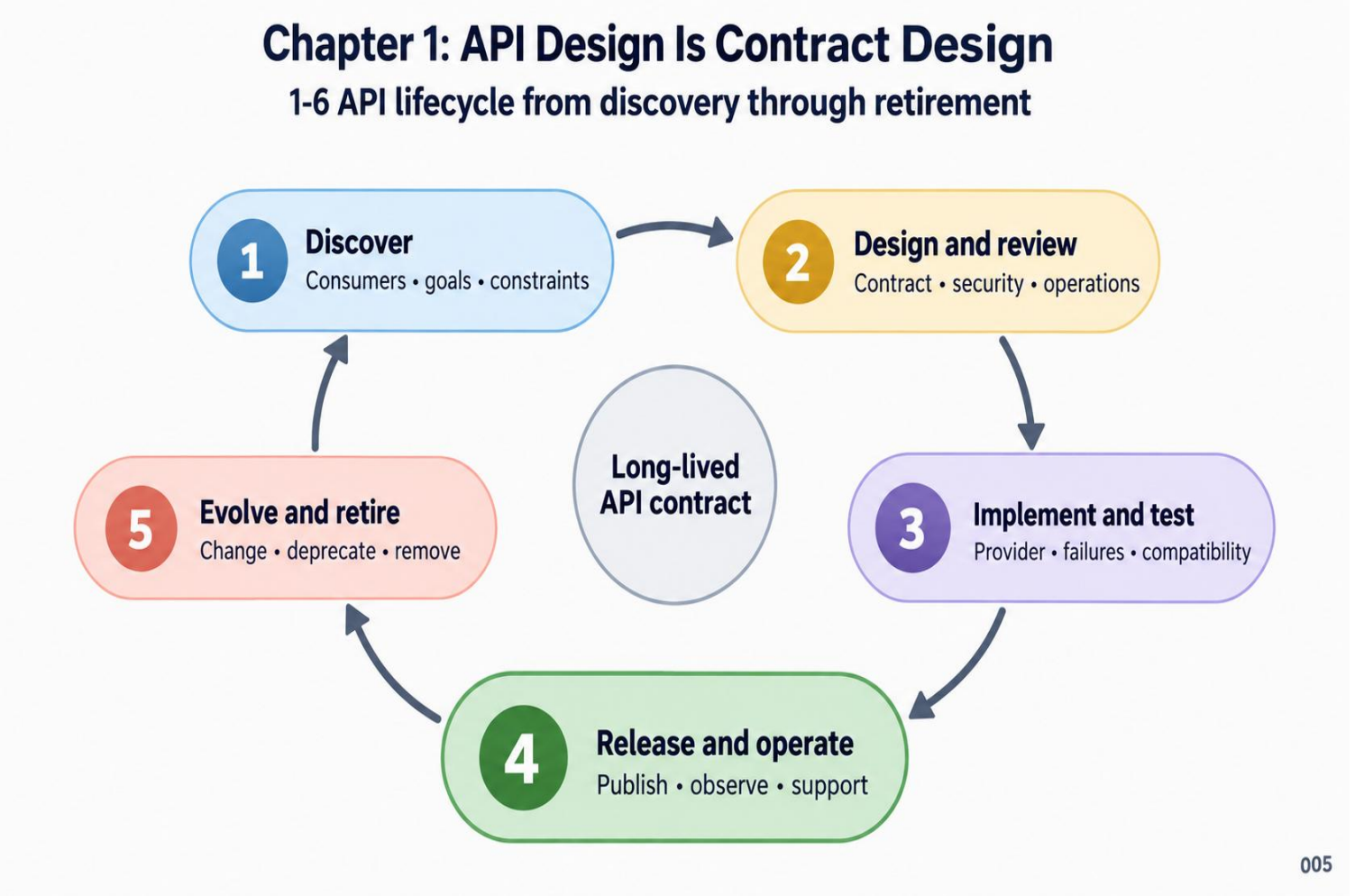



*Figure 1-5: API lifecycle from discovery through retirement*

The important point is that **the contract continues to exist after the code is deployed**. Real consumers will discover assumptions that were not obvious during development. Operational evidence may show that a limit is too small or an error is too vague. Design therefore remains an active responsibility.

## The support-ticket contract in context

Consider three consumers:

- The agent web application needs fast interactive reads and updates.
- The automation service needs safe retries because network failures can occur during batch processing.
- The reporting service needs bounded collection retrieval and stable meanings for fields over time.

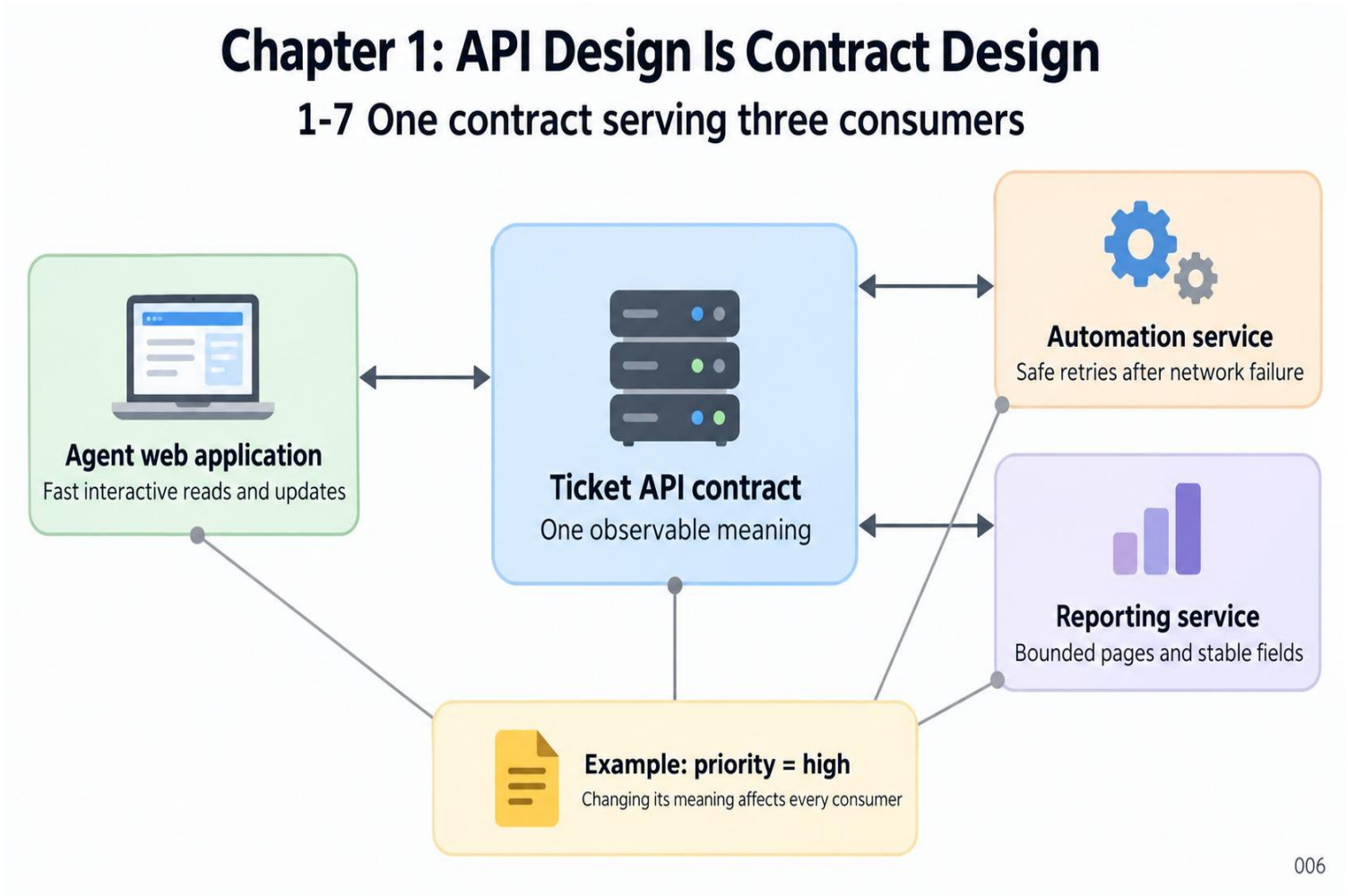



*Figure 1-6: One contract serving three consumers*

A single careless choice can affect them differently. For example, changing `priority` from a named value such as `high` to a number such as `3` may require user-interface changes, automation changes, and report corrections. The database migration might be simple, while the contract migration is expensive.

This is why design review must ask not only “Can the server implement this?” but also:

- Can each consumer understand it?
- Can each consumer recover from failure?
- Can the provider change its internals later?
- Can the contract evolve without coordinated replacement of every client?

## Two damaging misconceptions

### “API design is mainly choosing URL paths”

Paths are visible and easy to debate, so teams may spend disproportionate time on them. A path such as `/tickets/123` says little by itself. The complete contract also includes the method, input fields, status codes, response body, authorization rules, retry behavior, concurrency behavior, and compatibility expectations.

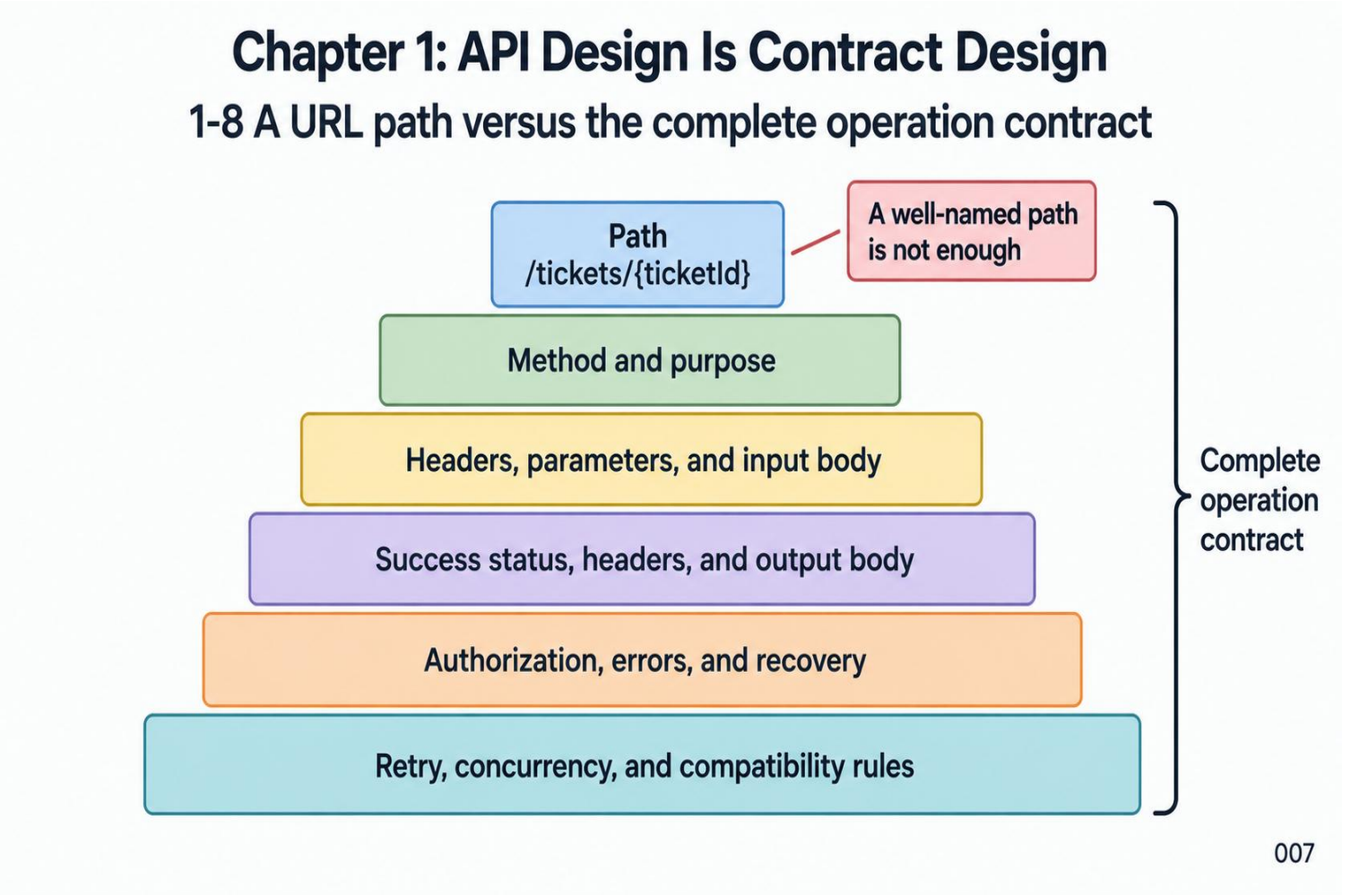



*Figure 1-7: A URL path versus the complete operation contract*

A well-named path cannot compensate for ambiguous behavior.

### "An internal API can be changed whenever the owning team wants"

Ownership gives a team responsibility for the API; it does not erase consumer dependencies. Before changing an internal API, the provider needs to know who uses it, whether the change is compatible, and how consumers will migrate.

## Before approving an API concept

Confirm that the proposed API has:

- Named consumers and concrete use cases.
- An explicit provider and owner.
- Consumer-oriented domain language rather than raw implementation structures.
- Defined successful behavior and important failure behavior.
- Quality goals appropriate to expected use.
- A plan for review, release, operation, and future change.

**The foundation for every later design decision is the same: expose a clear, stable boundary and keep internal choices behind it.**

## Chapter Summary

- An API gives consumers a stable boundary while allowing the provider's internals to change.
- The contract includes meanings, permissions, failures, quality expectations, compatibility, and ownership—not only paths and payloads.
- Internal APIs require the same design discipline because consumers and release schedules can diverge over time.

## Review Questions

1. What may an API consumer rely on besides a URL and a JSON structure?
2. Why is exposing a database row directly likely to create harmful coupling?
3. Which API lifecycle activities continue after the first production release?

# Chapter 2 — How an HTTP API Exchange Works

**Every HTTP API call can be understood as one request message followed by one response message.** Each message has a small number of parts. Once those parts are clear, endpoint definitions and troubleshooting become much easier.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Identify the method, target URI, headers, and optional body in an HTTP request.
- Read a response by interpreting its status code, headers, and optional representation.
- Explain how HTTPS, statelessness, gateways, proxies, and load balancers affect an API exchange.

## The request-response model

**Hypertext Transfer Protocol (HTTP)** is an application protocol that defines how a client sends a request and how a server returns a response. An HTTP API uses that exchange to expose system capabilities.

**HTTPS** is HTTP carried over a protected transport connection that encrypts data in transit and authenticates the server to the client. In ordinary API design, HTTPS should be treated as the baseline rather than an optional enhancement.

A simplified exchange looks like this:

1. The client connects to the API host over HTTPS.
2. The client sends an HTTP request.
3. The server identifies the requested operation and processes it.
4. The server sends an HTTP response.
5. The client interprets the status, metadata, and body according to the contract.

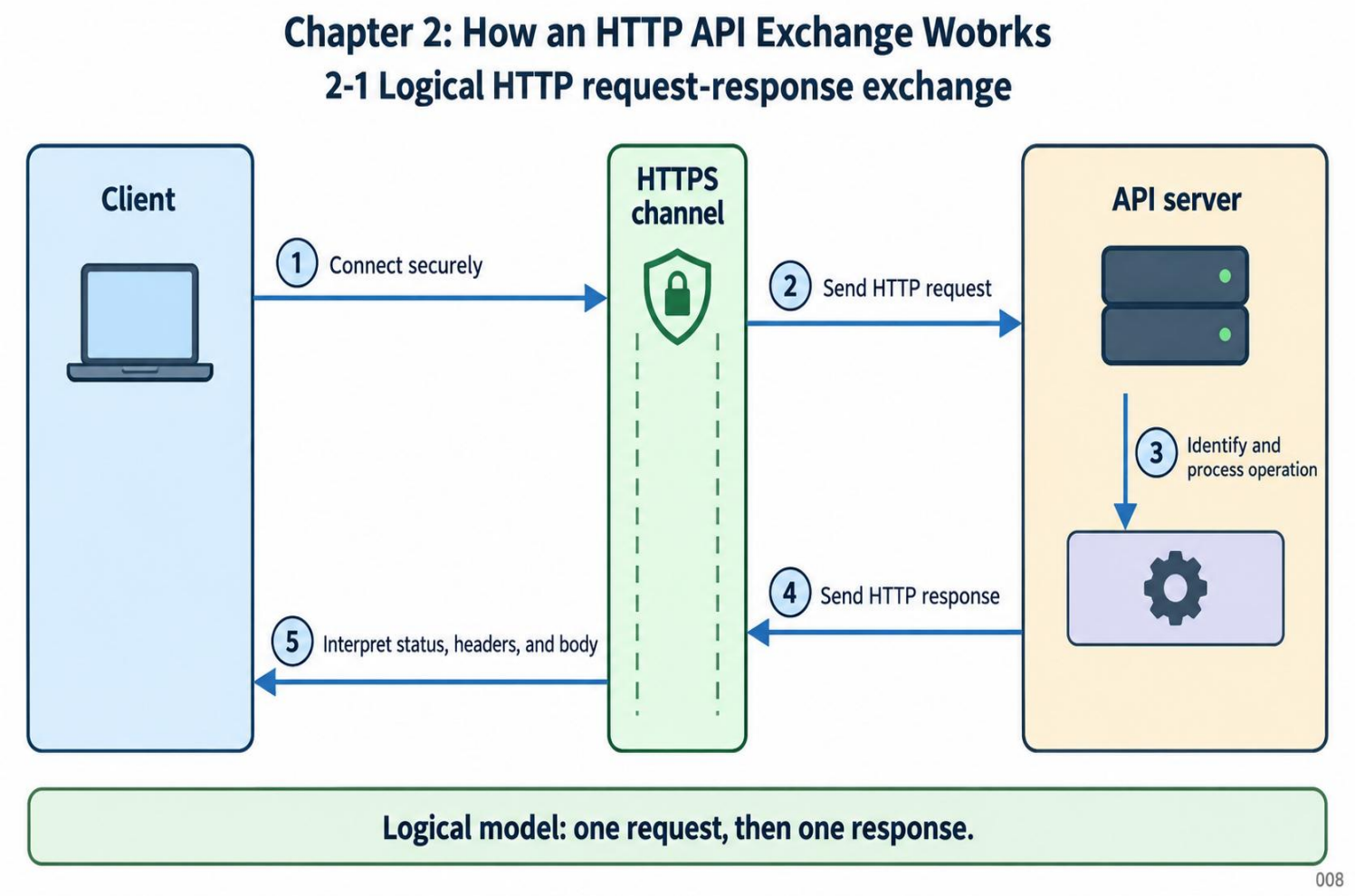



*Figure 2-1: Logical HTTP request-response exchange*

The network may contain intermediaries, but the logical exchange is still request followed by response.

## The four parts of a request

An HTTP request contains:

1. A **method**, which indicates the intended semantics of the request, such as retrieving or creating data.
2. A **target URI**, which identifies the addressed resource or operation.
3. **Headers**, which carry metadata about the message and its handling.
4. An optional **body**, which carries input data when the operation needs it.

A **Uniform Resource Identifier (URI)** is a string used to identify a resource or operation. In common HTTPS APIs, it appears in a form such as `https://api.example.com/tickets/tkt_48271`.

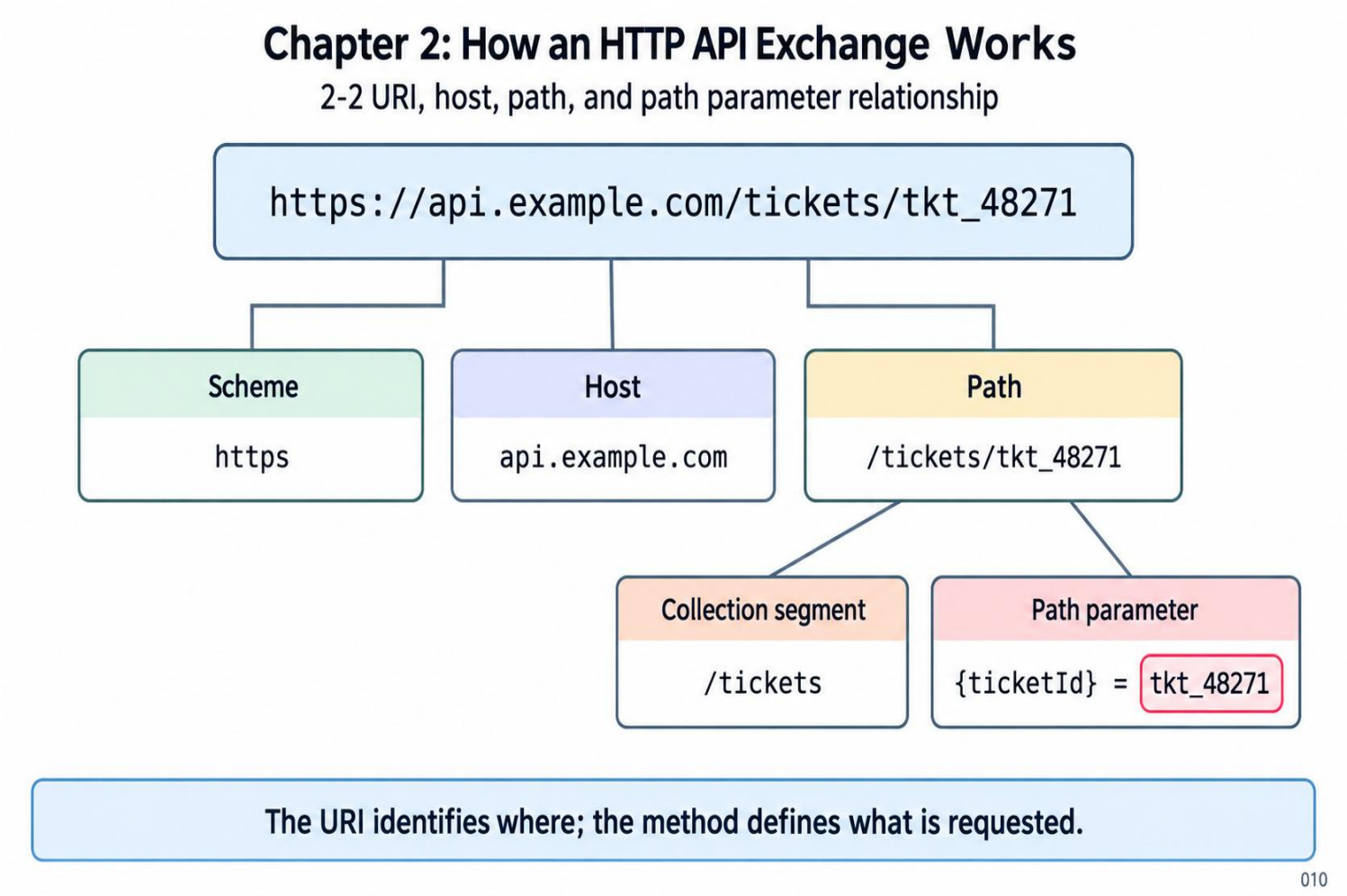



*Figure 2-3: URI, host, path, and path parameter relationship*

Here is a minimal request in raw HTTP form:

```
GET /tickets/tkt_48271 HTTP/1.1
Host: api.example.com
Authorization: Bearer <access-token>
Accept: application/json
```

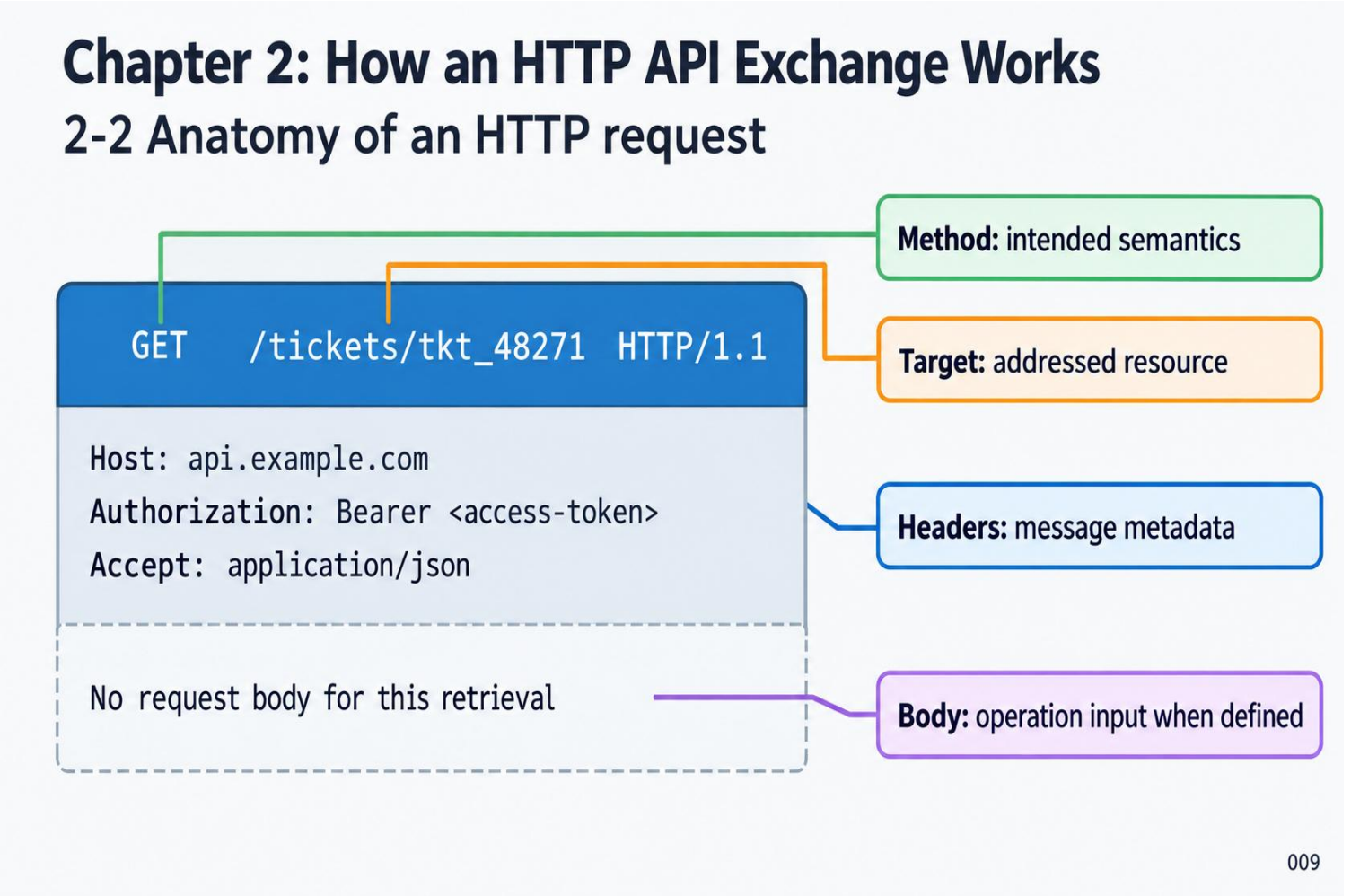



*Figure 2-2: Anatomy of an HTTP request*

The important lines are:

- `GET` is the method.
- `/tickets/tkt_48271` is the target path on the host.
- `Authorization` carries an access token in this example; an **access token** is a credential that represents a caller's granted access, and `Bearer` names the credential scheme used here.
- `Accept: application/json` says that the client can receive JSON.

**JavaScript Object Notation (JSON)** is a text format for structured data using objects, arrays, strings, numbers, booleans, and `null`. It is common in HTTP APIs because it is widely supported and human-readable.

A **media type** identifies the format and intended interpretation of message content. `application/json` is the media type for ordinary JSON data. A **schema** is a formal definition of the fields, types, and constraints allowed in a message.

## Reading a response

An HTTP response contains:

1. A **status code**, which gives a standardized summary of the outcome.
2. Headers containing response metadata.
3. An optional body containing a representation, result, or error.

A possible successful response is:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "status": "open",
  "priority": "normal",
  "createdAt": "2026-09-18T09:42:31Z"
}
```

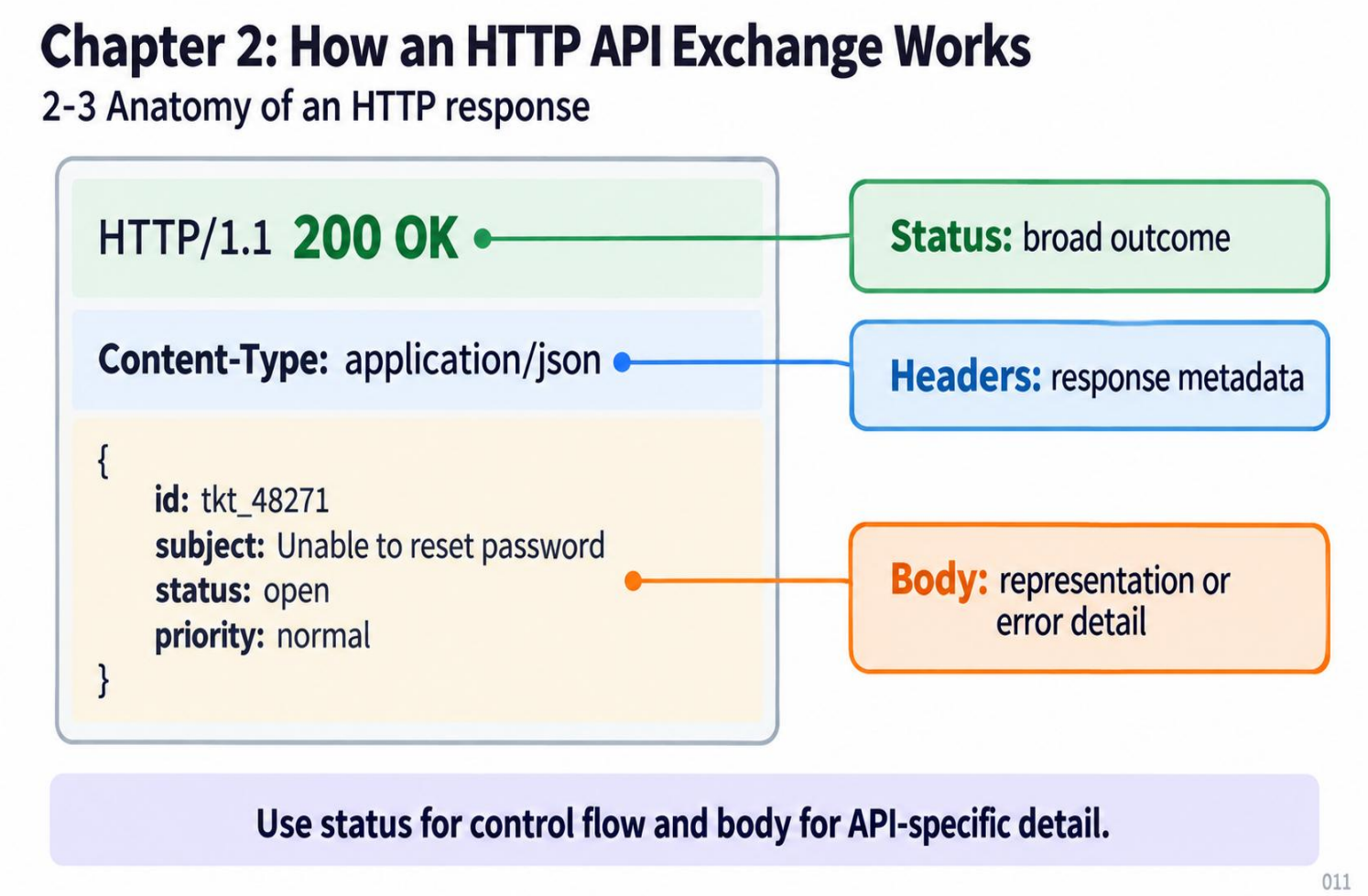



*Figure 2-4: Anatomy of an HTTP response*

`200 OK` indicates successful handling. `Content-Type` tells the client how to interpret the body. The JSON object is a **representation**, meaning the data form used in the message to describe the ticket.

The status code and body serve different purposes. The status code gives a general result that HTTP-aware software can understand. The body supplies API-specific details. A client should not have to parse a sentence in the body to discover whether the request succeeded.

## A first command-line request

The following command is one concrete example using `curl`, a command-line HTTP client, in a POSIX-style shell.

**Prerequisites:** `curl` is installed, the example API is reachable, and the caller has a valid access token stored in the `ACCESS_TOKEN` environment variable.

**Purpose:** Retrieve ticket `tkt_48271` and include response headers in the output.

```
curl --include \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --header "Accept: application/json" \
  "https://api.example.com/tickets/tkt_48271"
```

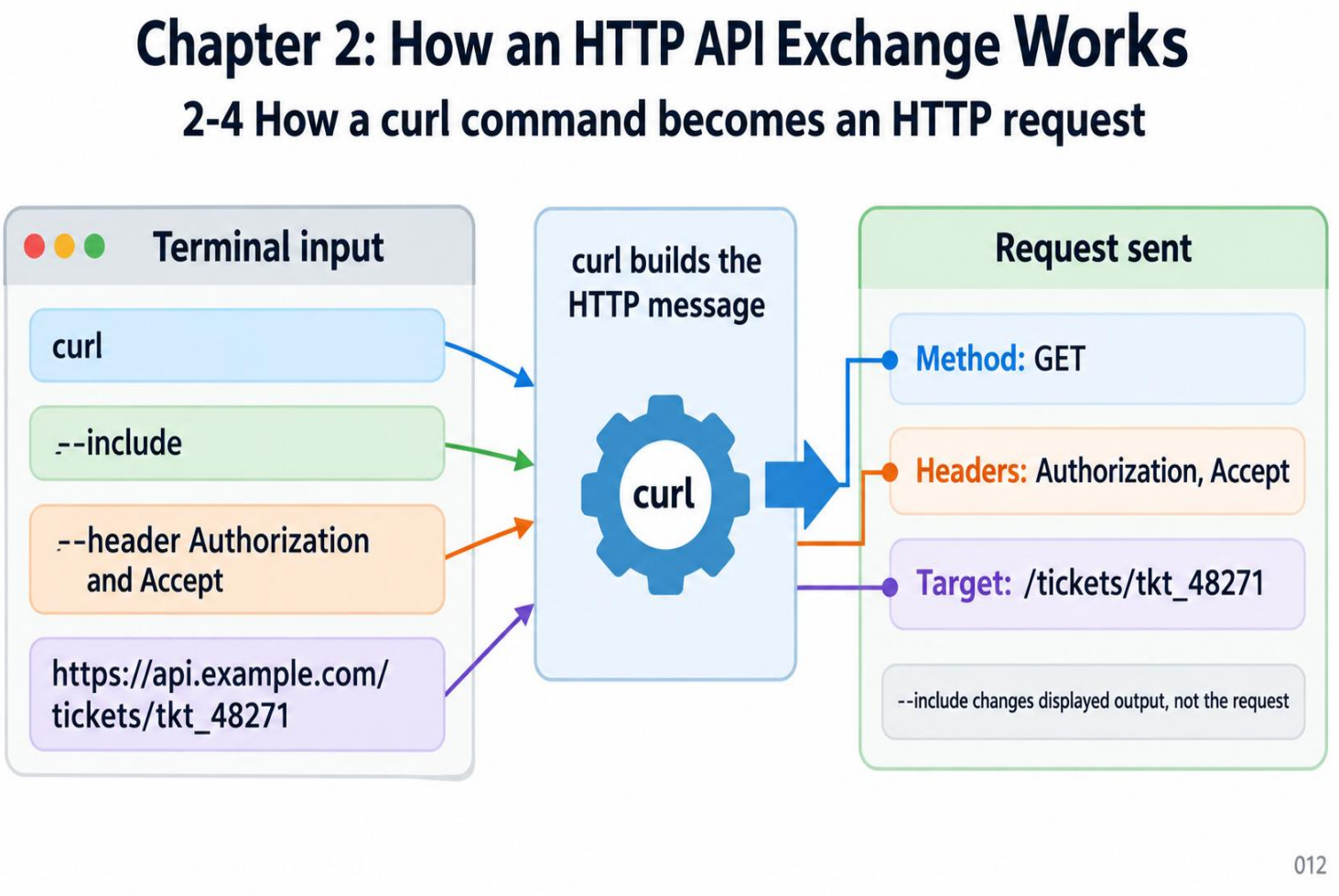



*Figure 2-5: How a `curl` command becomes an HTTP request*

The significant options are:

- `--include` prints response headers as well as the body.
- Each `--header` option adds request metadata.
- The final argument is the target URI.

**Expected result:** A successful call starts with a success status such as `200 OK`, followed by headers and a JSON ticket representation. A real service may return a defined error instead when the token is invalid, the caller lacks permission, or the ticket does not exist.

This is an environment-specific example. The exact host, credential method, and output format depend on the API being called.

## Method, URI, headers, and body work together

A frequent beginner error is to treat the path as the whole operation. These requests have the same path but different meanings:

```
GET /tickets/tkt_48271
DELETE /tickets/tkt_48271
```

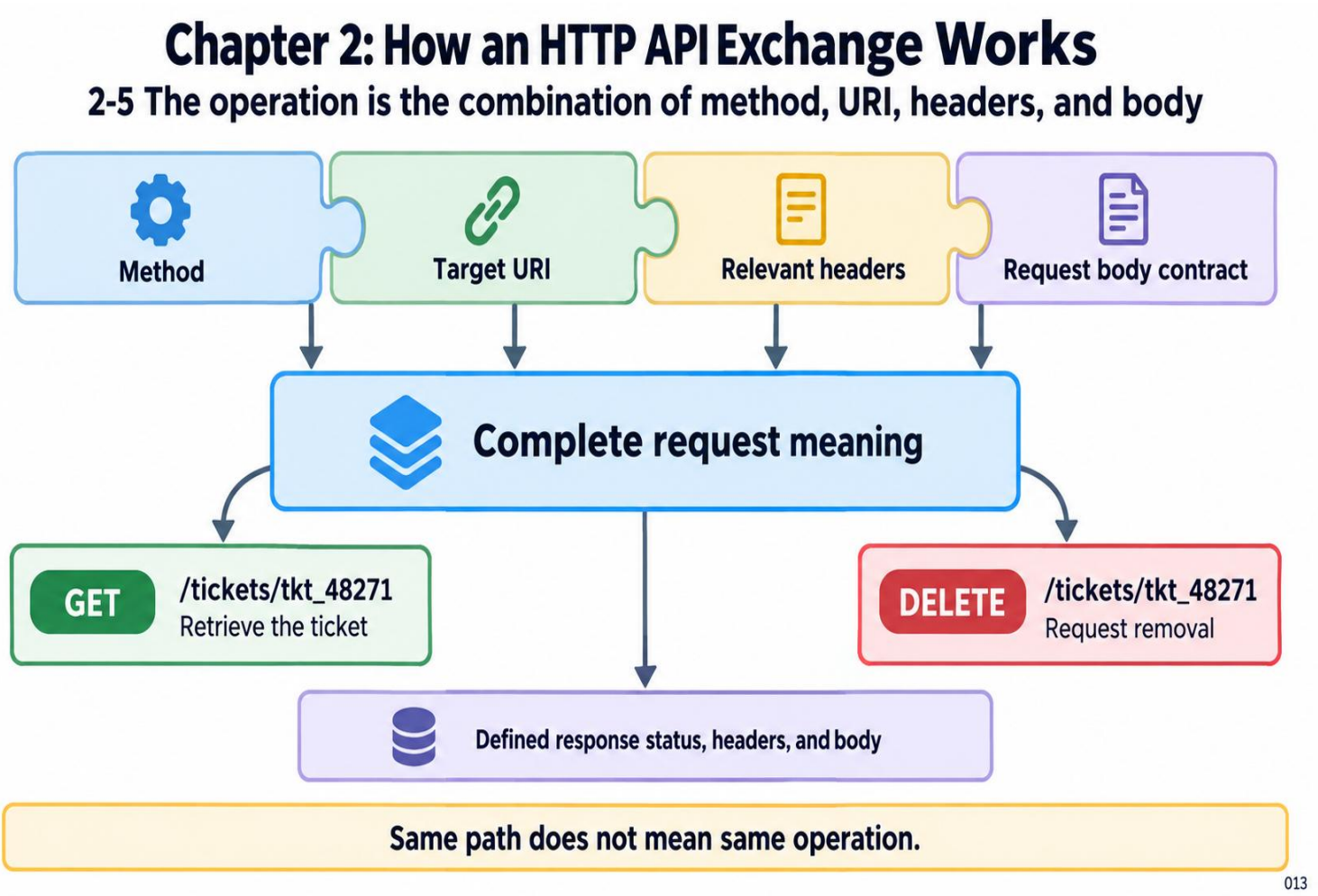



*Figure 2-6: The operation is the combination of method, URI, headers, and body*

The method changes the intended operation. Headers can also change handling. For example, a **conditional request**, which applies an operation only when a stated condition is true, may tell the server to update a ticket only if it has not changed since the client last read it. The body supplies operation-specific data, such as the new priority.

Therefore, an HTTP operation is identified and understood through the combination of:

- Method.
- Target URI.
- Relevant request headers.
- Request-body contract.
- Defined response status, headers, and body.

## Headers carry message metadata

A **header** is a named metadata field attached to an HTTP request or response. Common examples include:

- `Authorization`, which carries credentials in many API designs.
- `Accept`, which states which response media types the client can process.
- `Content-Type`, which identifies the format of a body that is present.
- `Location`, which may identify a newly created resource.
- `ETag`, which can identify a particular version of a representation.
- `Cache-Control`, which provides instructions for **caching**, meaning reuse of a stored response under defined conditions.

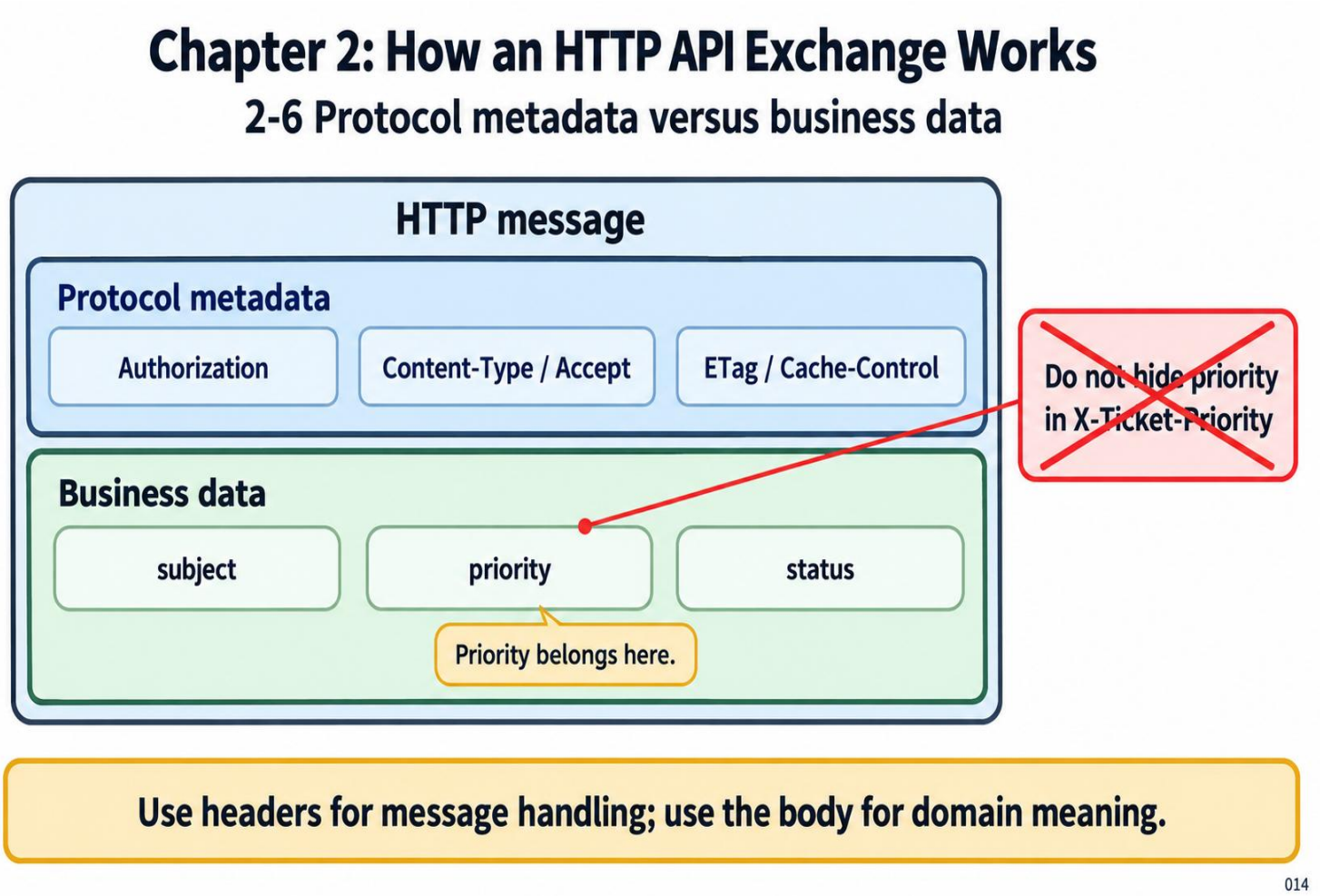



*Figure 2-7: Protocol metadata versus business data*

Headers should be used when the information describes message handling or protocol behavior. Business data usually belongs in the body or resource model. For example, ticket priority is business data; it should not be hidden in a custom protocol header.

## Stateless interactions

A **stateless interaction** is one in which each request carries the information needed to understand and authorize that request, rather than relying on undocumented conversational context from earlier calls.

Stateless does **not** mean that the server stores no data. The ticket service certainly stores tickets, users, and assignments. It may also maintain authorized sessions. Statelessness means the meaning of a call should not depend on a fragile hidden sequence such as:

1. Call `/select-ticket`.
2. The server silently remembers the selected ticket for this connection.
3. Call `/close` without identifying which ticket to close.

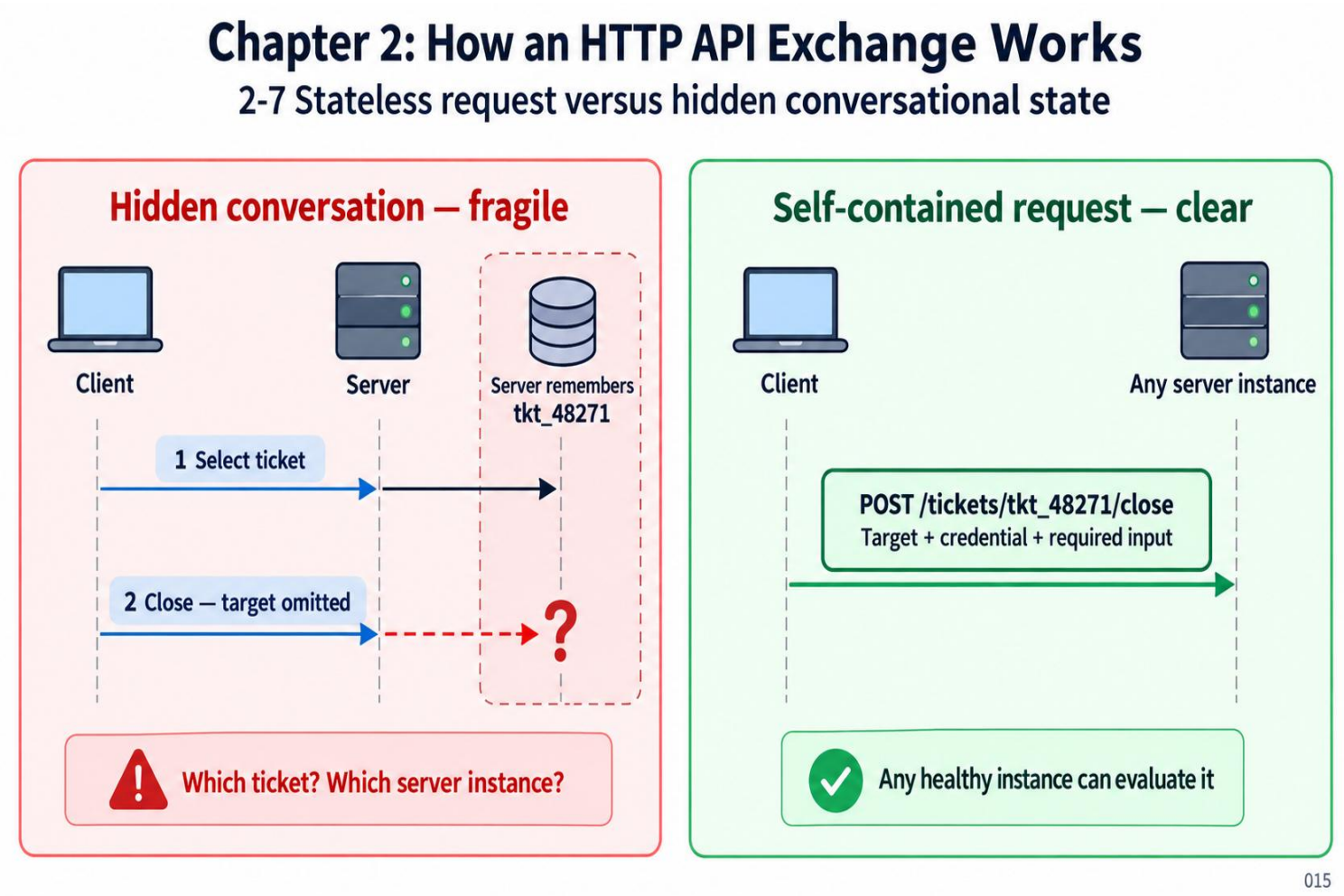



*Figure 2-8: Stateless request versus hidden conversational state*

A clearer request identifies its target directly:

```
POST /tickets/tkt_48271/close
```

An everyday analogy is submitting a complete form to a service desk. The clerk may have records in a system, but the form still identifies the requester and the case being changed. A note saying only “close the one we discussed before” depends on hidden context and is easy to misinterpret.

Stateless interactions support independent retries, operation across multiple interchangeable server instances, logging, and troubleshooting because each request can be evaluated on its own. They do not eliminate all server-side state.

## Intermediaries in the path

The client and application server are not always directly connected. Common intermediaries include:

- A **proxy**, which sends requests on behalf of another component.
- An **API gateway**, which provides shared functions such as routing, authentication integration, limits, and logging; **authentication** is the process of establishing the caller’s identity.
- A **load balancer**, which distributes traffic across multiple server instances.

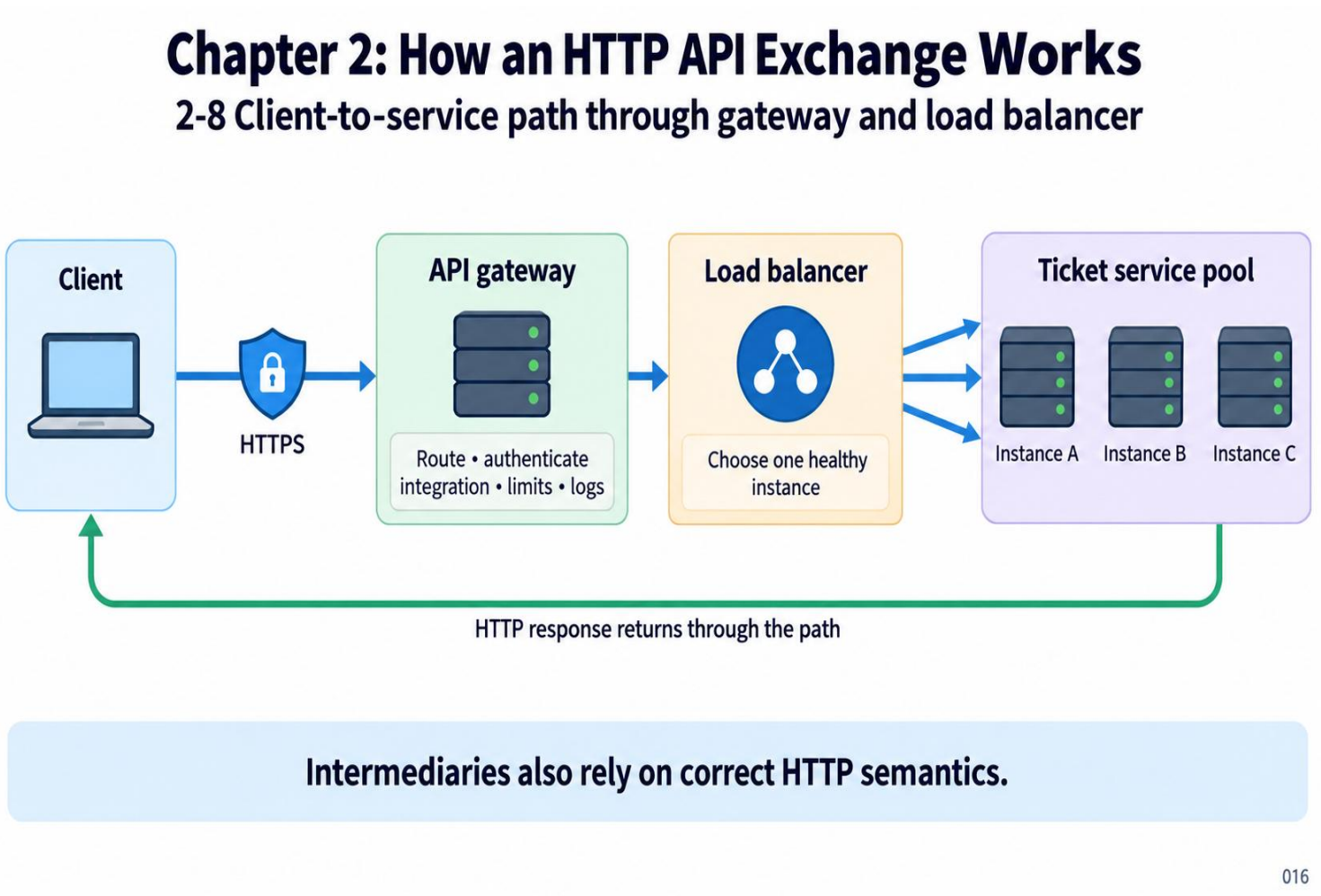



*Figure 2-9: Client-to-service path through gateway and load balancer*

These components matter because HTTP semantics are not used only by application code. An intermediary may cache a safe retrieval, retry a connection, reject an oversized request, or route by host and path. Using methods, status codes, and headers consistently gives these components enough information to behave correctly.

A design should not assume that every request travels over one permanent connection to one server instance. That assumption creates hidden state and unreliable behavior in a **distributed system**, where multiple components communicate over a network.

## Following one ticket retrieval

Consider `GET /tickets/tkt_48271` from end to end:

1. The web application constructs an HTTPS request with `GET`, the ticket URI, credentials, and an accepted response type.
2. A gateway receives the request and may validate basic credentials and route it to the ticket service.
3. A load balancer selects one healthy service instance.
4. The service checks whether the caller may view the ticket.
5. The service loads the current ticket state and builds the public representation.
6. The service returns a status, headers, and JSON body.
7. The client uses the status to select its control path and the body to render ticket data.

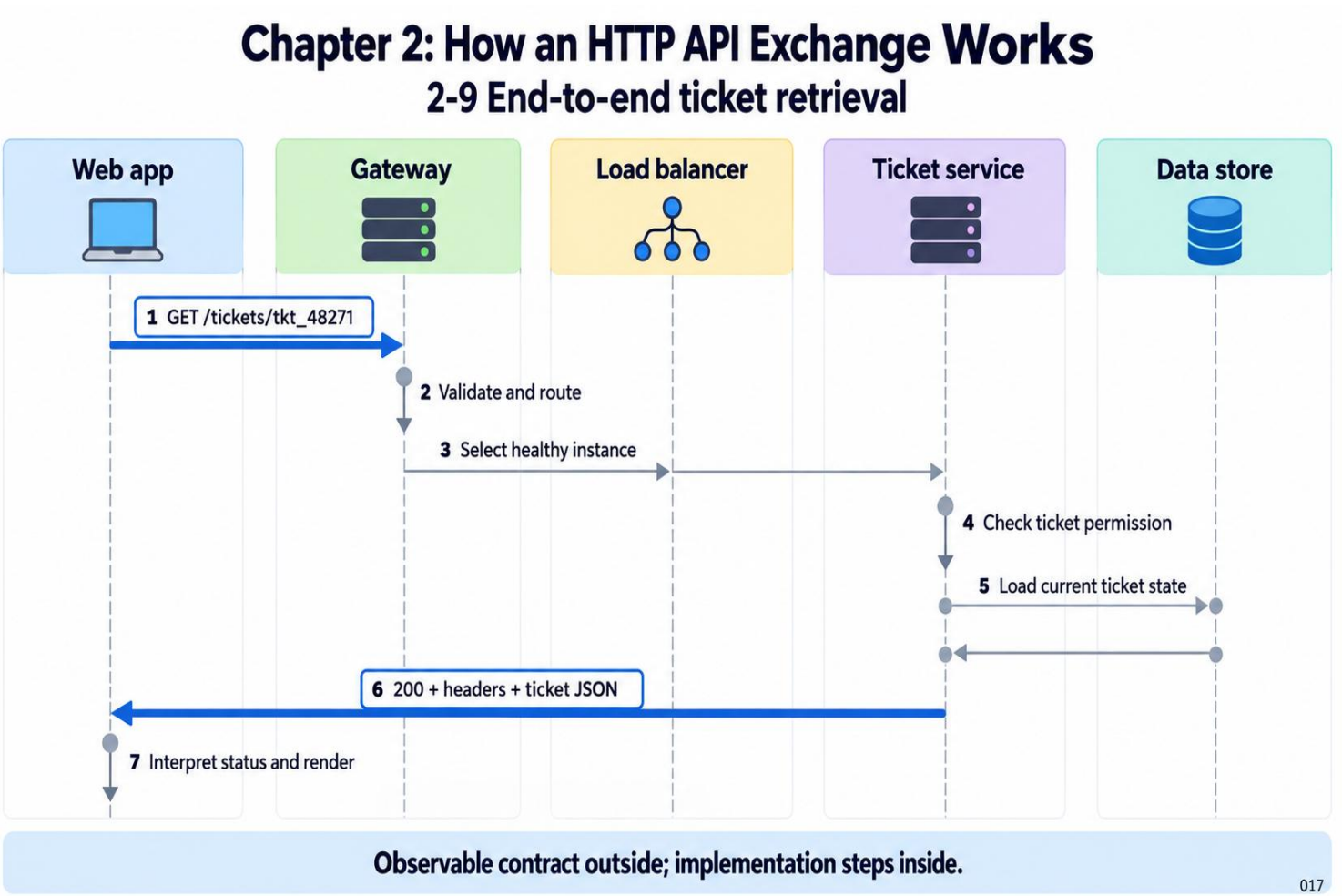



*Figure 2-10: End-to-end ticket retrieval*

The API contract concerns the observable request and response. The specific gateway, server instance, database query, and object mapping are implementation and operational details unless they change that observable behavior.

## Two common misunderstandings

### "The URL alone defines what an HTTP API call does"

The path identifies a target, but the method, headers, and body complete the request semantics. Documentation and tests must describe the whole operation, not only a list of paths.

### "Stateless means that the server cannot store any data"

A stateless API can store extensive persistent data. The requirement is that a request be understandable without an undocumented dependency on a previous conversational step or a particular server instance.

## A practical reading checklist

When examining any HTTP API call, identify:

- The method and its intended meaning.
- The target URI.
- Headers that affect interpretation or authorization.
- The request body's media type and schema, if a body exists.
- Every defined response status relevant to the operation.
- Response headers that clients must process.
- The success or error body contract.

**An HTTP exchange becomes predictable when every part of the request and response has an explicit role.**

## Chapter Summary

- An HTTP interaction is a request message followed by a response message.
- The operation is defined by the combination of method, URI, headers, and body rather than by the URL alone.
- Statelessness requires each request to carry the context needed to understand it; it does not prevent the server from storing durable data.

## Review Questions

1. Which four parts of a request determine what the client is asking the server to do?
2. What does HTTPS protect, and what does it not decide?
3. Why can an intermediary care about correct HTTP semantics even when it does not understand the business domain?

# Chapter 3 — Turning Use Cases into an API Contract

**Good API design starts with what consumers need to accomplish, not with database tables, controller methods, or a generic list of create-read-update-delete operations.** A use case provides the evidence needed to decide which resources, operations, inputs, outputs, rules, and failures belong in the contract.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Turn a concrete consumer goal into contract questions about inputs, outcomes, permissions, and failures.
- Use shared domain language and state transitions to describe business behavior before choosing endpoints.
- Define a coherent first contract and recognize when domain or security boundaries justify separate APIs.

## Start with consumers and outcomes

A **use case** is a concrete description of an actor trying to achieve an outcome under defined conditions. It is more useful than a broad statement such as "support ticket management" because it exposes the decisions the API must support.

For the ticket service, possible consumers and outcomes include:

- A requester creates a ticket and receives a trackable identifier.
- A support agent assigns an open ticket to themselves.
- An automation service adds diagnostic information to a ticket.
- A reporting service retrieves tickets changed during a time range.

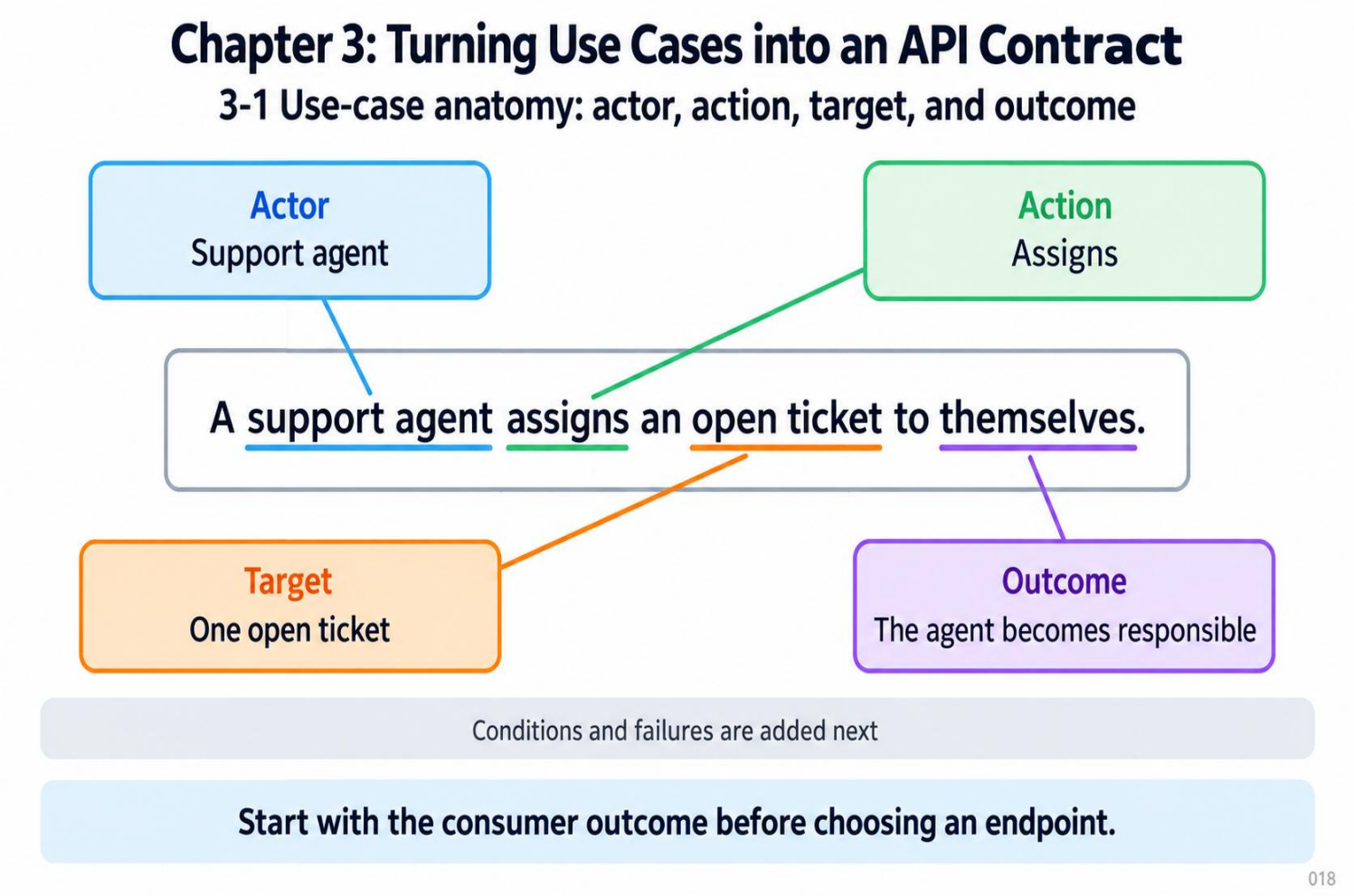



*Figure 3-1: Use-case anatomy: actor, action, target, and outcome*

Each sentence identifies an actor, an action, and a result. That is enough to begin asking precise questions.

Consider the use case:

> A support agent assigns an open ticket to themselves.

Before choosing a path or method, determine:

1. **Actor:** Which kind of caller performs the action?
2. **Target:** Which ticket is being assigned?
3. **Preconditions:** Must the ticket be open and currently unassigned?
4. **Input:** Does the caller send an assignee identifier, or does “to themselves” derive the identity from the authenticated caller?
5. **Authorization:** Which agents may assign which tickets?
6. **Business rule:** May a closed ticket be assigned?
7. **Concurrent behavior:** What happens if another agent assigns the ticket at the same time?
8. **Success result:** Does the response contain the updated ticket, no body, or a separate assignment record?
9. **Failures:** How are missing tickets, invalid states, and insufficient permission distinguished?

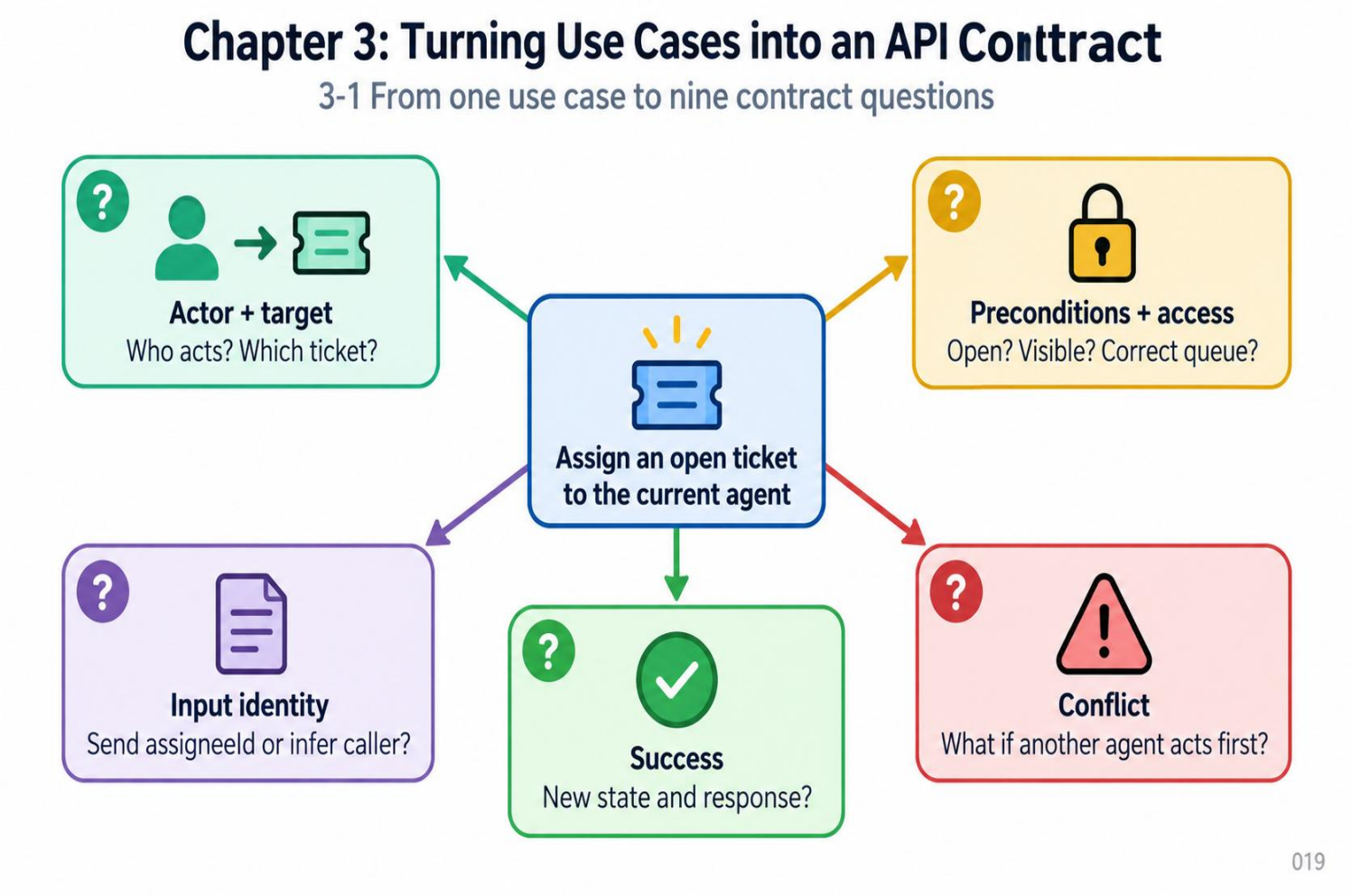



*Figure 3-2: From one use case to nine contract questions*

These questions define the contract more directly than beginning with a function named `updateTicket()`.

## Establish a shared domain language

A **domain** is the area of business knowledge and behavior that a system supports. **Domain language** is the agreed vocabulary used to describe concepts in that area.

For a support-ticket domain, important terms may include:

- **Ticket:** A recorded request for support that has identity, content, state, and history.
- **Requester:** The person or system asking for support.
- **Assignee:** The agent currently responsible for the ticket.
- **Comment:** A recorded message attached to a ticket.
- **Priority:** A business classification indicating relative urgency.

- **Status:** The ticket's current lifecycle state, such as `open`, `in_progress`, or `closed`.

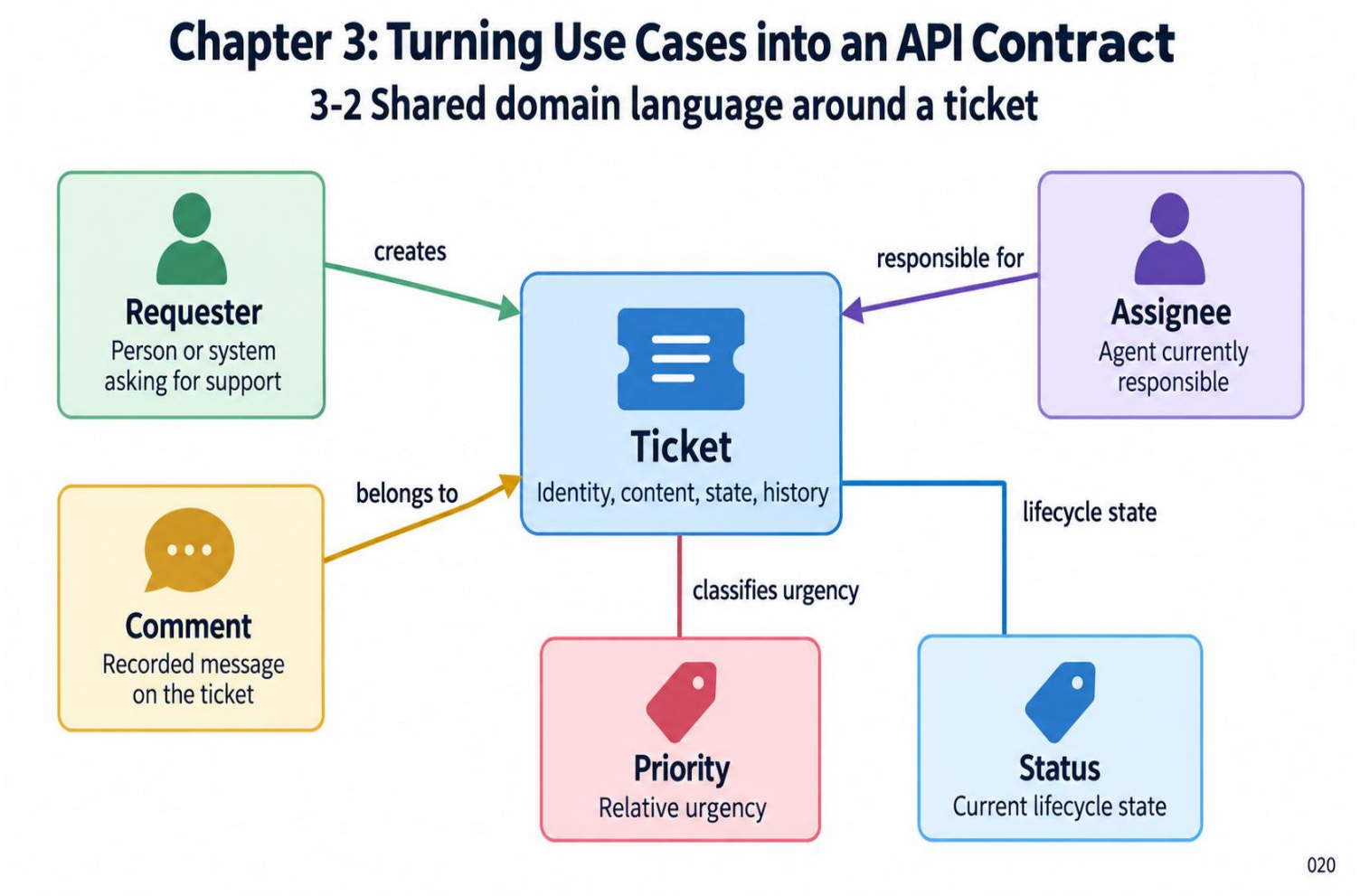



*Figure 3-3: Shared domain language around a ticket*

Definitions prevent teams from using the same word with different meanings. For example, one team may use "resolved" to mean that an agent believes the work is complete, while another uses it to mean that the requester confirmed completion. If the API exposes a `resolved` status without agreement, consumers will implement contradictory behavior.

The public contract should use language that consumers understand and that the provider can support consistently. It should not expose incidental terms from one codebase when those terms do not represent stable domain concepts.

## Describe behavior before endpoint shape

A useful use-case record is brief but specific. For the assignment example:

**Goal:** Assign an open ticket to the calling support agent.

**Preconditions:**

- The caller is authenticated as a support agent.
- The caller has access to the ticket's support queue.
- The ticket exists and is in an assignable state.

**Input:**

- Ticket identifier.
- The caller's current view of the ticket version when overwrite protection is required.

**Successful outcome:**

- The caller becomes the assignee.
- The assignment time and audit information are recorded.
- The response communicates the resulting ticket state.

**Important failures:**

- The ticket does not exist or is not visible to the caller.
- The caller lacks assignment permission.
- The ticket is closed.
- Another agent assigned or changed the ticket first.

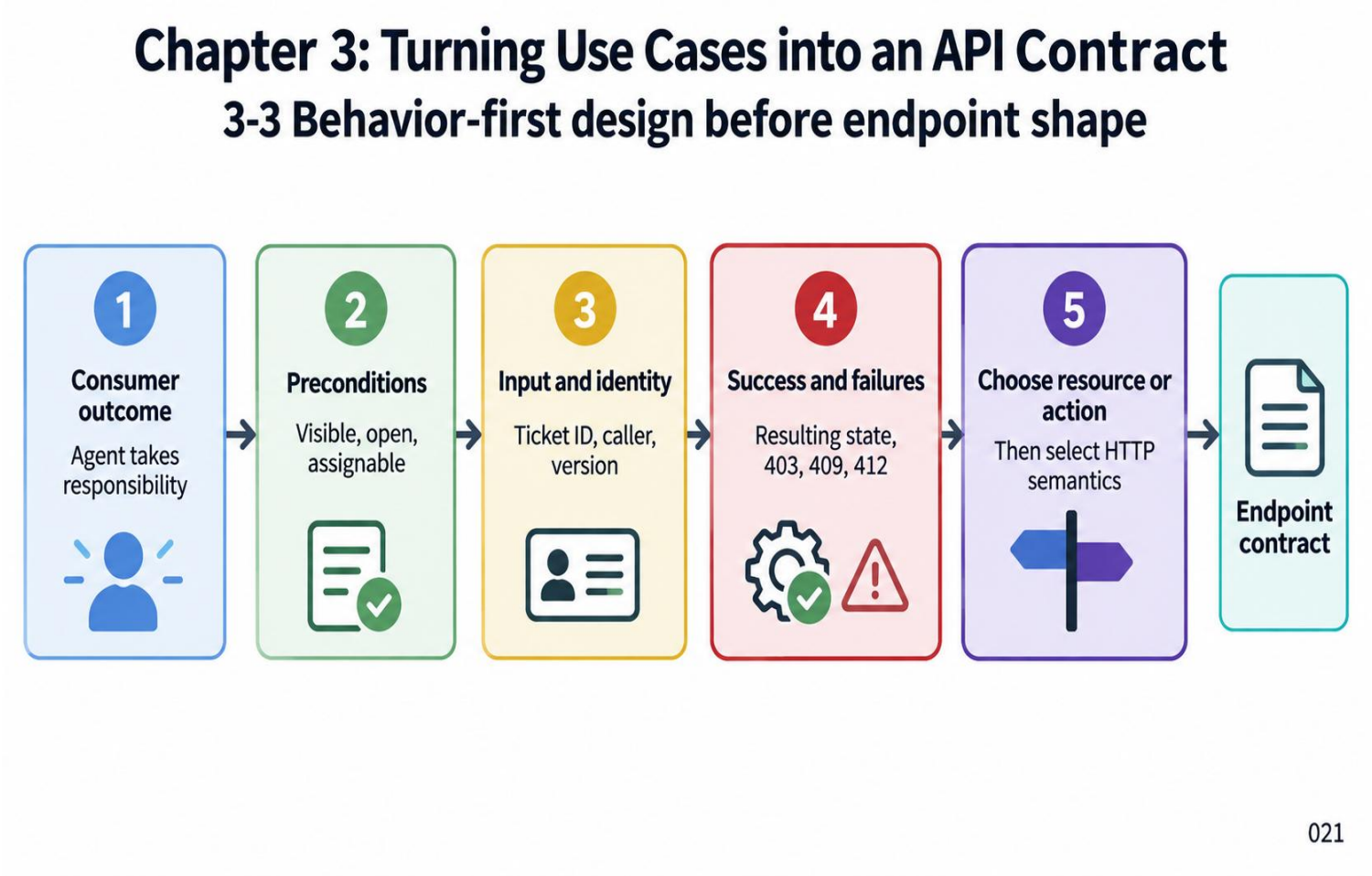



*Figure 3-4: Behavior-first design before endpoint shape*

This description does not yet decide whether assignment is represented as a partial update, an action, or a separate assignment resource. That decision comes after the behavior is understood.

## Separate business capability from implementation steps

A **business capability** is an outcome the system intentionally supports, such as assigning a ticket or closing it. An internal implementation may require several steps:

1. Read the ticket row.
2. Verify queue membership.
3. Update an assignment table.
4. Write an audit record.
5. Publish an internal notification.

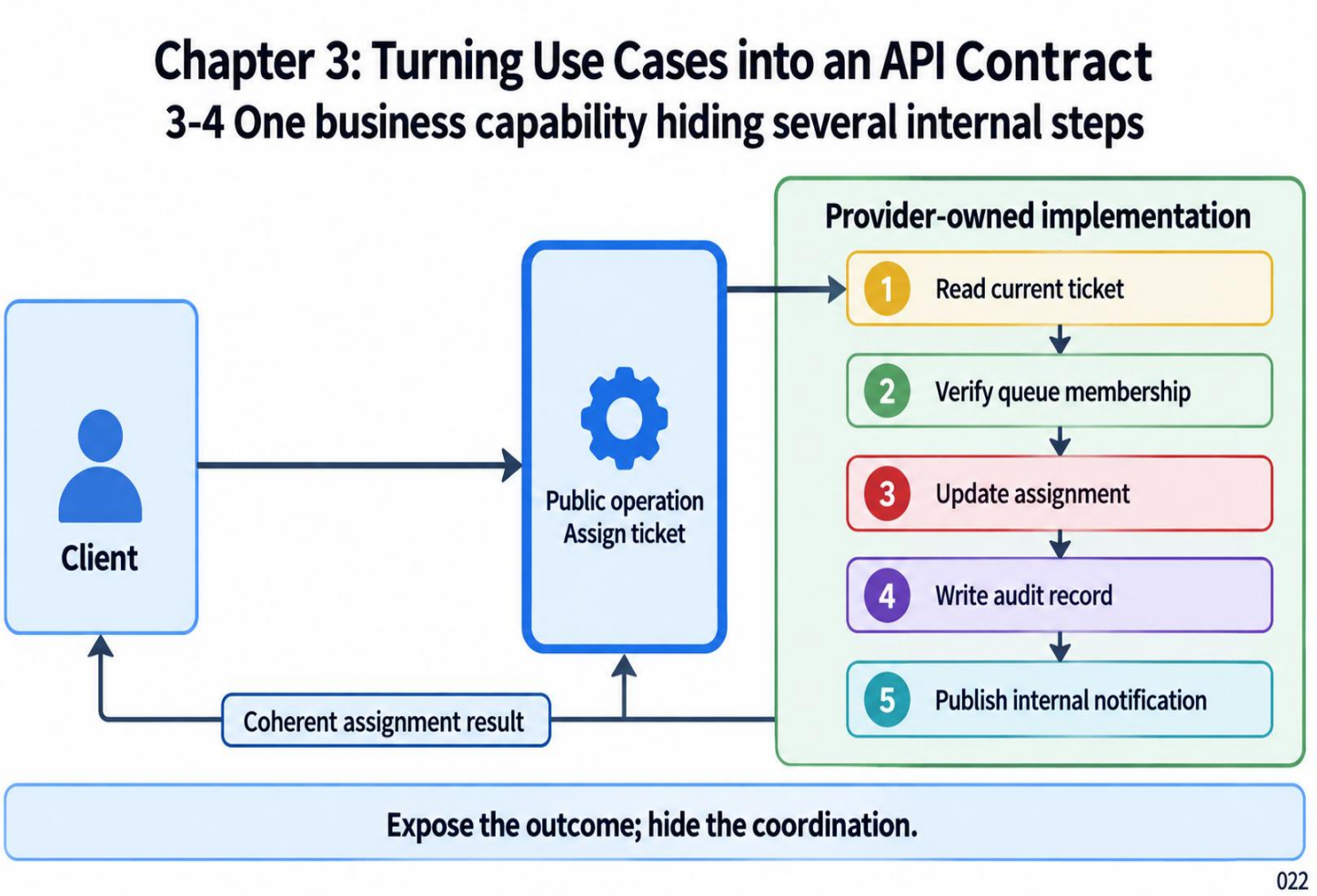



*Figure 3-5: One business capability hiding several internal steps*

Those steps do not automatically become five public API calls. Exposing them would make the client coordinate internal consistency and would couple it to the provider's design.

The public API should usually offer one coherent operation whose success means the business outcome has occurred. The provider remains responsible for its internal steps and transaction boundary.

A warning sign is an API that asks consumers to reproduce server logic:

```
1. GET /agent-queue-memberships
2. GET /ticket-status-rows
3. POST /ticket-assignment-rows
4. POST /audit-log-rows
```

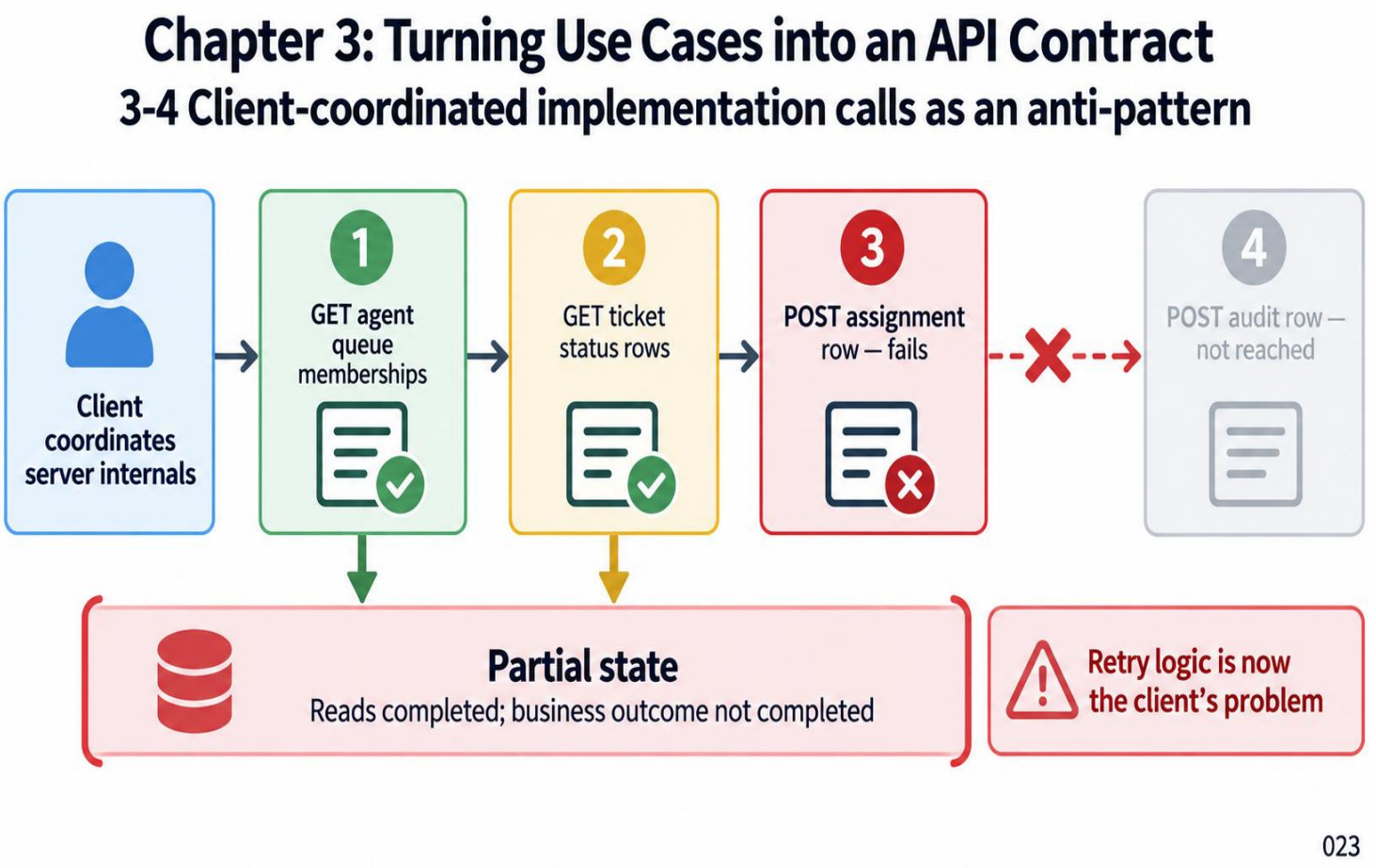



*Figure 3-6: Client-coordinated implementation calls as an anti-pattern*

Even if those paths match internal tables, they do not form a safe business contract. The consumer must know too much, and a failure between calls can leave incomplete work.

## Model workflows and state transitions

A **state transition** is a permitted change from one defined state to another. Ticket lifecycle behavior should be written explicitly rather than inferred from unrestricted field editing.

A simple lifecycle might allow:

- `open` to `in_progress` when work begins.
- `in_progress` to `resolved` when the agent believes the issue is solved.
- `resolved` to `closed` after confirmation or an elapsed period.
- `resolved` back to `in_progress` when the issue recurs.

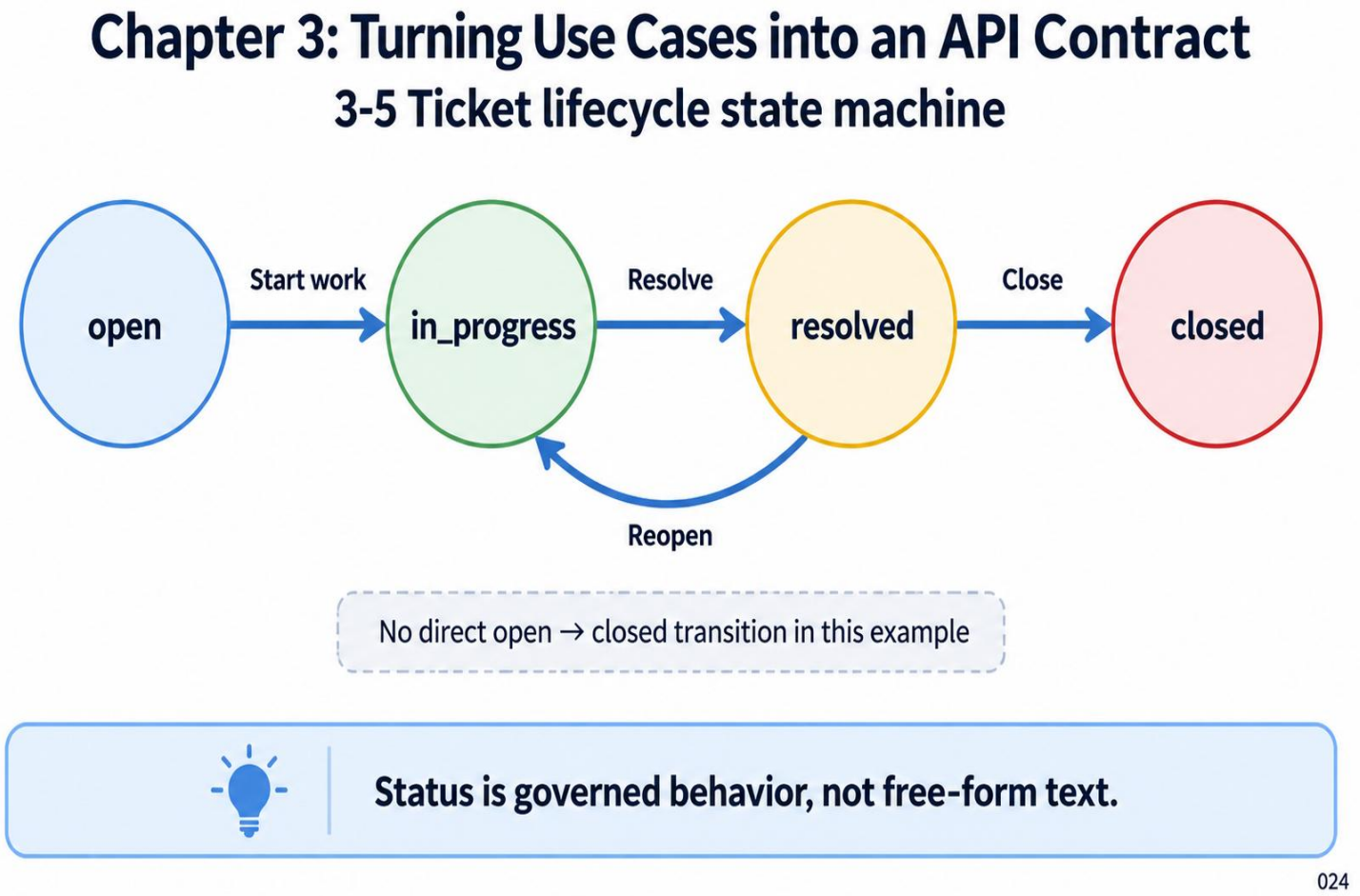



*Figure 3-7: Ticket lifecycle state machine*

The API design must also identify transitions that are not allowed. For example, a ticket might not move directly from `open` to `closed` because the organization requires a resolution record.

Think of a state machine as a railway map. Stations are states, and tracks are allowed transitions. A field that accepts any status value is like telling a train it may jump from any station to any other without following a track. It is easy to implement as a database update, but it ignores the workflow.

For each transition, identify:

- Who may request it.
- Which source states permit it.
- Which additional data is required.
- Which side effects are part of the outcome.
- What the consumer receives after success.
- How invalid or conflicting requests fail.

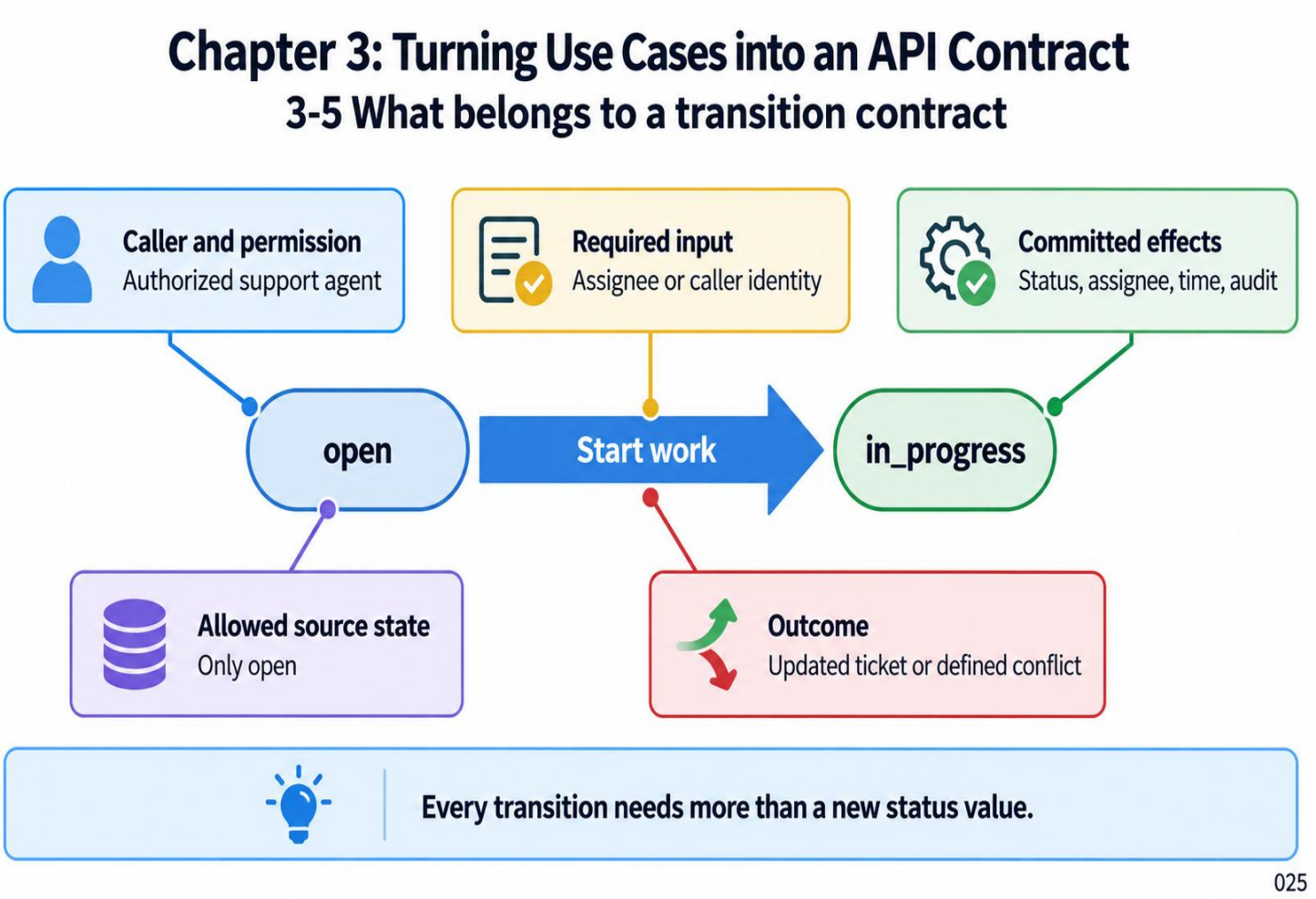



*Figure 3-8: What belongs to a transition contract*

## Capture validation and permission rules early

**Validation** is the process of checking whether input satisfies the contract and business rules. **Authorization** is the decision about whether a known caller may perform a specific action on a specific target.

For ticket assignment, possible validation and authorization rules include:

- The ticket identifier has the expected form.
- The ticket exists.
- The caller is a support agent.
- The caller belongs to the queue that owns the ticket.
- The ticket's current status allows assignment.
- The ticket has not changed since the caller viewed it.

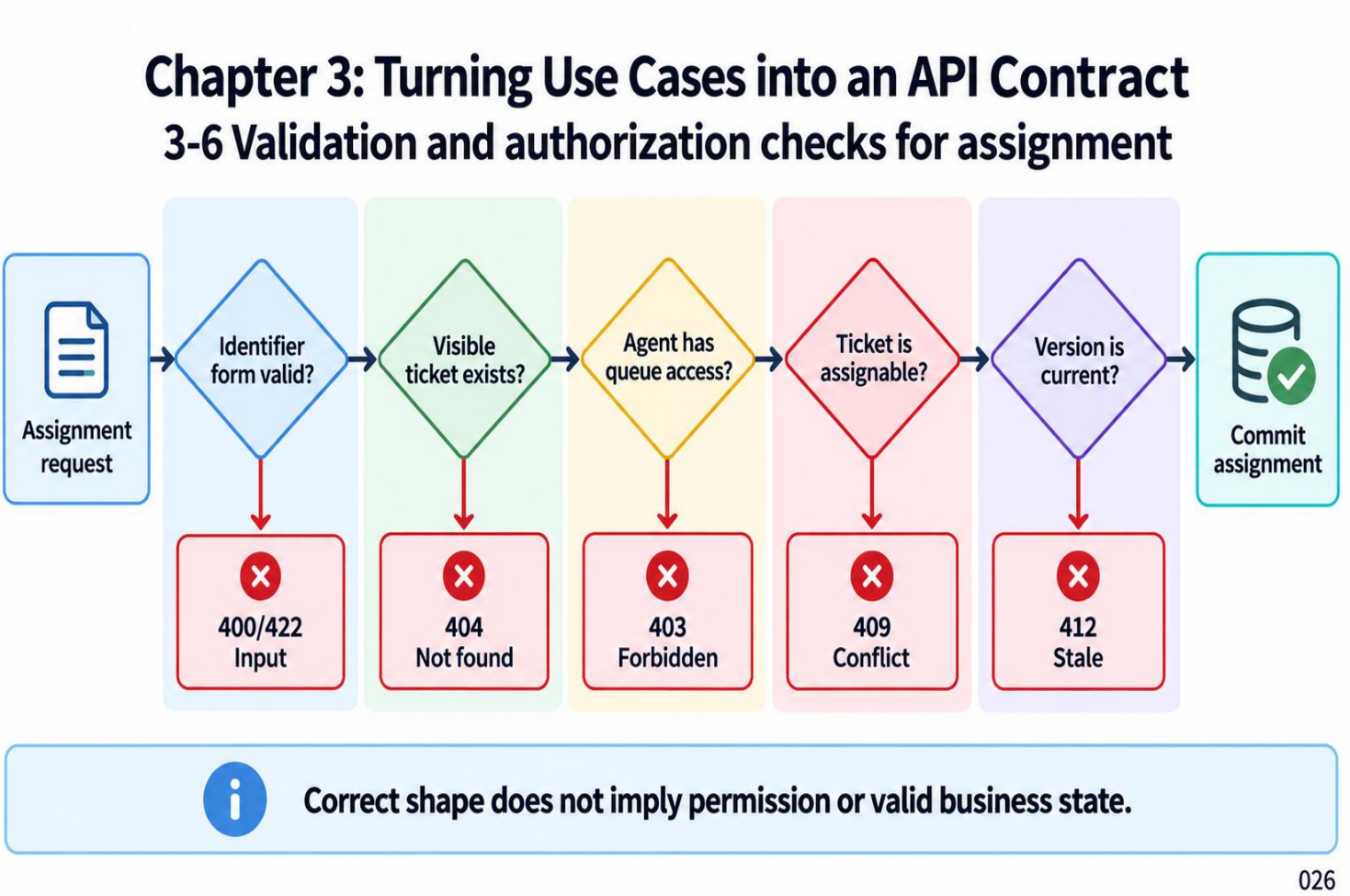



*Figure 3-9: Validation and authorization checks for assignment*

These checks are related but not interchangeable. A syntactically correct ticket identifier does not prove that the ticket exists. An existing ticket does not prove that the caller may see it. A caller with general agent permissions may still lack access to a particular queue.

Recording the rules during design prevents an implementation from returning one vague “bad request” response for every failure.

## Include nonfunctional requirements

A **nonfunctional requirement** describes an operating quality or constraint rather than a business capability. It can change the shape of the contract.

Relevant questions include:

- How many requests are expected per second?
- What response time is acceptable to the user interface?
- How much ticket data may be returned in one call?
- Is duplicate ticket creation financially or operationally harmful?
- Must changes be auditable?
- Does the data contain sensitive personal information?
- How long may consumers continue using an older contract?
- What availability is required during support hours?

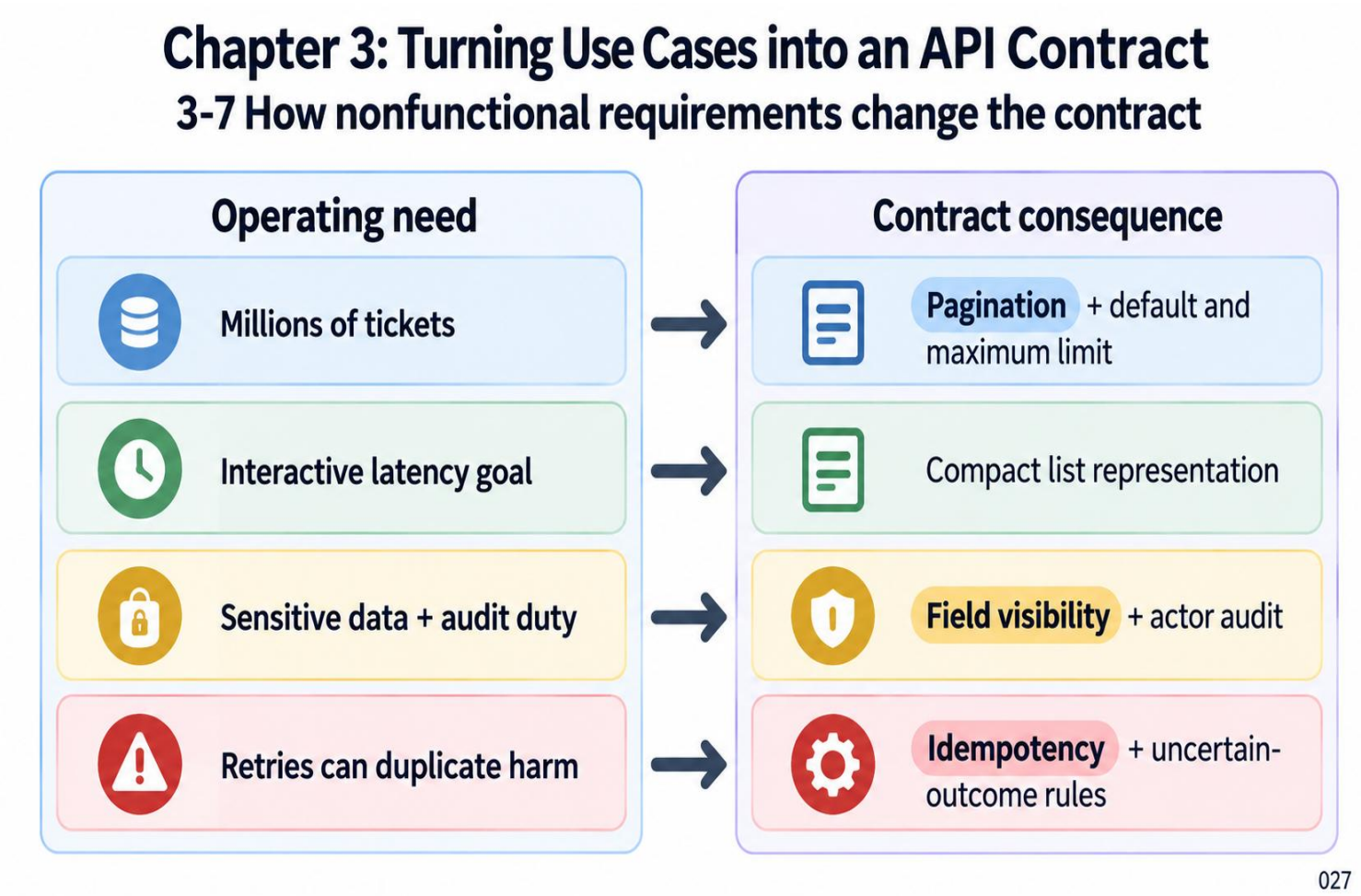



*Figure 3-10: How nonfunctional requirements change the contract*

For example, a reporting consumer that needs millions of tickets should not be given an unbounded `GET /tickets` response. A batch automation client operating over an unreliable network may require a way to retry ticket creation without creating duplicates. These are API-design consequences, not merely infrastructure details.

Do not attempt to predict every future requirement. Focus on constraints that are known, likely, and costly to ignore.

## Use examples to expose ambiguity

Example requests and responses are one of the fastest ways to test whether a proposed contract is understandable. An abstract statement such as “the assignment endpoint returns the ticket” may hide disagreements about which fields appear, whether the assignee can be absent, and how timestamps are represented.

A rough success example might be:

```
{
  "id": "tkt_48271",
  "status": "in_progress",
  "assignee": {
    "id": "usr_991",
    "displayName": "Morgan Lee"
  },
  "assignedAt": "2026-09-18T10:03:12Z"
}
```

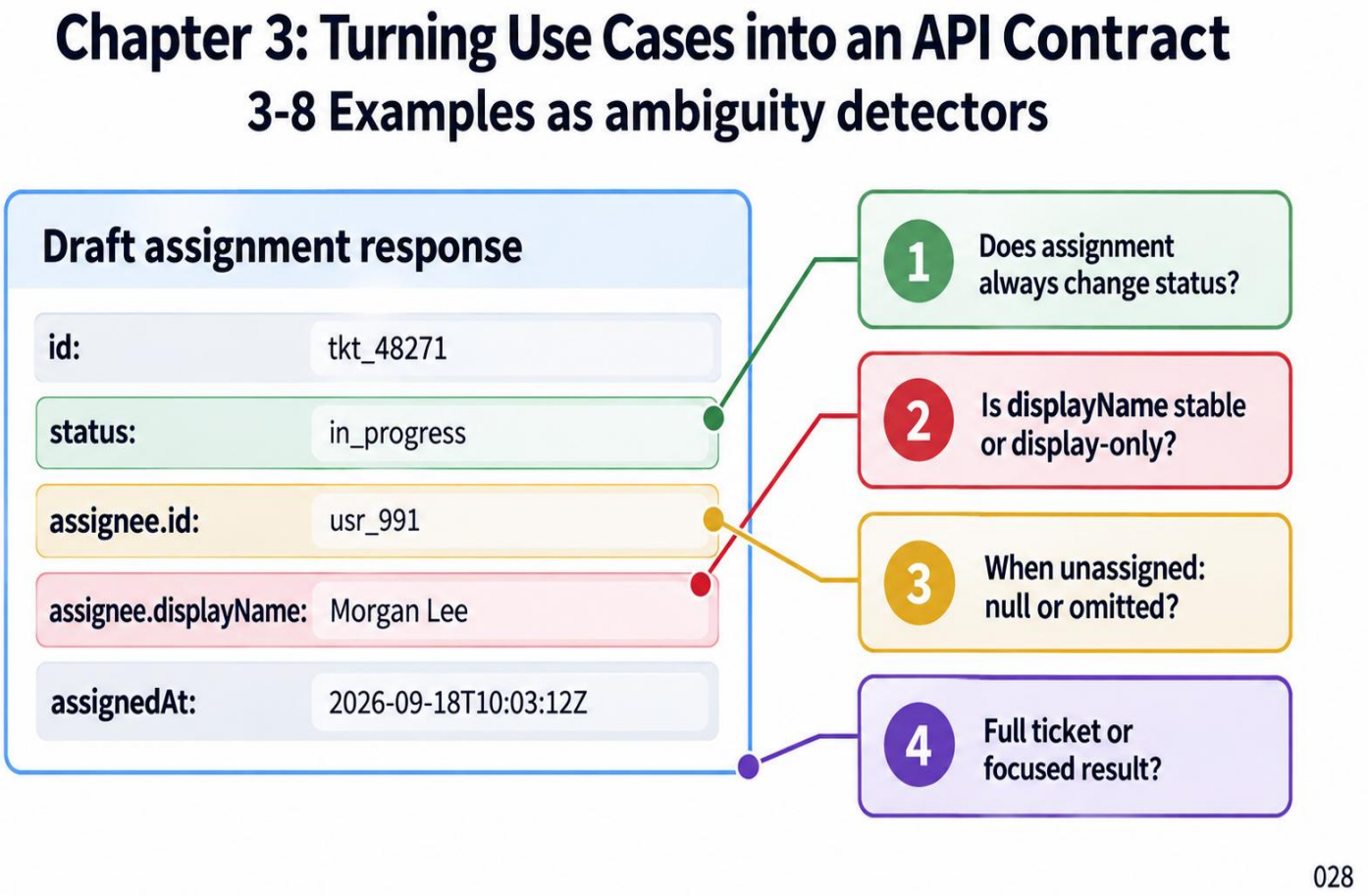



*Figure 3-11: Examples as ambiguity detectors*

Questions immediately become visible:

- Is assignment guaranteed to change the ticket to `in_progress`?
- Is `displayName` stable enough for automation, or is it display-only data?
- Is `assignedAt` absent or `null` when there is no assignee?
- Does the response contain the full ticket or an assignment-focused subset?

Examples are not substitutes for a formal schema, but they are effective design probes.

## Find the smallest coherent first contract

A **coherent contract** provides enough operations and definitions for a consumer to complete a real use case without depending on undocumented behavior. “Smallest” does not mean one generic endpoint that accepts arbitrary commands. It means avoiding speculative features while completing the chosen workflows.

For an initial ticket-assignment capability, the coherent contract may need:

- A way to identify or retrieve the ticket.
- A defined assignment operation.
- The current assignable states.
- The permission rule.
- The resulting representation.
- Errors for missing, forbidden, invalid-state, and concurrent-update cases.

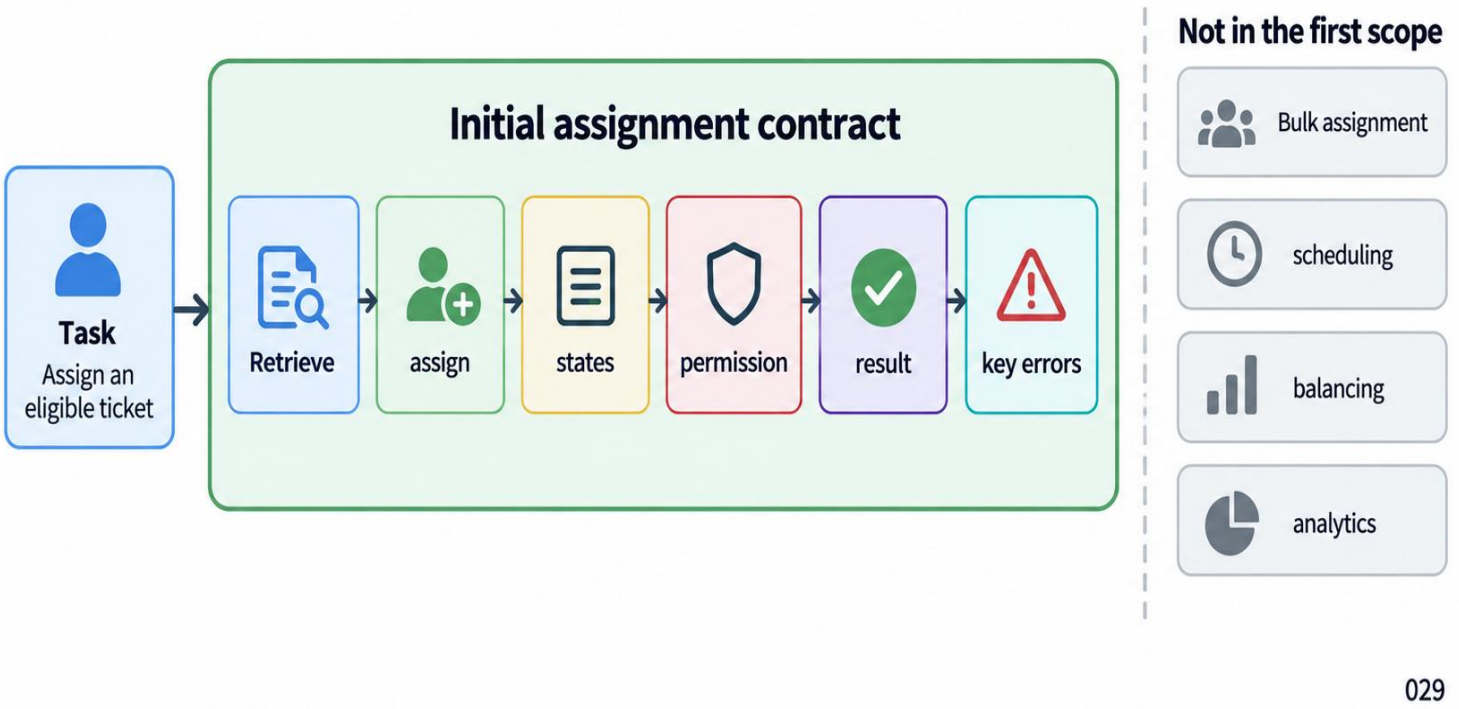



*Figure 3-12: Smallest coherent contract boundary*

It may not need bulk assignment, assignment scheduling, workload balancing, or historical analytics unless a current consumer requires them.

A practical test is: **Can the named consumer complete the named task using only documented behavior?** If not, the contract is incomplete. If the contract contains several operations with no named consumer or use case, it may be speculative.

## Decide where one API ends

A single API should not automatically contain every function related to a broad business area. A split may be appropriate when capabilities have different:

- Domain ownership.
- Security boundaries.
- Data sensitivity.
- Scaling patterns.
- Lifecycles.
- Consumer groups.

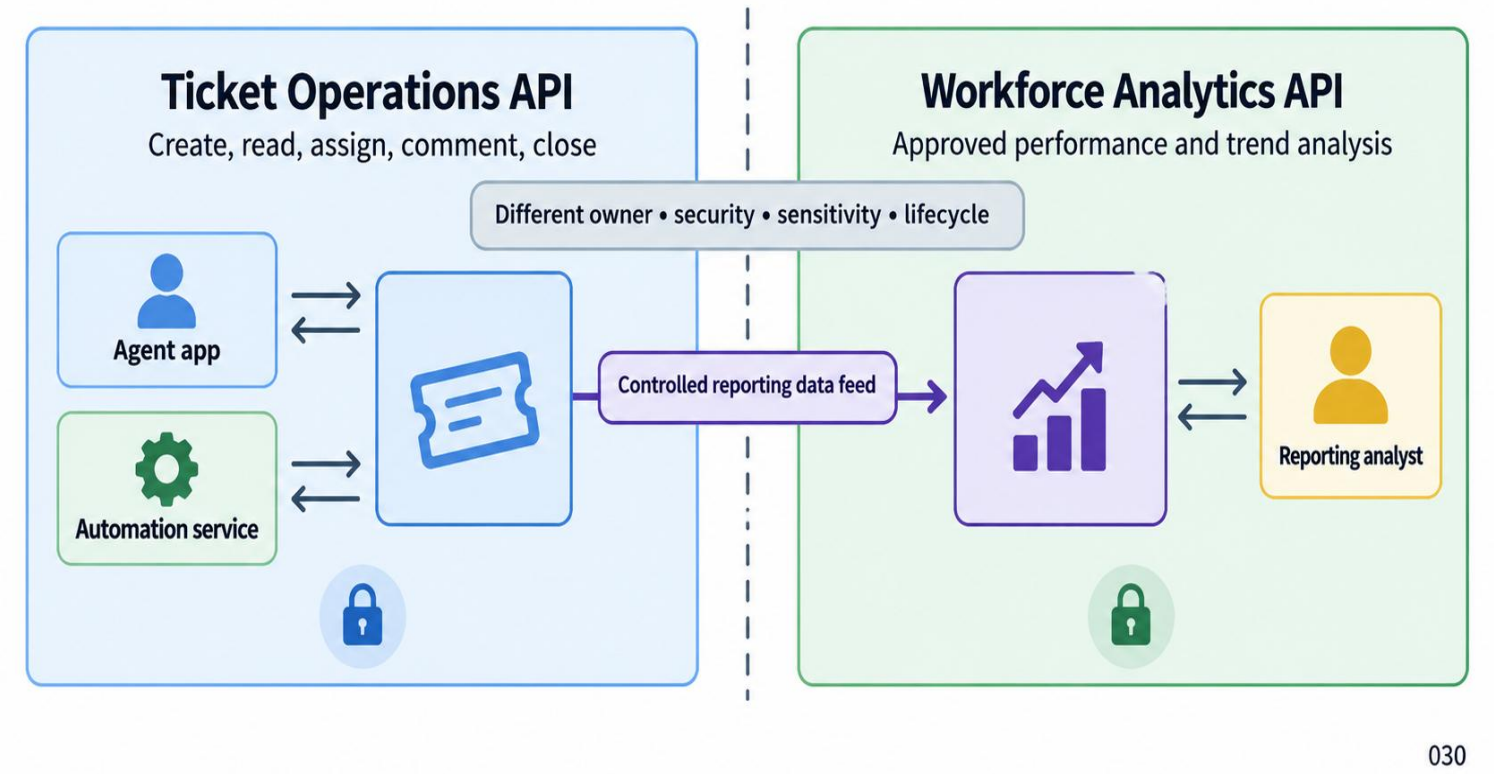



*Figure 3-13: Where one API ends*

For example, ordinary ticket operations and employee-performance analytics may both use ticket data, but they may belong to different services and access policies. The split should follow meaningful boundaries, not an arbitrary desire to create more services.

Conversely, splitting one business operation across several narrowly technical APIs can force consumers to coordinate internal work. The useful question is not “How small can each service be?” but “Where can a stable, independently owned contract be drawn?”

## Why generic CRUD is not enough

**CRUD** is shorthand for create, read, update, and delete. These operations are useful, but a generic CRUD surface does not automatically represent the real domain.

Suppose the API offers:

```
PATCH /tickets/tkt_48271
```

with any stored field accepted. The caller could attempt to set:

```
{
  "status": "closed",
  "assigneeId": "usr_991",
  "createdAt": "2025-01-01T00:00:00Z"
}
```

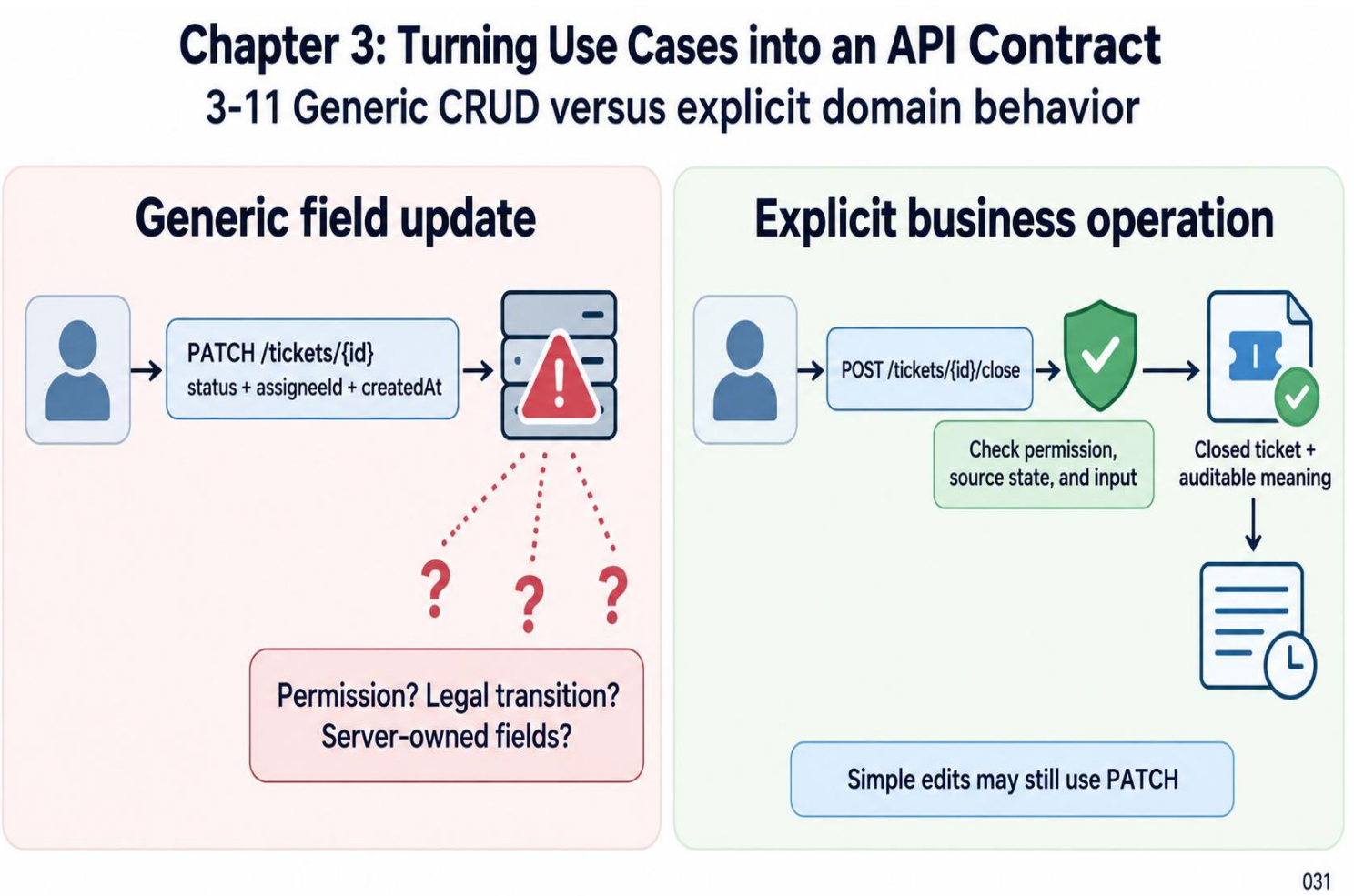



*Figure 3-14: Generic CRUD versus explicit domain behavior*

The server still needs to decide whether closing is allowed, whether the caller can choose that assignee, and whether `createdAt` is writable. The business behavior did not disappear; it was merely hidden behind a generic update.

CRUD can support straightforward resource management. Explicit operations are often clearer when the domain action has important rules, side effects, or audit meaning.

## Two mistakes to prevent

### "The database schema is the API model"

A database schema is optimized for storage, integrity, and queries inside the provider. An API model is optimized for stable communication with consumers. They may share concepts, but direct exposure creates accidental coupling to table names, keys, joins, normalization choices, and migration details.

### "A generic CRUD interface automatically covers the real workflow"

Create, read, update, and delete do not explain who may act, which state changes are legal, which fields are controlled by the server, or which side effects form one business outcome. The design must make those rules visible.

## A use-case design worksheet

For each important consumer task, record:

- Consumer and actor.
- Intended outcome.
- Target resource or business capability.
- Preconditions and allowed source states.
- Required and optional input.

- Permission decision.
- Successful result and resulting state.
- Important failures and recovery behavior.
- Expected request volume, latency, and data sensitivity.
- Duplicate, retry, and concurrent-update concerns.

**Only after these answers are clear should the team settle the endpoint shape.**

## Chapter Summary

- Use cases provide the evidence needed to design operations and data deliberately.
- Business capabilities should hide internal implementation steps instead of forcing clients to coordinate them.
- Examples, nonfunctional requirements, and explicit scope reveal ambiguity before implementation makes it expensive.

## Review Questions

1. What information should be known before selecting an endpoint shape for ticket assignment?
2. Why is a generic CRUD interface often insufficient for a real business workflow?
3. Which factors suggest that one API should be divided into separate ownership or security boundaries?

# Part II — Designing Everyday HTTP APIs

# Chapter 4 — Modeling Resources and Naming Endpoints

**A resource-oriented HTTP API is easiest to learn and use when stable domain concepts have clear identities, predictable collection paths, and relationships that do not expose internal storage structure.** Resource modeling is not an exercise in converting every noun or database table into a URL. It is the work of selecting the concepts consumers should address through the contract.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Model stable domain concepts as resources, collections, individual members, and relationships.
- Choose identifiers and URI paths that remain meaningful when implementation details change.
- Decide when a subordinate resource, reference, embedding, or explicit action best represents the domain.

## What counts as a resource

A **resource** is an API-addressable concept with identity and state. A support ticket is a resource because consumers need to identify it, retrieve its current representation, and request changes to it.

**Representational State Transfer (REST)** is an architectural style for networked systems; in this guide, *REST-style* means applying resource orientation and HTTP semantics consistently rather than claiming exhaustive conformance to every REST constraint.

A resource does not have to correspond to one database row. A ticket representation might combine information from several tables or services. It may also omit stored information that consumers do not need.

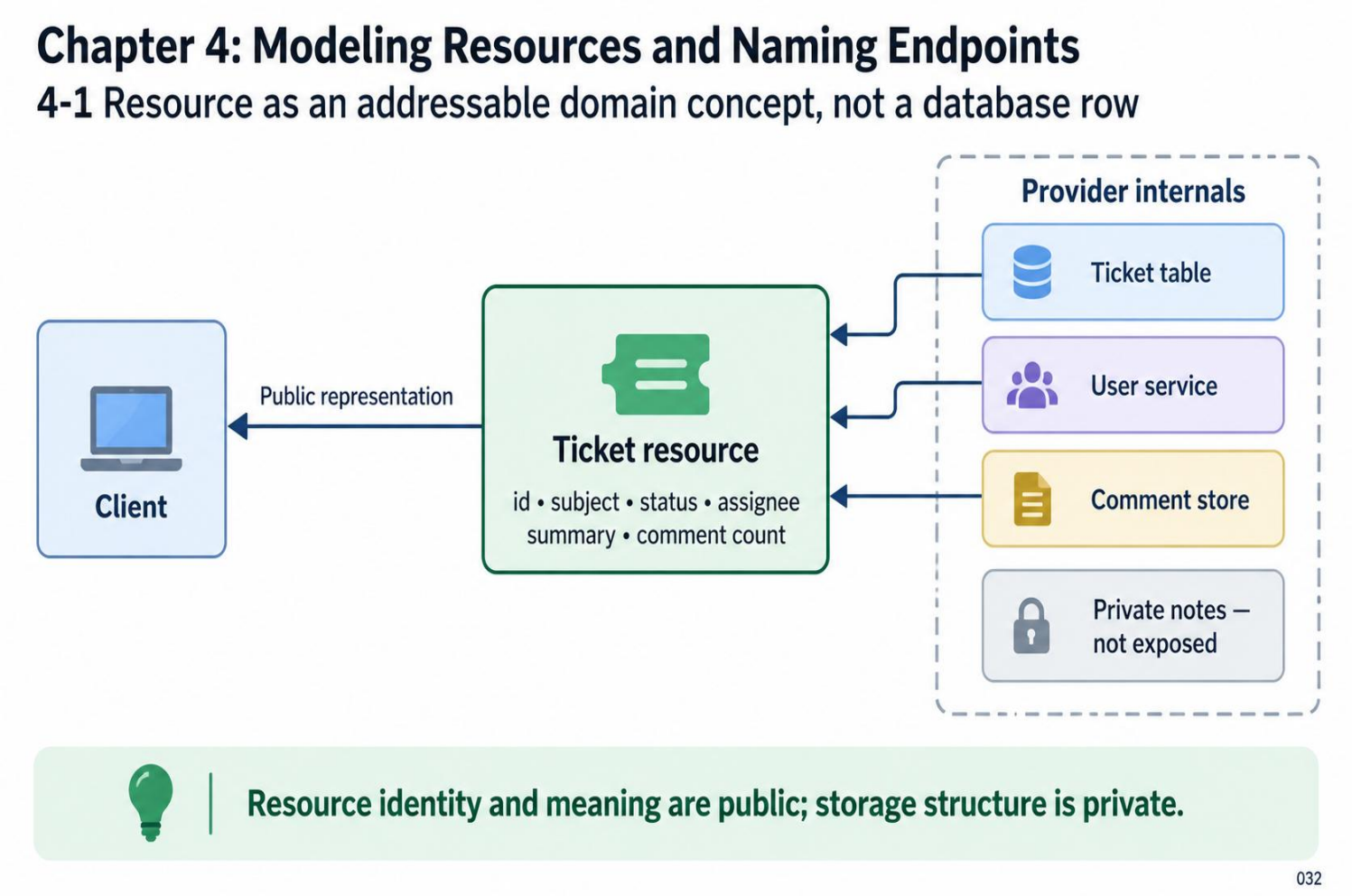



*Figure 4-1: Resource as an addressable domain concept, not a database row*

Resources commonly appear as:

- A **collection**, which represents a group such as `/tickets`.
- An **individual resource**, which represents one member such as `/tickets/tkt_48271`.

The collection path supports operations concerning the set, such as creating a ticket or listing tickets. The item path supports operations concerning one identified ticket.

A useful starting pattern is:

```
/tickets
/tickets/{ticketId}
```

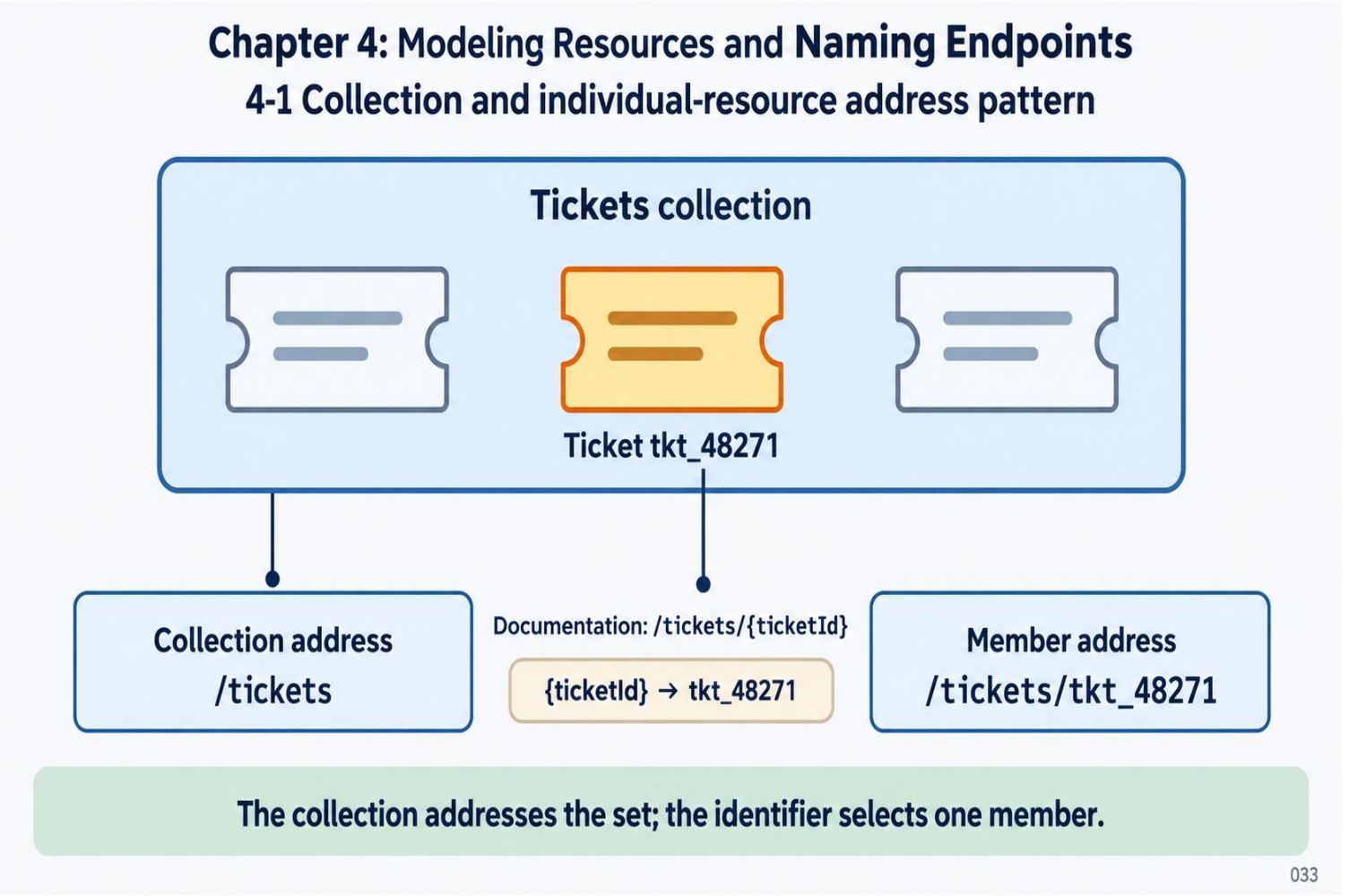



*Figure 4-2: Collection and individual-resource address pattern*

Curly braces in documentation indicate a path parameter. A real request replaces `{ticketId}` with a value such as `tkt_48271`.

## Give resources stable identities

A **resource identifier** is a value used to distinguish one resource from another. It should remain stable for the useful lifetime of the resource.

Good identifiers usually avoid mutable business labels. A subject such as “Unable to reset password” can change and is not unique. An email address may change and can expose personal data. A database sequence number may be technically stable, but exposing it can reveal internal assumptions and may create security or migration concerns.

The ticket example uses an opaque-looking string:

```
tkt_48271
```

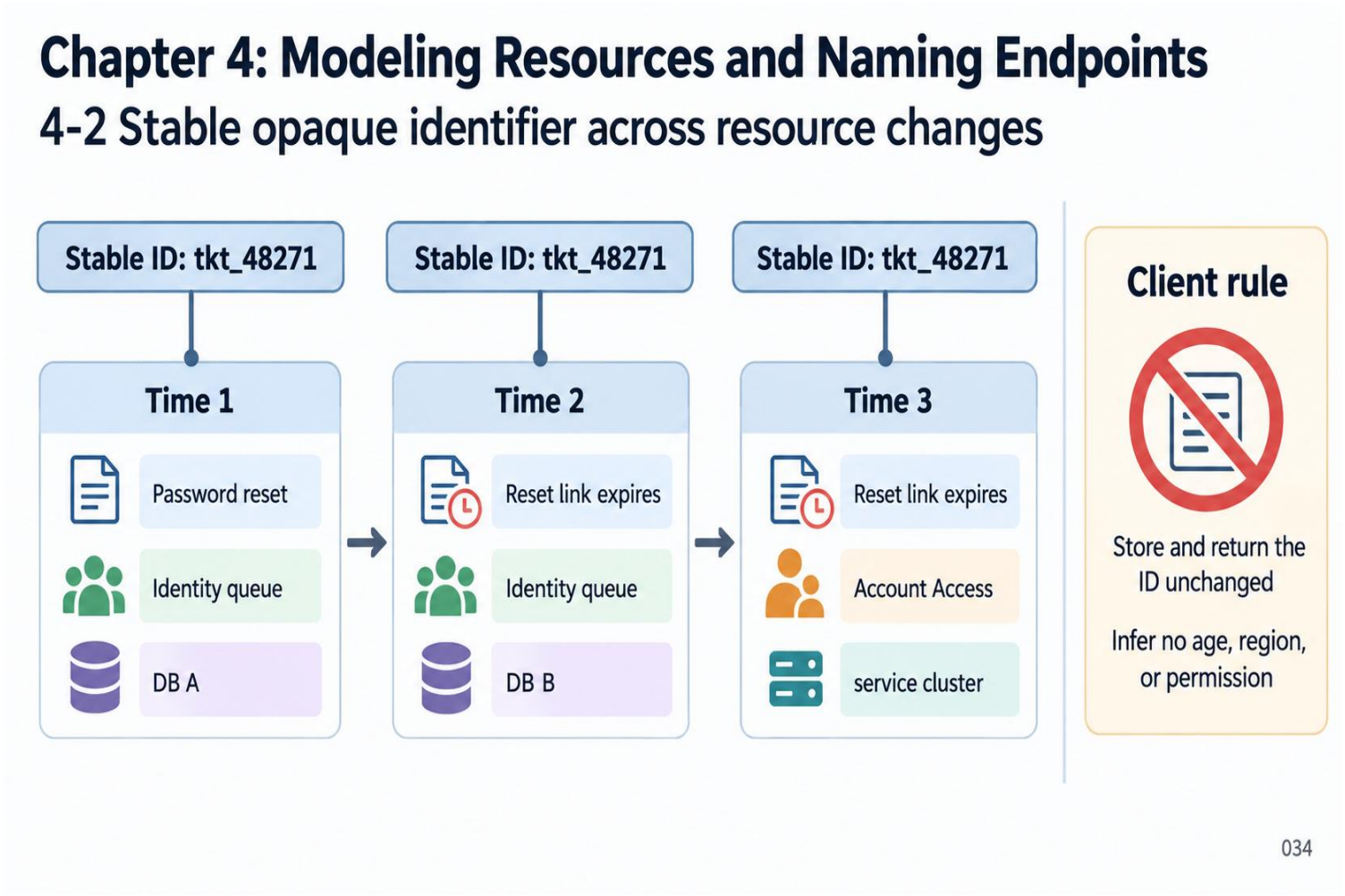



*Figure 4-3: Stable opaque identifier across resource changes*

**Opaque** means that consumers should treat the identifier as an indivisible value rather than interpret its internal structure. A prefix such as `tkt_` may help humans recognize the type, but consumers should not derive authorization, age, region, or routing behavior from it unless the contract explicitly defines that meaning.

The provider should document whether identifiers are:

- Case-sensitive.
- Globally unique or unique only within a parent or tenant.
- Reusable after deletion.
- Generated by the server or supplied by a client.

For most business resources, non-reuse and stable string treatment reduce surprises.

## Use nouns and consistent path conventions

Resource paths normally use nouns because the HTTP method communicates the broad operation. For example:

```
GET /tickets/tkt_48271
POST /tickets
DELETE /tickets/tkt_48271
```

Paths such as `/getTicket` or `/createNewTicket` repeat operation names that methods already express and tend to produce inconsistent naming as the API grows.

Common path conventions include:

- Use one grammatical number consistently; plural collection names such as `/tickets` are common.
- Use one word-separation style consistently, such as lowercase hyphenated segments when multiple words are necessary.
- Keep identifiers in predictable positions.

- Avoid exposing file extensions such as `.json` when content type is already expressed through HTTP.
- Avoid implementation terms such as `/ticketTableRows` or `/ticketController`.

These are conventions, not protocol laws. Consistency within the API and its surrounding portfolio is more valuable than debating small stylistic differences operation by operation.

## Model collections and members predictably

A coherent ticket API may begin with:

```
/tickets
/tickets/{ticketId}
```

Comments are concepts with their own content, author, and creation time. They can be modeled as a subordinate collection:

```
/tickets/{ticketId}/comments
/tickets/{ticketId}/comments/{commentId}
```

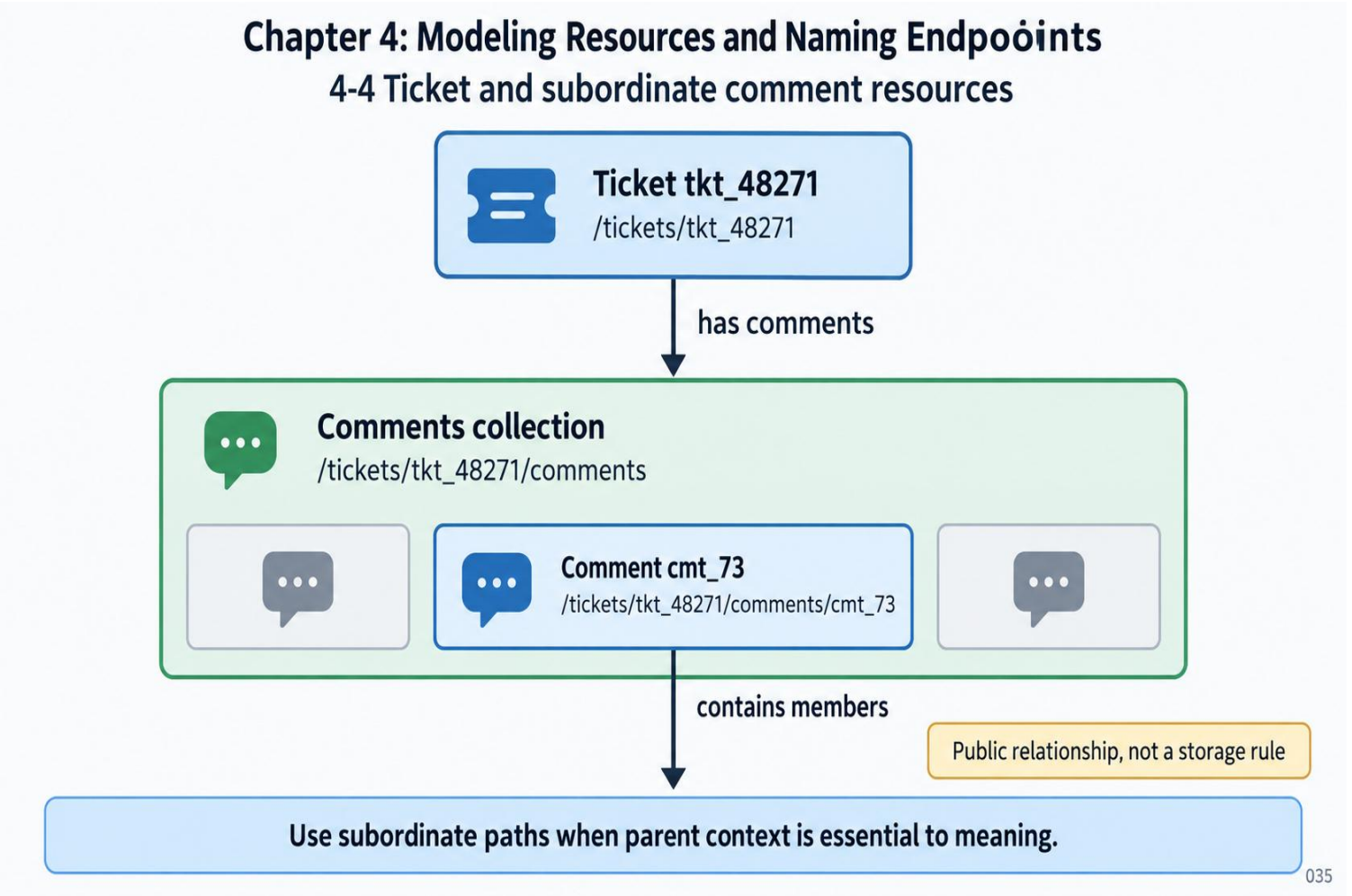



*Figure 4-4: Ticket and subordinate comment resources*

A **subordinate resource** is a resource whose API identity is presented in the context of a parent resource. This shape communicates that the comments being addressed belong to one ticket.

The path also supports natural reading:

```
GET /tickets/tkt_48271/comments
```

means “retrieve the comments belonging to ticket `tkt_48271`.”

Do not infer that comments must be stored inside the same database record as the ticket. URI structure expresses the public relationship, not physical storage.

## Represent relationships without over-nesting

Resources often relate to one another. A ticket may reference a requester, an assignee, comments, attachments, and a queue. There are two common ways to expose a relationship:

1. Include a related resource identifier or compact summary in the representation.
2. Provide a related or subordinate URI when consumers need to address the relationship directly.

A ticket response might contain:

```
{
  "id": "tkt_48271",
  "requesterId": "usr_204",
  "assigneeId": "usr_991",
  "queueId": "queue_identity",
  "commentCount": 4
}
```

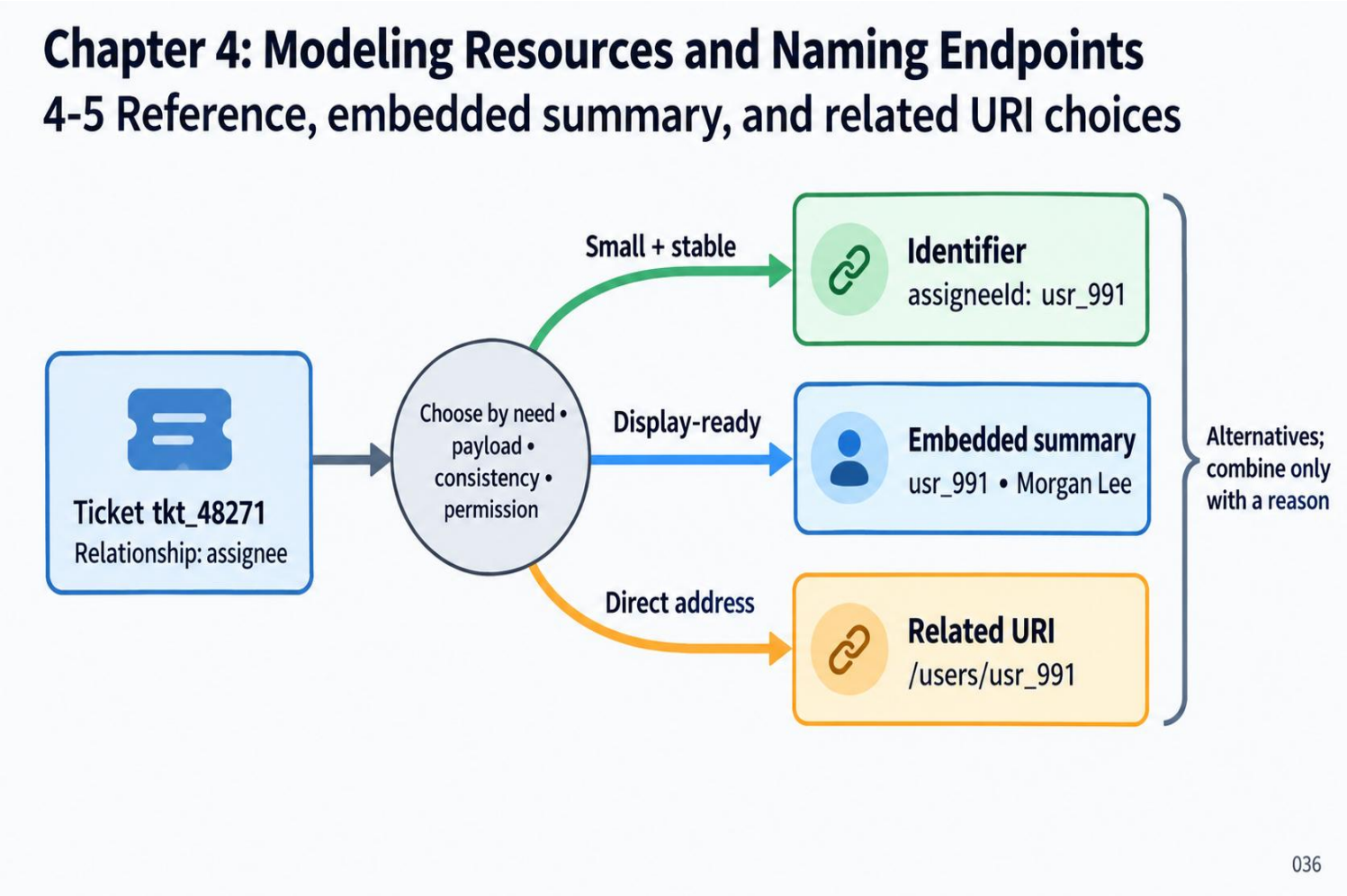



*Figure 4-5: Reference, embedded summary, and related URI choices*

This avoids copying full user and queue records into every ticket. Another design could include small embedded summaries for display convenience. The correct choice depends on consumer needs, payload cost, consistency expectations, and permission boundaries.

Deeply nested paths can become brittle:

```
/organizations/{organizationId}/departments/{departmentId}/queues/{queueId}/tickets/{ticketId}/comments/{commentId}
```

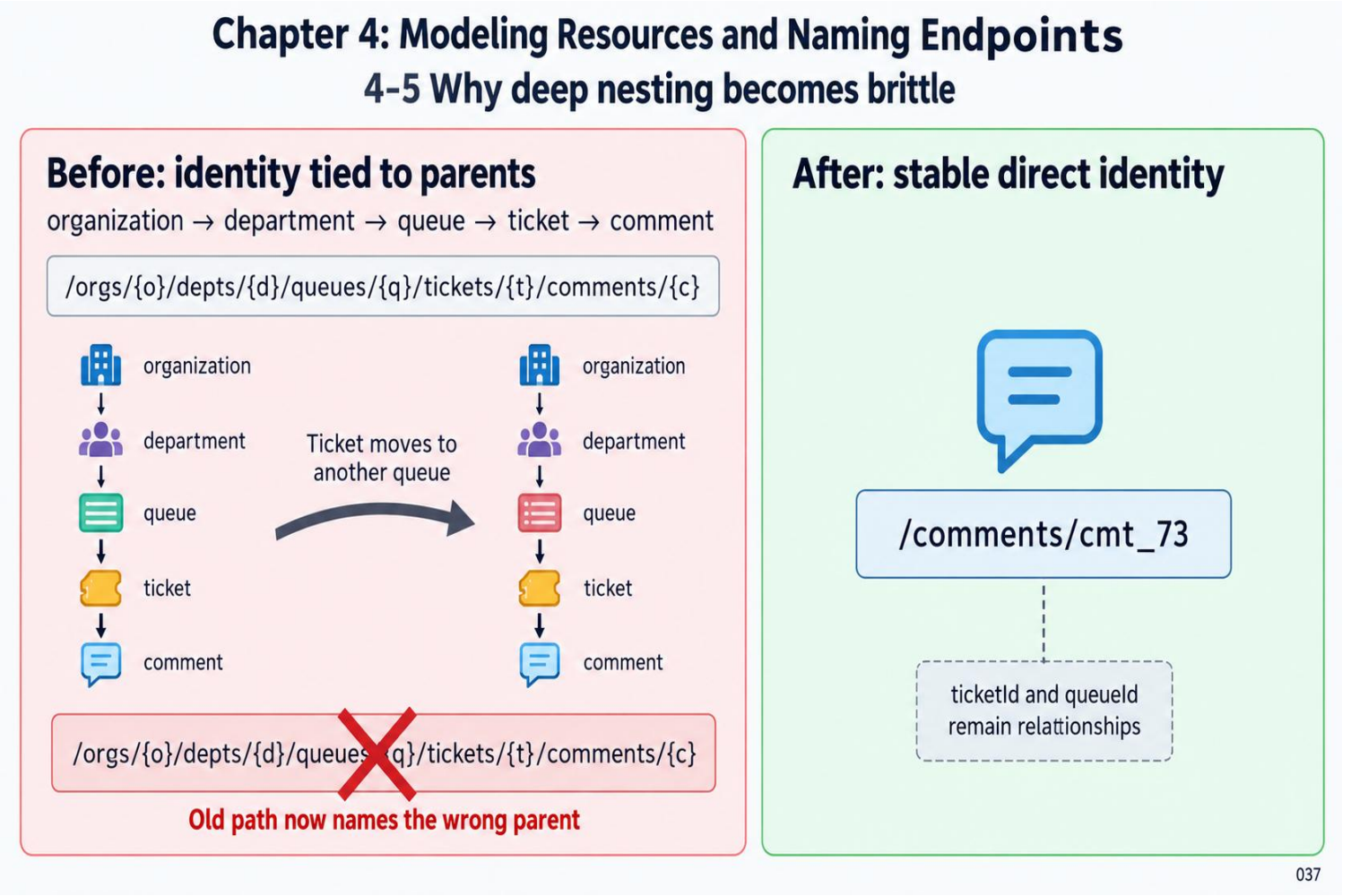



*Figure 4-6: Why deep nesting becomes brittle*

If `ticketId` and `commentId` are already globally unique, most of those parent segments may be unnecessary. They make clients carry information that the server can derive, and they imply that moving a ticket between queues changes its identity.

Use nesting when the parent relationship is essential to identification or meaning. Avoid nesting merely because database tables have foreign keys.

## Compare plausible comment designs

Several designs can be valid, but they communicate different models.

| Design | What it communicates | Main trade-off |
|---|---|---|
| /tickets/{ticketId}/comments | Comments are addressed in the context of a ticket. | Clear for common ticket workflows; direct comment access may require the ticket ID. |
| /comments/{commentId} | Comments are independently addressable resources. | Convenient for direct access; the ticket relationship must be represented elsewhere. |
| Both forms | The relationship and direct identity are both useful. | More surface area and more consistency obligations. |

Do not add both merely for convenience. Add an address form because a real consumer use case needs it.

## Distinguish resources from actions

Not every meaningful operation is naturally described as creating, retrieving, replacing, partially updating, or deleting a resource. A domain action may deserve an explicit command-style path when that shape communicates the behavior more honestly.

Examples might include:

```
/tickets/{ticketId}/assign
/tickets/{ticketId}/close
/tickets/{ticketId}/reopen
```

An **action endpoint** is an endpoint that represents a named business operation rather than ordinary manipulation of a resource representation.

Action endpoints should not be the default for every operation. Overuse can produce a list of unrelated remote commands with inconsistent HTTP behavior. They are useful when:

- The operation has important domain meaning.
- It has rules or side effects beyond changing one simple field.
- A generic update would hide required input or allowed transitions.
- The operation should be audited as a specific event.

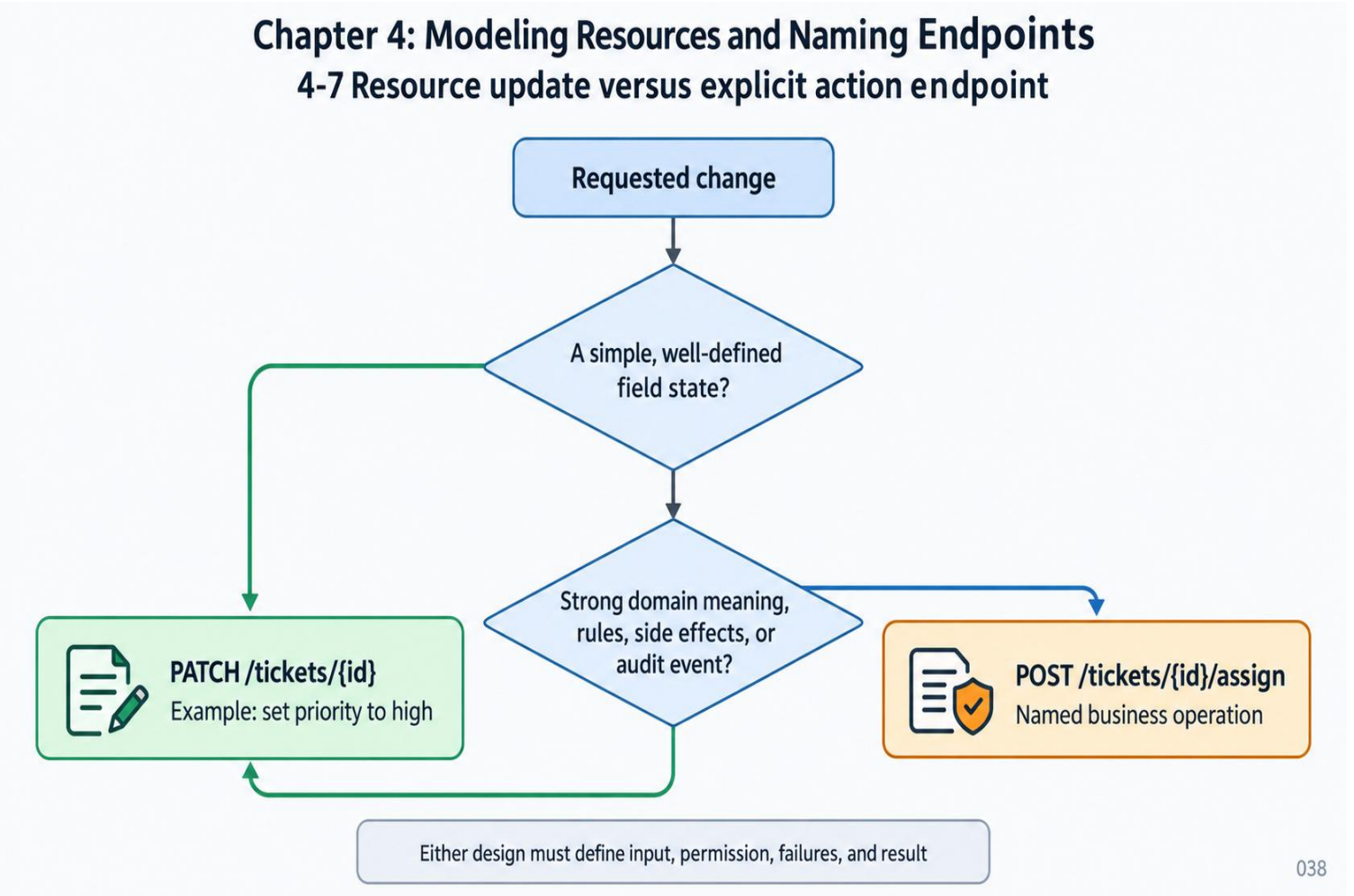



*Figure 4-7: Resource update versus explicit action endpoint*

Assignment illustrates the decision. In a simple domain, changing `assigneeId` with `PATCH /tickets/{ticketId}` may be clear and sufficient. In another domain, assignment may reserve capacity, change status, record acceptance, notify a queue, and enforce workload rules. There, an explicit assignment operation can make the contract clearer.

**REST-style resource orientation does not require pretending that every business capability is a trivial database update.**

## Avoid endpoints with unrelated responsibilities

An endpoint should have one coherent purpose. A request such as this is difficult to reason about:

```
POST /ticket-operations
```

```
{
  "createTicket": { "subject": "Printer offline" },
  "assignExistingTicket": { "ticketId": "tkt_48271" },
  "sendDailyReport": true
}
```

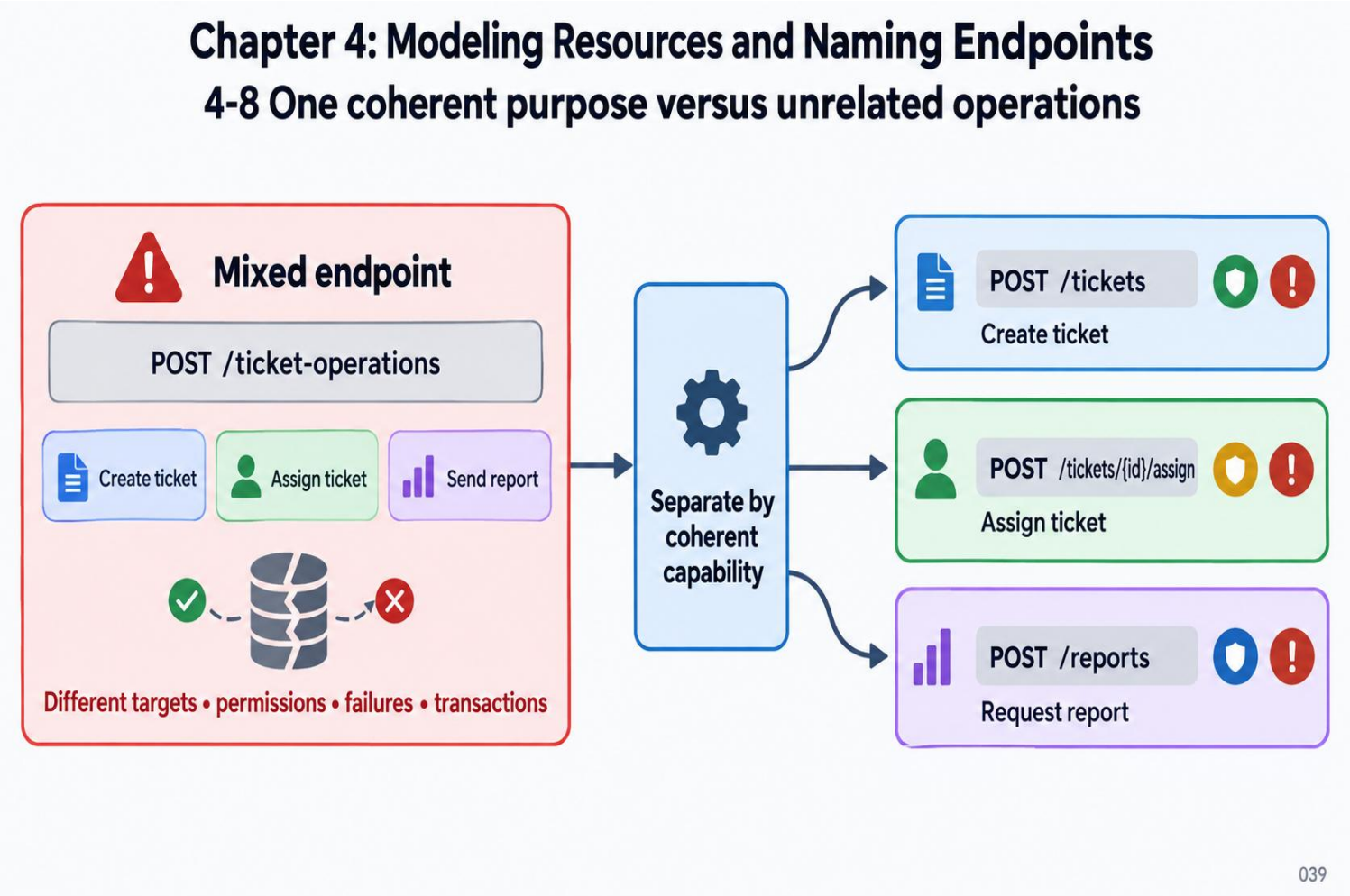



*Figure 4-8: One coherent purpose versus unrelated operations*

The operations have different targets, permissions, failures, and transaction boundaries. Combining them creates ambiguity about partial success and retry behavior.

A better design separates distinct capabilities unless the business explicitly requires them to succeed or fail as one atomic operation.

## A coherent ticket resource structure

One possible initial structure is:

```
/tickets
/tickets/{ticketId}
/tickets/{ticketId}/comments
/tickets/{ticketId}/comments/{commentId}
/tickets/{ticketId}/attachments
/tickets/{ticketId}/assign
/tickets/{ticketId}/close
```

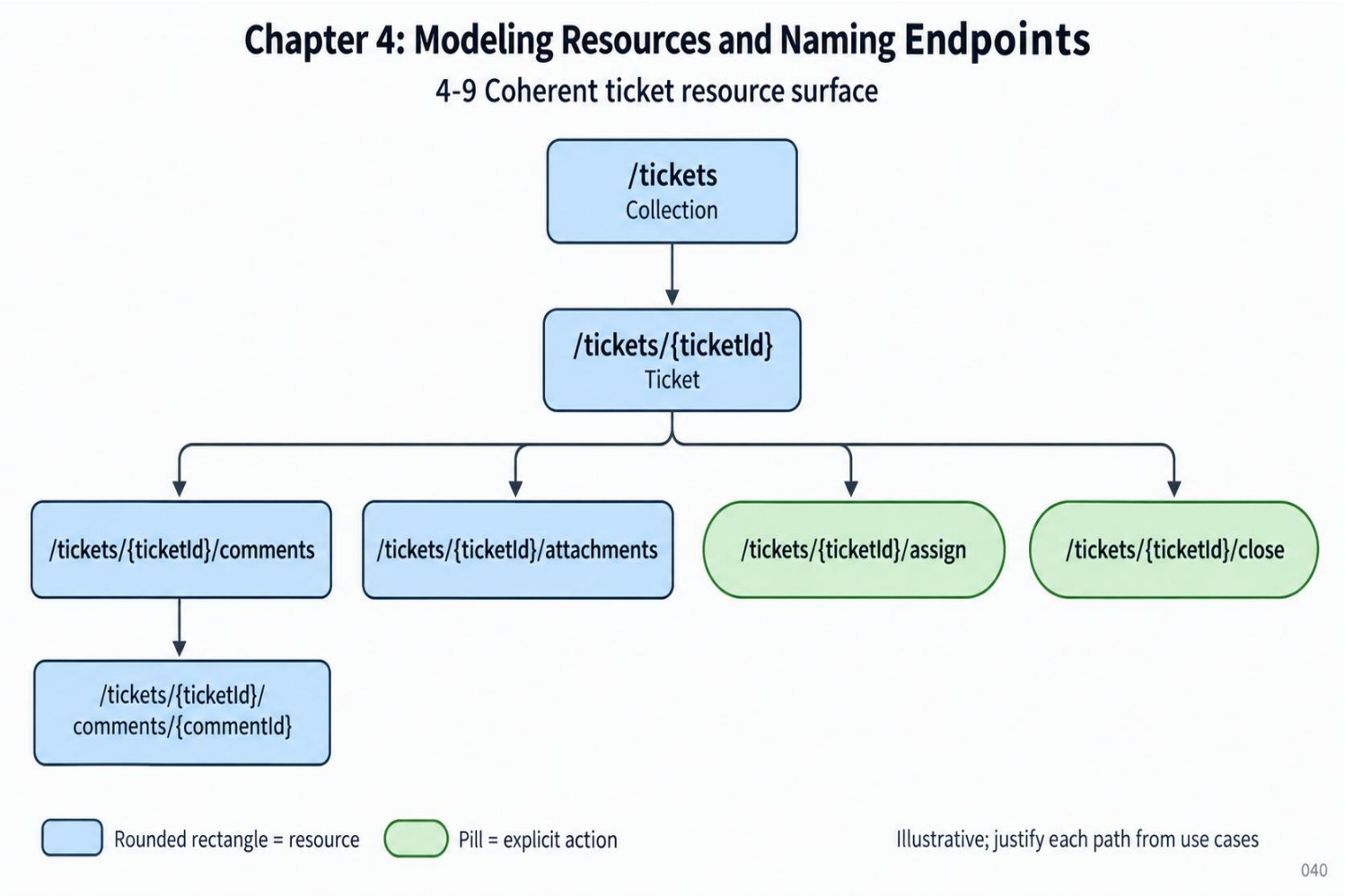



*Figure 4-9: Coherent ticket resource surface*

This is an example, not a universal template. A project might model assignments as their own resources, use partial ticket updates, or place attachments in a separate file service. The design is justified only when it matches the use cases and business rules.

For each path, be able to answer:

- What concept does it identify?
- Does that concept have stable identity?
- Why does the consumer need to address it?
- Is the parent-child relationship essential or merely an implementation detail?
- Is the path a resource or an explicit business action?
- Could its meaning remain stable if storage and internal code changed?

## Two recurring modeling errors

### "REST requires every operation to be pure CRUD"

Resource-oriented design encourages consistent use of resources and HTTP semantics. It does not require hiding meaningful domain behavior inside unrestricted field updates. When an action has distinct rules or consequences, an explicit operation or a well-modeled subordinate resource can be clearer.

### "Every database relationship should appear as a nested URL"

Database relationships support storage and queries. URI nesting should support consumer understanding and resource identity. Deep paths increase coupling and may make identity depend on mutable relationships.

## A naming and modeling review

Before accepting the endpoint structure, check that:

- Paths use stable domain terms rather than code or table names.
- Collections and individual resources follow a consistent pattern.
- Identifiers are stable and their scope is documented.
- Mutable labels and sensitive data are not used as identifiers without a strong reason.
- Nested paths express meaningful public relationships.
- Deep nesting is avoided when direct identity is sufficient.
- Explicit actions are used only where they clarify real business behavior.
- Each endpoint has one coherent responsibility.

**A good resource model lets consumers navigate the domain without learning how the provider stores or implements it.**

## Chapter Summary

- Resource-oriented design gives stable identities and predictable paths to domain concepts.
- Useful nesting expresses ownership or containment, while deep nesting can create brittle coupling.
- Explicit action endpoints are appropriate when a business transition cannot be represented honestly as ordinary field editing.

## Review Questions

1. What makes an API resource different from a database row?
2. When does nesting a subordinate resource improve clarity, and when does it become harmful?
3. Why might assigning a ticket deserve an explicit action rather than a generic update?

# Chapter 5 — Choosing Operations and HTTP Semantics

**HTTP methods and status codes are part of the API's meaning, not decorative syntax.** They tell clients and intermediaries whether an operation retrieves or changes state, whether it may be retried safely, and how to interpret the result.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Choose common HTTP methods according to their observable semantics.
- Distinguish safe behavior, idempotent effects, full replacement, partial modification, and deletion.
- Select success and redirect status codes that accurately describe the result a client can observe.

## Choose methods by observable behavior

The most common methods in JSON-based HTTP APIs are `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`.

| Method | Common purpose | Safe | Idempotent |
|---|---|---|---|
| GET | Retrieve a representation | Yes | Yes |
| POST | Create a resource or request a non-idempotent operation | No | Usually no |
| PUT | Create or fully replace the resource at a known URI | No | Yes |
| PATCH | Apply a partial modification | No | Depends on the patch semantics |
| DELETE | Remove a resource or make it no longer available | No | Yes |

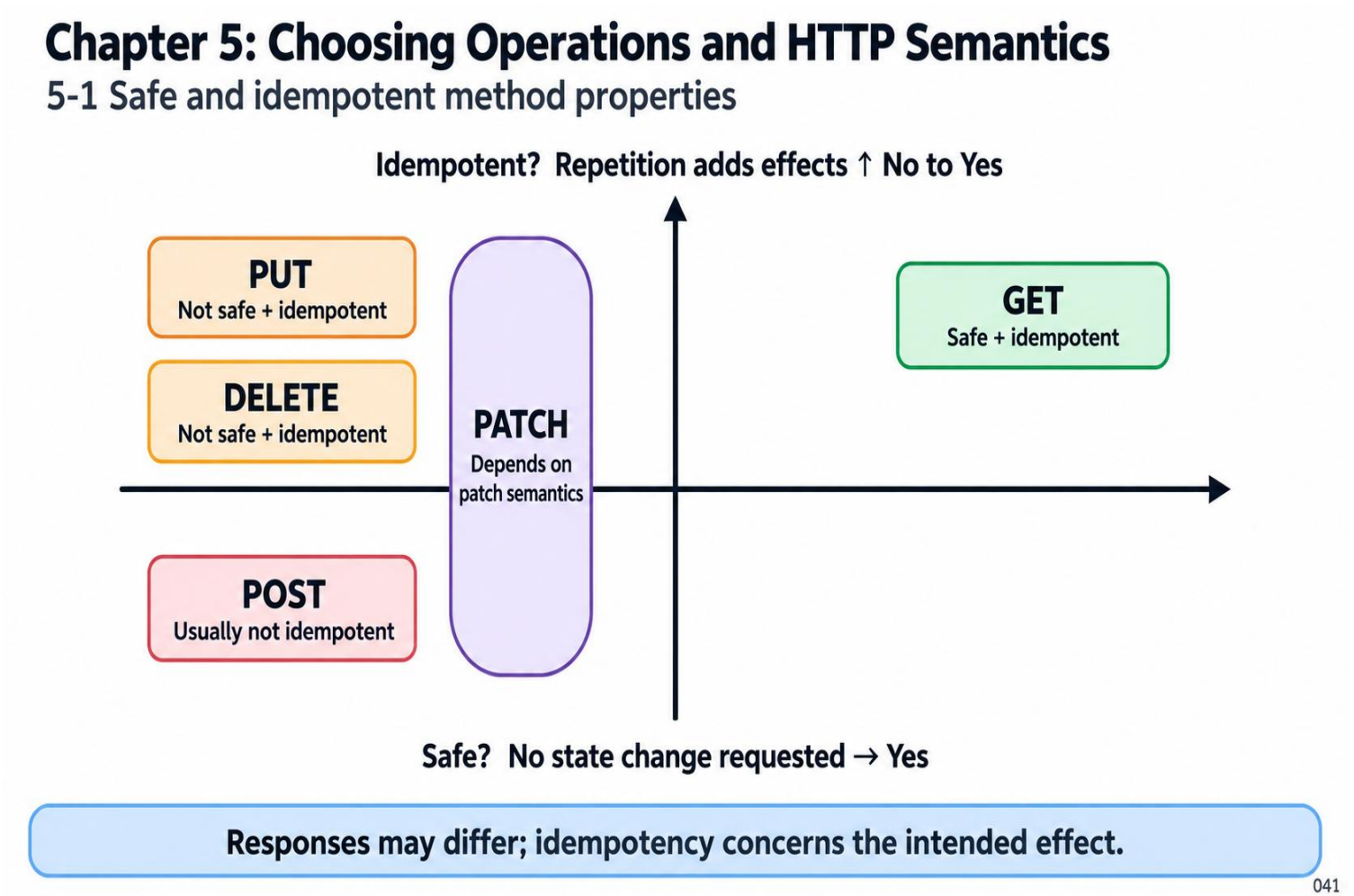



*Figure 5-1: Safe and idempotent method properties*

A **safe method** is intended only to retrieve information and not to request a state change. Logging a `GET` request or updating internal access statistics does not violate the idea; the client did not ask for a business-state change.

An **idempotent method** has the same intended effect when the same request is repeated as when it is sent once. The responses need not be byte-for-byte identical. A repeated `DELETE`, for example, may first return success and later report that the resource is already absent, while the intended final state remains the same.

These properties matter because requests can be duplicated. A client may not receive a response even though the server completed the operation. Network components may retry under specific conditions. If the method and operation semantics are clear, the client can make a safer recovery decision.

## Retrieval belongs on `GET`

Use `GET` when the caller is asking for a representation without requesting a business-state change:

```
GET /tickets/tkt_48271
```

A successful response commonly uses `200 OK` and includes the ticket representation.

Do not use `GET` for operations such as closing or deleting a ticket:

```
GET /tickets/tkt_48271/close
```

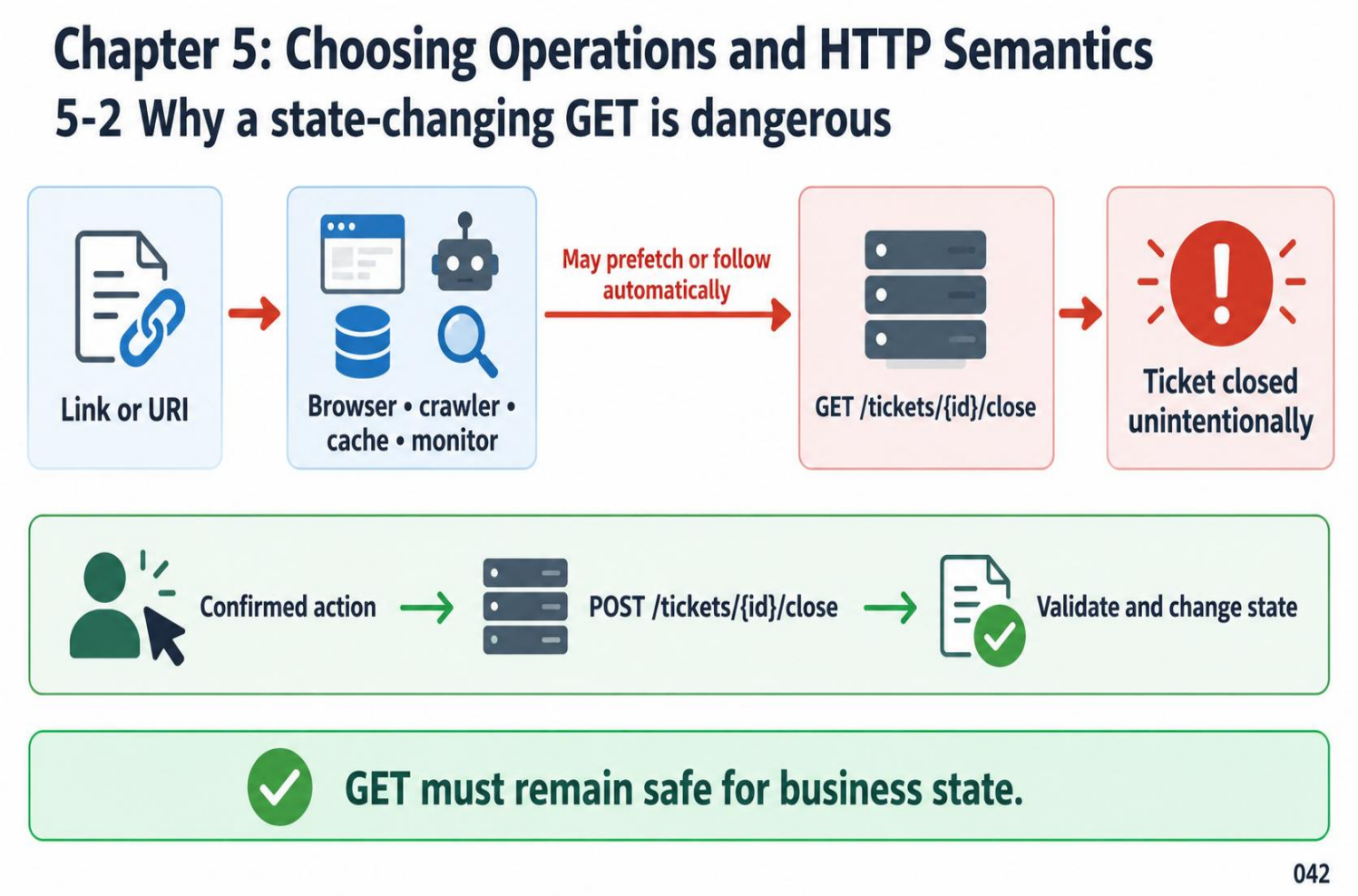



*Figure 5-2: Why a state-changing `GET` is dangerous*

That design is dangerous because browsers, crawlers, caches, monitoring tools, and software that follows links in advance may issue `GET` requests under the assumption that they are safe. A link should not close a ticket merely because software followed it.

## Creation commonly uses `POST`

Use `POST` on a collection when the server assigns the new resource's identifier:

```
POST /tickets
Content-Type: application/json

{
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "priority": "normal"
}
```

When creation completes immediately, a clear response is:

```
HTTP/1.1 201 Created
Location: https://api.example.com/tickets/tkt_48271
Content-Type: application/json

{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "status": "open",
  "priority": "normal",
  "createdAt": "2026-09-18T09:42:31Z"
}
```

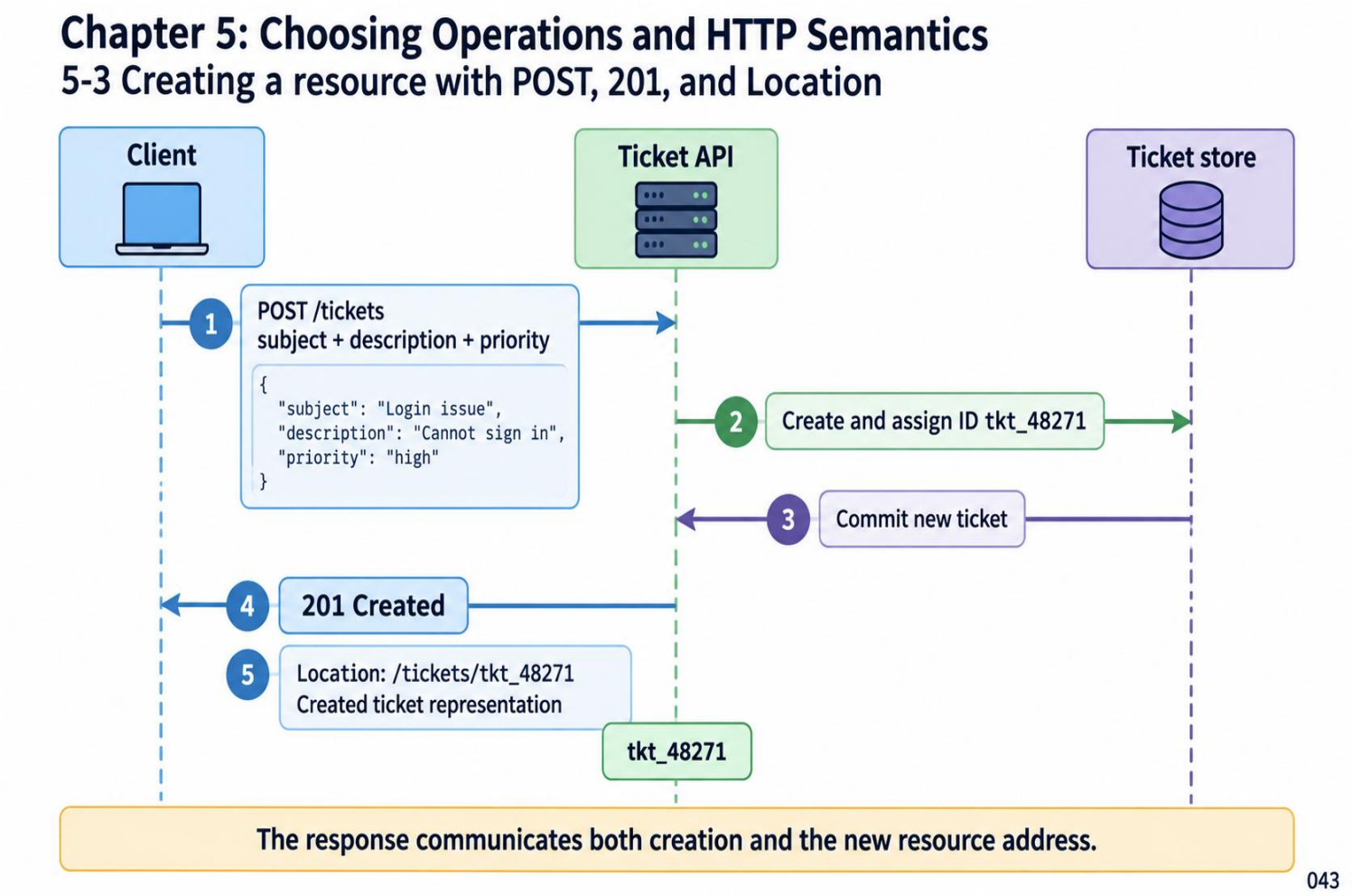



*Figure 5-3: Creating a resource with `POST`, `201`, and `Location`*

`201 Created` communicates that a resource was created. The `Location` header identifies the new resource. Returning its representation can save the client from making an immediate retrieval request, especially when the server generated fields or applied defaults.

### Creating a ticket with `curl`

**Prerequisites:** `curl` is available in a POSIX-style shell, the service is reachable, and `ACCESS_TOKEN` contains a valid credential accepted by this example service.

**Purpose:** Submit a JSON request that creates one support ticket.

```
curl --include \
  --request POST \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --header "Content-Type: application/json" \
  --data '{
    "subject": "Unable to reset password",
    "description": "The reset link expires immediately.",
    "priority": "normal"
  }' \
  "https://api.example.com/tickets"
```

`--request POST` selects the method. `Content-Type` identifies the request body as JSON. `--data` supplies the body.

**Expected result:** A successful response uses `201 Created`, includes a `Location` header for the new ticket, and returns the created representation if that is the documented contract. Exact authentication and response fields vary by environment.

A plain `POST` is generally not idempotent. Sending the same creation request twice may create two tickets. Chapter 9 explains how an idempotency key can make retries of a logical creation request safe.

## Use `PUT` for full replacement at a known URI

`PUT` means that the enclosed representation should become the state of the target resource, subject to server-controlled fields and business rules. It is typically used for full replacement rather than a small change.

For example:

```
PUT /ticket-preferences/usr_991
Content-Type: application/json

{
  "emailNotifications": true,
  "defaultQueueId": "queue_identity",
  "pageSize": 50
}
```

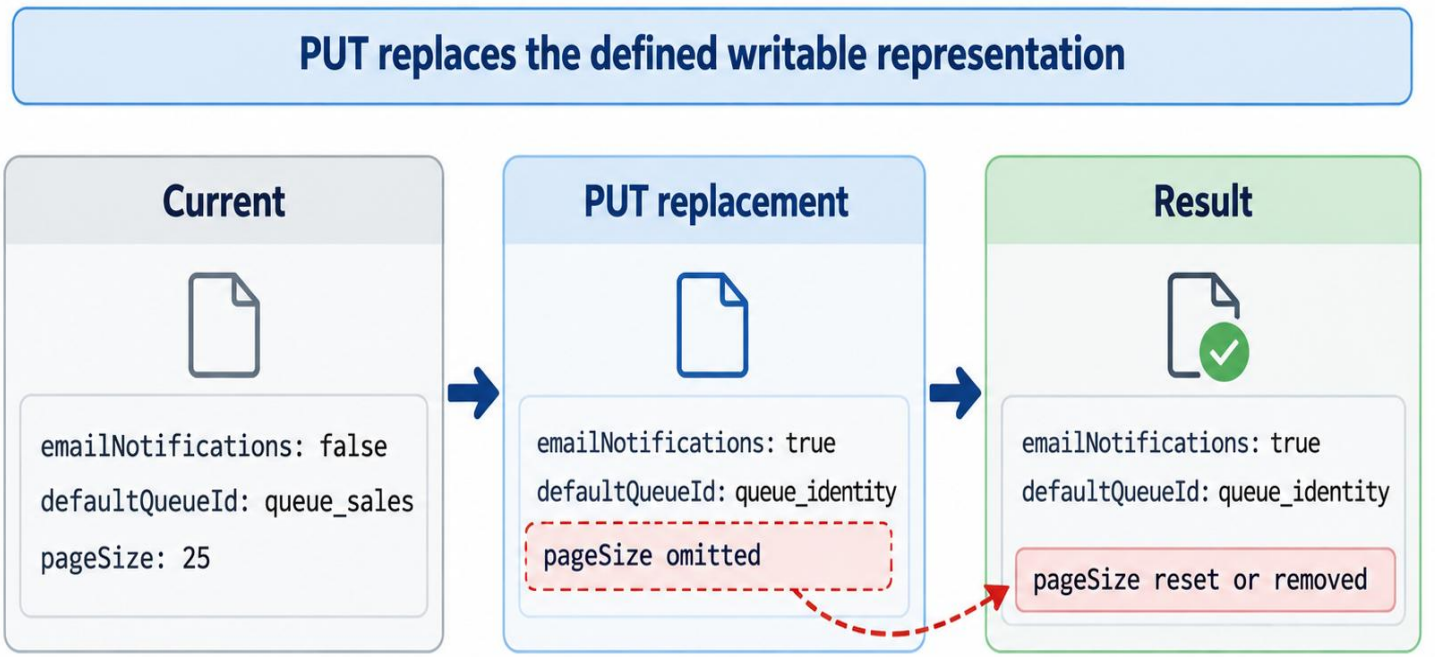





*Figure 5-4: Full replacement with `PUT`*

If the contract defines this as full replacement, omitting `pageSize` could mean that it is removed or reset rather than left unchanged. The API must state the rule clearly.

`PUT` is idempotent because repeating the same full replacement has the same intended effect. This does not mean that metadata such as `updatedAt` must remain identical on every response. The relevant question is whether repeating the request asks for another distinct business effect.

Do not label a partial, command-like update as `PUT` merely because a framework makes it convenient.

## Use `PATCH` for defined partial modification

`PATCH` requests a partial change to a resource. A simple ticket-priority update could be:

```
PATCH /tickets/tkt_48271
Content-Type: application/merge-patch+json

{
  "priority": "high"
}
```

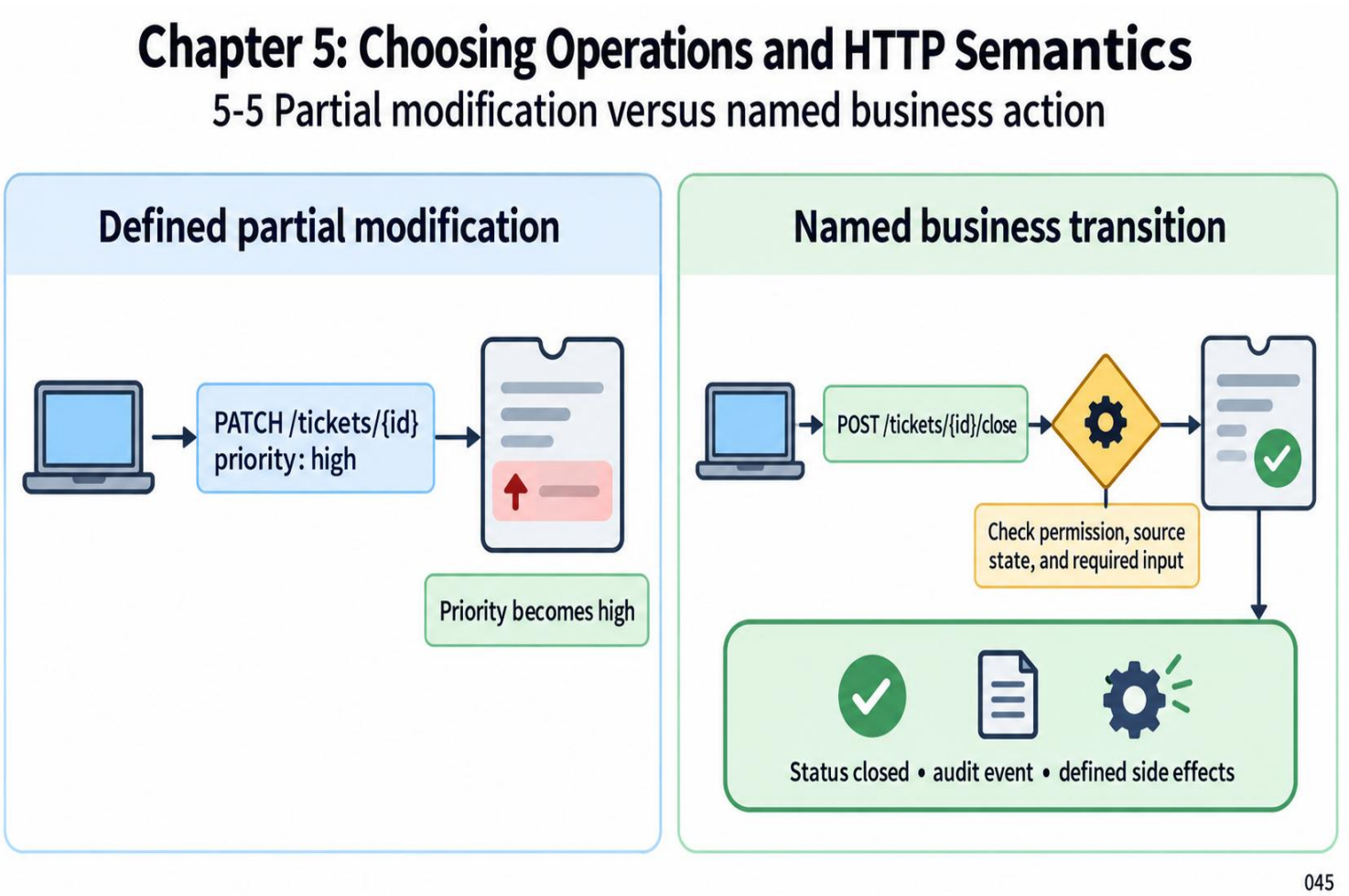



*Figure 5-5: Partial modification versus named business action*

The media type in this example identifies JSON Merge Patch, one standardized patch format. An API may use another documented patch format or a purpose-specific update body. The important requirement is to define exactly what omission, `null`, and each operation mean.

Not every `PATCH` is idempotent. Setting `priority` to `high` can be idempotent. A patch instruction such as "increment escalation count" is not: repeating it creates another effect. Method names alone do not eliminate the need to define behavior.

For business transitions with strong meaning, an explicit operation may be clearer than a general patch:

```
POST /tickets/tkt_48271/close
```

The design should state what closing means, what input is required, which source states are allowed, and what response follows.

## Use `DELETE` for removal semantics

A typical deletion request is:

```
DELETE /tickets/tkt_48271
```

The contract must define what "delete" means in the domain. Possibilities include permanent removal, **soft deletion** that marks the resource unavailable without immediately erasing stored data, removal or transformation of identifying data, or making a resource inaccessible through the API. The HTTP method alone does not choose the retention policy.

A successful deletion often uses:

- `204 No Content` when no response body is needed.
- `200 OK` when a response body communicates a useful result.

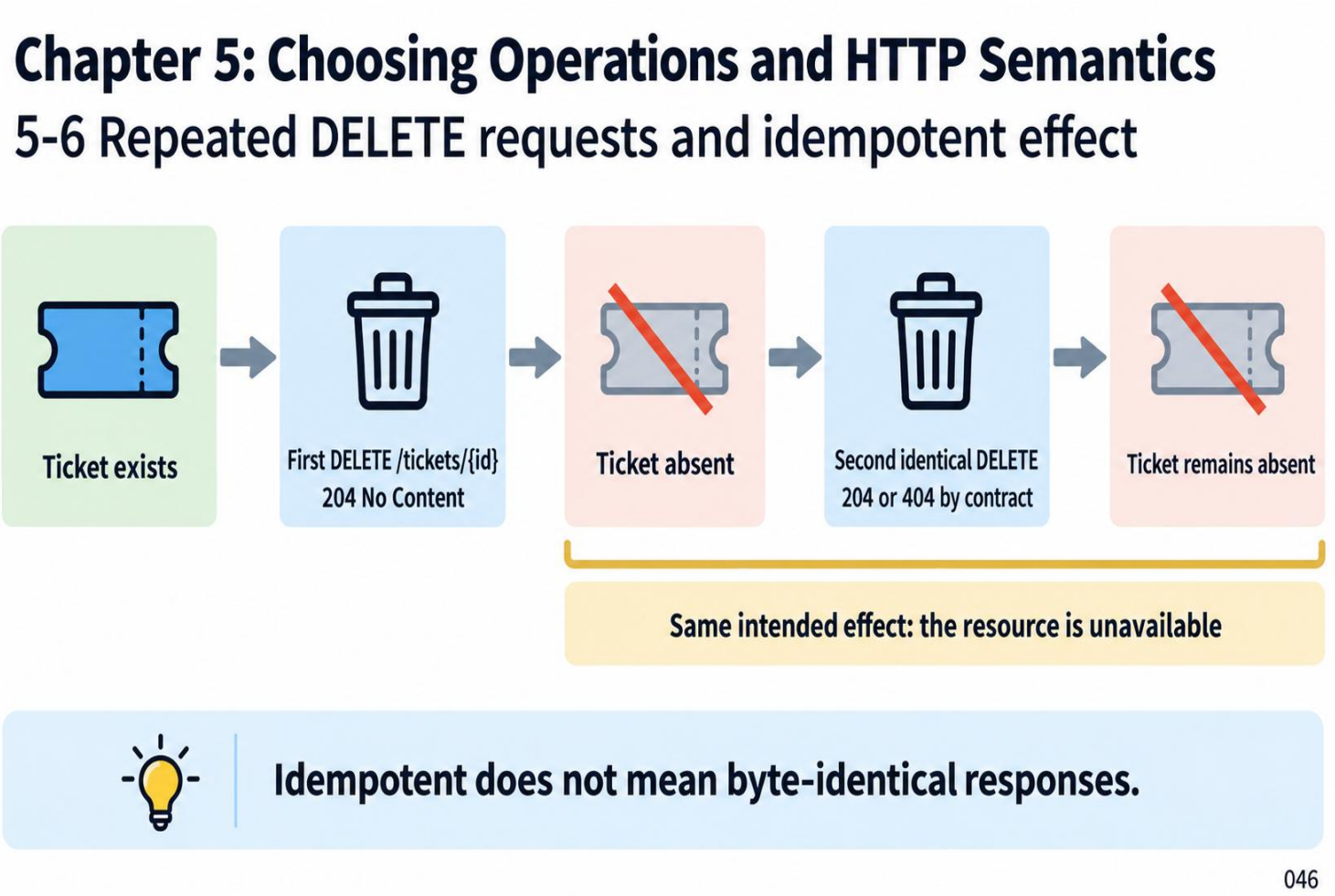



*Figure 5-6: Repeated `DELETE` requests and idempotent effect*

Because `DELETE` is idempotent, repeating the request should not create a new business effect. The second response may differ because the resource is already absent.

## Select success status codes deliberately

The status code should describe the observable result.

| Status | Meaning in an API contract | Typical example |
|---|---|---|
| 200 OK | The request succeeded and a response body is returned. | Retrieve or update a ticket and return its representation. |
| 201 Created | A resource was created. | Create a ticket and return its URI. |
| 202 Accepted | The request was accepted for processing, but processing is not complete. | Accept a long-running export request. |
| 204 No Content | The request succeeded and there is no response body. | Delete a ticket or apply an update with nothing further to return. |

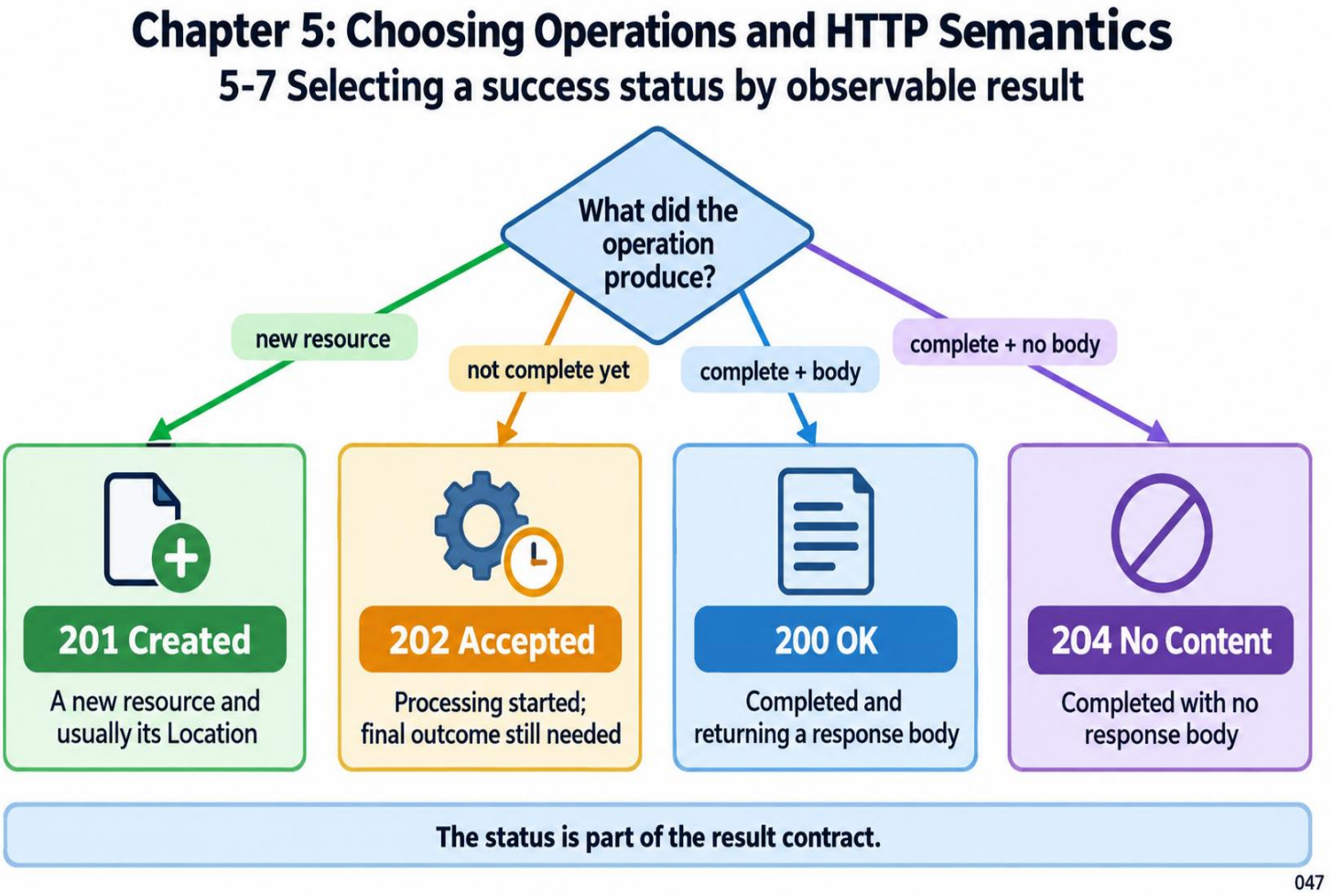



*Figure 5-7: Selecting a success status by observable result*

`202 Accepted` is not a promise that processing will ultimately succeed. If an API uses it, the contract needs some way for the consumer to learn the later outcome. The full asynchronous pattern is beyond the scope of these chapters, but the status must not be treated as completed success.

`204 No Content` means exactly that: clients should not expect a response body. If the provider needs to return the changed resource, use a status that permits the documented body, commonly `200 OK`.

**Do not return `200 OK` for every successful request simply to reduce status-code variety.** The distinctions communicate useful facts to clients, generated tools, tests, gateways, and operators.

## Redirects need explicit meaning

A **redirect** is a response that tells a client to use another URI, usually through a `Location` header and a status in the `3xx` class reserved for redirection. Redirects can be useful when a resource has moved or a preferred authoritative URI should be used.

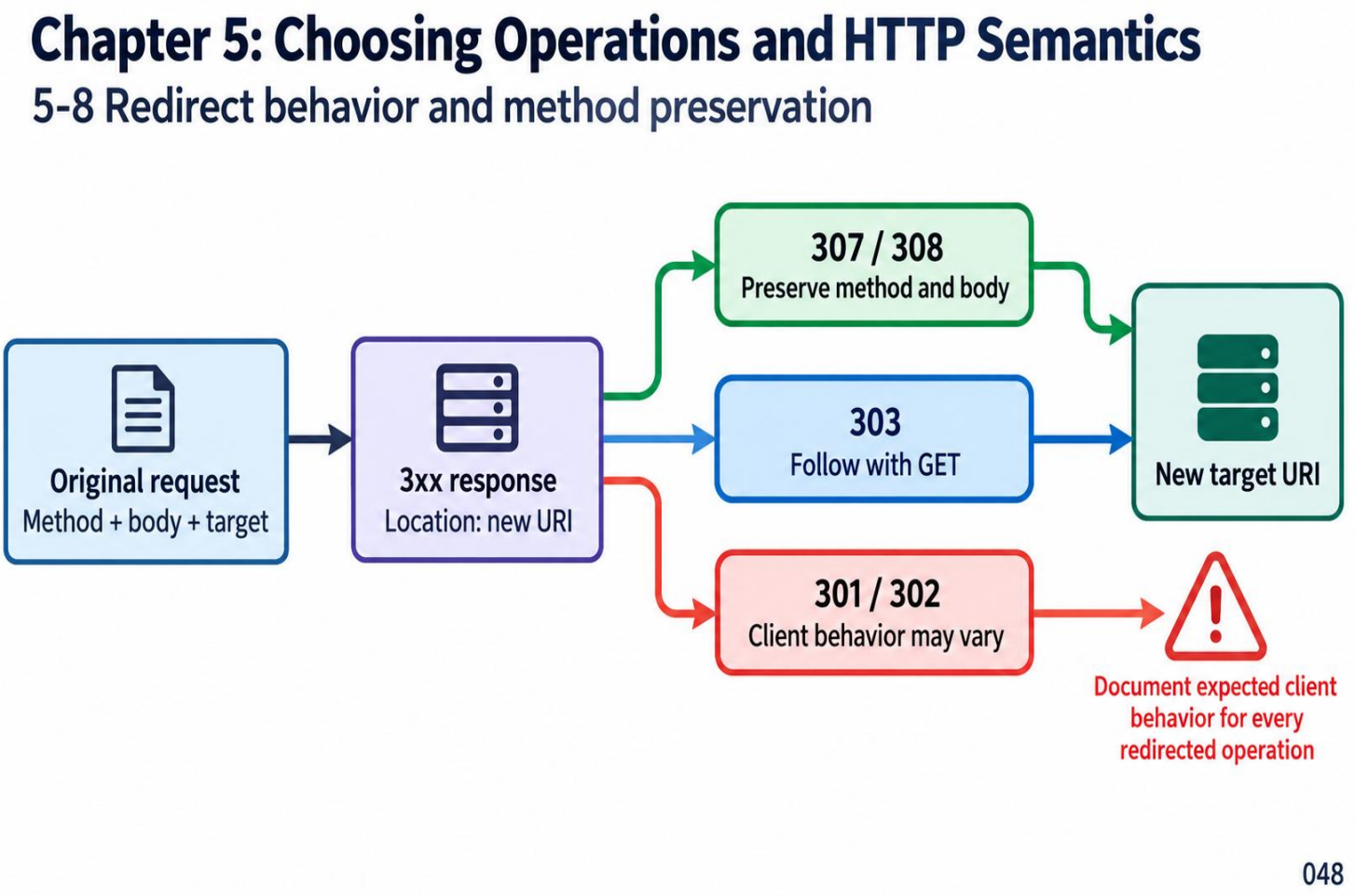



*Figure 5-8: Redirect behavior and method preservation*

Clients should not be forced to guess whether they may repeat a state-changing method at the new location or whether the redirect is temporary. Different redirect status codes have different method-preservation semantics, and client libraries do not all behave identically. When redirects are part of the API, document which operations may return them and what the client must do.

For most ordinary business operations, returning the intended resource location directly is simpler than relying on surprising redirect behavior.

## Do not assume every message supports a body equally

HTTP permits bodies in several contexts, but ecosystem support and semantics differ. Practical rules include:

- Do not depend on a request body with `GET`; many tools, caches, and intermediaries do not handle it consistently.
- Do not include a body with `204 No Content`.
- State the expected request media type whenever a body is accepted.
- State the response media type whenever a body is returned.
- Do not require clients to infer a body schema solely from the path.

If a search needs complex structured criteria that cannot reasonably fit **query parameters**, which are name-value inputs carried in the URI after `?`, a purpose-specific `POST` search operation can be clearer than a `GET` body, even though it may have different caching characteristics.

## Applying semantics to ticket operations

One coherent design could use:

| Consumer goal | Method and path | Typical success |
|---|---|---|
| Create a ticket | `POST /tickets` | `201 Created` with `Location` and representation |

| Consumer goal | Method and path | Typical success |
|---|---|---|
| Retrieve a ticket | `GET /tickets/{ticketId}` | `200 OK` with representation |
| Change editable ticket fields | `PATCH /tickets/{ticketId}` | `200 OK` with updated representation |
| Replace a complete settings resource | `PUT /ticket-preferences/{userId}` | `200 OK` or `204 No Content` |
| Close a ticket through a defined transition | `POST /tickets/{ticketId}/close` | `200 OK` with resulting representation |
| Delete a ticket where policy permits | `DELETE /tickets/{ticketId}` | `204 No Content` |

This is one example. A different domain might model closure as creation of a resolution resource or assignment as replacement of an assignment resource. The correct choice follows the observable business behavior.

## Two high-impact misunderstandings

### "Idempotent means every repeated response must be identical"

Idempotency concerns the intended server-side effect. Timestamps, response headers, or the status reported after the state already exists may differ. Repeating the request must not create another distinct effect.

### "Any successful request should return `200 OK`"

`201`, `202`, and `204` communicate facts that `200` does not. Using them correctly improves client control flow and makes the contract more precise.

## Operation review questions

For every endpoint, verify:

- Does the method match the requested behavior?
- Is a retrieval operation safe?
- Is the operation idempotent, and is that clear to callers?
- If `PUT` is used, is full replacement defined?
- If `PATCH` is used, are partial-update semantics defined?
- Does the success status describe the actual result?
- Are required response headers, such as `Location`, documented?
- Could a retry create duplicate or repeated business effects?
- Are request and response bodies valid for the selected semantics?

**Correct HTTP semantics make the API easier to use safely because the method and status already communicate part of the contract.**

## Chapter Summary

- HTTP methods communicate guarantees that clients and intermediaries use.
- Idempotency concerns the intended effect of repetition, not whether every response byte is identical.
- Status codes should report the actual outcome, including creation, no-content success, asynchronous acceptance, and redirection behavior.

## Review Questions

1. Why is a state-changing `GET` dangerous even when the endpoint works in a browser?
2. How do `PUT` and `PATCH` differ in the contract they communicate?
3. When is `201 Created` more informative than `200 OK`?

# Chapter 6 — Designing Requests and Responses

**A dependable data contract gives every field a stable meaning, a precise type, and explicit rules for presence, absence, validation, and ownership.** Clients should not need to guess whether an omitted field means “unchanged,” “unknown,” “not applicable,” or “cleared.”

## What You’ll Be Able to Do

After completing this chapter, you will be able to:

- Design request and response representations with clear field names, types, direction, and ownership.
- Distinguish missing values, explicit `null`, empty values, and server-applied defaults.
- Choose practical representations for identifiers, time, money, enumerations, media types, and files.

## A representation is not the resource itself

A **representation** is the data form used in a message to describe a resource. The ticket is the domain resource; a JSON object returned by the API is one representation of its current state.

This distinction matters because a representation can:

- Omit internal or sensitive data.
- Combine data from multiple internal sources.
- Use consumer-friendly names and types.
- Include derived values not stored directly.
- Differ between creation, update, and retrieval operations.

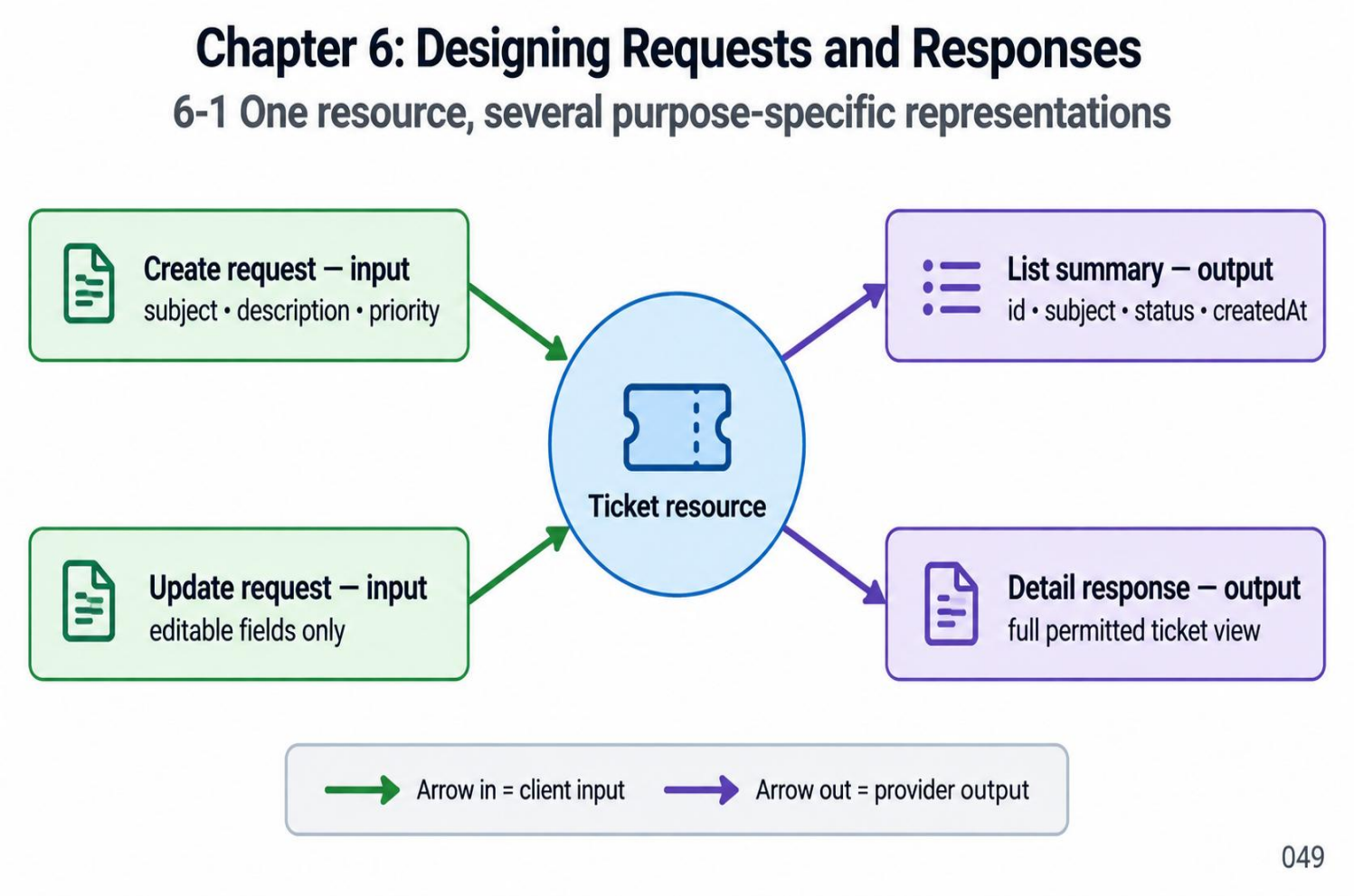



*Figure 6-1: One resource, several purpose-specific representations*

The provider does not need to expose its internal object or database row exactly as stored.

## JSON provides a small set of value types

JSON supports:

- Objects containing named members.
- Arrays containing ordered values.
- Strings.
- Numbers.
- Booleans: `true` and `false`.
- `null`.

A ticket-creation request might be:

```
{
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "priority": "normal",
  "tags": ["identity", "self-service"]
}
```

The object has strings and an array of strings. JSON itself does not know that `priority` must be one of a defined set or that `subject` has a maximum length. Those are contract rules that must be documented and validated.

## Use one field-naming convention

Choose a consistent naming convention, such as `camelCase`, and apply it throughout the API:

```
{
  "createdAt": "2026-09-18T09:42:31Z",
  "assigneeId": "usr_991"
}
```

Do not mix styles without a strong compatibility reason:

```
{
  "created_at": "2026-09-18T09:42:31Z",
  "AssigneeID": "usr_991"
}
```

Consistency reduces client mapping code and review effort. It also helps consumers predict fields they have not seen before.

Field names should express domain meaning. `resolutionDueAt` is clearer than `date2`; `requesterId` is clearer than `userRef` when the business role is a requester.

## Missing, `null`, empty, and default are different states

These values are often confused, but they can carry different meanings.

| Form | Example | Possible meaning |
|---|---|---|
| Missing field | `{}` | Not supplied, not selected, unchanged, or not available under this operation. |
| Explicit `null` | `{"assigneeId": null}` | Known absence, or a request to clear the value if the contract permits it. |
| Empty string | `{"description": ""}` | A present string with zero characters. |
| Empty array | `{"tags": []}` | A present collection with no members. |
| Defaulted value | Field omitted and server chooses `normal` | The provider applies a documented value. |

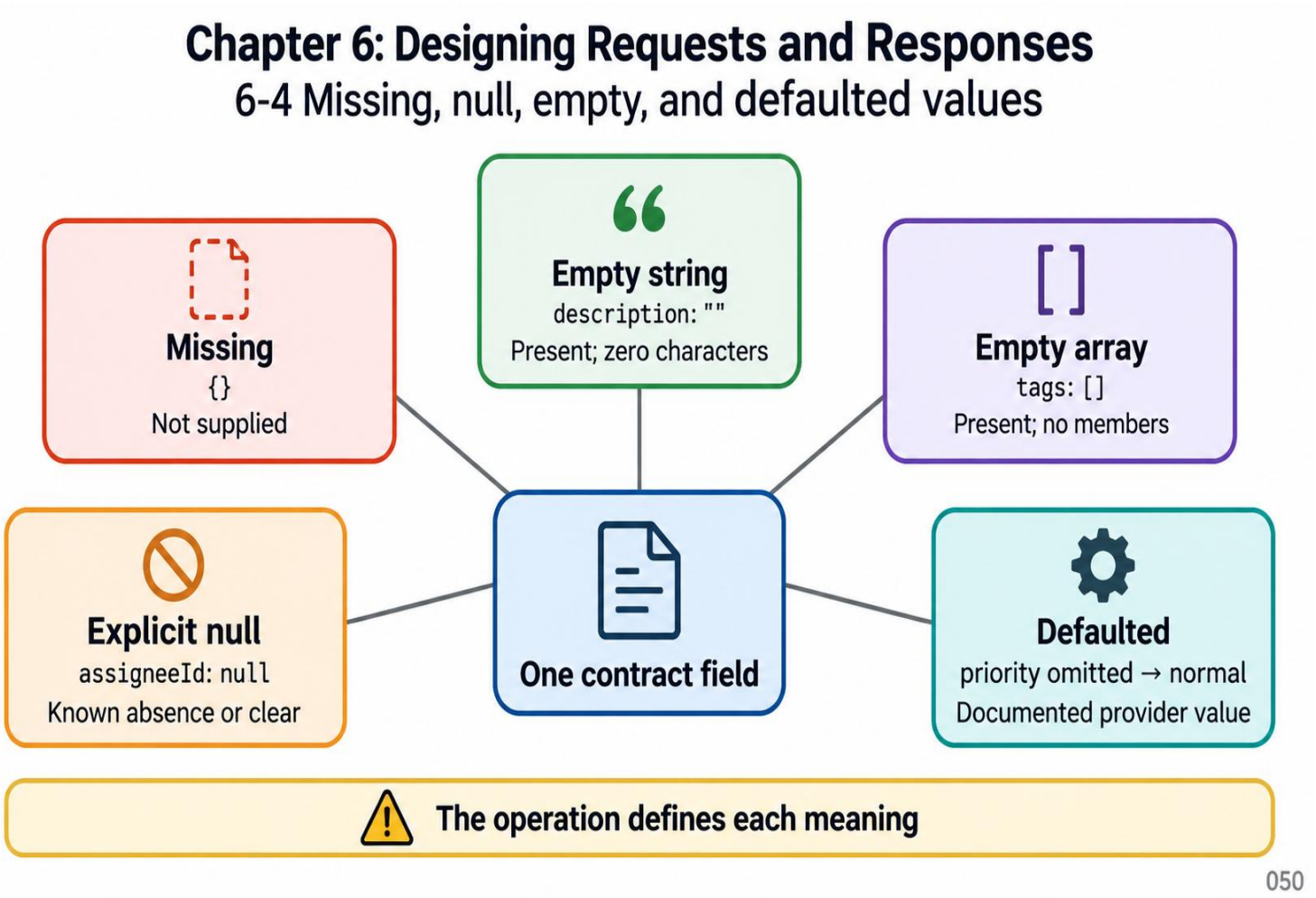



*Figure 6-2: Missing, null, empty, and defaulted values*

The contract must choose meanings instead of leaving them to implementation accident.

For a creation request, omitting `priority` might apply a documented default of `normal`. Sending `"priority": null` might be invalid because every ticket needs a priority. For an update request, omitting `assigneeId` might mean "do not change it," while `"assigneeId": null` might mean "unassign the ticket." These are distinct operations.

**Optional and nullable are not synonyms.** An optional field may be omitted. A nullable field may be present with `null`. A field can be optional but not nullable, required and nullable, both, or neither.

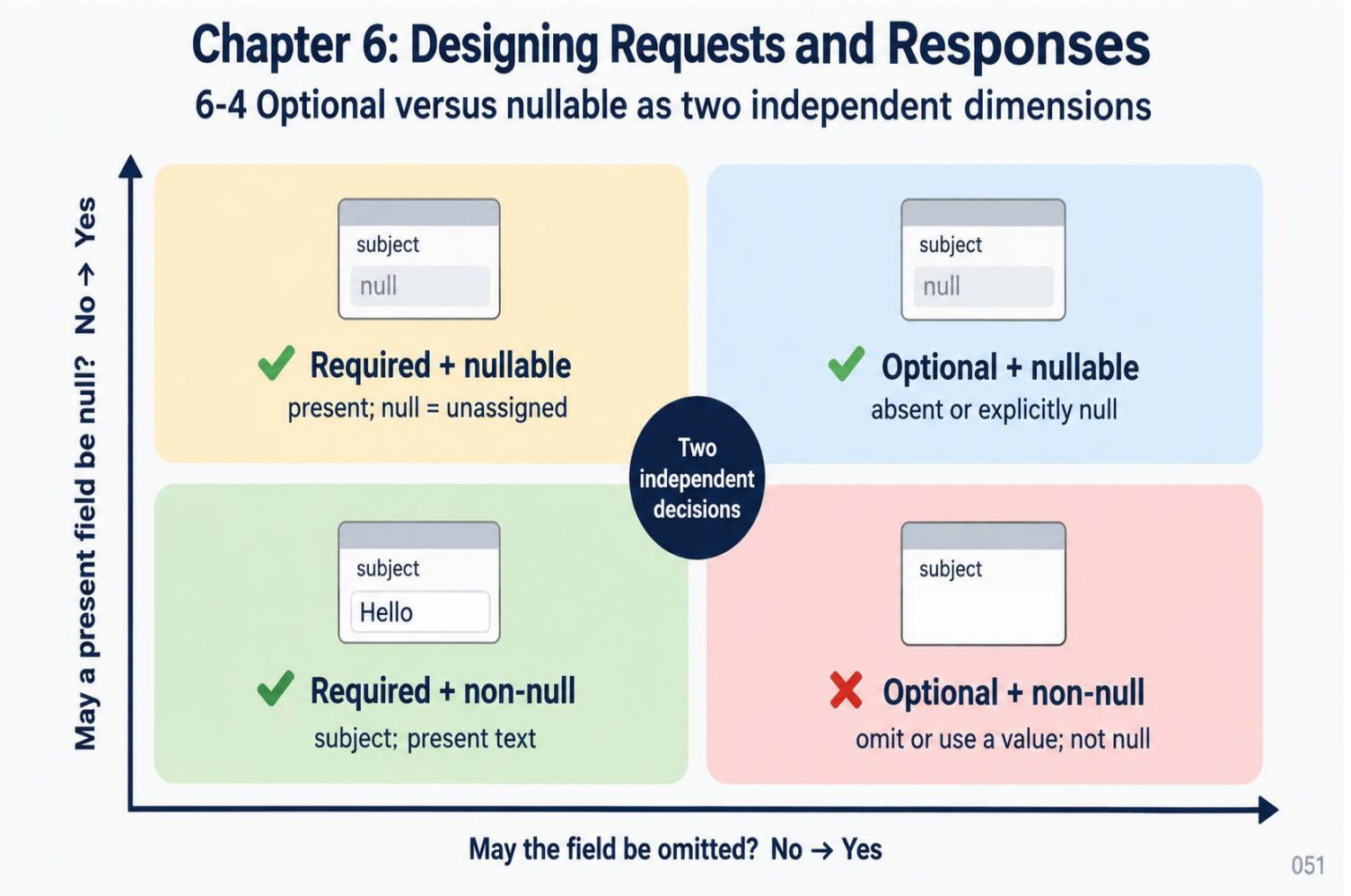



*Figure 6-3: Optional versus nullable as two independent dimensions*

## Define field direction and ownership

Fields can have different roles in requests and responses:

- **Required input:** The client must supply the field for that operation.
- **Optional input:** The client may supply it; omission has a defined meaning.
- **Read-only:** The server returns it, but clients must not set it.
- **Write-only:** The client sends it, but the server does not return it, as with some secrets.
- **Conditionally required:** It is required only when a documented condition holds.

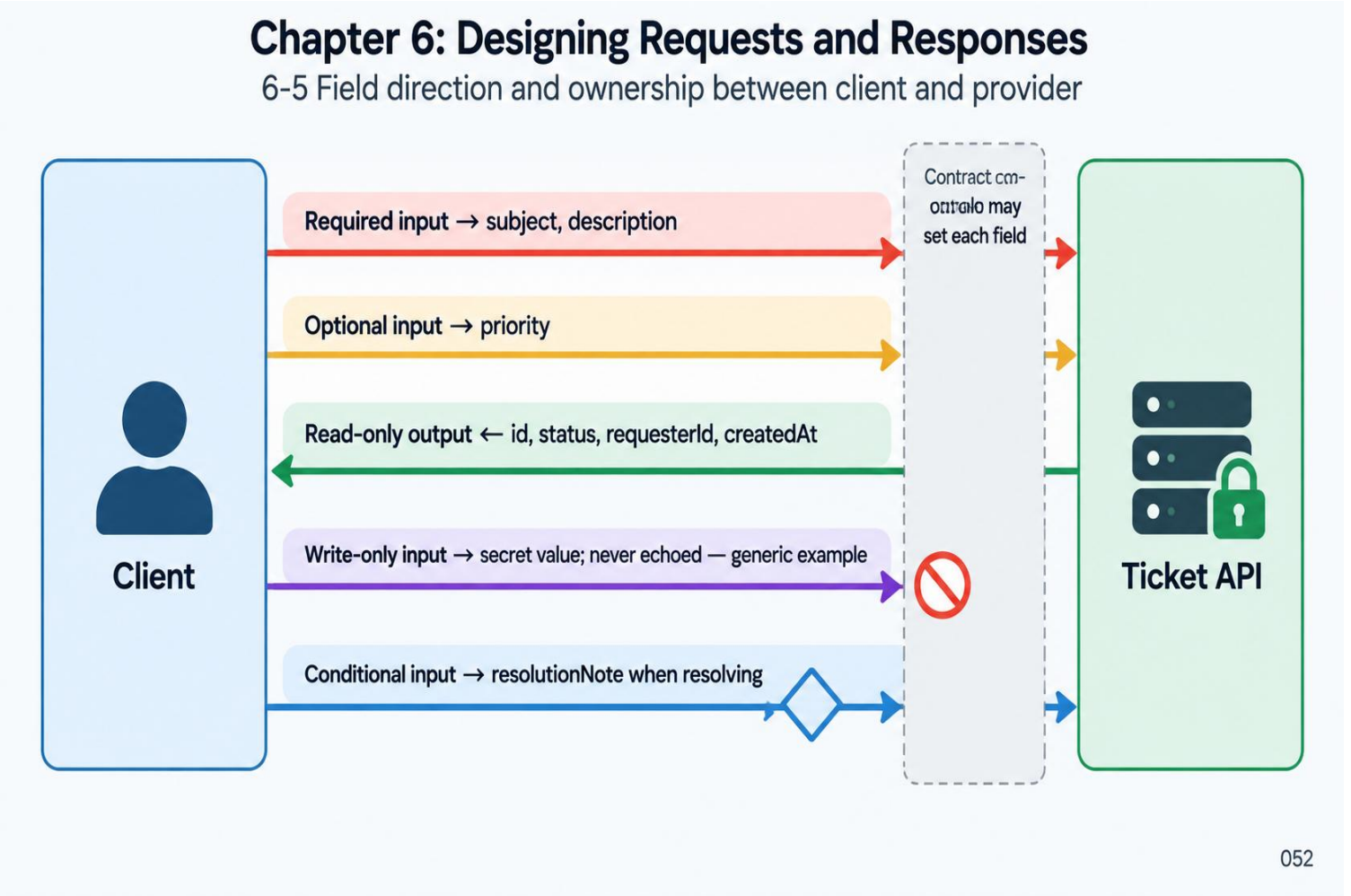



*Figure 6-4: Field direction and ownership between client and provider*

For ticket creation:

```
{
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "priority": "normal"
}
```

A response can include server-controlled fields:

```
{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "status": "open",
  "priority": "normal",
  "requesterId": "usr_204",
  "createdAt": "2026-09-18T09:42:31Z",
  "updatedAt": "2026-09-18T09:42:31Z"
}
```

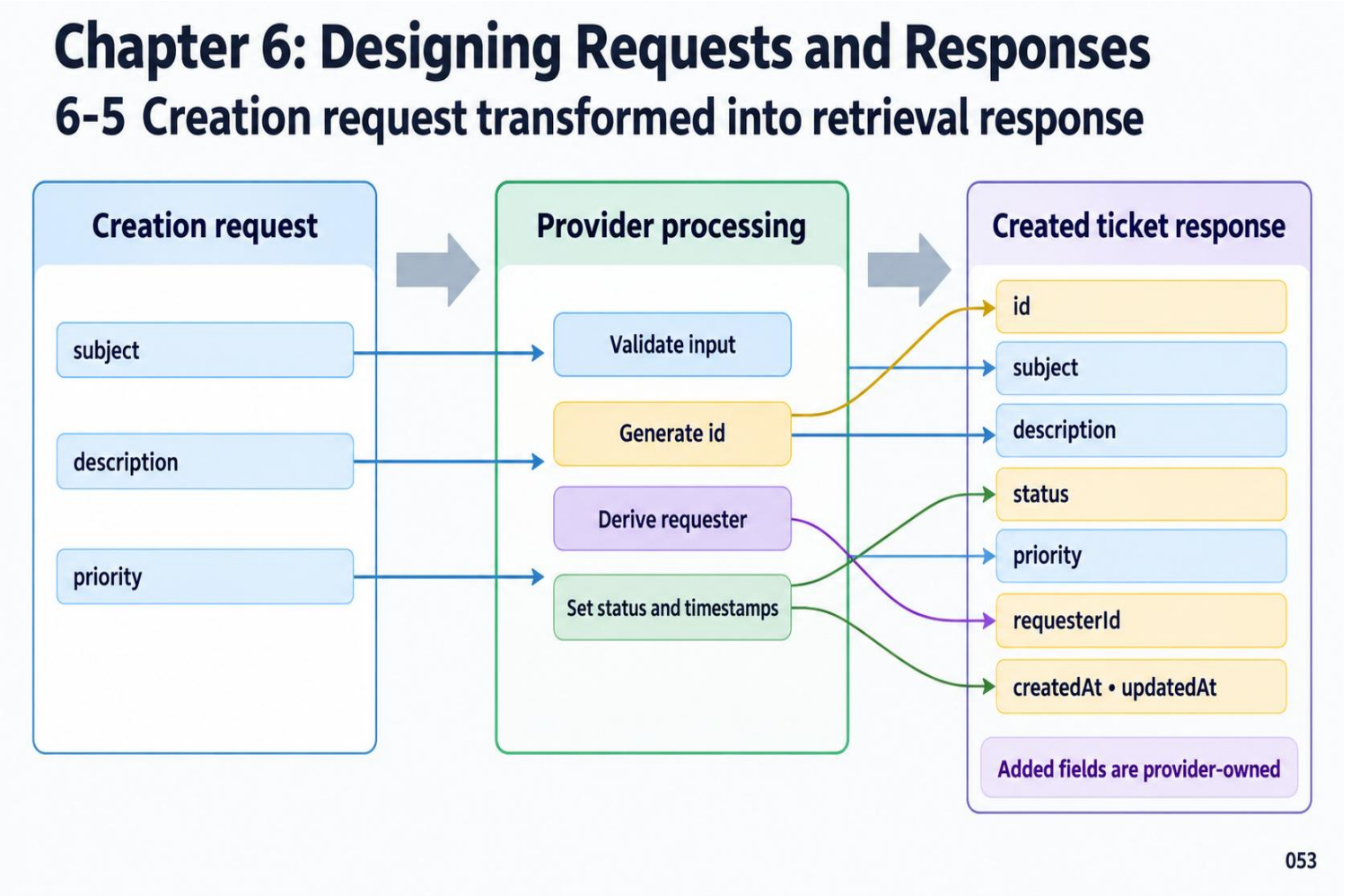



*Figure 6-5: Creation request transformed into retrieval response*

The server generates `id`, derives `requesterId` from the authenticated context in this example, sets the initial `status`, and records timestamps. Accepting those values from an ordinary client would create integrity and authorization problems.

Request and response models therefore do not have to be identical. Reusing one internal class for both is convenient in code but can accidentally make server-controlled fields writable.

## Choose types for meaning, not convenience

### Identifiers

Represent identifiers as strings unless consumers genuinely need numeric arithmetic. Even when an internal key is numeric, a JSON string avoids integer-size differences across languages and discourages mathematical interpretation.

```
{ "id": "tkt_48271" }
```

Clients should preserve identifiers exactly and not parse hidden structure unless the contract defines it.

### Timestamps and dates

A **timestamp** identifies a point in time. Use a standard, unambiguous text form with a time-zone offset, commonly an RFC 3339 form such as:

```
{ "createdAt": "2026-09-18T09:42:31Z" }
```

`Z` means Coordinated Universal Time. An explicit offset, such as `+09:00`, can also identify an instant.

A calendar date without a time, such as a contractual due date, should be represented separately:

```
{ "dueDate": "2026-09-25" }
```

Do not attach midnight and a time zone to a date-only concept merely because a programming type makes that convenient. Doing so can shift the displayed date across zones.

### Durations

A **duration** is an amount of elapsed time rather than a point on a calendar. Document the unit if a number is used:

```
{ "responseTargetSeconds": 1800 }
```

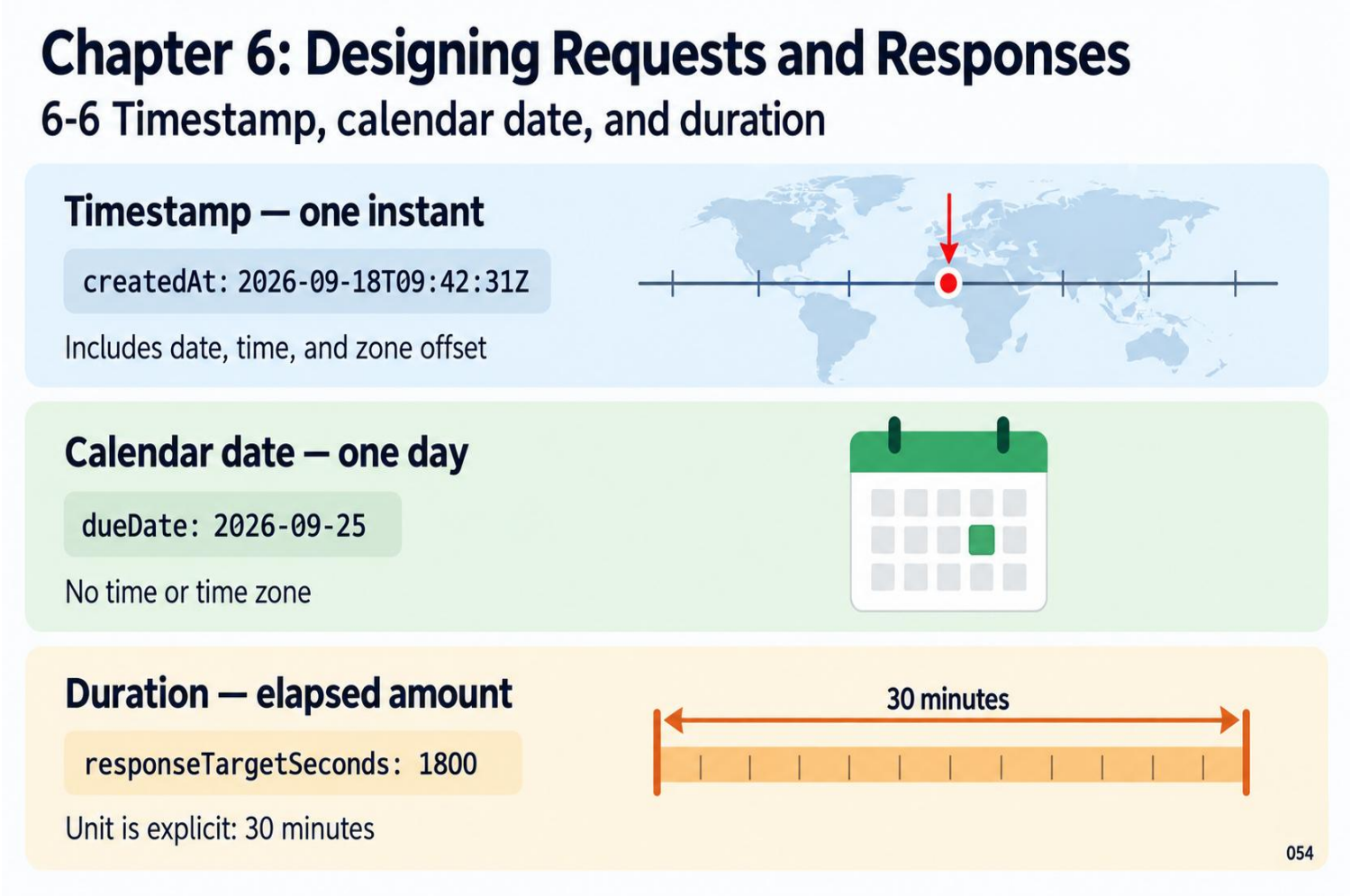



*Figure 6-6: Timestamp, calendar date, and duration*

A field named only `timeout: 30` is ambiguous: seconds, milliseconds, or minutes may be assumed by different consumers.

### Money and exact decimals

Binary floating-point numbers cannot exactly represent many decimal fractions. When exact decimal behavior matters, such as billing, common designs use:

- An integer number of minor units with an explicit currency.
- A decimal string with documented precision and scale.

For example:

```
{
  "amountMinor": 1250,
  "currency": "USD"
}
```

This represents 12.50 US dollars when the currency uses two minor-unit digits. Currency rules vary, so the contract must not assume that every currency has the same scale.

The ticket example does not require money, but systems engineers frequently review APIs that do.

### Enumerated values

An **enumeration** is a field restricted to a documented set of named values. Ticket priority might allow:

```
low, normal, high, urgent
```

Use meaningful names rather than unexplained numbers. The contract should define each value's meaning and capitalization. Consumers should not invent additional values, and providers should recognize that adding a value can affect clients that assume the set is closed.

## Make the creation and retrieval contracts distinct

A compact creation contract might define:

```
{
  "subject": "Printer is offline",
  "description": "The third-floor printer shows error E42.",
  "priority": "normal"
}
```

The successful response might be:

```
{
  "id": "tkt_59302",
  "subject": "Printer is offline",
  "description": "The third-floor printer shows error E42.",
  "status": "open",
  "priority": "normal",
  "requesterId": "usr_204",
  "assigneeId": null,
  "createdAt": "2026-09-19T01:15:44Z",
  "updatedAt": "2026-09-19T01:15:44Z"
}
```

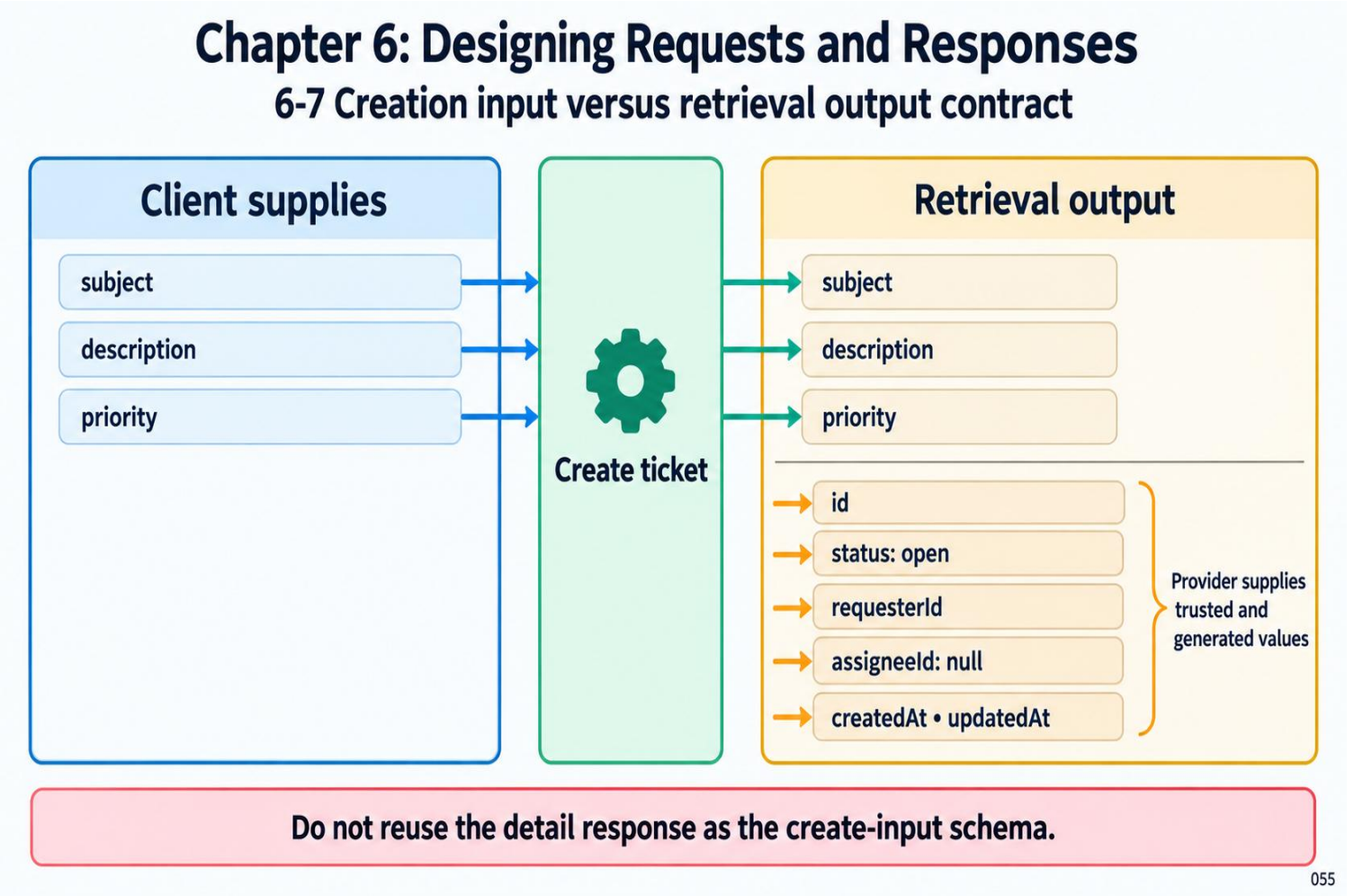



*Figure 6-7: Creation input versus retrieval output contract*

Important decisions include:

- `subject` and `description` are client-supplied.
- `priority` is client-supplied but restricted to allowed values.
- `id`, `status`, `requesterId`, and timestamps are controlled by the server.
- `assigneeId` is explicitly `null` because the ticket is known to be unassigned.
- The response does not expose internal routing flags, database versions, or private notes.

A different API might omit `assigneeId` when unassigned. Either design can work if its meaning is consistent and documented.

## Validate at several levels

Input can be valid at one level and invalid at another.

| Validation level | Question | Example failure |
|---|---|---|
| Syntax | Can the message be parsed? | Malformed JSON. |
| Schema | Do fields, types, and basic constraints match? | `priority` is an array instead of a string. |
| Domain rule | Is the requested business state valid? | A closed ticket cannot be reassigned. |
| Authorization | May this caller perform this action on this target? | An agent tries to change a ticket in another queue. |

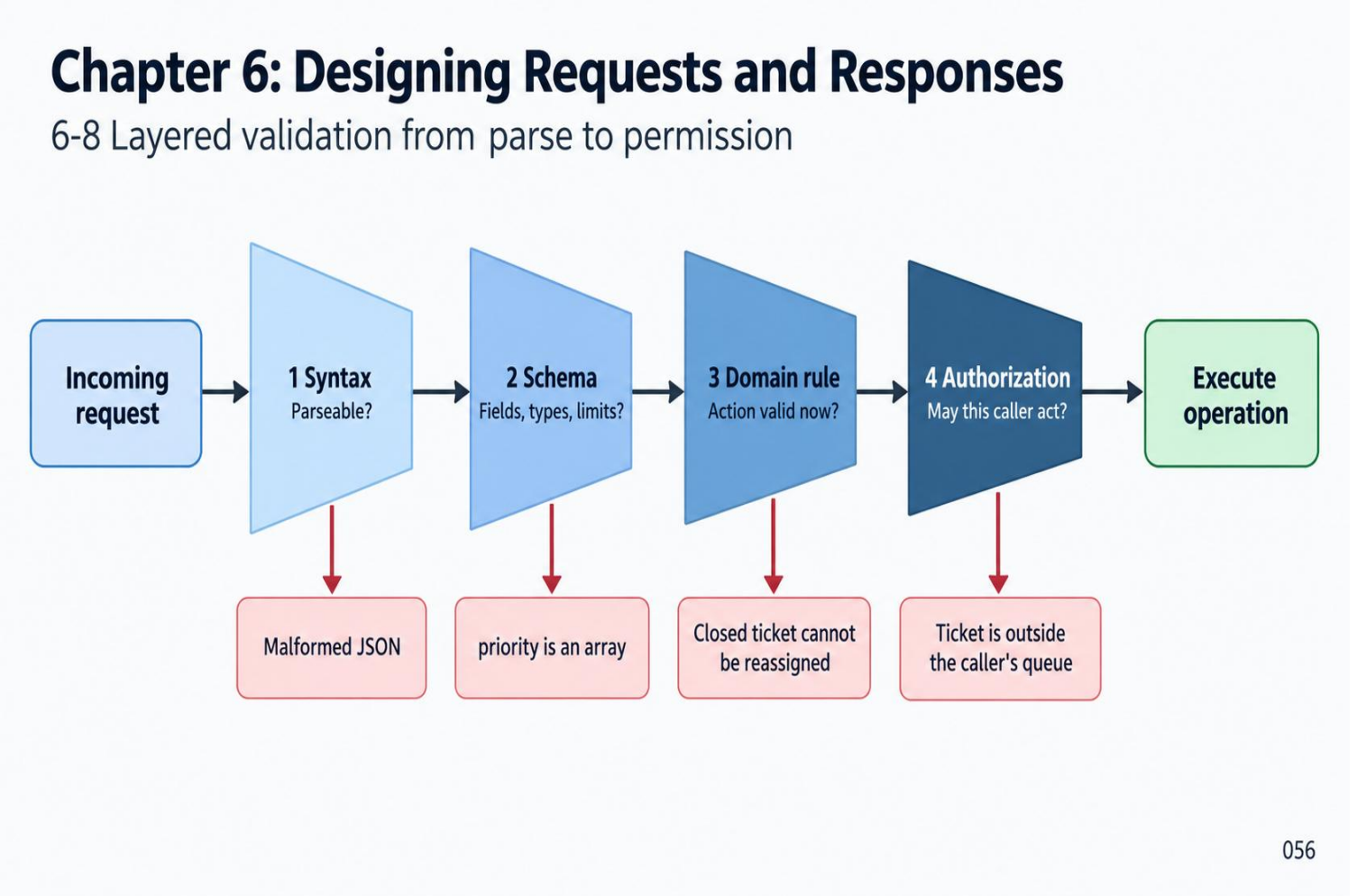



*Figure 6-8: Layered validation from parse to permission*

The order is implementation-dependent, and security considerations can affect which failure is exposed. From a contract perspective, consumers need stable outcomes for the distinctions that matter to their recovery logic.

Useful basic constraints include:

- Required fields.
- String length limits.
- Allowed enumeration values.
- Numeric ranges.
- Array-size limits.
- Format requirements for timestamps or identifiers.
- Cross-field rules, such as requiring `resolutionNote` when status becomes `resolved`.

Limits should reflect actual business and operational constraints. A maximum subject length is useful only when it is documented and consistently enforced.

## Defaults should be observable and stable

A **default** is a value the provider applies when the client omits an optional input. Defaults reduce client work, but they also become part of the contract.

If omitted `priority` defaults to `normal`, changing the default later to `high` can alter behavior for existing clients without any request change. Therefore:

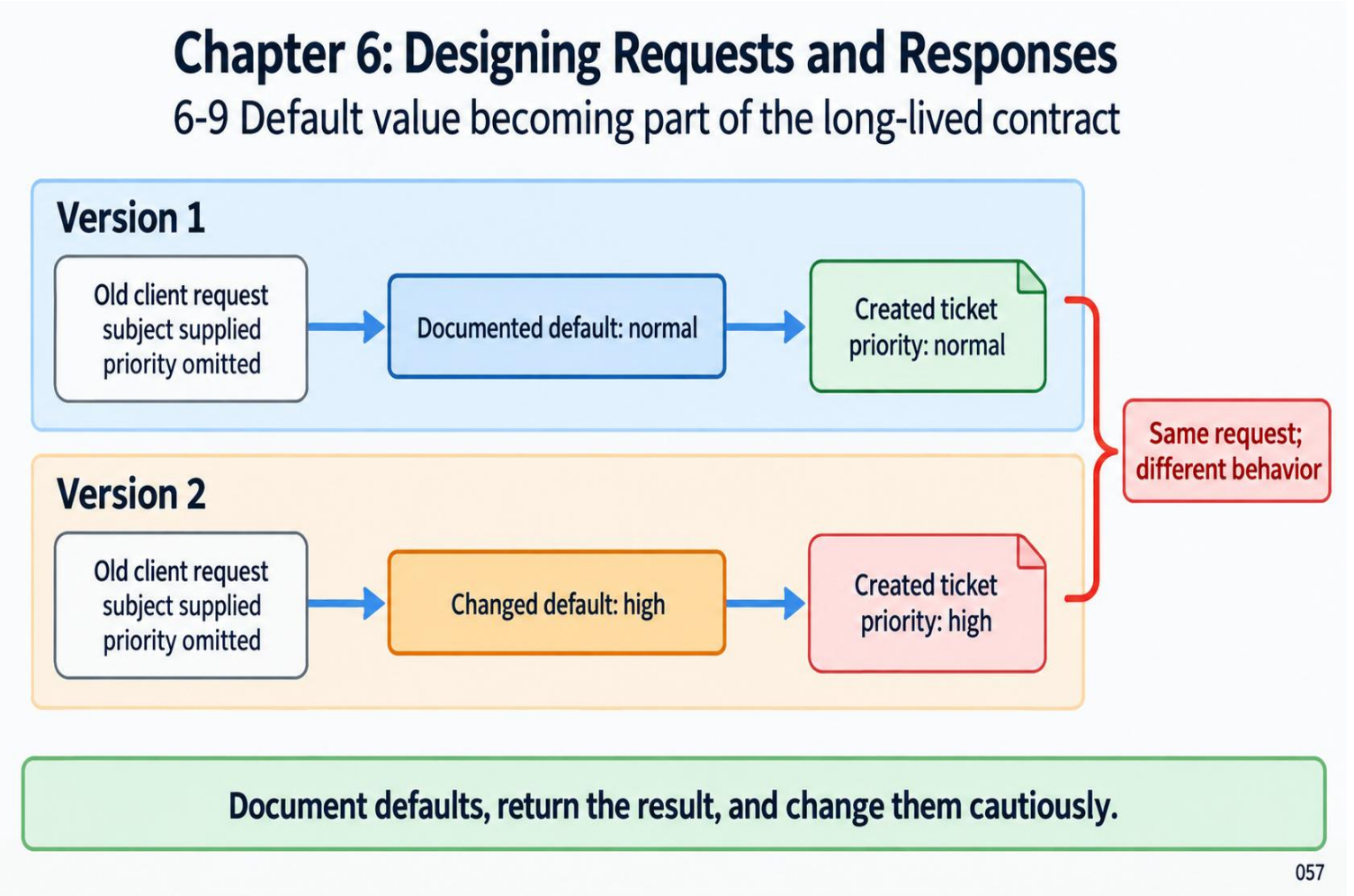



*Figure 6-9: Default value becoming part of the long-lived contract*

- Document the default.
- Return the resulting value.
- Avoid changing it casually.
- Do not use defaults to hide information that clients need to choose deliberately.

Server-generated values such as identifiers and creation timestamps are not quite the same as defaults; they are values owned by the provider.

## Content negotiation at a practical level

**Content negotiation** is the HTTP mechanism by which client and server agree on a representation format. Two headers are central:

- `Content-Type` describes the body being sent.
- `Accept` describes response media types the client can process.

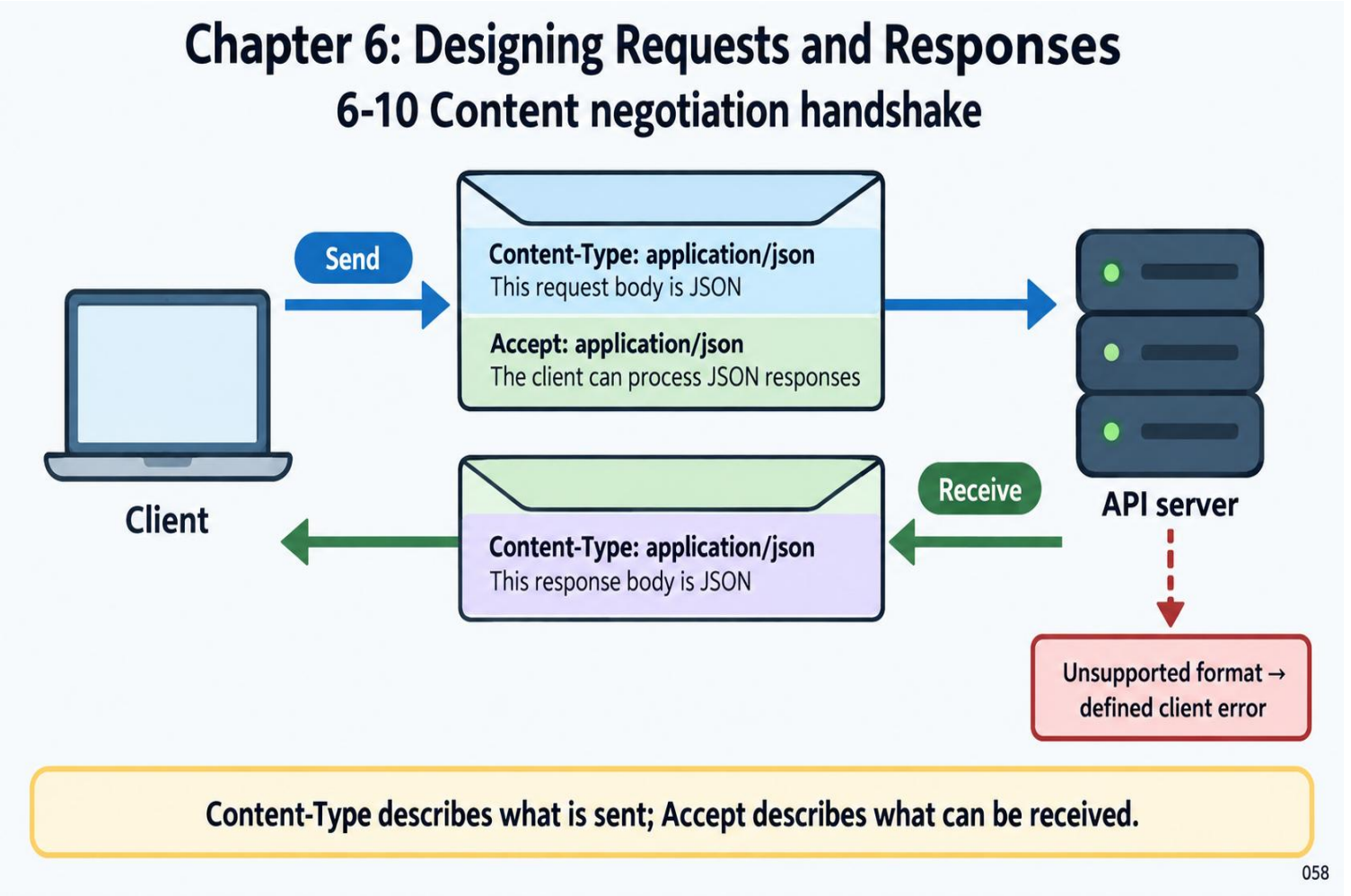



*Figure 6-10: Content negotiation handshake*

For a JSON creation request:

```
Content-Type: application/json
Accept: application/json
```

If the server does not support the request body format, it should return a defined client error rather than attempt to guess. If it cannot produce an acceptable response type, it should communicate that through an appropriate HTTP error.

Many project APIs support only JSON. That is acceptable when explicit. Do not add multiple formats without a real consumer requirement, because each format creates documentation, testing, and compatibility work.

## File transfer is an HTTP-specific design choice

A ticket may have attachments. One practical HTTP design uses `multipart/form-data` for upload and returns attachment metadata after storage. Another design lets a dedicated file-storage service handle file bytes through a temporary upload URI while the ticket API manages metadata.

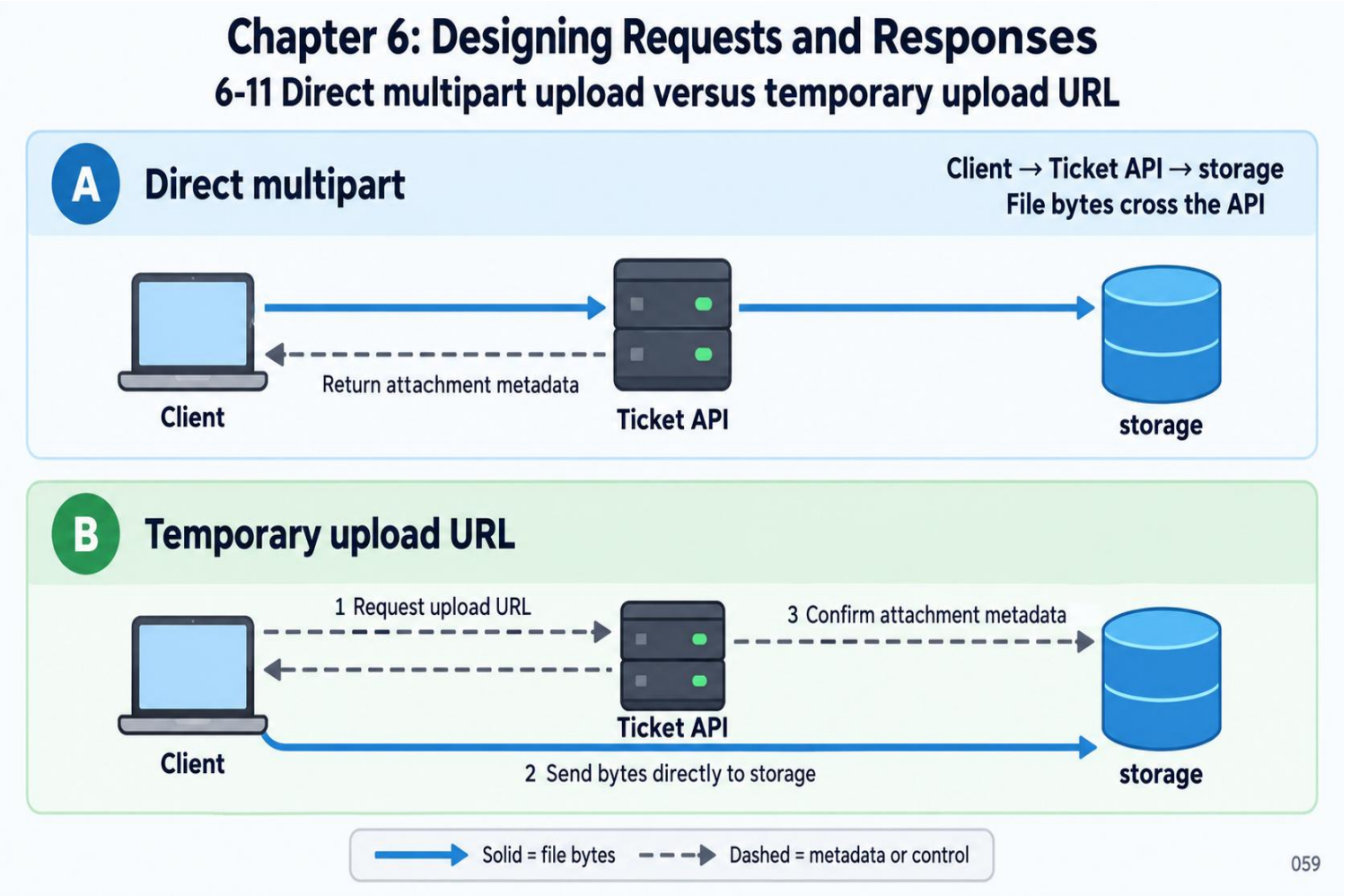



*Figure 6-11: Direct multipart upload versus temporary upload URL*

The contract should define:

- Maximum file size.
- Allowed content types, if restricted.
- Whether uploads are scanned or processed before becoming available.
- How the attachment is associated with the ticket.
- What completion and failure mean.

This is one HTTP-oriented example, not a universal pattern for every API technology. Large-file transfer has security and operational concerns beyond ordinary JSON fields, so it should not be treated as “just another string.”

## Two data-contract mistakes

### “Optional and nullable mean the same thing”

Optional controls whether a field may be absent. Nullable controls whether a present field may contain `null`. Confusing them causes update bugs, especially when `null` means “clear this value” and omission means “leave it unchanged.”

### “The safest response is to expose every stored field”

Exposing more data creates more security risk, payload cost, consumer coupling, and compatibility obligations. Return fields that support the contract. Keep internal notes, security data, implementation flags, and unstable storage details behind the boundary.

## A field-level design check

For every request and response field, answer:

- What does the field mean in domain terms?
- What is its JSON type and any more specific format?

- Is it required, optional, nullable, read-only, or conditionally required?
- What does omission mean?
- What does `null` mean?
- Who owns or generates the value?
- Which values and lengths are allowed?
- Is the field sensitive?
- Can its meaning remain stable as implementation changes?

**A consumer should be able to construct and interpret a message from the contract alone, without testing undocumented guesses against the server.**

## Chapter Summary

- A representation is a contract view of a resource, not the resource itself.
- Field semantics include type, optionality, nullability, mutability, defaults, and whether the client or server controls the value.
- Purpose-specific request and response schemas usually provide a safer contract than exposing one internal model everywhere.

## Review Questions

1. Why are an omitted field and a field set to `null` not automatically equivalent?
2. Which considerations make timestamps, durations, and dates different design problems?
3. Why should a creation request normally exclude server-owned fields such as identifiers and creation times?

# Chapter 7 — Designing Predictable Errors

**An error response is part of the API contract. It should tell a client what kind of failure occurred, whether the request can be corrected, and whether another attempt is sensible.** A human-readable sentence alone is not enough because software needs stable information for control flow.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Use HTTP status codes and stable error codes for different levels of failure meaning.
- Design a predictable machine-readable error body with useful validation and correlation details.
- Distinguish retriable failures from client, permission, business-state, and permanent failures.

## HTTP success and business success must agree

A response with a `2xx` status tells the HTTP client that the request succeeded according to the operation's contract. Therefore, this pattern is misleading:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "success": false,
  "error": "Ticket not found"
}
```

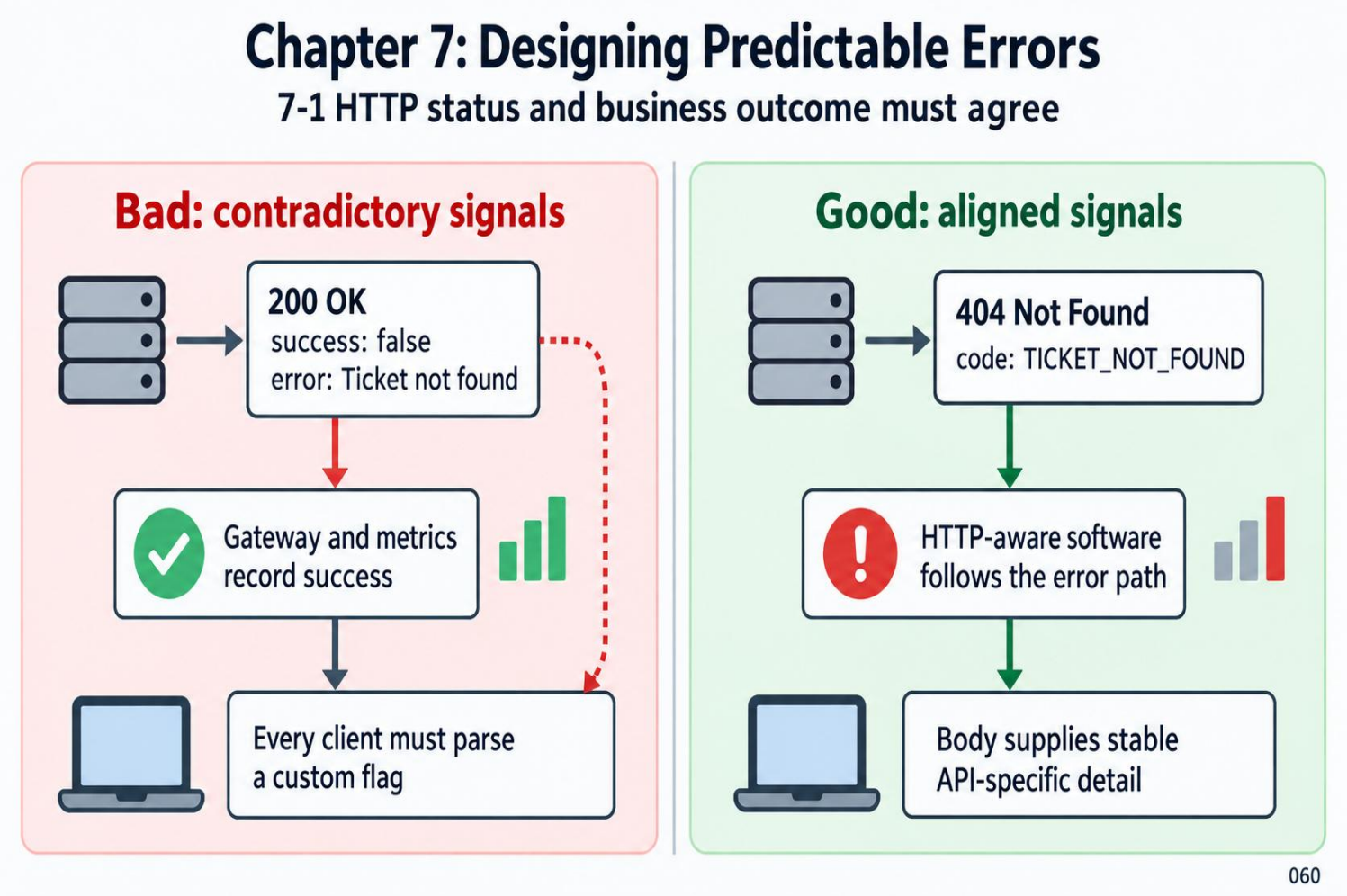



*Figure 7-1: HTTP status and business outcome must agree*

The transport-level signal says success while the body says failure. Generic clients, gateways, metrics, and monitoring may record it as successful. Every consumer must parse a custom flag before it knows the result.

A clearer response uses the status code for the general failure category and the body for API-specific detail:

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/ticket-not-found",
  "title": "Ticket not found",
  "status": 404,
  "code": "TICKET_NOT_FOUND",
  "detail": "No accessible ticket was found for the supplied identifier.",
  "requestId": "req_8f3d92"
}
```

This lets HTTP-aware software recognize the failure and lets ticket-specific client code use the stable `code` value.

## Use status codes for broad categories

The following subset covers many everyday API failures.

| Status | General meaning | Ticket-service example |
|---|---|---|
| 400 Bad Request | The request is malformed or cannot be interpreted as required. | Invalid JSON or a malformed query parameter. |
| 401 Unauthorized | Authentication is missing, invalid, or no longer accepted. | The access token is absent or expired. |
| 403 Forbidden | The caller is known but is not permitted to perform the operation. | An agent cannot access the ticket's queue. |
| 404 Not Found | The addressed resource is absent or is not exposed to this caller. | No accessible ticket matches the identifier. |
| 409 Conflict | The request conflicts with the resource's current business state. | A caller attempts to assign a closed ticket. |
| 412 Precondition Failed | A request condition supplied by the client is no longer true. | The ticket changed after the client read it. |
| 422 Unprocessable Content | The message is structurally understandable but violates defined input rules. | priority contains an unsupported value. |
| 429 Too Many Requests | The caller has exceeded an applicable request limit. | An automation client sends more requests than permitted. |
| 500 Internal Server Error | The provider failed unexpectedly. | An unhandled server defect prevents completion. |
| 503 Service Unavailable | The service cannot currently handle the request. | Temporary overload or an unavailable critical dependency. |

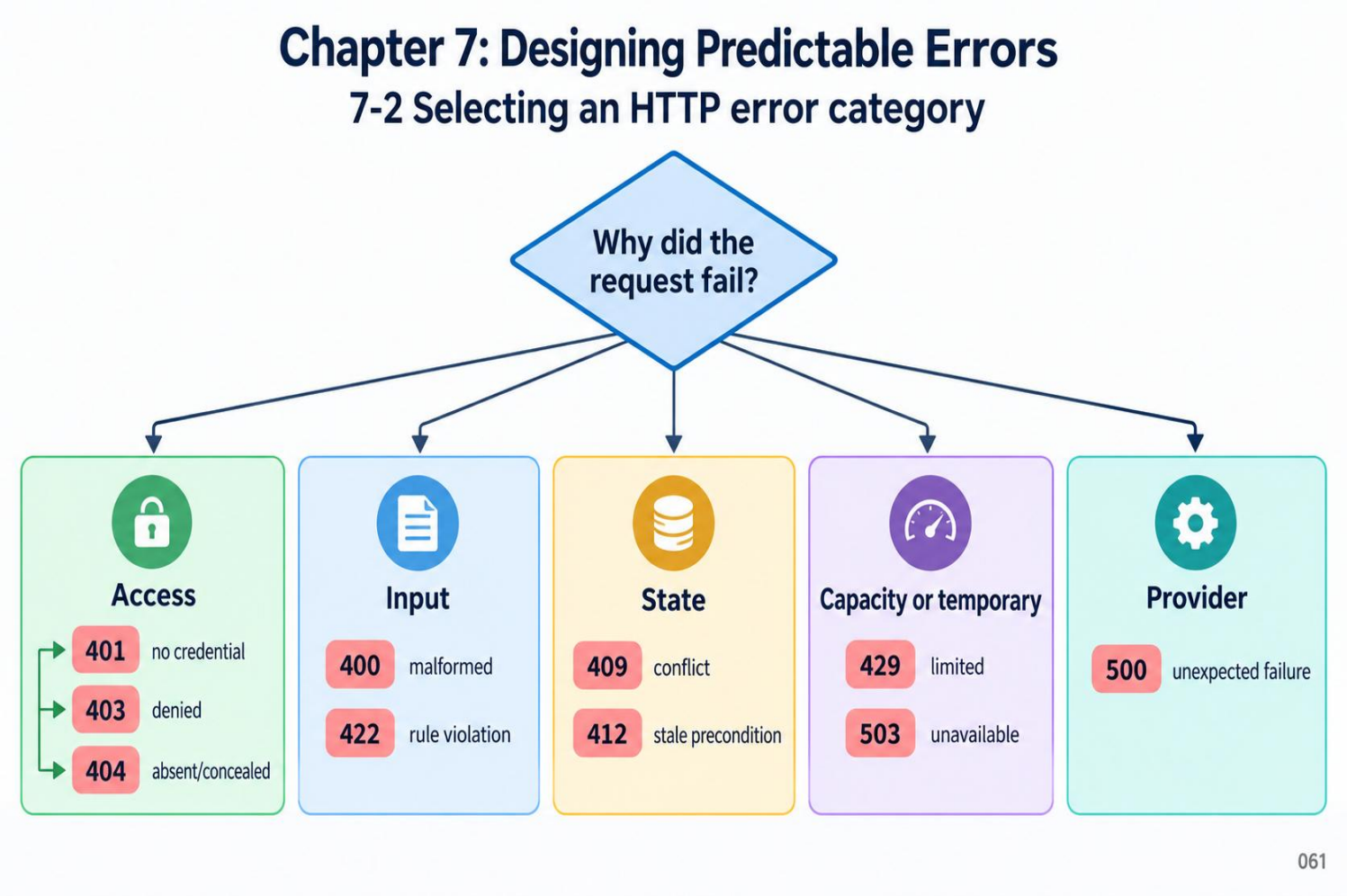



*Figure 7-2: Selecting an HTTP error category*

The exact boundary between `400` and `422`, or between `409` and another client-error status, should be chosen consistently across the API. Consistency matters more than creating an elaborate code for every minor variation.

Some systems deliberately return `404` rather than `403` when revealing that a resource exists would disclose sensitive information. That is a security decision and must be applied consistently; it does not mean the two statuses are generally interchangeable.

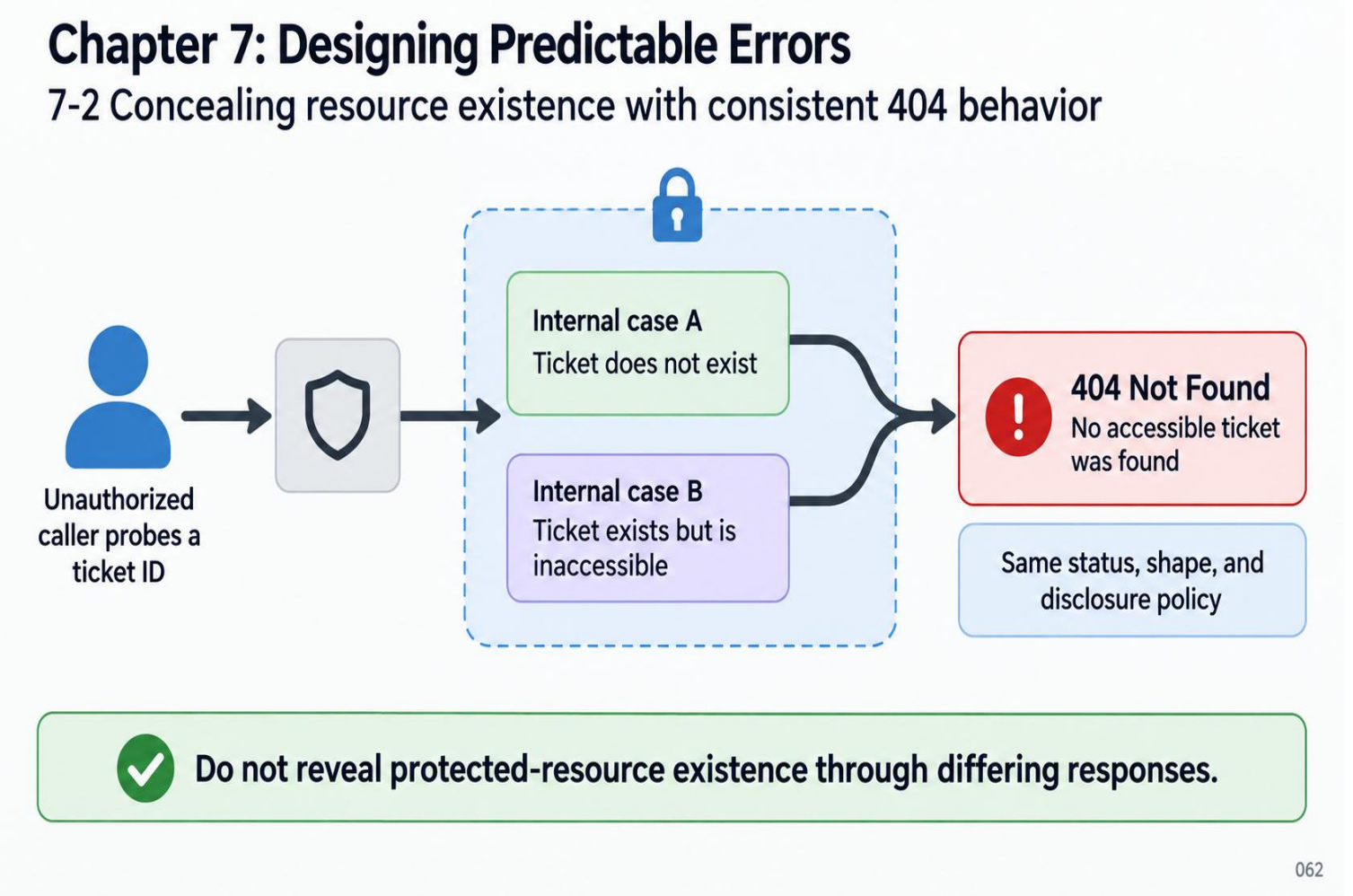



*Figure 7-3: Concealing resource existence with consistent `404` behavior*

## Give errors a stable machine-readable shape

A useful error body usually contains:

- A broad HTTP status.
- A stable API-specific error code.
- A short human-readable title or message.
- Additional details needed to correct the request.
- A request or correlation identifier for support and diagnostics.

**Problem Details** is a standardized JSON structure for describing HTTP API errors. Its common members include `type`, `title`, `status`, `detail`, and `instance`, and an API may add extension members, meaning additional fields such as a stable code or field errors.

A validation response can be:

```
{
  "type": "https://api.example.com/problems/validation-error",
  "title": "Request validation failed",
  "status": 422,
  "code": "TICKET_VALIDATION_FAILED",
  "detail": "One or more fields are invalid.",
  "requestId": "req_d15ae7",
  "errors": [
    {
      "field": "priority",
      "code": "UNSUPPORTED_VALUE",
      "message": "Priority must be one of: low, normal, high, urgent."
    }
  ]
}
```

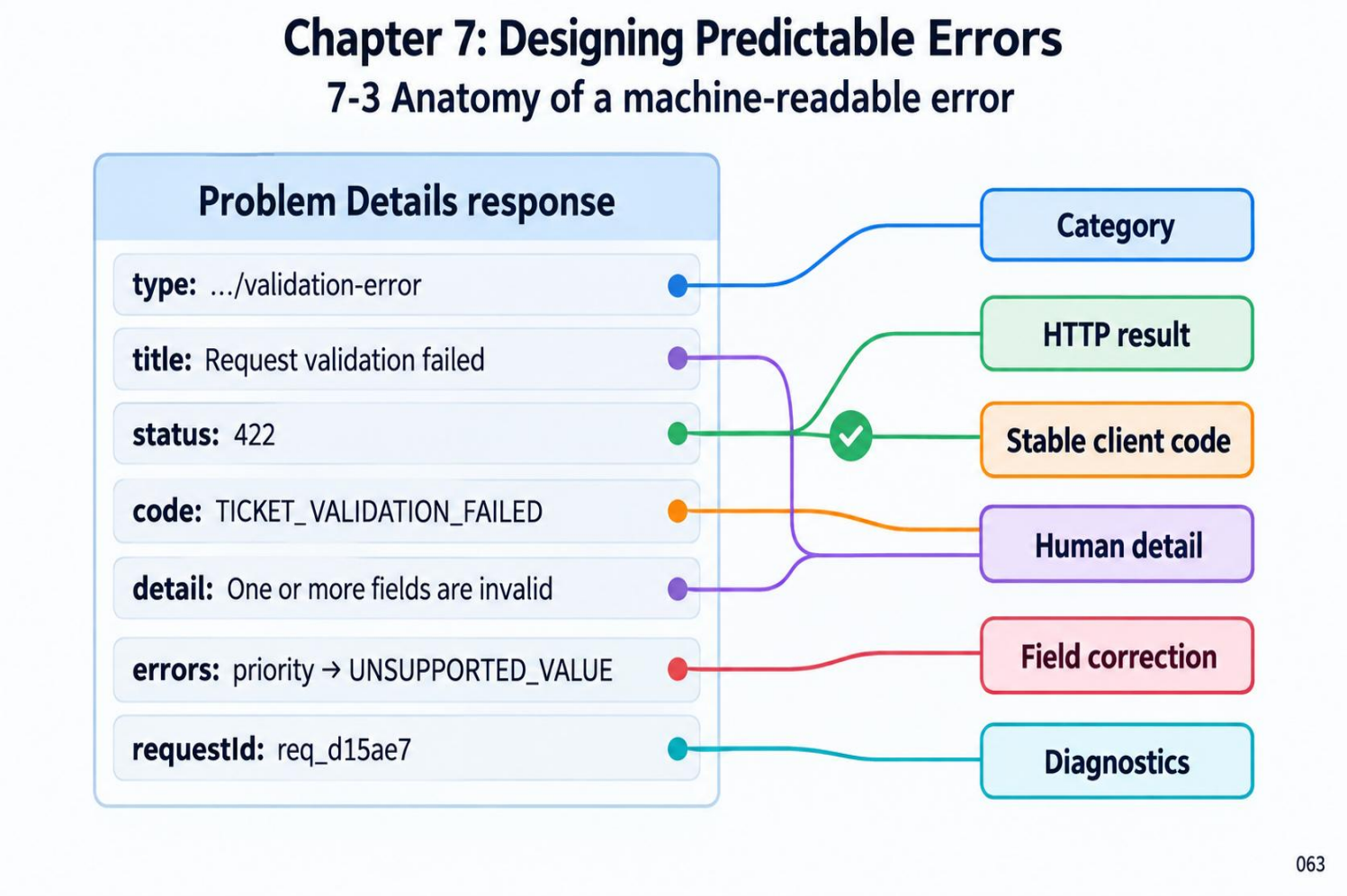



*Figure 7-4: Anatomy of a machine-readable error*

In this design:

- `type` identifies the stable problem category.
- `status` matches the HTTP status.
- `code` is stable enough for application logic.
- `detail` and `message` help people understand the failure.
- `errors` identifies specific invalid fields.
- `requestId` helps operators locate the corresponding server-side records.

The API should define which members are always present and which are optional. Returning several unrelated error shapes forces clients to write special handling for every endpoint.

## Stable codes and human messages have different jobs

A stable error code such as `TICKET_NOT_ASSIGNABLE` is intended for program logic. A message such as “Closed tickets cannot be assigned” is intended for humans and may be reworded or localized.

Client code should not do this:

```
if error.message contains "closed" then show Reopen button
```

It should use defined machine-readable values:

```
if status == 409 and code == "TICKET_NOT_ASSIGNABLE" then
    refresh the ticket and show the allowed next actions
```

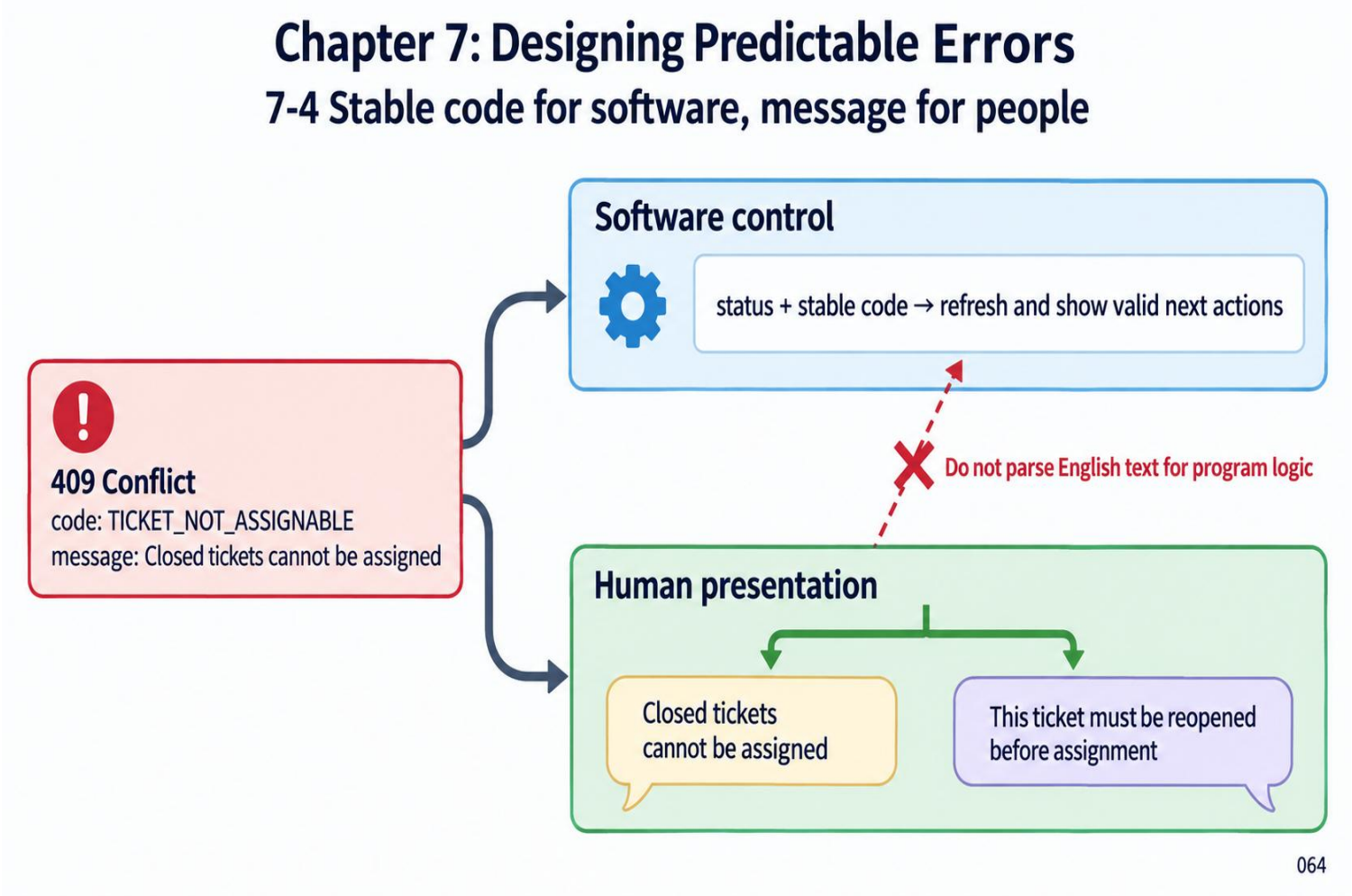



*Figure 7-5: Stable code for software, message for people*

The pseudocode is intentionally minimal. The important point is that prose is not a stable protocol field.

Avoid creating hundreds of codes that merely repeat minor implementation details. Add a code when a consumer needs to distinguish the condition or when the distinction materially improves support and diagnostics.

## Return field-level validation details

When a request contains several independent invalid fields, reporting them together can reduce correction cycles. For example:

```
{
  "type": "https://api.example.com/problems/validation-error",
  "title": "Request validation failed",
  "status": 422,
  "code": "TICKET_VALIDATION_FAILED",
  "errors": [
    {
      "field": "subject",
      "code": "REQUIRED",
      "message": "Subject is required."
    },
    {
      "field": "priority",
      "code": "UNSUPPORTED_VALUE",
      "message": "Priority must be one of: low, normal, high, urgent."
    }
  ],
  "requestId": "req_72c01b"
}
```

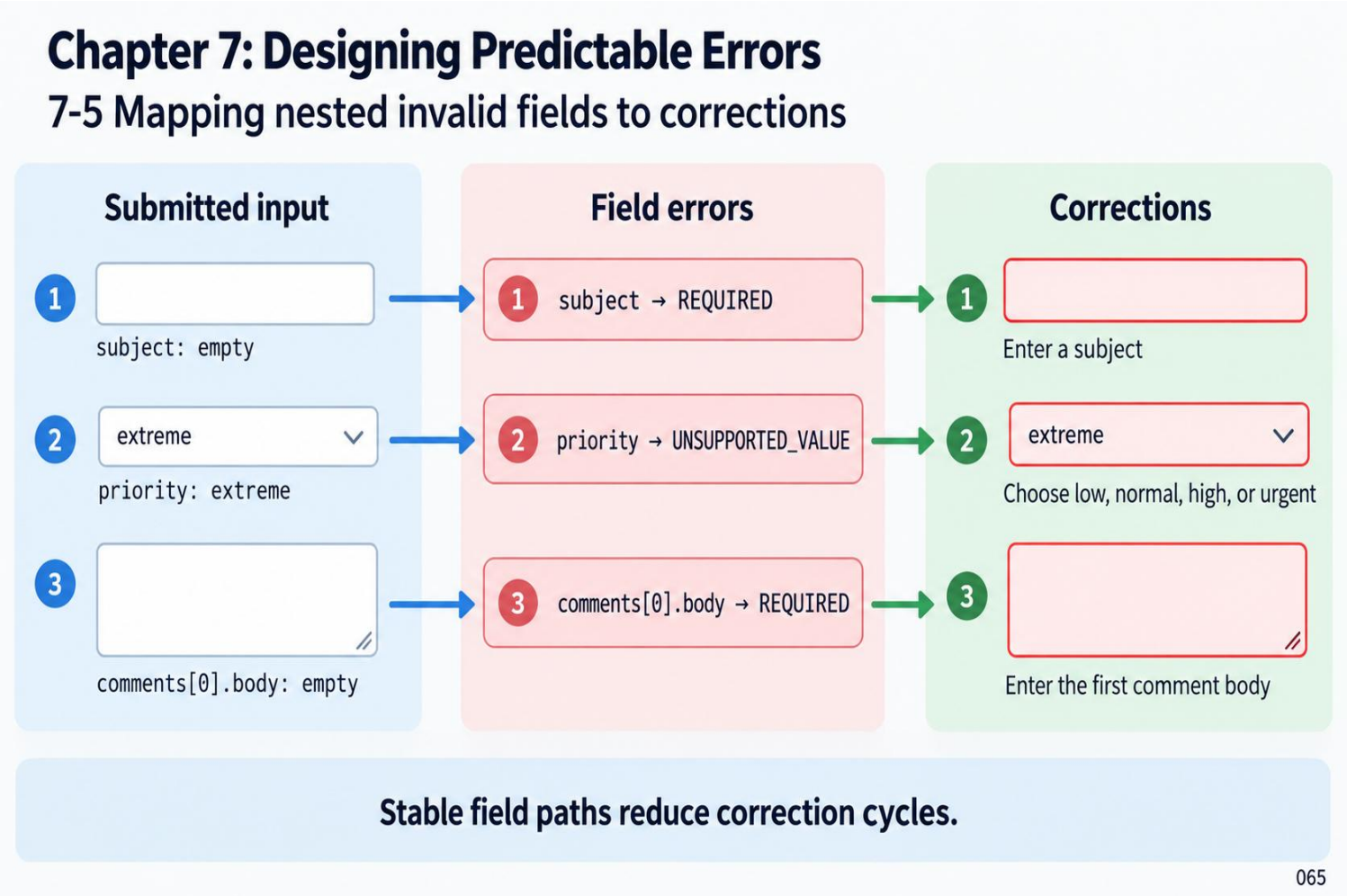



*Figure 7-6: Mapping nested invalid fields to corrections*

The field path convention should support nested objects and arrays if the API uses them. For example, `comments[0].body` is understandable, but another structured path format may be chosen. Use one convention throughout the API.

Do not promise to return every possible validation failure in one response. Some checks depend on earlier checks, current state, or authorization. State whether the server may stop after certain failures.

## Four practical ticket failures

### A missing ticket

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/ticket-not-found",
  "title": "Ticket not found",
  "status": 404,
  "code": "TICKET_NOT_FOUND",
  "detail": "No accessible ticket was found for 'tkt_99999'.",
  "requestId": "req_101a8c"
}
```

The wording “accessible ticket” allows a security policy that does not reveal whether an inaccessible ticket exists.

### An invalid priority

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/validation-error",
  "title": "Request validation failed",
  "status": 422,
  "code": "TICKET_VALIDATION_FAILED",
  "errors": [
    {
      "field": "priority",
      "code": "UNSUPPORTED_VALUE",
      "message": "Priority must be one of: low, normal, high, urgent."
    }
  ],
  "requestId": "req_b91e04"
}
```

The client can correct the field before trying again.

### A forbidden assignment

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/assignment-forbidden",
  "title": "Assignment is not permitted",
  "status": 403,
  "code": "TICKET_ASSIGNMENT_FORBIDDEN",
  "detail": "The caller may not assign tickets in this queue.",
  "requestId": "req_403ddb"
}
```

The response explains the denied capability without exposing internal policy data.

### A conflicting state transition

```
HTTP/1.1 409 Conflict
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/ticket-not-assignable",
  "title": "Ticket cannot be assigned in its current state",
  "status": 409,
  "code": "TICKET_NOT_ASSIGNABLE",
  "detail": "A closed ticket cannot be assigned.",
  "currentStatus": "closed",
  "requestId": "req_ee1297"
}
```

`currentStatus` is useful because the client needs it to decide what to show next. It is safe here because the caller is already permitted to know the ticket state.

## Distinguish retriable and non-retriable failures

A **retriable failure** is one for which repeating the request later may succeed without changing the request's intended meaning. A non-retriable failure requires correction, different authorization, or a different business decision.

General guidance is:

- Malformed input and validation failures should not be retried unchanged.
- Authentication failures require valid credentials before retry.
- Forbidden operations require a permission or target change.
- Missing resources usually require a different identifier or workflow.
- Business conflicts usually require reading current state or choosing another action.
- Limit and temporary-availability failures may be retriable after server guidance.
- Unexpected server failures require caution because the client may not know whether a state-changing operation completed.

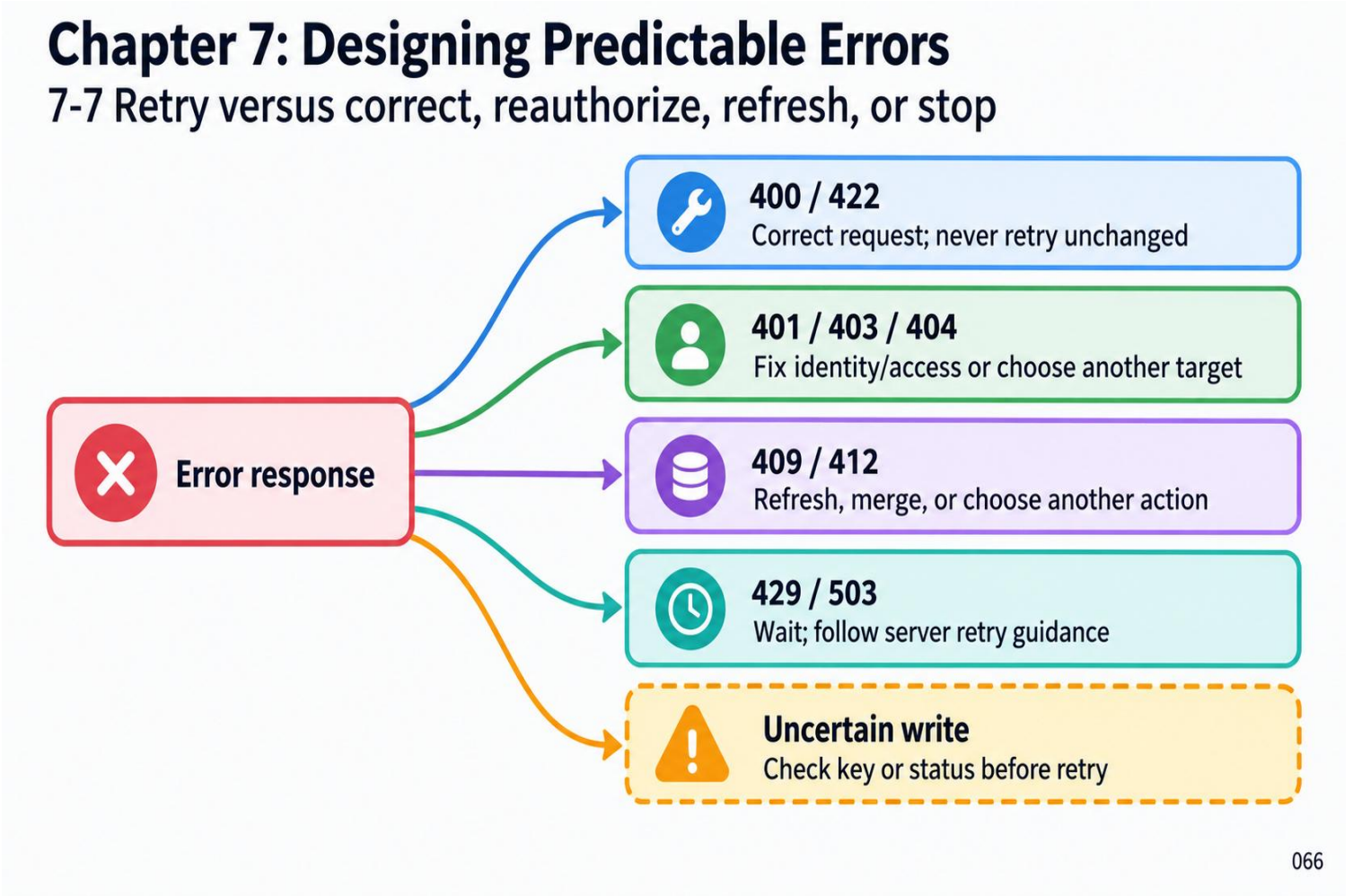



*Figure 7-7: Retry versus correct, reauthorize, refresh, or stop*

The last case is important. If ticket creation returns no usable response because the connection failed, a blind retry might create a duplicate. Retry behavior therefore depends on both the error and the operation's idempotency design.

An API may include retry guidance through defined headers or fields, but it should not mark every server error as safely retriable.

## Correlation identifiers support diagnosis

A **correlation identifier** is a value used to connect a client-observed request with related server-side logs and traces. The error examples use `requestId`.

A useful identifier should be:

- Safe to show to the caller.
- Present in server diagnostics.
- Unique enough for the provider to locate the request.
- Accepted in support reports without revealing infrastructure details.

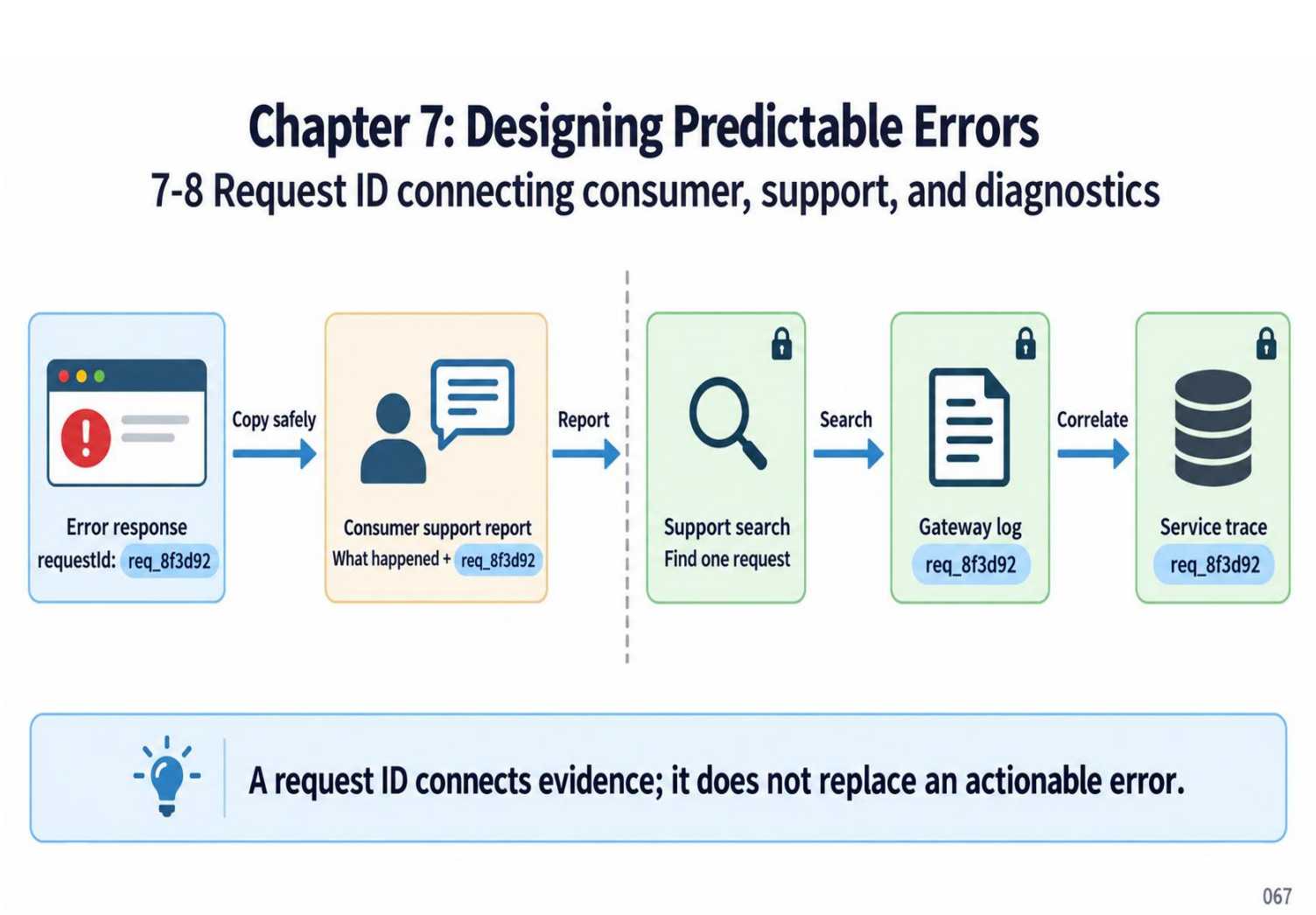



*Figure 7-8: Request ID connecting consumer, support, and diagnostics*

The identifier does not replace a meaningful error. “Contact support with request ID” is inadequate when the client can correct a missing field immediately.

## Do not leak sensitive internals

Error responses should provide enough information for the consumer to act. A **stack trace** is a diagnostic record of the active code calls around a failure. Public error responses should not expose:

- Stack traces.
- Database query statements or table names.
- Internal hostnames and file paths.
- Secrets or credential fragments.
- Personal data unrelated to the caller’s task.
- Detailed authorization logic that would aid abuse.

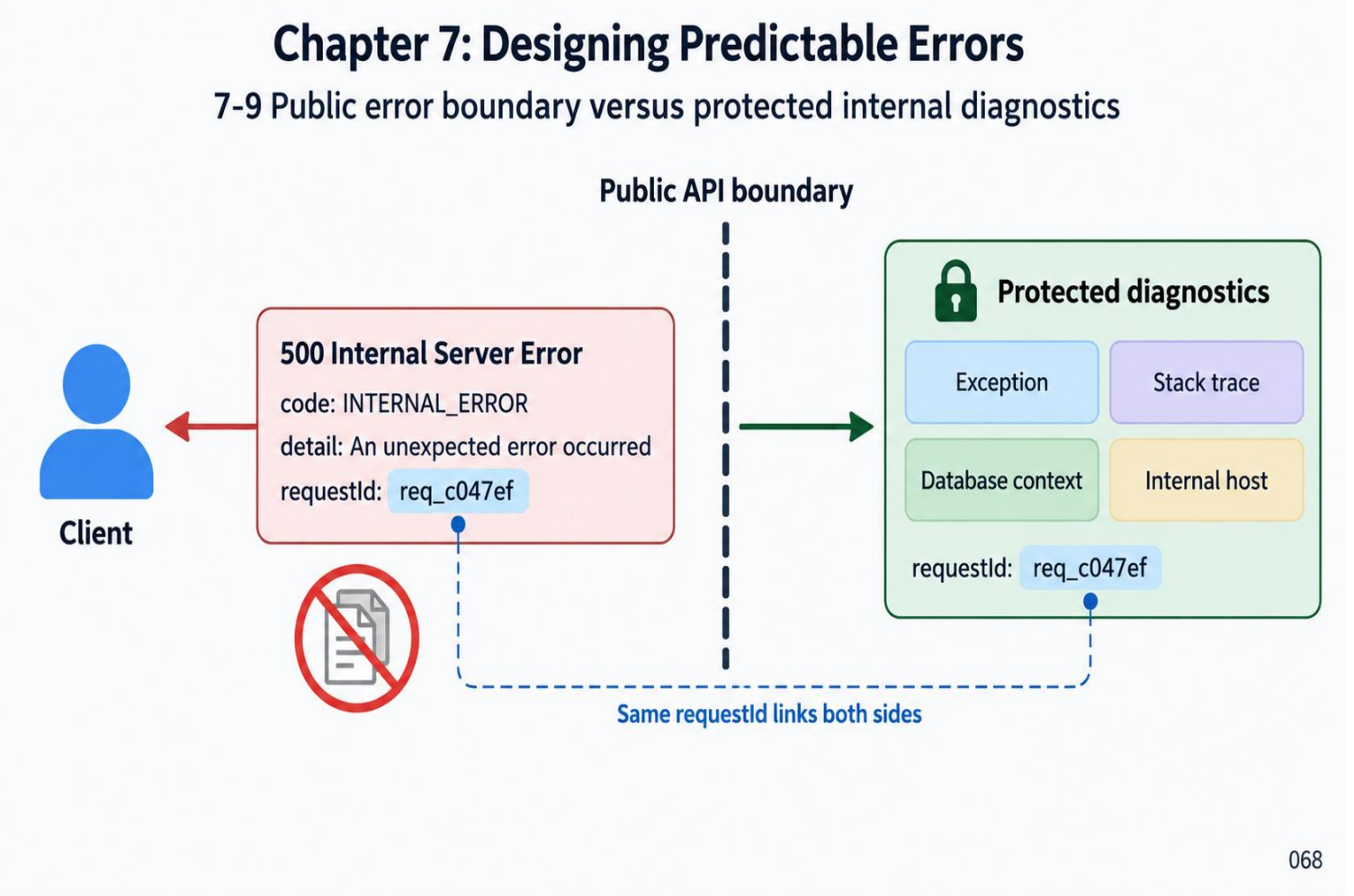



*Figure 7-9: Public error boundary versus protected internal diagnostics*

An internal log may contain richer diagnostic information under controlled access. The public error contract should remain stable, safe, and consumer-oriented.

For an unexpected server failure, this is sufficient:

```
{
  "type": "https://api.example.com/problems/internal-error",
  "title": "The request could not be completed",
  "status": 500,
  "code": "INTERNAL_ERROR",
  "detail": "An unexpected error occurred.",
  "requestId": "req_c047ef"
}
```

The server should log the actual exception with the same request identifier rather than send the exception to the caller.

## Two damaging error patterns

### “Returning `200 OK` with an error flag is simpler for clients”

It is simpler only for a client written specifically around that custom pattern. It makes generic HTTP behavior, monitoring, tooling, and cross-API consistency worse. Use HTTP status for the broad outcome and a structured body for detail.

### “A stack trace helps consumers diagnose the problem”

A stack trace describes the provider’s implementation and may expose exploitable or sensitive information. Consumers need a stable error category, actionable details, and a correlation identifier. Provider engineers need protected internal diagnostics.

## Error-contract review

Before releasing an operation, verify that:

- Every important failure has an appropriate HTTP status.
- Error bodies use one consistent structure.
- Machine-readable codes are stable and documented.
- Human messages are not required for client branching.
- Field errors identify correctable inputs clearly.
- Retry behavior can be determined safely.
- A request identifier connects the response to diagnostics.
- Sensitive implementation and personal data are excluded.
- Business conflicts are distinguishable from malformed input and server failure.

**A predictable error contract turns failure from an exception to the design into a normal, manageable part of integration.**

## Chapter Summary

- The HTTP status and response body must agree about whether the operation succeeded.
- Stable codes support program logic, while human-readable messages support diagnosis and explanation.
- Errors should reveal enough for safe recovery without exposing secrets, stack traces, or internal implementation details.

## Review Questions

1. Why is `200 OK` with an error flag a harmful error pattern?
2. What should remain stable when an error message is reworded?
3. How can a client decide whether retrying a failure is appropriate?

# Chapter 8 — Designing Collection Endpoints

**A collection endpoint must return a bounded, ordered, and predictable window of data.** Retrieving every matching resource in one response does not scale, and adding pagination after consumers depend on an unbounded shape can become a breaking contract change.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Design a stable collection response with filtering, sorting, search, and pagination semantics.
- Choose between offset and cursor pagination based on data volatility and access needs.
- Define deterministic ordering, page-size limits, continuation behavior, and count expectations.

## Give collections a stable response shape

A collection response should clearly separate resource items from pagination metadata. One possible shape is:

```
{
  "items": [
    {
      "id": "tkt_59302",
      "subject": "Printer is offline",
      "status": "open",
      "createdAt": "2026-09-19T01:15:44Z"
    },
    {
      "id": "tkt_48271",
      "subject": "Unable to reset password",
      "status": "open",
      "createdAt": "2026-09-18T09:42:31Z"
    }
  ],
  "page": {
    "limit": 2,
    "nextCursor": "eyJjcmVhdGVkQXQiOiIyMDI2LTA5LTE4VDA5OjQyOjMxWiIsImlkIjoidGt0XzQ4MjcxIn0",
    "hasMore": true
  }
}
```

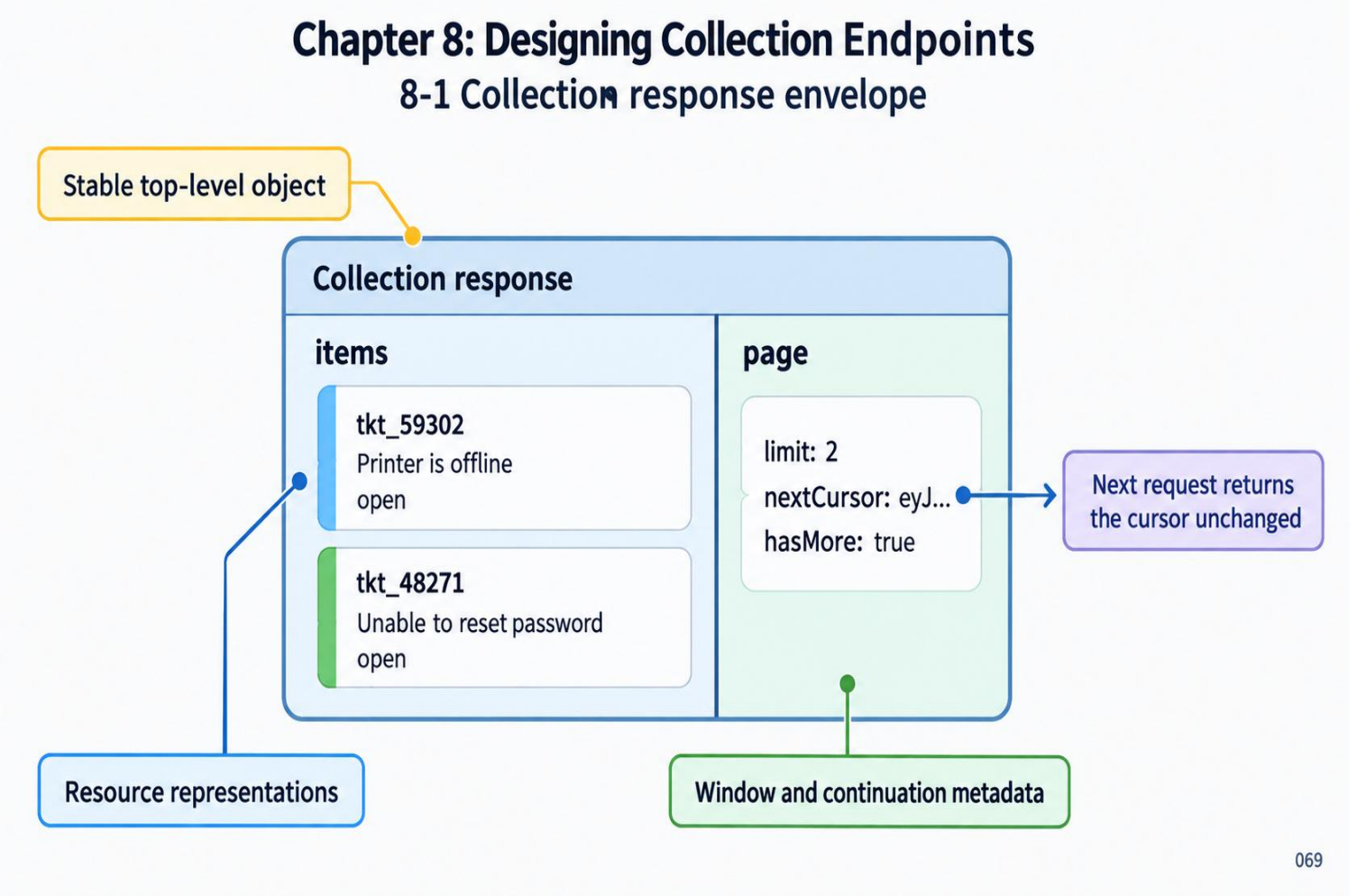



*Figure 8-1: Collection response envelope*

The `items` array holds ticket representations. The `page` object describes this result window and how to continue.

Returning a bare array is possible, but an envelope object makes it easier to add pagination links or metadata without changing the top-level type later.

## Pagination is a contract decision

**Pagination** is the division of a potentially large result set into bounded pages. It protects the provider from unbounded queries and the consumer from unbounded transfer and memory use.

Without pagination, `GET /tickets` may work during development with 200 records and fail in production with 20 million. Common consequences include:

- Slow database queries.
- Large memory use.
- Gateway or client timeouts.
- Excessive network transfer.
- Responses too large for client processing.
- One expensive request affecting other users.

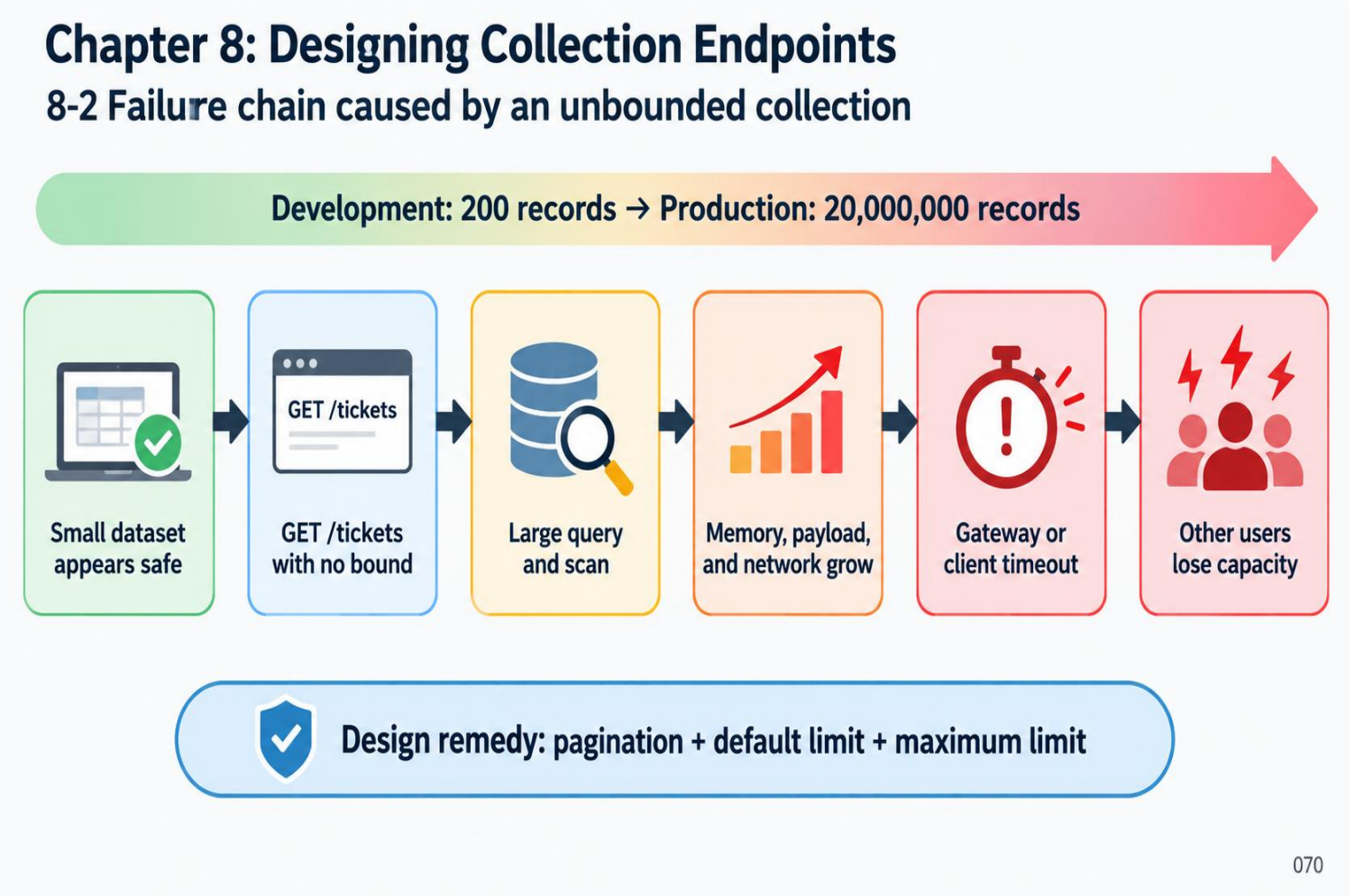



*Figure 8-2: Failure chain caused by an unbounded collection*

Design pagination before the dataset becomes large. Even if the initial number of tickets is small, the response can already use a documented default limit.

## Offset and cursor pagination solve different problems

Two common approaches are offset-based and cursor-based pagination.

| Aspect | Offset-based | Cursor-based |
|---|---|---|
| Example | `?limit=25&offset=50` | `?limit=25&cursor=abc...` |
| Client model | Skip a number of items. | Continue after a position represented by a token. |
| Random page access | Easy when page numbers are exposed. | Usually not supported directly. |
| Behavior during inserts and deletes | Items may shift between pages. | Often more stable when based on deterministic ordering. |
| Large-result performance | Large offsets can become expensive in some data stores. | Can often continue efficiently from the last sort values. |
| Implementation complexity | Usually simpler. | Requires careful cursor and ordering design. |

## 8-3 Offset pagination under inserts and deletes

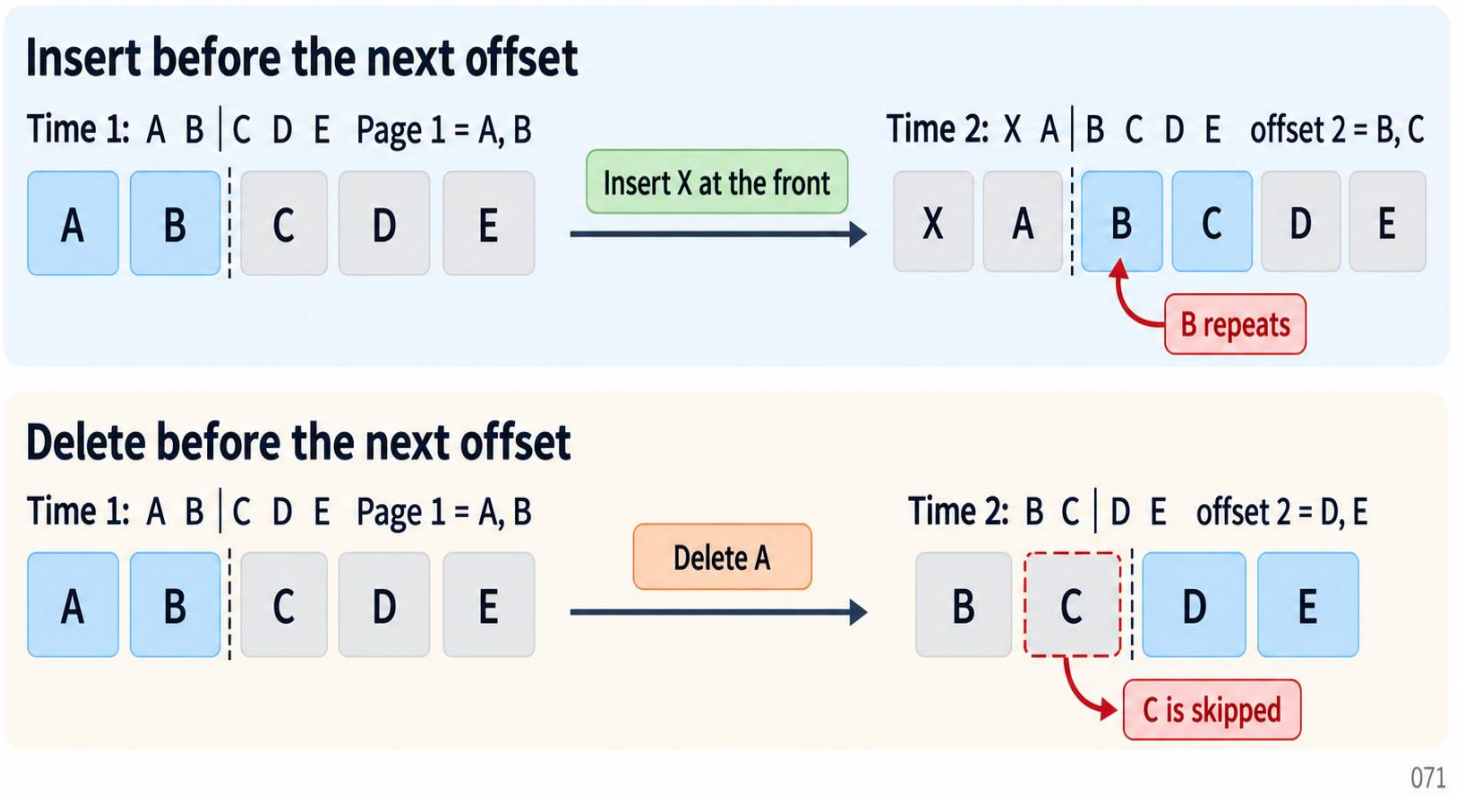





*Figure 8-3: Offset pagination under inserts and deletes*

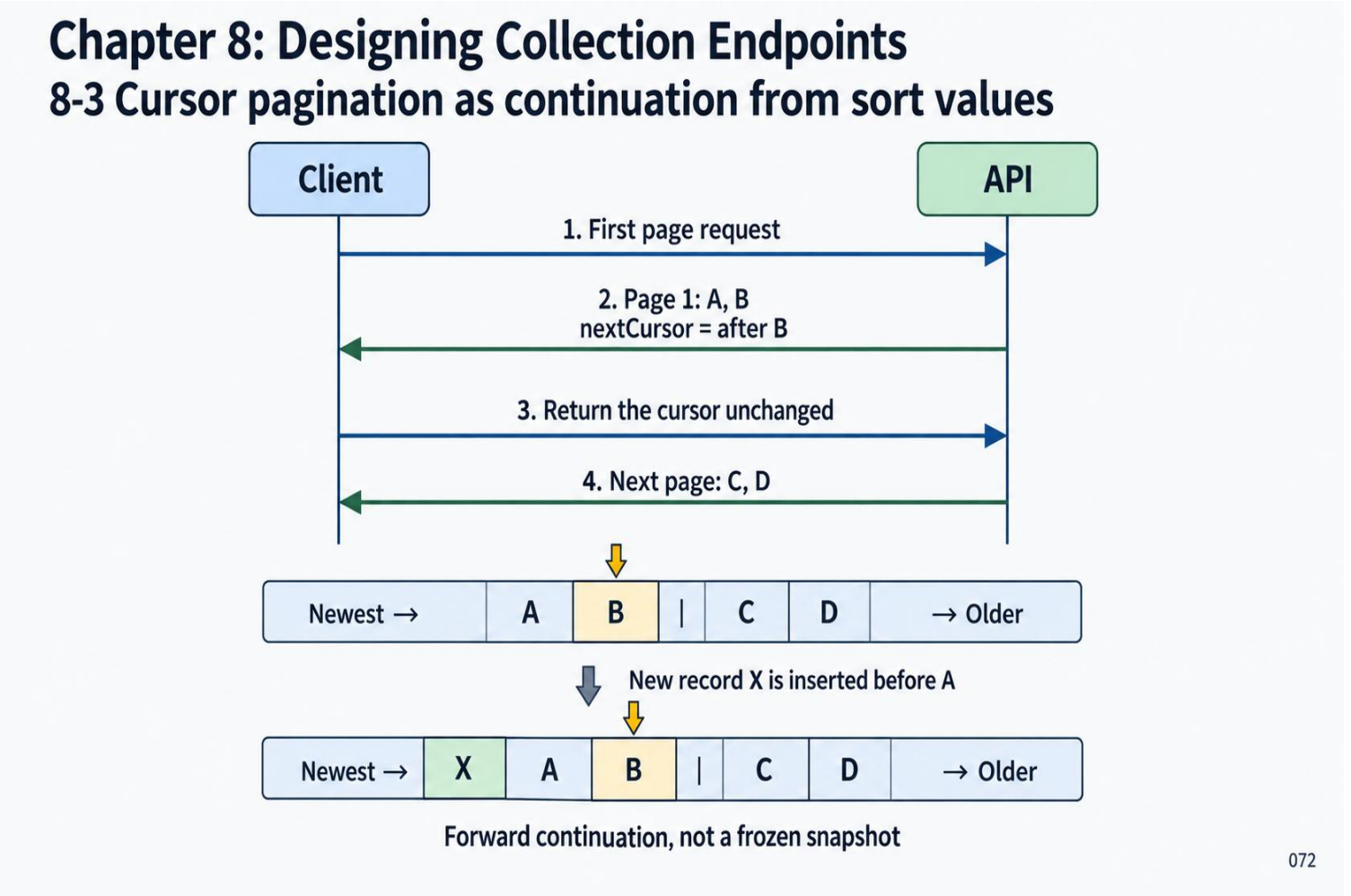



*Figure 8-4: Cursor pagination as continuation from sort values*

Offset pagination can be appropriate for small administrative lists where page-number navigation matters. Cursor pagination is often better for large or frequently changing datasets and for consumers that process results sequentially.

Neither approach automatically creates a frozen snapshot. The API must document what can happen when data changes between requests.

## Treat cursor tokens as opaque

A **cursor** is a token that represents a continuation position in an ordered result set. Clients should return it exactly as received and should not construct, decode, or modify it.

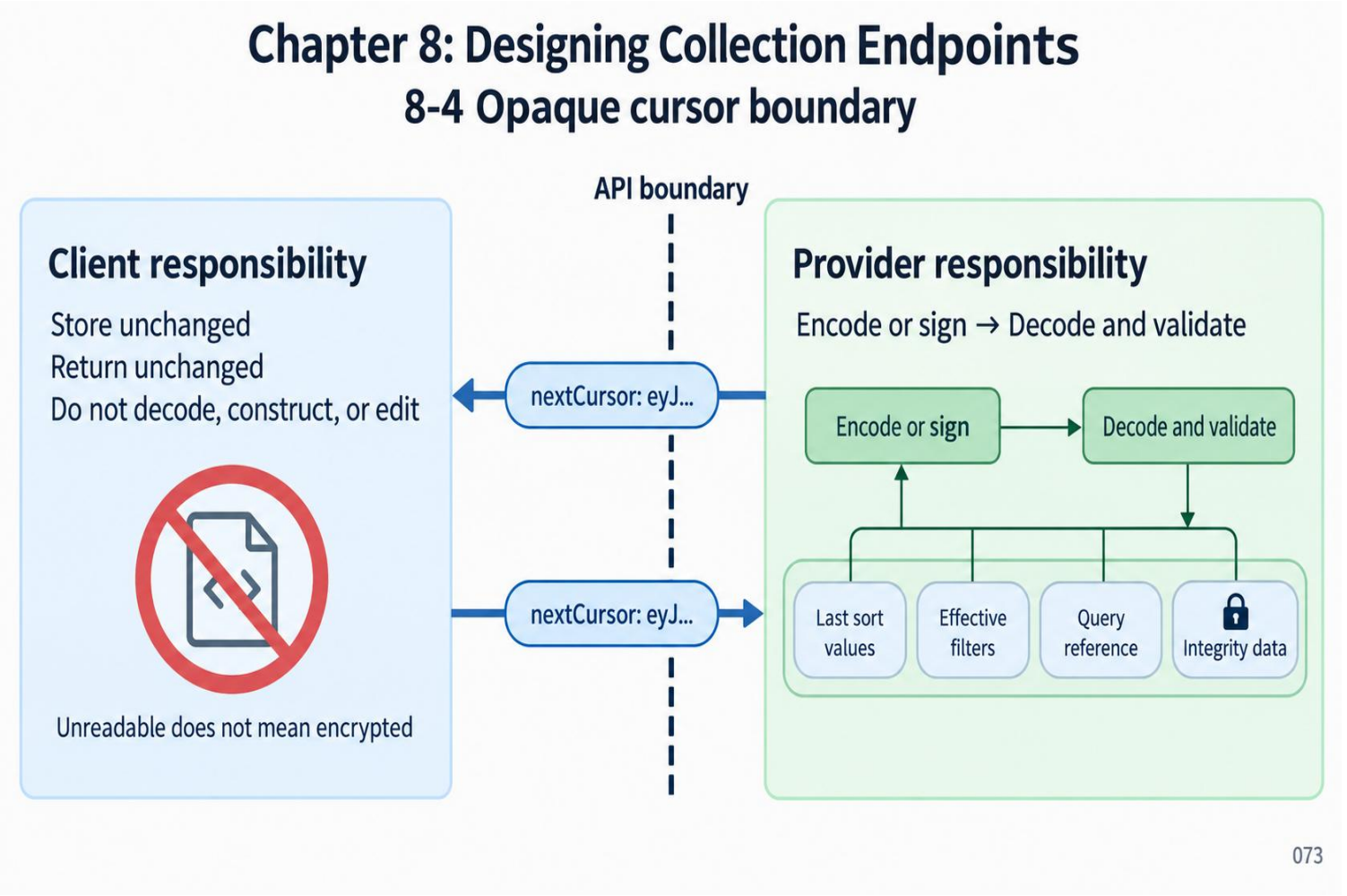



*Figure 8-5: Opaque cursor boundary*

Although the example cursor may encode values internally, that encoding is an implementation detail. The provider may later sign, encrypt, or change the token format. The contract should define the cursor as an opaque string.

A cursor may capture:

- The last item's sort values.
- The effective filter and sort configuration.
- A snapshot or query identifier where supported.
- Integrity information that prevents tampering.

Do not put sensitive data in a cursor merely because the token looks unreadable. Encoding is not necessarily encryption.

## Ordering must be deterministic

Pagination requires a defined order. A sort is **deterministic** when the same current dataset and sort rules produce one unambiguous order.

Sorting only by `createdAt` is insufficient because several tickets can share the same timestamp. Add a unique tie-breaker:

```
createdAt descending, then id descending
```

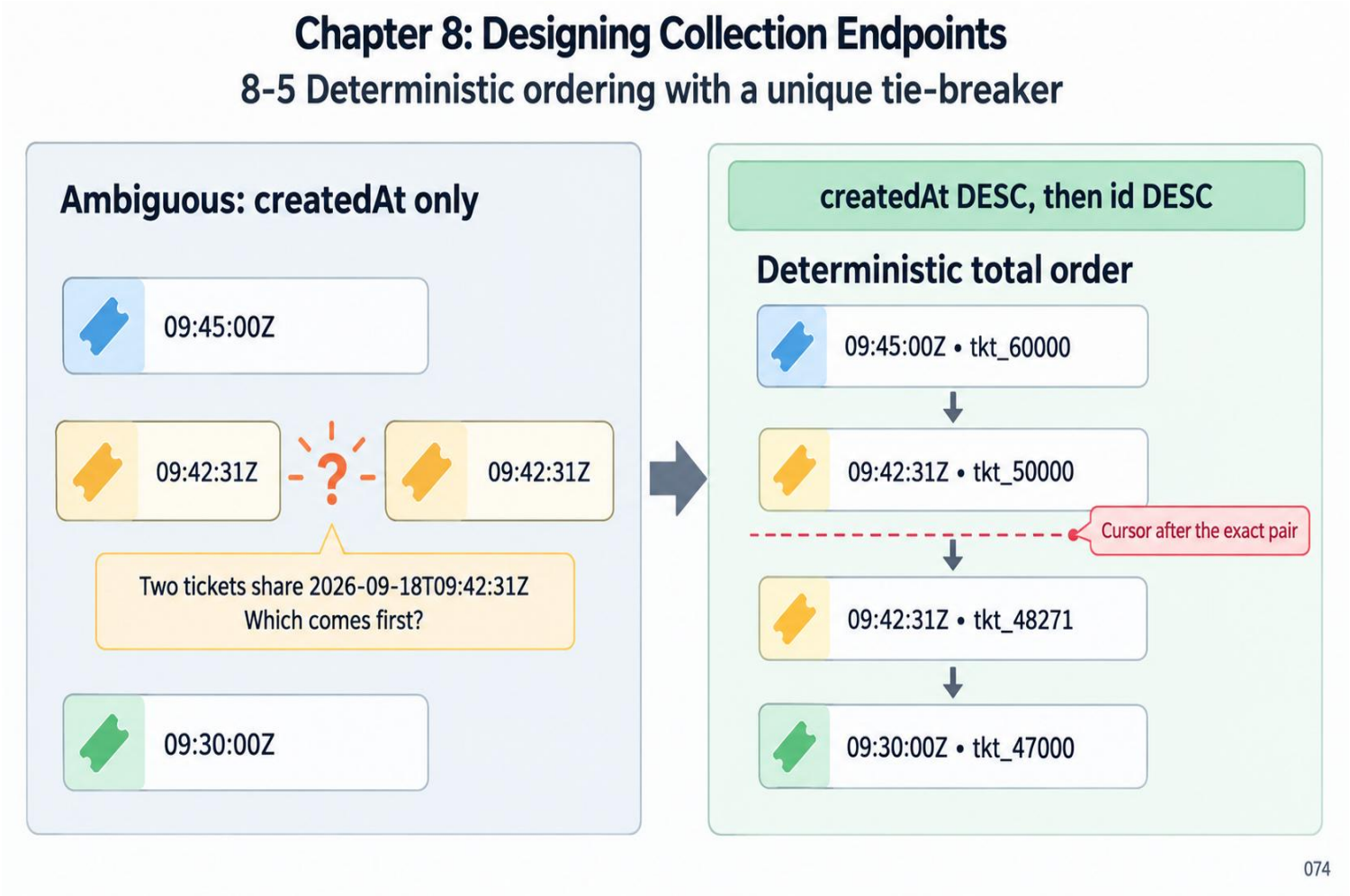



*Figure 8-6: Deterministic ordering with a unique tie-breaker*

Conceptually, the order is:

```
ORDER BY createdAt DESC, id DESC
```

The database syntax is an illustration, not part of the API contract. The contract is the ordering rule.

The cursor can identify the last pair of values returned. The next page then asks for records after that pair in the defined order. Without the tie-breaker, tickets with equal timestamps can appear twice or be skipped.

Default ordering should be documented. Do not rely on a database's accidental row order.

## Filtering, sorting, and search are distinct

These concepts are often combined under "query," but they have different meanings.

| Capability | Purpose | Example |
|---|---|---|
| Filtering | Include resources that match exact or structured conditions. | `status=open&assigneeId=usr_991` |
| Sorting | Choose the order of matching resources. | `sort=-createdAt` |
| Search | Match less structured text or relevance criteria. | `q=password+reset` |

A filter should have defined comparison semantics. For example, `status=open` is an exact match to an allowed status. A search query may split text into words, match related word forms, or rank results by relevance, and those behaviors can evolve without guaranteeing exact matches.

Do not call every string parameter "search." Consumers need to know whether results are exact, partial, case-sensitive, language-dependent, or relevance-ranked.

## Define query-parameter conventions

A ticket collection request might be:

```
GET /tickets?status=open&assigneeId=usr_991&sort=-createdAt&limit=50
```

One possible convention is:

- `status=open` filters to open tickets.
- `assigneeId=usr_991` filters to the named assignee.
- `sort=-createdAt` sorts by creation time descending; a missing minus sign means ascending.
- `limit=50` asks for at most 50 items.

Other conventions are valid. The API must define:

- Supported filter fields and operators.
- Whether names and values are case-sensitive.
- How multiple values are expressed.
- Whether multiple filters use AND or OR logic.
- Supported sort fields and directions.
- The default sort.
- Date and timestamp boundary semantics.
- Handling of unknown parameters.

Silently ignoring a misspelled filter such as `stats=open` can produce a dangerously broad result. Rejecting unsupported parameters with a clear client error is often safer.

## Set default and maximum page sizes

A collection should have:

- A **default limit** used when the client does not specify one.
- A **maximum limit** that protects the provider and network.
- Defined behavior when the requested limit exceeds the maximum.

For example, the API might default to 25 and allow at most 100. It can either reject `limit=1000` or reduce it to 100, but the behavior must be documented. Silent capping can be acceptable when the response reports the effective limit.

Choose limits using representation size, query cost, client needs, and response-time goals. Do not copy an arbitrary number from another API without considering the actual data.

## Retrieve a filtered page with `curl`

**Prerequisites:** `curl` is available in a POSIX-style shell, the example service is reachable, and `ACCESS_TOKEN` contains an accepted credential.

**Purpose:** Retrieve up to 25 open tickets assigned to `usr_991`, ordered from newest to oldest.

```
curl --include \
  --get \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --header "Accept: application/json" \
  --data-urlencode "status=open" \
  --data-urlencode "assigneeId=usr_991" \
  --data-urlencode "sort=-createdAt" \
  --data-urlencode "limit=25" \
  "https://api.example.com/tickets"
```

`--get` places the supplied parameters in the **query string**, the portion of the URI after `?`. `--data-urlencode` safely encodes each parameter.

**Expected result:** A `200 OK` response contains an `items` array with no more than 25 matching tickets and continuation metadata when additional items exist. The exact host and authorization method are specific to this example environment.

To retrieve the next page, the client sends the returned cursor unchanged:

```
GET /tickets?status=open&assigneeId=usr_991&sort=-createdAt&limit=25&cursor=<returned-cursor>
```

The client should preserve the filters and sort used for the first page unless the cursor contract explicitly includes and enforces them.

## Continuation links can reduce client mistakes

Instead of returning only a cursor, an API may return a continuation URI:

```
{
  "items": [],
  "links": {
    "next": "https://api.example.com/tickets?status=open&sort=-createdAt&limit=25&cursor=abc123"
  }
}
```

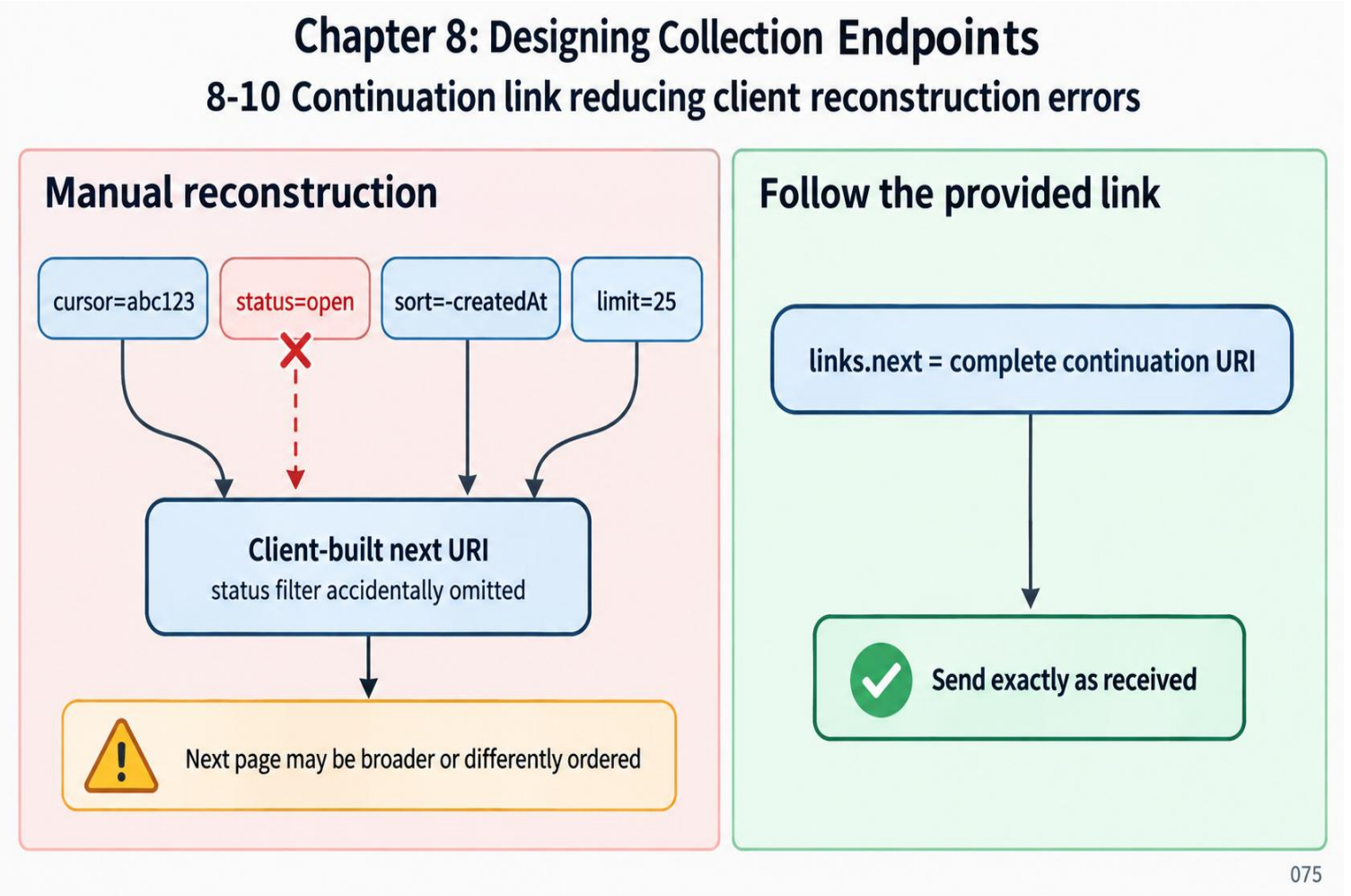



*Figure 8-7: Continuation link reducing client reconstruction errors*

A link packages the correct continuation parameters. A cursor token gives the client more control over constructing the next request. Either can work. If both are returned, their relationship should be clear.

Do not require clients to calculate the next offset or cursor from the returned items unless that algorithm is intentionally part of the contract.

## Total counts are not free

A field such as `totalCount` appears convenient:

```
{
  "items": [],
  "totalCount": 1842390
}
```

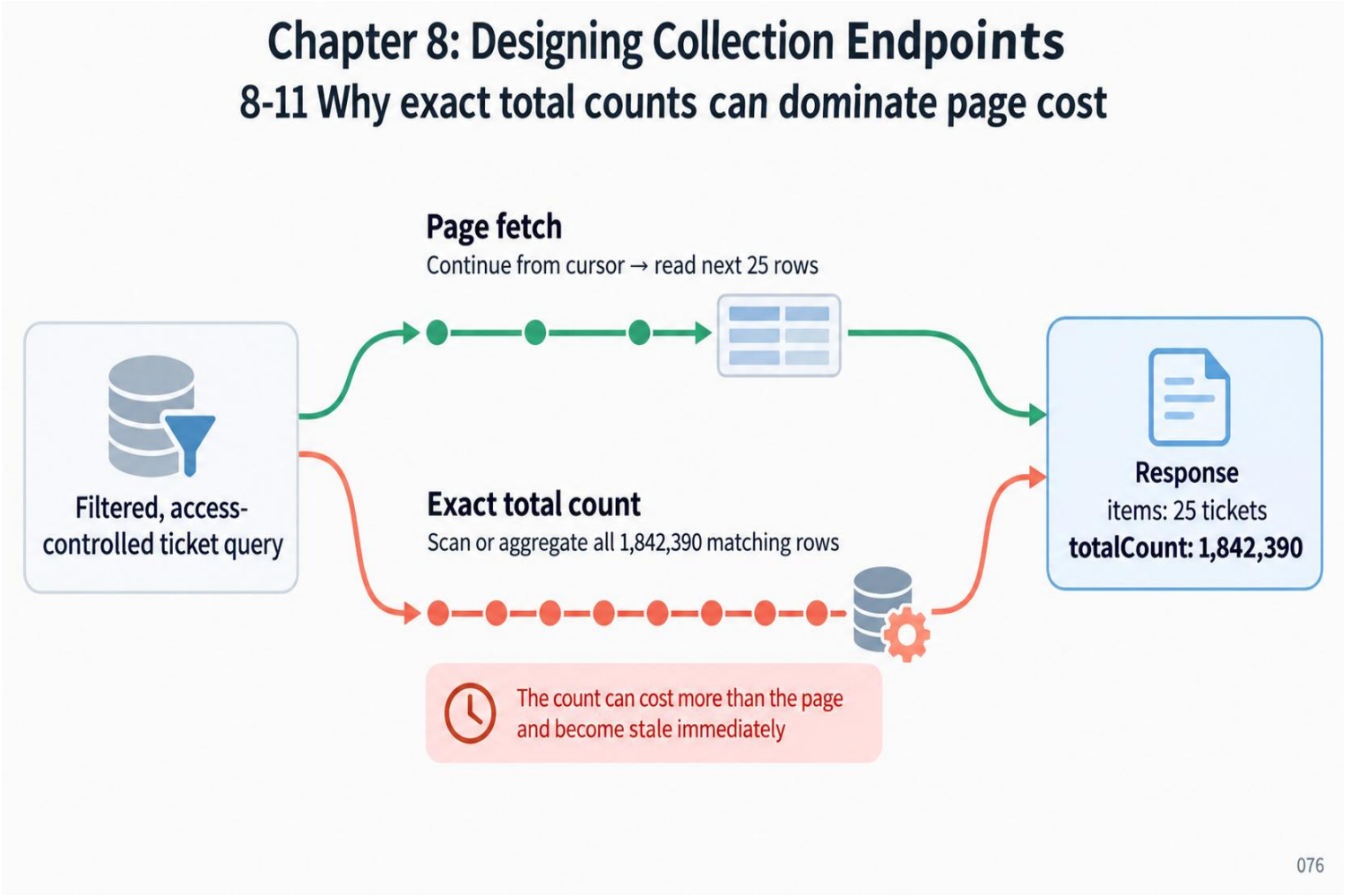



*Figure 8-8: Why exact total counts can dominate page cost*

For large, filtered, distributed, or access-controlled datasets, obtaining an exact count can be more expensive than retrieving one page. The count may also become stale immediately as records change.

Before including it, ask what consumer decision requires it. Alternatives include:

- Omit the total and expose only whether another page exists.
- Return an approximate count and label it clearly.
- Provide a separate count operation for use cases that truly require it.
- Compute an exact total only for bounded or inexpensive queries.

Once `totalCount` is in the contract, clients may depend on its exactness and availability.

## Understand changes between pages

Assume a client retrieves page one, then another user creates or updates tickets before page two is requested. The ordered dataset has changed.

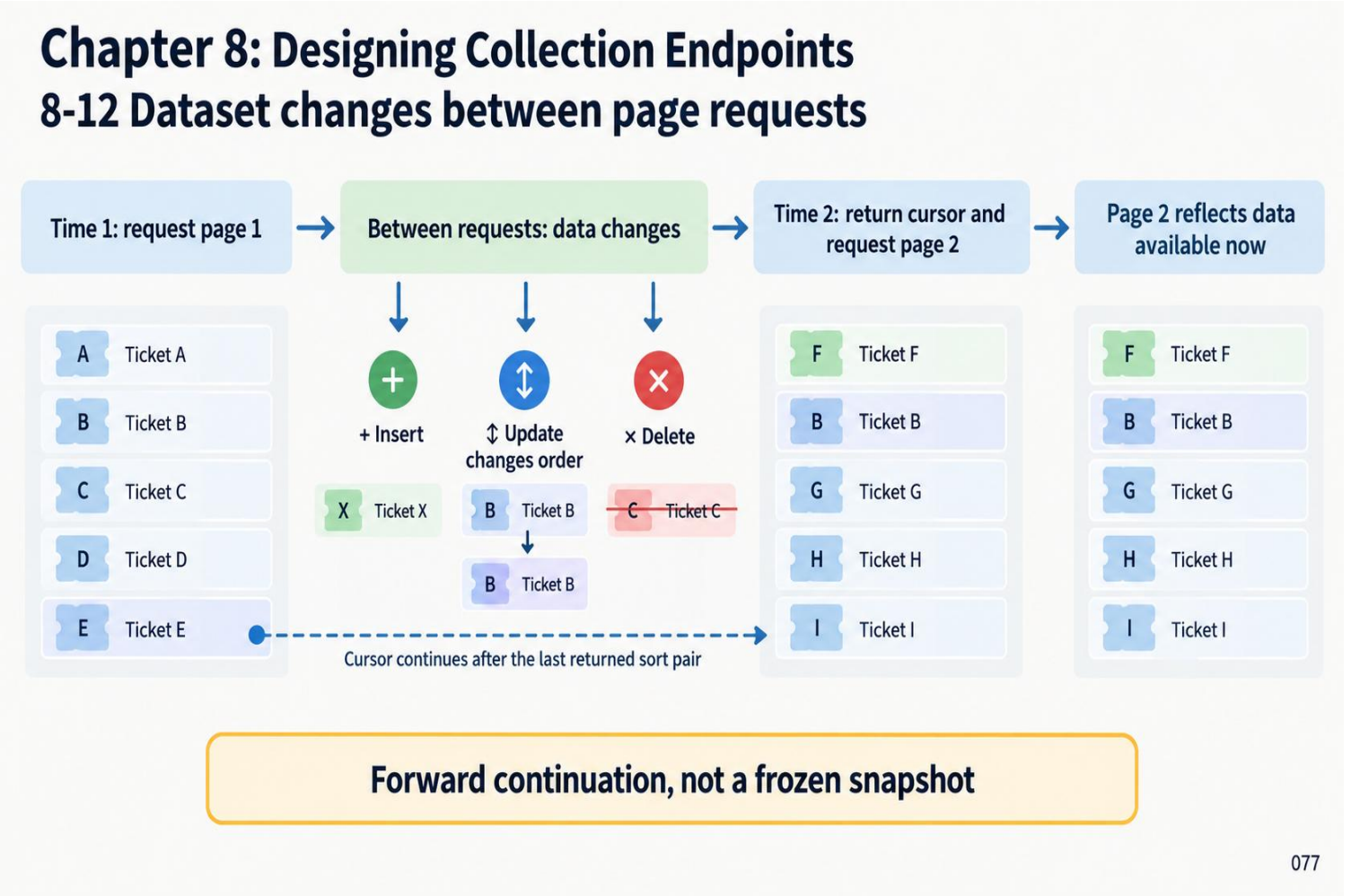



*Figure 8-9: Dataset changes between page requests*

With offset pagination, an inserted item near the front can shift existing items, causing one to be repeated or skipped. Cursor pagination based on stable sort values usually reduces this problem, but it does not necessarily provide a historical snapshot. Updated items may move in the order, and deleted items disappear.

The contract should state the consistency model in practical terms. For many operational lists, this is sufficient:

> Each page reflects data available when that page is requested. Results may change between pages. The cursor provides stable forward continuation but does not guarantee a frozen snapshot.

A reporting export that requires an exact point-in-time dataset needs a stronger mechanism than ordinary interactive pagination. That is a separate use case and should not be implied by a normal list endpoint.

## Sparse field selection is optional

**Sparse field selection** lets a client request only selected response fields, for example:

```
GET /tickets?fields=id,subject,status
```

It can reduce payload size for broad representations, but it adds schema, authorization, caching, testing, and client-complexity considerations. Do not add it by default when a compact collection representation already satisfies consumers.

Another simple design is to define a concise list-item representation and let clients retrieve full details only when needed.

## Two collection-design mistakes

### “Pagination can be added later without affecting the contract”

Changing a top-level array into an envelope, introducing limits, or requiring continuation can break existing clients. Design bounded retrieval from the first release when the collection can grow.

### “Search, filtering, and sorting are interchangeable”

They make different promises. Exact filters, deterministic ordering, and relevance-based text search need separate definitions so clients can predict results.

## Collection endpoint review

Confirm that the contract defines:

- The top-level response shape.
- Default and maximum page sizes.
- Offset or cursor behavior and why it fits the use case.
- A deterministic default order with a unique tie-breaker.
- Supported filters and their comparison rules.
- Supported sort fields and direction syntax.
- Search behavior, when search is offered.
- The continuation token or link contract.
- Whether counts are exact, approximate, absent, or separately requested.
- What may happen when data changes between pages.

**A well-designed collection endpoint lets the consumer process large data sets incrementally without relying on accidental order, unlimited responses, or undocumented query behavior.**

## Chapter Summary

- Pagination is part of the public contract because it affects navigation, consistency, and performance.
- Cursor tokens should be treated as opaque and ordering must include a deterministic tie-breaker.
- Filtering, sorting, and search solve different problems and need explicit parameter conventions.

## Review Questions

1. Why can offset pagination repeat or skip items when a collection changes?
2. What makes an ordering deterministic across pages?
3. Why might an exact total count be optional or expensive?

# Part III — Protecting Correctness and Access

# Chapter 9 — Making State Changes Safe

**A state-changing API must assume that requests can be duplicated, clients can act on stale data, and several users can change the same resource at nearly the same time.** Correctness depends on explicit lifecycle rules, retry-safe operations, concurrency checks, and clear transaction boundaries.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Design retry-safe state changes using natural idempotency or idempotency keys.
- Detect and recover from stale writes with optimistic concurrency controls.
- Define transaction and batch boundaries that match the business outcome promised to clients.

## Begin with the resource lifecycle

A **resource lifecycle** is the set of states a resource may occupy and the allowed transitions between those states. For a ticket, a simplified lifecycle may be:

```
open → in_progress → resolved → closed
          ↑              |
          └──────────────┘
```

The return arrow represents reopening a resolved ticket into `in_progress`. The diagram is only a summary; the contract still needs to define the conditions for each transition.

A transition table can make the rules precise:

| Requested transition | Allowed source state | Required information | Result |
|---|---|---|---|
| Start work | `open` | Assignee or authenticated agent identity | `in_progress` |
| Resolve | `in_progress` | Resolution note | `resolved` |
| Reopen | `resolved` | Reopen reason | `in_progress` |
| Close | `resolved` | No additional data in this example | `closed` |

The table does not imply that these rules fit every support organization. It demonstrates that **status is governed behavior, not an unrestricted text field**.

When a caller requests an invalid transition, the provider should reject it as a business conflict rather than silently force the new state. A client can then refresh the ticket and present valid next actions.

## Why duplicate requests happen

A duplicate request is not necessarily a user mistake. It can arise through normal distributed-system behavior:

1. The client sends a ticket-creation request.
2. The server creates the ticket and begins sending the response.
3. The network connection fails before the client receives the response.
4. The client cannot tell whether the server completed the request.
5. The client sends the request again.

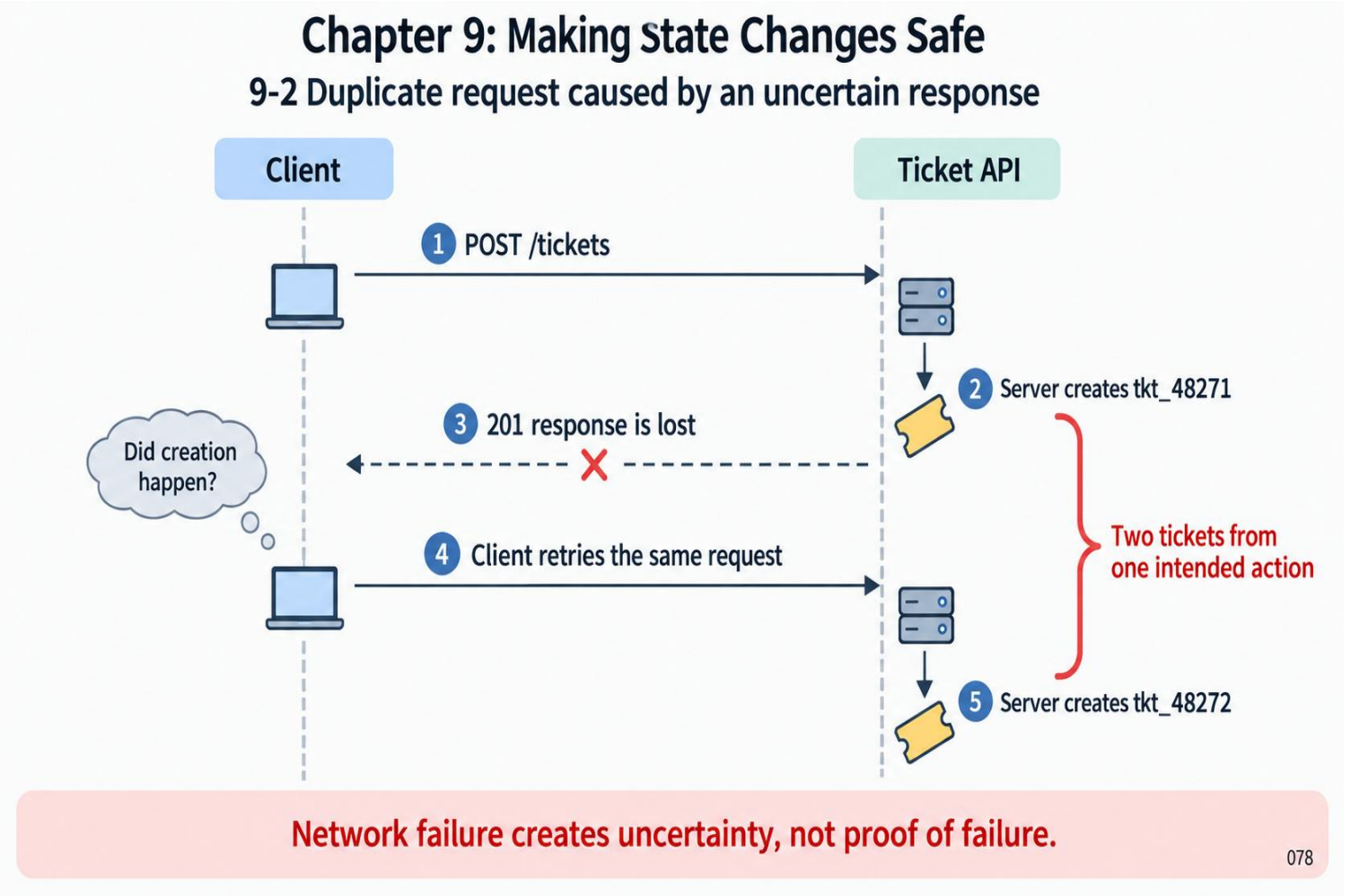



*Figure 9-1: Duplicate request caused by an uncertain response*

Without a retry-safety mechanism, the second request may create another ticket.

Duplicates can also come from:

- A user pressing a button more than once.
- A client timeout configured shorter than server processing time.
- A **job runner**, meaning software that executes scheduled or queued work, retrying after a process restart.
- A gateway or library retry under documented conditions.
- Message redelivery in an integration workflow.

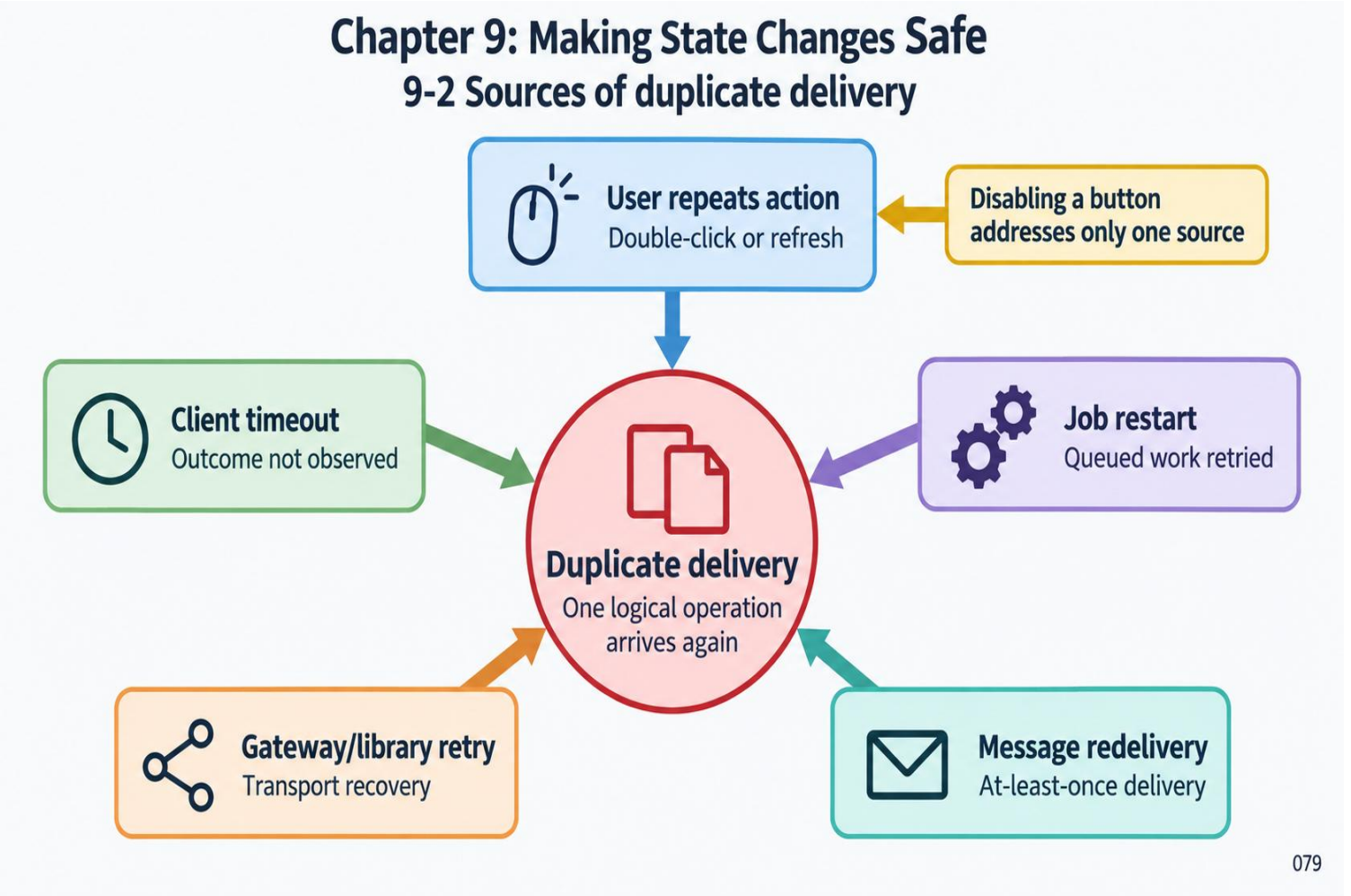



*Figure 9-2: Sources of duplicate delivery*

Disabling a submit button improves the user experience, but it does not solve network uncertainty or automated retries. Correctness belongs in the API contract and provider behavior.

## Prefer naturally idempotent operations where they fit

An operation is naturally idempotent when repeating the same request does not create another intended effect.

Setting priority to a specific value can be idempotent:

```
PATCH /tickets/tkt_48271
Content-Type: application/merge-patch+json

{
  "priority": "high"
}
```

Repeating the request still asks for `priority` to be `high`.

An incremental command is not naturally idempotent:

```
{
  "increasePriorityBy": 1
}
```

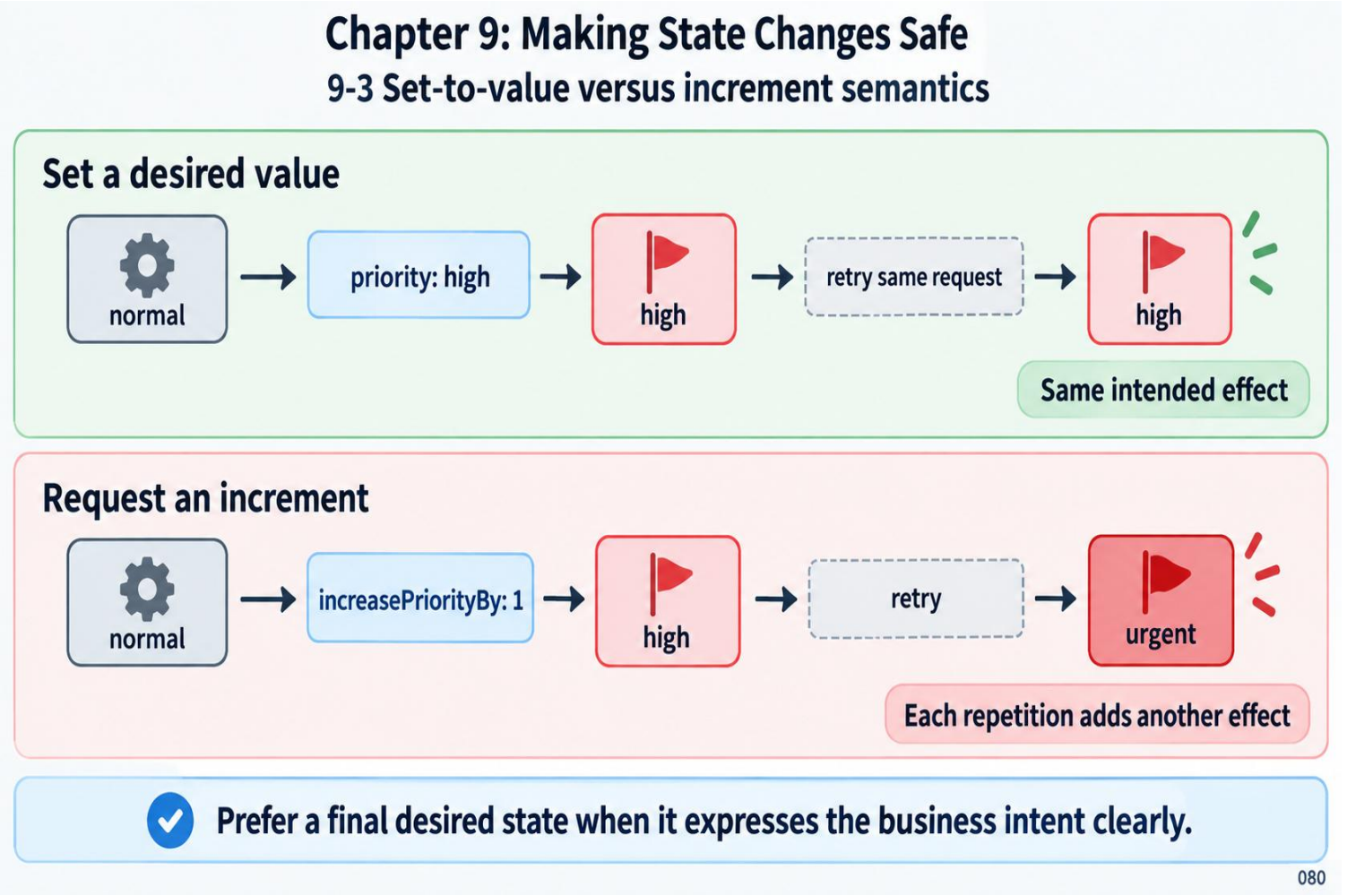



*Figure 9-3: Set-to-value versus increment semantics*

Every repetition requests another increase. When a final desired state can be expressed clearly, setting that state is often safer than requesting an unbounded increment.

Natural idempotency does not solve every problem. Creating a server-identified resource with `POST /tickets` is not naturally idempotent, and some business actions intentionally create a new event each time. Those operations need another mechanism when safe retries are required.

## Use idempotency keys for retryable logical operations

An **idempotency key** is a client-provided identifier that lets a server recognize repeated attempts at the same logical operation. The client creates one key for the business attempt and reuses that key only when retrying that attempt.

A creation request can include:

```
POST /tickets HTTP/1.1
Host: api.example.com
Authorization: Bearer <access-token>
Content-Type: application/json
Idempotency-Key: 8f0b7f35-2f37-49e3-8ddb-f7b7f68e4604

{
  "subject": "Unable to reset password",
  "description": "The reset link expires immediately.",
  "priority": "normal"
}
```

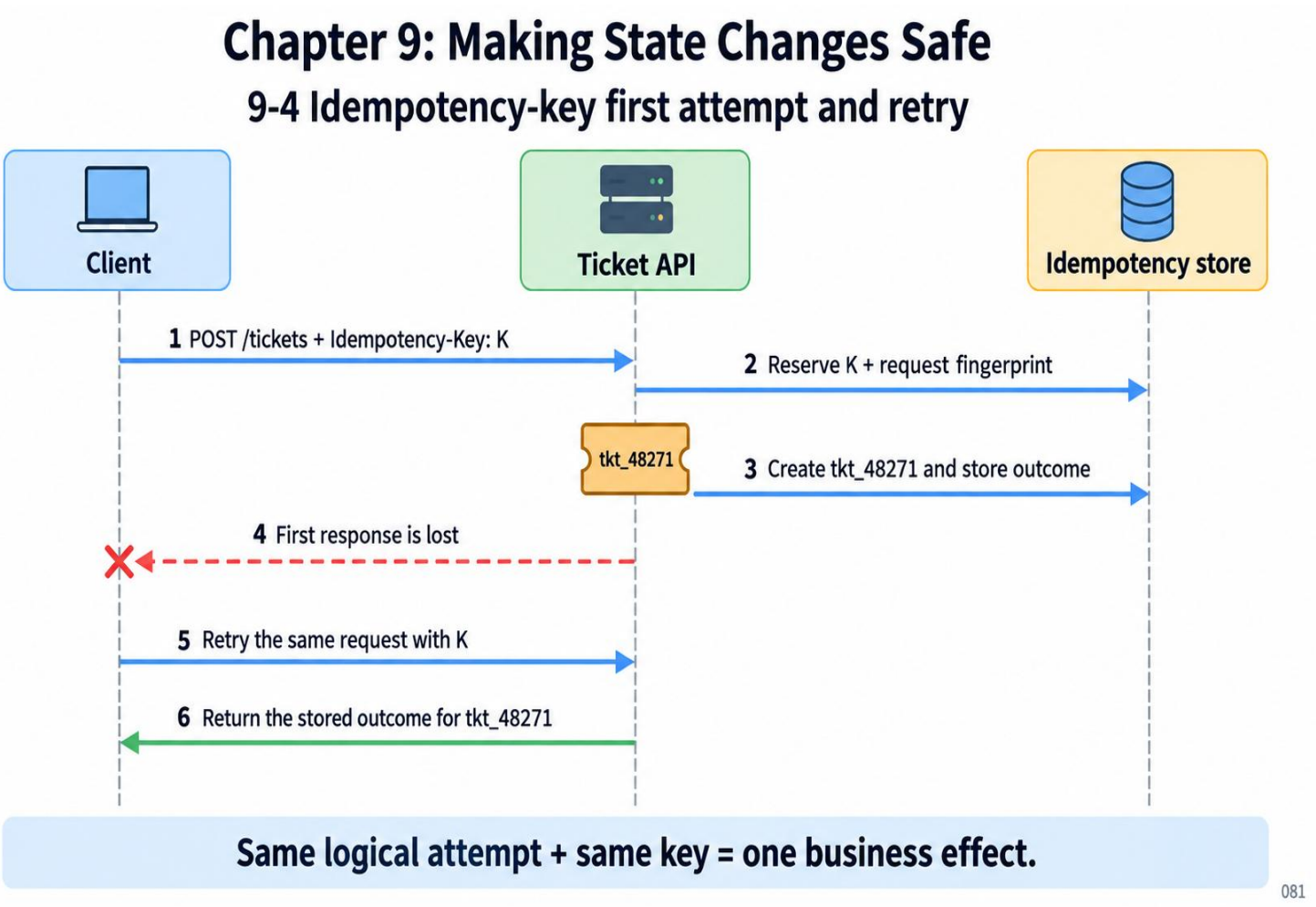



*Figure 9-4: Idempotency-key first attempt and retry*

On the first attempt, the provider processes the request and records the key with the logical request and outcome. If the same caller repeats the same operation with the same key, the provider recognizes it as a retry instead of creating another ticket.

The key should be unique enough that accidental reuse is extremely unlikely. A random **universally unique identifier (UUID)**, a standard identifier designed to have an extremely low collision probability, is a common choice, but the API need not require one specific generation method unless interoperability requires it.

### One key represents one logical attempt

Suppose a user intentionally creates two tickets with identical text. Those are two logical operations and need two keys. Conversely, retries caused by a timeout use the same key.

```
First intended ticket  → key A for every retry of that creation
Second intended ticket → key B for every retry of that creation
```

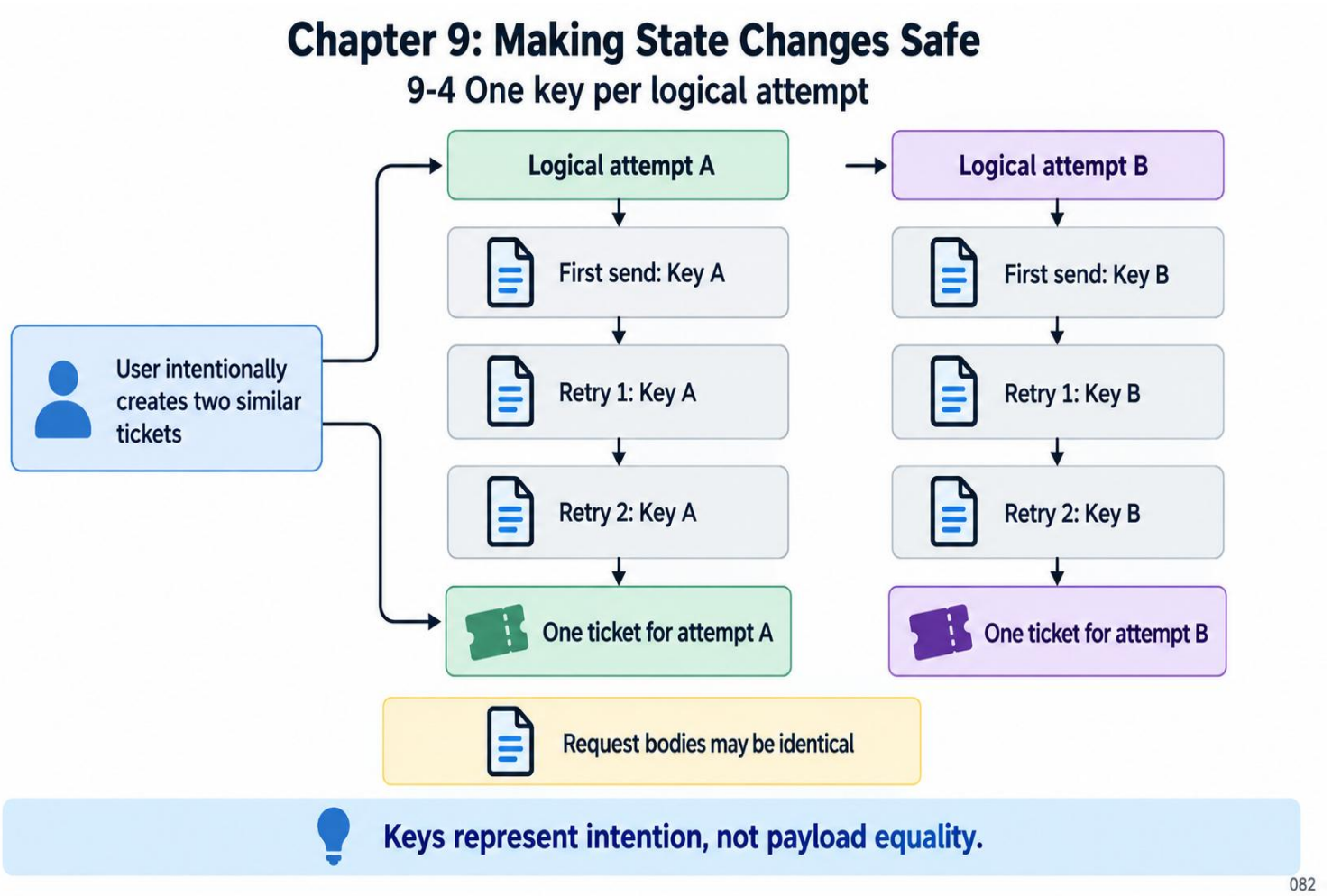



*Figure 9-5: One key per logical attempt*

The request body is not a reliable idempotency key by itself. Two legitimate tickets may have identical bodies, and semantically identical JSON can have different whitespace or member order.

## Define idempotency-key behavior completely

Adding an `Idempotency-Key` header name is not enough. The contract should define the following points.

### Scope

The **scope** determines where a key must be unique. A practical scope may include:

- The authenticated caller or **tenant**, meaning an organization or customer whose data is logically isolated.
- The HTTP method.
- The operation or target path.

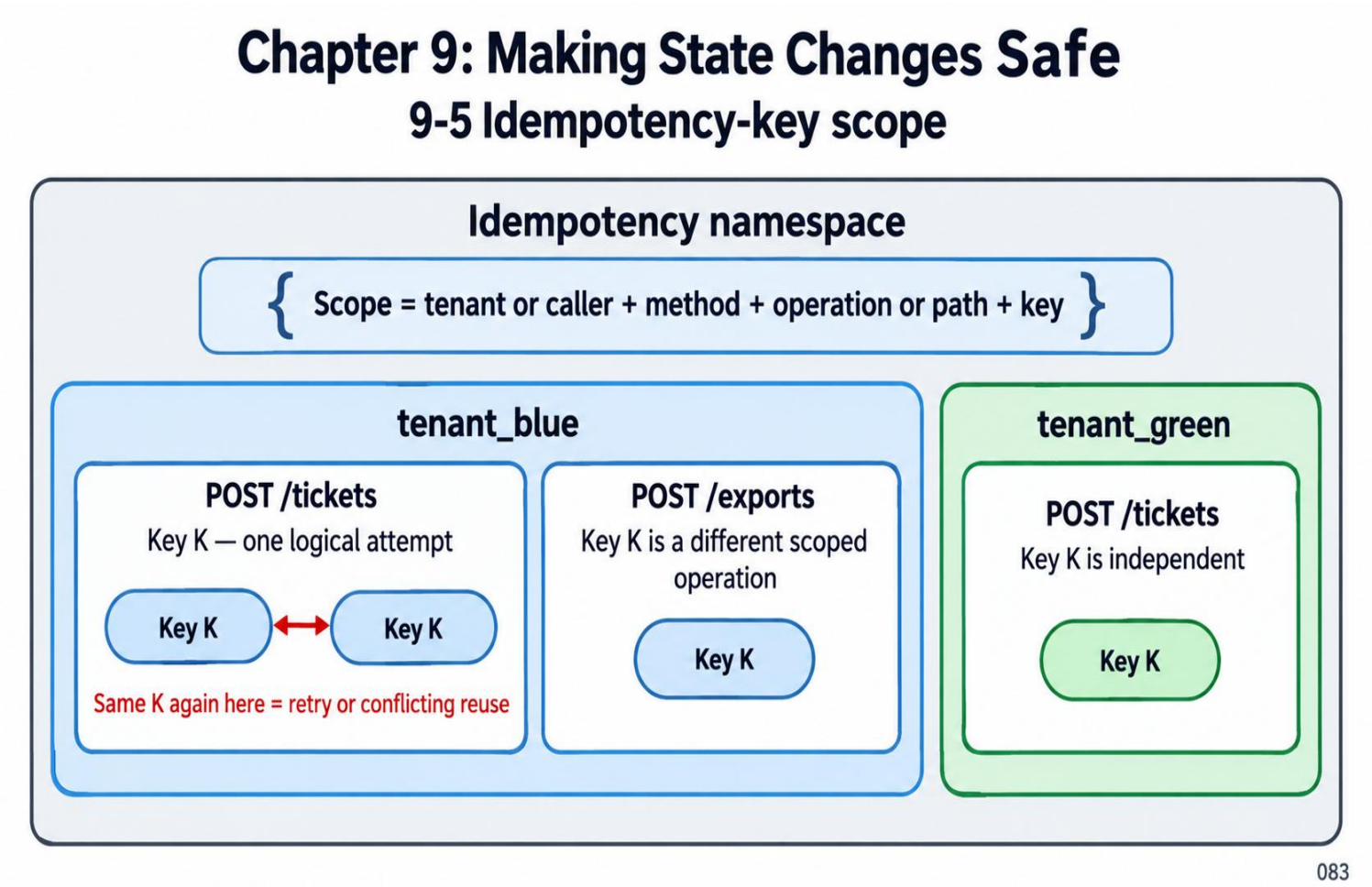



*Figure 9-6: Idempotency-key scope*

This lets two different tenants safely use the same random value and prevents a key from one operation from accidentally matching another.

Do not use a global scope without considering storage, privacy, and abuse consequences. Do not use a scope so narrow that the same logical operation can bypass detection through an equivalent path.

### Request matching

The provider should associate the key with a **request fingerprint**, which is a comparison value derived from the operation and relevant input. If the same key is reused with materially different input, the provider should reject it rather than return an unrelated result.

For example:

```
Key K + subject "Password reset" → accepted
Key K + subject "Printer offline" → rejected as inconsistent key reuse
```

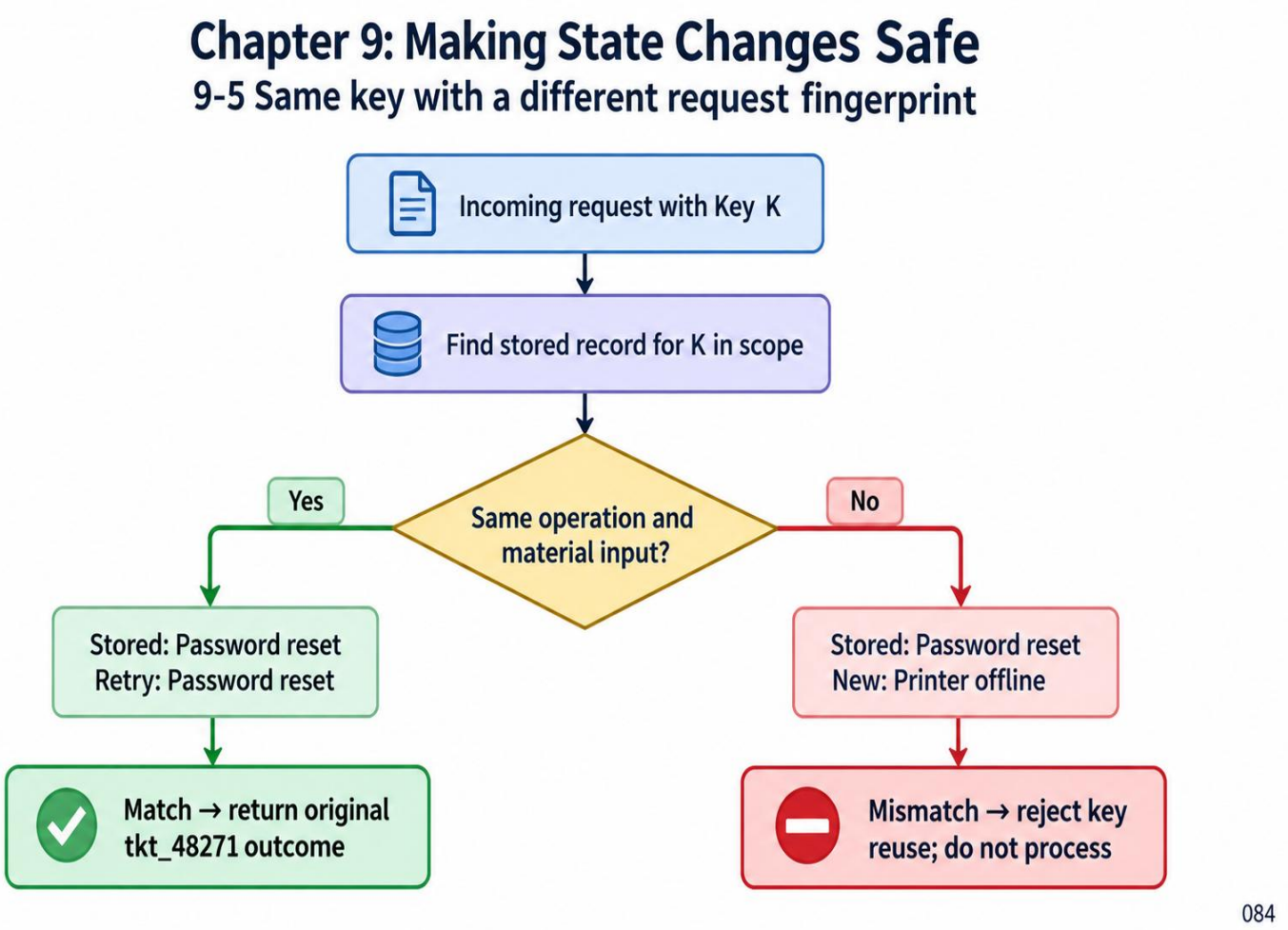



*Figure 9-7: Same key with a different request fingerprint*

The fingerprint is a server implementation detail. The public contract needs only to state which parts of the request must remain the same.

### In-progress duplicates

A retry can arrive while the first request is still running. The provider must define what happens, such as:

- Wait for the original attempt and return its outcome.
- Return a specific response indicating that processing is still in progress.
- Reject concurrent use temporarily and tell the client how to check or retry.

Whichever behavior is chosen, the provider must not start a second independent creation simply because the first has not finished.

### Completed duplicates

When the first request has completed, a duplicate should produce the defined original outcome or an equivalent outcome that identifies the same created resource. For example, it may return the same `201 Created`, `Location`, and ticket representation.

If response replay is not exact, document which values can differ. A new diagnostic request identifier may be reasonable, while a different ticket identifier is not.

### Retention period

The provider cannot necessarily retain every key forever. The contract should state how long retry recognition is guaranteed, such as 24 hours. After expiration, reuse may be treated as a new request or rejected according to the documented policy.

The retention period should be longer than the realistic client retry window. A five-minute retention is inadequate for a batch job that may recover after several hours.

### Failure behavior

The contract should define whether failed attempts are stored. A useful policy often distinguishes:

- Requests rejected before processing because the input is invalid.
- Requests that begin processing and reach a durable result.
- Requests whose final outcome is uncertain.

The design must avoid trapping a corrected request behind a key associated with invalid input while also avoiding duplicate processing after an uncertain server failure.

There is no single policy for every system, but the policy must be deliberate and testable.

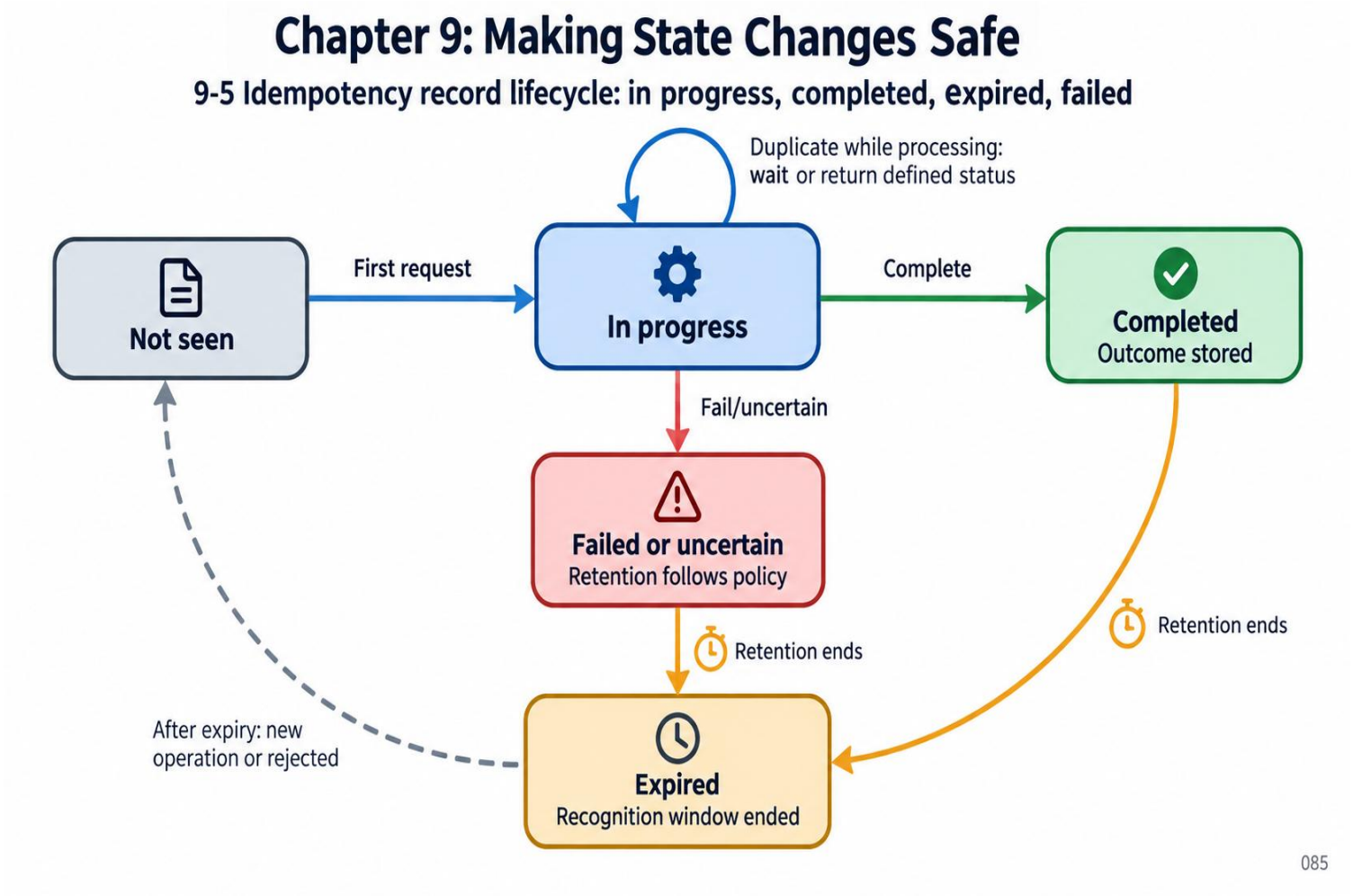



*Figure 9-8: –9.5.6 Idempotency record lifecycle: in progress, completed, expired, failed*

## Create safely with an idempotency key

The following is one HTTP API example.

**Prerequisites:** `curl` is available in a POSIX-style shell, the example service is reachable, `ACCESS_TOKEN` contains an accepted credential, and `IDEMPOTENCY_KEY` contains a newly generated value that will be reused only for retries of this one ticket creation.

**Purpose:** Create one logical ticket safely even if the client must repeat the request after an uncertain network failure.

```
curl --include \
  --request POST \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --header "Content-Type: application/json" \
  --header "Idempotency-Key: $IDEMPOTENCY_KEY" \
  --data '{
    "subject": "Unable to reset password",
    "description": "The reset link expires immediately.",
    "priority": "normal"
  }' \
  "https://api.example.com/tickets"
```

**Expected result:** The first successful attempt creates one ticket and returns `201 Created` with its location and representation. Repeating the same request with the same key during the documented retention period returns the outcome for that same logical creation rather than creating another ticket. Reusing the key with different ticket data produces a defined client error.

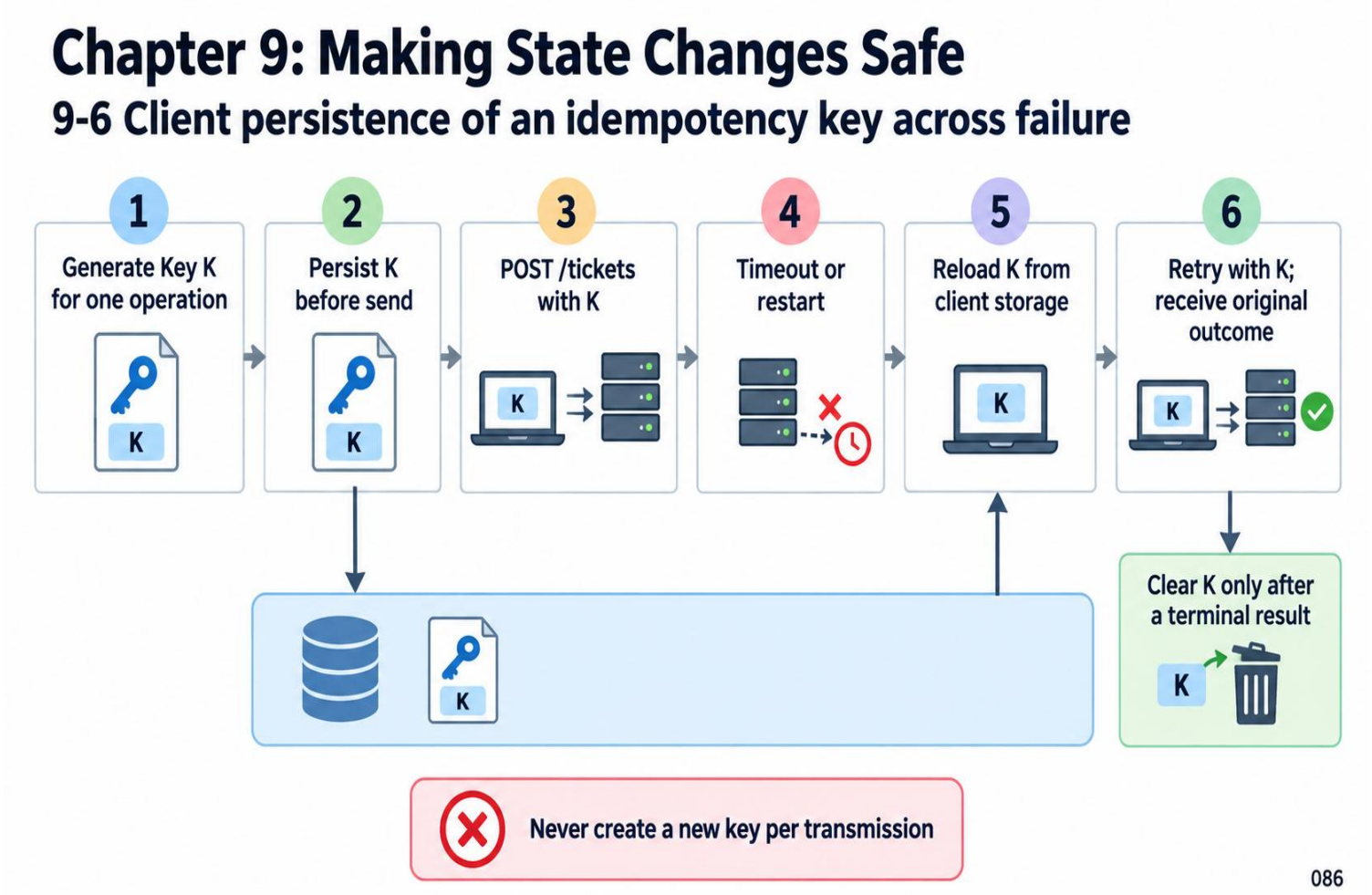



*Figure 9-9: Client persistence of an idempotency key across failure*

The client must persist the key long enough to retry after a process or network failure. Generating a fresh key for every transmission defeats duplicate detection.

## Idempotency keys do not replace business uniqueness

An idempotency key answers: “Is this a retry of the same client operation?” It does not answer: “Should the business allow two resources with the same real-world identity?”

For example, an external monitoring system may include an incident identifier. The business may require only one open ticket for each incident. That rule should be modeled as a business uniqueness constraint or a defined create-or-update operation, not only as a temporary idempotency record.

The two mechanisms solve different problems:

- **Idempotency key:** Prevent duplicate effects from repeated delivery of one request.
- **Business key:** Identify the real-world entity or uniqueness rule across independently initiated requests.

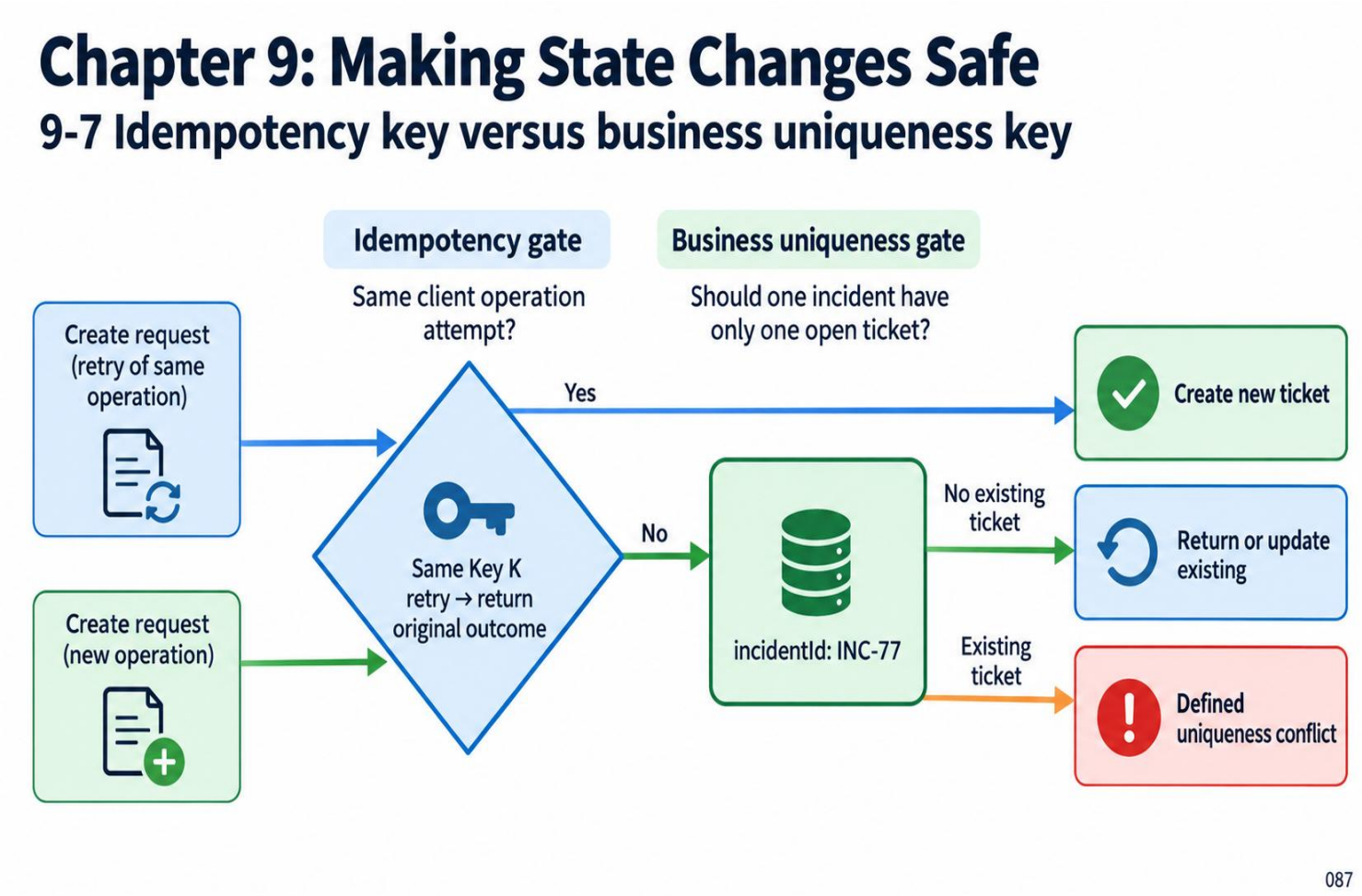



*Figure 9-10: Idempotency key versus business uniqueness key*

A client may submit the same business incident tomorrow with a new idempotency key. The provider still needs the business rule if duplicates are not allowed.

## Concurrent updates create lost changes

Even when every request arrives only once, concurrent clients can overwrite one another.

Consider this sequence:

1. Agent A retrieves ticket version 7.
2. Agent B retrieves the same version 7.
3. Agent A changes the priority to `urgent`; the ticket becomes version 8.
4. Agent B, still working from version 7, changes the subject and sends the entire old representation.
5. If the server accepts it blindly, Agent B's old `priority` value can overwrite Agent A's change.

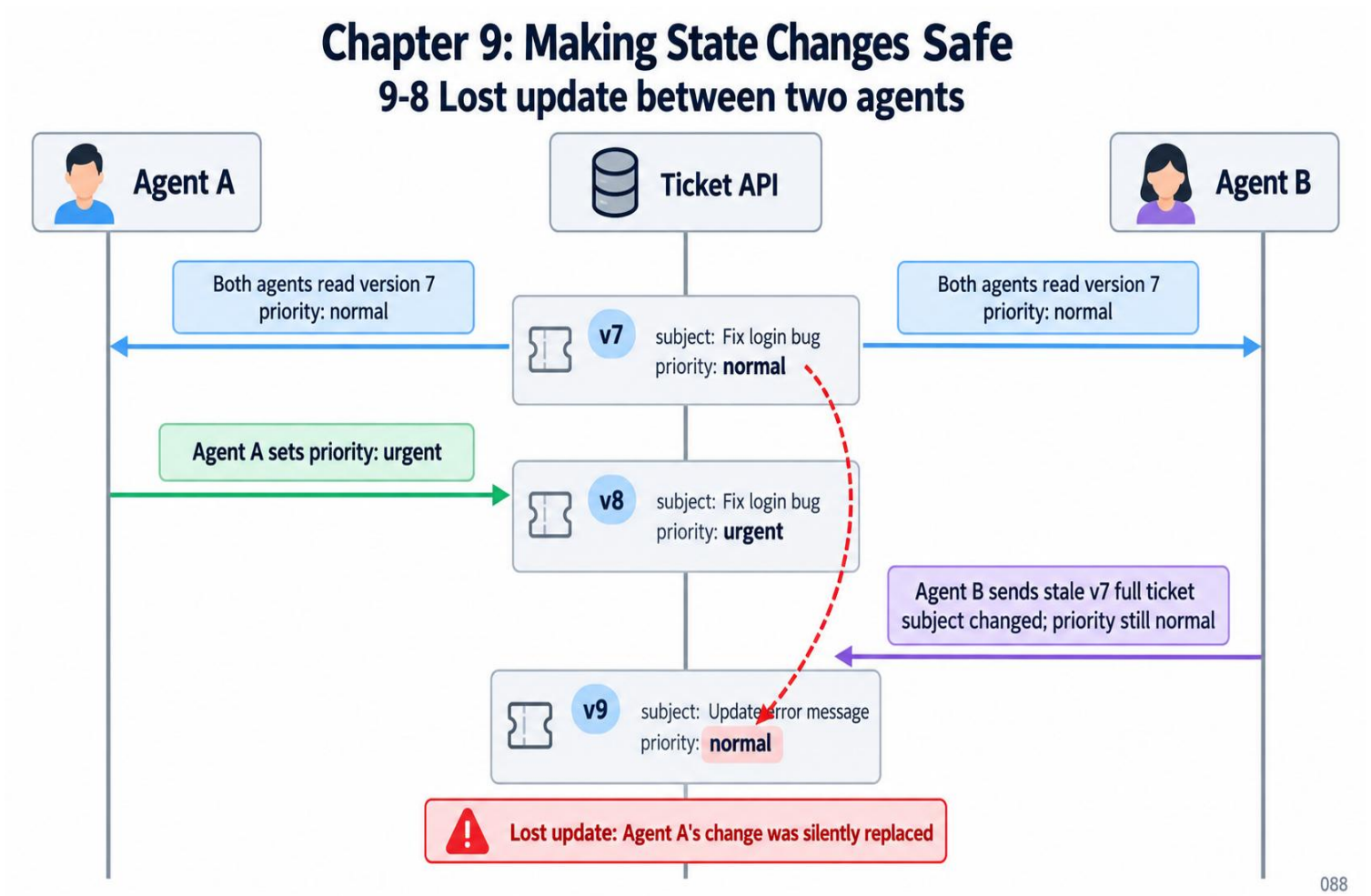



*Figure 9-11: Lost update between two agents*

This is a **lost update**, which occurs when one successful write unintentionally replaces another successful write because the later writer used stale state.

“Last write wins” is simple, but it silently discards work. That may be unacceptable for assignments, financial data, inventory, approvals, and many ordinary business records.

## Optimistic concurrency detects stale writes

**Optimistic concurrency control** allows clients to read and work independently but rejects an update when the resource has changed since the client read it. It is “optimistic” because it assumes conflicts are uncommon and checks at write time instead of locking the resource for the whole editing period.

HTTP provides a useful mechanism through entity tags and conditional requests.

An **entity tag (`ETag`)** is a response header value that identifies a particular version of a resource representation for validation purposes. The value is opaque to the client.

A retrieval can return:

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "ticket-48271-v7"

{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "priority": "normal",
  "status": "open"
}
```

The client stores the ETag and includes it in an update condition:

```
PATCH /tickets/tkt_48271 HTTP/1.1
Content-Type: application/merge-patch+json
If-Match: "ticket-48271-v7"

{
  "priority": "urgent"
}
```

`If-Match` means: apply the request only if the current resource version still matches this tag.

The server performs the update atomically with the version check:

1. Compare the current version with `"ticket-48271-v7"`.
2. If it matches, apply the update and create a new version.
3. Return the updated representation and new ETag.
4. If it does not match, reject the request without applying the stale update.

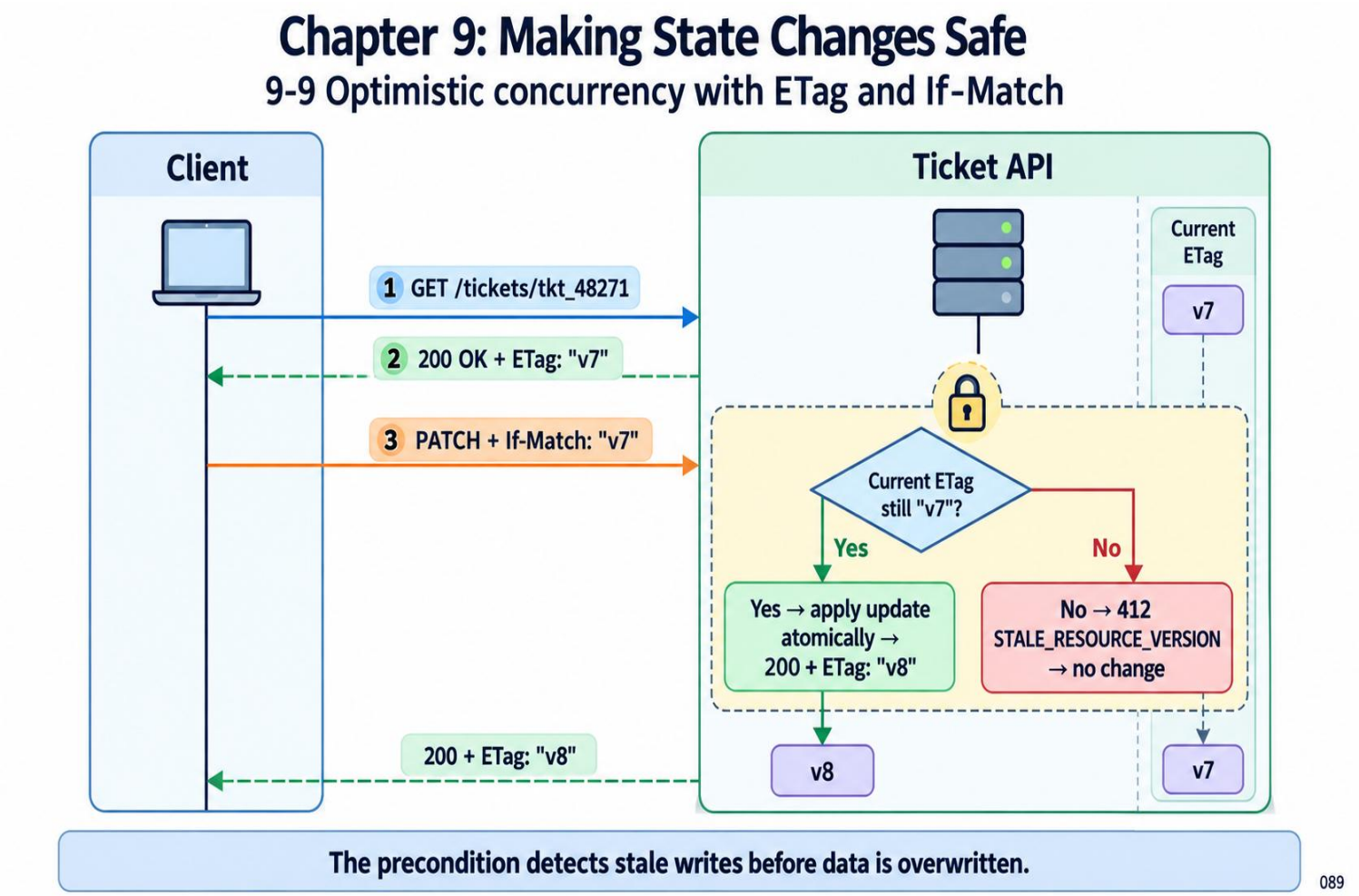



*Figure 9-12: Optimistic concurrency with `ETag` and `If-Match`*

A successful response can be:

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "ticket-48271-v8"

{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "priority": "urgent",
  "status": "open"
}
```

A stale request can receive:

```
HTTP/1.1 412 Precondition Failed
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/stale-resource-version",
  "title": "The ticket has changed",
  "status": 412,
  "code": "STALE_RESOURCE_VERSION",
  "detail": "Retrieve the current ticket before attempting the update again.",
  "requestId": "req_a026d2"
}
```

`412 Precondition Failed` is appropriate because the `If-Match` condition supplied by the client is false.

## Recover from a concurrency conflict deliberately

After a stale-version response, the client should not automatically repeat the same update without review. A safe recovery flow is:

1. Retrieve the current ticket and its new ETag.
2. Compare the current state with the state the user edited.
3. Decide whether the intended change can be merged safely.
4. Present a conflict to the user when human judgment is needed.
5. Submit the revised update with the new ETag.

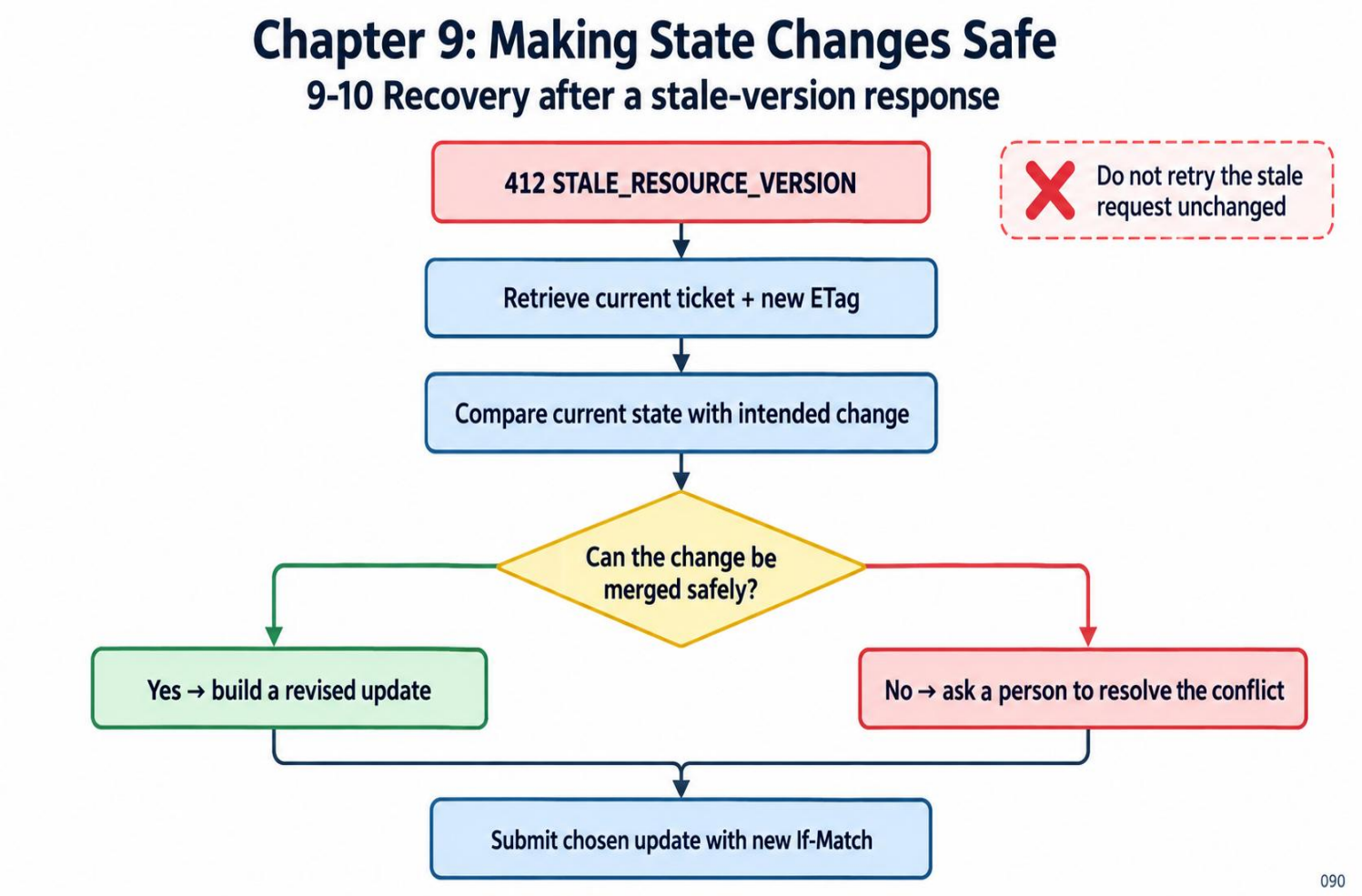



*Figure 9-13: Recovery after a stale-version response*

For a narrow patch that changes only `priority`, an automatic merge may be acceptable if no related rule changed. For a large edited description, silently replacing another agent's work may not be acceptable.

The API exposes the conflict; the consumer decides how to resolve it according to the user experience and business rules.

## Use the right failure for the right conflict

Two failures can appear similar:

- `412 Precondition Failed`: The client's explicit version condition is stale.
- `409 Conflict`: The requested business operation conflicts with current state even apart from an ETag comparison.

Examples:

- Agent A submits `If-Match: "v7"`, but the ticket is now `v8` → `412`.
- The client has the current version, but asks to assign a ticket whose status is `closed` → `409`.

Keeping these conditions distinct helps a client choose the recovery path. A stale version suggests refresh and merge. An invalid business transition suggests a different action.

## Timestamps alone are a weak concurrency mechanism

A client might send `updatedAt` and ask the server to compare it. This can work when carefully designed, but it has common risks:

- Timestamp precision may be too low for rapid updates.
- Converting the timestamp to and from message text can alter precision or time-zone representation.
- Clients may treat a timestamp as editable business data.
- Database and application clocks or rounding may differ.

An opaque version or ETag communicates intent more clearly. Internally, the provider may still derive the ETag from a version number or other change token.

Clients must not interpret the ETag value or attempt to calculate the next one.

## Define the transaction boundary

A **transaction boundary** defines which changes succeed or fail as one unit. An operation is **atomic** when consumers observe either its complete intended effect or none of it, rather than a partially applied state.

Suppose assignment means:

- Set the assignee.
- Change status from `open` to `in_progress`.
- Record `assignedAt`.
- Write an audit entry.

## 9-13 Atomic transaction boundary for assignment

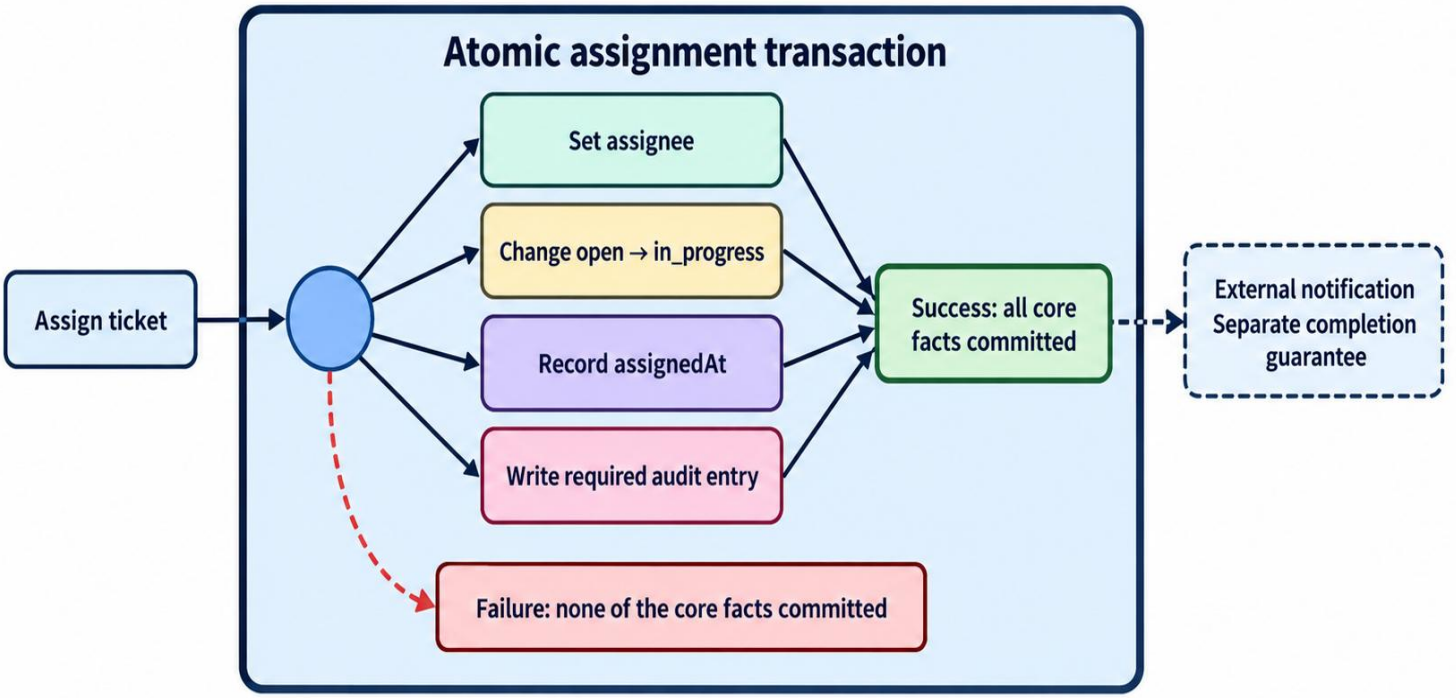



*Figure 9-14: Atomic transaction boundary for assignment*

If the API presents assignment as one operation, consumers will reasonably expect these core changes to agree. Returning success after setting the assignee but before recording the required status would violate the business outcome.

The contract should state the visible guarantee. For example:

> A successful assignment response means that the ticket's assignee, status, assignment time, and required audit record have been committed as one operation.

Not every side effect can always share one storage transaction. Sending email or notifying another independent system may occur outside the core transaction. The API must avoid claiming that external effects are complete when they are not. It should define success in terms of the durable business state it controls.

A useful design question is: **After the client receives success, which facts are guaranteed to be true?**

### Avoid exposing multi-call transactions to ordinary clients

A fragile design might require:

1. Update the ticket assignee.
2. Update the ticket status.
3. Add an audit record.
4. Notify the queue.

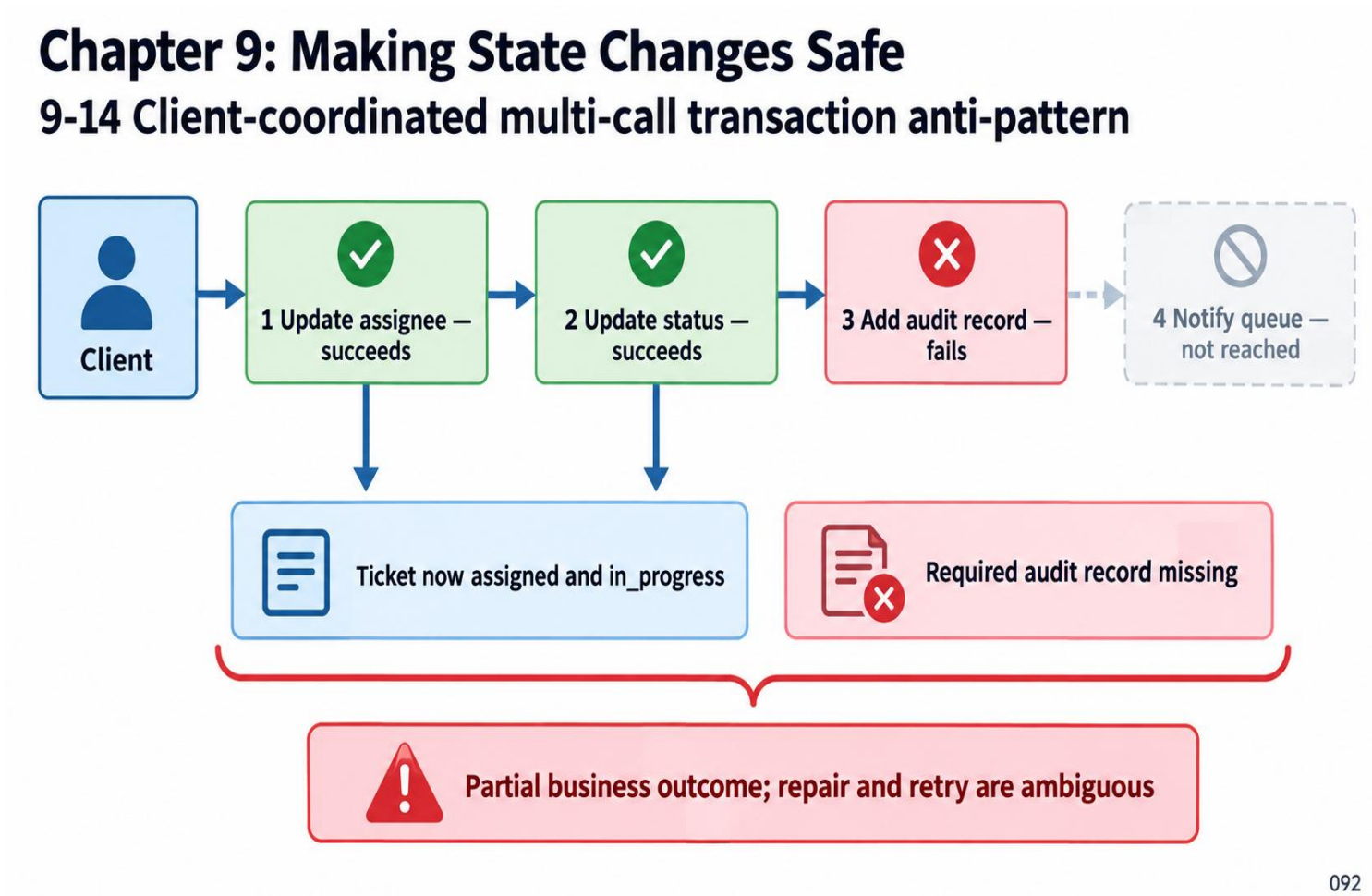



*Figure 9-15: Client-coordinated multi-call transaction anti-pattern*

If the third call fails, the consumer must repair partial state. Different consumers may implement different repair logic. Retrying the sequence can repeat some effects.

When those changes form one business outcome, provide one operation and let the provider manage the internal transaction and recovery behavior.

This does not mean every workflow must fit in one synchronous database transaction. It means the public contract should present coherent outcomes and explicit intermediate states rather than leak accidental internal steps.

## Treat batch operations as a separate design problem

A **batch operation** submits several item operations in one request. It can reduce network overhead, but it introduces difficult questions:

- Is the entire batch atomic?
- Can some items succeed while others fail?
- Are items processed in order or independently?
- How are per-item results correlated with input?
- What happens when the request times out after partial processing?
- Does one idempotency key cover the batch or each item?
- What is the maximum batch size?

Two broad models are common.

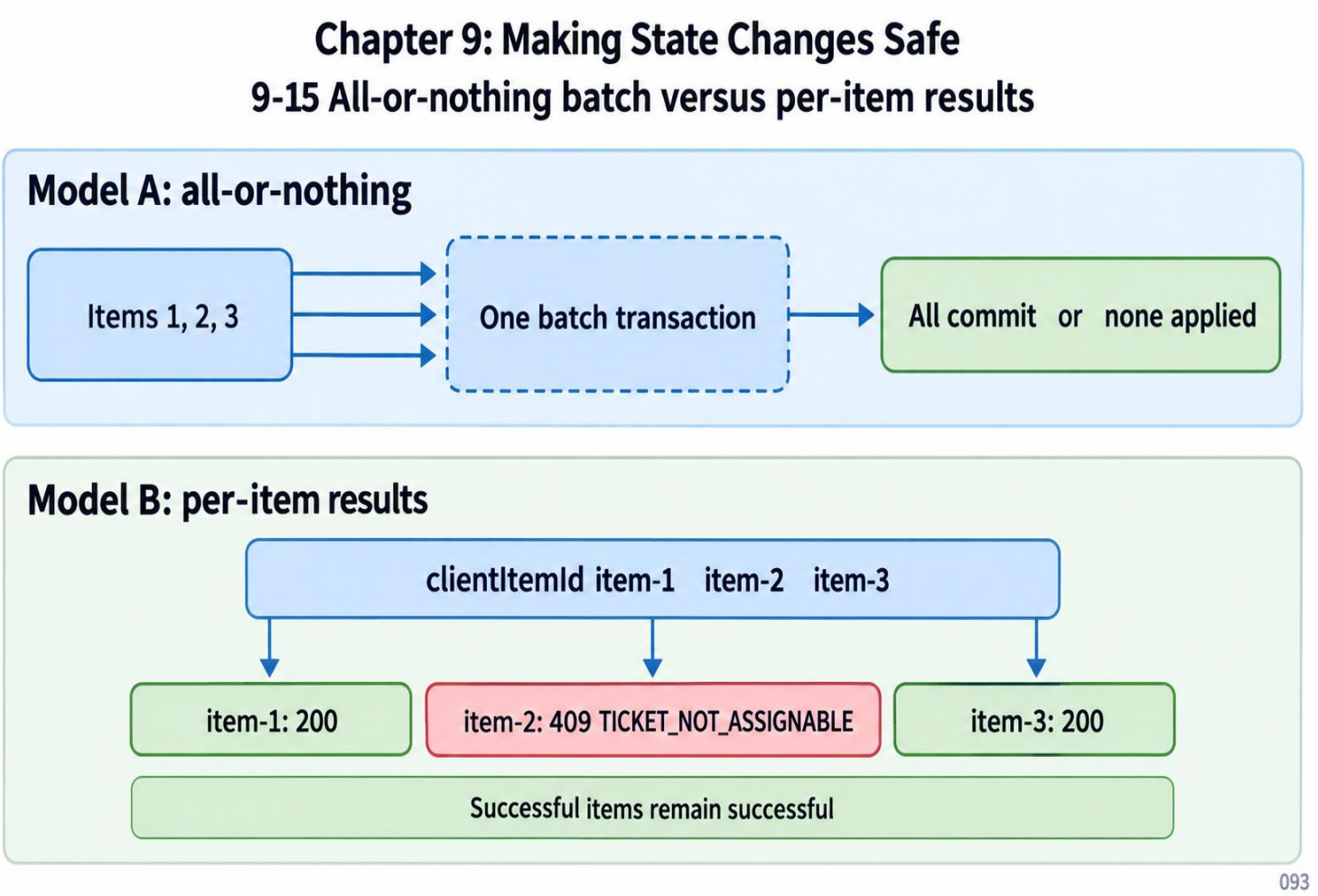



*Figure 9-16: All-or-nothing batch versus per-item results*

### All-or-nothing batch

Every item succeeds, or the provider applies none. This is simple for the consumer but may be expensive or impossible across many independent records or systems.

### Per-item results

Each item can succeed or fail independently. The response must give a result for every input item, using a stable client-supplied item identifier or position.

For example:

```
{
  "results": [
    {
      "clientItemId": "item-1",
      "status": 200,
      "ticketId": "tkt_48271"
    },
    {
      "clientItemId": "item-2",
      "status": 409,
      "code": "TICKET_NOT_ASSIGNABLE"
    }
  ]
}
```

This shape is illustrative. A production batch contract also needs top-level request behavior, limits, retry rules, and idempotency semantics.

Do not add a batch endpoint merely because it appears efficient. First confirm that consumers need it and that the provider can define partial failure without ambiguity. Many teams are better served initially by reliable single-resource operations with controlled concurrency.

## Two correctness mistakes

### "The client can prevent all duplicates by disabling its submit button"

A disabled button prevents some human double-clicks. It cannot determine whether a timed-out server request completed, and it does not protect automation, jobs, gateways, or process restarts. The provider needs idempotent semantics for operations that must be safely retried.

### "Last write wins is harmless for ordinary business data"

Lost updates can change ownership, priority, deadlines, descriptions, or approvals without any visible error. Optimistic concurrency provides a lightweight way to detect stale writes before data is silently replaced.

## State-change design checklist

For every operation that changes state, verify:

- The allowed source states and resulting state are defined.
- Invalid transitions return a stable business conflict.
- The operation is naturally idempotent or has a documented duplicate-request mechanism.
- An idempotency key has defined scope, matching rules, retention, and failure behavior when used.
- The client knows whether a retry can repeat a business effect.
- Concurrent updates cannot silently overwrite important changes.
- Version tokens are opaque and checked atomically with the update.
- Stale preconditions and business-state conflicts are distinguishable.
- The success response states which changes are durably complete.
- The transaction boundary matches the business outcome exposed to the consumer.
- Batch behavior, when offered, defines atomicity and per-item failure clearly.

**A safe API does not assume perfect networks or perfectly coordinated users. It turns duplicate delivery, stale state, and partial failure into explicit parts of the contract.**

## Chapter Summary

- Duplicate delivery and concurrent updates are normal distributed-system conditions, not rare exceptions.
- An idempotency key identifies one logical attempt and must have defined scope, matching, retention, and failure behavior.
- Concurrency preconditions and atomic transaction boundaries prevent silent lost updates and partial business outcomes.

## Review Questions

1. When should a retry reuse an idempotency key, and when should a new key be generated?
2. How do `ETag` and `If-Match` prevent a stale client from overwriting newer work?
3. What must an API define before offering a batch operation?

# Chapter 10 — Designing Security Boundaries

**API security begins with two separate decisions: establish who the caller is, then decide whether that caller may perform the requested action on the specific resource.** Transport encryption, tokens, validation, and rate limits support these decisions, but none replaces them.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Identify assets, callers, trust boundaries, and sensitive data before choosing security mechanisms.
- Separate authentication, authorization, scopes, roles, claims, and resource-level checks.
- Design least-privilege access, credential lifecycle, input validation, CORS, and abuse controls as distinct responsibilities.

## Start with assets, callers, and trust boundaries

An **asset** is something worth protecting, such as ticket content, customer contact details, credentials, administrative functions, or service availability. A **trust boundary** is a point where data or control passes between components that should not automatically trust one another.

Before choosing an authentication technology, describe the security situation in plain language:

- Which assets does the API expose or change?
- Which kinds of caller exist?
- Which actions may each caller perform?
- Which data belongs to a particular user, team, customer, or tenant?
- Where do requests cross a trust boundary?
- How could a caller misuse a valid capability?
- Which failures would have the greatest impact?

This is a lightweight form of **threat modeling**, which is the structured process of identifying what must be protected, how it could be harmed, and which controls reduce that risk.

For the ticket service, the assets include ticket descriptions, attachments, requester details, internal comments, assignment controls, deletion controls, and audit history. The callers include requesters, support agents, administrators, automation services, and reporting jobs. A design that says only “all calls require a token” has not yet answered the important questions.

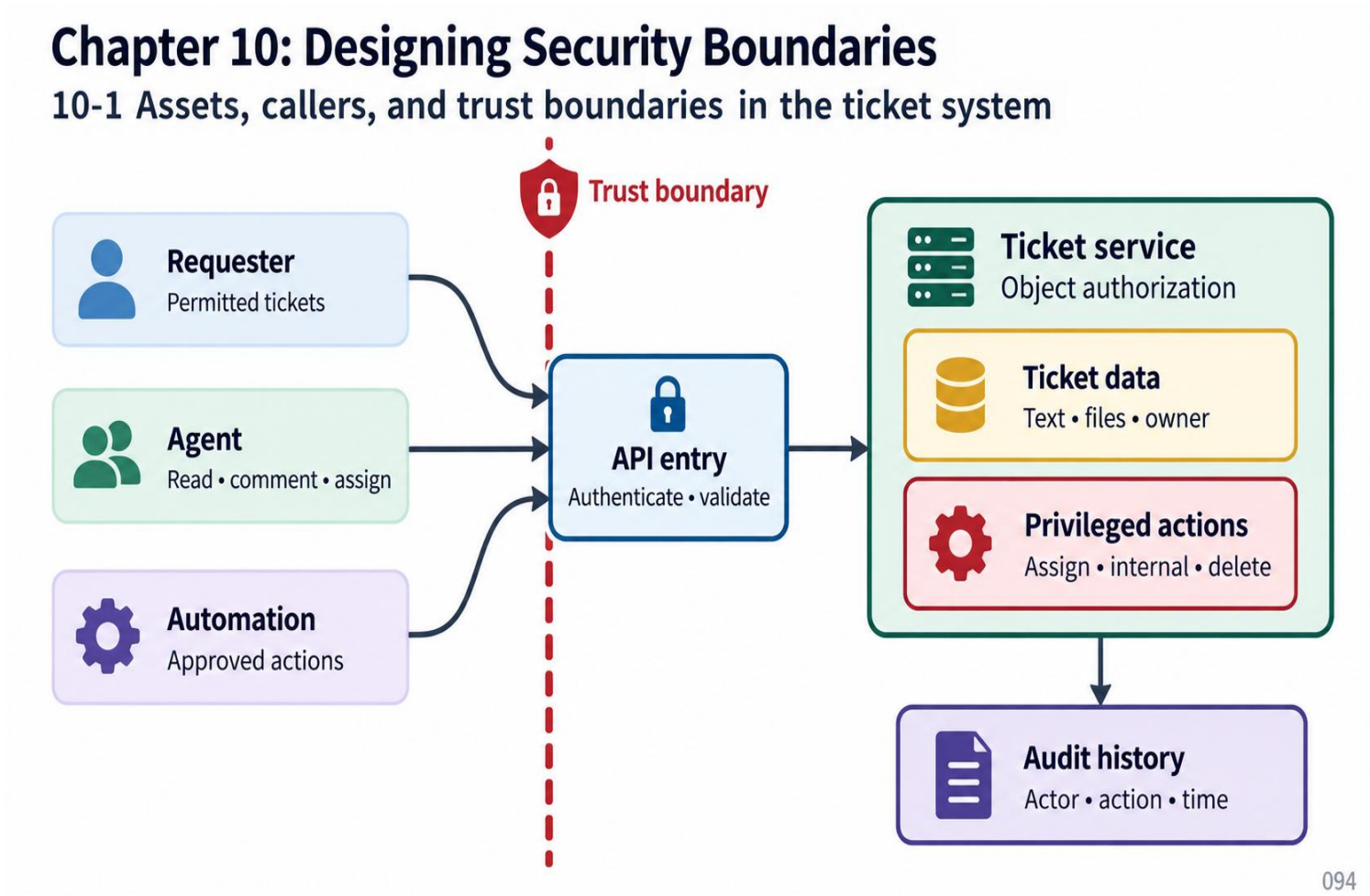



*Figure 10-1: Assets, callers, and trust boundaries in the ticket system*

A useful analogy is a secured office building. Showing a valid badge at the entrance proves that a person has an accepted identity. It does not automatically permit that person to enter every room, open every cabinet, or remove equipment. The API needs the equivalent of both the entrance check and the room-by-room access rules.

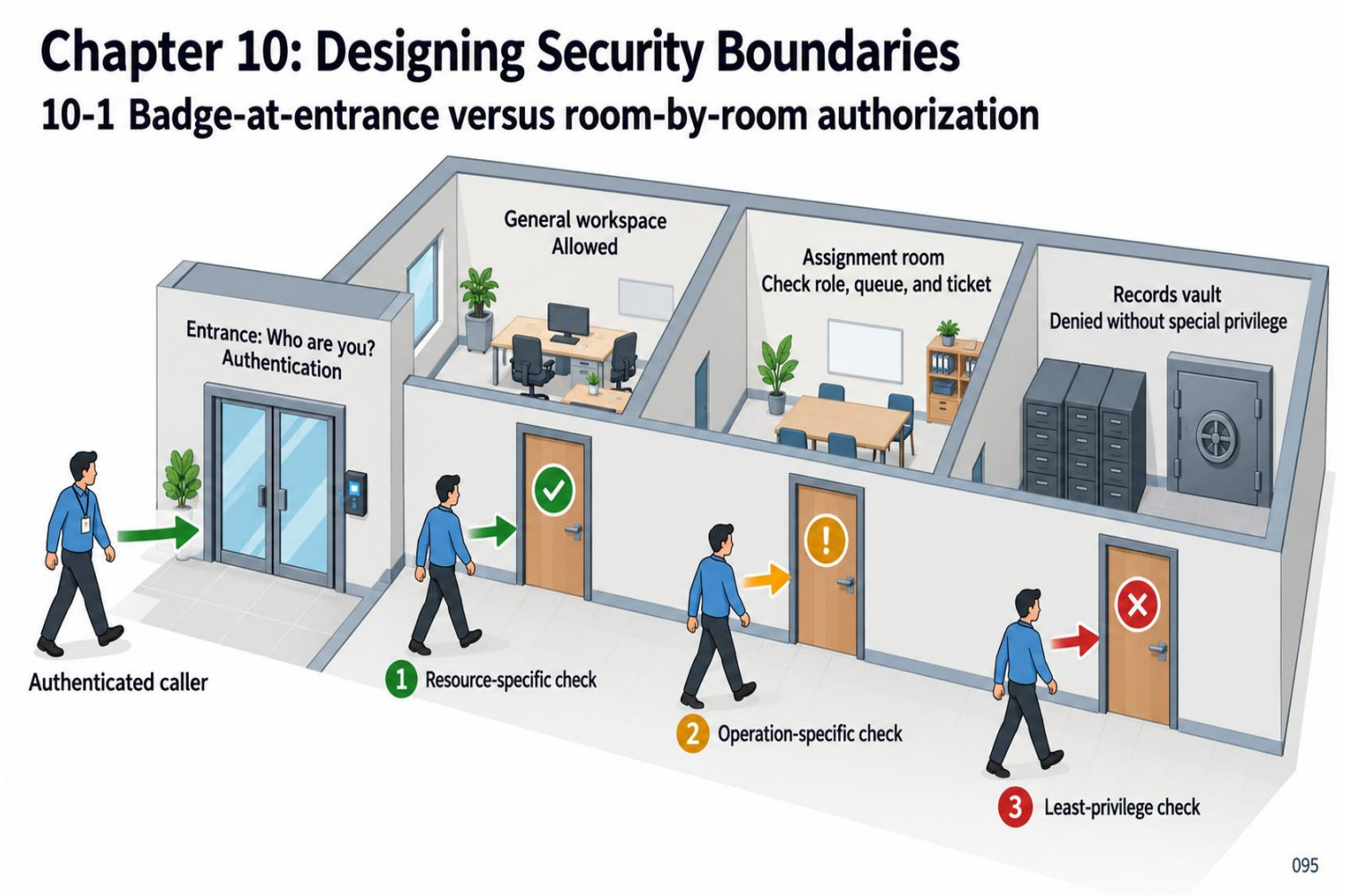



*Figure 10-2: Badge-at-entrance versus room-by-room authorization*

## HTTPS is the baseline, not the complete security model

**HTTPS** is HTTP protected by Transport Layer Security, which encrypts data in transit and authenticates the server to the client when certificate validation succeeds. It protects credentials and business data from ordinary network interception and modification.

Every production API that carries credentials or nonpublic data should use HTTPS. Redirecting an initial unencrypted request to HTTPS is not equivalent to starting securely, because sensitive information may already have been sent. Clients should call the HTTPS address directly.

HTTPS does not determine whether a valid caller may read a ticket. It also does not protect data after the provider has received it, prevent insecure logging, validate input, or stop an authorized caller from abusing excessive privileges. It secures the transport path; the application still needs its own security decisions.

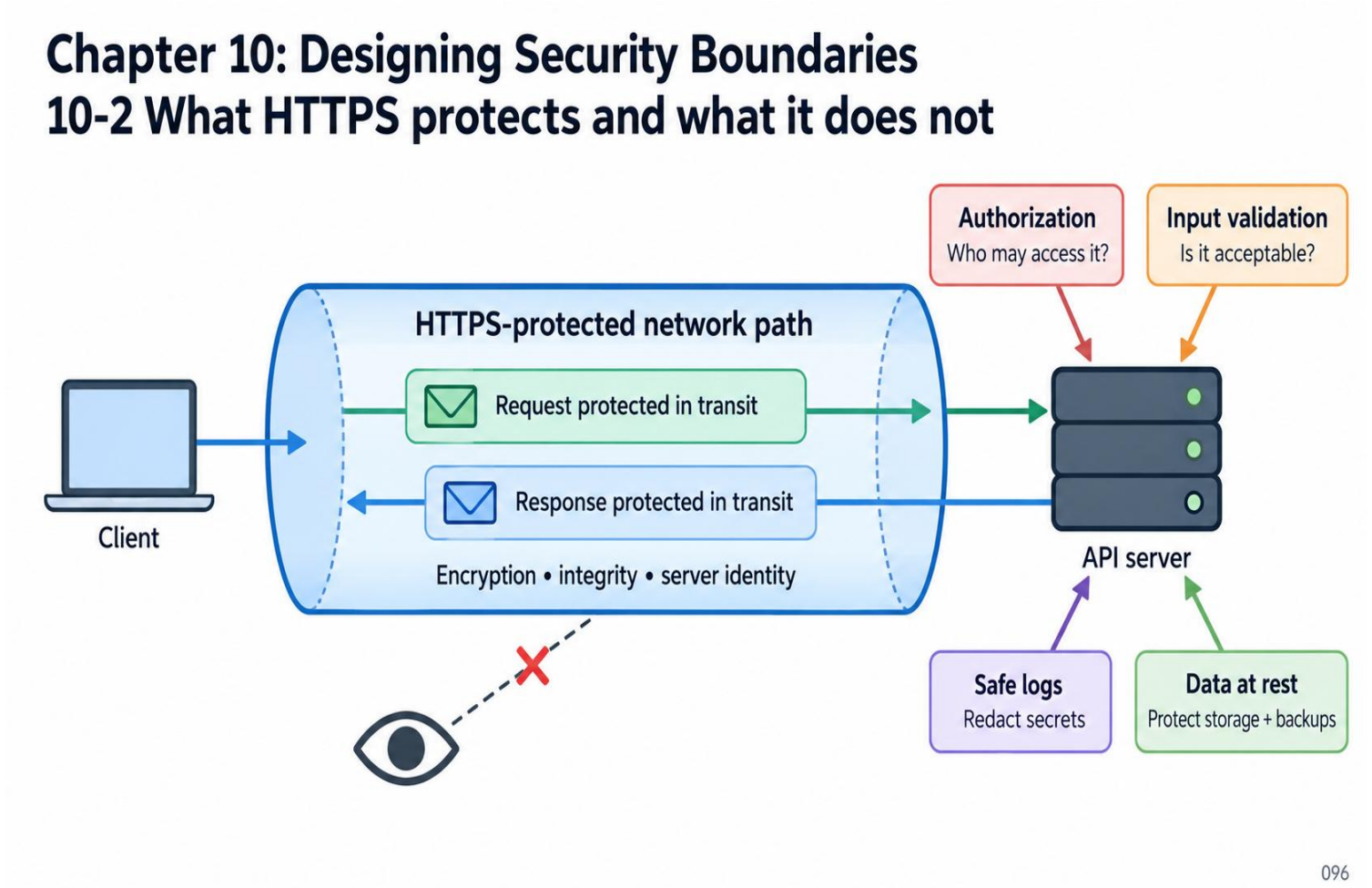



*Figure 10-3: What HTTPS protects and what it does not*

## Authentication proves an identity

**Authentication** is the process of establishing the identity of a caller. The resulting identity may represent a person, an application, a workload, or a combination of these.

An API commonly receives a **credential**, which is evidence used to authenticate the caller. Several credential forms appear in practical systems:

| Credential form | Typical identity | Useful characteristics | Important limitations |
|---|---|---|---|
| API key | Usually an application or integration | Simple to issue and send | Often carries little user context; must be treated as a secret |
| Bearer access token | User, application, or delegated caller | Can carry or reference permissions and expiration | Anyone who obtains it can generally use it while valid |
| Session credential | Signed-in browser user | Fits browser login and server-managed sessions | Requires browser-specific protections and lifecycle handling |

An **API key** is a secret value issued to identify an application or integration and, in some systems, grant it defined access. It is appropriate for some controlled integrations, but it is usually a weak basis for detailed user-level decisions unless combined with other identity information.

A **bearer access token** is a credential accepted based on possession: the caller that bears the token can use the authority represented by it. Access tokens should be short-lived enough to limit exposure, transmitted only over HTTPS, and validated for the intended API.

A **session credential** is a value, often stored in a browser cookie, that links requests to a server-managed signed-in session. Browser sessions can be secure, but they require controls appropriate to cookies and browser request behavior; they should not be treated as identical to service-to-service bearer tokens.

The API should define authentication failure consistently. A request with no acceptable credential, an expired credential, or a credential intended for another API normally fails before business processing. The client-visible response should not reveal secret validation details.

## OAuth 2.0 and OpenID Connect have different jobs

**OAuth 2.0** is a framework for granting an application limited access to protected resources, commonly through access tokens. **OpenID Connect (OIDC)** is an identity layer built on OAuth 2.0 that allows a client application to learn that a user has authenticated and receive identity information.

The practical distinction is:

- OAuth 2.0 helps a client obtain authority to call an API.
- OpenID Connect helps a client establish a signed-in user session and obtain identity claims.
- The API normally validates an access token; it should not accept an identity token merely because that token proves a user signed in to some client.

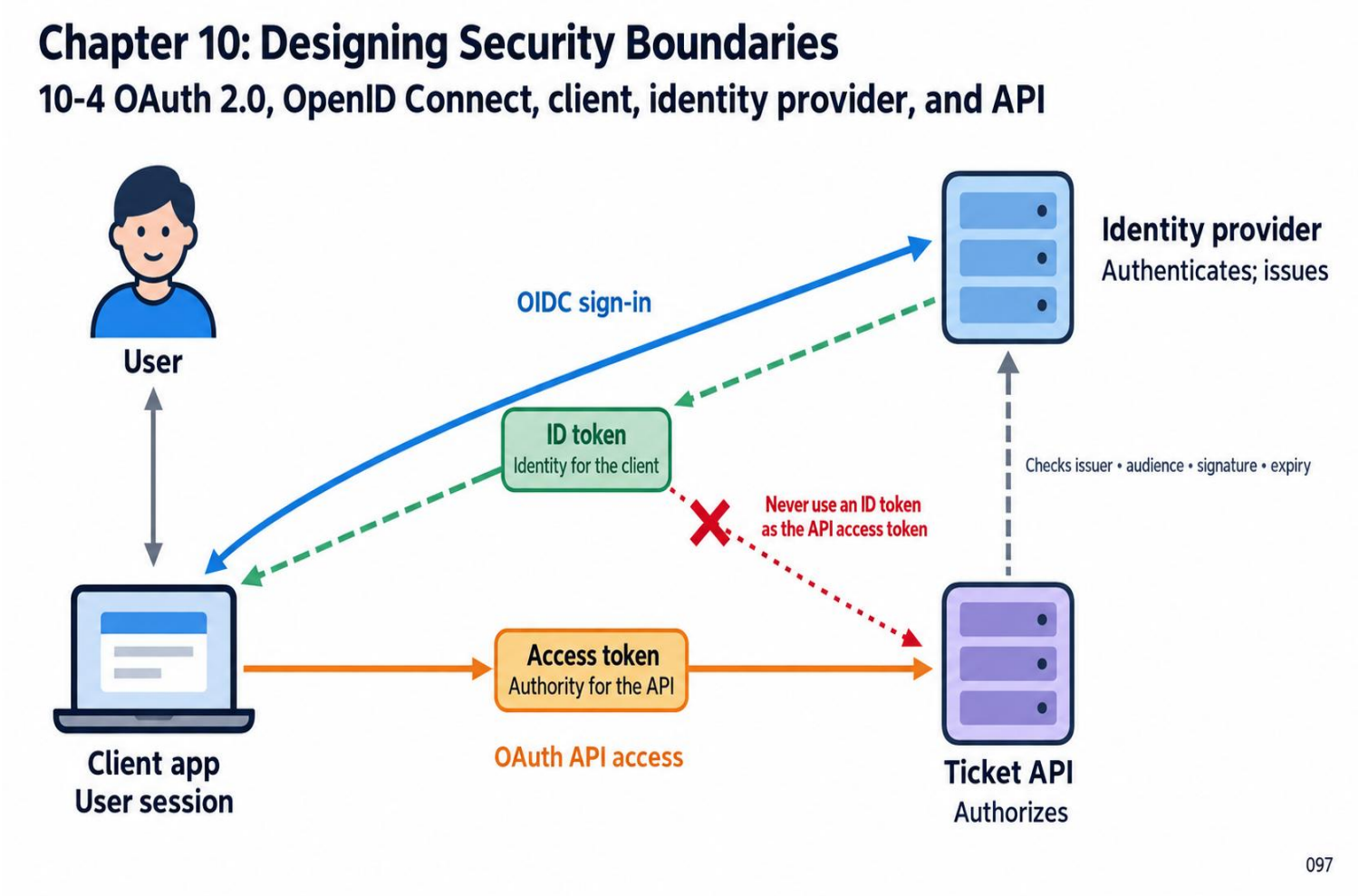



*Figure 10-4: OAuth 2.0, OpenID Connect, client, identity provider, and API*

A beginner does not need to implement these protocols from scratch. In project work, an identity provider, framework, gateway, or validated library usually handles protocol details. The API designer still must specify:

- Which caller types are supported.
- Which token audience is accepted; the **audience** identifies the API or resource server for which a token was issued.
- Which permissions or scopes are required.
- How expired or revoked credentials behave.
- Whether the API acts on behalf of a user, an application, or both.

**Established identity mechanisms should be integrated, not reinvented.** Custom token formats, home-grown signing schemes, and hand-written protocol implementations create risks that are difficult to review correctly.

## Authorization decides what the identity may do

**Authorization** is the decision about whether an authenticated identity may perform a specific action on a specific resource. It must be evaluated using current business rules and resource context, not merely the existence of a valid credential.

The ticket service has three human roles:

- A requester may create a ticket, read their own tickets, and add requester-visible comments.
- A support agent may read tickets in queues they serve, add internal or public comments, and assign eligible tickets.
- An administrator may manage queues and perform narrowly defined administrative actions.

A service identity may have a different set of capabilities. For example, the reporting service may read approved reporting fields but have no permission to retrieve attachments or change tickets.

A useful authorization decision follows this sequence:

1. Authenticate the caller.
2. Identify the requested operation.
3. Load or identify the target resource when needed.
4. Determine the caller's relevant permissions and relationship to the resource.
5. Check resource state and other policy conditions.
6. Permit or deny the operation.
7. Record security-significant outcomes where required.

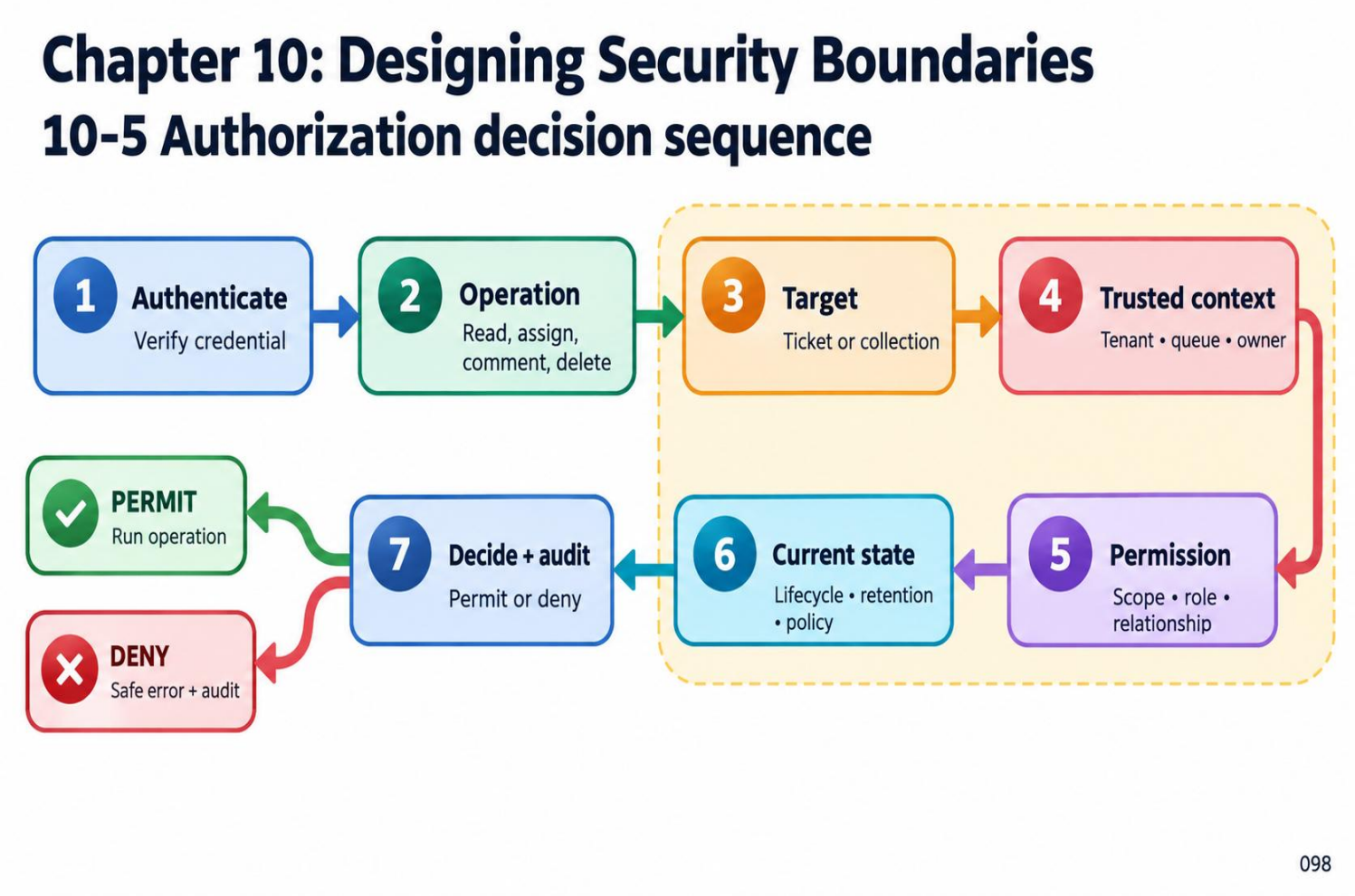



*Figure 10-5: Authorization decision sequence*

At the broad HTTP level, a missing or unacceptable credential normally produces `401 Unauthorized`, while an authenticated caller that lacks permission normally receives `403 Forbidden`. The API may conceal a protected resource with a consistent `404 Not Found` policy when revealing its existence would itself disclose information.

The order matters. A token may identify Agent A correctly, while the target ticket belongs to a queue that Agent A cannot access. Authentication succeeds; authorization fails.

## Scopes, roles, claims, and resource checks serve different purposes

A **scope** is a named permission or capability associated with delegated access, such as `tickets:read` or `tickets:assign`. A **role** is a named grouping of responsibilities, such as `support_agent`. A **claim** is an asserted piece of information about an identity or token, such as subject identifier, organization, role, or granted scope.

These values help make authorization decisions, but they rarely contain the entire decision.

Suppose a token contains:

```
{
  "sub": "usr_314",
  "roles": ["support_agent"],
  "scope": "tickets:read tickets:assign",
  "tenantId": "tenant_blue"
}
```

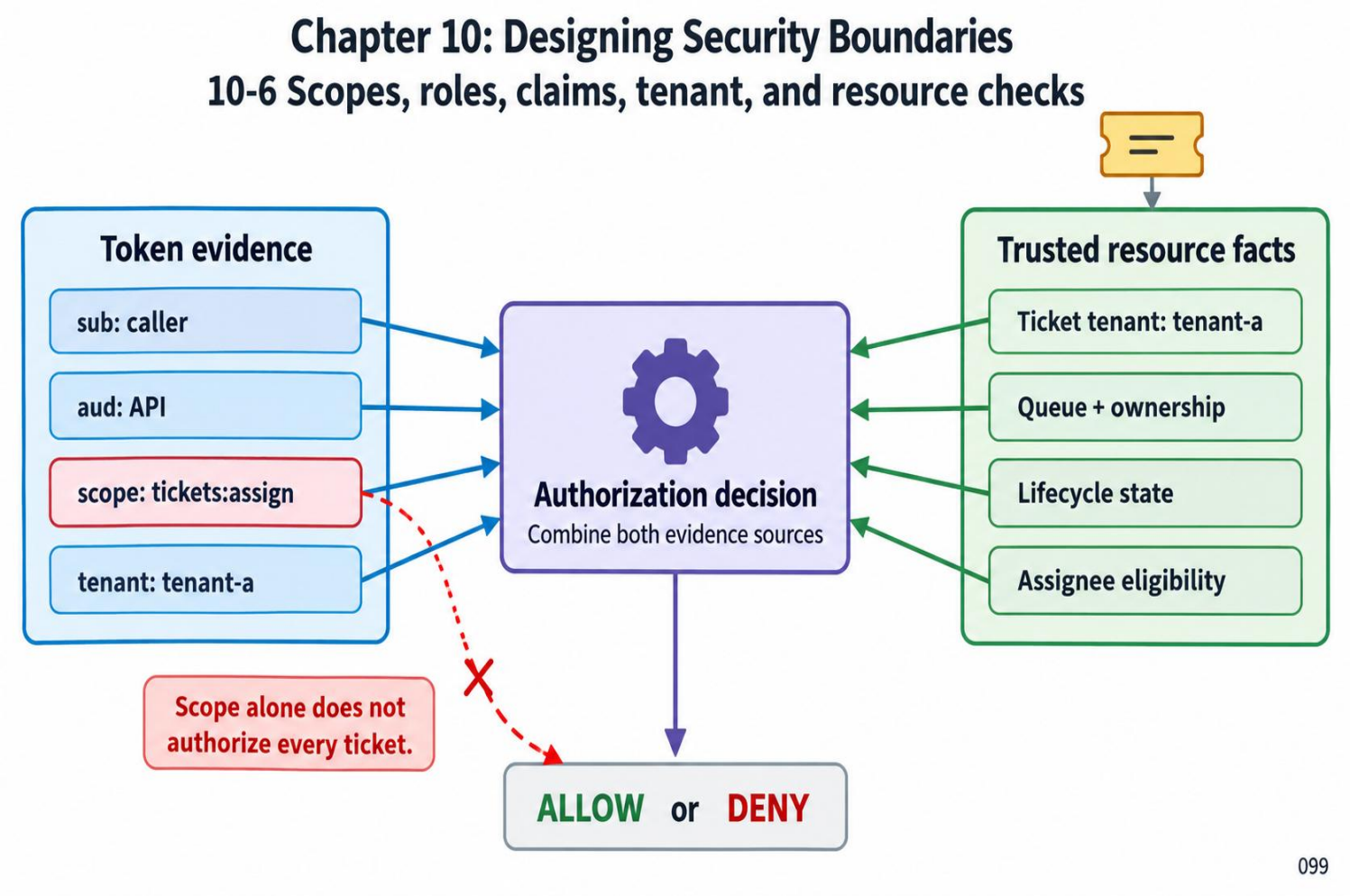



*Figure 10-6: Scopes, roles, claims, tenant, and resource checks*

The token suggests that the caller is an agent with assignment authority in `tenant_blue`. The API must still check that the requested ticket belongs to `tenant_blue`, that the agent may access its queue, and that the ticket is in an assignable state. Treating `tickets:assign` as permission to assign every ticket would ignore resource-level policy.

A **tenant** is a customer or organizational partition whose data and permissions must be isolated from other tenants in a shared system. Tenant identifiers supplied by a client are not proof of tenant membership. The provider must derive or validate tenant context from trusted identity and server-side resource data.

## Object-level authorization prevents cross-resource access

**Object-level authorization** checks whether the caller may access the particular record or resource identified by the request. It prevents a common failure in which a caller can change an identifier and reach another user's or tenant's data.

Consider:

```
GET /tickets/tkt_48271
Authorization: Bearer <access-token>
```

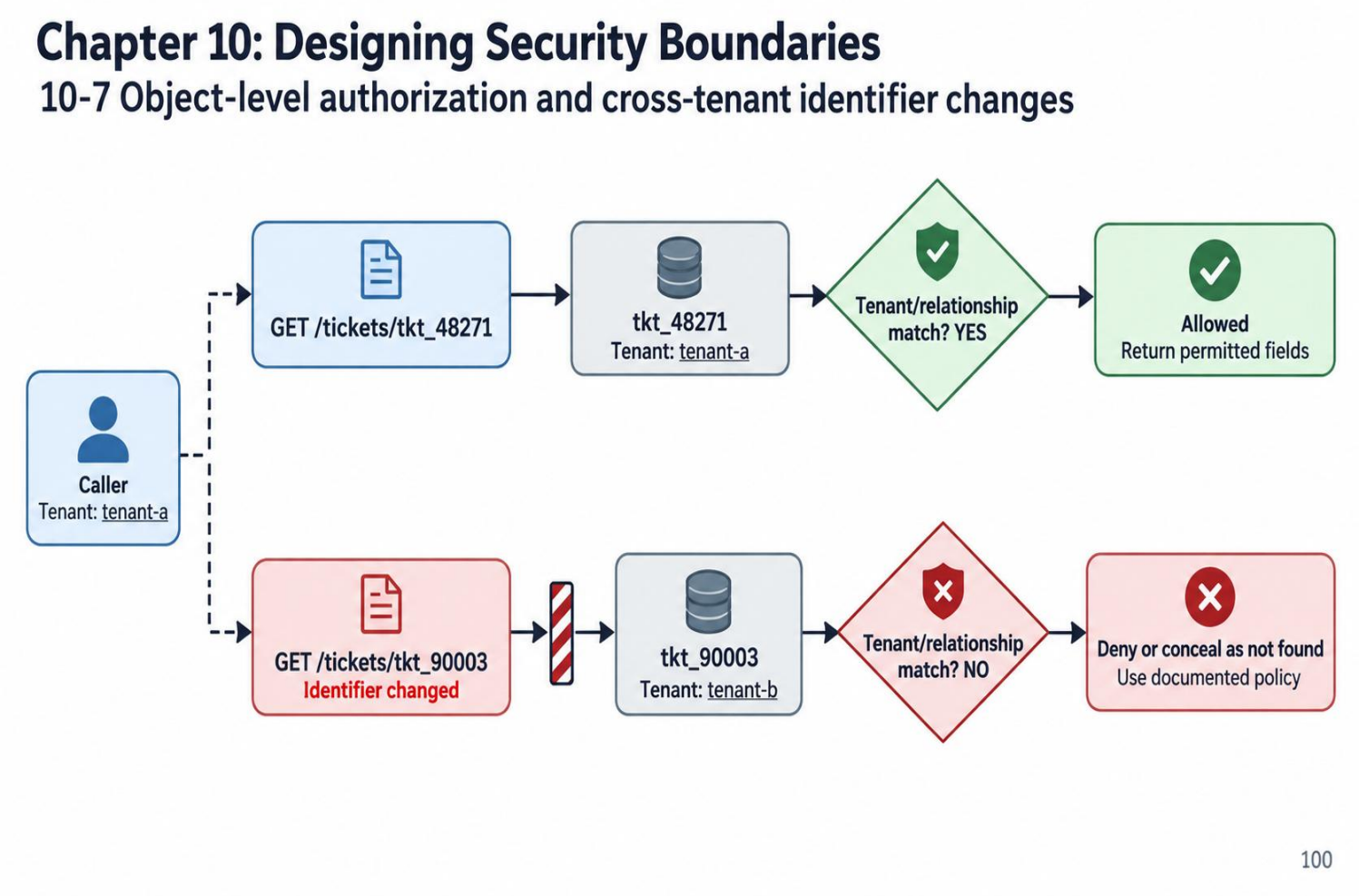



*Figure 10-7: Object-level authorization and cross-tenant identifier changes*

The provider must not stop after validating the token. It must answer questions such as:

- Does the ticket exist within the caller's permitted tenant?
- Is the requester the owner of this ticket?
- Is the agent assigned to an allowed queue?
- Is this field visible to this caller type?

The same principle applies to subordinate resources. Permission to read a ticket does not automatically imply permission to download every attachment, especially when attachments can contain restricted data.

Whether an unauthorized resource should produce `403 Forbidden` or be concealed with `404 Not Found` depends on the API's information-disclosure policy. The important requirement is consistency: the response must not allow unauthorized callers to discover protected records through differences in status, message, timing, or metadata.

## Least privilege limits the effect of mistakes and compromise

**Least privilege** means granting only the capabilities needed for the caller's task, for only as long and as broadly as necessary. It reduces damage when credentials are misused or an application contains a defect.

Examples for the ticket service include:

- The reporting service receives read access to reporting fields, not general ticket-update access.
- An automation job that creates tickets cannot delete them.
- Administrative deletion requires a separate capability from ordinary ticket editing.
- A temporary migration credential expires after the migration window.
- A user token is restricted to the user's tenant and permitted queues.

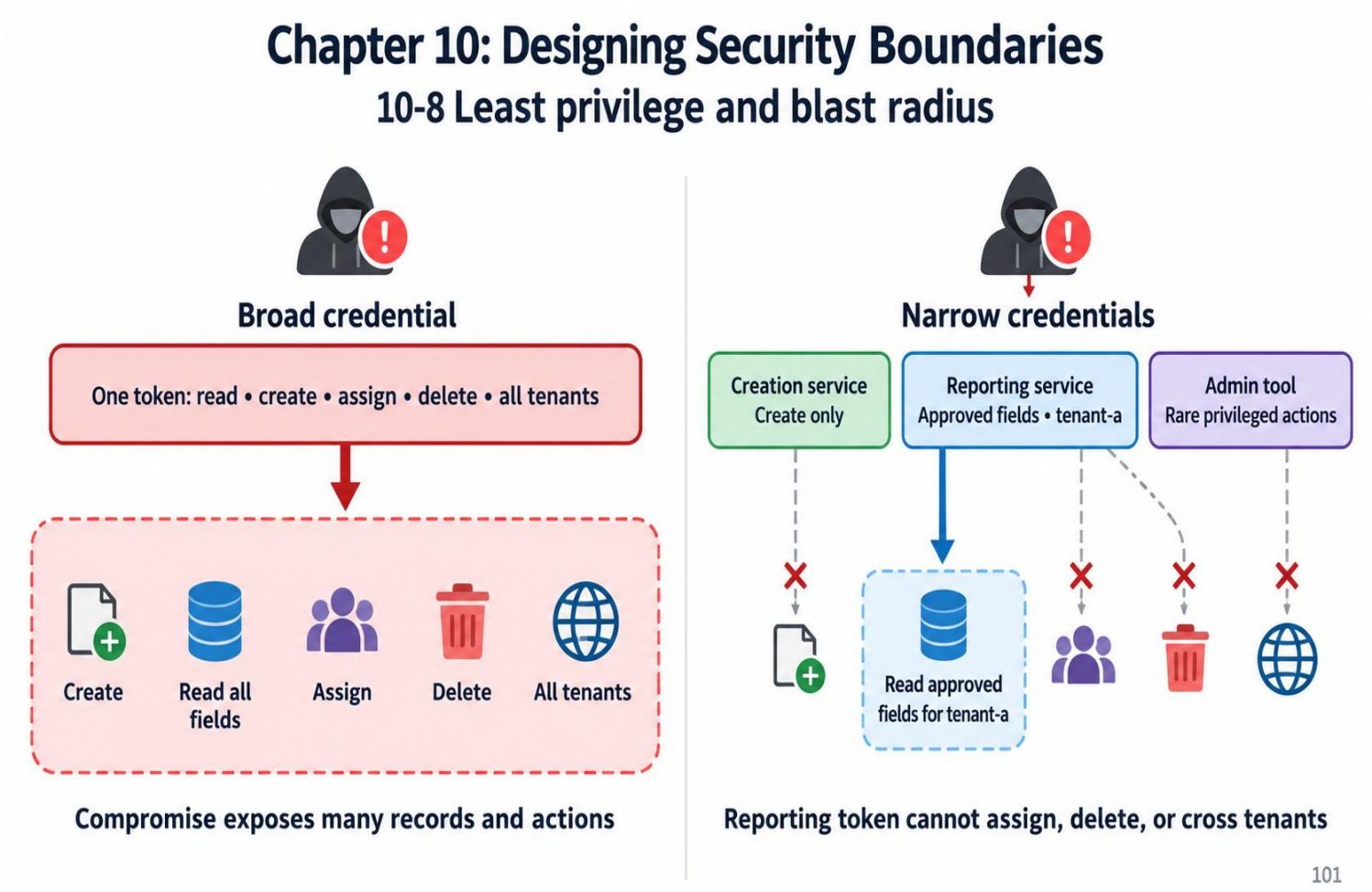



*Figure 10-8: Least privilege and blast radius*

Administrative operations deserve explicit separation because they commonly bypass ordinary restrictions or affect many resources. Hiding an administrative button in the user interface is not authorization; the provider must enforce the permission.

## Walk through four ticket operations

The following examples show how authentication and authorization combine. They are policy examples for this ticket system, not universal rules.

### Reading a ticket

1. Validate the credential and identify the caller.
2. Locate the ticket within the trusted tenant context.
3. Check whether the caller is the requester, an authorized agent, or an approved service.
4. Filter fields according to the caller's visibility rules.
5. Return the ticket or the API's consistent unauthorized-resource response.

Field filtering matters. A requester may be allowed to read the ticket but not an internal investigation note.

### Assigning a ticket

1. Authenticate the caller as a user or service allowed to act as an agent.
2. Require the assignment capability.

3. Confirm access to the ticket's tenant and queue.
4. Confirm that the target assignee is eligible for the queue.
5. Check the ticket's current lifecycle state and concurrency precondition.
6. Apply the assignment and record the actor.

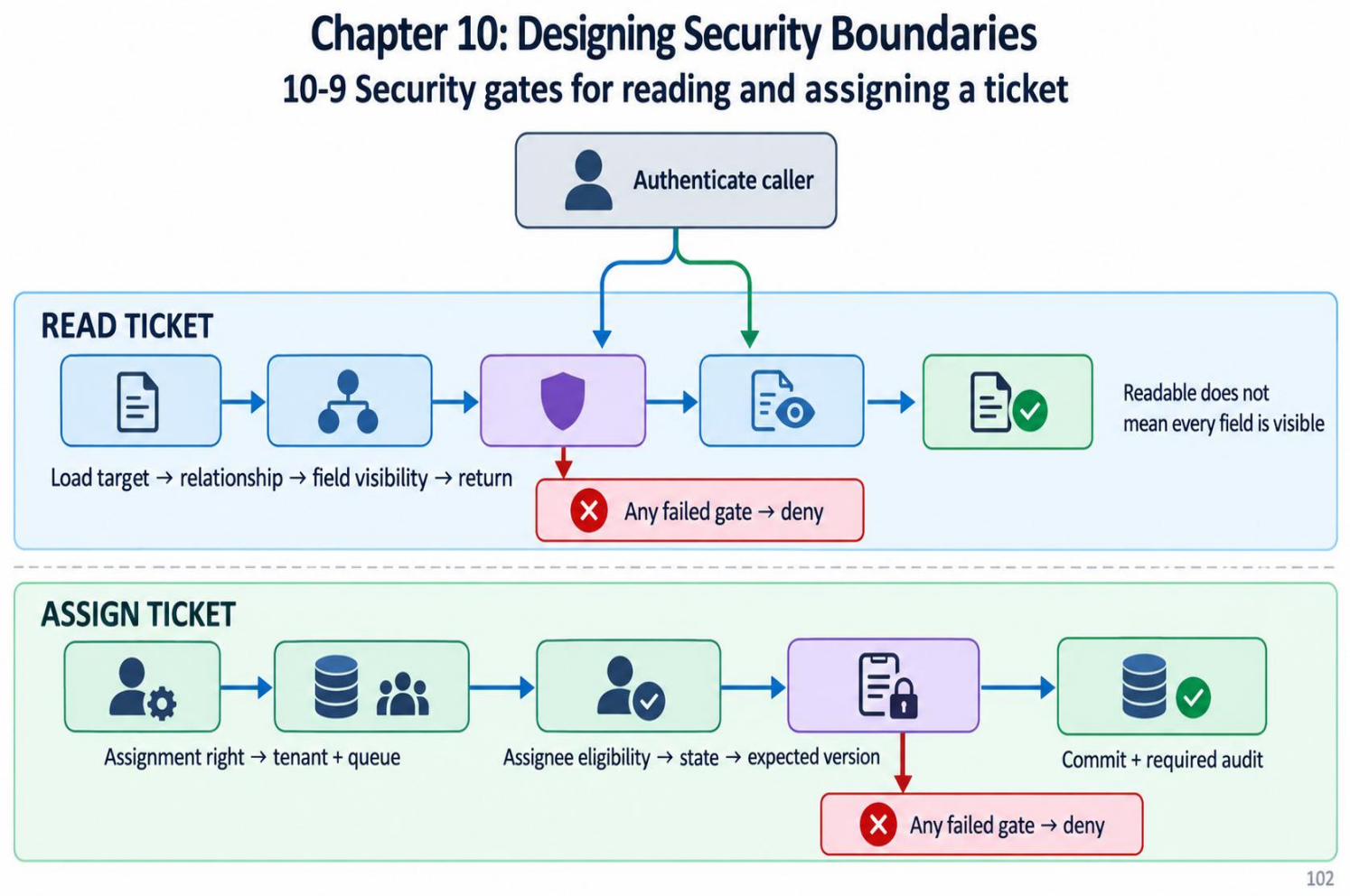



*Figure 10-9: –10.9.2 Security gates for reading and assigning a ticket*

The capability, the resource relationship, the state rule, and the concurrency rule are separate checks.

### Adding a comment

1. Authenticate the caller.
2. Determine whether the comment is public or internal.
3. Check permission for that visibility level.
4. Validate the content and attachment references.
5. Store the comment with trusted actor and timestamp information.

The client must not be allowed to claim another author by sending an arbitrary `authorId`. Actor identity is provider-controlled data.

### Deleting a ticket

1. Authenticate the caller.
2. Require a narrowly defined administrative capability.
3. Verify tenant and resource scope.
4. Check retention, legal, and lifecycle restrictions.
5. Record an auditable reason and actor.
6. Apply the documented deletion or archival behavior.

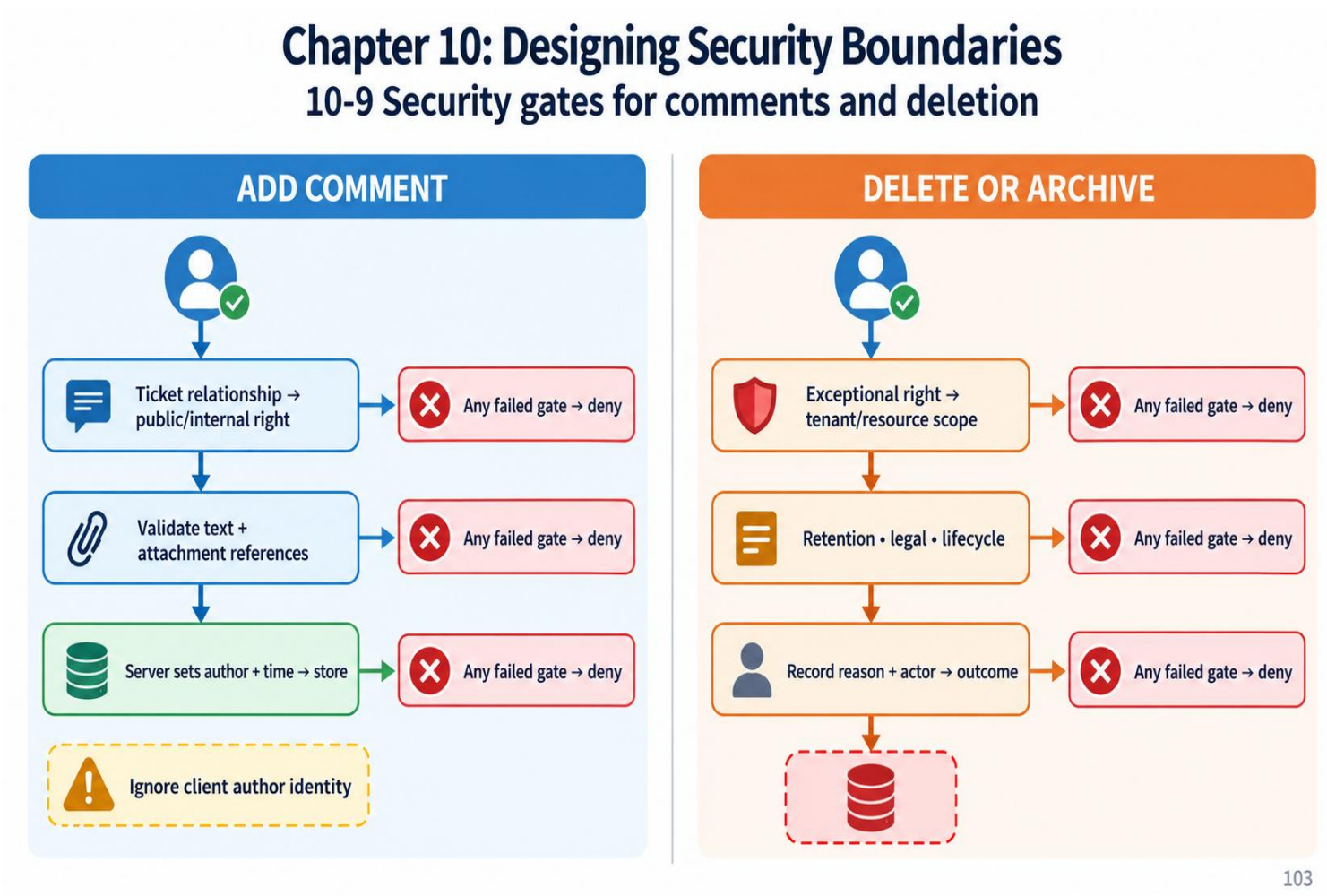



*Figure 10-10: –10.9.4 Security gates for comments and deletion*

Deletion is often more constrained than update. A generic `tickets:write` permission may be too broad.

## Input validation is not authorization

Input validation checks whether supplied data has an acceptable shape and value. It helps prevent malformed requests and reduces some injection risks, but it does not answer whether the caller may perform the operation.

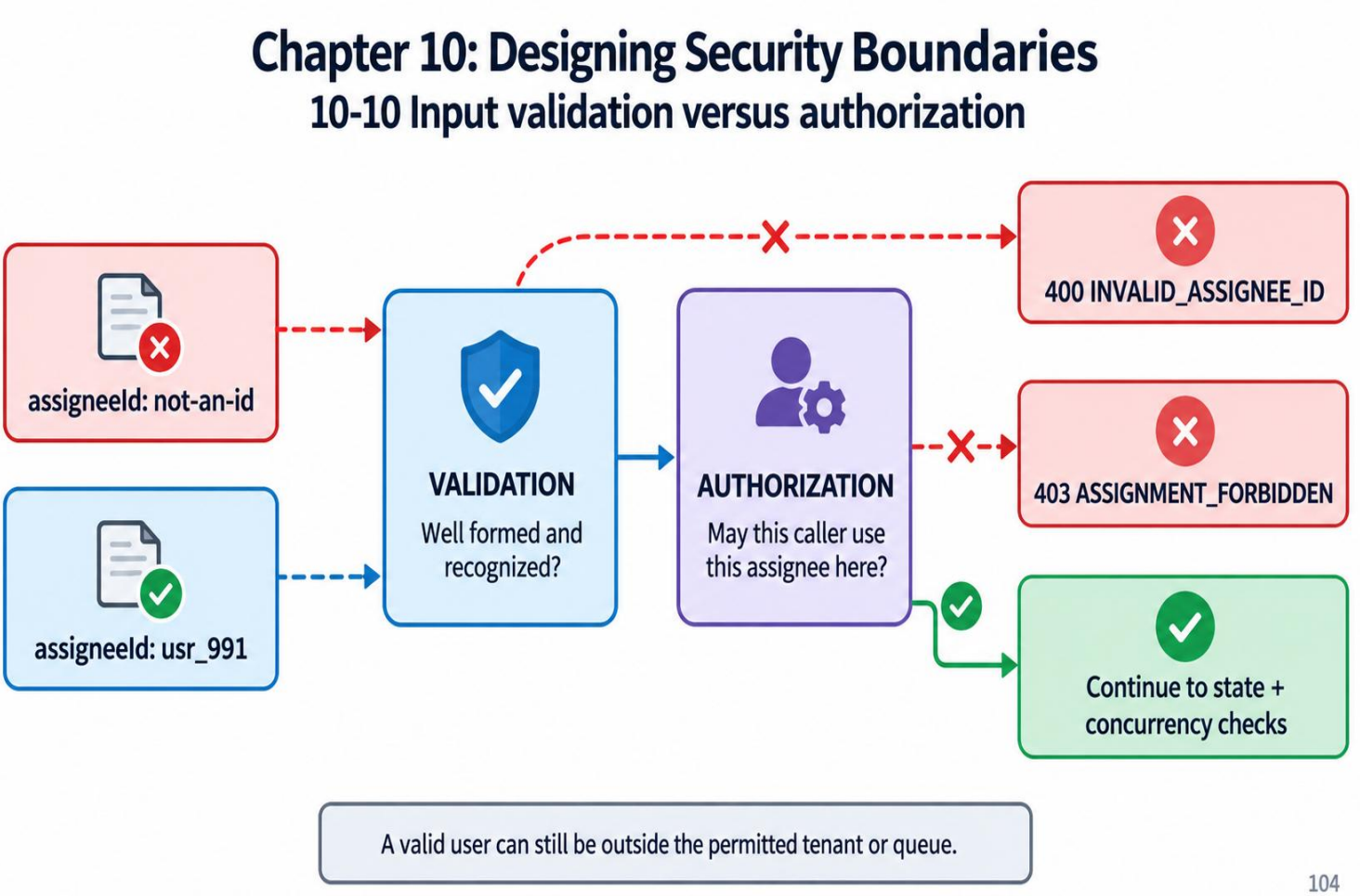



*Figure 10-11: Input validation versus authorization*

For example, validating that `assigneeId` matches the identifier format does not prove that the caller may assign the ticket or that the assignee belongs to the correct tenant. Both validation and authorization are required.

Similarly, a database query that returns no row because tenant criteria were included can support object-level authorization, but the policy must remain explicit and tested. Security should not depend on every developer remembering an undocumented query convention.

## Keep secrets and sensitive data out of exposed locations

A **secret** is a value whose confidentiality is required for security, such as an API key, signing key, client secret, or session credential. Secrets and sensitive personal data should not appear in URLs because URLs are commonly copied into logs, browser history, monitoring systems, and intermediary records.

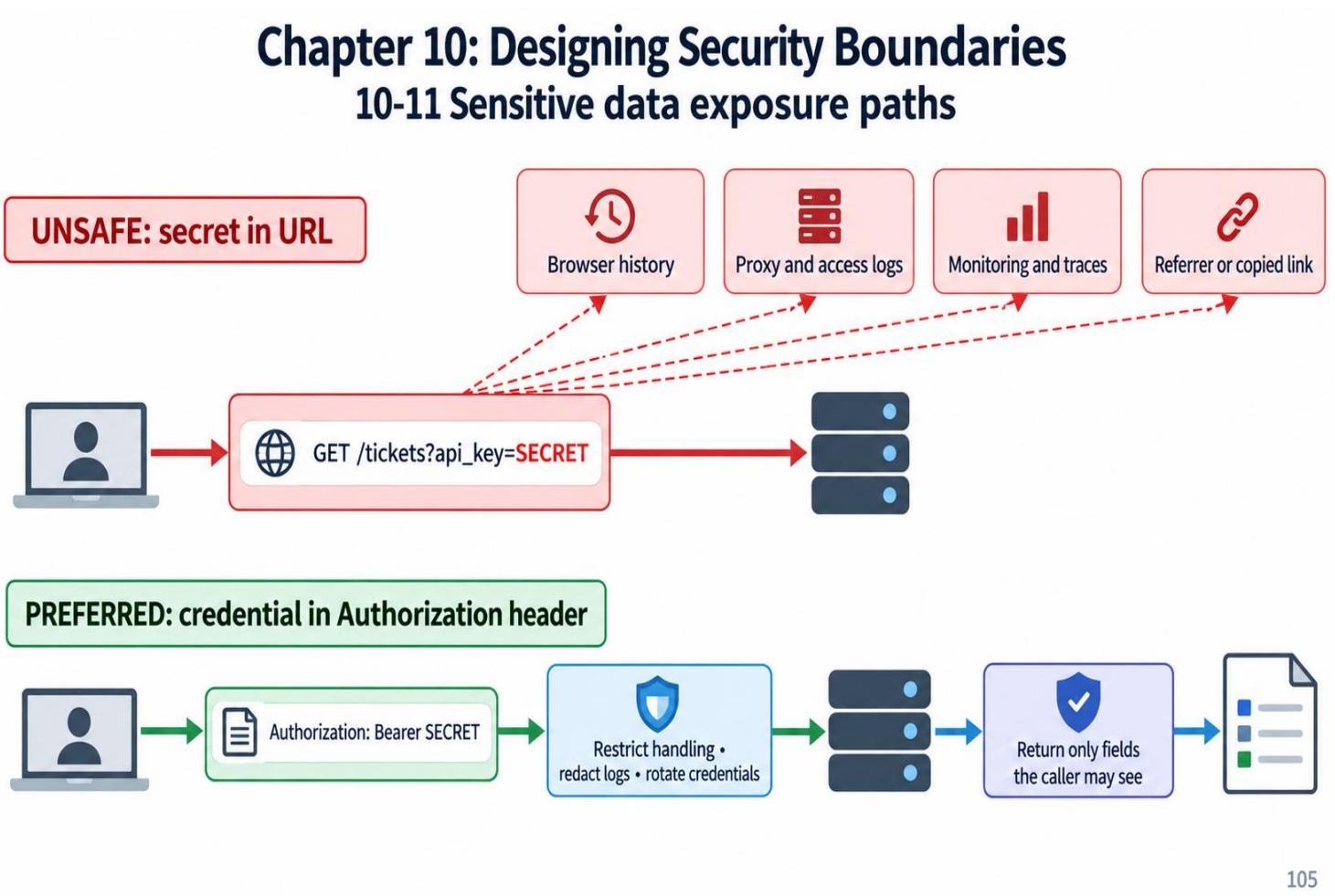



*Figure 10-12: Sensitive data exposure paths*

Prefer credentials in the designated authorization mechanism rather than query parameters:

```
Authorization: Bearer <access-token>
```

Do not return secrets in ordinary resource representations after creation. If a system must display a newly issued secret once, make that exceptional behavior explicit and ensure that later retrieval returns metadata rather than the secret value.

Error messages should not include token contents, stack traces, database queries, encryption keys, or personal data unrelated to correcting the request. Logs require the same discipline; moving sensitive data from a response into a log does not make it safe.

## Credentials need a lifecycle

Credential security includes more than initial issuance. A complete design addresses:

- **Storage:** Keep secrets in an approved secret store, not source code or ordinary configuration files committed to a repository.
- **Rotation:** Replace credentials without requiring a dangerous all-at-once outage.
- **Expiration:** Limit how long a credential remains useful.
- **Revocation:** Provide a way to invalidate compromised or obsolete credentials where the mechanism supports it.
- **Ownership:** Record which team or system owns each credential.
- **Observation:** Detect unusual failures, usage locations, or request patterns without logging the credential itself.

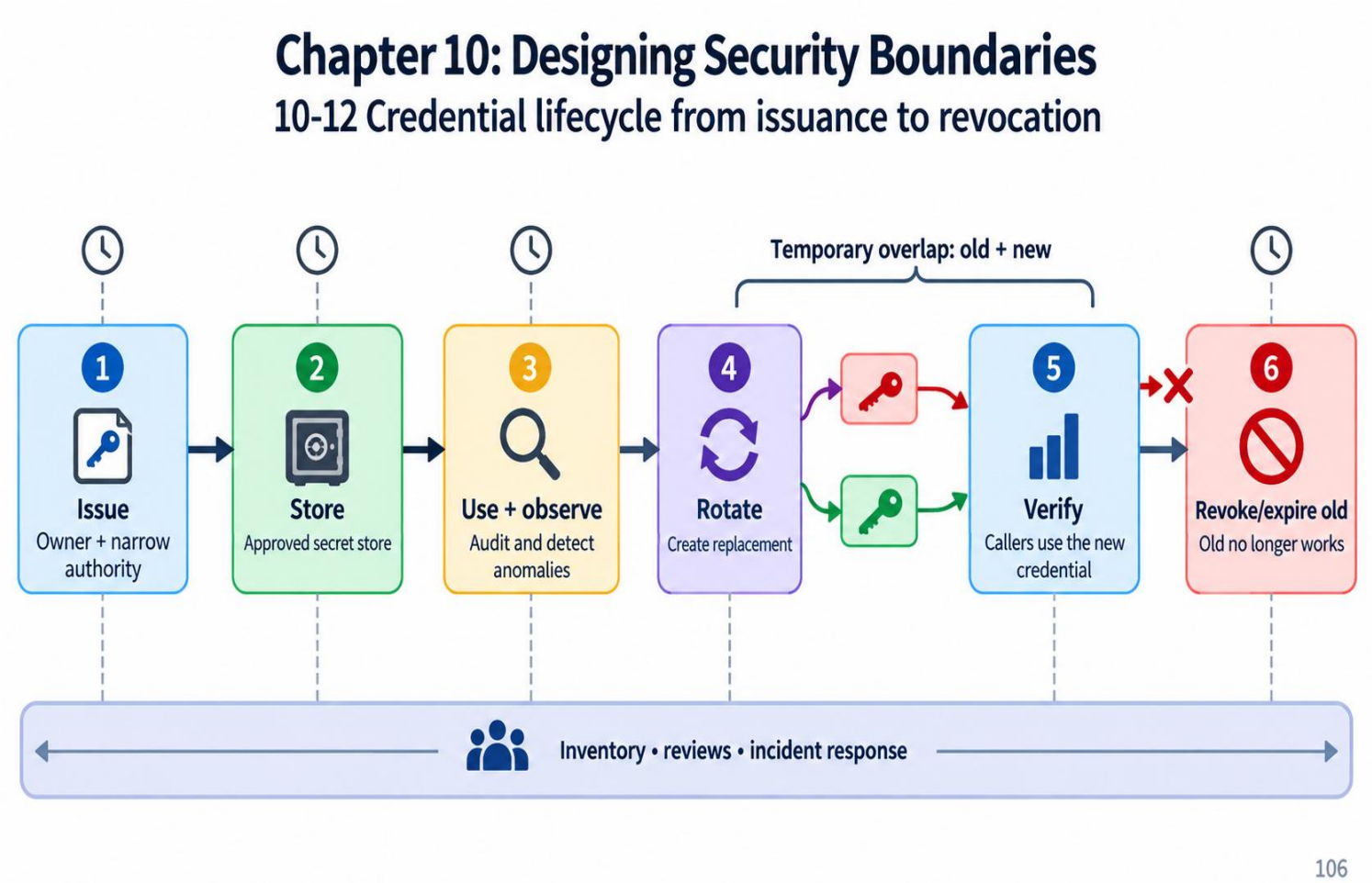





*Figure 10-13: Credential lifecycle from issuance to revocation*

For long-lived integrations, overlap during rotation may be necessary: issue a new credential, update the consumer, verify use, and then revoke the old credential. The exact process depends on the identity platform, but the API contract and operational plan should not assume secrets live forever.

## CORS controls browsers, not general API authorization

**Cross-Origin Resource Sharing (CORS)** is a browser mechanism that controls whether script from one web origin may read responses from another origin. An **origin** is the combination of scheme, host, and port used by a browser security boundary.

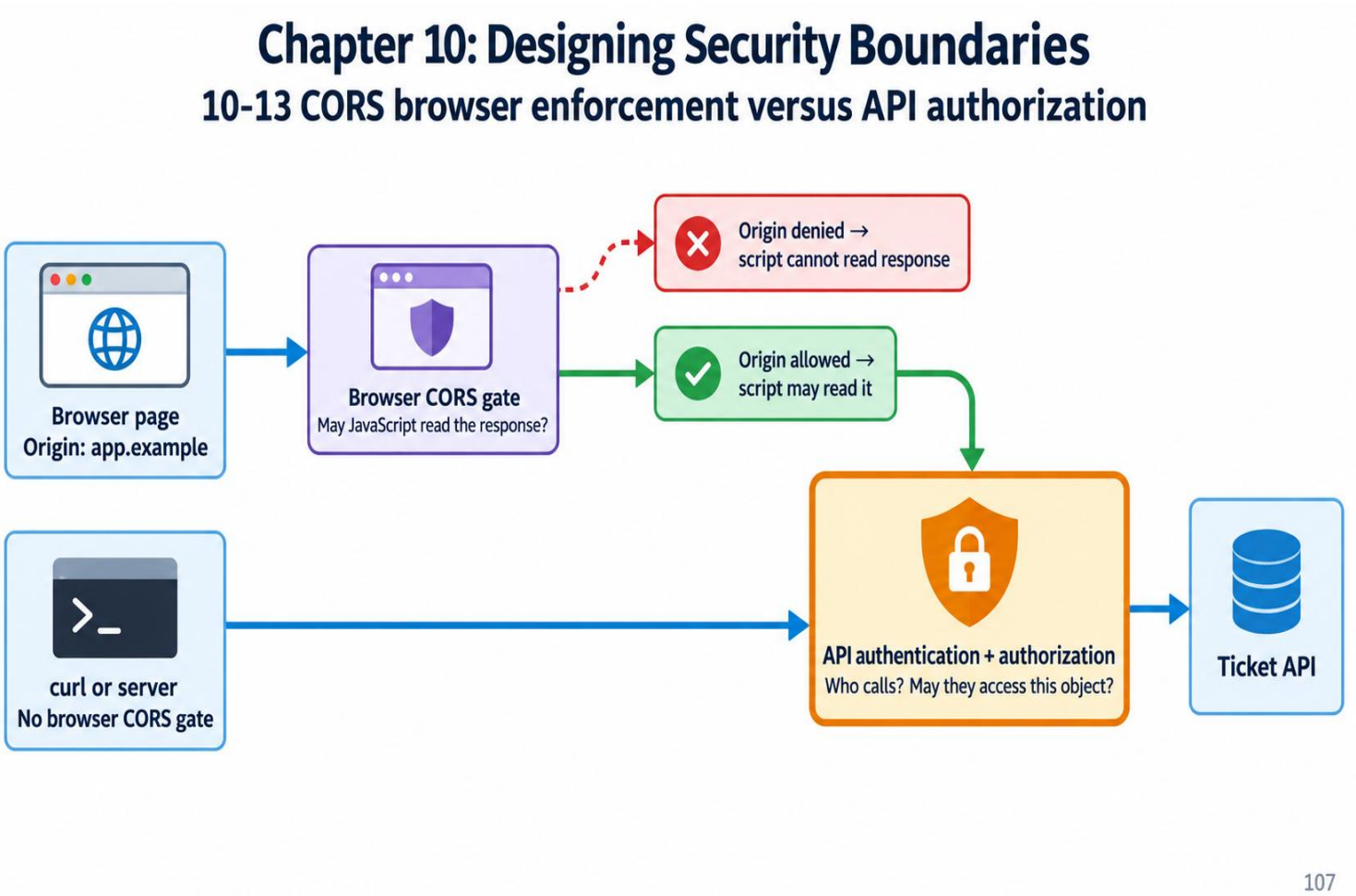



*Figure 10-14: CORS browser enforcement versus API authorization*

CORS can allow the agent web application at one origin to call the ticket API at another. It does not prevent `curl`, a server program, or malicious software from sending requests. Those clients are not required to enforce browser CORS rules.

Therefore:

- CORS configuration does not authenticate a caller.
- An allowed origin does not prove a user's permission.
- A denied origin does not protect the API from non-browser traffic.
- Authentication and authorization remain mandatory.

When browser sessions use cookies, additional browser-request protections may be required. Those details depend on the session architecture; they should be reviewed with the security design rather than inferred from CORS alone.

## Rate limits and abuse controls complement permission checks

**Rate limiting** restricts how many requests a caller may make within a defined period or capacity budget. It protects service availability and can reduce automated abuse, but it does not turn an unauthorized action into an authorized one.

Useful abuse controls can include:

- Per-credential or per-tenant request limits.
- Stricter limits on expensive search or export operations.
- Maximum payload, page, and batch sizes.
- Detection of repeated authentication failures.
- Quotas for sustained usage over longer periods.
- Manual or automated suspension procedures for compromised integrations.

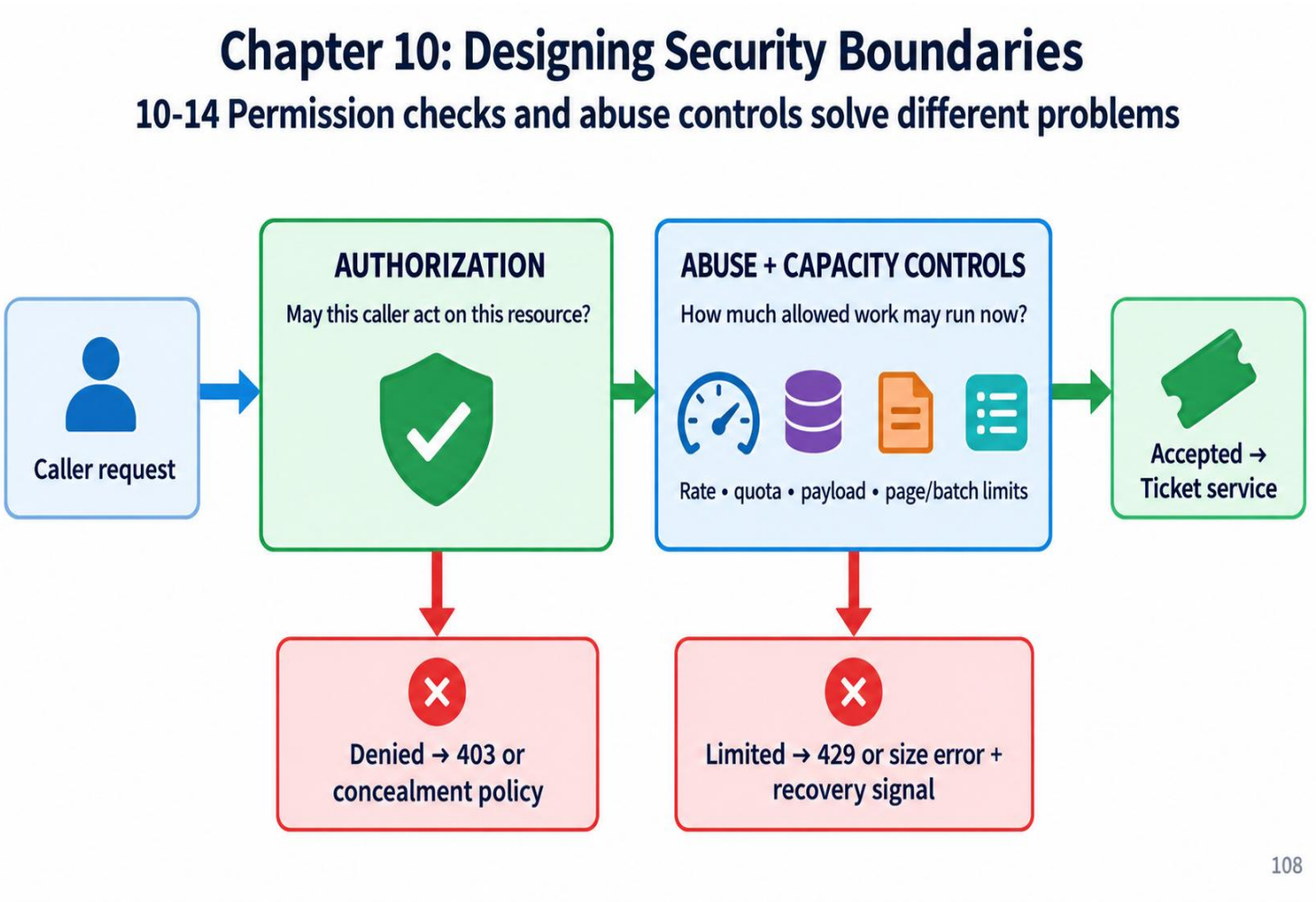



*Figure 10-15: Permission checks and abuse controls solve different problems*

The contract should identify how a limited request is reported and whether the caller receives retry guidance. Detailed retry behavior is addressed with reliability in Chapter 14.

## Two security mistakes with large consequences

### "A valid access token means the caller may access every resource"

A valid token establishes accepted identity and perhaps broad capabilities. The provider must still enforce tenant, ownership, queue, field, state, and action restrictions for the requested resource. Missing object-level checks commonly produce cross-user or cross-tenant exposure.

### "CORS protects an API from unauthorized non-browser clients"

CORS is enforced by cooperating browsers. A non-browser caller can send the same network request regardless of the CORS policy. The API must authenticate and authorize every protected operation itself.

## Security boundary review

Before approving an operation, verify that:

- Protected traffic uses HTTPS directly.
- The supported caller identities and credential types are explicit.
- The API validates that a token or credential is intended for this API.
- Authentication and authorization failures have defined behavior.
- Every protected operation checks action and resource-level permission.
- Tenant and ownership context comes from trusted information.
- Administrative capabilities are separated from ordinary access.
- Client-supplied identifiers cannot bypass object-level controls.

- Secrets and sensitive data are excluded from URLs, errors, and routine logs.
- Credential expiration, rotation, revocation, and ownership are planned.
- CORS is treated as browser policy rather than authentication.
- Rate limits and abuse controls support, rather than replace, authorization.

**A secure API does not ask only whether the caller has a credential. It asks who the caller is, what authority was granted, which resource is involved, and whether the requested action is permitted now.**

## Chapter Summary

- HTTPS protects transport but does not decide who may perform an operation or access a specific resource.
- Authorization must be evaluated for the requested action and object, not inferred from possession of a valid token.
- Secrets, credentials, sensitive fields, rate controls, and audit needs require explicit lifecycle and boundary decisions.

## Review Questions

1. What is the difference between proving identity and authorizing access to one ticket?
2. Why can a scope or role be necessary but still insufficient for object-level authorization?
3. Why does CORS not protect an API from unauthorized non-browser clients?

# Part IV — Making the Contract Usable and Verifiable

# Chapter 11 — Describing and Documenting the Contract

**An API contract should be precise enough for tools to process and clear enough for people to understand why the API behaves as it does.** Machine-readable descriptions provide structure; explanatory text and examples provide meaning that structure alone cannot express.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Describe an HTTP API contract with OpenAPI while recognizing what structured schemas cannot express alone.
- Use examples, prose, reusable components, and security definitions without hiding operation-specific meaning.
- Keep descriptions, implementation, tests, changelogs, and migration guidance aligned.

## Documentation has two audiences

Human readers need to understand tasks, permissions, lifecycle rules, failure recovery, and examples. Tools need structured definitions of paths, parameters, message schemas, responses, and security requirements.

These needs overlap but are not identical. A field can be declared as a string in a schema while still leaving important questions unanswered:

- Who sets it?
- May it change after creation?
- Which values are meaningful in each ticket state?
- Is it visible to all callers?
- What should a client do when it is absent?

Good API documentation therefore combines:

- A machine-readable API description.
- Task-oriented human guidance.
- Representative requests and responses.
- Error and recovery behavior.
- Authentication and authorization instructions.
- Compatibility, deprecation, and release information.

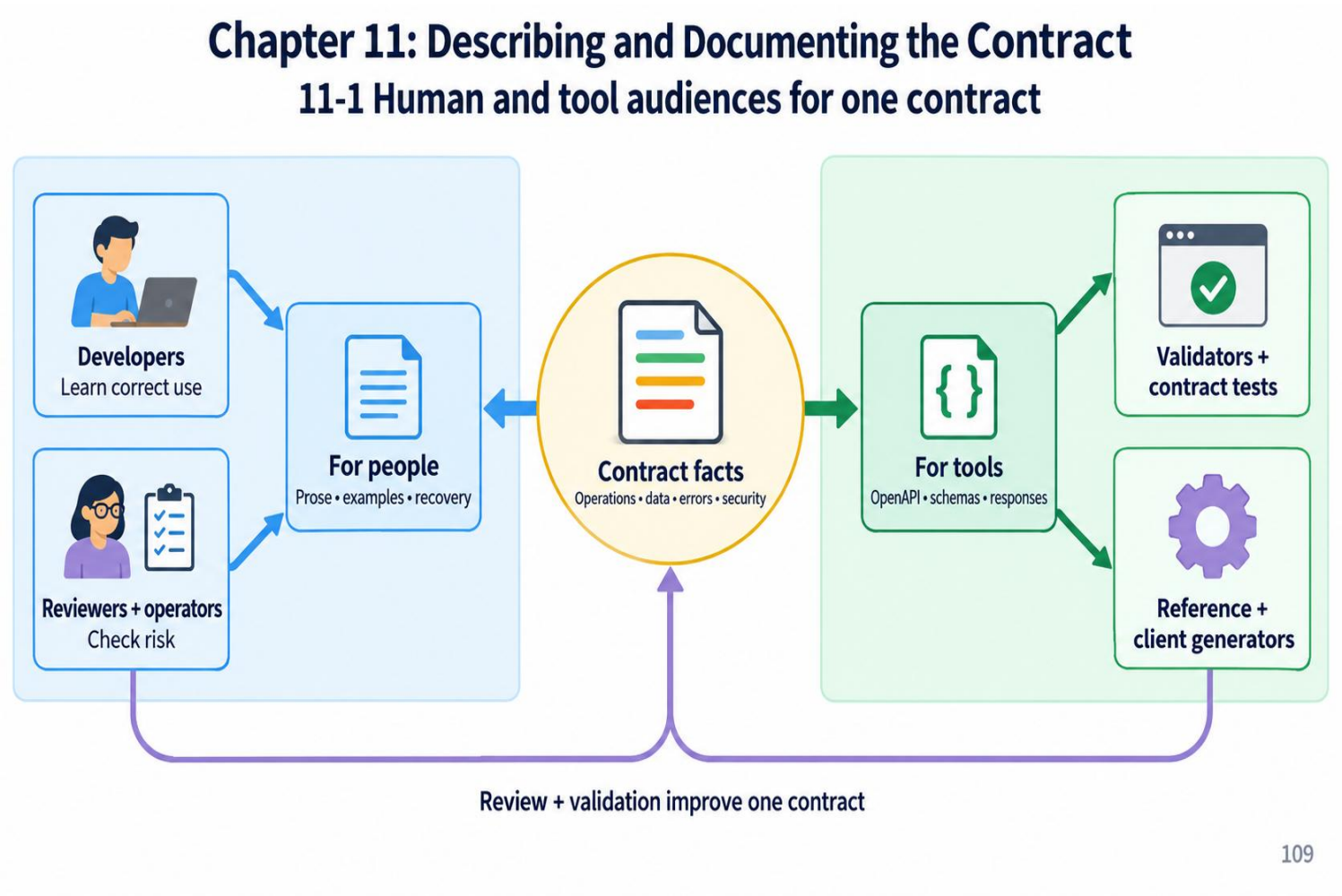



*Figure 11-1: Human and tool audiences for one contract*

The goal is not to create more documents. The goal is to create one coherent contract expressed in forms appropriate to its readers.

## OpenAPI describes HTTP API structure

**OpenAPI** is a widely used, machine-readable description format for HTTP APIs. An OpenAPI document can describe available paths, operations, parameters, request bodies, responses, schemas, and security requirements.

The main structural elements are:

- A **path**, such as `/tickets/{ticketId}`, identifies an address pattern.
- An **operation** describes an HTTP method at a path, such as `GET` on that path.
- A **parameter** supplies input through a path, query string, header, or cookie.
- A **request body** describes content sent as the HTTP message body.
- A **response** describes a possible status and its headers or body.
- A **schema** describes the allowed structure and values of data.
- A **security scheme** describes how credentials are supplied.

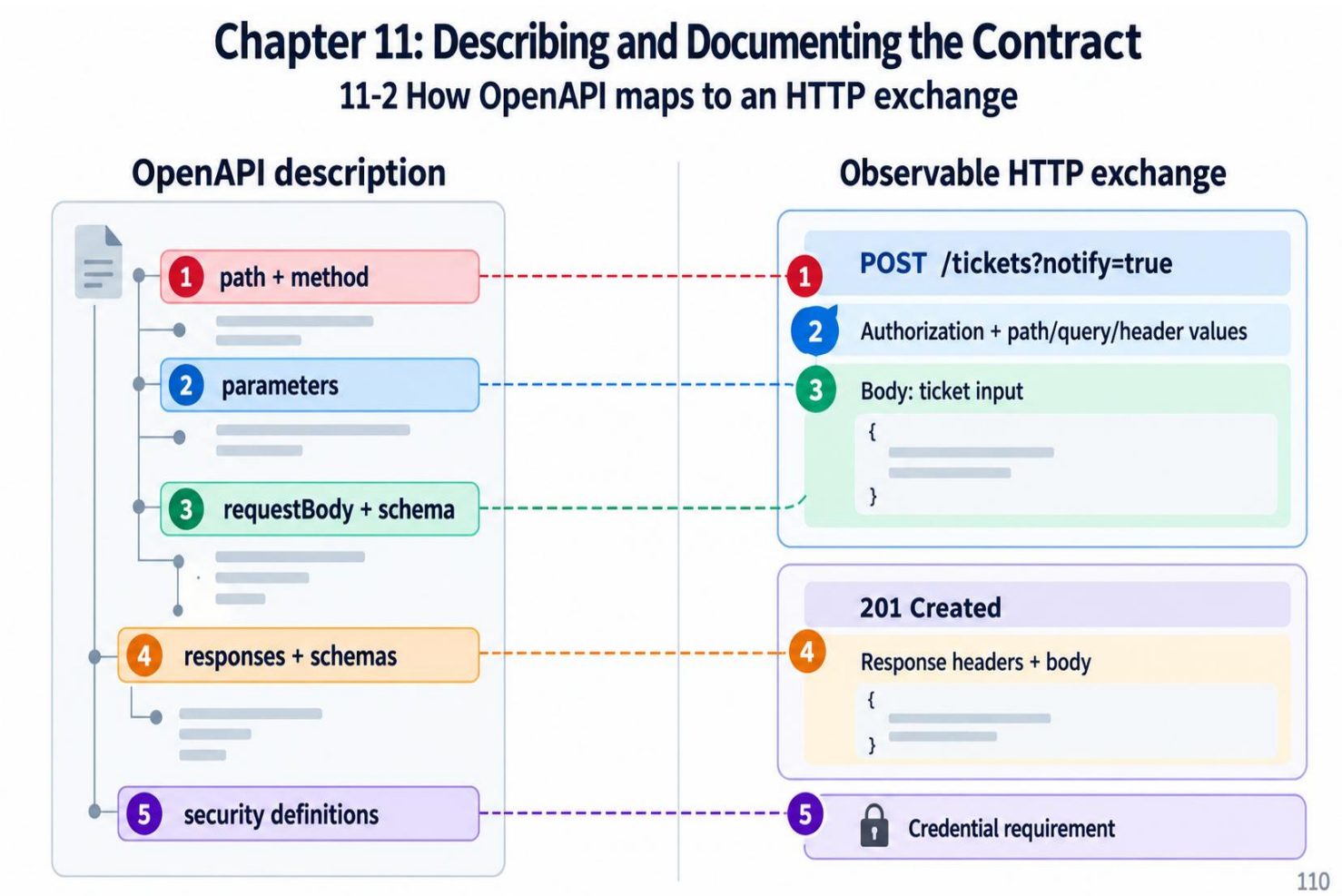



*Figure 11-2: How OpenAPI maps to an HTTP exchange*

OpenAPI records the shape of the contract. It does not decide whether the resource model, permission policy, or state transitions are good.

## A minimal ticket description

The following OpenAPI example describes one ticket retrieval operation. It is deliberately small so that each part has a clear purpose.

```
openapi: 3.1.0
info:
  title: Ticket API
  version: 1.0.0
paths:
  /tickets/{ticketId}:
    get:
      operationId: getTicket
      summary: Retrieve one ticket
      security:
        - bearerAuth: []
      parameters:
        - name: ticketId
          in: path
          required: true
          schema:
            type: string
      responses:
        "200":
          description: The ticket was found and is visible to the caller.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Ticket"
```

```
        "404":
          description: No visible ticket was found for the identifier.
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
  schemas:
    Ticket:
      type: object
      required: [id, subject, status]
      properties:
        id:
          type: string
          example: tkt_48271
        subject:
          type: string
          maxLength: 200
        status:
          type: string
          enum: [open, in_progress, resolved, closed]
```

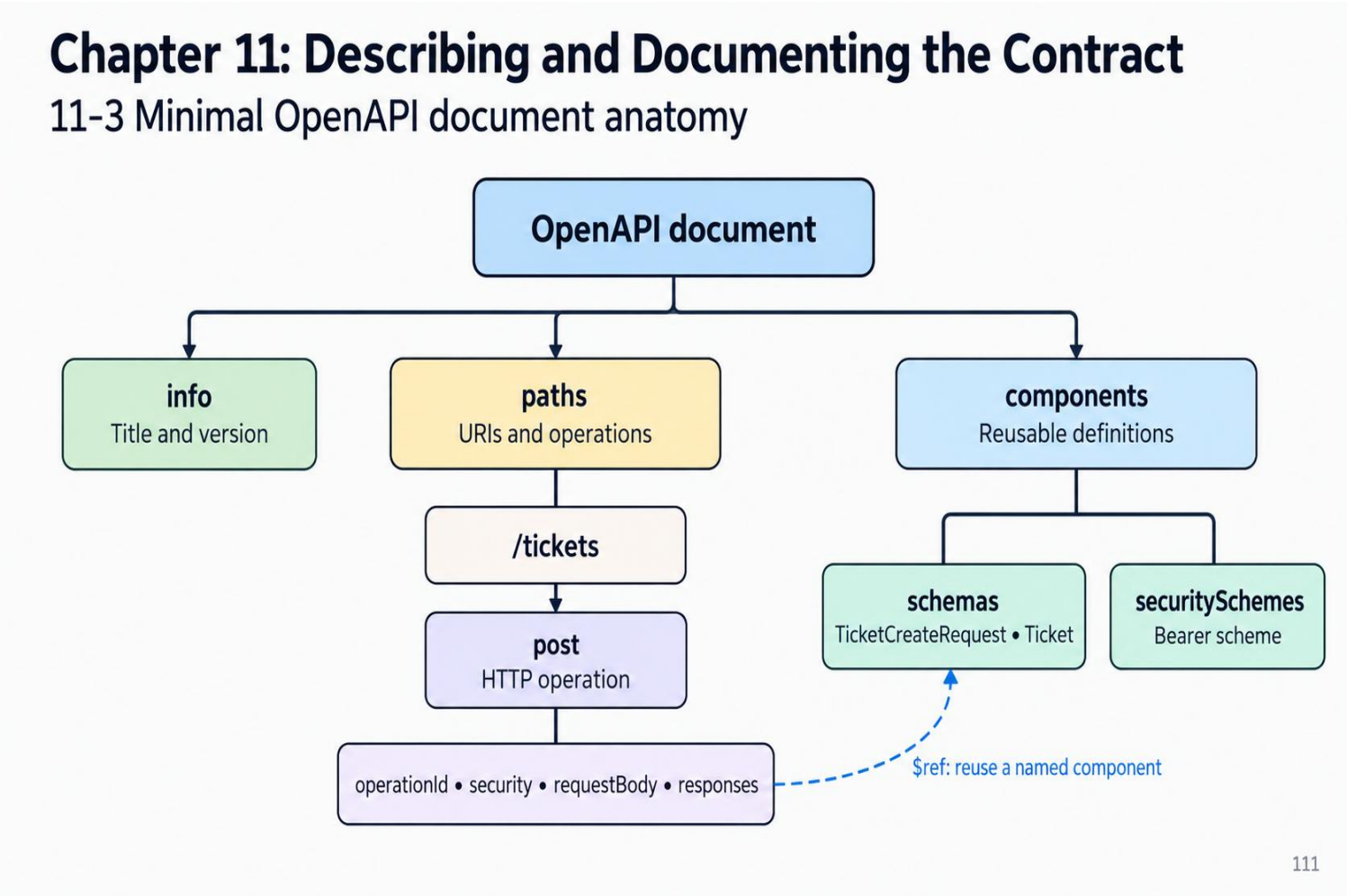



*Figure 11-3: Minimal OpenAPI document anatomy*

The lines that matter most are:

- `operationId: getTicket` gives the operation a stable tool-friendly name.
- `security` states that the bearer-token scheme applies.
- The path parameter is required because the URI cannot be formed without it.
- The `200` response points to a reusable `Ticket` schema.
- The `404` description reflects the API's policy of not distinguishing missing tickets from tickets invisible to the caller.
- `required` lists fields guaranteed in this representation; it does not mean every ticket field must be present in every operation.

An **operation identifier** is a unique name for an API operation that tools may use when generating client methods, server interfaces, or test references. Once consumers depend on generated names, changing an operation identifier can create unnecessary source-code churn even when the wire contract is unchanged.

## A schema is necessary but not sufficient

The example schema can validate that `status` is one of four values. It cannot fully express statements such as:

- Only agents may move a ticket from `open` to `in_progress`.
- A requester may not see internal comments.
- A closed ticket cannot be reassigned.
- The response may omit fields that the caller is not permitted to view.
- A stale `If-Match` condition fails before the update is applied.




*Figure 11-4: Schema boundary versus behavioral rules*

These are behavioral rules. They belong in operation descriptions, state-transition documentation, permission tables, examples, or related design material.

A useful test is: **Could a careful reader implement a correct client using only the schema?** If not, document the missing behavior near the relevant operation rather than assuming that consumers will infer it.

## Use examples to make abstract rules concrete

Examples help readers understand how several contract rules work together. A good example is valid, representative, and consistent with the schema.

For ticket creation, one example might show:

```
{
  "subject": "Unable to reset password",
  "description": "The reset link reports that it has expired.",
  "priority": "normal"
}
```

A successful response might show the provider-controlled fields that appear after creation:

```
{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "description": "The reset link reports that it has expired.",
  "priority": "normal",
  "status": "open",
  "createdAt": "2026-09-18T09:42:31Z"
}
```

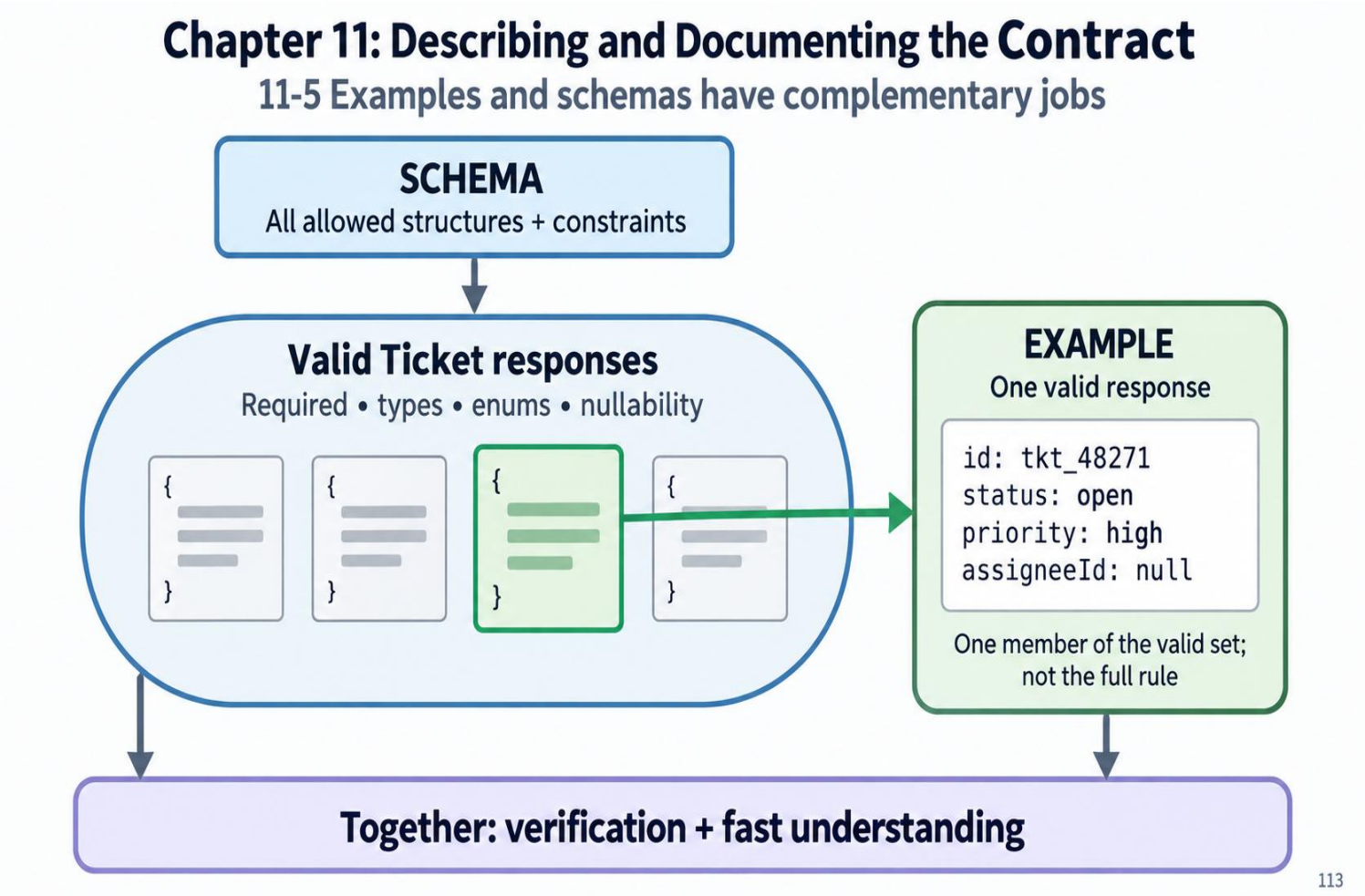



*Figure 11-5: Examples and schemas have complementary jobs*

The example teaches that the client supplies the subject, description, and requested priority, while the provider supplies the identifier, initial status, and creation timestamp.

Examples are not substitutes for constraints. A single example cannot prove maximum length, nullability, all allowed enumeration values, or every possible response. Use the schema for enforceable structure and examples for understanding.

## Describe formats, defaults, and constraints precisely

Documentation should distinguish several ideas that are often blurred:

- A **format** describes an expected representation, such as a date-time string.
- A **constraint** limits acceptable values, such as a maximum length or numeric range.
- A **default** is a value the provider applies when the client omits input, when the contract explicitly defines that behavior.

- An **example** is an illustrative value and does not itself create a rule.

If omitted priority becomes `normal`, say so in both the human description and the structured definition where the chosen OpenAPI usage accurately reflects server behavior. Do not add a default merely to improve generated documentation if the server does not actually apply it.

Descriptions should explain meaning rather than restate the type. “The ticket status” adds little. “The current lifecycle state; clients must tolerate documented future values” helps the reader make a safer implementation choice.

## Choose contract-first or code-first consciously

A **contract-first workflow** begins by designing and reviewing the API description before the provider implementation is complete. A **code-first workflow** derives much of the API description from implementation declarations, annotations, or runtime metadata.

| Workflow | Main advantage | Main risk | Suitable use |
|---|---|---|---|
| Contract-first | Consumers and reviewers can examine behavior before implementation commits the design | The description can drift if implementation is not checked against it | Cross-team or externally consumed APIs where early agreement matters |
| Code-first | Description can stay close to application types and routes | Internal code structure can leak into the public contract | Small, controlled APIs with strong review and conformance checks |

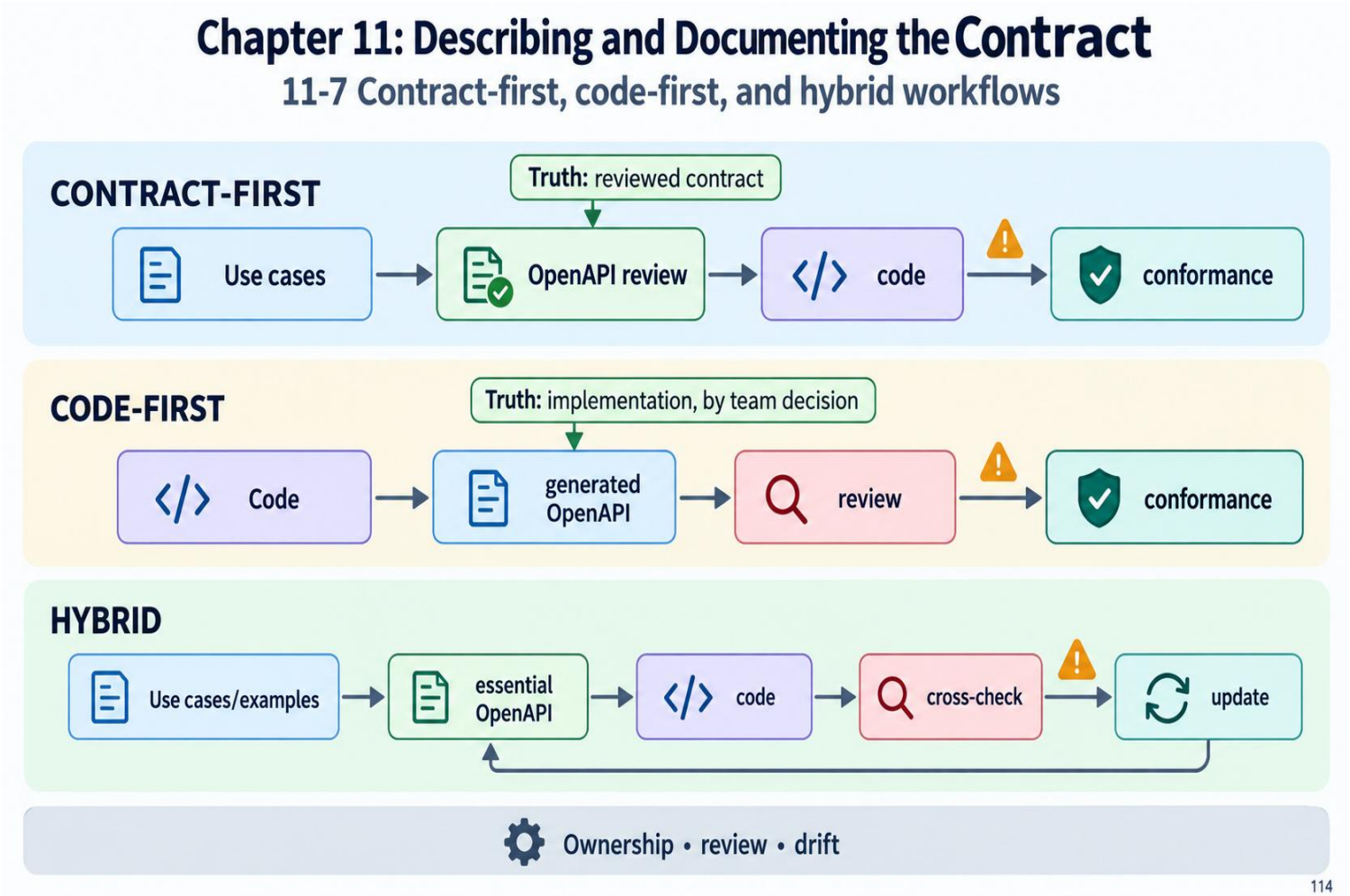



*Figure 11-6: Contract-first, code-first, and hybrid workflows*

Neither workflow guarantees quality. A contract-first team can produce a vague contract. A code-first team can produce a coherent one if it treats generated output as a reviewable artifact rather than an automatic truth.

A practical hybrid is common:

1. Draft consumer use cases and example interactions.
2. Create enough OpenAPI to review the contract.
3. Implement the provider.

4. Check the running implementation against the agreed description.
5. Generate selected documentation or client code from the verified contract.

The key is a clear source of authority and an automated way to detect disagreement.

## Reuse schemas without hiding operation meaning

Reusable schemas reduce repetition, but excessive reuse can make operations harder to understand. One universal `Ticket` schema with dozens of optional properties may hide which fields are actually required, accepted, or returned in each context.

Prefer schemas that reflect meaningful contract roles, such as:

- `TicketCreateRequest`
- `TicketUpdateRequest`
- `TicketSummary`
- `TicketDetail`
- `TicketError`

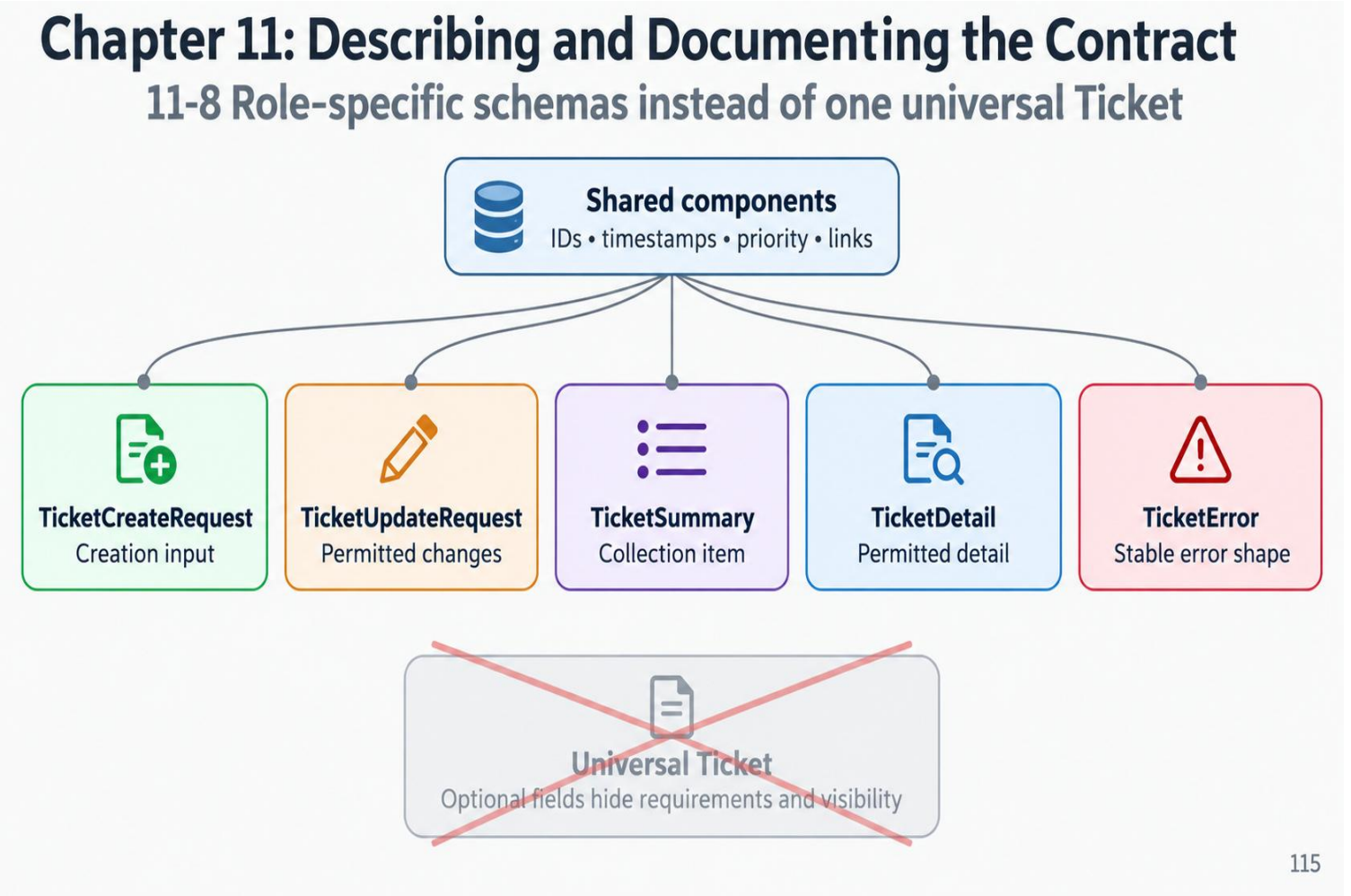



*Figure 11-7: Role-specific schemas instead of one universal Ticket*

These names do not require completely duplicated definitions. Shared field components can still be reused. The important point is that creation input, update input, list output, and detail output have different guarantees.

Avoid constructing an inheritance hierarchy merely to remove a few repeated lines. Documentation should optimize for clarity of the contract, not the smallest possible OpenAPI file.

## Document security where consumers need it

A security declaration tells tools how a credential is supplied. Consumers also need prose that answers operational questions:

- How does a caller obtain credentials?
- Which environment or issuer creates them?
- Which audience and scopes are required?
- How should expiration be handled?
- Which operations require additional permissions?
- Where should a consumer request access or report compromise?

Do not put real tokens or secrets in examples. Use obvious placeholders and ensure copied commands do not encourage users to store credentials in shell history or shared documents.

Permission behavior belongs near the operation, not only in a separate security overview. A reader examining ticket deletion should see that it requires administrative authority and may be prohibited by retention policy.

## Make the error contract discoverable

For each operation, document the failures that consumers must handle. A generic statement that “other HTTP errors may occur” is not enough for predictable client behavior.

The operation documentation should identify:

- Relevant status codes.
- Stable machine-readable error codes.
- Field-level validation details when applicable.
- Whether retry may be appropriate.
- Which recovery action the consumer should take.

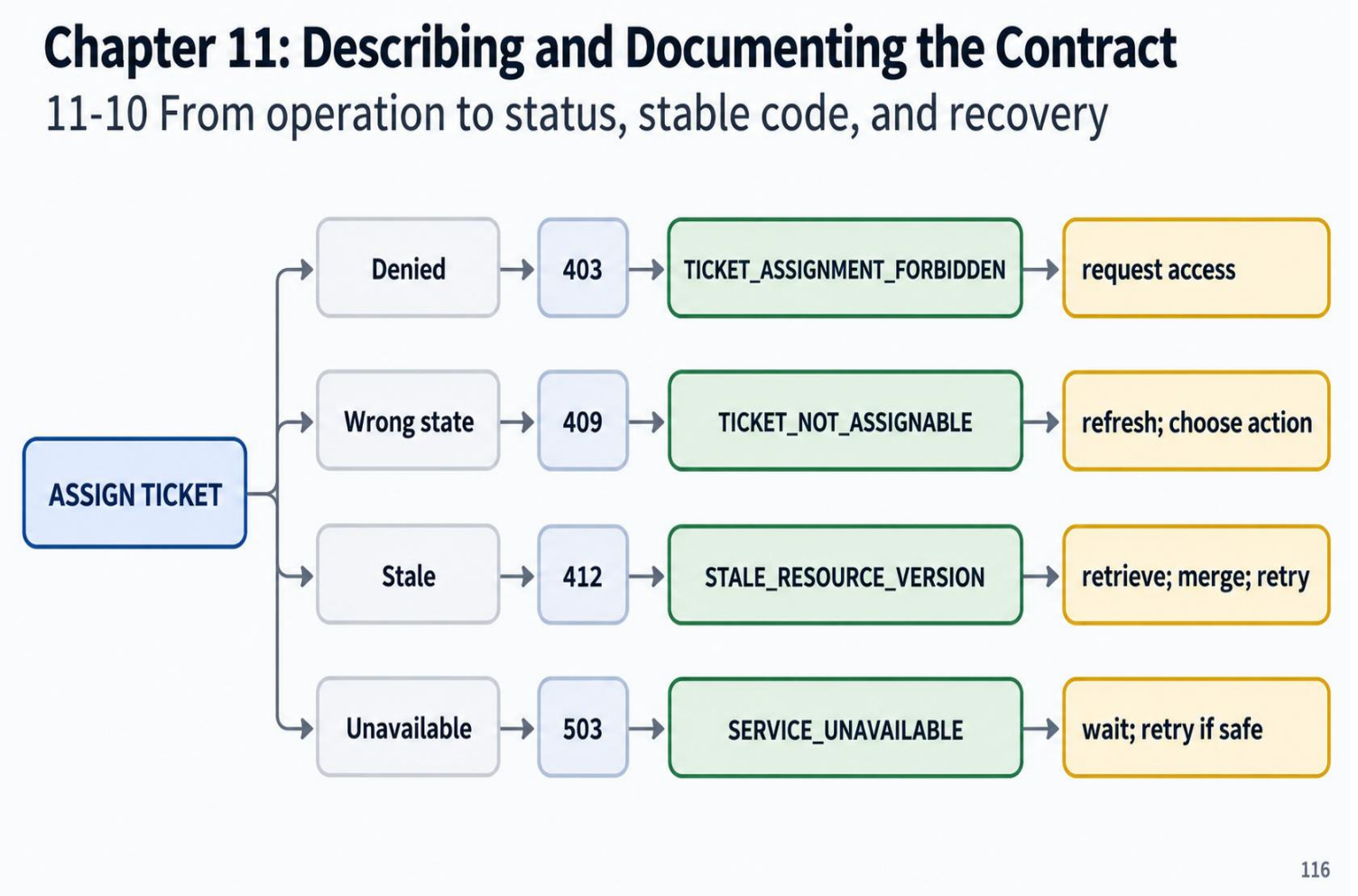



*Figure 11-8: From operation to status, stable code, and recovery*

Reusable error schemas are useful, but operation-specific error lists still matter. `409 Conflict` on assignment may mean `TICKET_NOT_ASSIGNABLE`, while `409` on creation may mean a business uniqueness conflict. The shared structure does not replace the specific meaning.

## Generated artifacts accelerate work but do not replace judgment

An OpenAPI document can support generated reference pages, client libraries, server interfaces, test data, mocks, and validation rules. These tools reduce repetitive work and can improve consistency.

One example of generated reference documentation is Swagger UI. The official Petstore screen below shows methods and paths as expandable operations and provides server selection and authorization controls. This is an example interface; the ticket API in this book uses its own contract.

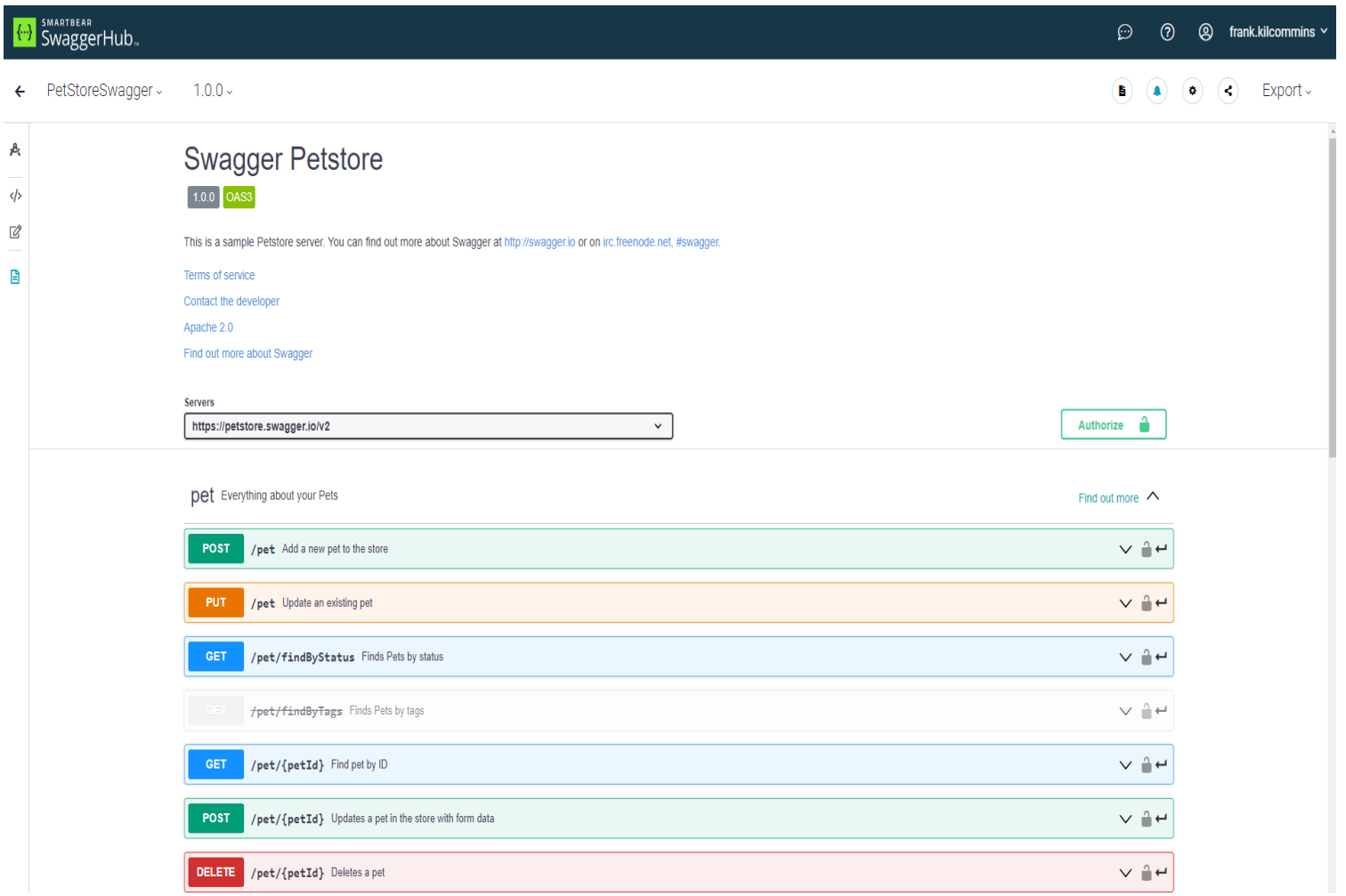



*Figure 11-R1: Generated OpenAPI documentation in Swagger UI. Source: Swagger official website (SmartBear), https://swagger.io/tools/swagger-ui/ (accessed October 3, 2026).*

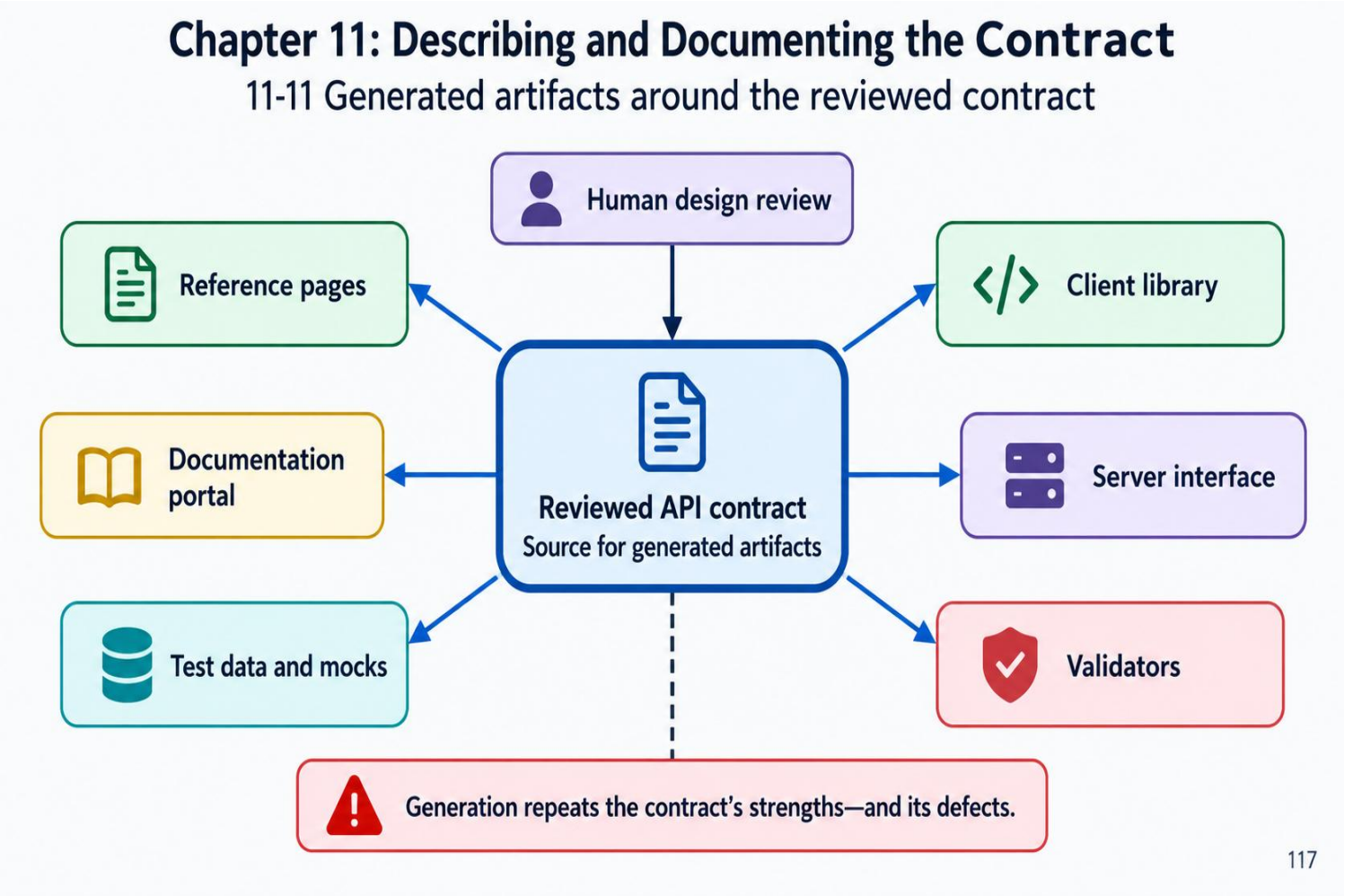



*Figure 11-9: Generated artifacts around the reviewed contract*

They cannot determine:

- Whether the resource boundaries match business capabilities.
- Whether an operation has a safe retry model.
- Whether permission checks are complete.
- Whether a field name communicates the right meaning.
- Whether a supposedly additive change breaks a real consumer assumption.

Generated code also inherits weaknesses in the description. If the contract marks every field optional, the generated client may expose uncertainty rather than eliminate it.

Treat generation as an accelerator after design decisions have been made and reviewed.

## Keep description, implementation, and tests aligned

Contract drift occurs when the published description and actual API behavior differ. Drift destroys trust: consumers learn to test guesses against the server rather than rely on the contract.

A practical alignment loop is:

1. Review changes to the contract as code changes are reviewed.
2. Validate the OpenAPI document structurally.
3. Run conformance tests against the provider.
4. Check example messages against schemas.
5. Detect incompatible changes before release.
6. Publish the description and documentation from the reviewed version.

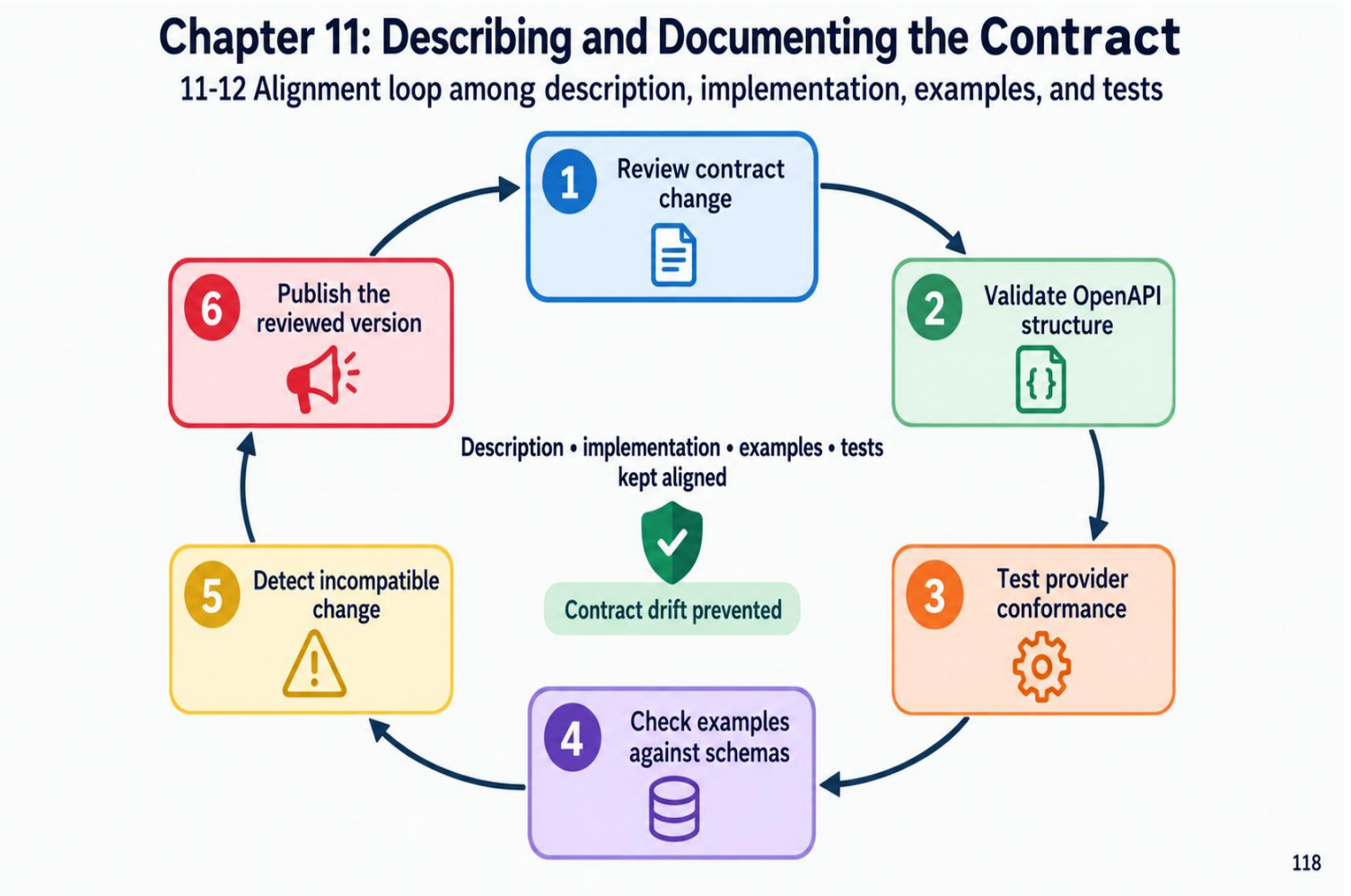



*Figure 11-10: Alignment loop among description, implementation, examples, and tests*

A **conformance test** verifies that implementation behavior matches an agreed specification. It may check response status, headers, body shape, required fields, and documented error forms.

The contract repository or location should have clear ownership. Consumers need to know which version describes each deployed environment and release.

## Changelogs and migration notes answer different questions

A **changelog** records notable contract changes by release or date. **Migration notes** explain what a consumer must do when a change requires adoption work.

A useful changelog entry states:

- What changed.
- Whether the change is compatible for existing consumers.
- When it became available.
- Which operations or schemas are affected.
- Where to find migration details when needed.

Migration notes should be task-oriented. For example, “The assignment response now contains `assignment`; continue reading `assigneeId` until the stated retirement date, then migrate to `assignment.assigneeId`.” A list of altered schema lines is not a migration plan.

## Two documentation misconceptions

### “Generating an OpenAPI file means the API is well designed”

OpenAPI can describe an inconsistent, insecure, or unusable API with great precision. Design quality still comes from consumer analysis, domain modeling, semantic choices, security review, and operational planning.

### “Examples are optional because the schema is sufficient”

Schemas define structure, but examples show how fields work together in realistic interactions. Without examples, consumers often misread field ownership, omission behavior, error details, and lifecycle transitions.

## Documentation readiness check

Before publishing an operation, confirm that:

- Its purpose and consumer task are clear.
- Paths, methods, parameters, bodies, responses, and security are described.
- Input and output schemas reflect their distinct roles.
- Required fields, constraints, formats, defaults, and examples agree.
- Permission and state-transition rules are explained in prose where schemas cannot express them.
- Important errors and recovery behavior are listed for the operation.
- Examples validate against the contract.
- Generated artifacts are reviewed rather than trusted automatically.
- Implementation conformance is tested.
- Changes, deprecations, and migrations have a visible record.

**Documentation is successful when a consumer can build the intended integration without reverse-engineering hidden behavior from experiments or source code.**

## Chapter Summary

- Documentation serves both humans making decisions and tools processing structure.
- OpenAPI can describe operations and schemas, but business rules, recovery behavior, and compatibility policy still require deliberate explanation.
- Generated artifacts accelerate work only when the reviewed contract remains the source of truth.

## Review Questions

1. Which parts of an API contract are difficult to communicate through a schema alone?
2. How do examples complement formal constraints?
3. What practices prevent the description, implementation, and tests from drifting apart?

# Chapter 12 — Testing and Reviewing an API Contract

**API quality is established by testing the contract from several directions and reviewing whether the contract itself is complete, safe, and usable.** A successful happy-path response proves only that one example worked under one condition.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Build a test matrix from contract decisions rather than testing only successful endpoint reachability.
- Distinguish schema, conformance, integration, consumer-contract, permission, boundary, retry, and compatibility tests.
- Separate a contract defect from an implementation defect during review and release verification.

## Test behavior, not only endpoint reachability

An API test should verify an observable promise. For ticket creation, relevant promises include:

- Valid input creates one ticket.
- Missing required input is rejected predictably.
- Unauthorized callers cannot create in another tenant.
- Repeating the same logical submission does not create an unintended duplicate when the idempotency contract applies.
- The response contains the documented status, headers, and data shape.

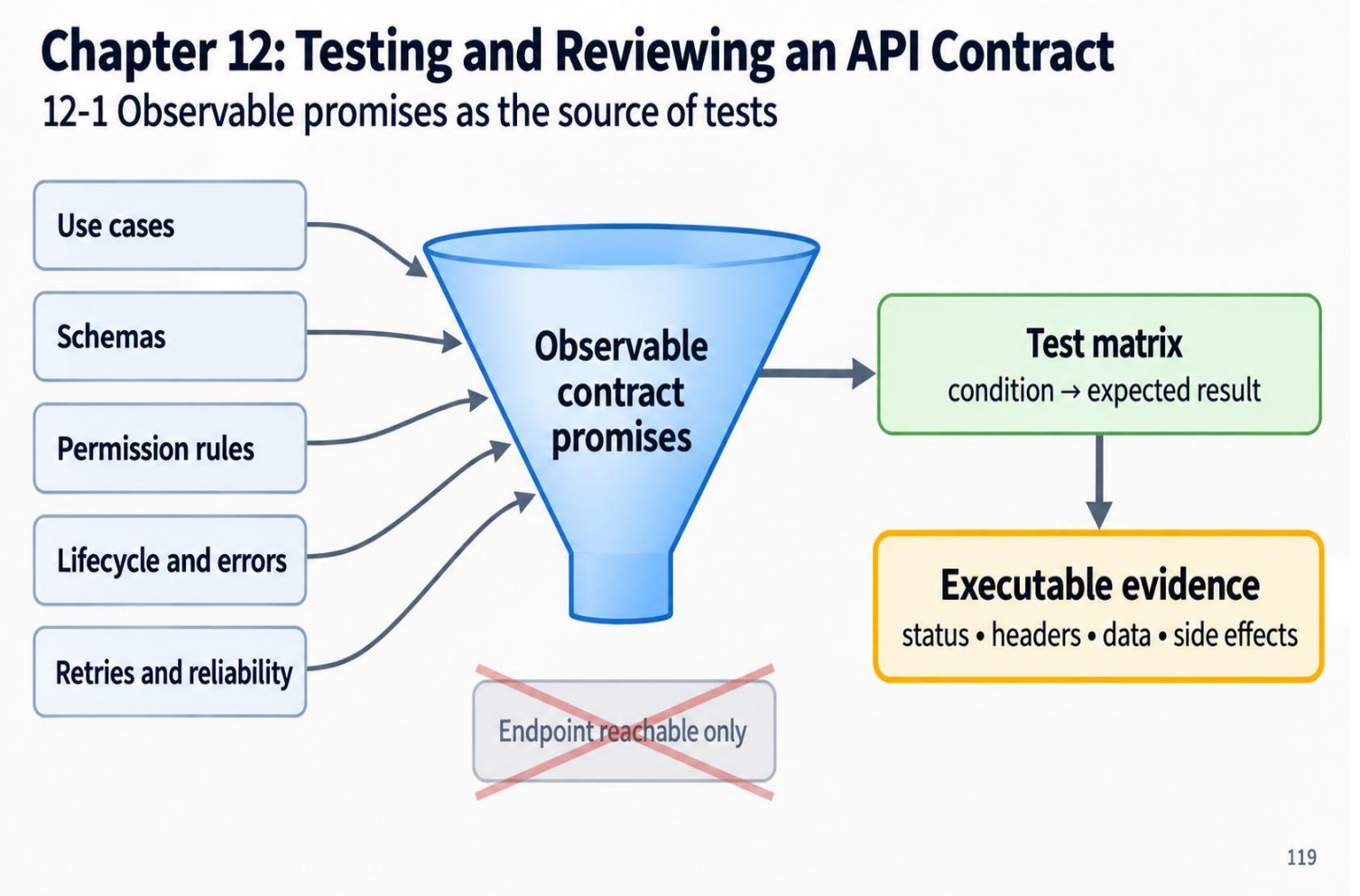



*Figure 12-1: Observable promises as the source of tests*

A test that checks only "the response was `200` or `201`" misses most of the contract. It can pass while the response omits required data, returns the wrong tenant's ticket, or applies the wrong default.

Tests should be derived from use cases, schemas, permission rules, lifecycle transitions, error definitions, and reliability guarantees. The API description provides valuable input, but the design decisions outside the schema must also appear in the test plan.

## Use a compact set of test categories

Several categories cover most foundational contract risks:

- **Positive tests:** Valid requests produce the intended result.
- **Negative tests:** Invalid or forbidden requests fail in the documented way.
- **Boundary tests:** Values at and just beyond limits behave correctly.
- **Authorization tests:** Each caller type can perform only permitted actions on permitted resources.
- **Concurrency tests:** Stale writes and simultaneous changes do not silently lose data.
- **Reliability tests:** Retries, duplicate delivery, timeouts, rate limits, and partial dependency failures have defined outcomes.
- **Compatibility tests:** Existing consumer expectations continue to hold after a change.

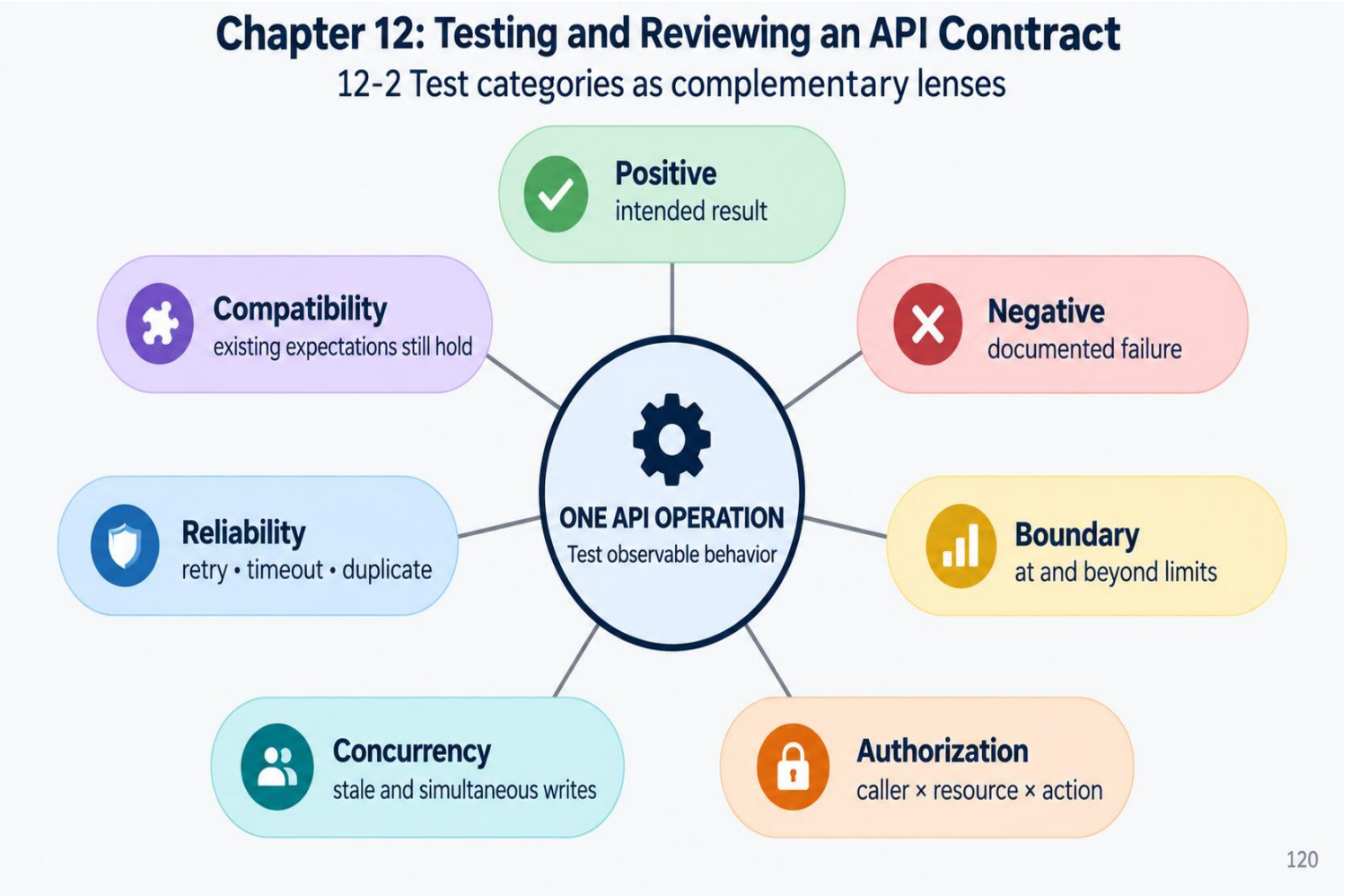



*Figure 12-2: Test categories as complementary lenses*

These categories are not separate test tools. They are perspectives that prevent a team from concentrating only on the easiest successful examples.

## Build the test matrix from contract decisions

A **test matrix** organizes important conditions and expected outcomes so that coverage can be reviewed before implementation details obscure the design.

For ticket creation and assignment, a compact matrix might be:

| Operation | Condition | Expected contract result |
|---|---|---|
| Create ticket | Valid requester and valid body | 201 Created; one ticket; provider-controlled fields present |
| Create ticket | Missing subject | Validation error with field detail; no ticket created |
| Create ticket | Subject exceeds maximum length | Validation error at the documented boundary |
| Create ticket | Same idempotency key and same request | Original logical result returned; no duplicate ticket |
| Create ticket | Same idempotency key and different request | Defined key-reuse error; no second operation performed |
| Assign ticket | Authorized agent, assignable state, current version | Assignment succeeds and a new version is returned |
| Assign ticket | Caller lacks queue access | Authorization failure without protected data leakage |
| Assign ticket | Ticket is closed | Business conflict with stable error code |
| Assign ticket | Stale If-Match value | 412 Precondition Failed; no overwrite |
| Assign ticket | Dependency times out before commit | Bounded failure with documented retry guidance |

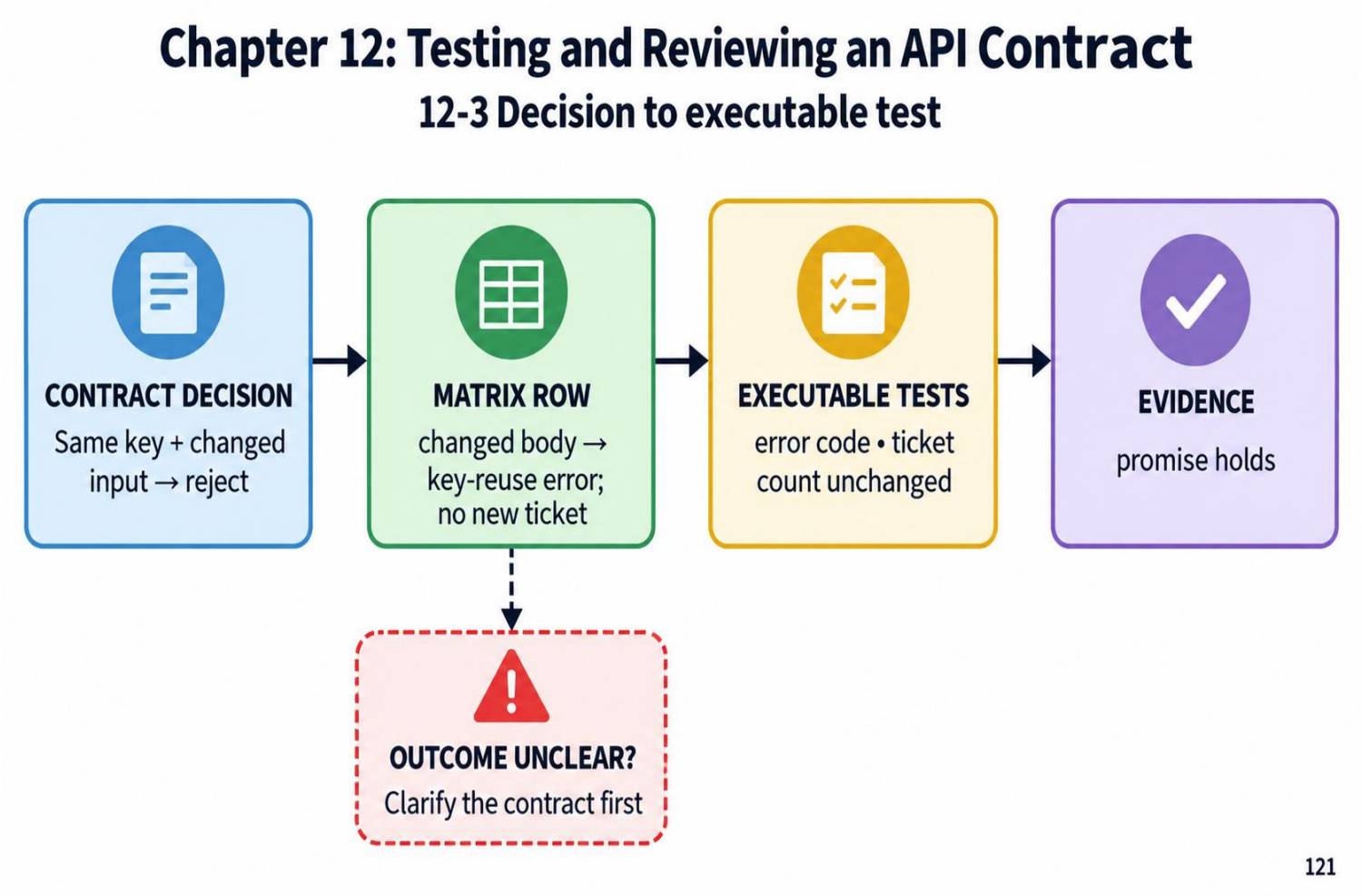



*Figure 12-3: From contract decision to matrix row to executable test*

Each row should map to one or more executable tests. The matrix also exposes missing design. If the team cannot state what happens when the same idempotency key is used with different input, the problem is not missing test code; the contract is incomplete.

## Schema validation checks message structure

**Schema validation** checks whether a message conforms to defined structural rules such as required properties, types, lengths, and enumerated values.

Useful schema checks include:

- Example requests conform to request schemas.
- Successful responses conform to their response schemas.
- Error responses conform to the shared error structure.
- Required headers are present and valid.
- Undocumented response fields are detected when strict control is intended.

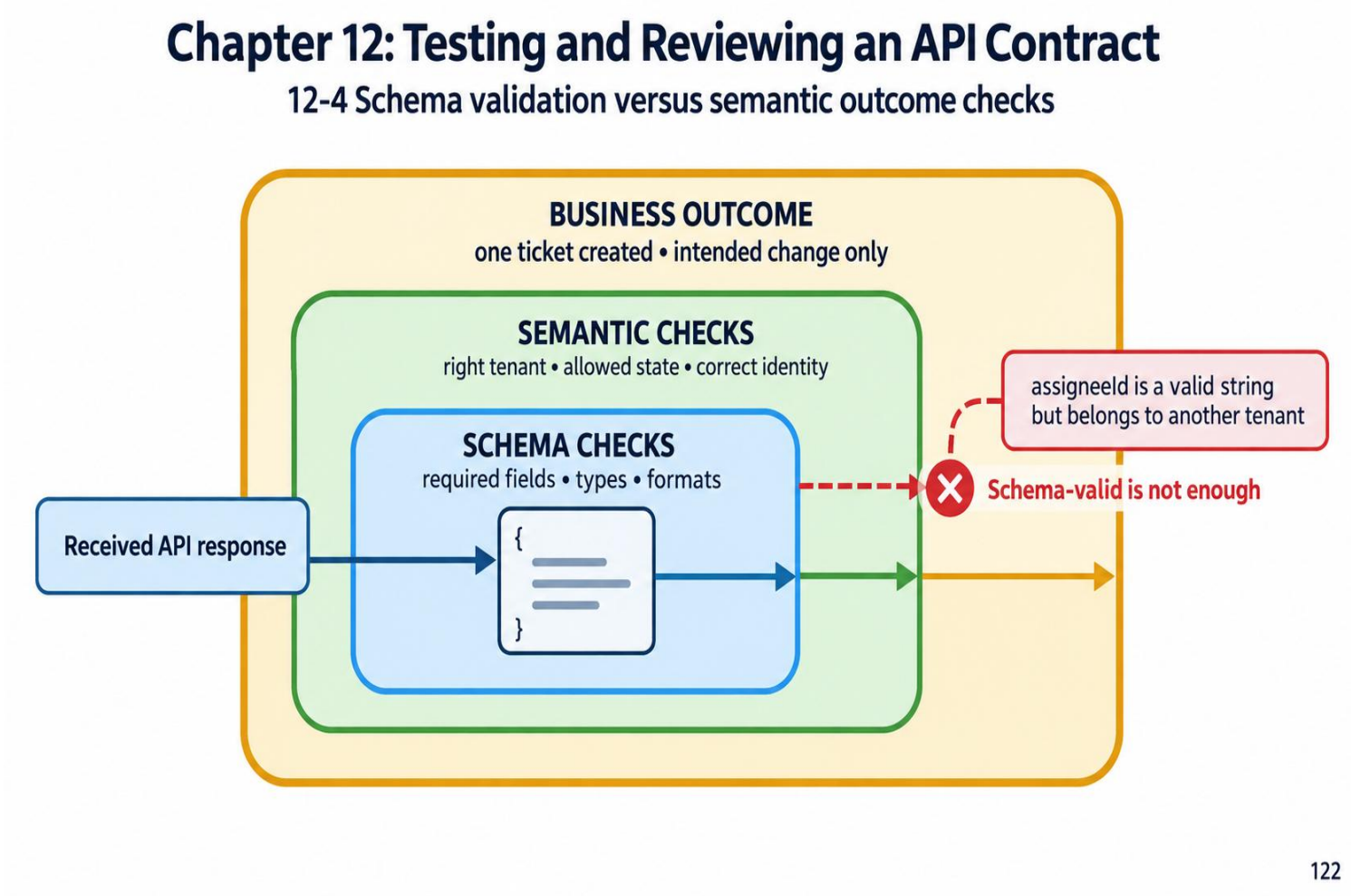



*Figure 12-4: Schema validation versus semantic outcome checks*

Schema validation is necessary but cannot prove business correctness. A response can have a valid `assigneeId` string while identifying an agent from the wrong tenant. A valid schema also cannot prove that one ticket rather than two was created.

Use schema validation as one layer, then assert semantic outcomes separately.

## Conformance tests compare the provider with the contract

A conformance test sends requests to a running provider and compares observable results with the agreed API description and behavioral rules. It can detect differences such as:

- The implementation returns `200` where the contract promises `201`.
- A required response header is missing.
- The implementation accepts a value outside the schema.
- An error body uses an undocumented shape.
- A supposedly optional field is always required by the server.

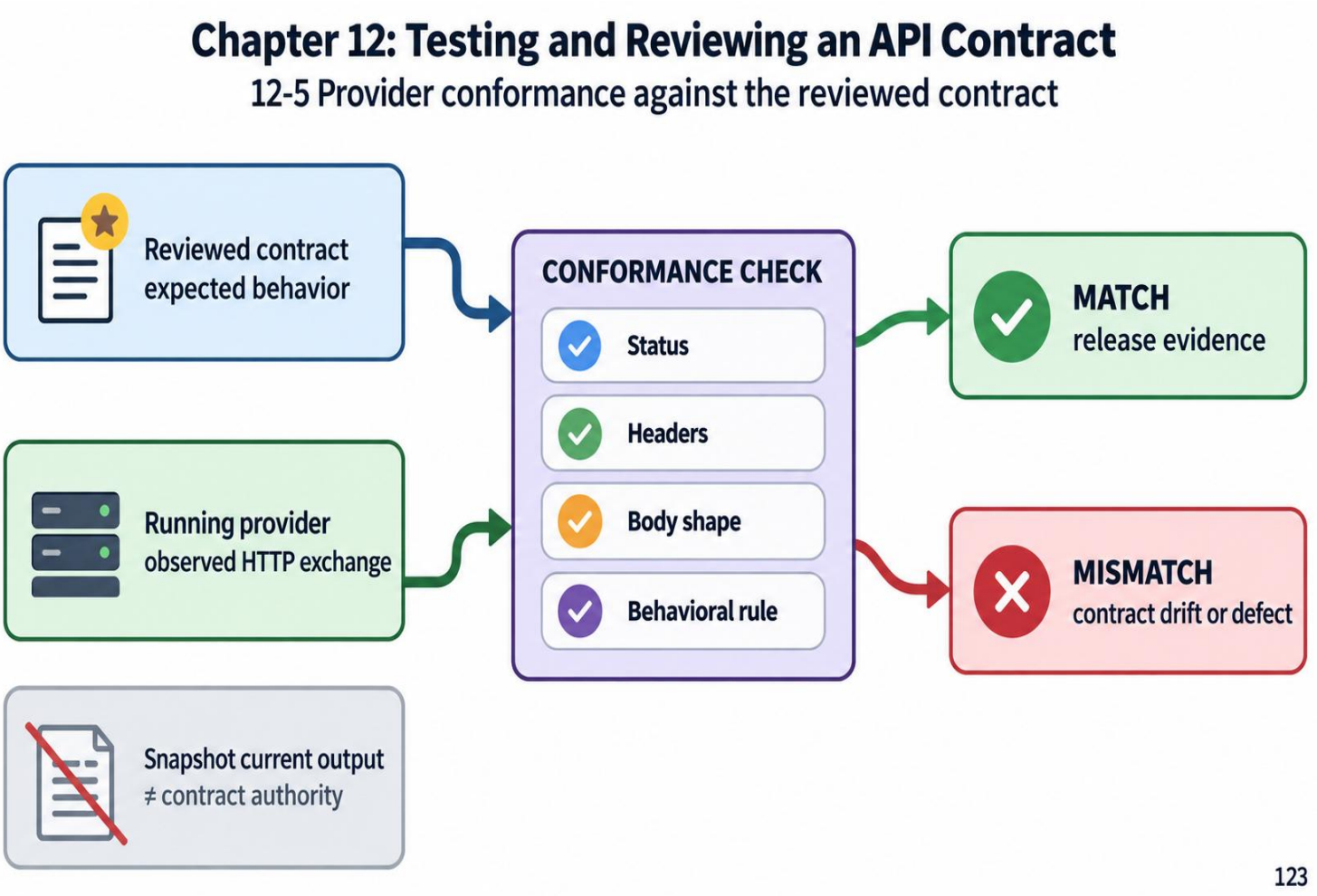



*Figure 12-5: Provider conformance against the reviewed contract*

Conformance can be checked at build time, in a test environment, and through selected production monitoring. The closer the check is to release, the more likely it is to catch configuration and gateway behavior as well as application behavior.

Avoid designing tests that merely reproduce current implementation output and call that output the contract. The expected result should come from reviewed requirements and contract artifacts.

## Integration tests cross the real network boundary

An **integration test** verifies cooperation between components rather than only one isolated function. For an API, useful integration tests send actual HTTP messages through the same important layers used in deployment, such as routing, authentication, serialization, and persistence.

These tests catch problems that unit tests commonly miss:

- Header names are altered or dropped by a gateway.
- Authentication configuration rejects the correct audience.
- JSON serialization changes null or timestamp behavior.
- Database constraints produce an unexpected error response.
- A transaction commits only part of the intended business outcome.

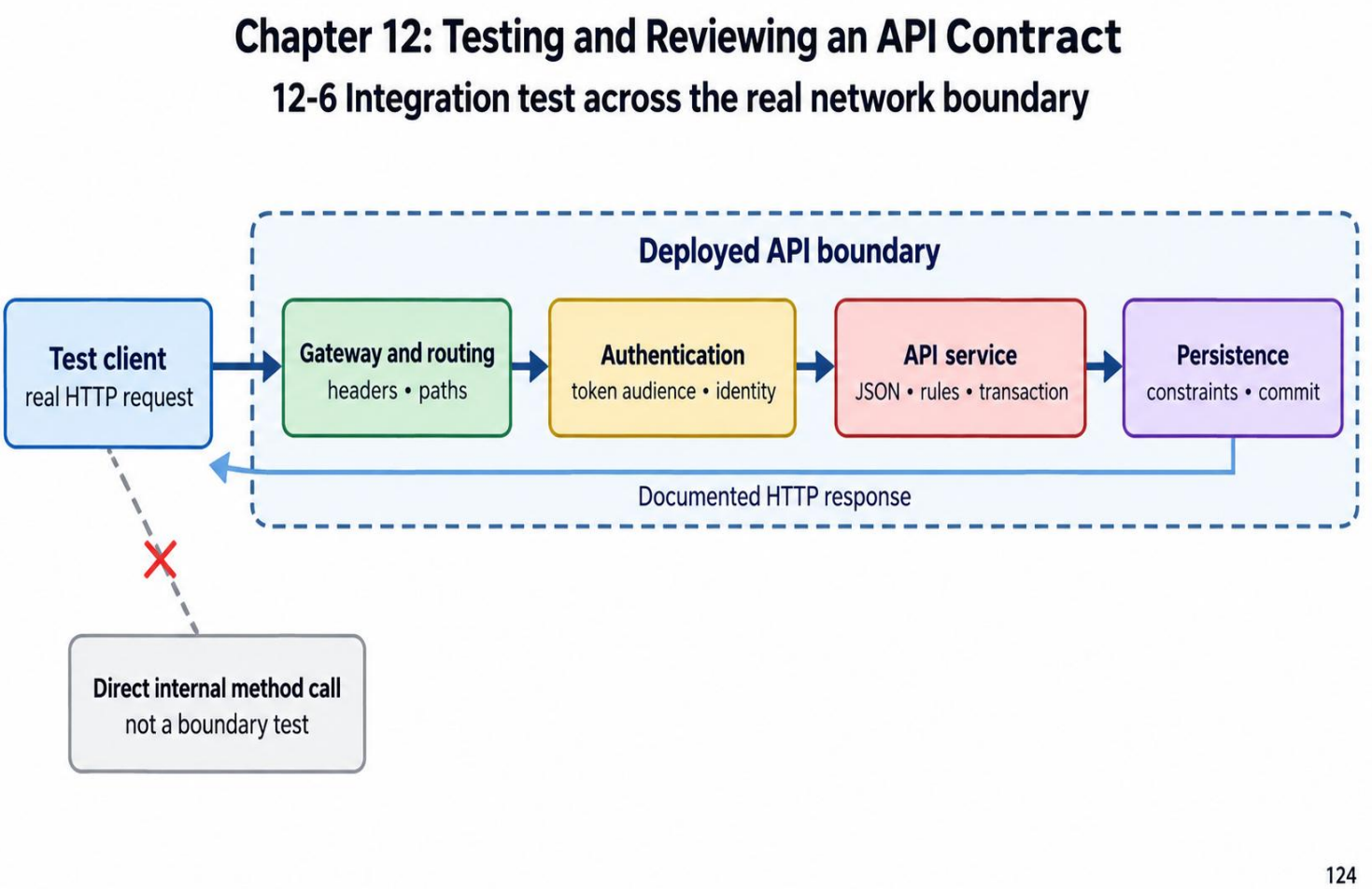





*Figure 12-6: Integration test across the real network boundary*

Not every test must use a complete production-like environment. A focused set of network-boundary tests provides more value than attempting to reproduce every infrastructure detail in every test run.

## Consumer-driven contract tests capture real consumer expectations

**Consumer-driven contract testing** records examples of interactions that a particular consumer depends on and verifies that the provider continues to satisfy them. The consumer specifies relevant requests and expected response characteristics; the provider verifies those expectations against its implementation.

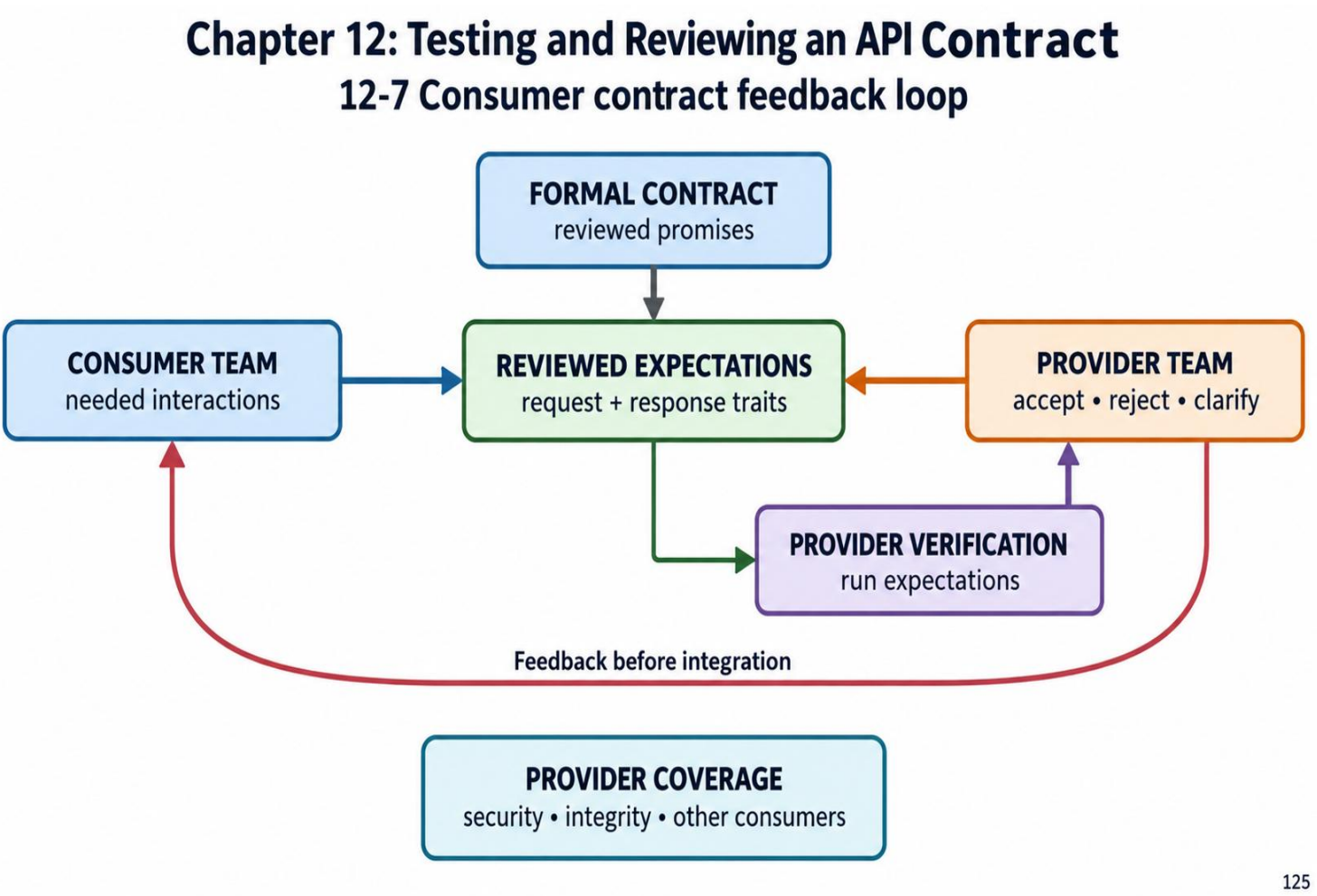



*Figure 12-7: Consumer-driven contract feedback loop*

For the reporting service, a contract test might state that each ticket summary includes `id`, `status`, and `createdAt`, and that the service tolerates additional fields. For the agent web application, another test may cover the assignment conflict structure.

This approach is useful when:

- Provider and consumers are developed by different teams.
- Consumers deploy independently.
- The formal schema allows more variation than a consumer actually handles.
- Provider changes need fast feedback before integration environments are available.

Consumer-driven tests do not replace provider-owned tests. A provider must still test security, data integrity, undocumented consumers, and rules that no single consumer exercises.

The provider should review consumer expectations rather than automatically accepting every accidental assumption as permanent contract. A consumer test that requires array fields to appear in an undocumented order may reveal either a missing contract decision or an unnecessarily brittle consumer.

## Mocks help early work but can preserve a false assumption

A **mock** is a controlled substitute that returns programmed responses instead of invoking the real provider. A **simulator** is a richer substitute that imitates a broader set of provider behaviors.

Mocks help consumers develop before the provider is available, demonstrate example flows, and test rare errors. Their limitation is simple: a mock behaves according to what its author believed, not necessarily what the real API does.

Common mock failures include:

- Returning only successful responses.
- Responding instantly when the real network can time out.
- Ignoring authentication and authorization.
- Reusing one fixed identifier or timestamp.
- Returning fields that are absent in the actual API.
- Accepting invalid transitions that the provider rejects.

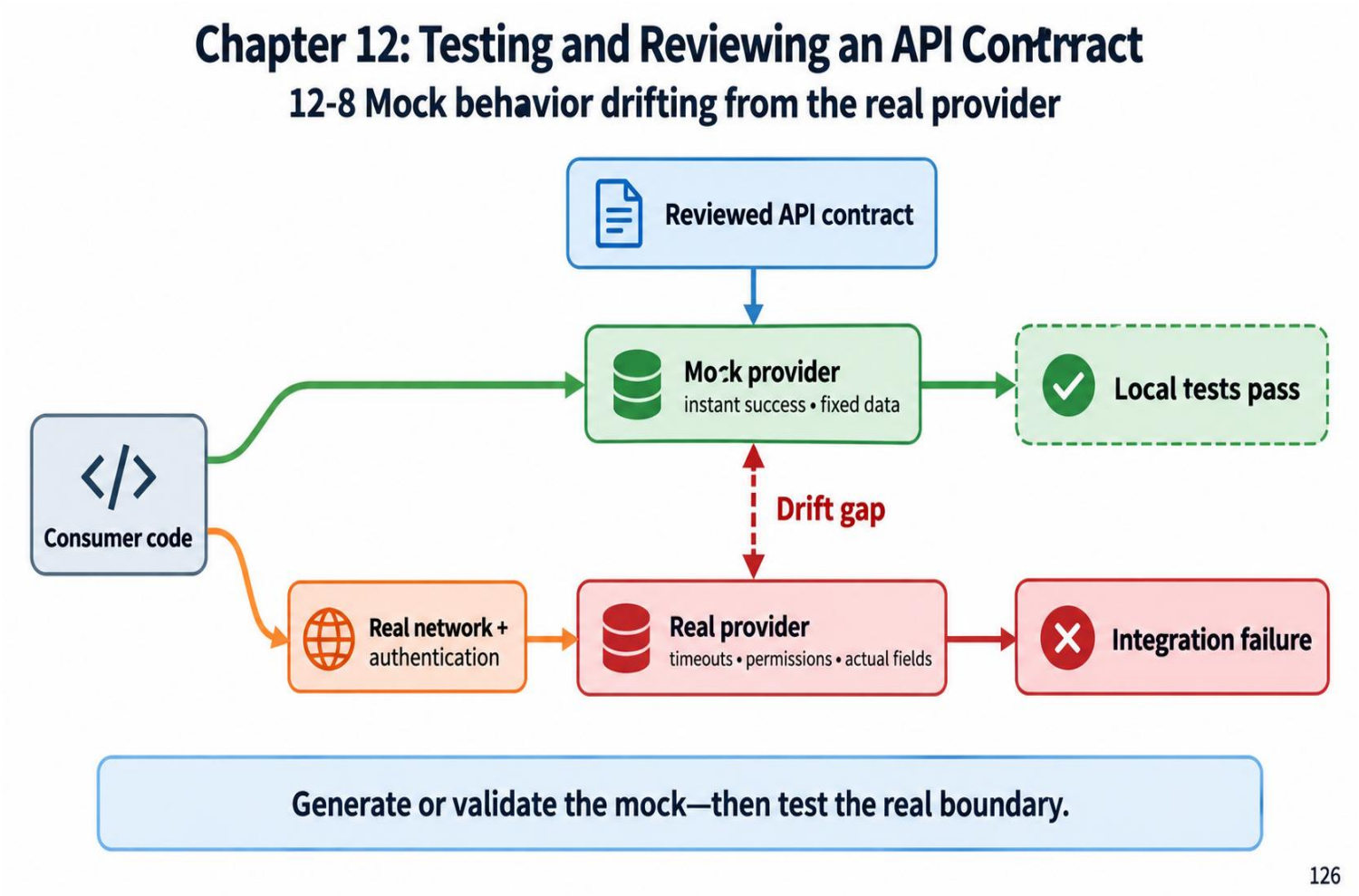



*Figure 12-8: Mock behavior drifting from the real provider*

Generate or validate mocks from the reviewed contract where practical, and still run integration and conformance tests against the real provider. A mock should accelerate feedback, not become an alternative source of truth.

## Test permissions as a matrix, not one denial

Authorization testing needs both positive and negative cases across caller and resource relationships. For the ticket API, the matrix may include:

| Caller | Own ticket | Same-tenant unrelated ticket | Other-tenant ticket | Administrative operation |
|---|---|---|---|---|
| Requester | Allowed for defined requester actions | Denied | Denied | Denied |
| Support agent | Allowed when queue policy permits | Allowed when queue policy permits | Denied | Denied unless separately granted |

| Caller | Own ticket | Same-tenant unrelated ticket | Other-tenant ticket | Administrative operation |
|---|---|---|---|---|
| Reporting service | Read approved fields only | Read approved fields only | Denied | Denied |
| Administrator | Allowed within assigned administrative scope | Allowed within scope | Denied outside scope | Allowed only for defined admin actions |

The exact entries are ticket-service policy, not a universal model. The lesson is that “authenticated request denied once” is inadequate coverage. Test horizontal access between peer resources, vertical access to more privileged actions, field-level visibility, and tenant isolation.

Tests should also verify that denial does not leak protected information through response bodies or inconsistent existence behavior.

## Exercise boundaries where mistakes cluster

Boundary tests target values near limits and transitions. Examples include:

- Subject length at 199, 200, and 201 characters when the maximum is 200.
- Page size at the default, maximum, and one above maximum.
- Empty comment text versus omitted comment text.
- A ticket just before and just after an expiration or retention boundary.
- Allowed state transition `open` to `in_progress` versus disallowed `closed` to `in_progress`.

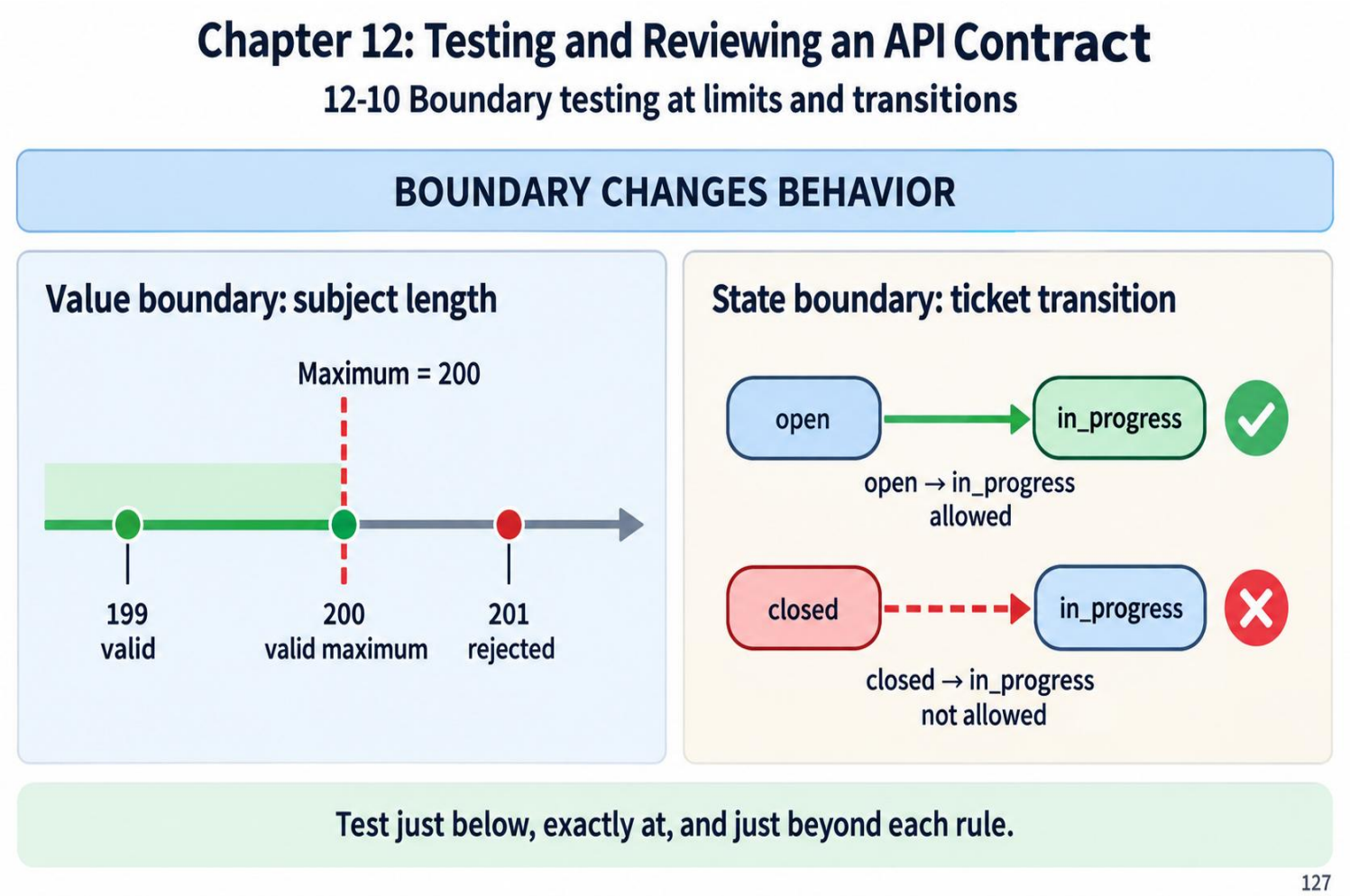



*Figure 12-9: Boundary testing at limits and transitions*

The goal is not to test every possible number. Choose boundaries created by explicit rules, data types, security policy, or operational limits.

Pay particular attention to combinations. A request can have a valid field value but an invalid value for the current ticket state. Contract tests should reflect both field-level and business-level validation.

## Reproduce retries and duplicate delivery

Reliability behavior cannot be proven by sending a request once. Test scenarios should include:

1. Send a state-changing request with an idempotency key.
2. Arrange for the client to lose or ignore the first response.
3. Repeat the same request with the same key.
4. Verify that the logical result is not duplicated.
5. Repeat the key with changed input and verify the documented rejection.

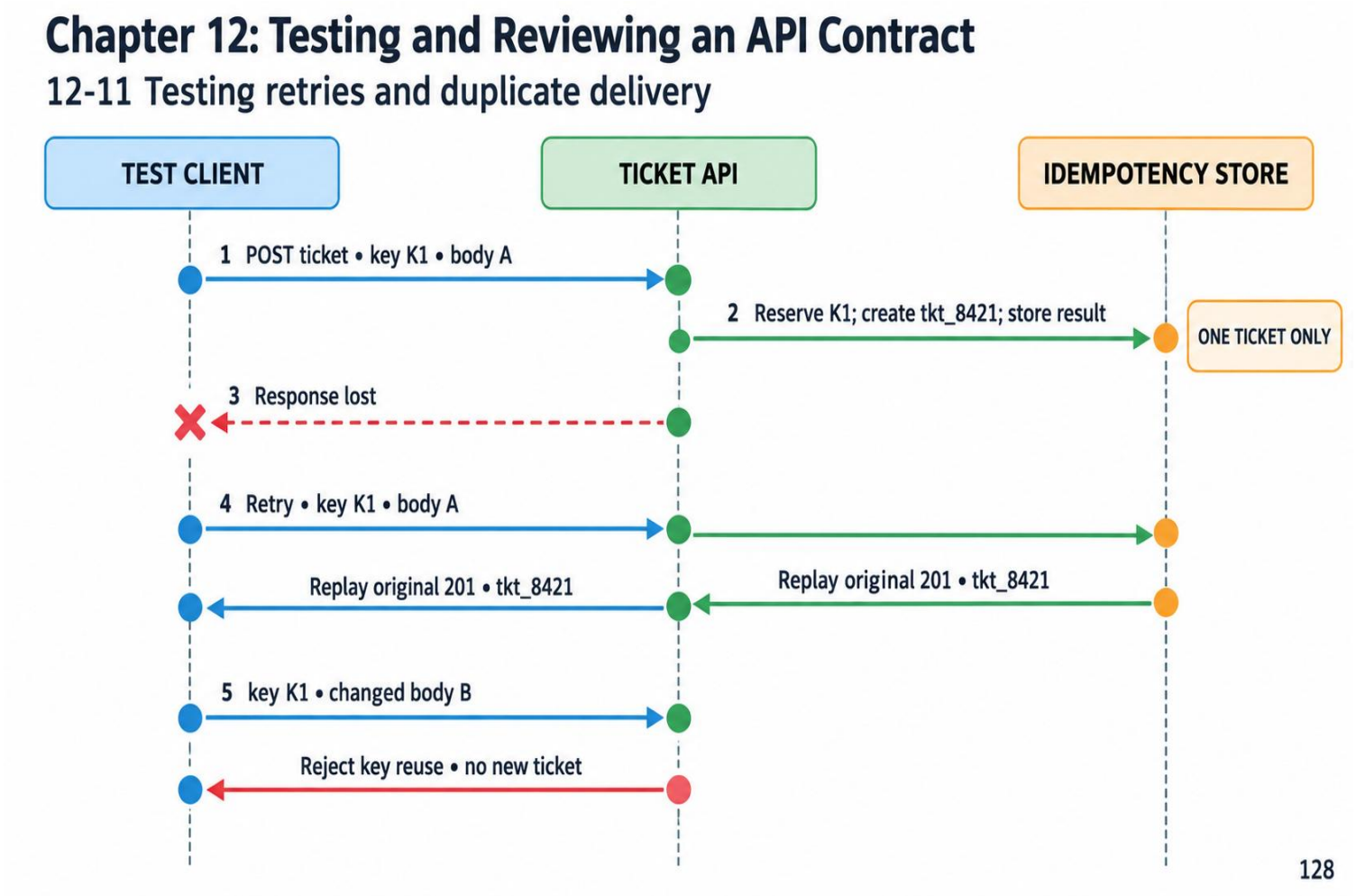



*Figure 12-10: Testing retries and duplicate delivery*

For an operation without idempotent retry semantics, test that documentation and client behavior do not falsely assume safe repetition.

Duplicate testing also applies to webhooks and events later in the guide. A receiver should be tested with the same delivery identifier more than once, because duplicate delivery is a normal reliability condition rather than an exotic fault.

## Test timeouts at uncertain points

A timeout tells the client that it did not receive a result within its waiting limit. It does not necessarily reveal whether the provider completed the operation.

Useful test cases place delay or failure at different points:

- Before the provider receives the request.
- After validation but before state change.
- After the durable state change but before the response reaches the client.
- While a downstream dependency is slow.

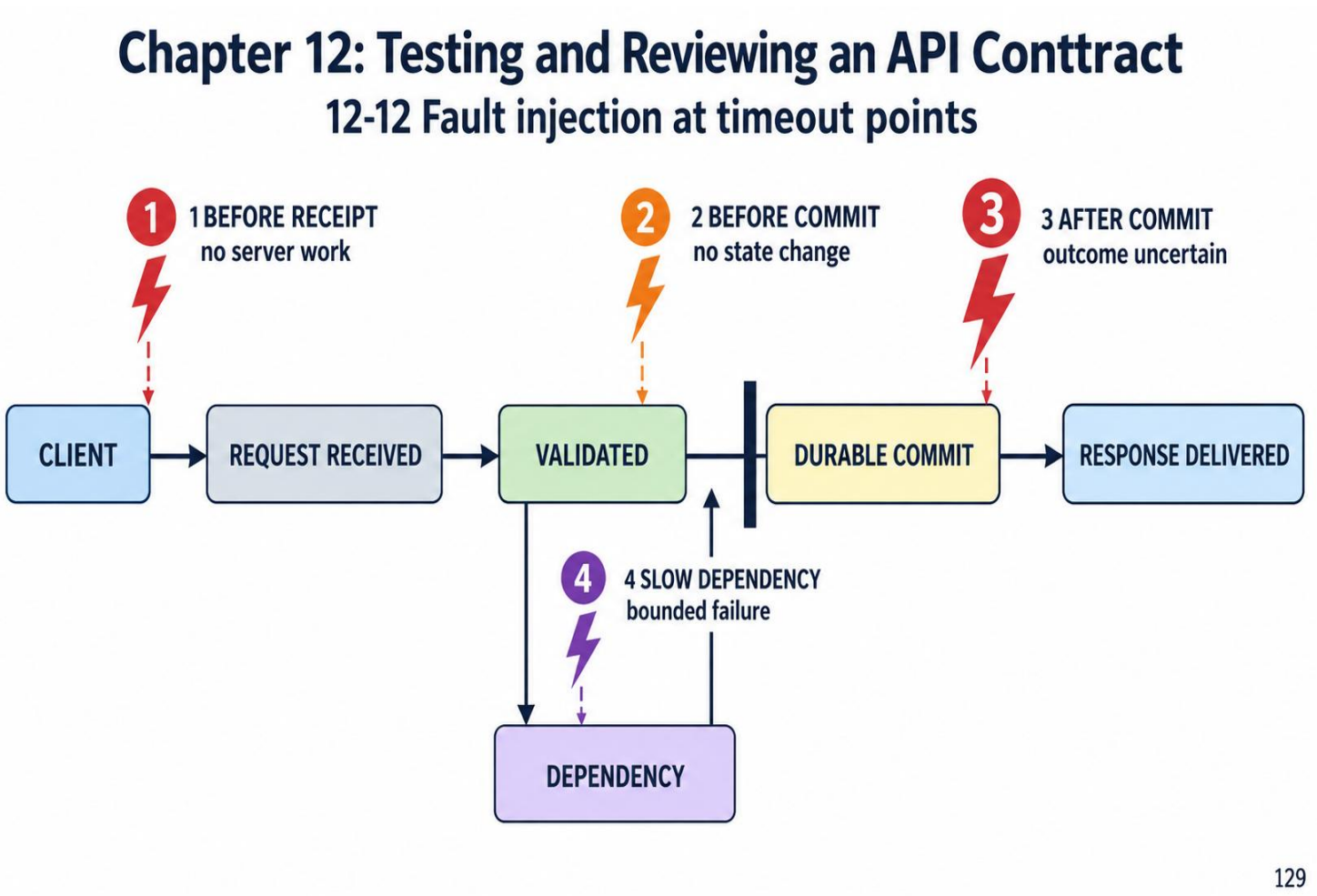



*Figure 12-11: Fault injection at four timeout points*

The expected client action differs. If the provider may have committed a creation, the idempotency contract matters. If a retrieval times out before any state change, a bounded retry may be simpler.

These tests often require controlled fault injection in a test environment. **Fault injection** deliberately introduces delays or failures to verify how the system behaves under non-ideal conditions.

## Verify rate limits and partial failures

Rate-limit testing should confirm:

- Which identity or tenant consumes the limit.
- The documented response status and error body.
- Whether retry guidance is present.
- Whether one abusive caller affects unrelated callers.
- Whether expensive operations have appropriate limits.

For operations that depend on multiple systems, simulate one dependency failing while others remain healthy. Verify that the API returns the documented overall result and does not expose a misleading success after only part of the promised outcome completed.

Do not assert internal implementation steps unless they are contractually visible. Test the durable outcome and client-visible failure behavior.

## Include basic security and robustness checks

A foundational API test plan should include practical misuse cases without becoming a full penetration-testing program:

- Missing, malformed, expired, and wrong-audience credentials.
- Attempts to access another user's or tenant's identifier.
- Unexpected content types and oversized payloads.

- Unknown fields where the contract defines strict handling.
- Invalid encodings or malformed JSON.
- Injection-like text in fields that should be treated as data.
- Repeated failures that should trigger abuse controls.
- Sensitive data absence from errors and routine logs.

Specialist security testing remains necessary for high-risk systems. The contract team should still cover the obvious boundary behavior rather than assuming a later security review will define it.

## Check compatibility against the previous contract

A **compatibility check** compares a proposed contract with a previously released contract to identify changes that may break consumers. Automated checks can detect many structural changes, such as removed fields, narrowed types, or newly required inputs.

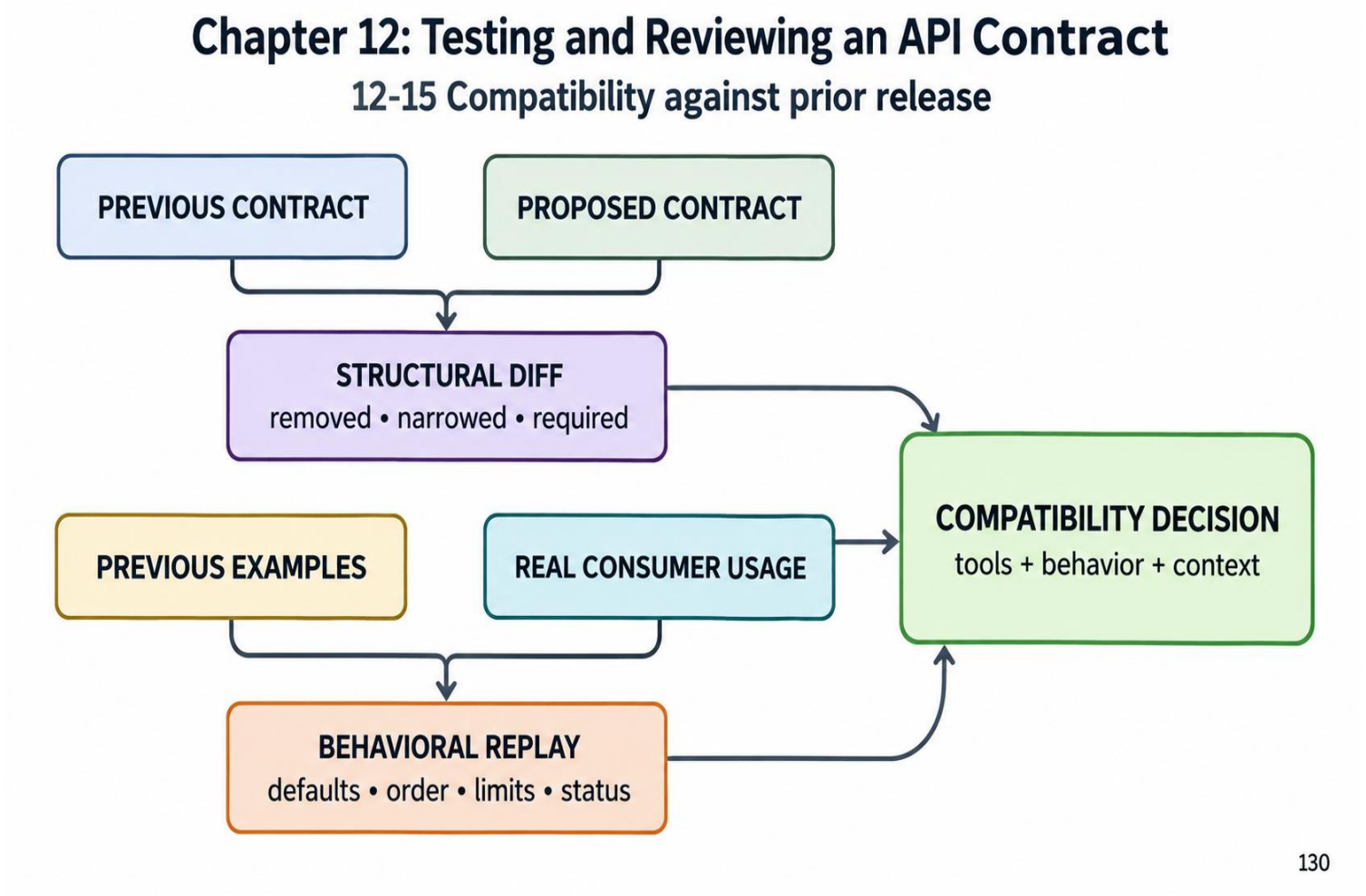



*Figure 12-12: Compatibility check against the previous contract*

Automation cannot reliably judge every semantic change. Changing a default, reinterpreting a status, reducing a limit, or altering ordering can break clients without a schema difference. Compatibility review therefore combines tool output, behavioral tests, and knowledge of real consumer use.

Keep representative requests and responses from previous releases. Re-run them against the new provider and verify that old client assumptions covered by the contract remain valid.

## Review the design before testing implementation

An API design review evaluates whether the contract is suitable before defects become expensive code changes. The review should examine:

- **Consistency:** Similar concepts use similar names and behavior.
- **Completeness:** Success, errors, permissions, limits, and lifecycle rules are defined.

- **Safety:** Retries, concurrency, transactions, and authorization cannot silently damage data.
- **Usability:** A consumer can complete its task without excessive calls or hidden knowledge.
- **Evolvability:** Likely additions can be made without routine breaking changes.
- **Operational support:** The API can be monitored, diagnosed, limited, and retired.

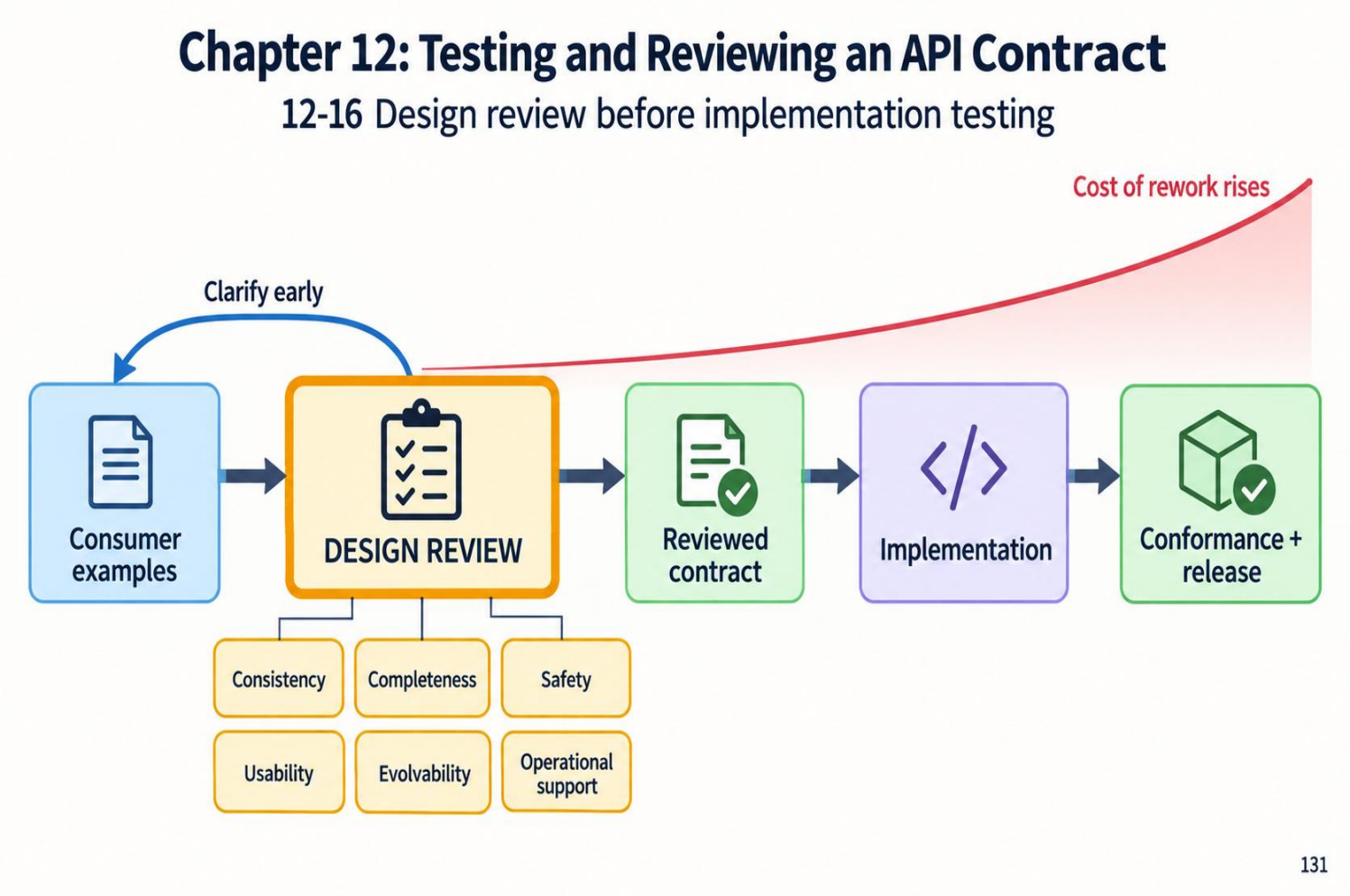



*Figure 12-13: Design review before implementation testing*

Review example interactions early. People can often identify ambiguity in a concrete request and response more easily than in a large schema document.

A productive review explains the consumer or operational consequence of a concern. “I dislike this path” is weaker than “This path models assignment as a field edit, but the business rule requires an auditable actor, eligibility check, and conflict outcome.”

## Separate contract defects from implementation defects

An **implementation defect** occurs when code fails to meet a clear contract. A **contract defect** occurs when the contract itself is missing, contradictory, unsafe, or unsuitable.

Examples:

- The contract requires `404`, but the code returns `500` → implementation defect.
- The contract does not state whether a missing ticket is distinguishable from a forbidden ticket → contract defect.
- The contract defines idempotency-key reuse, but duplicate creation still occurs → implementation defect.
- The contract offers no duplicate-request behavior for a retry-prone creation workflow → contract defect.

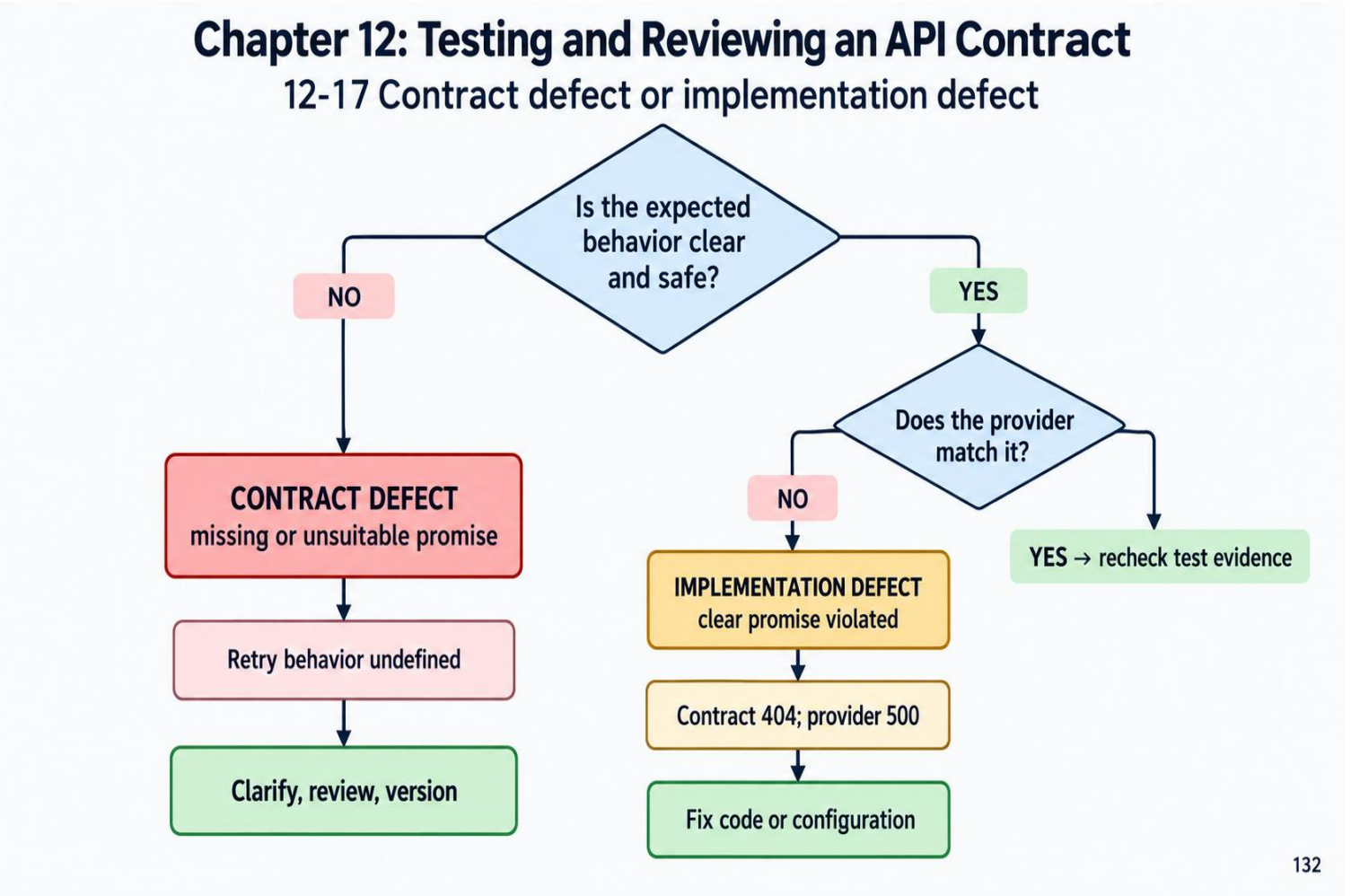



*Figure 12-14: Contract defect versus implementation defect*

The distinction guides the remedy. Fixing code cannot resolve an undefined policy. Changing documentation alone cannot fix code that violates an agreed promise.

## Two testing misconceptions

### "A `200` response proves the API works"

A status code is one observation. Correctness also includes data meaning, side effects, authorization, headers, errors, duplicate behavior, concurrency, and durability. Tests must assert the complete relevant outcome.

### "Generated clients eliminate the need for contract testing"

Generated clients reduce serialization and calling boilerplate. They can still call a provider that behaves differently from its description, and they do not verify business rules, permissions, or compatibility with consumer assumptions.

## Release-level verification checklist

Before releasing a contract change, confirm that:

- Positive, negative, boundary, authorization, concurrency, and reliability cases are covered.
- Examples and schemas validate.
- A running provider conforms to documented statuses, headers, and bodies.
- Object-level and tenant authorization are tested in both allowed and denied directions.
- Retries and duplicate requests produce the documented logical outcome.
- Timeout and dependency-failure behavior is bounded and observable.
- Previous supported consumer interactions still pass.
- Mocks and generated clients are checked against the real provider.
- Review findings identify whether the contract or implementation must change.

- Operationally significant failures can be diagnosed without exposing secrets.

**Testing is strongest when every important design promise has an observable check and every failed check points either to a clear implementation obligation or to a contract decision that still needs to be made.**

## Chapter Summary

- No single test layer proves an entire API contract.
- Important tests cover success, invalid input, authorization, state conflicts, duplicate delivery, timeouts, limits, and prior consumer expectations.
- Design review should occur before implementation testing so that executable checks are based on a coherent contract.

## Review Questions

1. Why does a successful `200` response not prove that the API works correctly?
2. What is the difference between provider conformance and consumer-driven contract testing?
3. Which failure points should be exercised when a state-changing request times out?

# Part V — Evolving and Operating APIs

# Chapter 13 — Evolving an API Without Breaking Clients

**Compatibility must be treated as a design constraint from the first release because providers and consumers rarely change at the same time.** A change is safe only when supported consumers can continue to perform their tasks, not merely when the new server accepts the old URL.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Classify proposed changes from the consumer's perspective before implementation.
- Recognize when additive fields, enumeration values, absence semantics, or behavior changes may still break clients.
- Plan versioning, deprecation, migration, measurement, rollout, and removal as one managed lifecycle.

## Compatibility is judged from the consumer side

**Backward compatibility** means that a newer provider continues to support the documented behavior on which existing consumers rely. The relevant question is not "Did we deploy without errors?" It is "Can supported consumers continue working without an unplanned change?"

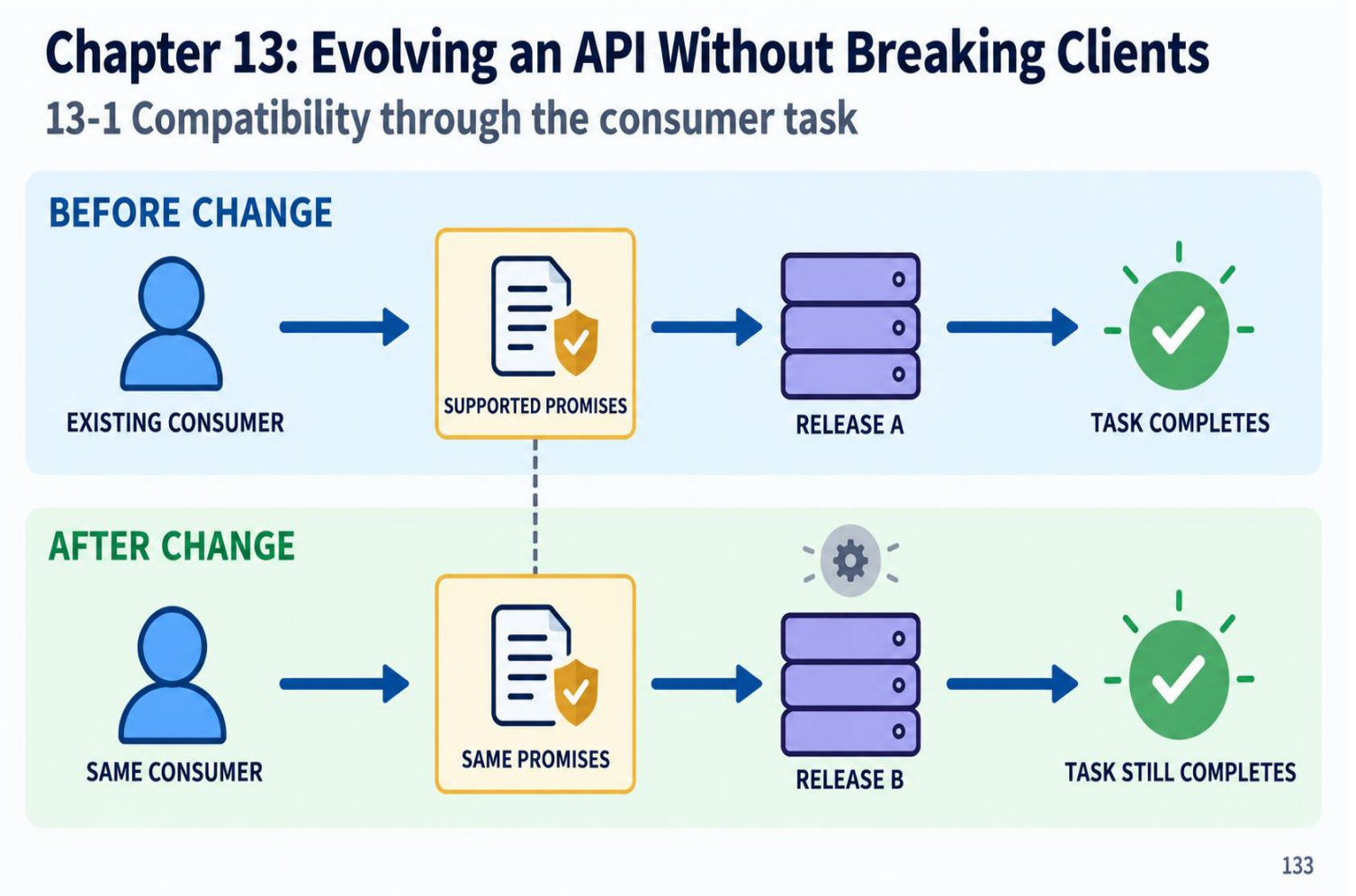



*Figure 13-1: Compatibility judged through the consumer task*

Compatibility includes several dimensions:

- **Structural compatibility:** Messages still have shapes and types that old clients can process.
- **Behavioral compatibility:** Operations retain their documented meaning, defaults, ordering, and side effects.
- **Operational compatibility:** Limits, latency, authentication, and failure behavior remain within supported expectations.
- **Source compatibility:** Generated or hand-written client code can still compile when a published client library changes.

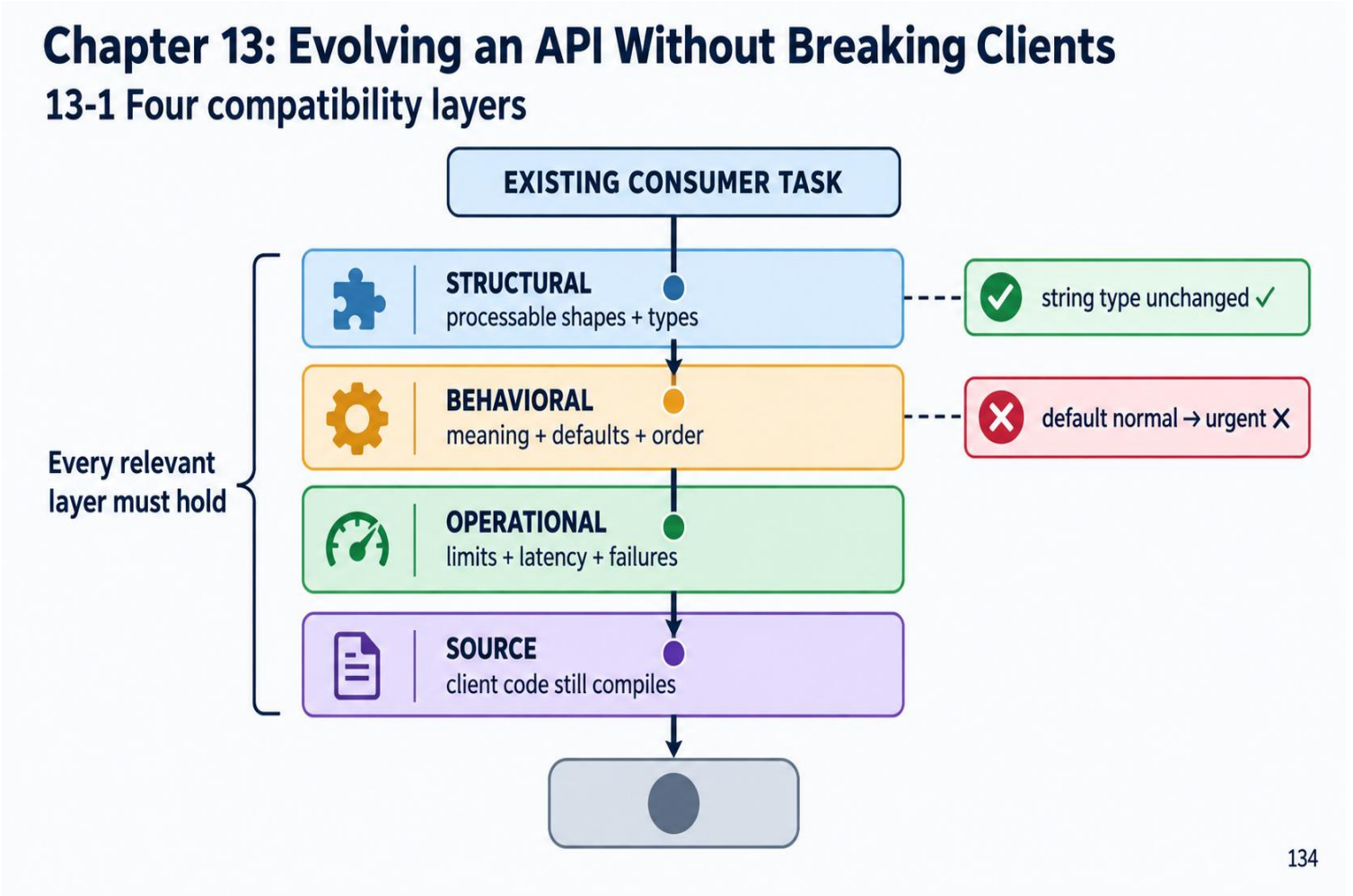



*Figure 13-2: Structural, behavioral, operational, and source compatibility*

An API may remain structurally compatible while breaking behavior. For example, keeping `priority` as a string but changing the default from `normal` to `urgent` requires no schema change, yet it can alter queue routing and escalation.

Consumers also differ in resilience. One client ignores unknown response fields, while another rejects them. The contract should state which behavior consumers are expected to support, but the provider should still measure real usage before assuming every client follows ideal practices.

## Classify changes before implementing them

The following categories are useful starting points. They are not automatic verdicts; context and documented expectations still matter.

| Proposed change | Often compatible when | Common breaking risk |
|---|---|---|
| Add an optional response field | Clients are required to ignore unknown fields | Strict deserializers reject unknown data or generated equality logic changes |
| Add an optional request field | Omission preserves previous behavior | The server silently changes defaults or validation for old requests |
| Add a new operation | Existing operations are unchanged | Shared limits, permissions, or routing behavior change unexpectedly |
| Add an enumeration value | Clients are required and tested to handle unknown values | Exhaustive switches fail or map the value incorrectly |
| Increase a maximum limit | Resource usage and response behavior remain acceptable | Clients make assumptions about bounded payload size |
| Decrease a maximum limit | Rarely safe without notice and evidence | Previously valid requests are rejected |
| Remove or rename a field | Only after no supported consumer depends on it | Deserialization, display, or business logic fails |
| Change a field type | Almost never transparent | Parsing, comparison, storage, and generated clients break |
| Tighten validation | Old accepted input remains valid | Previously successful requests begin to fail |
| Change ordering | Order was explicitly undefined and clients comply | Pagination, display, or processing order changes |
| Change error status or code | No consumer relies on the old recovery branch | Retry or user-correction behavior becomes wrong |

The value of classification is to force an explicit compatibility discussion. “Small code change” and “small contract change” are unrelated measures.

## Additive changes can still break consumers

Adding rather than removing is generally safer, but “additive” is not a guarantee.

A new response field can cause failure when a client uses strict decoding. A new enumeration value can reach an exhaustive switch with no default branch. A new optional request field can change server behavior when omitted if the provider also changes the default. A new endpoint can alter load patterns enough to affect existing operations.

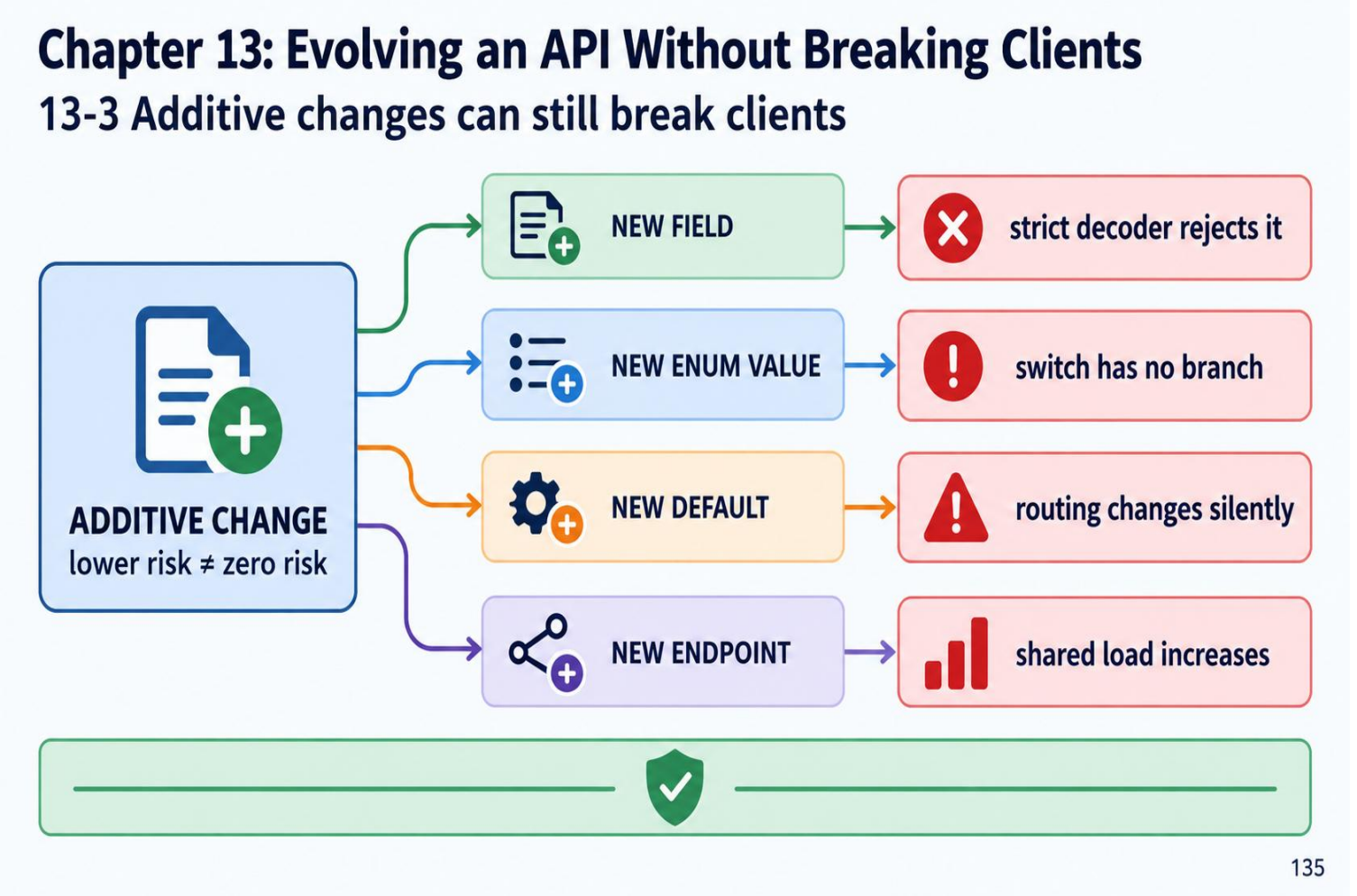



*Figure 13-3: How an additive change can break a real consumer*

Provider and consumer responsibilities should be balanced:

- Providers should avoid unnecessary changes and test compatibility.
- Consumers should ignore unknown response fields unless the contract requires strictness.
- Providers should define how unknown enumeration values are handled.
- Consumers should avoid depending on undocumented order, timing, or fields.
- Both sides should record assumptions that materially affect integration.

This is sometimes called **robust client behavior**: a client depends only on documented guarantees and handles allowed variation without failing. Robustness does not require accepting malformed or unsafe data. It means distinguishing legitimate evolution from contract violation.

## Enumeration values deserve special care

An **enumeration** defines a field whose value comes from a named set, such as `low`, `normal`, `high`, and `urgent`. Enumerations look simple but often become embedded in switch statements, database constraints, user-interface labels, and reporting logic.

Before release, decide whether the set is:

- **Closed:** No new value will appear without a versioned or otherwise coordinated change.
- **Extensible:** New values may appear, and consumers must use a defined fallback behavior.

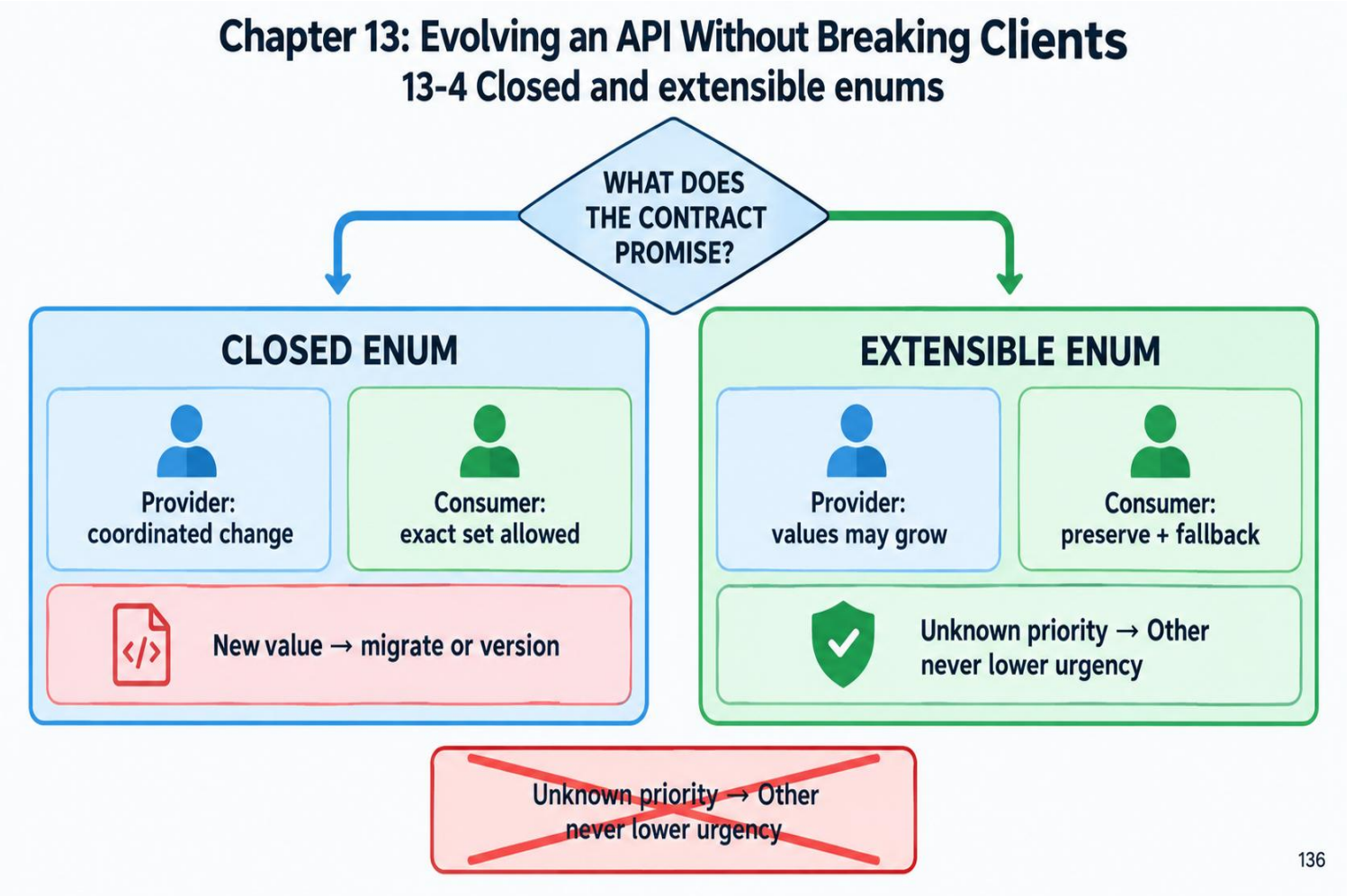



*Figure 13-4: Closed versus extensible enumerations*

For an extensible `priority`, a client might display unknown values as “Other” and preserve the original value when forwarding data. A client must not silently convert every unknown value to `normal` if that can reduce urgency.

The API documentation should not say both “allowed values are exactly these four” and “new values may appear without notice.” Choose and test one evolution policy.

## Optional response fields need clear absence semantics

An optional response field can be absent for several reasons: the value is unknown, the caller lacks permission, the field does not apply, the provider has not calculated it, or an older server does not support it. These meanings should not be mixed casually.

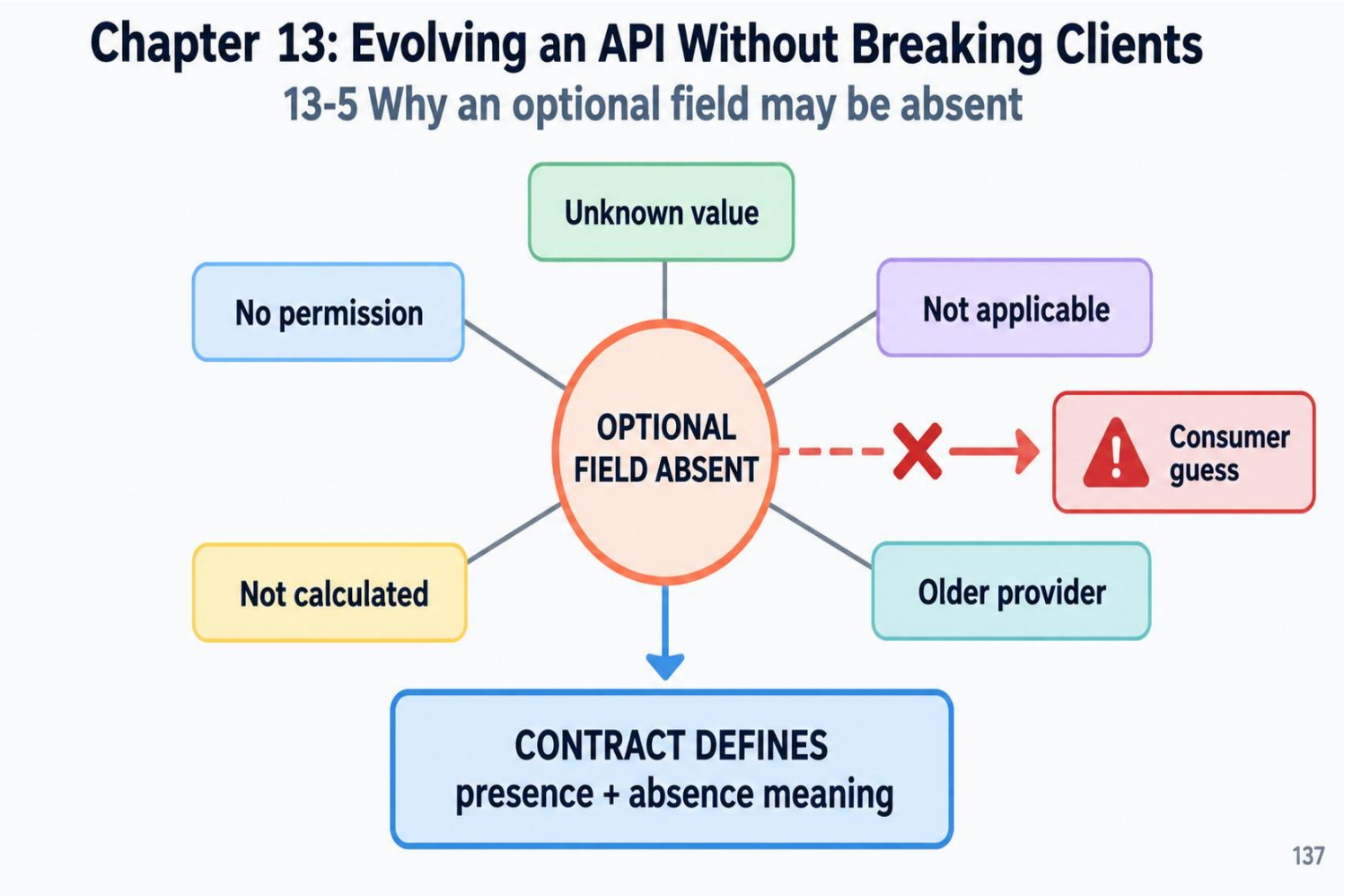



*Figure 13-5: Meanings hidden behind an absent optional field*

When adding an optional field:

- Define what its presence means.
- Define what absence means.
- Avoid changing an old required field into an unreliable optional field.
- Check whether permission-based omission could reveal information through inconsistent behavior.
- Ensure old clients can ignore the addition.

For capability expansion, adding a new optional object can be safer than changing the meaning of an established scalar field. The old field can remain stable while new consumers adopt richer data.

## Versioning identifies a contract line; it does not ensure compatibility

**API versioning** is the practice of identifying distinct contract lines so that incompatible behavior can coexist or migrate over time. Common placement strategies include a path segment, a request header, or a media type.

| Strategy | Example | Strength | Cost or limitation |
|---|---|---|---|
| Path version | `/v2/tickets/tkt_48271` | Visible, easy to route and discuss | Versions become part of resource addresses; duplication is common |
| Header version | `Api-Version: 2` | Keeps resource paths stable | Less visible in links and simple debugging tools |
| Media-type version | `Accept: application/vnd.example.ticket-v2+json` | Ties version to representation negotiation | More complex for clients, gateways, and documentation |

No placement strategy removes the need to define compatibility within a version. If `/v1` changes field meaning every month, the version label provides little protection. Conversely, a stable API may evolve for years through compatible additions without creating a new major version.

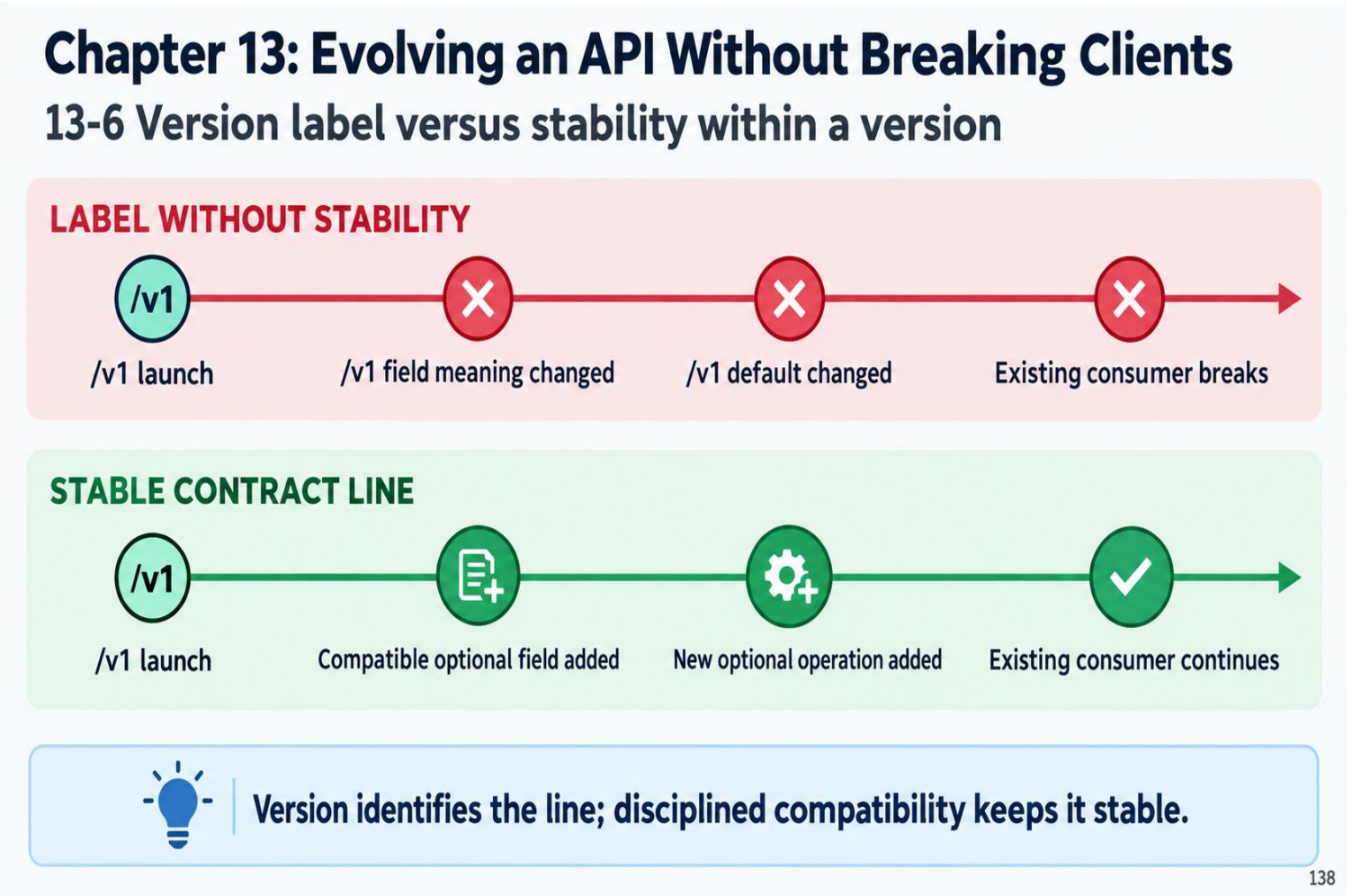



*Figure 13-6: Version label versus stability within a version*

Choose a versioning approach based on consumer capabilities, routing, caching, documentation, organizational practice, and support cost. Apply it consistently. Do not introduce multiple mechanisms for the same distinction without a strong need.

## Version only when the compatibility boundary requires it

A new version is appropriate when the provider must make a change that cannot reasonably preserve supported consumer behavior, such as:

- Replacing a central resource model.
- Changing required request structure substantially.
- Changing the meaning or type of established fields.
- Removing behavior that consumers still use.
- Applying a different security or lifecycle model that cannot coexist transparently.

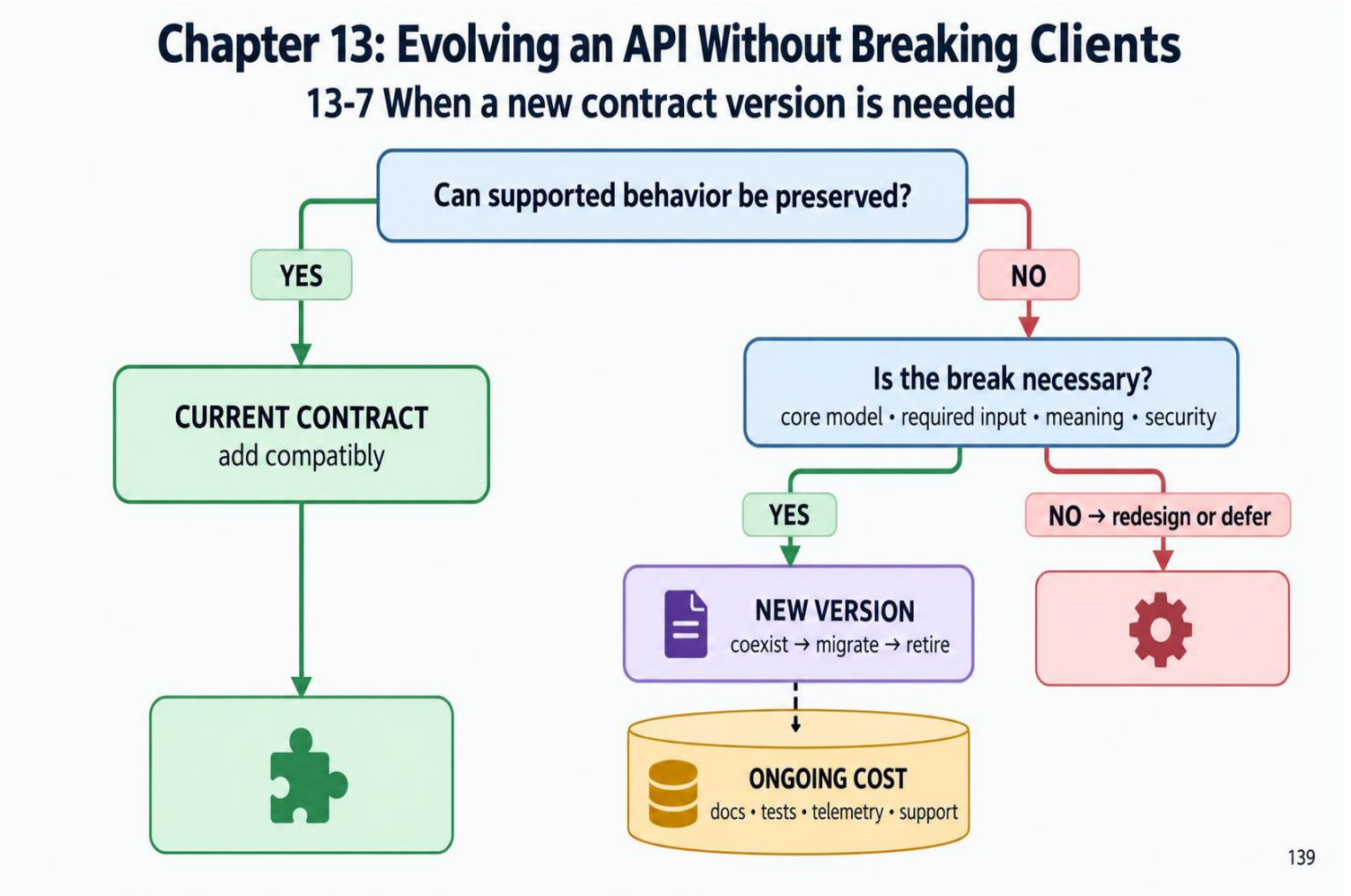



*Figure 13-7: When to create a new contract version*

A new version is not automatically needed for every new field or operation. Excessive versioning fragments documentation, tests, telemetry, and support. It can also postpone necessary retirement by allowing old versions to remain indefinitely.

The decision should compare two costs:

1. The complexity of an additive migration within the current contract.
2. The long-term cost of operating and retiring a parallel contract line.

## Deprecation is a managed period, not a warning label

**Deprecation** marks a supported feature or contract as planned for future retirement and directs consumers toward a replacement. Deprecation does not mean immediate removal.

A credible deprecation plan includes:

- The exact operation, field, version, or behavior being deprecated.
- The supported replacement.
- The date deprecation begins.
- The earliest possible retirement date.
- Evidence of affected consumer usage.
- Migration instructions and test support.
- A communication channel for questions and exceptions.
- A decision process for extending or completing retirement.

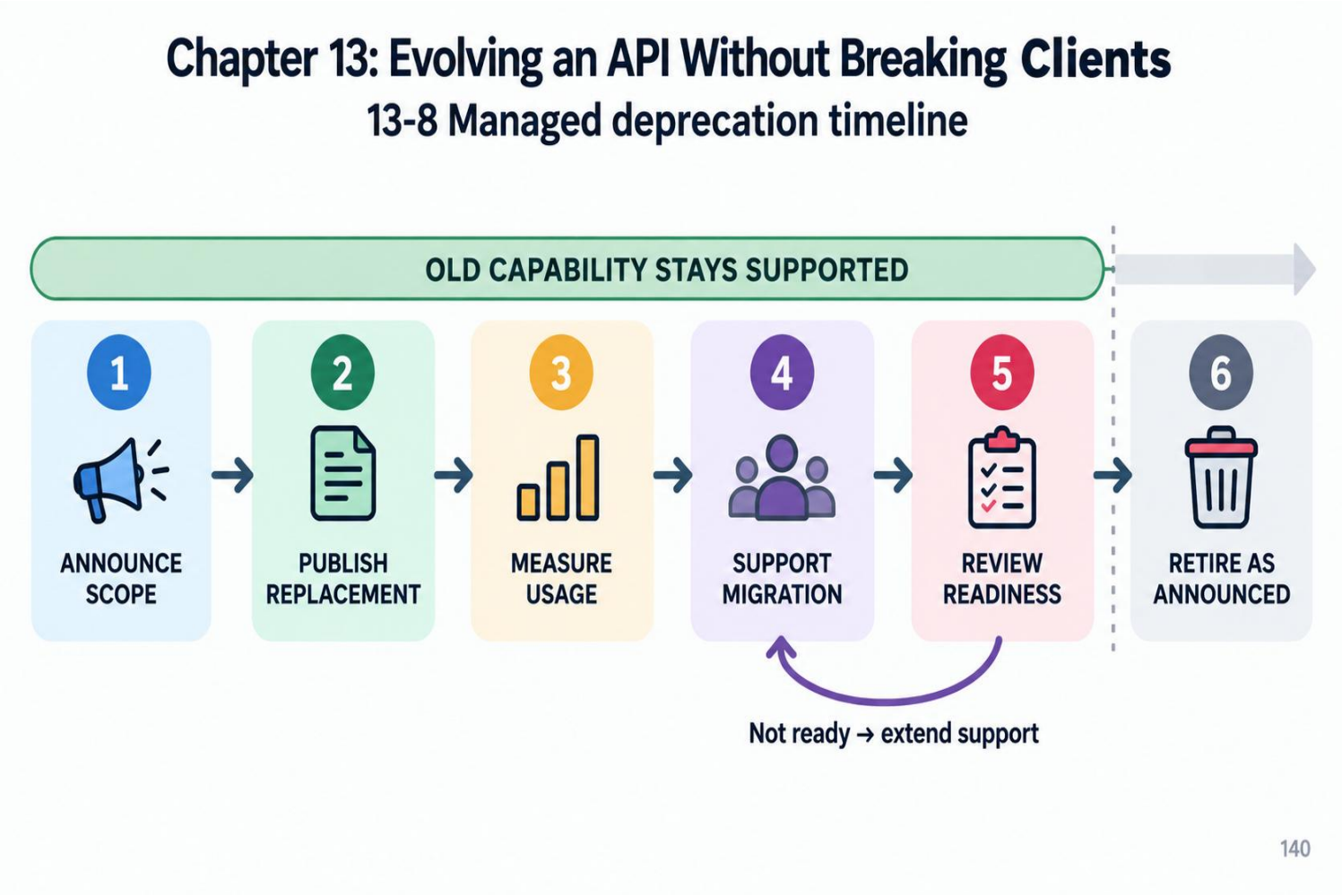



*Figure 13-8: Managed deprecation timeline*

Publishing “deprecated” without a replacement or timeline creates uncertainty. Publishing a retirement date without measuring use creates avoidable outages.

Where appropriate, responses can include standardized or organization-defined deprecation information, but headers alone are insufficient. The owning team should contact known consumers and track migration progress.

## Measure usage before removal

A provider should know which consumers call a deprecated capability, how often they call it, and whether those calls are business-critical. This requires stable consumer identification and operation-level telemetry.

Usage evidence should be interpreted carefully:

- No calls during one quiet week may not prove that a quarterly process is gone.
- A shared credential may hide several consumers.
- A consumer may call only during incident recovery.
- Synthetic monitoring may look like real business usage.

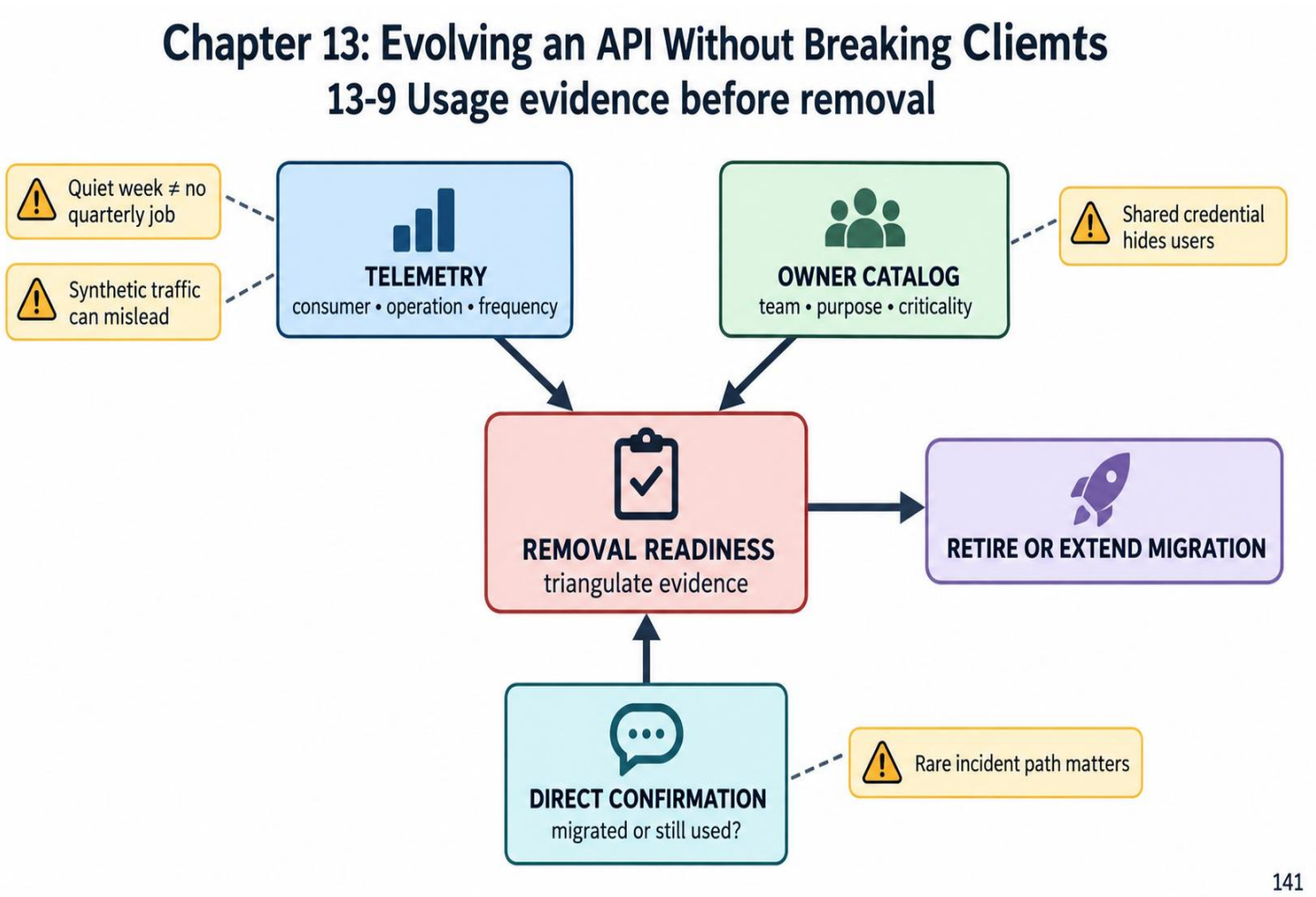



*Figure 13-9: Usage evidence before removal*

Combine telemetry with an ownership catalog and direct confirmation. Retirement is a product and operational decision, not merely a source-code cleanup.

## Parallel versions carry continuing operational cost

Supporting two versions means more than maintaining two route groups. Each version can require:

- Security updates and policy review.
- Documentation and examples.
- Contract and integration tests.
- Monitoring and incident procedures.
- Client support and defect fixes.
- Compatibility with shared downstream systems.
- Separate deprecation and retirement management.

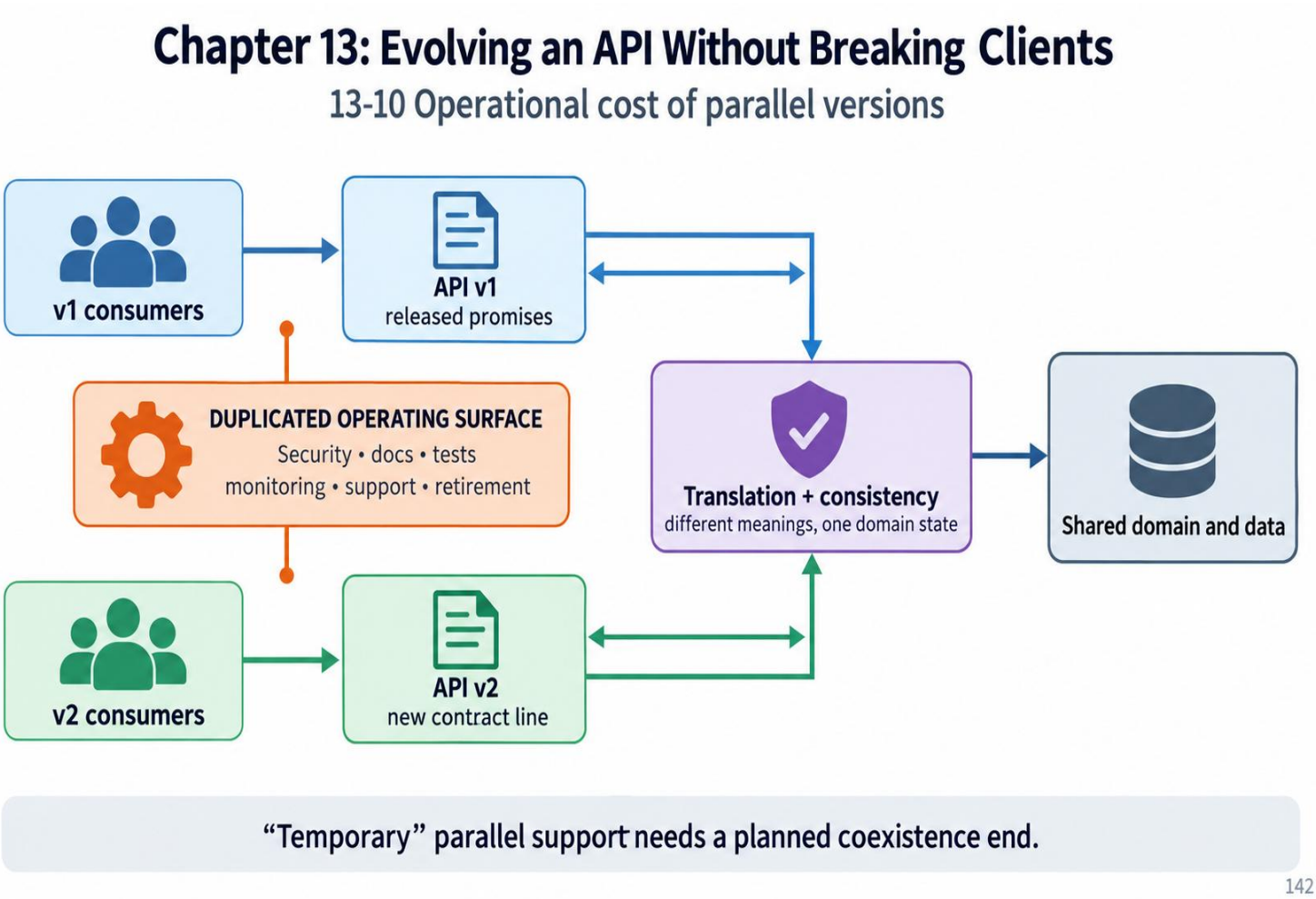



*Figure 13-10: Operational cost of parallel versions*

If both versions write the same underlying data, their different meanings must remain consistent. A field removed from v2 may still be needed by v1. A new v2 state may need a safe representation when read through v1.

Plan the coexistence period before launching the new version. Otherwise, "temporary" support often becomes permanent.

## Capability negotiation can reduce unnecessary versions

**Capability negotiation** allows a consumer and provider to identify whether a particular optional feature or representation is supported. It can be useful when capabilities vary by environment, account, or gradual rollout.

Examples include:

- A documented request preference that enables a richer response.
- A capability document listing supported export formats.
- A new optional operation that consumers discover from documentation and configuration.
- A feature field in an integration onboarding response.

Negotiation should be used sparingly. It creates combinations that must be tested and supported. A hidden feature flag known only to one client is not a durable contract.

Use capability negotiation for genuine optional variation, not to avoid making a clear compatibility or versioning decision.

## Contract migration and data migration are related but different

A **data migration** changes stored data or internal schema. A **contract migration** changes what consumers send, receive, or rely on. One can occur without the other.

For example:

- The provider can move ticket data to a new database with no contract migration.

- The provider can add an `assignment` object while storing the same columns, requiring a contract migration but little data migration.
- A new assignment model may require both stored-data conversion and consumer adoption.

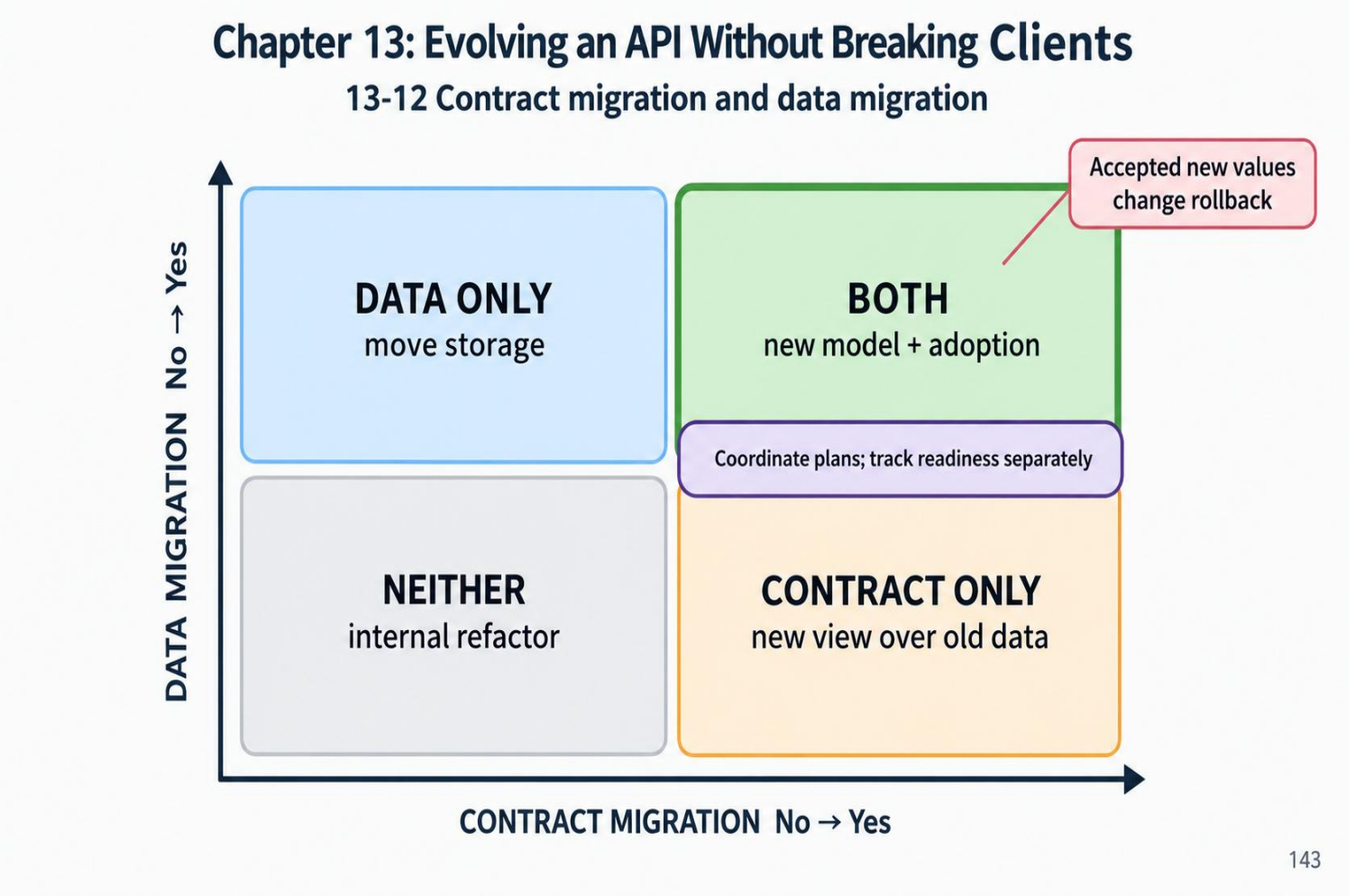



*Figure 13-11: Contract migration versus data migration*

Coordinate the two plans but do not treat successful database conversion as proof that consumers are ready. Rollback also needs separate thought: reverting storage may be difficult after new contract values have been accepted.

## Example: introducing a new priority

Assume the original contract defines:

```
priority = low | normal | high | urgent
```

The business now wants `critical`. The apparently simple change can break:

- A client switch that has exactly four branches.
- A database constraint maintained by a consumer.
- A report that orders priorities through a fixed mapping.
- An alerting rule that treats unknown values as low importance.

A safe path depends on the original promise.

If the enumeration was documented and tested as extensible, the provider can add `critical` after confirming that supported clients apply a safe fallback. It should still communicate the change, update examples, and monitor unknown-value errors.

If the enumeration was effectively closed, options include:

1. Keep the existing `priority` values stable.
2. Add a separate field such as `impactLevel` or `requiresImmediateResponse` when it represents a genuinely different concept.

3. Offer an opt-in representation or new contract version when consumers need the expanded closed set.
4. Migrate consumers before allowing `critical` to appear through the old contract.

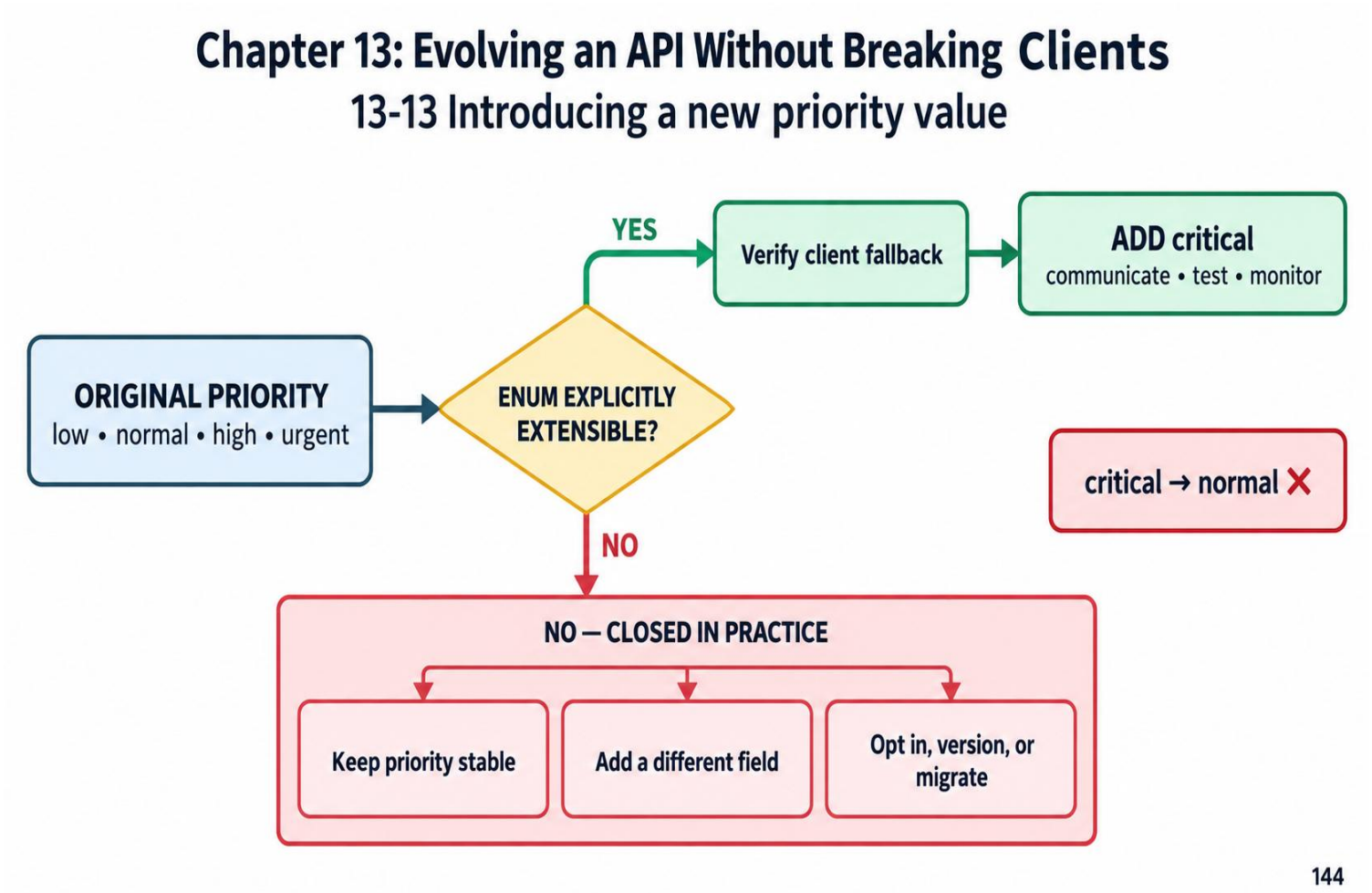



*Figure 13-12: Introducing a new priority value*

Do not map `critical` to `normal` for old clients merely to avoid an error. A compatibility adaptation must preserve the safety-relevant meaning as closely as possible.

## Example: evolving assignment from one user to a richer model

The initial ticket representation contains:

```
{
  "assigneeId": "usr_314"
}
```

The new domain model supports assignment to a queue, a team, and optionally a user. Replacing `assigneeId` with a differently typed `assignee` field would break clients. An additive migration can be:

```
{
  "assigneeId": "usr_314",
  "assignment": {
    "queueId": "queue_account_access",
    "teamId": "team_identity",
    "userId": "usr_314"
  }
}
```

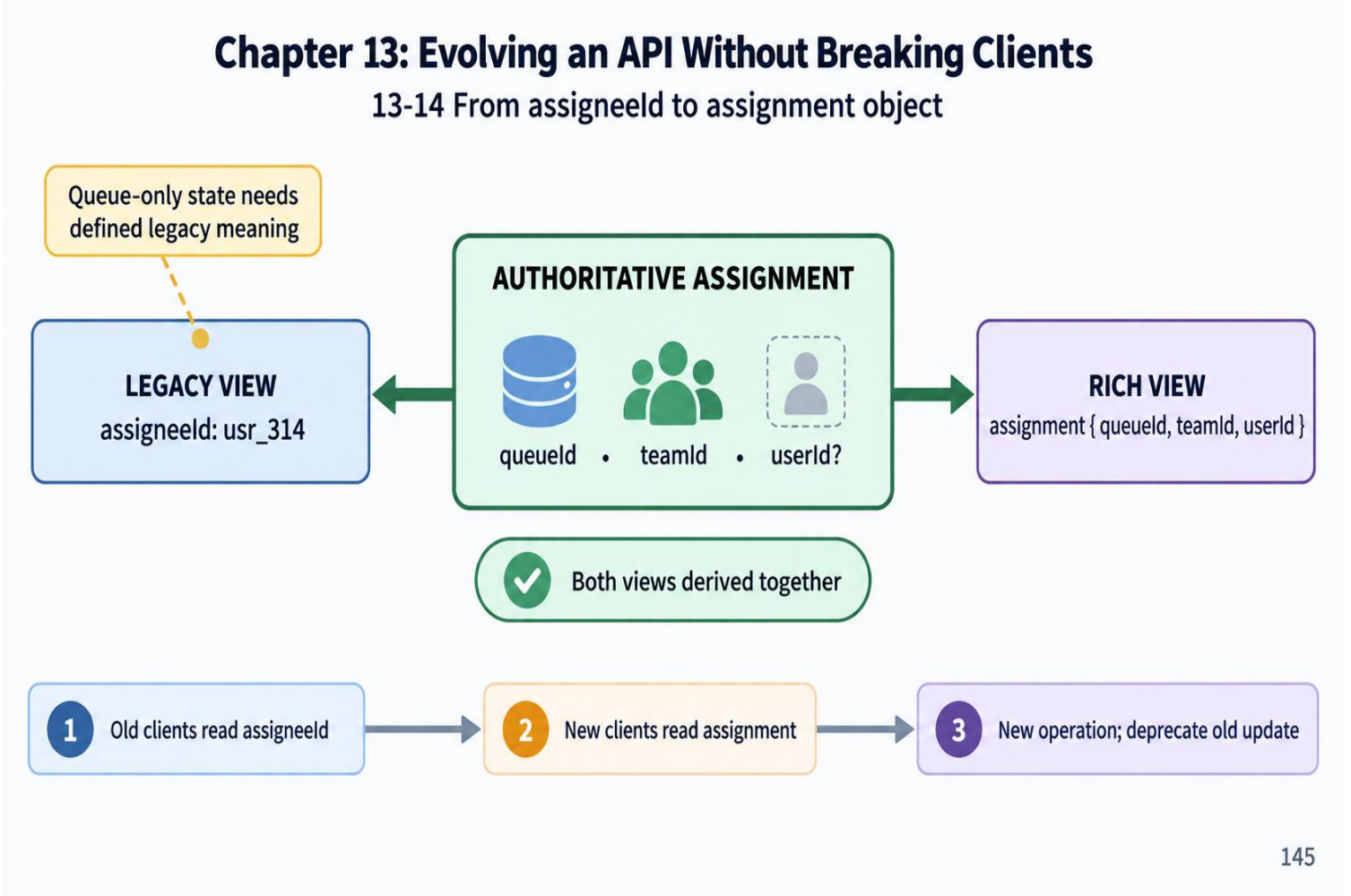



*Figure 13-13: Evolving from `assigneeId` to a richer assignment model*

The migration rules should state:

- `assigneeId` remains the assigned user when one exists.
- `assignment` is the authoritative richer representation for adopting clients.
- Both fields are derived from one server-side assignment state, so they cannot disagree.
- Existing update operations continue during a defined transition period.
- A new assignment operation supports queue-only and team assignment.
- The old update field is deprecated only after consumers can use the new operation.

If a queue-only assignment has no user, the old contract needs a defined representation. It may omit `assigneeId`, return `null` if that meaning was already supported, or avoid exposing queue-only assignments through the old version. The correct choice depends on previous absence semantics; it must not be improvised after rollout.

## Plan rollout as a sequence of reversible steps

A compatible rollout often follows this order:

1. Add provider support that accepts old behavior and can produce only old-visible behavior by default.
2. Publish the updated contract, examples, and migration notes.
3. Update and verify consumers that need the new capability.
4. Enable new output or behavior for controlled consumers or environments.
5. Monitor errors, usage, and fallback behavior.
6. Expand availability.
7. Deprecate old behavior after adoption evidence exists.
8. Retire old behavior through the announced process.

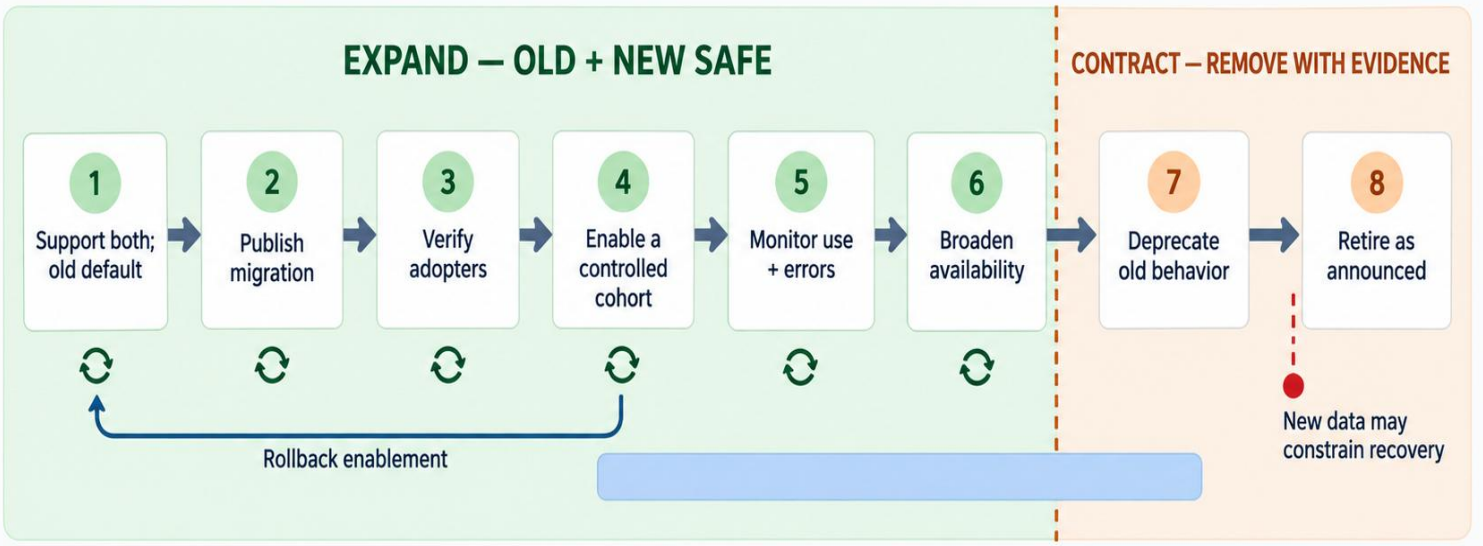





*Figure 13-14: Expand-and-contract rollout*

This sequence is sometimes called **expand and contract**: first expand the system to support both old and new forms, then contract by removing the old form after migration. It reduces the need for provider and consumers to deploy simultaneously.

Not every step is reversible once new data or behavior is accepted. Identify those points before release and prepare recovery procedures.

## Two compatibility mistakes

### "Adding a field can never break a client"

Well-designed clients should ignore unknown response fields, but real clients may use strict decoders, generated models, equality checks, or signature calculations. Treat additions as lower risk, not zero risk, and verify important consumers.

### "Putting `/v2` in the URL solves compatibility"

A version label separates contract lines. It does not define stable behavior within either line, migrate consumers, maintain data consistency, or retire the old version. Those responsibilities remain.

## Compatibility decision checklist

For every proposed contract change, ask:

- Which released consumer behavior could be affected?
- Is the change structural, behavioral, operational, or source-level?
- Does the existing contract require clients to tolerate this variation?
- Are enumeration and absence semantics explicit?
- Can the change be introduced additively with one source of truth?

- Does it require a new contract version, or would versioning add needless cost?
- How will affected consumers be identified and tested?
- What telemetry will confirm adoption and reveal failures?
- Are deprecation, migration, and retirement dates credible?
- How do data migration and contract migration interact?
- Which rollout steps are reversible?

**A provider earns long-term trust by making change predictable: additions are disciplined, incompatibilities are explicit, migrations are observable, and retirement happens only through a managed process.**

## Chapter Summary

- Compatibility is determined by supported consumer behavior, not by whether a provider considers a change small.
- Version identifiers name contract lines but do not remove the need for compatible evolution within each line.
- Deprecation requires discoverable consumers, adoption evidence, support periods, migration guidance, and a controlled removal plan.

## Review Questions

1. How can adding a response field break a poorly written client?
2. When might a new version be justified instead of an additive change?
3. What evidence should be gathered before a deprecated operation is removed?

# Chapter 14 — Designing for Failure, Load, and Performance

**A production API should fail within defined bounds and guide clients toward safe recovery.** Networks delay messages, dependencies become unavailable, callers retry, traffic spikes, and capacity is finite. These conditions belong in the contract rather than being treated as implementation surprises.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Design timeouts, deadlines, retries, backoff, jitter, and retry limits as coordinated reliability mechanisms.
- Use rate limits, load shedding, caching, payload controls, circuit breakers, and asynchronous work to protect capacity.
- Define honest behavior for partial dependency failure and connect reliability decisions to service-level objectives.

## Bound waiting with timeouts and deadlines

A **timeout** is a limit on how long a component waits for an operation before giving up on that attempt. Without timeouts, slow dependencies can consume connections, threads, memory, and caller patience indefinitely.

A **deadline** is the latest time by which the overall operation should complete. A deadline is useful across multiple service calls because each component can see the remaining time rather than starting a fresh full timeout at every hop.

Suppose the agent web application allows three seconds for assignment:

1. The gateway spends 100 milliseconds authenticating and routing.
2. The ticket service spends 300 milliseconds validating and loading the ticket.
3. A downstream workforce service receives the remaining budget, not a new three-second allowance.
4. The ticket service reserves time to handle the downstream result and return a response.

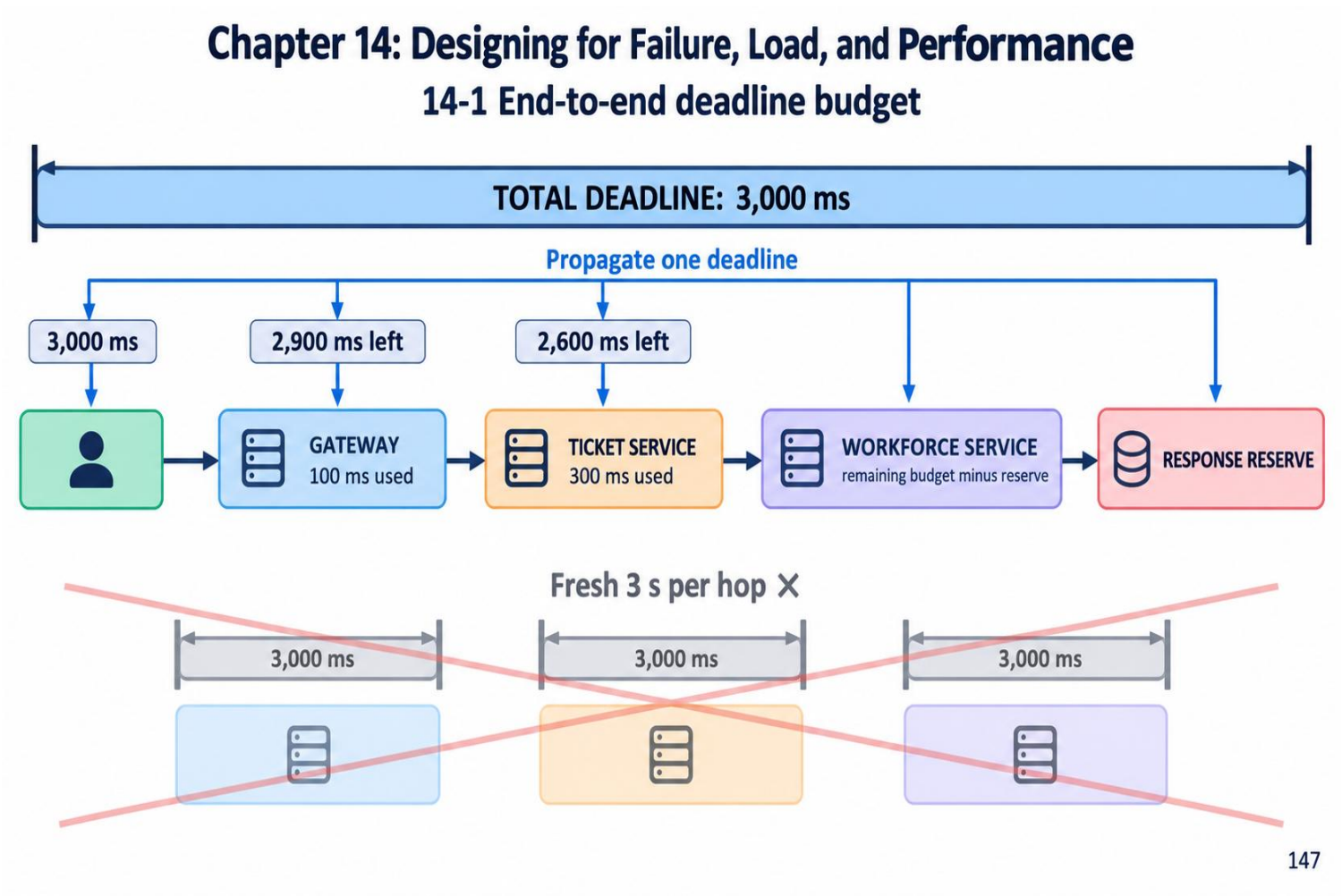



*Figure 14-1: End-to-end deadline budget across service hops*

If each hop independently waits three seconds, a request can greatly exceed the user's expectation. Time budgets should include network, processing, and cleanup time.

Timeouts should be chosen from user needs, service-level objectives, and measured behavior. An extremely short timeout causes avoidable failures; an extremely long timeout delays recovery and consumes capacity.

## A timeout creates uncertainty for state-changing requests

When a client times out, it knows only that no response arrived in time. The provider may not have received the request, may still be processing it, or may have committed the change before the response was lost.

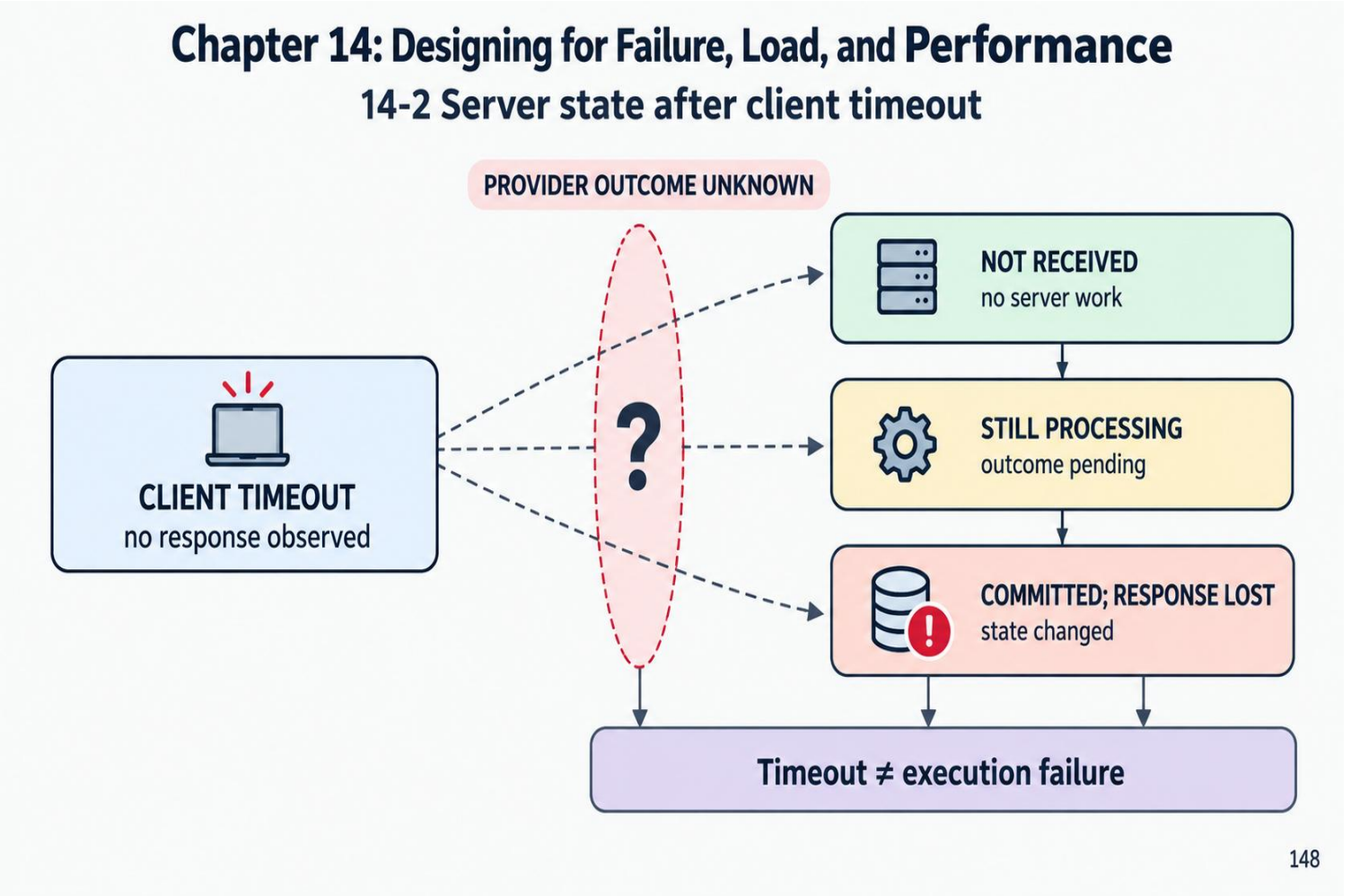



*Figure 14-2: Three possible server states after a client timeout*

For a read, retrying is usually straightforward when the operation is safe. For ticket creation, retrying without the duplicate-request mechanism defined earlier can create two tickets.

The client recovery process should be explicit:

- If the operation is safe or idempotent by contract, retry within the retry policy.
- If an idempotency key protects the logical operation, repeat with the same key and same request.
- If completion can be queried by operation or business identifier, check status before issuing a new logical action.
- If no safe recovery path exists, surface the uncertain outcome rather than guessing.

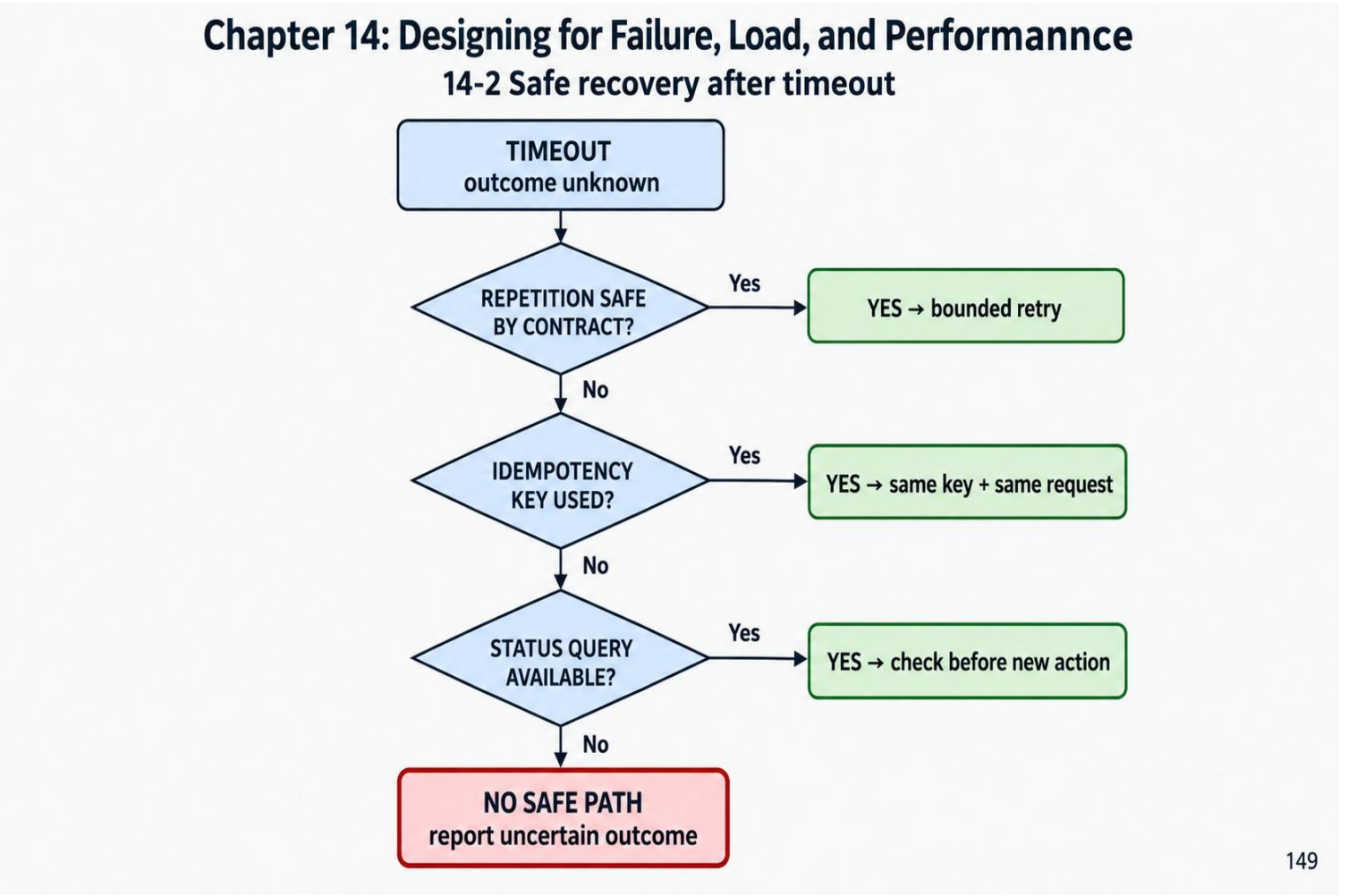



*Figure 14-3: Safe recovery paths after timeout*

**Timeout does not mean failure to execute.** It means failure to observe completion within the waiting limit.

## Retry only failures that may improve on another attempt

A **retry** is a new attempt to perform an operation after a previous attempt did not produce a usable result. Retrying is appropriate only when the cause may be temporary and repeating the operation is safe.

A simple decision table is:

| Condition | Typical client action | Reason |
|---|---|---|
| Connection failure before a safe read completes | Retry within policy | No state change is requested |
| Transient 503 Service Unavailable | Retry if guidance and time budget allow | Capacity or dependency condition may recover |
| 429 Too Many Requests | Wait according to server guidance | Immediate repetition increases overload |
| Validation error | Correct the request; do not retry unchanged | The same input will fail again |
| Authorization denial | Obtain proper authority or stop | Delay does not grant permission |
| Business conflict | Refresh state or choose another action | The requested transition is currently invalid |
| Timed-out protected creation | Repeat with the same idempotency key | The first attempt may already have committed |
| Stale concurrency condition | Retrieve current state before a new update | Repeating the stale write cannot succeed safely |

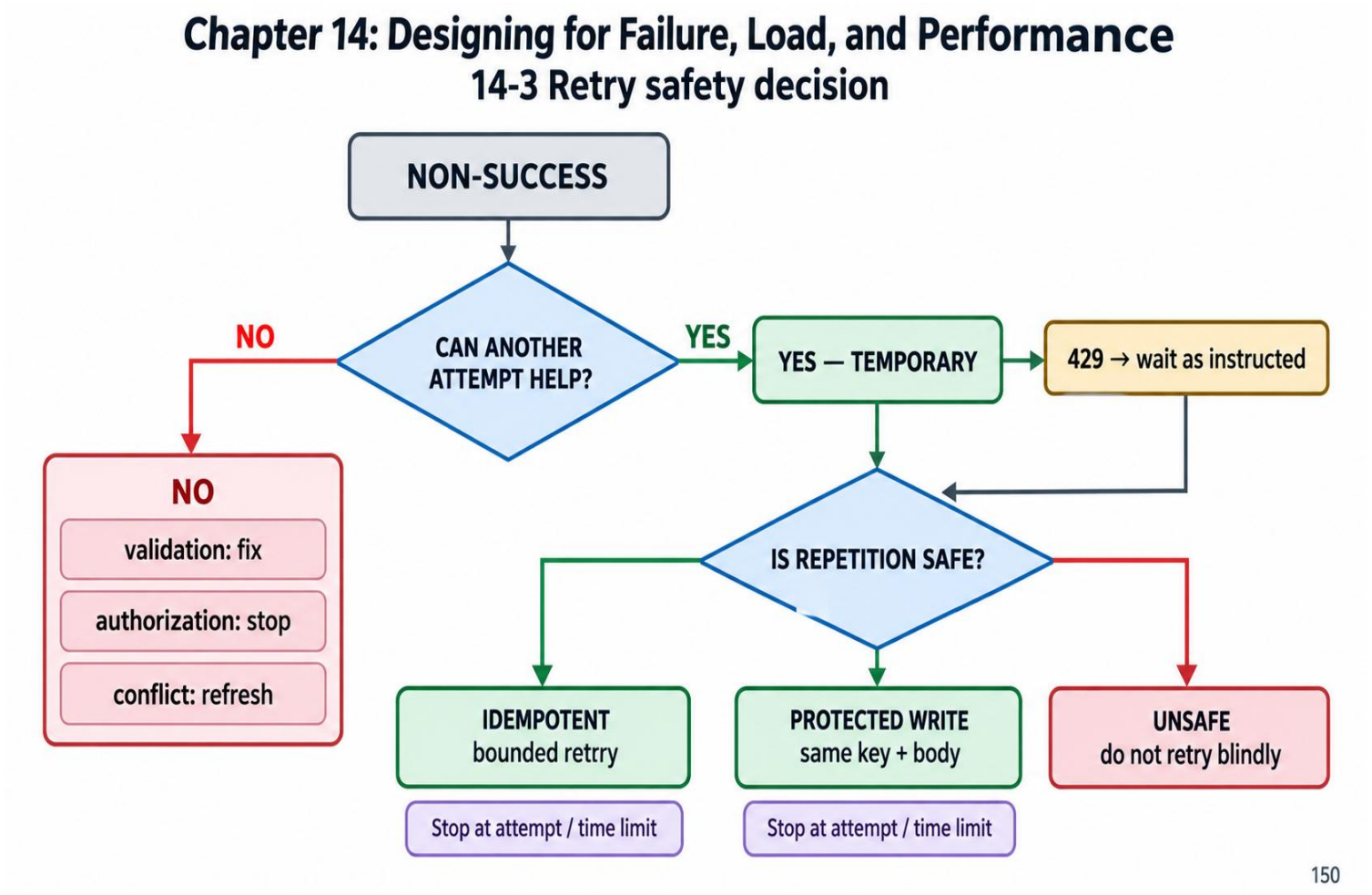



*Figure 14-4: Retry decision by failure type and operation safety*

The table gives common behavior, not a universal response to every status. The contract should identify operation-specific retry guidance.

Retry policies need limits. A caller should stop after a maximum number of attempts, a total time budget, or a deadline. Infinite retries convert temporary failure into sustained load and delayed work.

## Use backoff and jitter to avoid synchronized retries

**Exponential backoff** increases the delay between attempts, commonly by multiplying a base delay after each failure. **Jitter** adds controlled randomness so that many clients do not retry at exactly the same moment.

Without jitter, thousands of clients that fail together may all retry one second later, then two seconds later, creating repeated traffic spikes. This is sometimes called a retry storm.

A conceptual sequence might be approximately:

- First retry after about 0.5 seconds.
- Second retry after about 1 second.
- Third retry after about 2 seconds.

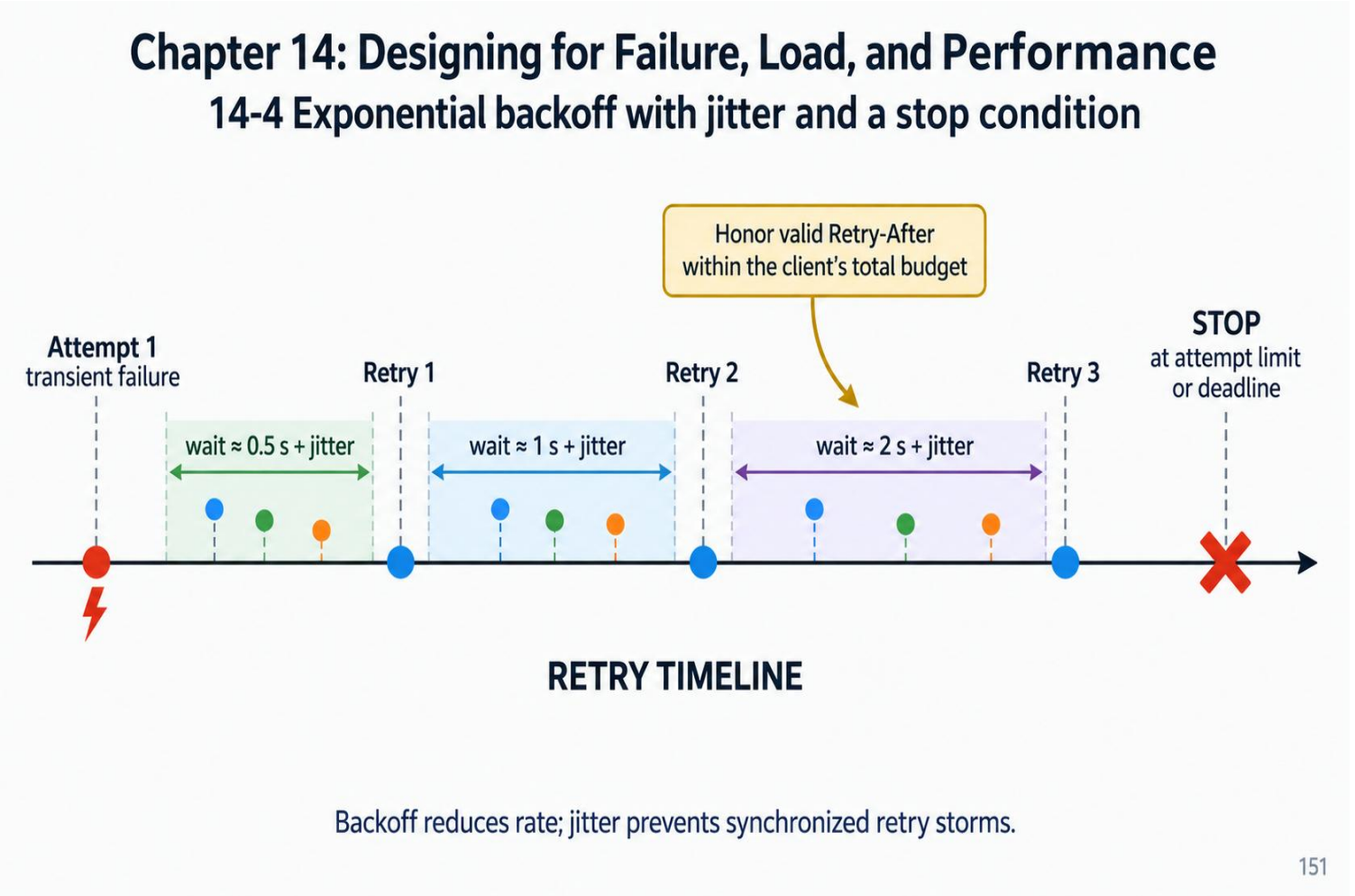



*Figure 14-5: Exponential backoff with jitter and a stop condition*

- Stop when the attempt limit or overall deadline is reached.

The actual delay varies because of jitter and server guidance. Clients should honor a valid `Retry-After` response header where the API uses it, while still respecting their own maximum waiting budget.

Do not prescribe one retry schedule for every operation. Interactive user actions, background jobs, and high-volume integrations have different latency and load requirements.

## Retries multiply across service layers

If a client retries three times, the gateway retries three times, and the service retries a dependency three times, one user action can create many downstream attempts. Layered retries can amplify overload precisely when a dependency is unhealthy.

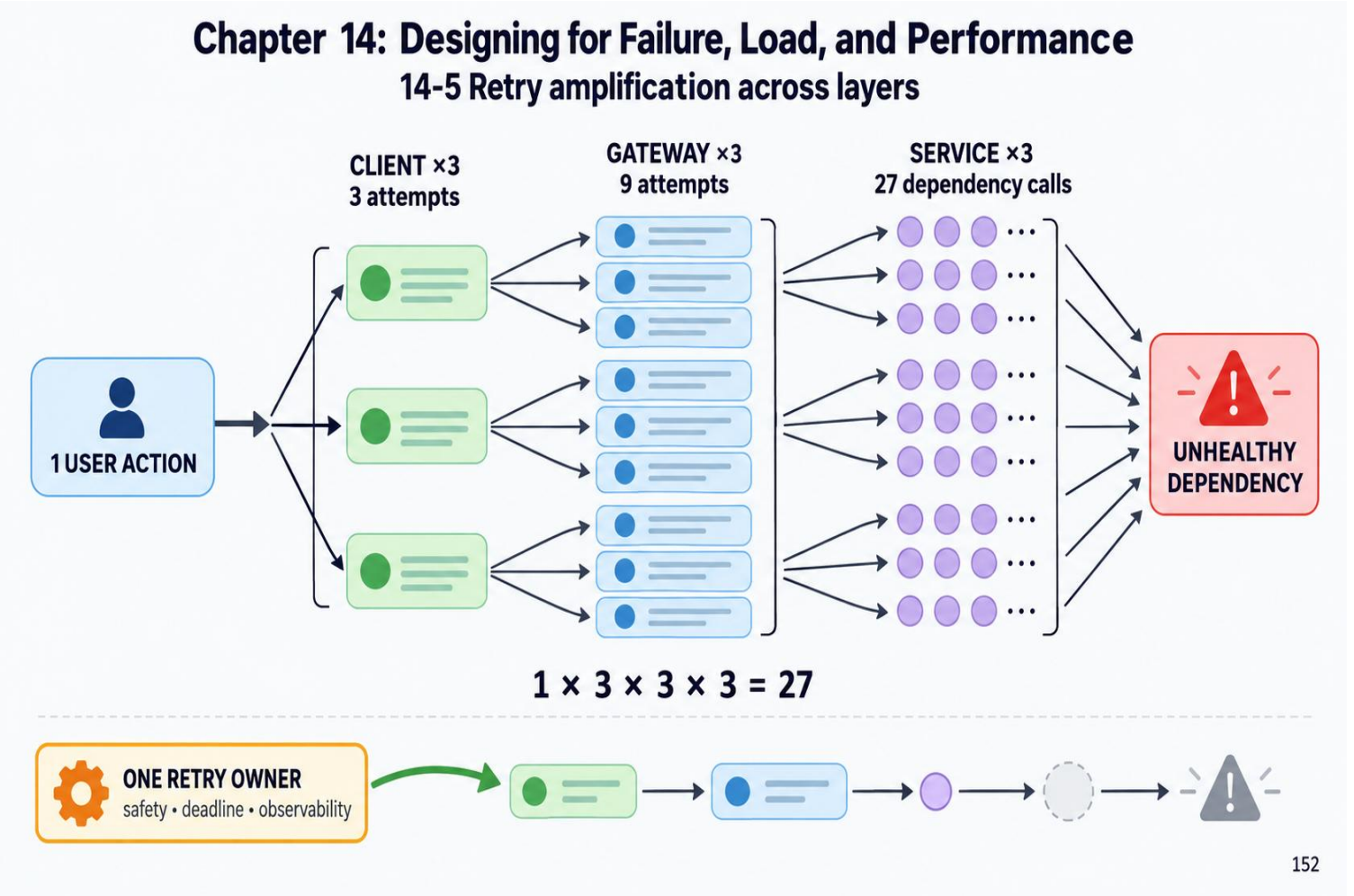



*Figure 14-6: Retry amplification across client, gateway, and service layers*

Assign retry responsibility deliberately:

- Retry near the component that understands whether repetition is safe.
- Avoid retries at every layer by default.
- Propagate deadlines so later attempts do not outlive the original request.
- Count and observe retries separately from original requests.
- Use idempotency or deduplication where repeated state changes can occur.

A gateway retry can be useful for a failed connection to a stateless read replica. It is dangerous for an unprotected creation request whose first response was lost after commit.

## Rate limits protect capacity and fairness

A **rate limit** bounds request frequency over a defined interval or algorithm. A **quota** bounds total permitted usage over a longer period, such as a day or month. They protect different concerns: rate limits address bursts and immediate capacity, while quotas manage sustained allocation or policy.

When a caller exceeds a defined rate limit, an HTTP API commonly returns `429 Too Many Requests`. The response should include the normal error structure and, where possible, useful retry guidance.

For example:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/problem+json
Retry-After: 8

{
  "type": "https://api.example.com/problems/rate-limit-exceeded",
  "title": "Request rate limit exceeded",
  "status": 429,
  "code": "RATE_LIMIT_EXCEEDED",
  "detail": "Retry after the indicated delay.",
  "requestId": "req_71d302"
}
```

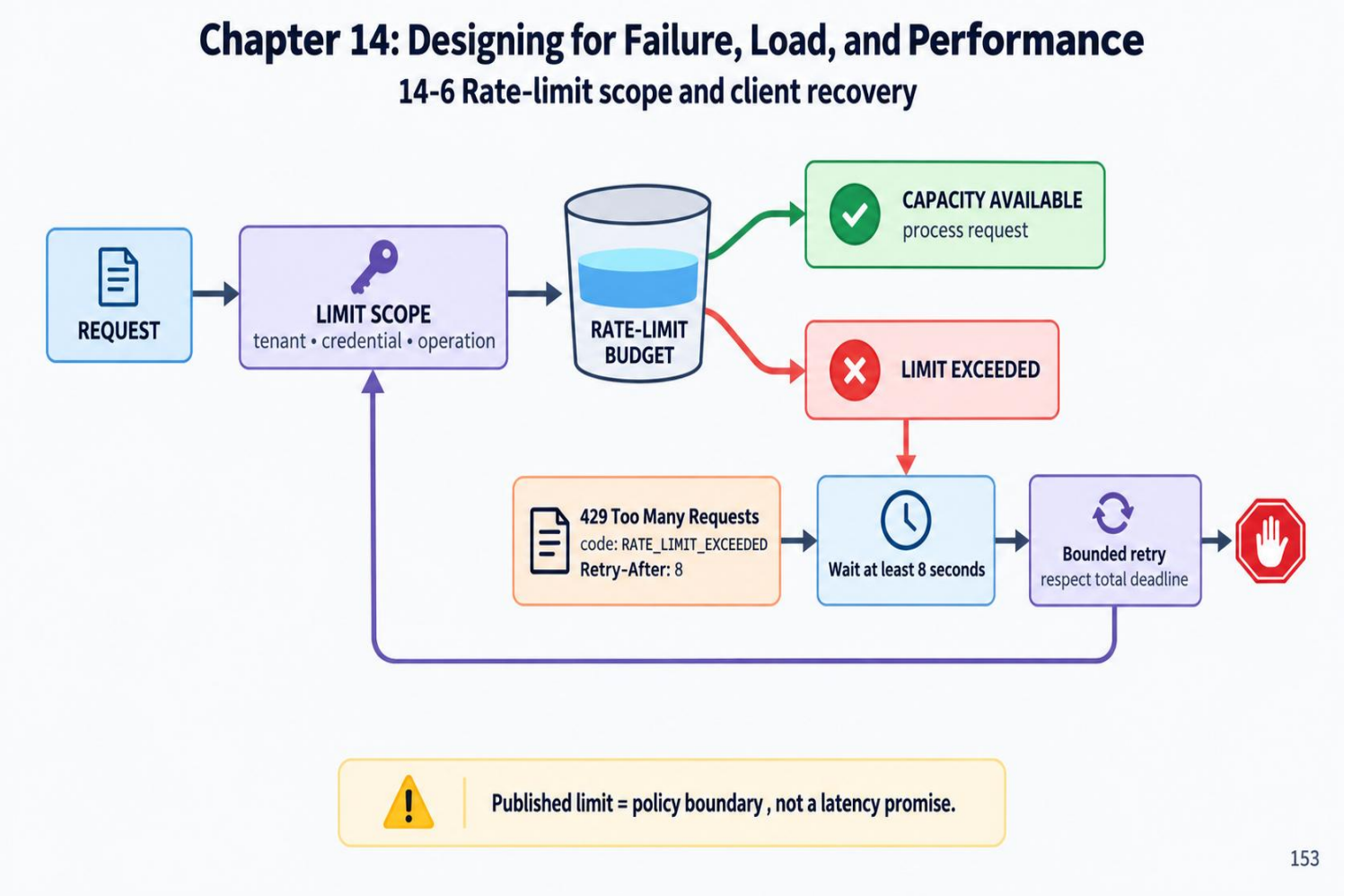



*Figure 14-7: Rate-limit scope and client recovery*

The contract should define the useful parts without exposing an implementation that may change. Consumers need to know:

- Which credential, tenant, user, or operation the limit applies to.
- Whether limits are shared across operations.
- Whether short bursts are allowed.
- How a limited request is reported.
- Whether the provider supplies retry timing or remaining-capacity metadata.

Do not promise exact capacity that the service cannot sustain. A published limit is an upper policy boundary, not necessarily a latency guarantee at every moment.

## Load shedding preserves essential service

**Load shedding** is the deliberate rejection or deferral of work when accepting it would endanger overall service health. Returning a bounded failure can be safer than allowing every request to wait until the entire system collapses.

Possible policies include:

- Rejecting expensive exports before simple ticket retrievals.
- Limiting low-priority background work during an incident.
- Returning `503 Service Unavailable` when the service cannot safely accept more work.
- Moving suitable operations to an asynchronous queue.
- Applying per-tenant fairness so one consumer cannot exhaust shared capacity.

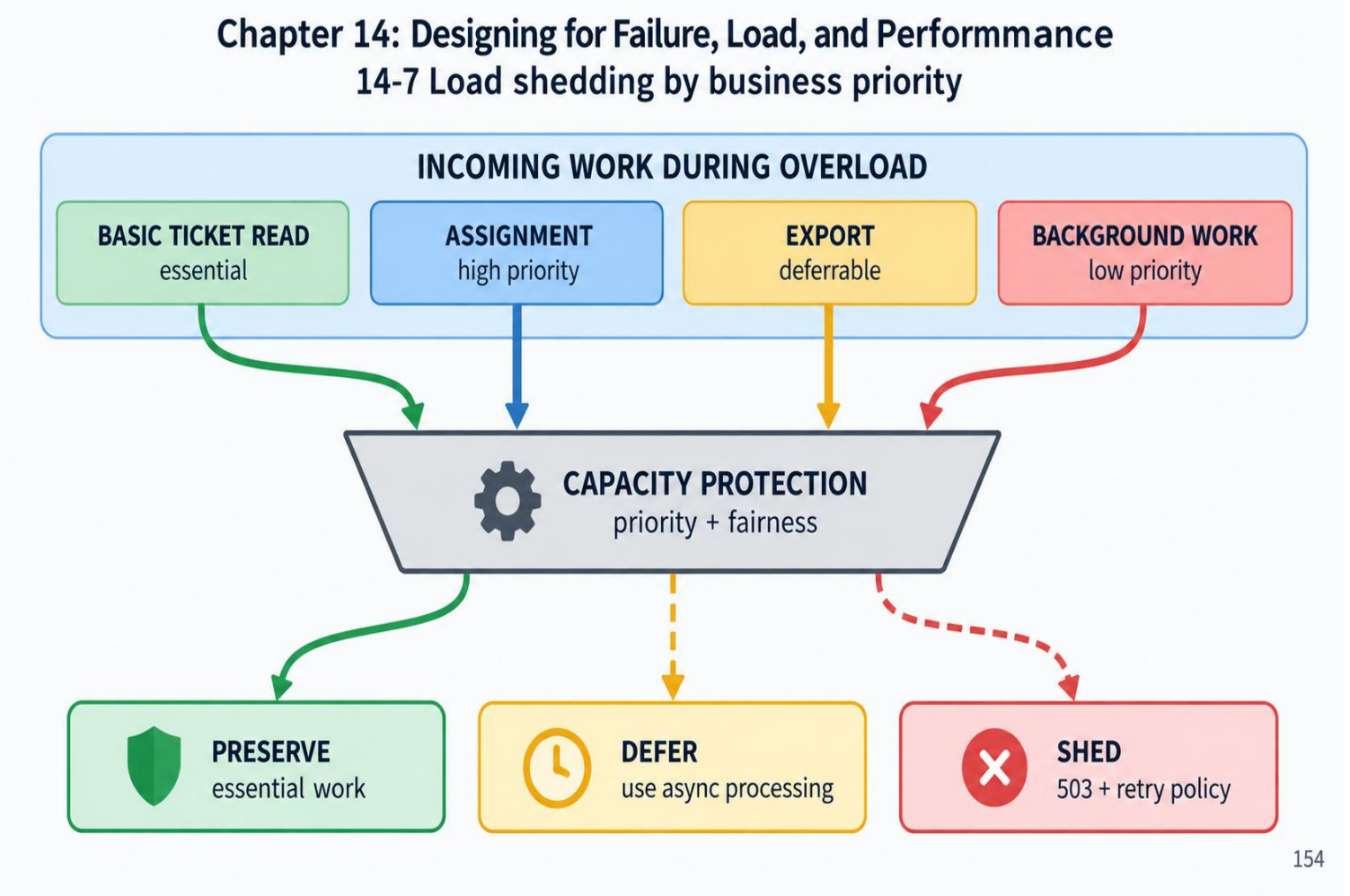



*Figure 14-8: Load shedding by business priority*

The priority policy should reflect business importance and contractual obligations, not whichever endpoint is easiest to reject. Clients need a consistent error and retry policy for shed requests.

## Caching changes observable behavior

**Caching** reuses a previous response when defined conditions indicate that the stored response remains suitable. It can reduce latency and provider load, but it also affects freshness, privacy, invalidation, and authorization.

A cacheable response needs a clear policy. Common HTTP mechanisms include:

- `Cache-Control` directives that state whether and for how long a response may be reused.
- Validators such as `ETag` that let a client or cache ask whether its stored representation is still current.
- Conditional requests that return `304 Not Modified` when the cached representation can be reused.

For example:

```
GET /public/status-categories HTTP/1.1
If-None-Match: "status-categories-v4"
```

If unchanged, the provider can respond:

```
HTTP/1.1 304 Not Modified
ETag: "status-categories-v4"
Cache-Control: public, max-age=3600
```

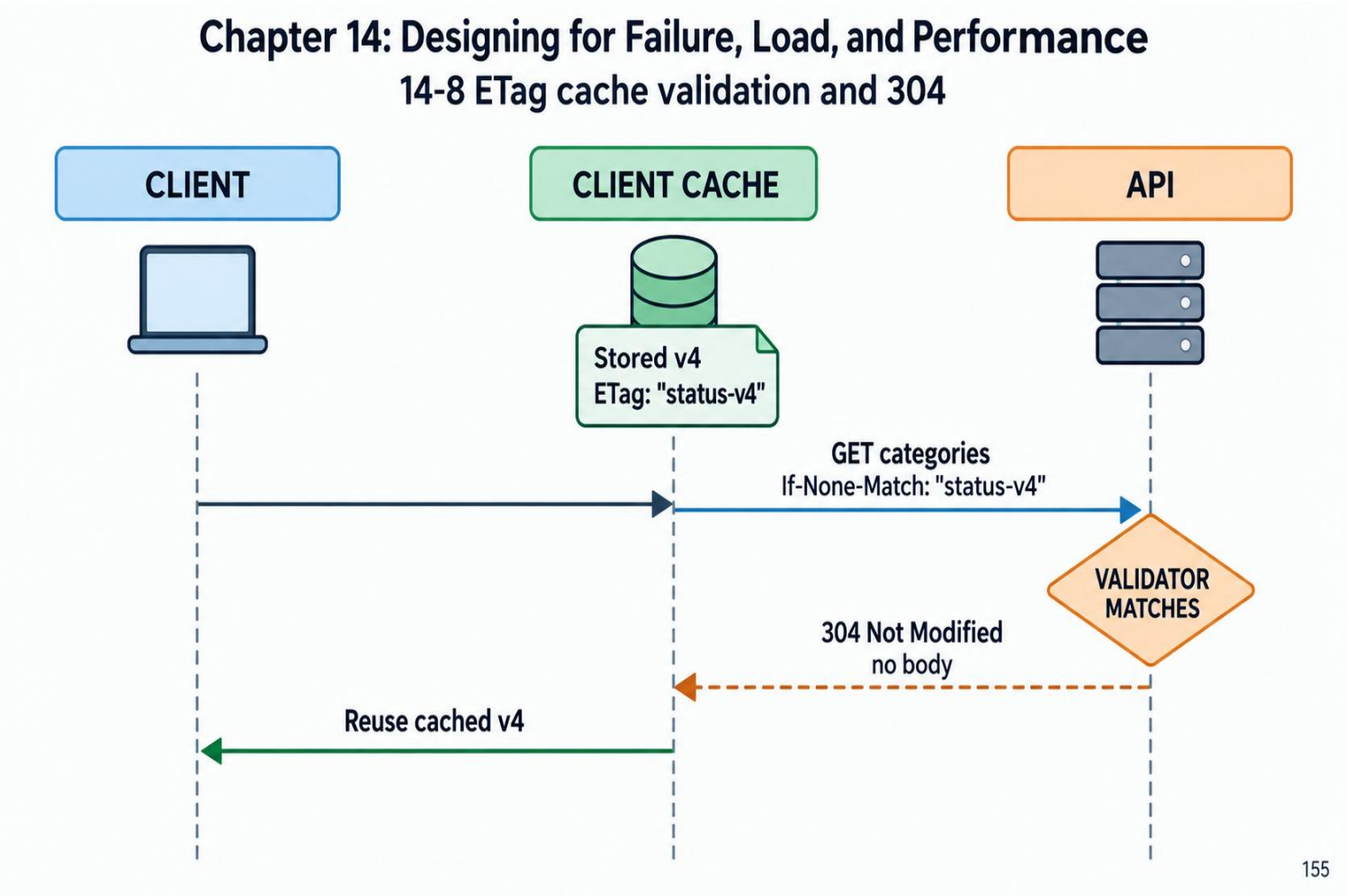



*Figure 14-9: HTTP cache validation with `ETag` and `304`*

No response body is needed because the client already has the representation.

This example is appropriate for public, slowly changing reference data. A user-specific ticket response requires more restrictive handling. Shared caches must not serve one user's protected data to another. Authentication, `Vary` behavior, cache-control directives, and gateway configuration must agree.

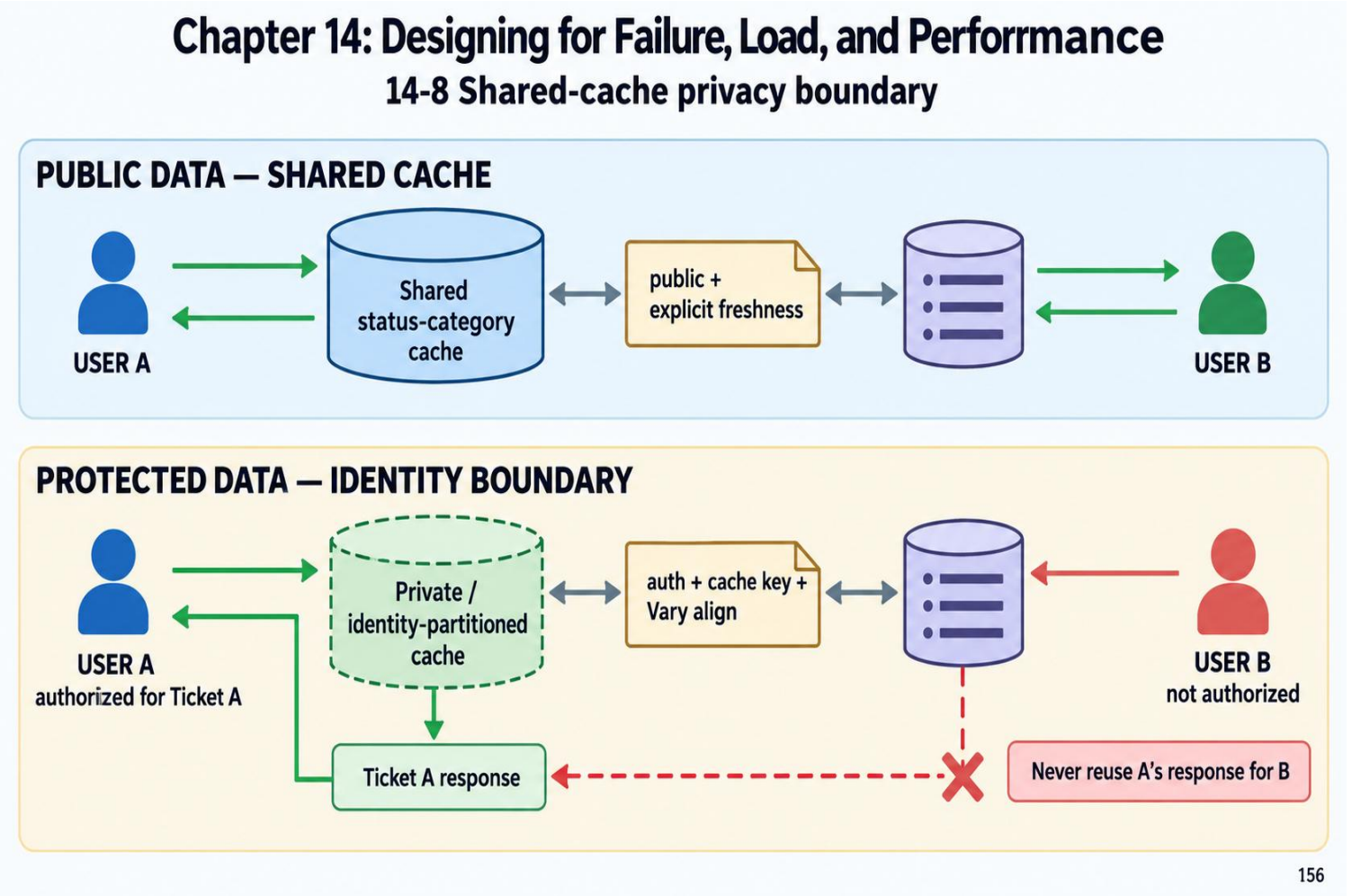



*Figure 14-10: Shared-cache privacy boundary*

Caching is part of the contract because consumers observe freshness and may depend on validators. Changing a response from uncached to ten-minute caching can make updates appear delayed even though the response schema is unchanged.

## Decide freshness from the use case

Not all data needs the same freshness:

- Ticket assignment shown to an agent may need near-current state.
- Public status-category descriptions may be cached for an hour.
- A historical monthly report may use a fixed snapshot.
- An audit view may require authoritative retrieval rather than a stale shared cache.

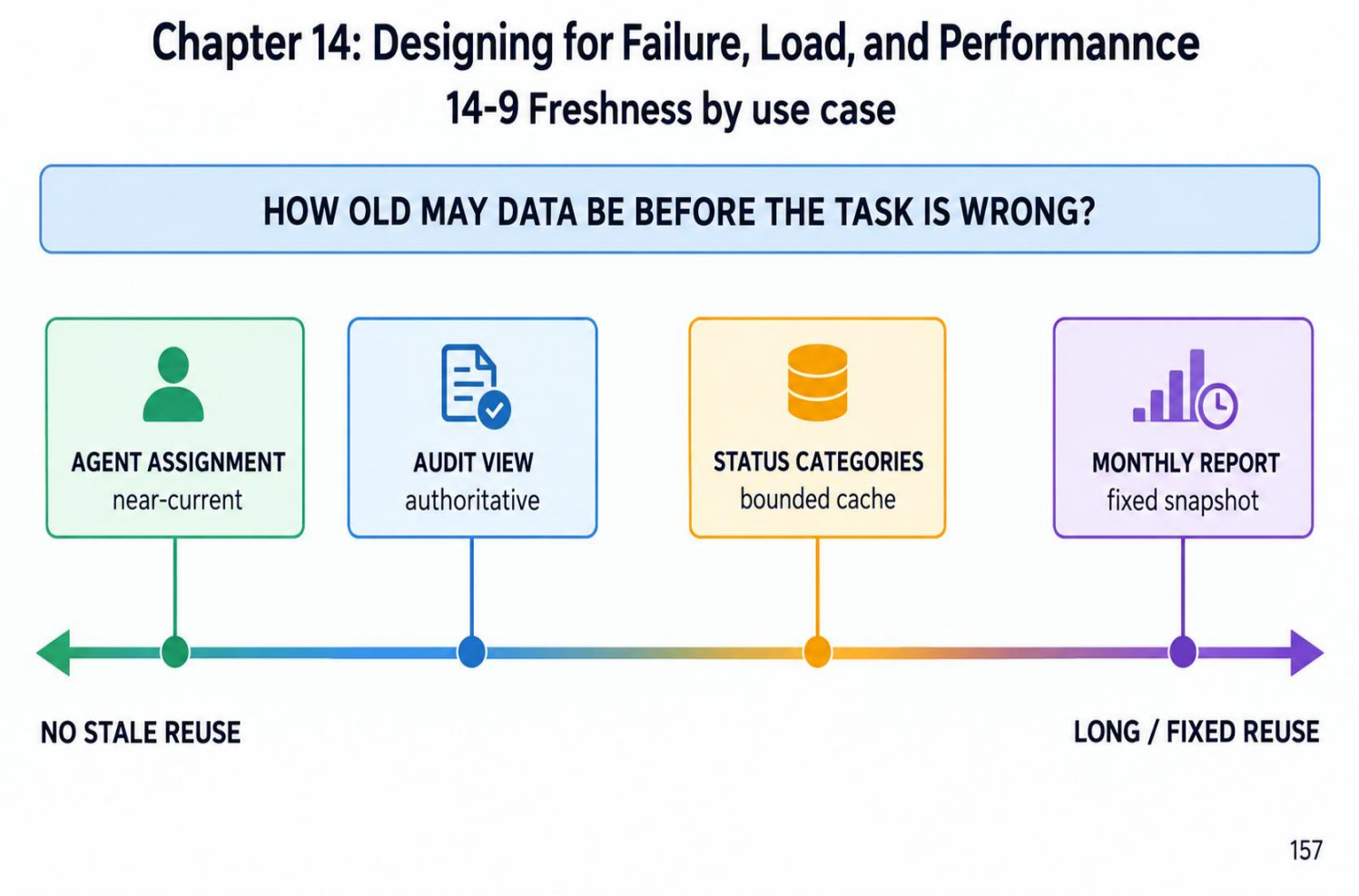



*Figure 14-11: Freshness requirements by use case*

Define acceptable staleness in business terms. “Use caching for performance” is incomplete without stating how old a reused response may be and how a consumer can request or obtain fresh state when necessary.

Avoid making every response uncacheable by habit. Also avoid caching protected or rapidly changing data merely because a gateway supports it.

## Control payload size deliberately

Large payloads increase transfer time, memory use, serialization cost, and failure impact. Contract-level controls include:

- Bounded collection pages.
- Summary representations for lists and detailed representations for individual resources.
- Explicit attachment download operations rather than embedding large content.
- Compression for suitable content when client and server support it.
- Maximum request and response sizes.
- Optional field selection only when a real use case justifies its complexity.

Compression reduces network bytes but consumes processing capacity and may provide little benefit for already compressed files. It should be configured and measured rather than assumed to be free.

A client that needs 10 ticket identifiers should not be forced to download full descriptions, comments, and attachment metadata. Conversely, creating many tiny calls for fields commonly needed together can increase latency and load. Design representations around real tasks.

## Watch request fan-out

**Fan-out** occurs when one incoming request triggers calls to several downstream services or data sources. Fan-out can make a simple-looking operation slow and fragile because its success depends on multiple components.

Suppose `GET /tickets/{id}` synchronously calls:

- The ticket database.
- A user directory for the assignee name.
- A customer service for account tier.
- An attachment service for file metadata.
- A workforce service for queue details.

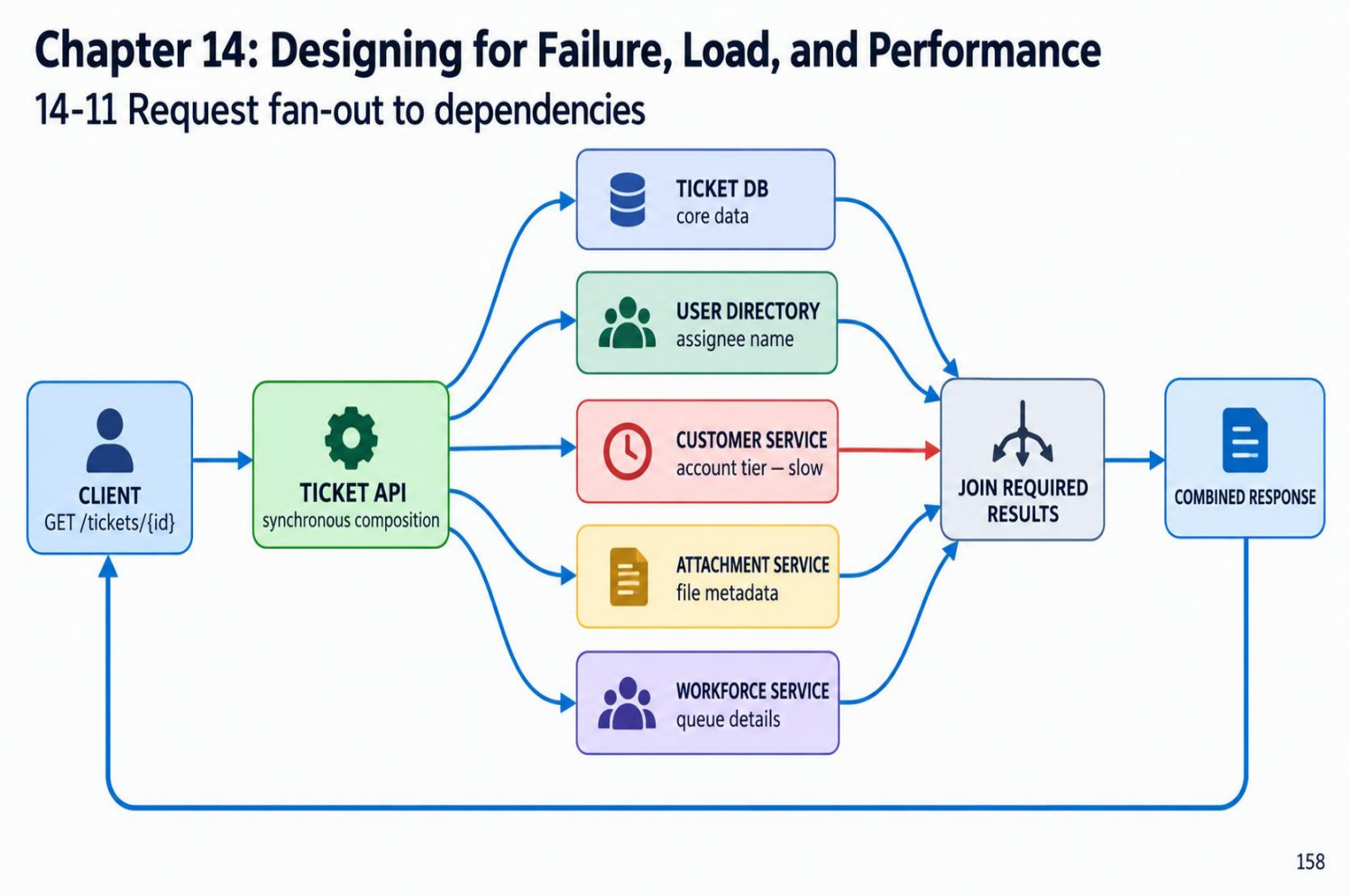



*Figure 14-12: Request fan-out to five dependencies*

The response now inherits latency and failure from all five sources. One slow optional dependency may delay essential ticket data.

Possible design responses include:

- Store stable display data when appropriate.
- Return identifiers and let clients retrieve optional details only when needed.
- Separate core and expanded representations.
- Use bounded parallel calls and explicit partial-data rules.
- Build a **read model**, which is a data view prepared specifically for efficient retrieval, for the consumer task.
- Move expensive aggregation to an asynchronous report.

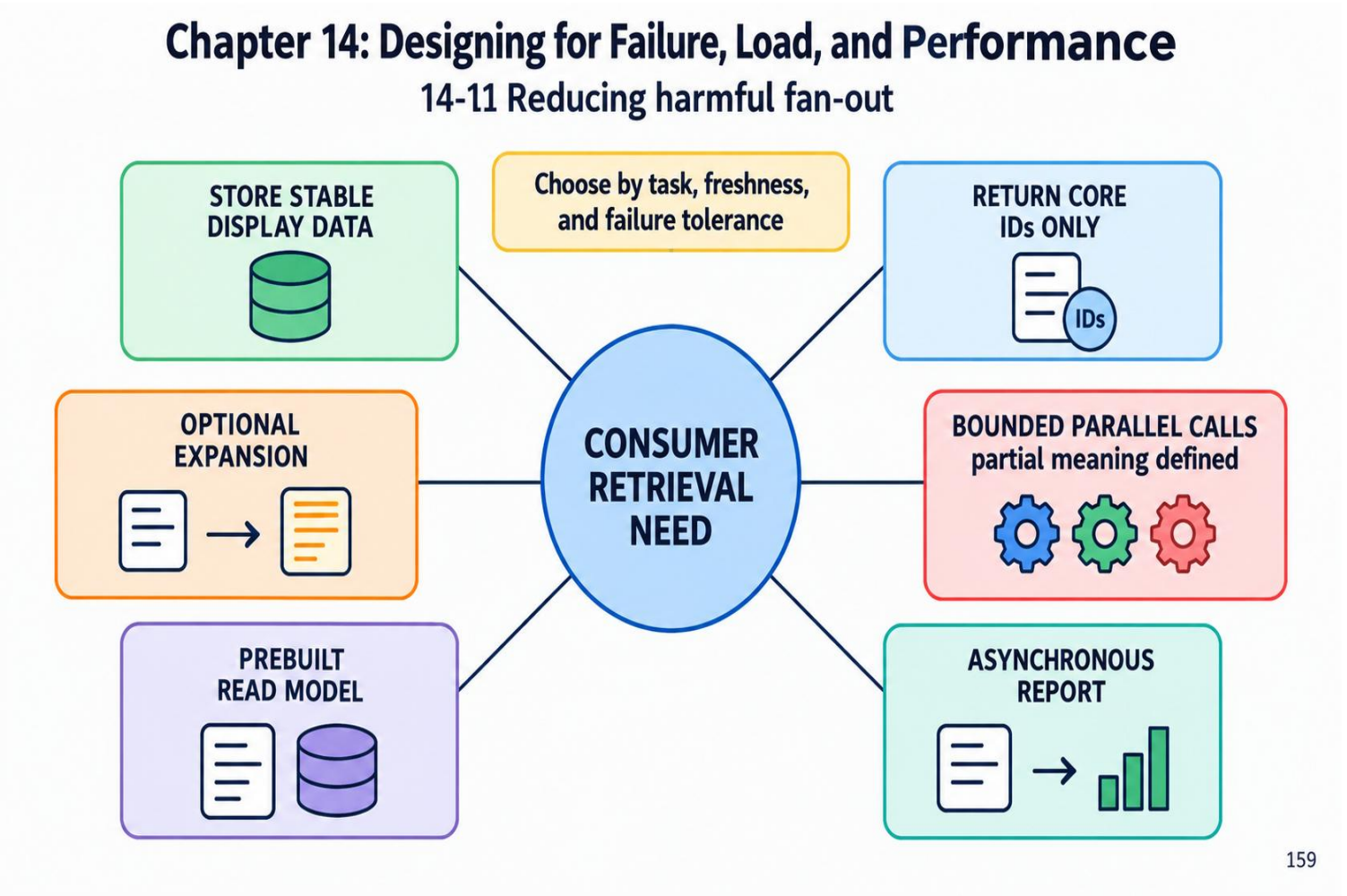



*Figure 14-13: Ways to reduce harmful fan-out*

Do not expose internal fan-out directly as a confusing mixture of success and missing fields. If partial data is allowed, define which fields are optional, how incompleteness is signaled, and whether the result may be cached.

## Partial dependency failure needs an honest response

When an operation depends on several systems, the provider must decide what success means.

For a ticket retrieval, the core ticket may be available while optional customer-tier enrichment is unavailable. The API may return the core ticket without the enrichment only if the contract defines that field as optional and its absence is safe.

For ticket assignment, if eligibility validation is required before commit, the provider must not return assignment success when that validation could not be performed. A bounded server error is more honest than a successful response with invalid state.

The decision follows the business guarantee:

1. Identify the minimum facts required for a truthful success.
2. Separate optional enrichment from required processing.
3. Fail when a required fact cannot be established.
4. Signal documented partial data only where consumers can use it safely.
5. Avoid leaking raw downstream errors or topology.

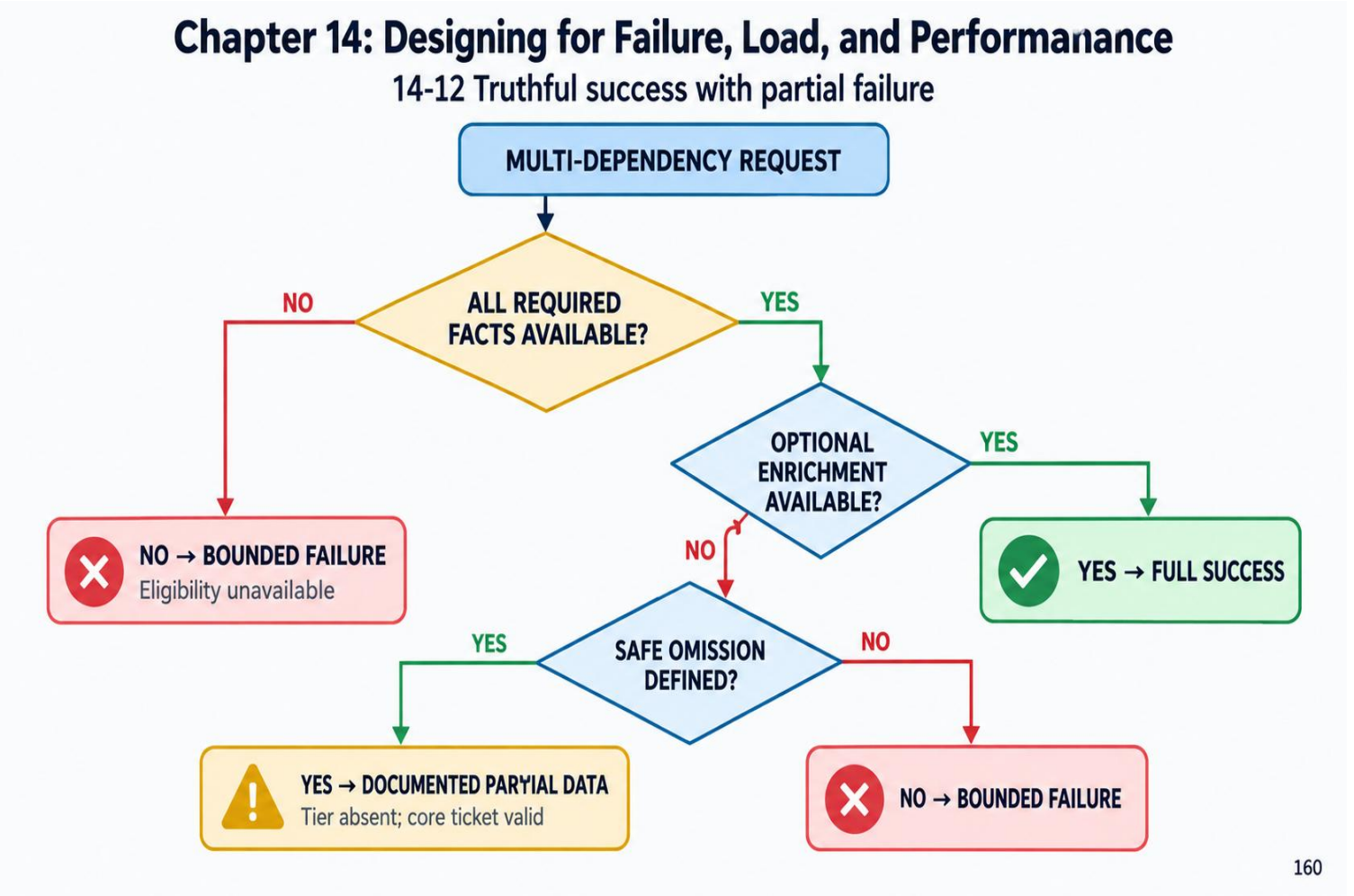



*Figure 14-14: Truthful success under partial dependency failure*

## Circuit breakers limit repeated calls to an unhealthy dependency

A **circuit breaker** is an operational pattern that temporarily stops calls to a dependency after repeated failures, allowing fast failure or fallback while the dependency recovers. The name comes from an electrical breaker that opens a circuit to prevent further damage.

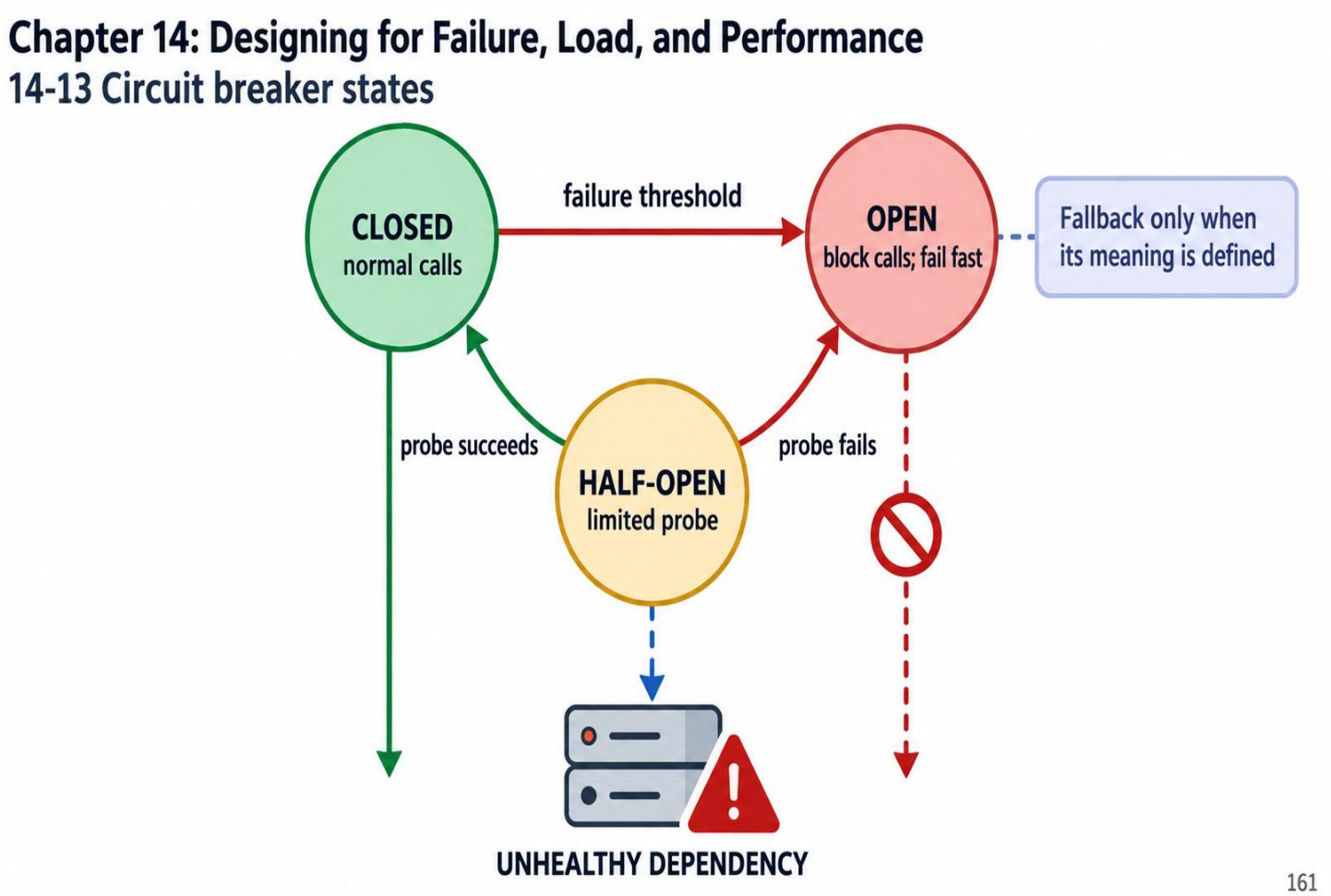



*Figure 14-15: Circuit breaker states and client-visible behavior*

From the API contract perspective, a circuit breaker affects latency and failure behavior. When open, the provider may quickly return a defined unavailable error instead of waiting for another timeout. If a fallback exists, the contract must state whether the result may be stale or incomplete.

Circuit breakers do not repair a dependency and should not hide required failures behind misleading success. Their implementation is operational, but their client-visible consequences require design.

## Bulk and asynchronous work can reduce synchronous pressure

Many small calls can be inefficient, but a large synchronous batch can create long transactions, large failures, and difficult retry semantics. Alternatives include:

- A bounded bulk-read operation for a known set of identifiers.
- An asynchronous export operation for large result sets.
- Event publication for downstream processing that need not block the request.
- A purpose-built summary endpoint for a common dashboard.

Choose based on consumer need and failure model. Do not add a generic batch mechanism before defining per-item results, limits, permissions, idempotency, and timeout behavior.

Long-running asynchronous contracts are developed in Chapter 15.

## Service-level objectives shape contract decisions

A **service-level objective (SLO)** is a measurable target for service behavior, such as the proportion of requests completed successfully within a stated latency over a period. An SLO connects business expectations with engineering decisions. An **error budget** is the amount of unreliability allowed while the SLO is still being met.

Examples might include:

- 99.9% of authorized ticket retrievals succeed over a rolling month, excluding defined client errors.
- 95% of ticket assignments complete within 800 milliseconds under supported load.
- 99% of accepted exports reach a terminal state within 15 minutes.

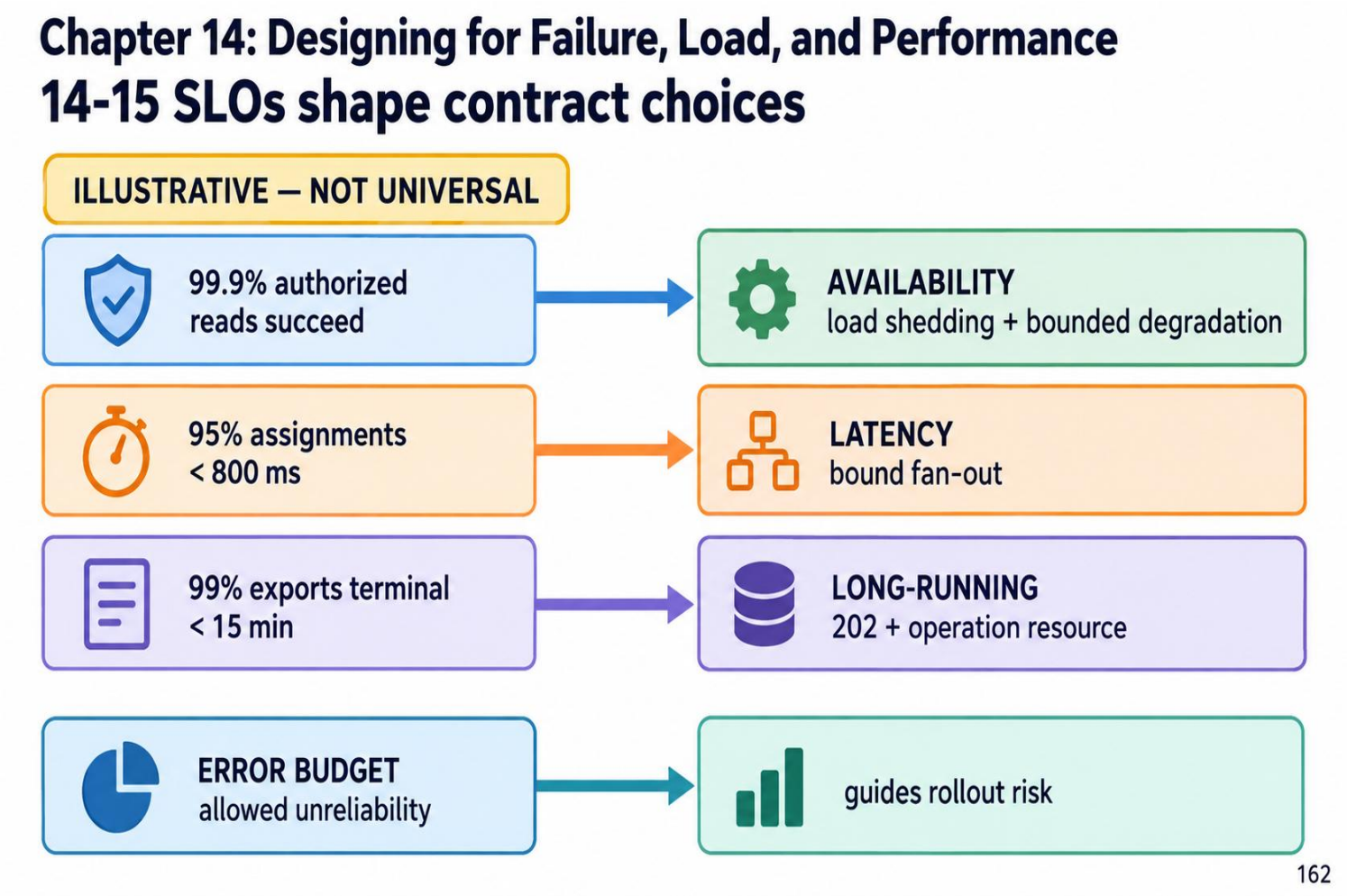



*Figure 14-16: How SLOs shape contract choices*

These are illustrative, not universal targets. The correct values depend on the business workflow and system architecture.

SLOs influence API design:

- A strict interactive latency goal may rule out synchronous calls to slow enrichment systems.
- A long-running report may require `202 Accepted` and an operation resource.
- High availability may require a defined degraded response or load-shedding policy.
- An error budget can guide how much change risk the team accepts.

A contract should not promise performance more precise than the system can measure and support. Consumer-facing expectations and internal SLO definitions should align.

## Example: ticket search during high traffic

Consider `GET /tickets` during a sharp traffic increase. A resilient design can combine:

1. Enforced maximum page size and bounded query complexity.
2. Deterministic pagination rather than unbounded result transfer.
3. Per-tenant rate limits so one consumer cannot exhaust shared capacity.
4. Short, measured timeouts for search dependencies.
5. Load shedding for expensive full-text searches before basic filtered retrievals.
6. Cache use only for responses whose permission and freshness rules make it safe.
7. `429` for caller-specific limit excess and `503` for temporary service unavailability.
8. Metrics that distinguish limited, shed, timed-out, and successful requests.

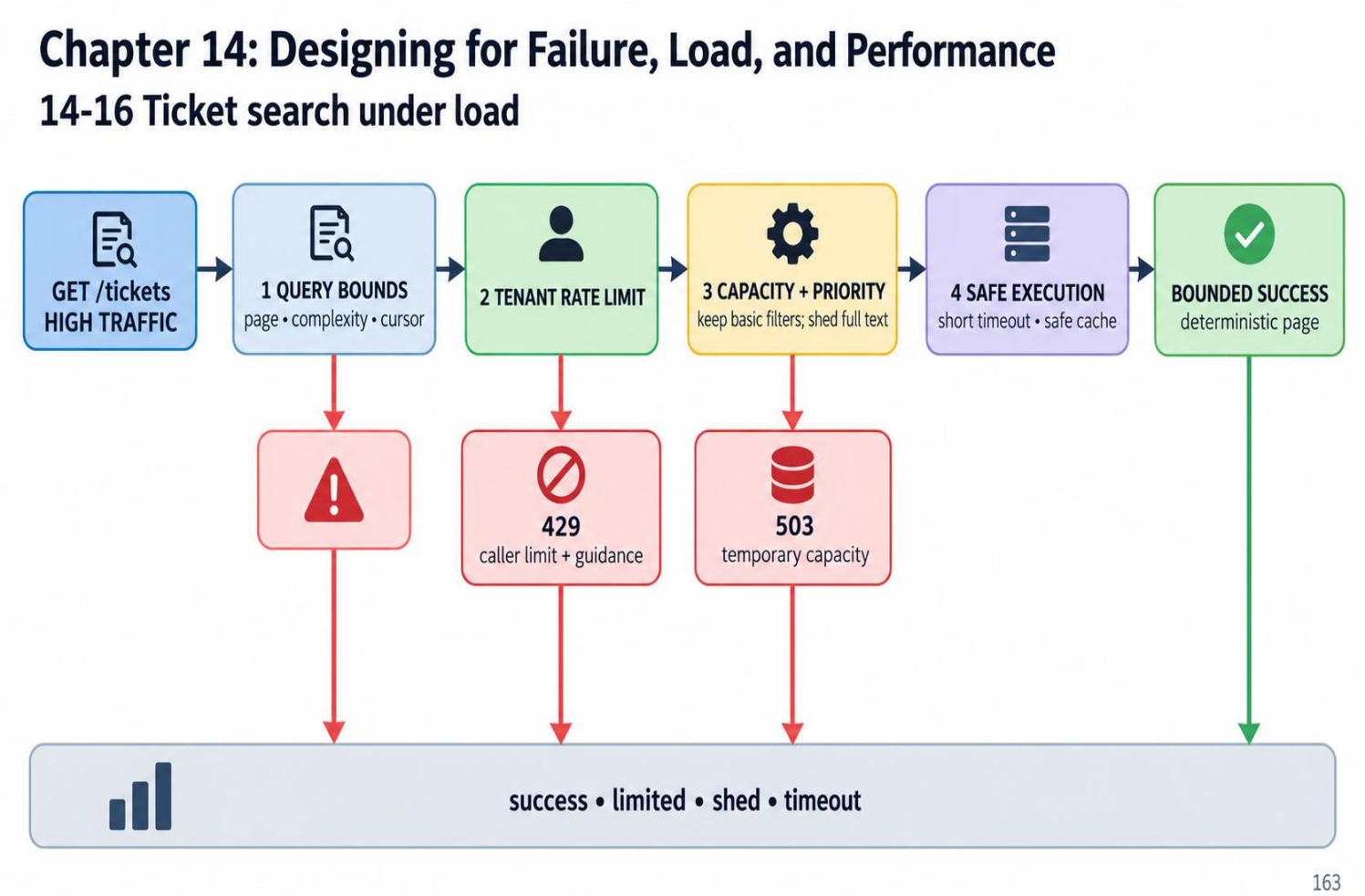



*Figure 14-17: Ticket search during high traffic*

The API should not respond to overload by returning partial pages without saying so or by silently ignoring filters. A bounded, explicit failure preserves contract truth.

## Example: ticket creation after a client timeout

A client sends a protected creation request and times out before receiving the response. Its safe process is:

1. Keep the original idempotency key and request body.
2. Wait according to the retry policy rather than retrying immediately in a tight loop.
3. Resend the same logical request with the same key.
4. Accept the original result if the provider already completed it.
5. Handle an in-progress response or retry guidance if processing is not yet complete.
6. Stop at the overall deadline and surface an uncertain or pending result according to the contract.

The provider must retain key state long enough for the documented retry window and must not create another ticket for the same protected logical operation.

## Two reliability misconceptions

### "Every failed request should be retried immediately"

Permanent errors do not improve with repetition, and immediate retries can amplify overload. Retry only transient conditions, only when repetition is safe, with delay, jitter, and a bounded attempt or time budget.

### "Caching is only an implementation optimization and has no contract implications"

Caching changes freshness, validation, privacy, and what consumers observe after an update. Cache behavior and validators are part of the HTTP contract when clients or intermediaries rely on them.

## Failure and load design check

For each operation, confirm that:

- Client and server waiting are bounded.
- The overall deadline is compatible with downstream timeouts.
- Timeout uncertainty has a safe recovery path for state changes.
- Retriable and non-retriable failures are distinguished.
- Retry attempts use backoff, jitter, and a stopping condition.
- Retry responsibility is not multiplied accidentally across layers.
- Rate limits and quotas have clear scope and error behavior.
- Load shedding protects essential work and returns honest failures.
- Cache freshness, privacy, validators, and invalidation expectations are defined.
- Payload and page sizes are bounded.
- Required and optional downstream data are separated.
- Partial failure cannot produce misleading success.
- SLOs support the chosen synchronous or asynchronous interaction.

**Reliability does not mean that every request succeeds. It means that success, delay, overload, and failure each have bounded and predictable behavior that protects both consumers and the service.**

## Chapter Summary

- Timeouts bound waiting but can leave a client uncertain about whether a state change committed.
- Uncoordinated retries amplify load, while backoff, jitter, idempotency, and retry budgets reduce that risk.
- Caches, limits, fan-out, partial responses, and SLOs affect observable behavior and therefore belong in contract design.

## Review Questions

1. Why is retrying every failure immediately likely to worsen an incident?
2. What additional contract decision is needed when a client times out after sending a create request?
3. How can one incoming request create excessive downstream load?

# Chapter 15 — Asynchronous Operations, Webhooks, and Events

**Work that cannot complete reliably within one request-response exchange needs an explicit asynchronous contract.** That contract must describe acceptance, progress, completion, failure, cancellation, duplicate delivery, and how consumers learn the result.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Choose between synchronous responses, operation resources, webhooks, and published events based on interaction needs.
- Design asynchronous status, retention, cancellation, polling, signatures, replay prevention, acknowledgement, duplication, and ordering.
- Distinguish commands, events, and resource representations and evolve their schemas conservatively.

## Choose synchronous interaction only when waiting is reasonable

A **synchronous interaction** keeps the caller waiting for the operation's result within the same request-response exchange. An **asynchronous interaction** accepts or records work first and reports its eventual outcome separately.

Synchronous interaction fits when:

- Work normally completes within the caller's latency budget.
- The caller needs the result before continuing.
- Failure can be reported completely in the immediate response.
- Holding the connection and processing capacity is acceptable.

Asynchronous interaction fits when:

- Work can take seconds, minutes, or longer.
- Processing depends on queues or several systems.
- Traffic must be smoothed rather than processed immediately.
- The caller can continue and check later.
- Completion needs independent retries or recovery.

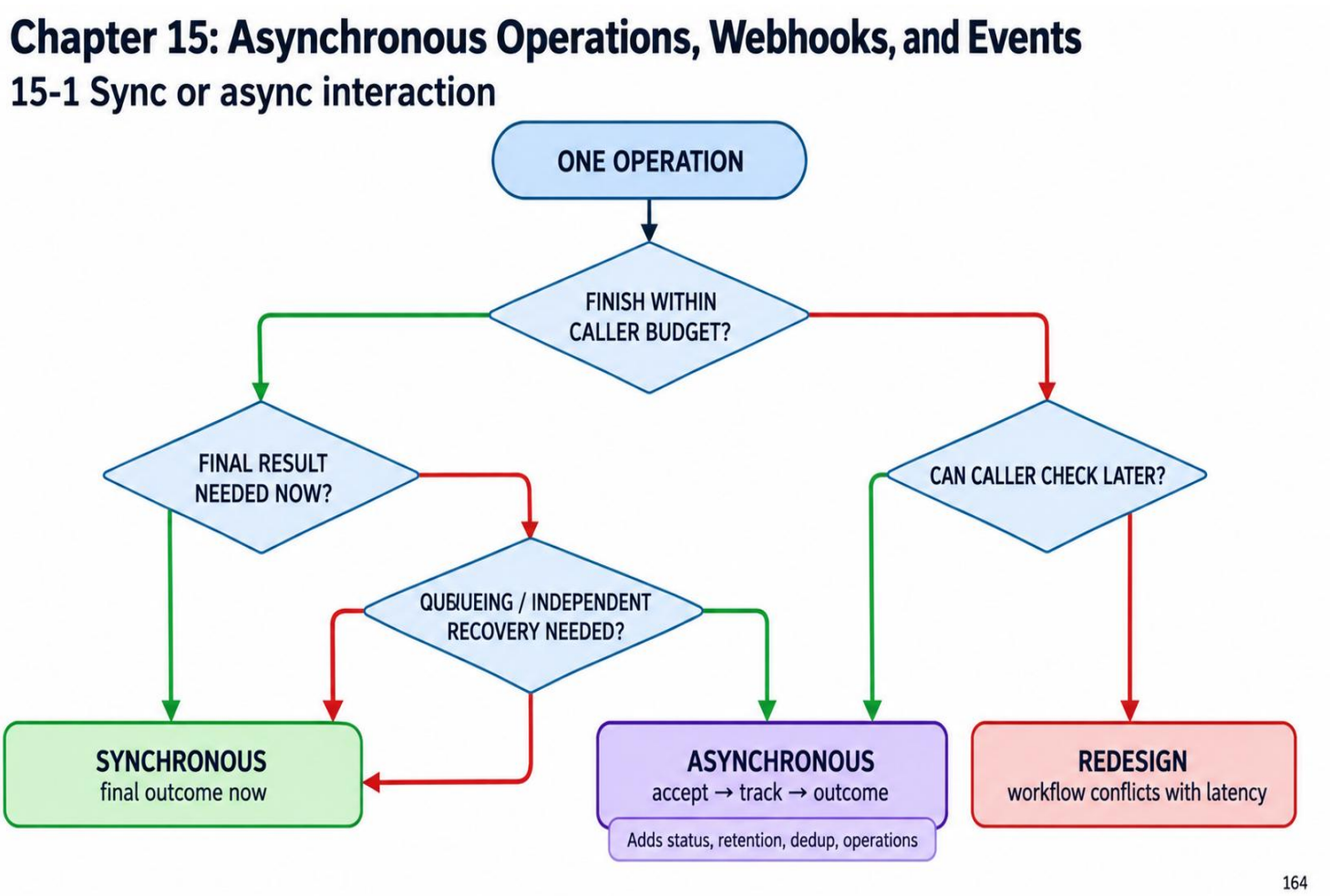



*Figure 15-1: Choosing synchronous or asynchronous interaction*

Do not make a fast operation asynchronous merely because queues are available. Asynchrony introduces state, polling or notification, retention, deduplication, and operational responsibilities. Use it when those costs solve a real problem.

## `202 Accepted` means accepted for processing

`202 Accepted` means that a request has been accepted for processing but has not completed. It is not evidence that the requested business outcome succeeded.

For a large export of resolved tickets, the client can request comma-separated values (CSV), a plain-text format for tabular data:

```
POST /ticket-exports HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access-token>
Idempotency-Key: 5a6d2963-322b-49de-b2f9-d2e8c640c658

{
  "status": "resolved",
  "format": "csv"
}
```

The provider can respond:

```
HTTP/1.1 202 Accepted
Content-Type: application/json
Location: /operations/op_9017
Retry-After: 5

{
  "id": "op_9017",
  "type": "ticket_export",
  "status": "pending",
  "statusUrl": "/operations/op_9017"
}
```

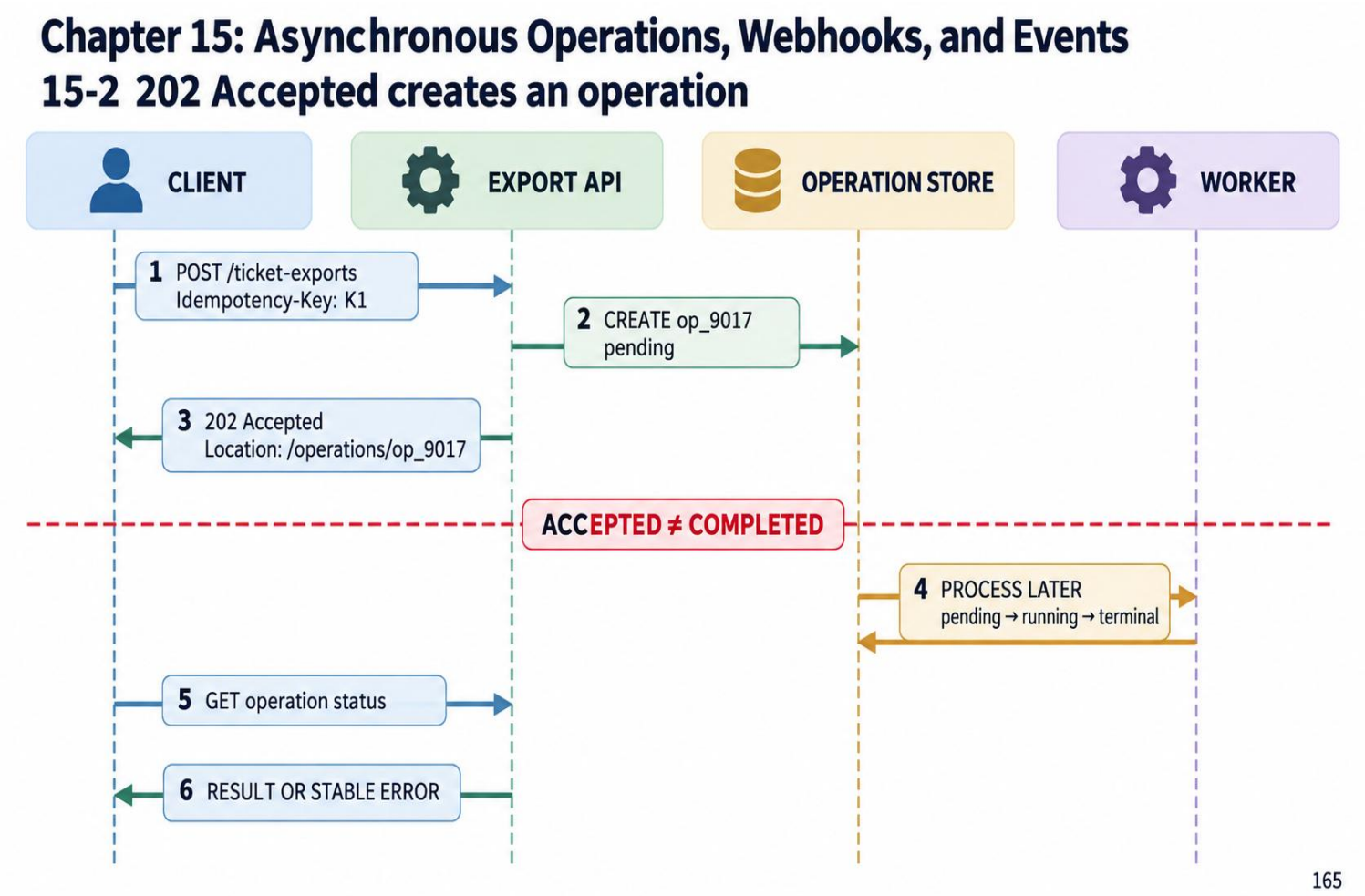



*Figure 15-2: `202 Accepted` and creation of an operation resource*

The important promises are:

- The request was valid enough to accept for processing.
- `op_9017` identifies this work.
- The operation resource reports later state.
- The idempotency key protects the creation of the export operation from duplicate submission.
- `Retry-After` gives initial polling guidance in this example.

The provider should reject known validation or authorization failures immediately rather than accept work that is already certain to fail.

## Model long-running work as an operation resource

An **operation resource** represents the lifecycle of asynchronous work. It gives the client a stable place to retrieve status and eventual results.

A small state model can be:

- `pending`: accepted but not yet running.

- `running`: processing has begun.
- `succeeded`: completed successfully.
- `failed`: reached a terminal failure.
- `cancelled`: stopped by an accepted cancellation request.

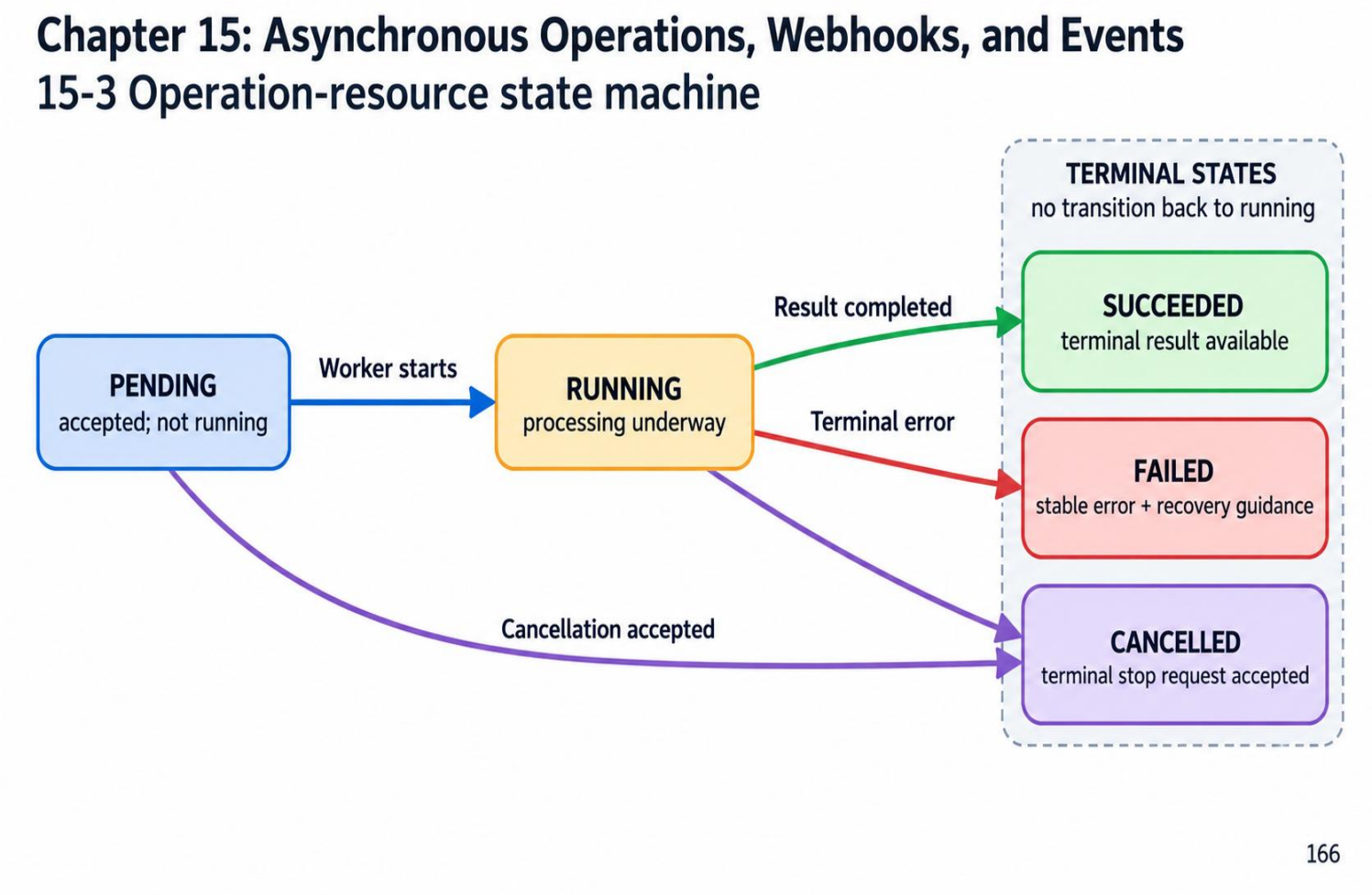



*Figure 15-3: Operation-resource state machine*

The operation representation should include only information that helps the consumer manage the work. For example:

```
{
  "id": "op_9017",
  "type": "ticket_export",
  "status": "running",
  "createdAt": "2026-09-18T10:00:00Z",
  "updatedAt": "2026-09-18T10:00:08Z",
  "progress": {
    "completedItems": 420,
    "estimatedTotalItems": 1000
  }
}
```

Progress can be approximate. If total work cannot be known accurately, do not present a misleading percentage. A simple state and last-updated time may be more honest.

When successful, the resource can point to the result:

```
{
  "id": "op_9017",
  "type": "ticket_export",
  "status": "succeeded",
  "completedAt": "2026-09-18T10:00:31Z",
  "result": {
    "downloadUrl": "/ticket-exports/exp_771/content",
    "expiresAt": "2026-09-19T10:00:31Z"
  }
}
```

When failed, it should contain a stable error structure appropriate to the operation:

```
{
  "id": "op_9017",
  "type": "ticket_export",
  "status": "failed",
  "completedAt": "2026-09-18T10:00:19Z",
  "error": {
    "code": "EXPORT_SOURCE_UNAVAILABLE",
    "title": "The export failed",
    "detail": "The export could not be completed. Submit a new export later."
  }
}
```

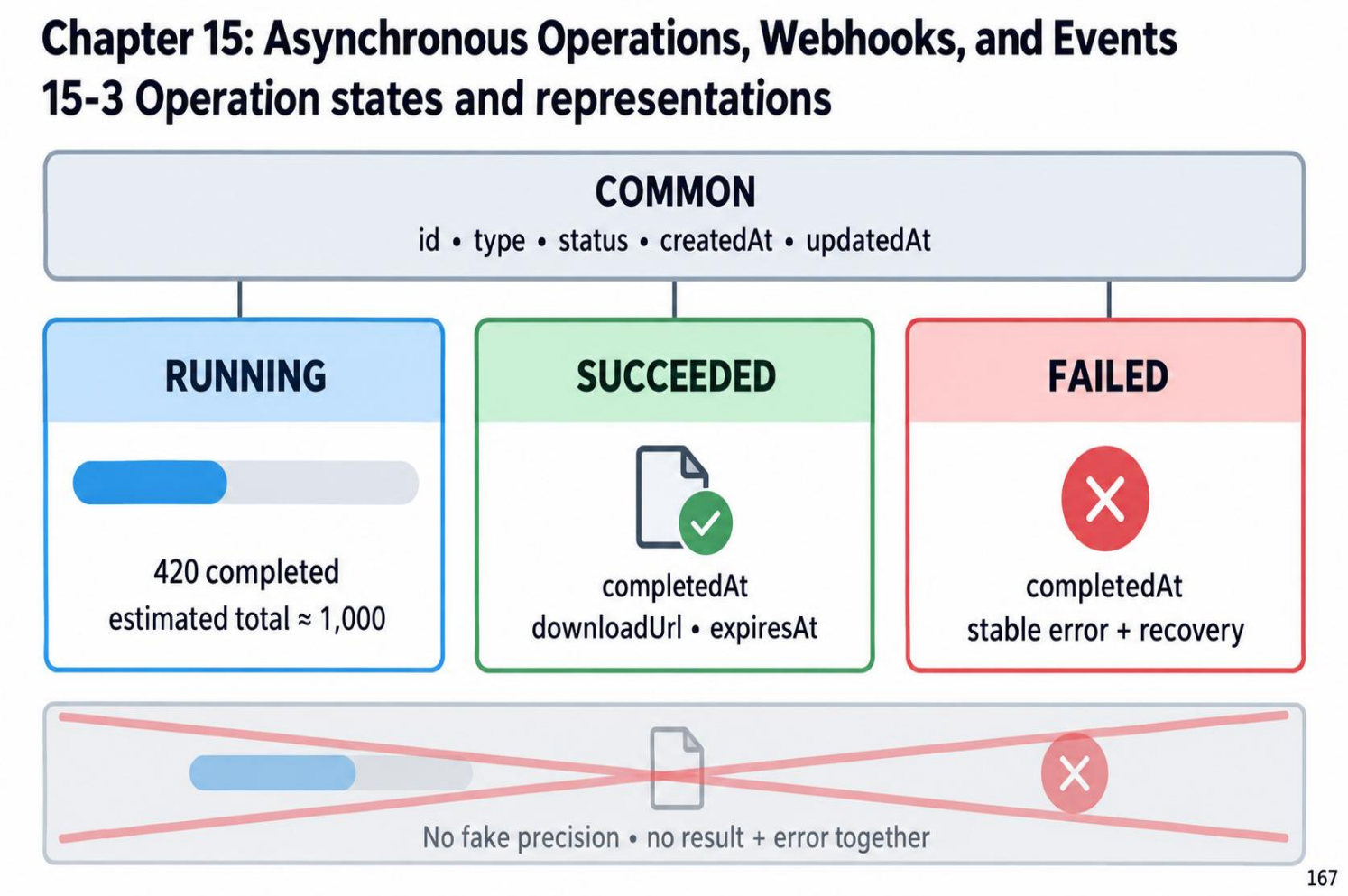



*Figure 15-4: Operation representation at running, success, and failure*

Do not expose an internal stack trace or raw queue exception. The client needs the result, recovery guidance, and a correlation identifier where support investigation is necessary.

## Define terminal behavior and retention

An operation contract should answer:

- Which states are terminal?
- Can a failed operation be retried in place, or must the client create a new operation?
- How long are operation resources retained?
- How long are downloadable results retained?
- What response follows expiration?
- Are results immutable after success?
- Who may read or cancel the operation?

Operation and result retention may differ. The provider might keep the operation metadata for 30 days but keep a generated export file for only 24 hours. The client needs both dates.

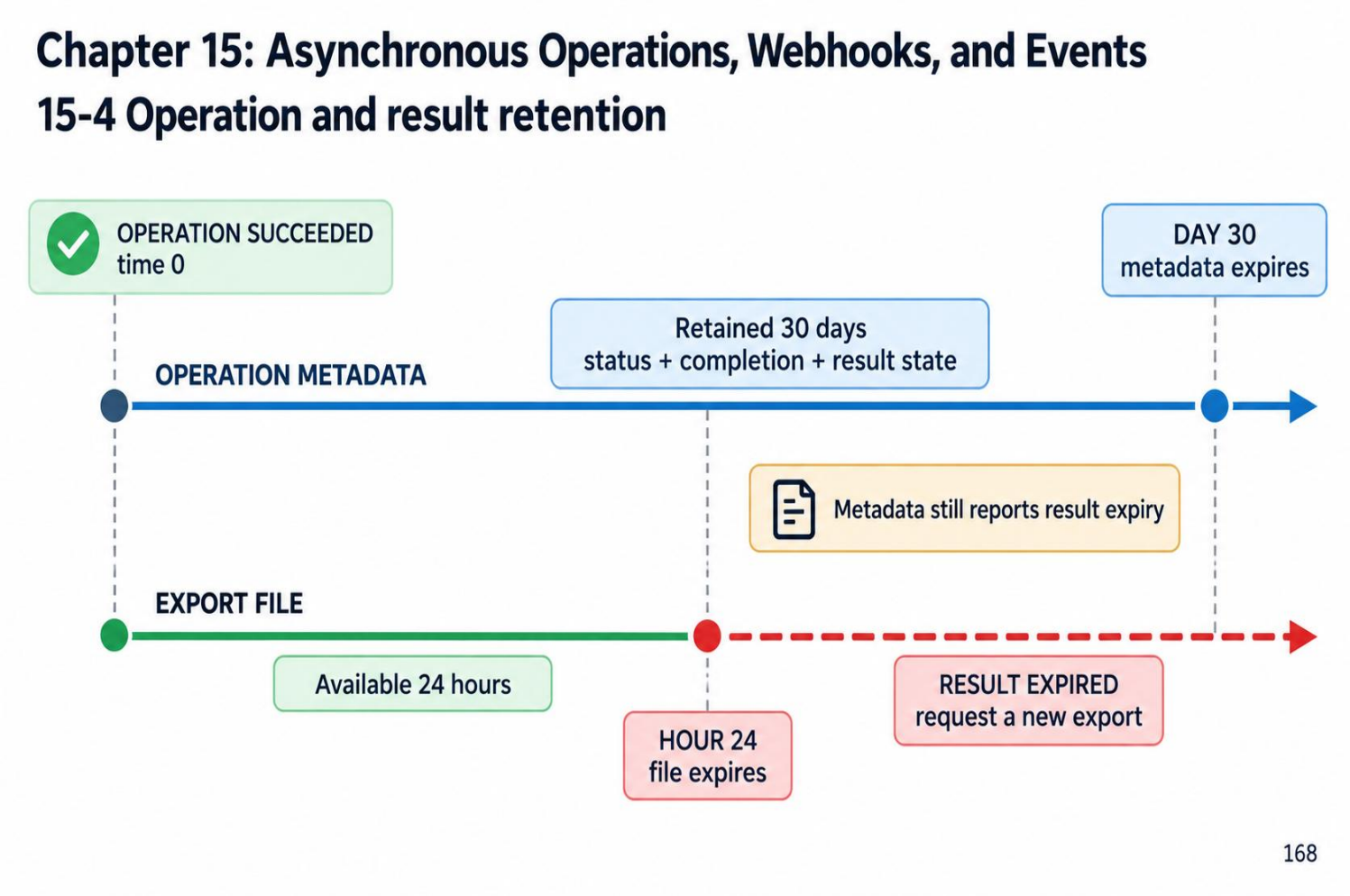



*Figure 15-5: Operation metadata retention versus result retention*

Expiration should not silently change a successful operation back to “not found” without documentation. A retained operation can state that its result expired and direct the client to create a new export.

## Cancellation is a request, not a guaranteed reversal

Cancellation can be offered when work has meaningful intermediate stages. The contract should say whether cancellation is best-effort and which states allow it.

For this example, the API might use an explicit command:

```
POST /operations/op_9017/cancel HTTP/1.1
Authorization: Bearer <access-token>
```

Possible outcomes include:

- Cancellation accepted and operation becomes `cancelled`.
- Operation already completed, so cancellation conflicts with current state.
- Operation is at a non-interruptible stage and remains `running`.

- Caller lacks permission to cancel this operation.

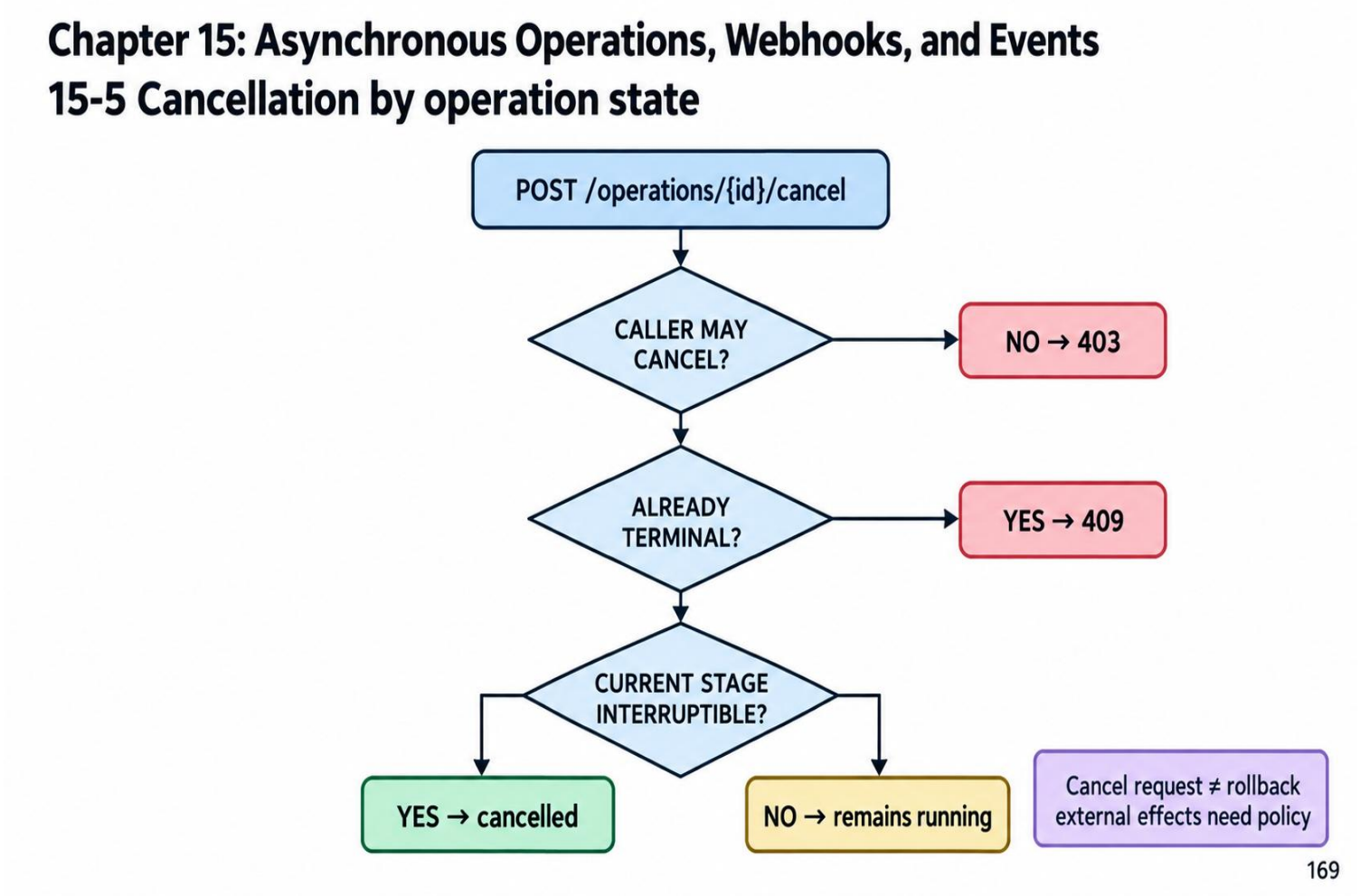



*Figure 15-6: Cancellation outcomes by current state*

Cancellation does not necessarily undo external effects already completed. If an operation sent notifications or wrote partial outputs, define whether those effects remain, are compensated, or are never visible until final success.

## Poll without creating avoidable load

**Polling** means that a client periodically retrieves status to learn whether an asynchronous operation has changed. Polling is simple and reliable when intervals are controlled.

A safe polling process is:

1. Use the status URL returned by the provider.
2. Wait at least the recommended interval.
3. Retrieve the operation resource.
4. Stop when it reaches a terminal state.
5. Increase delay or follow updated guidance for long operations.
6. Stop at the client's overall deadline or user-defined limit.

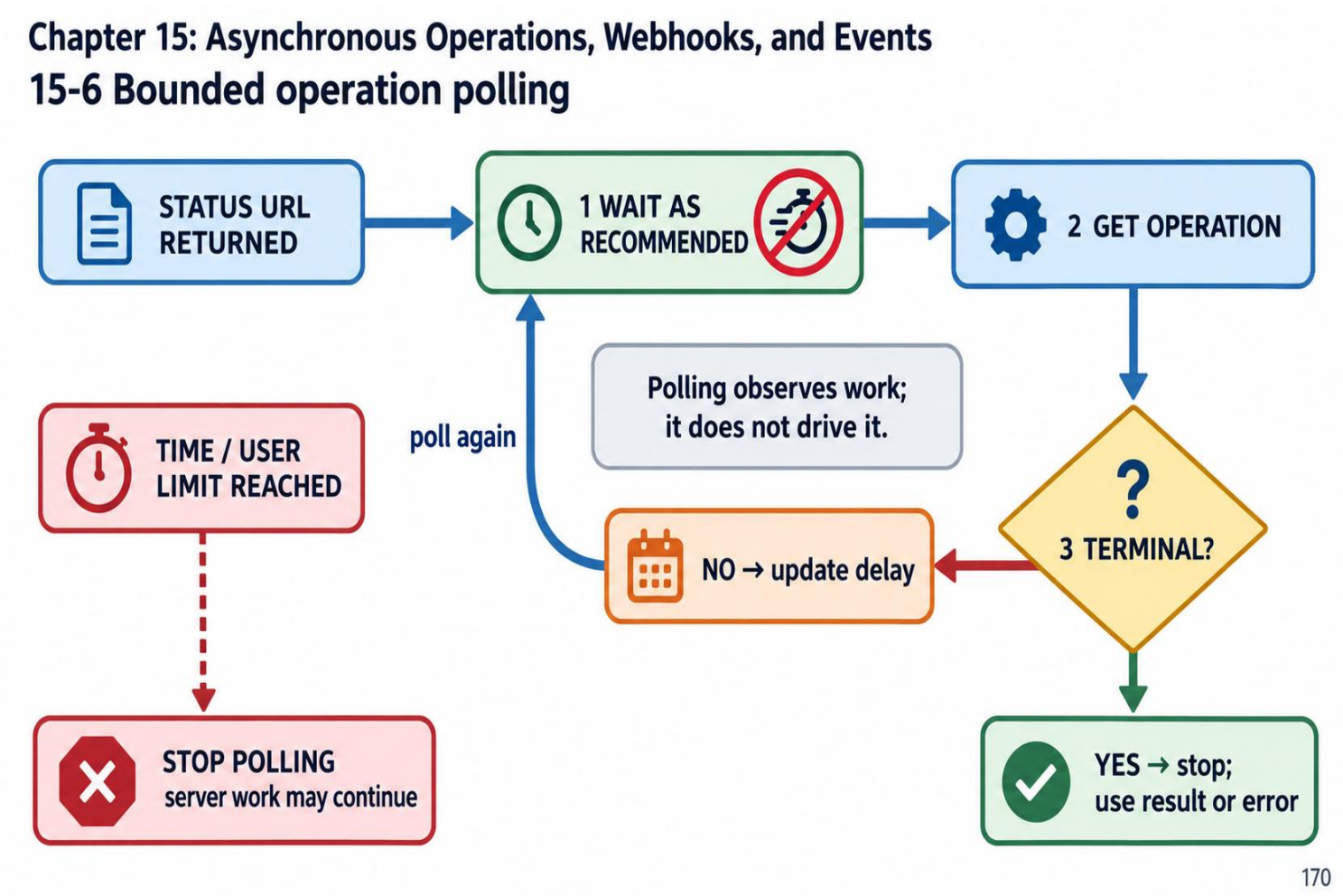



*Figure 15-7: Bounded polling loop*

Rapid polling by many clients can become more expensive than the original work. The provider can use `Retry-After`, documented minimum intervals, conditional retrieval, or push notification to reduce unnecessary traffic.

The client should not assume that progress changes on every poll. The server should not require a poll to continue processing.

## A webhook lets the provider call the consumer

A **webhook** is an HTTP notification sent by a provider to an endpoint registered by a consumer when a relevant event occurs. It reverses the usual direction: the ticket service becomes the HTTP client for the delivery, while the consumer operates the receiving endpoint.

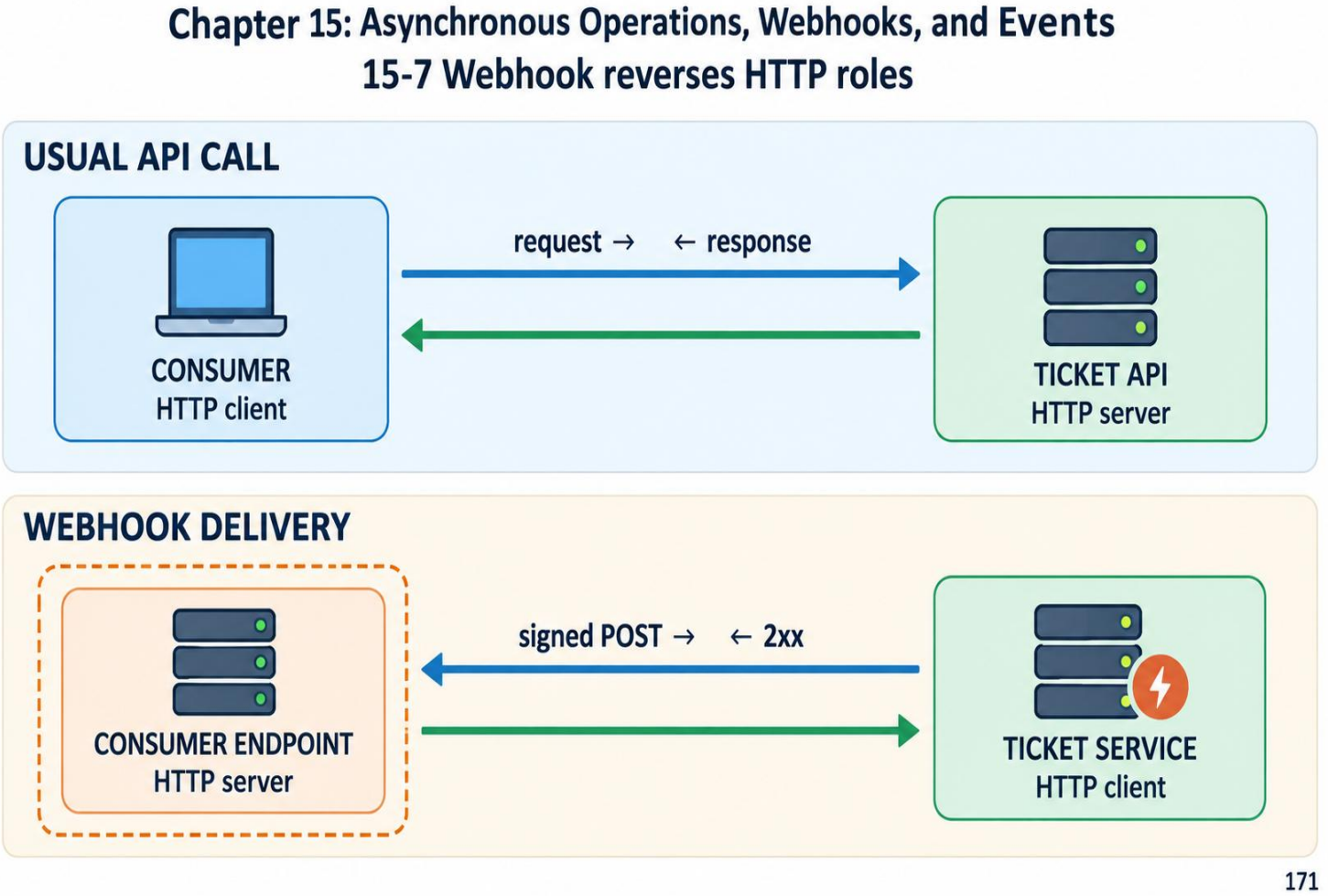





*Figure 15-8: Webhook reverses the HTTP client/server direction*

A webhook is useful when a consumer needs timely notification but does not need a continuously open connection. For example, an external workforce system may need to know when a ticket is assigned.

A webhook contract includes more than the JSON body:

- How endpoints are registered and owned.
- Which event types can be selected.
- How deliveries are authenticated.
- How the receiver detects replay and duplicates.
- Which response counts as successful receipt.
- How often failed deliveries are retried.
- How long the provider retries or retains undelivered events.
- How subscriptions are disabled after persistent failure.
- How event schemas evolve.

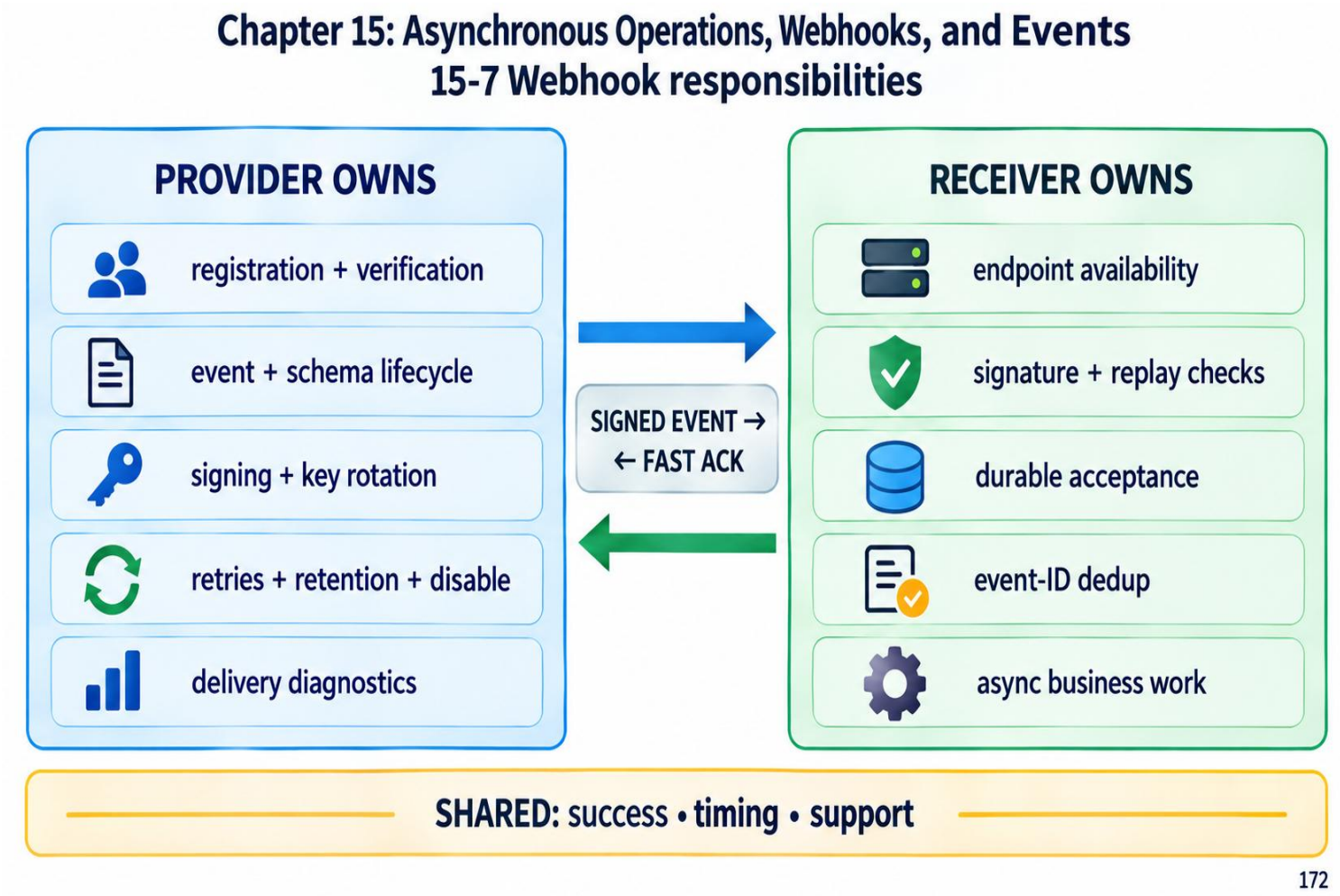



*Figure 15-9: Webhook contract responsibilities on both sides*

The receiver is now part of a distributed reliability path. Both sides need operational ownership.

## Verify endpoint ownership before sending sensitive events

A provider should avoid sending ticket data to an arbitrary URL supplied by an unverified caller. Registration should require authorization and endpoint ownership verification appropriate to the environment.

One common pattern is a challenge:

1. The consumer registers an HTTPS endpoint.
2. The provider sends a one-time challenge value.
3. The endpoint returns the required response.
4. The provider activates the subscription only after successful verification.

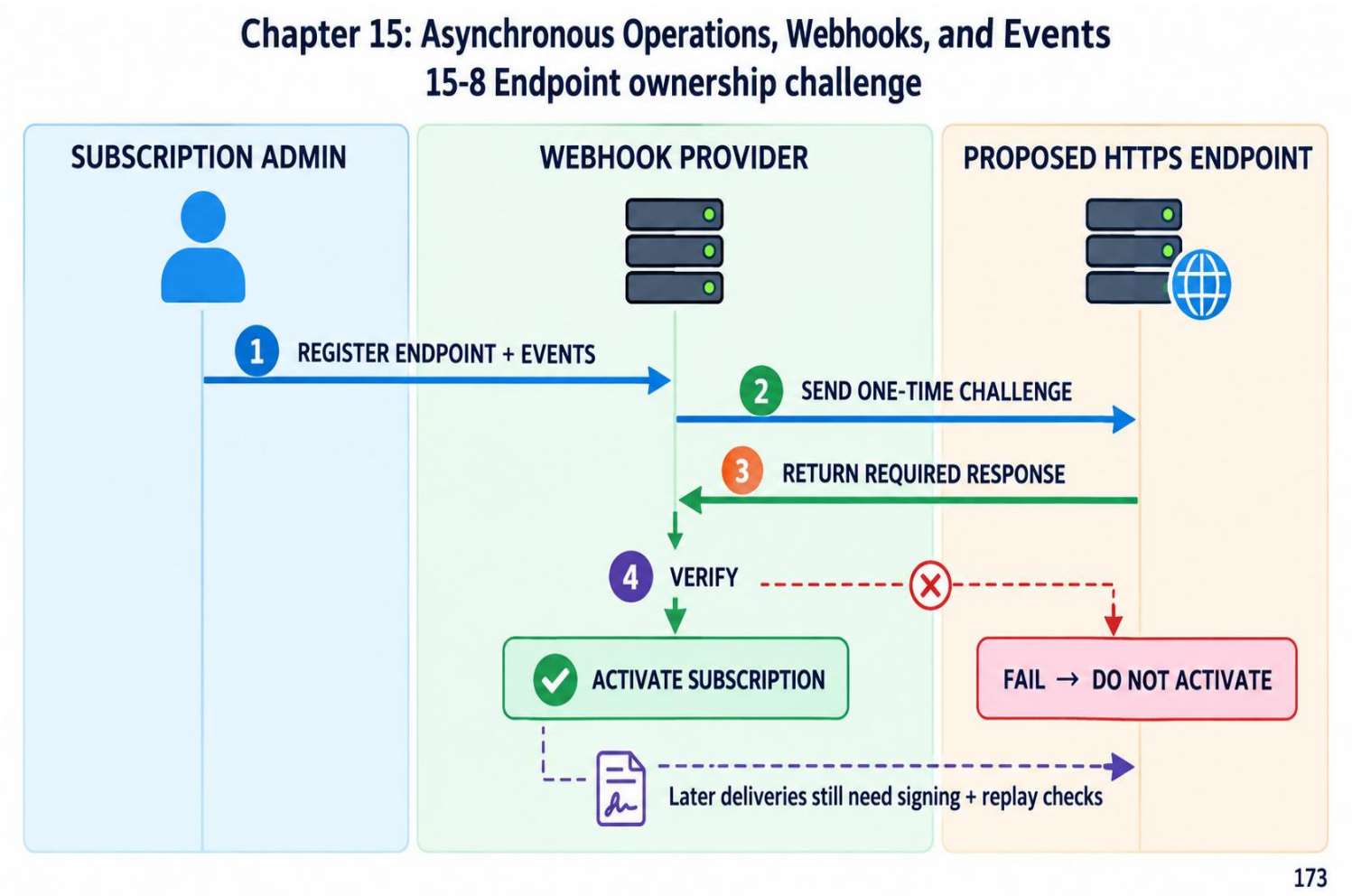



*Figure 15-10: Endpoint ownership verification challenge*

Verification proves control of the endpoint at registration time. It does not replace ongoing authentication of each delivery or authorization to subscribe to particular event data.

Providers should also restrict destination addresses to reduce **server-side request forgery (SSRF)**, an attack in which a service is induced to send requests to unintended internal or external destinations. The exact network policy is environment-specific and should be reviewed with security specialists.

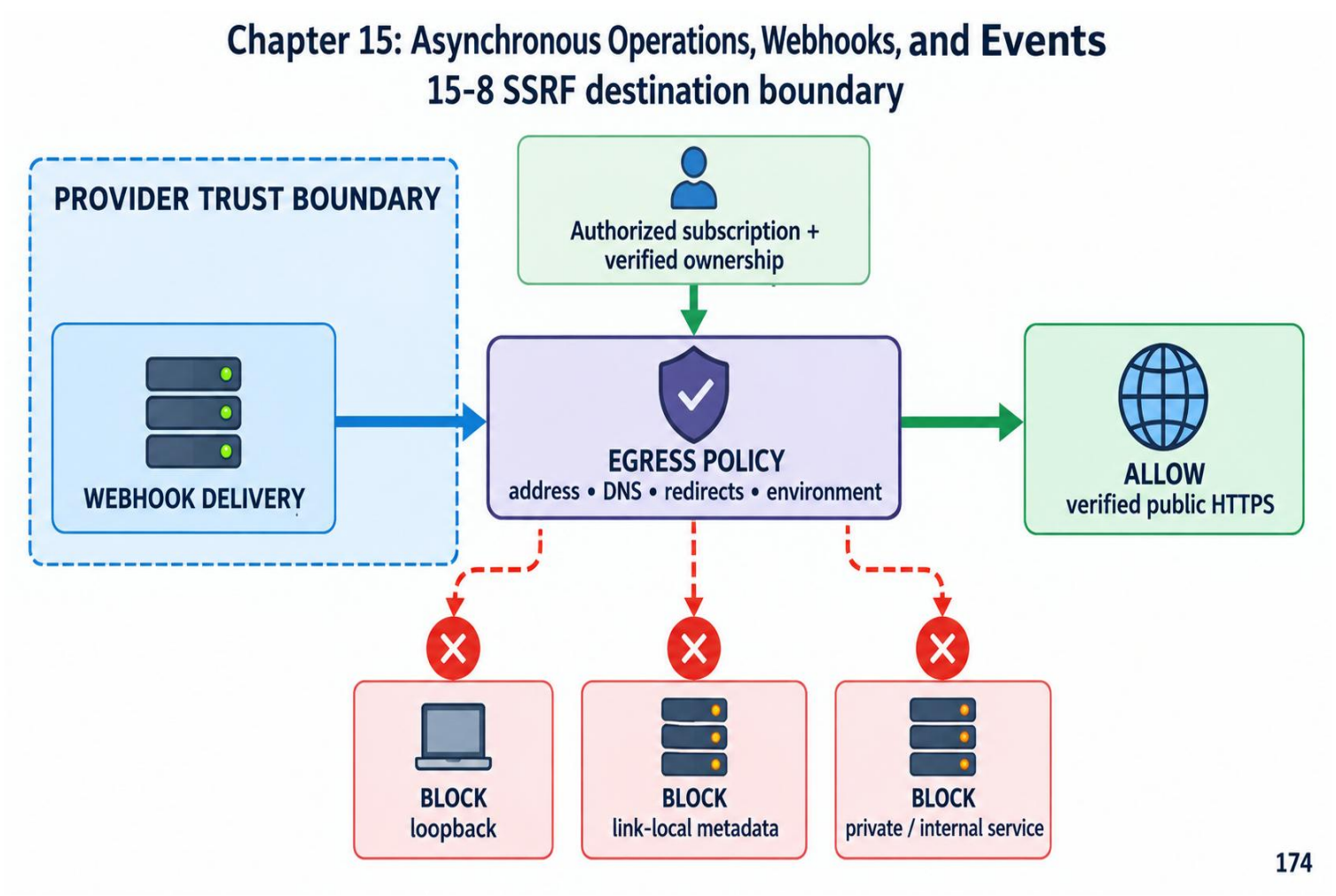



*Figure 15-11: SSRF destination boundary*

## Sign deliveries and prevent replay

A **webhook signature** is a cryptographic value that lets the receiver verify that the delivery was produced by a holder of the agreed signing secret and that the signed content was not modified. Use an established signing construction and library rather than inventing one.

A typical delivery includes:

```
POST /webhooks/tickets HTTP/1.1
Content-Type: application/json
X-Webhook-Id: evt_68321
X-Webhook-Timestamp: 1790244312
X-Webhook-Signature: v1=<signature>
```

The receiver generally:

1. Reads the raw request body exactly as received.
2. Checks that the timestamp is within an allowed replay window.
3. Calculates and securely compares the expected signature.
4. Rejects an invalid signature without processing the event.
5. Checks whether the event identifier was already accepted.
6. Durably records the event before returning success.

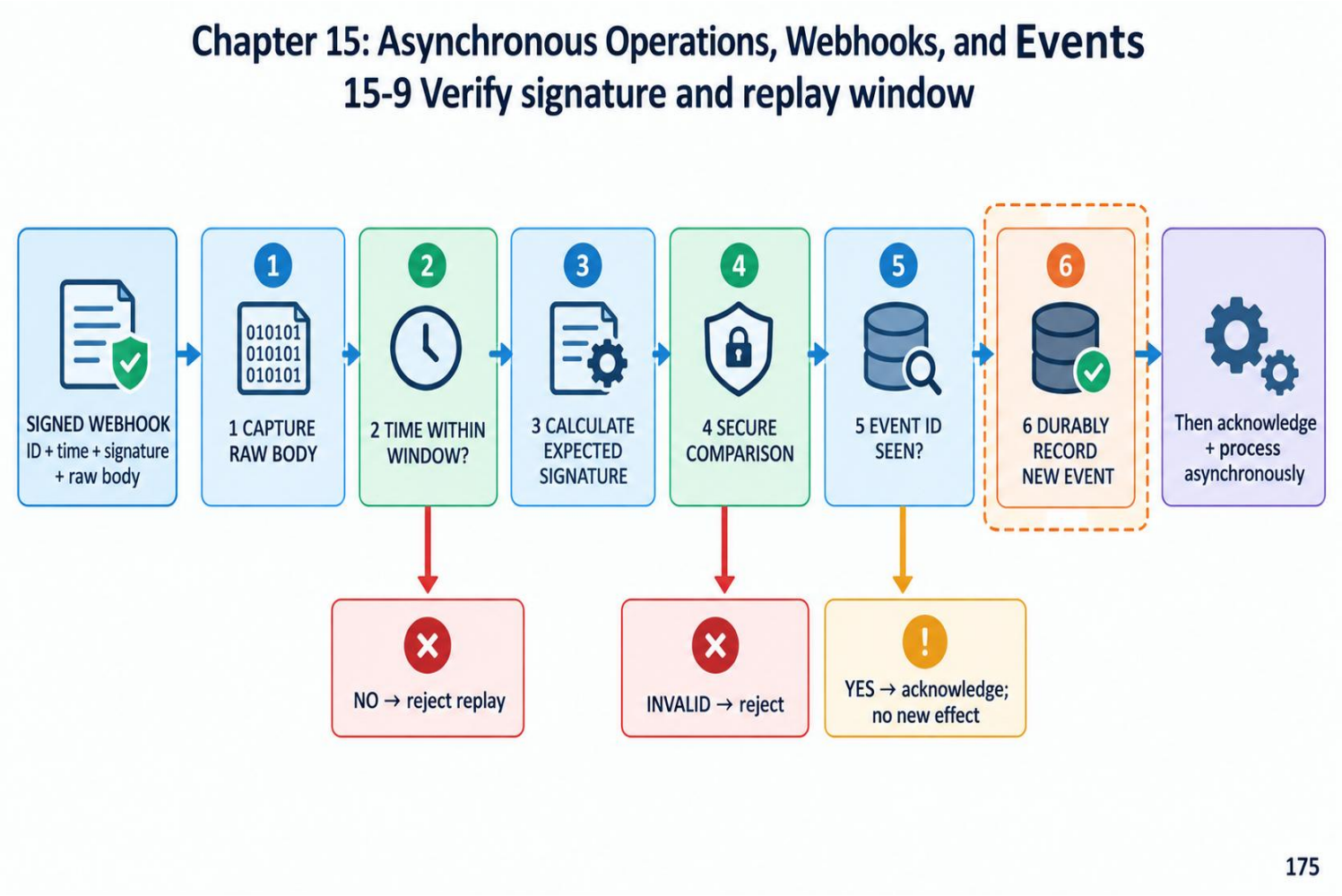



*Figure 15-12: Webhook signature and replay-window verification*

**Replay protection** prevents an old valid signed request from being accepted again outside the intended delivery process. Timestamp checks and stored delivery identifiers support this protection.

The signed material, header names, timestamp units, key rotation, and comparison algorithm must be documented precisely. A signature scheme that each consumer guesses differently is not a usable contract.

## Acknowledge only after durable acceptance

A receiver should return a successful status only after it has durably accepted responsibility for the event. That does not require completing all business processing during the webhook request.

A reliable receiver can:

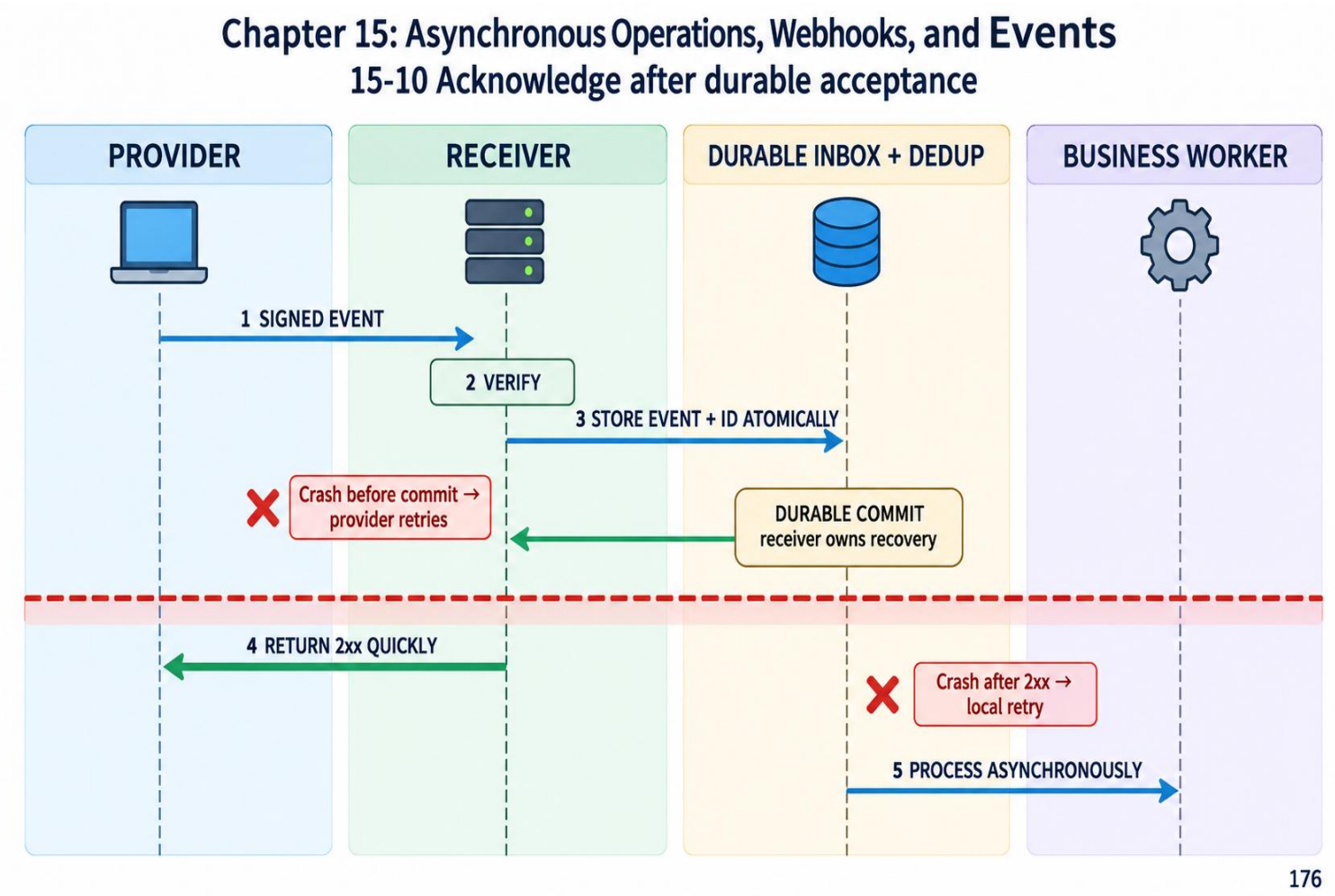



*Figure 15-13: Acknowledge only after durable acceptance*

1. Authenticate and validate the delivery.
2. Store the event or enqueue it durably with its identifier.
3. Return a success status quickly.
4. Process the business effect asynchronously.
5. Retry its own processing when necessary.

Returning success before durable storage risks losing the event if the receiver crashes. Performing slow business work before responding increases provider timeouts and duplicate delivery.

The provider should document which status range counts as success and how timeouts or non-success responses trigger retries.

## Duplicate webhook delivery is normal

A provider may send the same event more than once because the first response was lost, delivery timed out, or retry state was uncertain. Receivers must make processing idempotent or deduplicate by a stable event identifier.

For `ticket.assigned`, the delivery might be:

```
{
  "id": "evt_68321",
  "type": "ticket.assigned",
  "occurredAt": "2026-09-24T08:45:12Z",
  "subject": {
    "type": "ticket",
    "id": "tkt_48271"
  },
  "data": {
    "assigneeId": "usr_314",
    "queueId": "queue_account_access"
  }
}
```

The receiver can process it safely:

1. Begin a local transaction.
2. Check a table or store for `evt_68321`.
3. If already processed, commit no new business effect and return success.
4. If new, apply the local effect and record `evt_68321` atomically where possible.
5. Commit and return success.

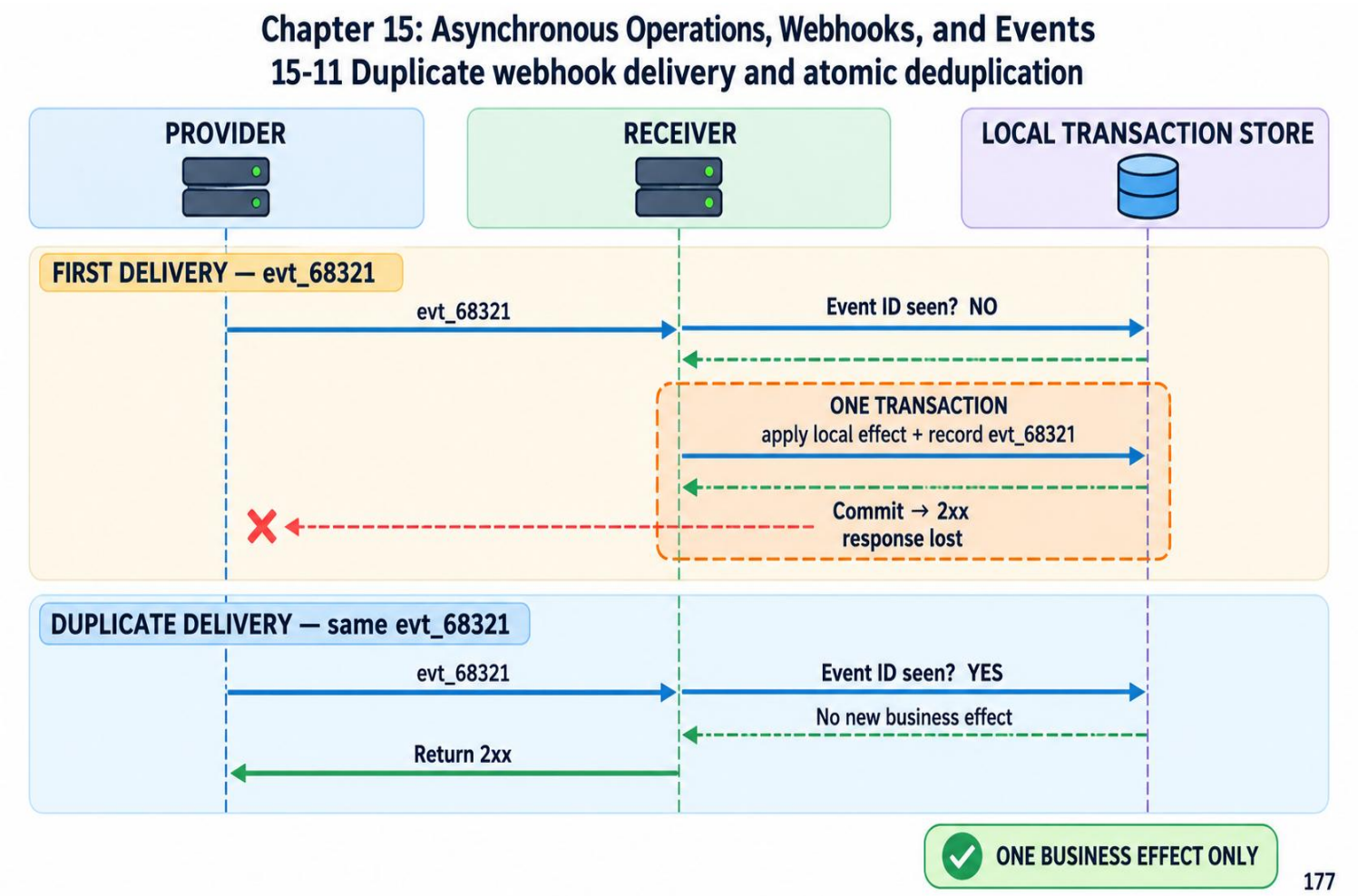



*Figure 15-14: Duplicate webhook delivery and atomic deduplication*

This reuses the earlier idempotency principle without assuming that network delivery is exactly once.

## Events use publish-subscribe communication

An **event** is a record that something of business significance happened, such as “ticket assigned.” In a **publish-subscribe model**, a producer publishes events without directly invoking each subscriber’s business operation, and interested consumers subscribe through an event broker or delivery mechanism.

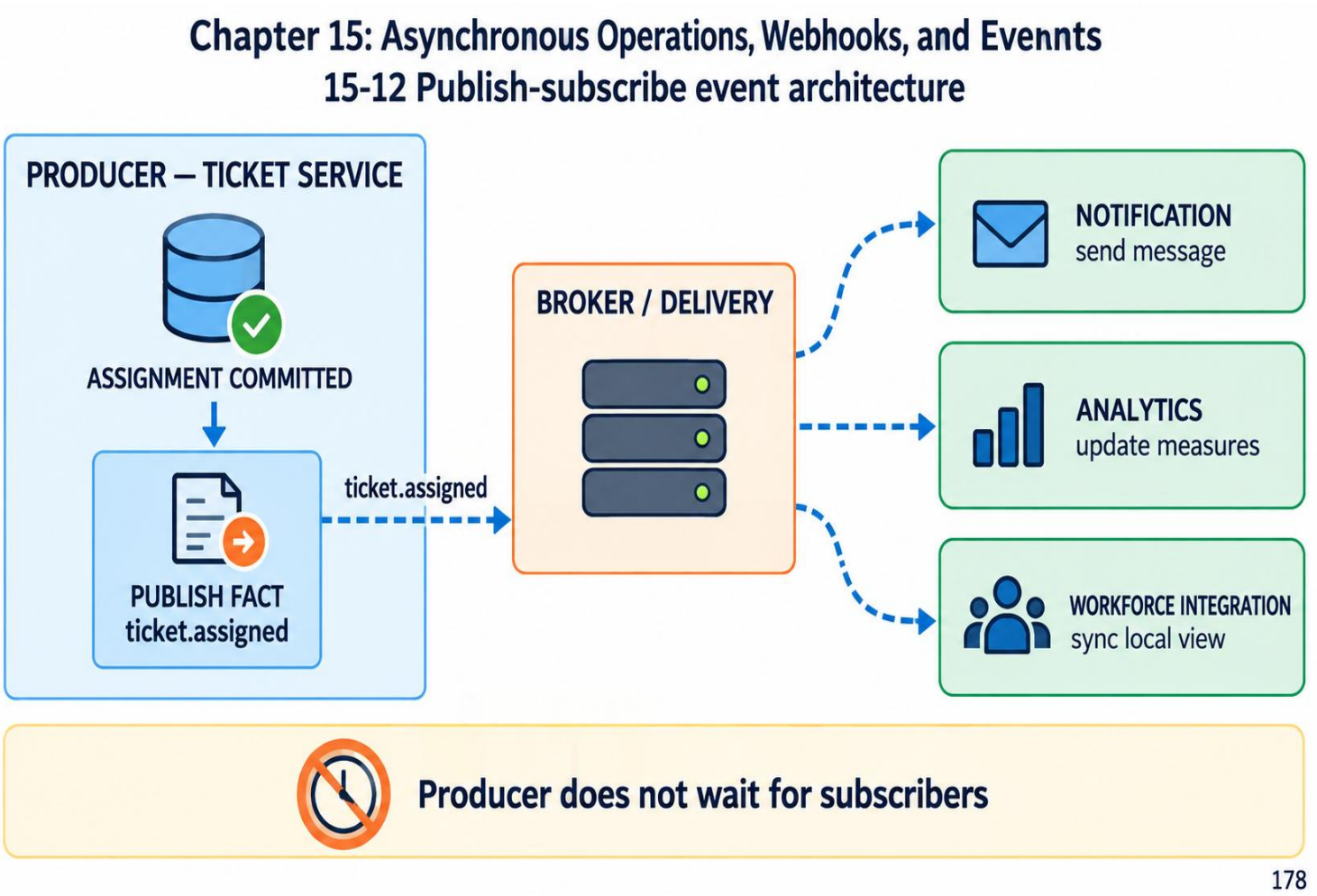



*Figure 15-15: Publish-subscribe event architecture*

Events support loose timing between systems: the ticket service can commit its state and publish notification work without waiting for every consumer to finish. They also introduce questions about delivery, order, duplication, retention, and schema ownership.

A webhook often transports an event over HTTP to one registered endpoint. An event broker can deliver events through queues or streams to several consumers. The message meaning should remain clear regardless of transport.

## Delivery guarantees describe outcomes, not magic settings

Three terms are commonly used:

- **At-most-once delivery:** A message is delivered zero or one time; loss is possible, but the delivery layer does not retry duplicates.
- **At-least-once delivery:** The system retries to reduce loss, so a message may be delivered more than once.
- **Effectively-once processing:** Duplicate delivery may occur, but deduplication and idempotent processing make the intended business effect occur once from the consumer's perspective.

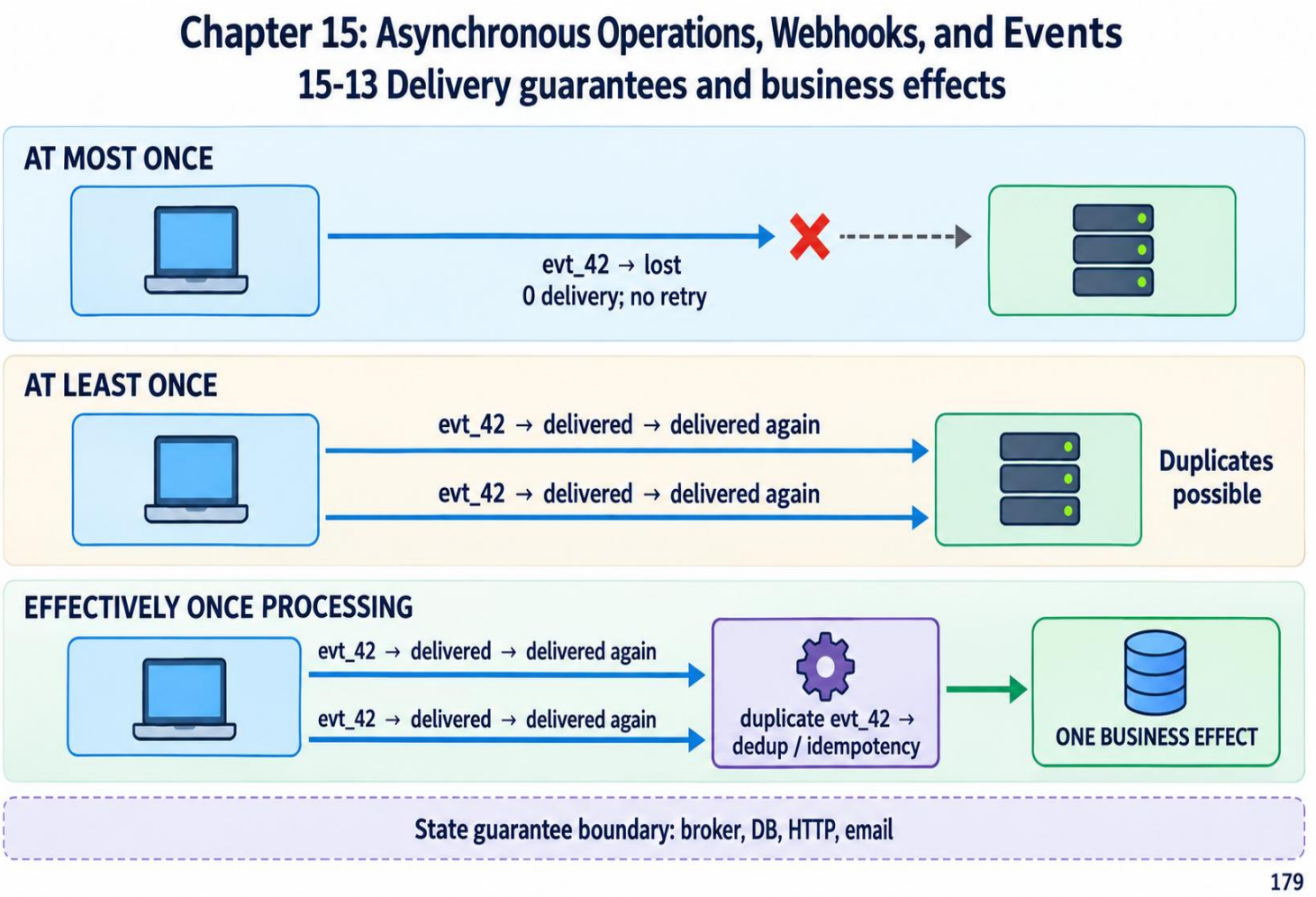



*Figure 15-16: At-most-once, at-least-once, and effectively-once outcomes*

“Exactly once” is often used imprecisely. A broker may provide strong guarantees within a defined boundary, yet an external database update, email, or HTTP call can still be repeated after a crash. State the boundary and observable effect rather than relying on the label.

For most business integrations, at-least-once delivery plus idempotent or deduplicated processing is a practical model.

## Ordering is limited in scope

Distributed events are not automatically observed in the same order by every consumer. Retries, partitions, parallel processing, and different routes can reorder delivery.

If order matters, define its scope. For example:

- Events for one ticket carry a monotonically increasing `ticketVersion`.
- The provider preserves order only within one ticket partition.
- Consumers ignore an event whose version is older than the latest version they processed.
- No global order is promised across different tickets.

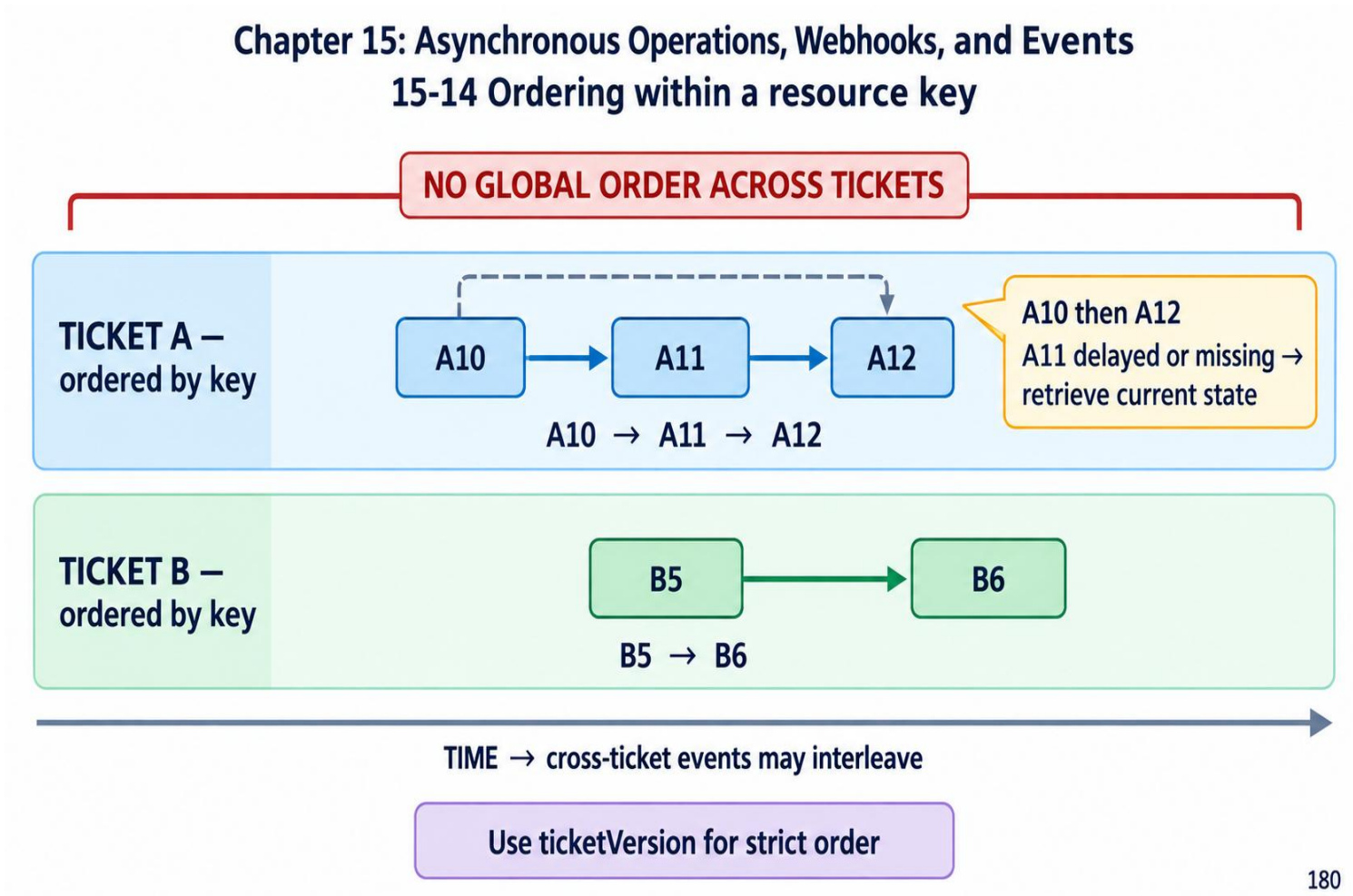



*Figure 15-17: Ordering limited to one resource or partition*

A timestamp alone is usually insufficient for strict ordering because clocks differ and several changes can share a timestamp. A per-resource version or sequence can provide a clearer ordering signal.

Consumers should also handle gaps. Receiving version 12 after version 10 may mean version 11 is delayed or missing. The contract can direct the consumer to retrieve the current ticket when a complete sequence is required.

## Correlate messages across a workflow

A **correlation identifier** links messages that belong to the same broader workflow. A request identifier usually identifies one API request, while a correlation identifier can follow work across an initial request, asynchronous operation, event, and downstream call.

For example:

- A client requests export with request ID `req_100`.
- The provider creates operation `op_9017` with correlation ID `corr_550`.
- Completion publishes event `evt_700`, also carrying `corr_550`.
- A notification service logs its delivery under the same correlation ID.

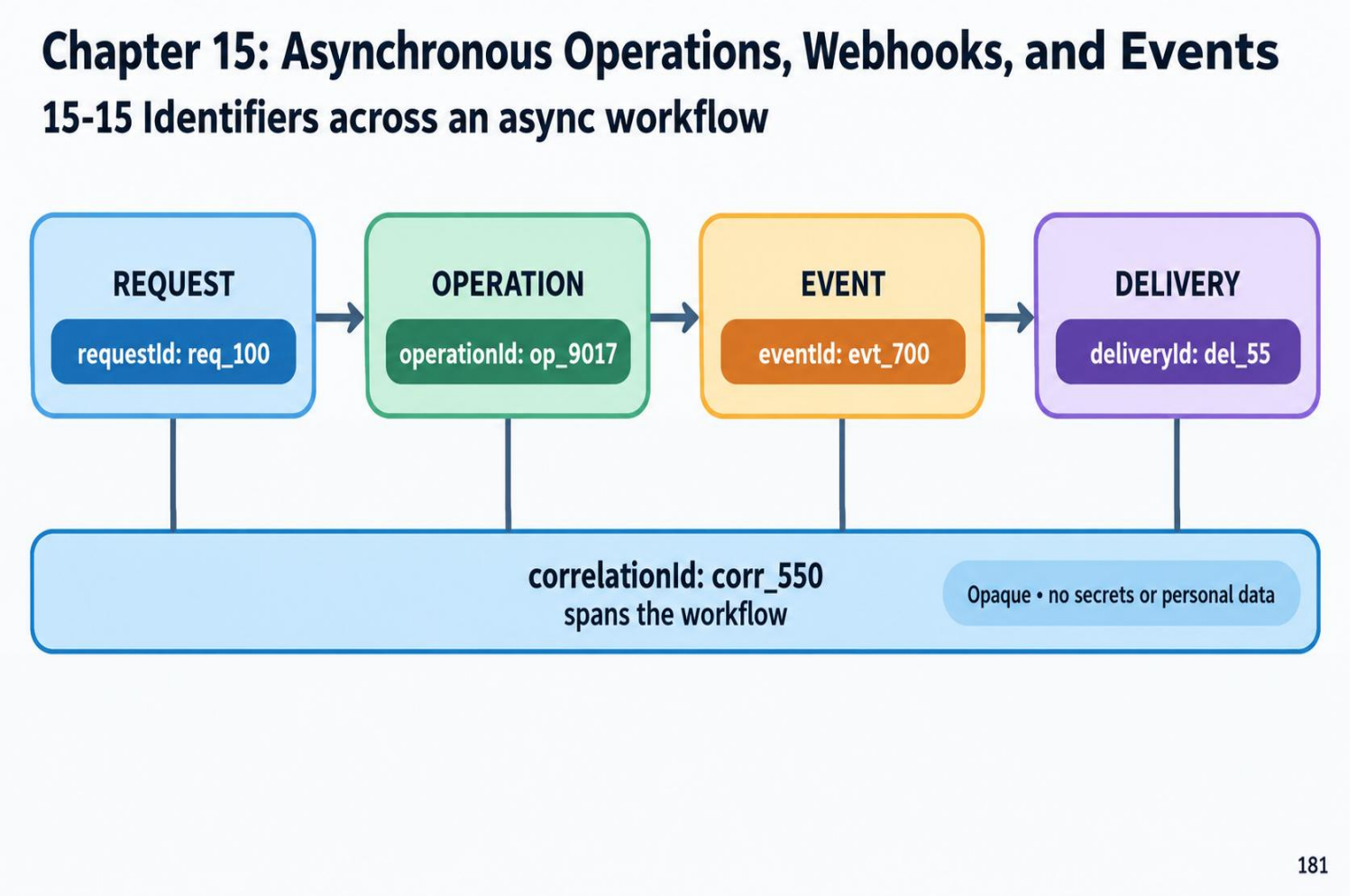



*Figure 15-18: Request, operation, event, and correlation identifiers*

Identifiers support diagnosis but should not carry secrets or personal data. They should be treated as opaque values.

## Commands, events, and resource representations have different purposes

| Message type | Meaning | Typical naming | Expected receiver behavior |
|---|---|---|---|
| Command | A request to perform an action | AssignTicket | Validate and decide whether to execute |
| Event | A statement that something happened | ticket.assigned | React without changing the historical fact |
| Resource representation | A view of current state | Ticket JSON object | Read or synchronize current known state |

Confusing these purposes creates weak contracts. An event named `assignTicket` sounds like a command and leaves unclear whether assignment already happened. A resource snapshot sent as an event may be useful, but consumers need to know whether omitted fields are unchanged, unavailable, or intentionally absent.

Events should be named in the past tense or otherwise clearly indicate completed facts. Commands should not imply that acceptance guarantees success when execution can still fail.

## Evolve event schemas conservatively

Event consumers can remain disconnected for long periods, replay old messages, or update on different schedules. Event schema evolution therefore requires the same compatibility discipline as request-response APIs, often with longer coexistence.

Practical rules include:

- Give each event a stable type and identifier.
- Include a schema or event version when incompatible forms may coexist.
- Add optional fields rather than changing established meanings.

- Do not remove fields until all supported replay and consumer windows have passed.
- Treat new enumeration values as a compatibility risk.
- Keep envelope metadata separate from event-specific data.
- Test old event samples against new consumers and new event samples against supported consumers.

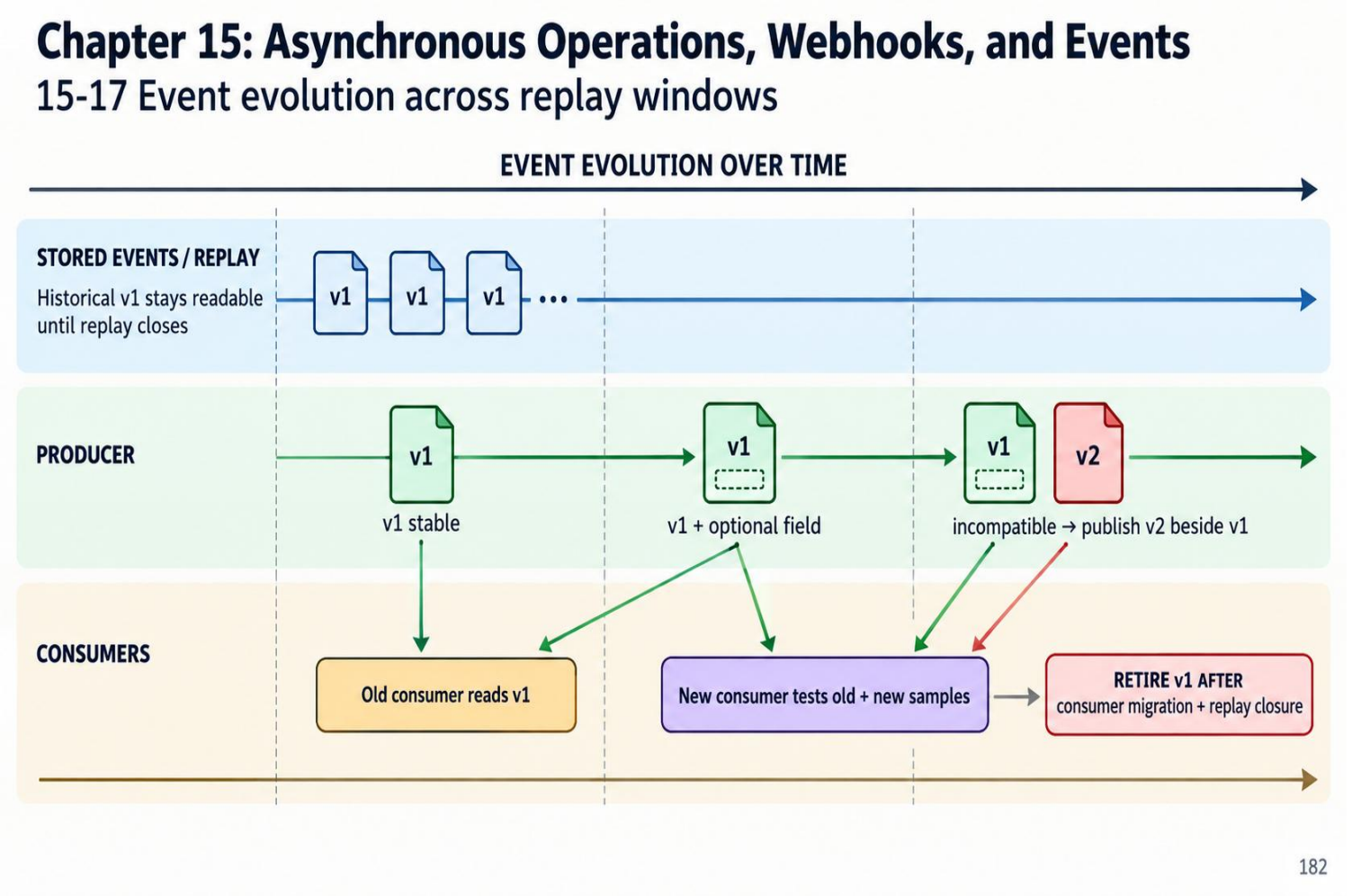



*Figure 15-19: Event-schema evolution across replay and consumer windows*

When a change is incompatible, publish a new event type or version and run both during migration. Do not silently reinterpret the same field in stored historical events.

## Use events only when the coupling trade-off helps

Events and webhooks are not automatically more scalable or modern than synchronous APIs. They add:

- Subscription and schema management.
- Duplicate and order handling.
- Eventual consistency.
- Harder end-to-end debugging.
- Replay and retention decisions.
- More operational components and ownership boundaries.

A synchronous call is often clearer when one service needs an immediate decision from another. A periodic read may be enough for a low-urgency report. A webhook is useful when timely notification matters and the receiver can operate a reliable endpoint.

Choose the simplest interaction that satisfies latency, reliability, and ownership needs.

## Complete export and assignment-notification designs

For the resolved-ticket export, the complete flow is:

1. Client submits a validated export request with an idempotency key.

2. Provider returns `202 Accepted`, an operation resource, and polling guidance.
3. Client polls at a bounded interval.
4. Provider reports `pending` or `running` without implying completion.
5. Operation reaches `succeeded` with an expiring result link, or `failed` with a stable error.
6. Client downloads the result under normal authorization.
7. Operation metadata and result expire according to documented retention.

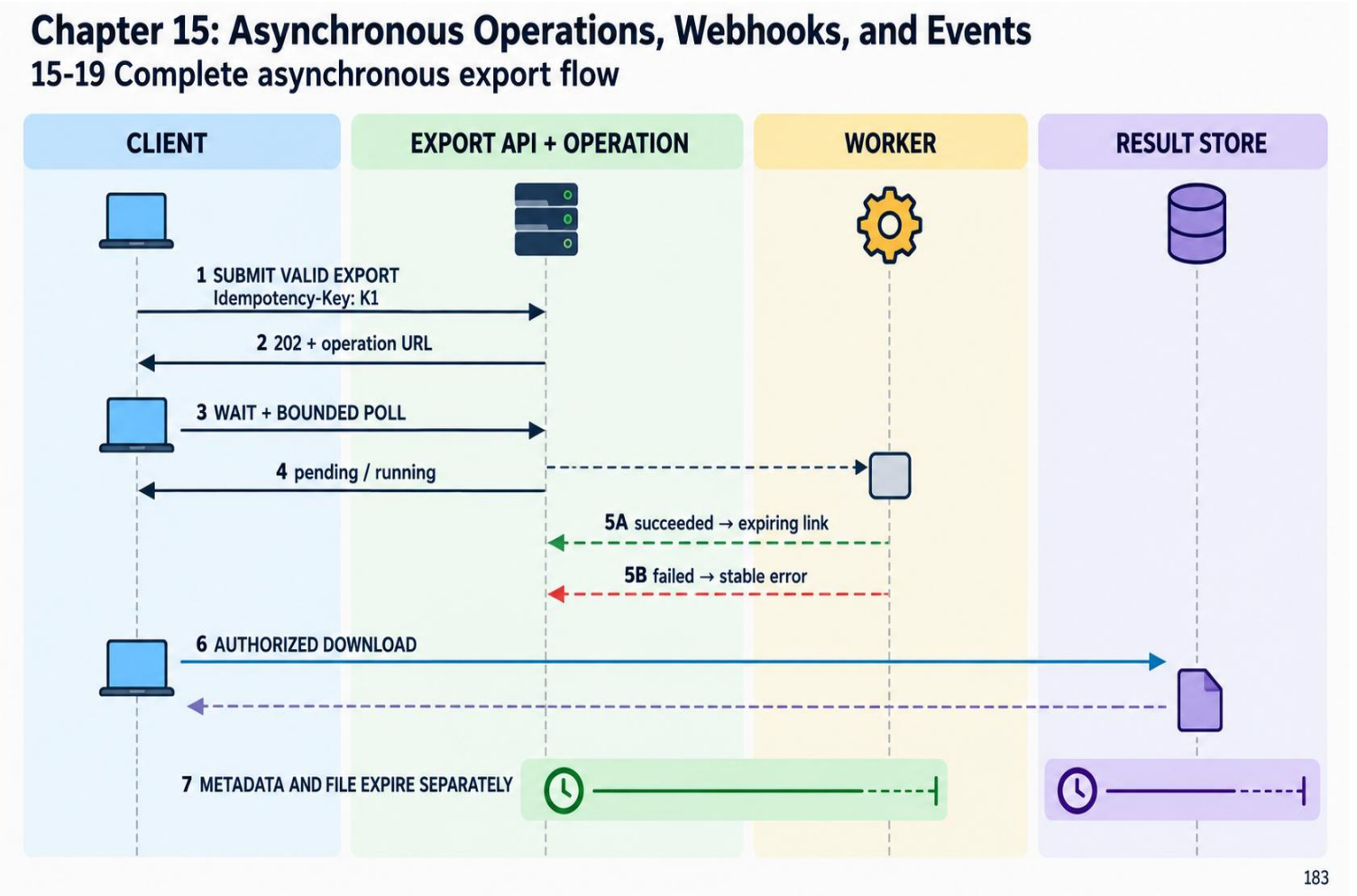



*Figure 15-20: Complete asynchronous export flow*

For `ticket.assigned` webhook delivery:

1. An authorized consumer registers and verifies an HTTPS endpoint.
2. Provider stores event type selection and signing configuration.
3. Ticket assignment commits before the event is treated as a completed fact.
4. Provider sends a signed delivery with event ID and timestamp.
5. Receiver verifies signature and replay window.
6. Receiver durably records the event and deduplicates by ID.
7. Receiver returns success quickly and performs longer work asynchronously.
8. Provider retries bounded failures according to policy and exposes delivery diagnostics to the subscription owner.

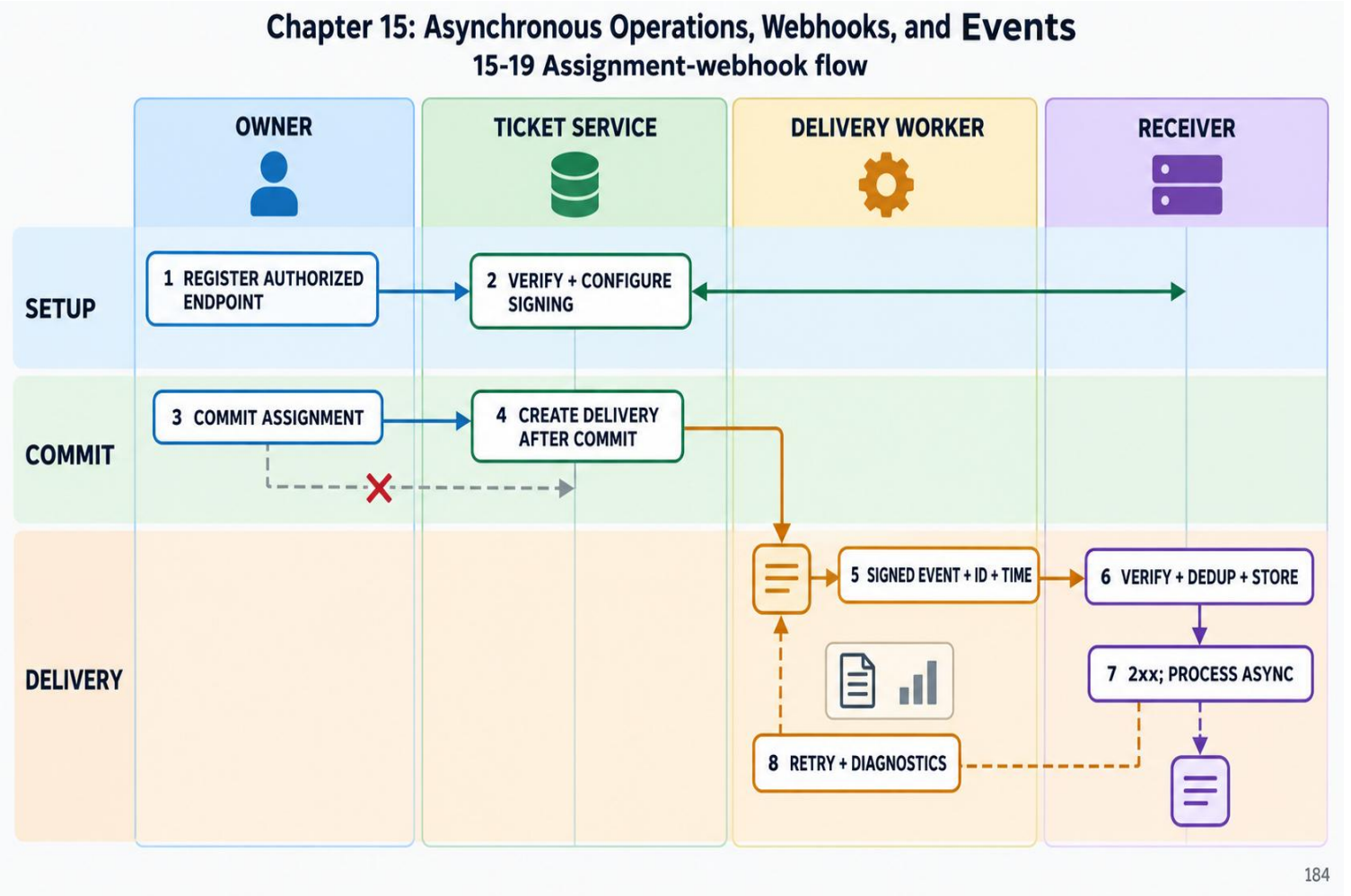



*Figure 15-21: Complete assignment-webhook flow*

These flows make acceptance, durable state, notification, and consumer processing separate responsibilities.

## Two asynchronous-design misconceptions

### "Returning `202 Accepted` means the operation succeeded"

`202` confirms acceptance for processing, not completion. The operation can later succeed, fail, be cancelled, or expire. The client needs a status contract and terminal result.

### "A webhook will be delivered exactly once and in order"

Timeouts and retries create duplicates, and parallel delivery can change order. Receivers need stable event identifiers, deduplication or idempotent effects, and an explicit ordering strategy where order matters.

## Asynchronous contract review

Before introducing asynchronous work or notification, confirm that:

- The use case cannot be served more simply by a bounded synchronous interaction.
- `202` responses point to a durable status mechanism.
- Operation states, terminal results, failures, cancellation, and expiration are defined.
- Polling has minimum or recommended intervals.
- Operation creation has safe duplicate-submission behavior.
- Webhook endpoints are authorized, verified, and restricted appropriately.
- Deliveries are authenticated and protected against replay.
- Receivers acknowledge only after durable acceptance.
- Duplicate delivery and limited ordering are normal tested conditions.
- Event IDs, resource versions, timestamps, and correlation IDs have distinct purposes.

- Command, event, and resource messages are not confused.
- Event schemas have a compatibility and replay strategy.
- Ownership exists for subscriptions, dead deliveries, and incident response.

**Asynchronous design is complete only when the consumer can determine what was accepted, what eventually happened, and how to recover when messages are delayed, duplicated, or never reach a successful terminal outcome.**

## Chapter Summary

- `202 Accepted` confirms acceptance for processing, not successful completion.
- Webhook and event delivery must assume retries, duplicates, limited ordering, and receiver failure.
- Asynchronous designs need durable identity, correlation, terminal outcomes, retention, and security rules just as synchronous APIs do.

## Review Questions

1. What information must an operation resource expose after a long-running request is accepted?
2. Why should a webhook receiver acknowledge only after durable acceptance?
3. How do a command, an event, and a resource representation communicate different meanings?

# Chapter 16 — Designing for Observability and Operations

**An API should make each important interaction traceable enough to diagnose failures, measure service quality, and investigate significant actions without exposing sensitive data.** Observability is not an afterthought added by an operations team; the contract and implementation must produce signals that support real consumer and support workflows.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Distinguish logs, metrics, traces, audit records, health checks, readiness checks, and runbooks.
- Propagate request and correlation identifiers across synchronous and asynchronous work.
- Design telemetry that supports diagnosis while controlling sensitive data and metric cardinality.

## Observability turns signals into understanding

**Observability** is the ability to understand a system's internal behavior from the signals it produces, such as logs, metrics, and traces. The goal is not to collect the most data. The goal is to answer questions that matter:

- Which operation is failing?
- Which consumers are affected?
- Did the request reach the provider?
- Which dependency consumed the time?
- Was a state change committed?
- Was access denied correctly?
- Is the problem isolated or widespread?
- Did a retry or rate limit contribute to the outcome?

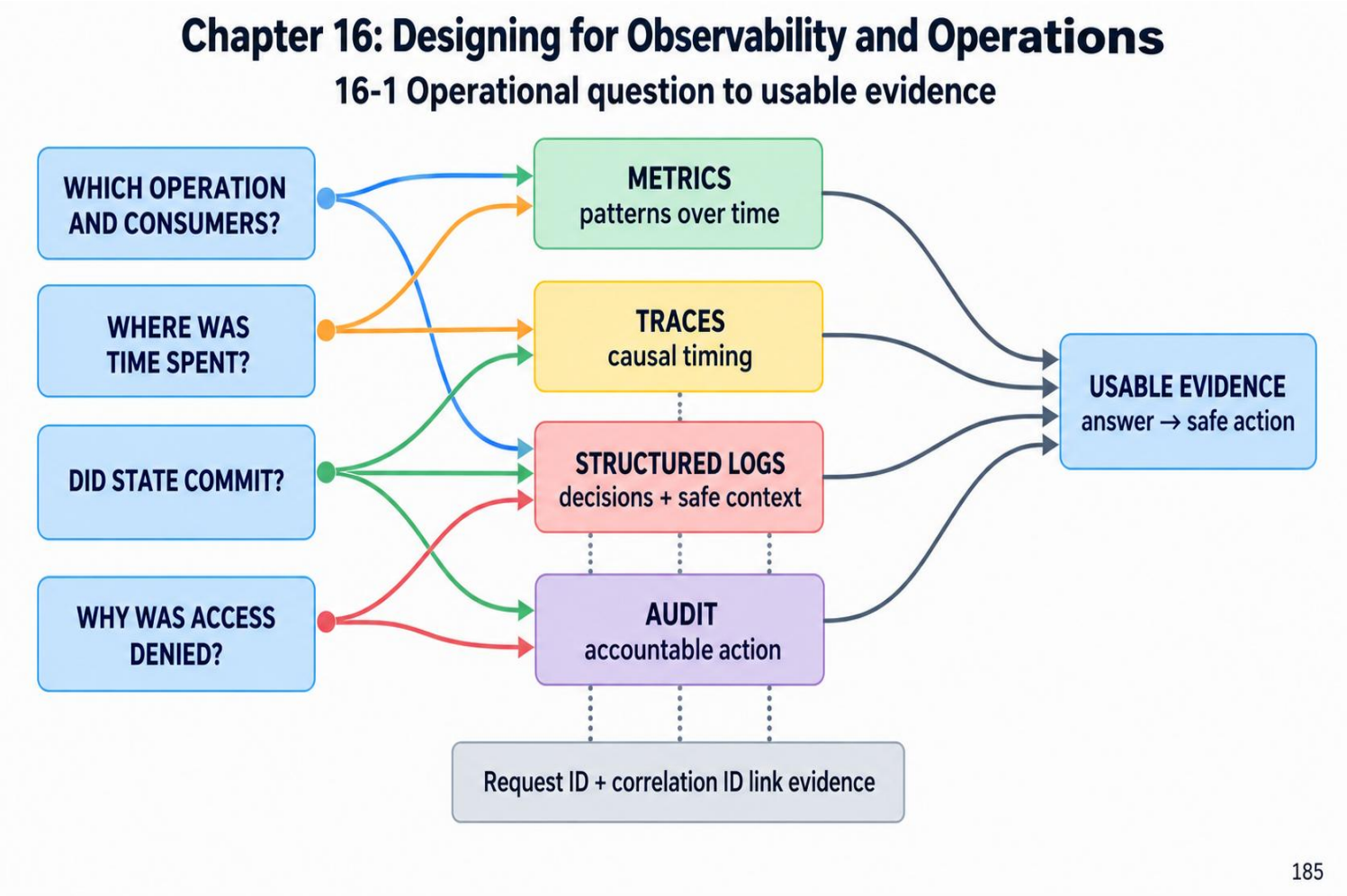



*Figure 16-1: From operational question to usable evidence*

An observable API connects a consumer-visible symptom to provider-side evidence. An unobservable API leaves engineers comparing timestamps, searching unstructured text, and guessing which request a user meant.

## Distinguish the main operational signals

Several signal types serve different purposes:

| Signal | Main purpose | Typical contents | Retention and access tendency |
|---|---|---|---|
| Access log | Record that a request crossed an HTTP boundary | Method, route template, status, duration, caller class, request ID | Operational retention; restricted because metadata can be sensitive |
| Application log | Explain notable application behavior | Validation branch, dependency failure, retry, state-transition decision | Retained for diagnosis; detail varies by environment |
| Audit record | Establish who performed a significant action and what outcome occurred | Actor, action, resource, time, result, reason, correlation | Longer controlled retention; stronger integrity requirements |
| Metric | Measure behavior as numeric time series | Request count, latency, error rate, saturation, retry count | Aggregated over time; optimized for monitoring and alerting |
| Trace | Follow one workflow across components | Spans, timing, parent-child relationships, selected attributes | Sampled or bounded due to volume and sensitivity |

These signals overlap but are not interchangeable. A metric can reveal that assignment latency increased, while a trace shows which dependency caused one slow request. An audit record can prove who changed the assignee, while an application log helps explain why an internal retry occurred.

## Give every request a usable identifier

A **request identifier** is an opaque value that uniquely identifies one request as it passes through the API boundary. The provider should accept a caller-supplied identifier only if it can validate and safely use it; otherwise, it should generate one.

A response can return:

```
X-Request-Id: req_71d302
```

The same value can appear in the error body where that API's error format includes it:

```
{
  "type": "https://api.example.com/problems/ticket-not-assignable",
  "title": "The ticket cannot be assigned",
  "status": 409,
  "code": "TICKET_NOT_ASSIGNABLE",
  "detail": "A closed ticket cannot be assigned.",
  "requestId": "req_71d302"
}
```

When a consumer reports a failure, the request ID lets support staff locate the relevant gateway and service evidence without asking for the complete request body.

Request IDs should be:

- Opaque to clients.
- Safe to include in logs and support tickets.
- Unique enough for the retention period.
- Propagated through important internal calls.
- Returned consistently for both success and failure where practical.

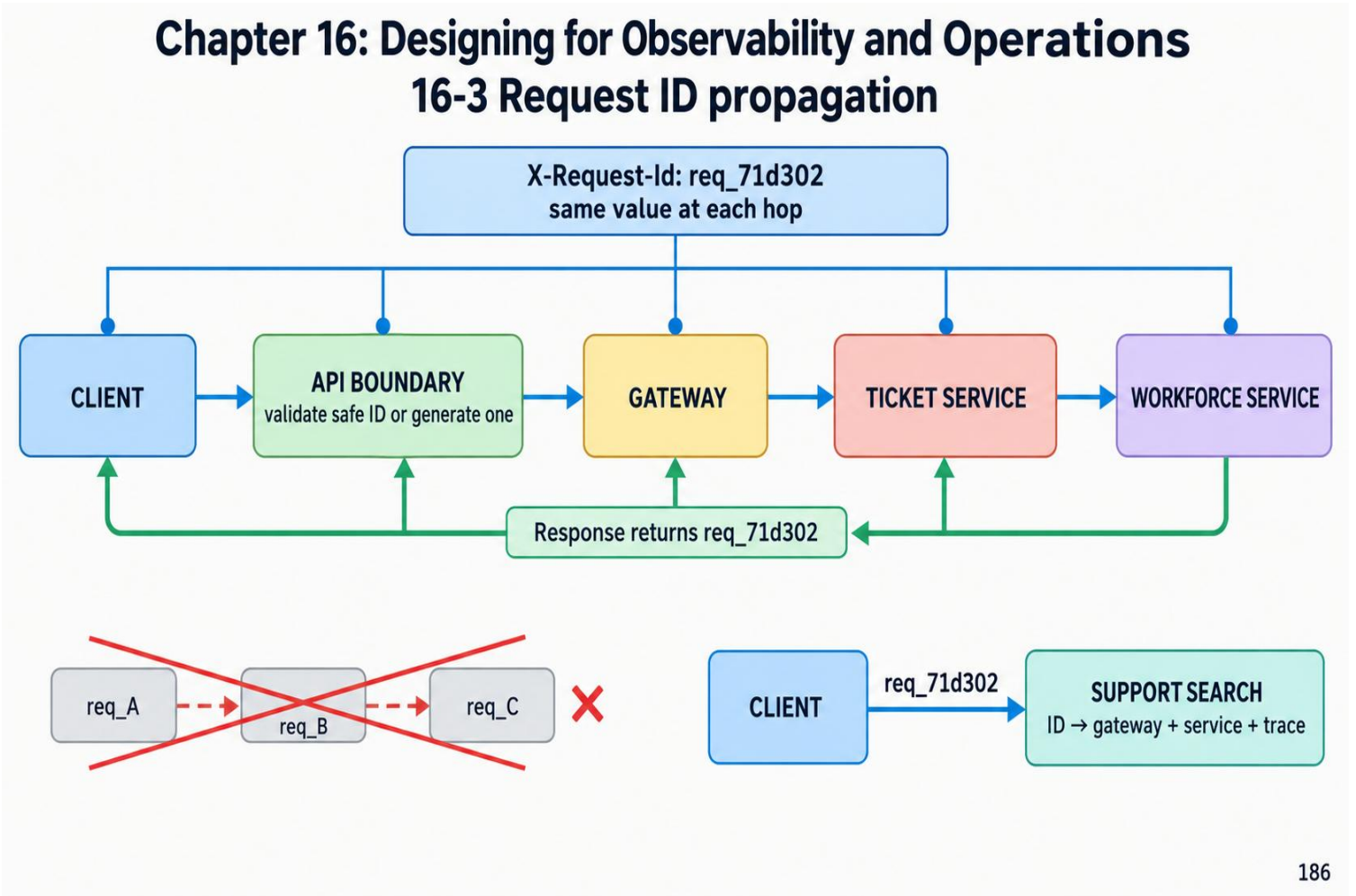



*Figure 16-2: Request ID propagation through the API path*

Do not encode user email addresses, ticket subjects, tokens, or other sensitive information in the identifier.

## Correlation connects several requests and messages

A single workflow may include several requests, asynchronous work, and events. A **correlation identifier** links those related interactions, while each interaction still has its own request or delivery identifier.

For example, assigning a ticket may produce:

- Client request `req_101`.
- Workforce-service request `req_102`.
- Audit write `audit_550`.
- Event delivery `evt_68321`.
- Shared correlation identifier `corr_900`.



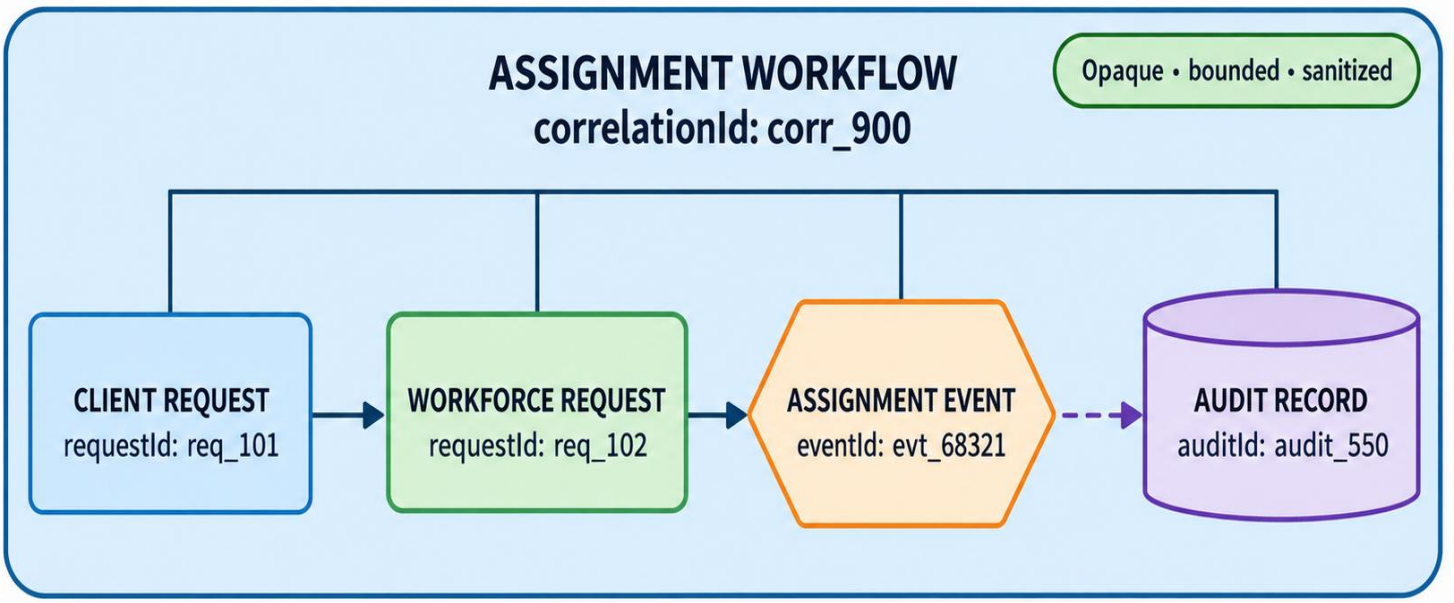





*Figure 16-3: Request, correlation, event, and audit identifiers*

The distinction prevents one identifier from being overloaded. The request ID answers “Which network interaction?” The correlation ID answers “Which broader business workflow?” The event ID answers “Which event delivery fact?”

Correlation context should be propagated through supported headers or messaging metadata according to the organization’s tracing standard. A client-provided correlation value should be bounded and sanitized before logging.

## Structured logs are easier to search and govern

A **structured log** stores named fields rather than relying on a sentence that must be parsed later. A ticket-assignment log entry might be:

```
{
  "timestamp": "2026-09-24T08:45:12.318Z",
  "level": "INFO",
  "service": "ticket-api",
  "operation": "assignTicket",
  "requestId": "req_71d302",
  "correlationId": "corr_900",
  "actorId": "usr_314",
  "ticketId": "tkt_48271",
  "outcome": "rejected",
  "reasonCode": "TICKET_NOT_ASSIGNABLE",
  "durationMs": 47
}
```

Named fields allow engineers to filter by operation, outcome, request ID, or reason code. The text message, if present, is for readability; stable fields support automation.

Use route templates such as `/tickets/{ticketId}` in access-log aggregation and metrics rather than raw paths such as `/tickets/tkt_48271`. Raw identifiers create **high cardinality**, meaning too many distinct dimension values for efficient aggregation, and can expose business information. A restricted per-request application log may record a ticket identifier when diagnosis requires it, but that identifier should not become a general metric dimension.

## Log decisions, not every piece of data

More logging does not automatically produce better diagnosis. Full request and response bodies can contain credentials, personal information, attachments, internal comments, or data subject to retention rules.

A safe logging policy identifies:

- Fields that may be logged directly.
- Fields that must be removed or masked.
- Fields that must never enter routine telemetry.
- Environments in which diagnostic detail differs.
- Retention and access controls.
- Procedures for exceptional temporary debugging.

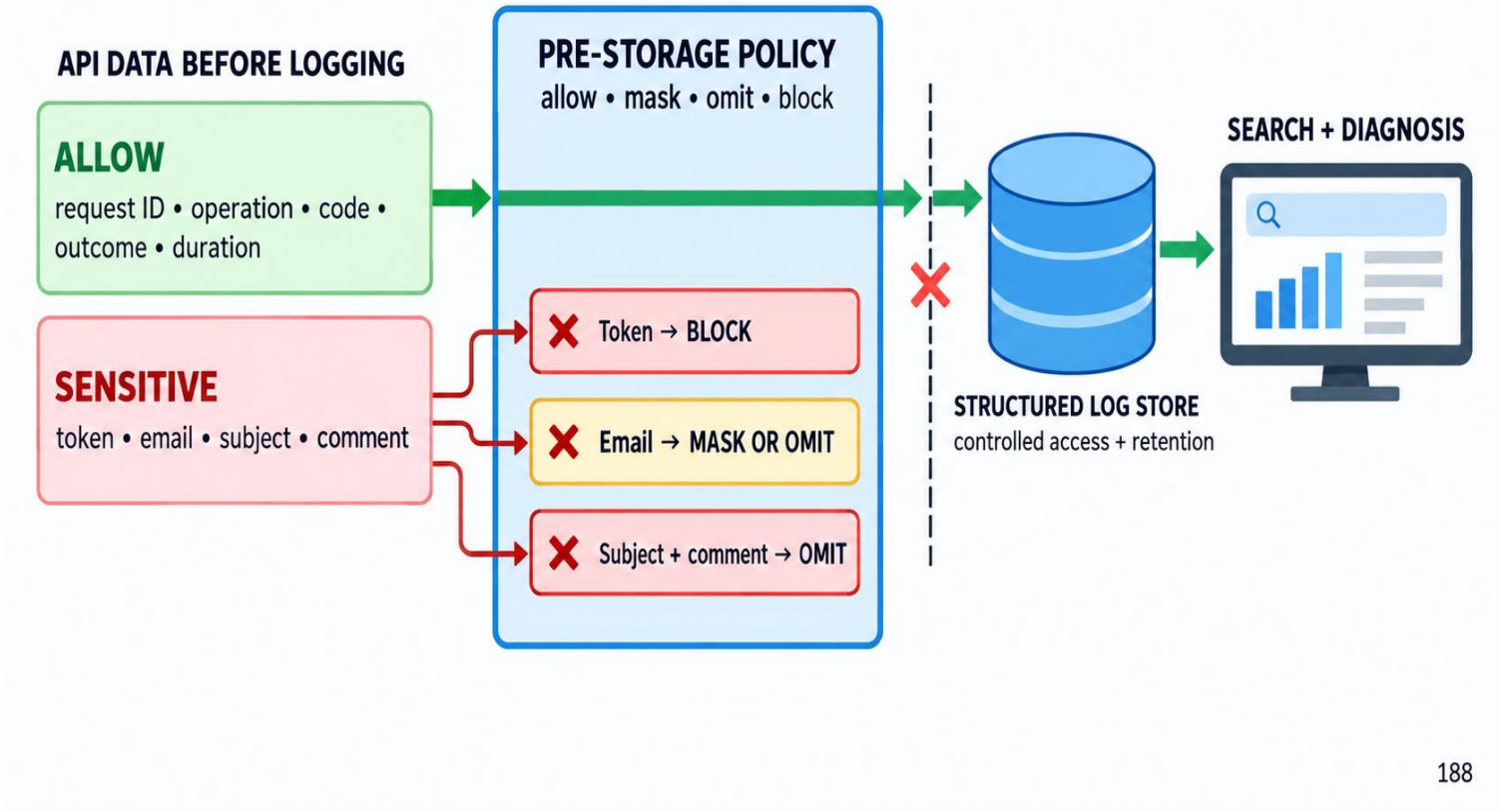





*Figure 16-4: Safe logging and redaction boundary*

Useful logs often record decisions and references rather than content:

- “Authorization denied because queue access was absent.”
- “Assignment rejected because ticket state was closed.”
- “Downstream eligibility check timed out after 300 ms.”
- “Idempotency key matched a completed operation.”

They do not need the access token, entire ticket description, or full comment body to convey those facts.

**Redaction** removes or masks sensitive values before they are stored or transmitted to telemetry systems. It must occur before data reaches the logging sink; a dashboard that hides a token visually does not remove it from stored logs.

## Metrics show patterns over time

A **metric** is a numerical measurement recorded over time, usually with a controlled set of dimensions. Foundational API metrics include:

- Request volume by operation and status class.
- Latency by operation, including relevant percentiles.
- Error count by stable error category.
- Current concurrency, queue depth, connection use, or other saturation indicators; **saturation** means how close a constrained resource is to its usable capacity.
- Retry and timeout counts.
- Rate-limit and load-shedding events.
- Asynchronous operation age and terminal outcome.
- Webhook delivery success, retry, and permanent failure.

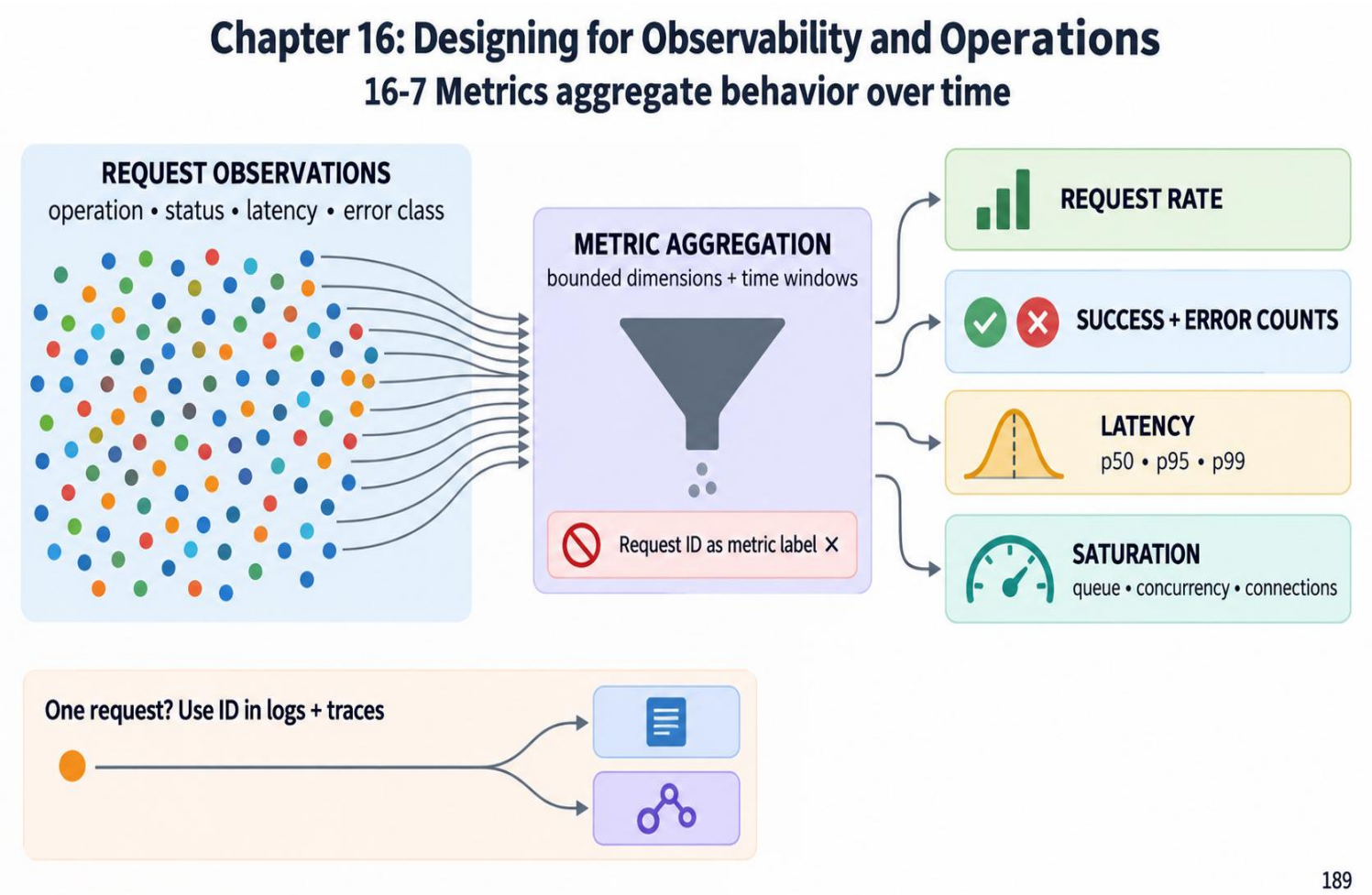



*Figure 16-5: Metrics aggregate behavior over time*

A **percentile** indicates a value below which a proportion of observations falls; for example, the 95th percentile latency is the duration under which 95% of measured requests completed. Percentiles reveal slow-tail behavior that an average can hide.

Metrics should align with consumer experience. Combining fast health checks with slower ticket searches into one latency number can make the overall average look healthy while users experience delays.

## Control metric cardinality

**Cardinality** is the number of distinct values a metric dimension can take. High-cardinality dimensions consume monitoring resources and make queries unreliable or expensive.

Good metric dimensions often include:

- Route or operation name.
- Response status class.
- Stable error category.
- Deployment region.
- Controlled consumer class or service identity.

Poor general-purpose metric dimensions include:

- Ticket ID.
- Request ID.
- User email address.
- Raw query string.
- Error message text.

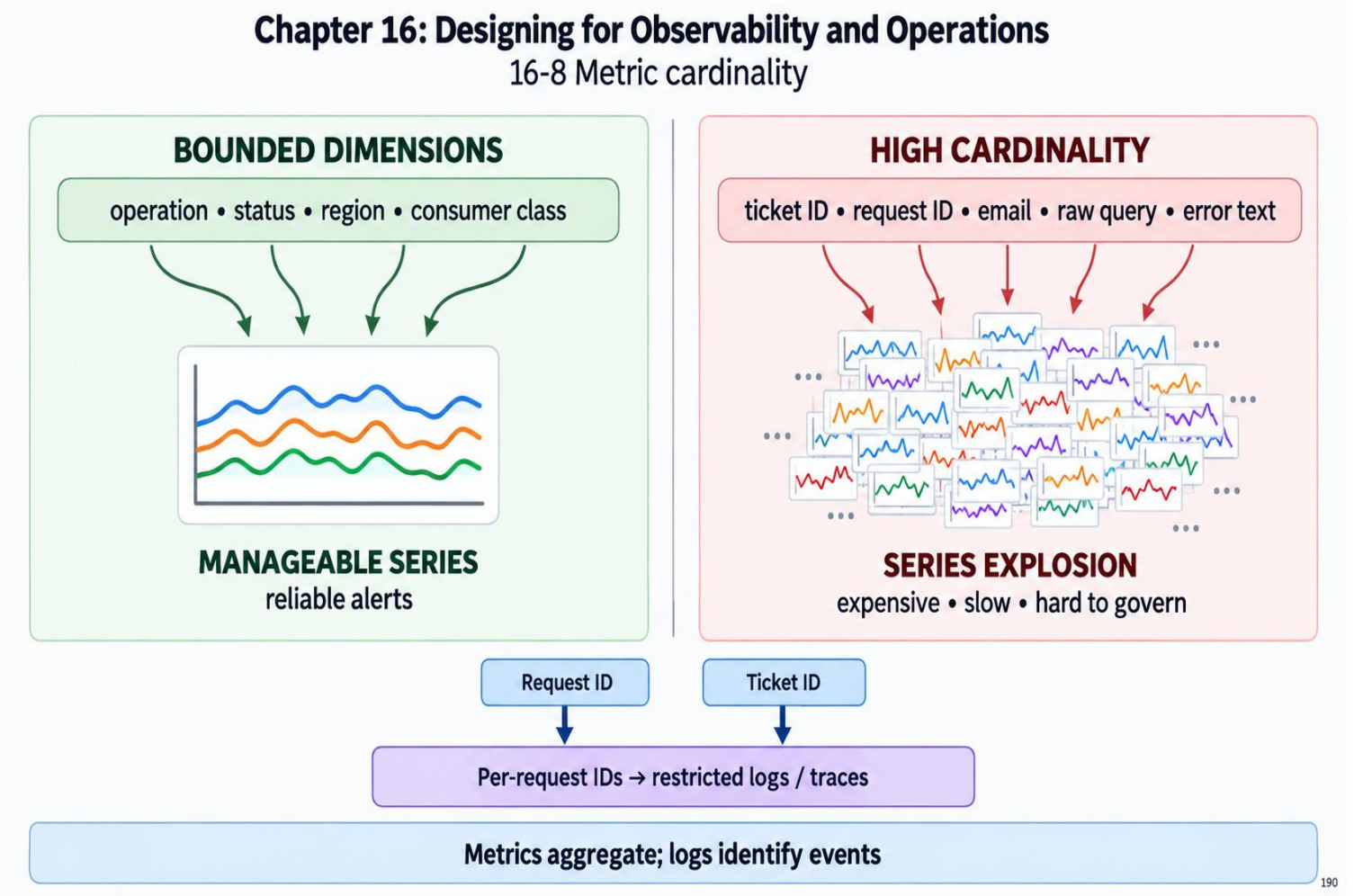



*Figure 16-6: Metric-cardinality explosion*

Use logs or traces for per-request identifiers. Use metrics for bounded aggregation.

## Monitor by operation and consumer where it changes action

Overall service health can hide concentrated failures. The provider should be able to identify whether:

- `assignTicket` is failing while reads remain healthy.
- One tenant is being rate-limited heavily.
- A particular service credential sends invalid requests after a deployment.
- One API version has worse latency than another.
- Webhook delivery failures affect one destination.

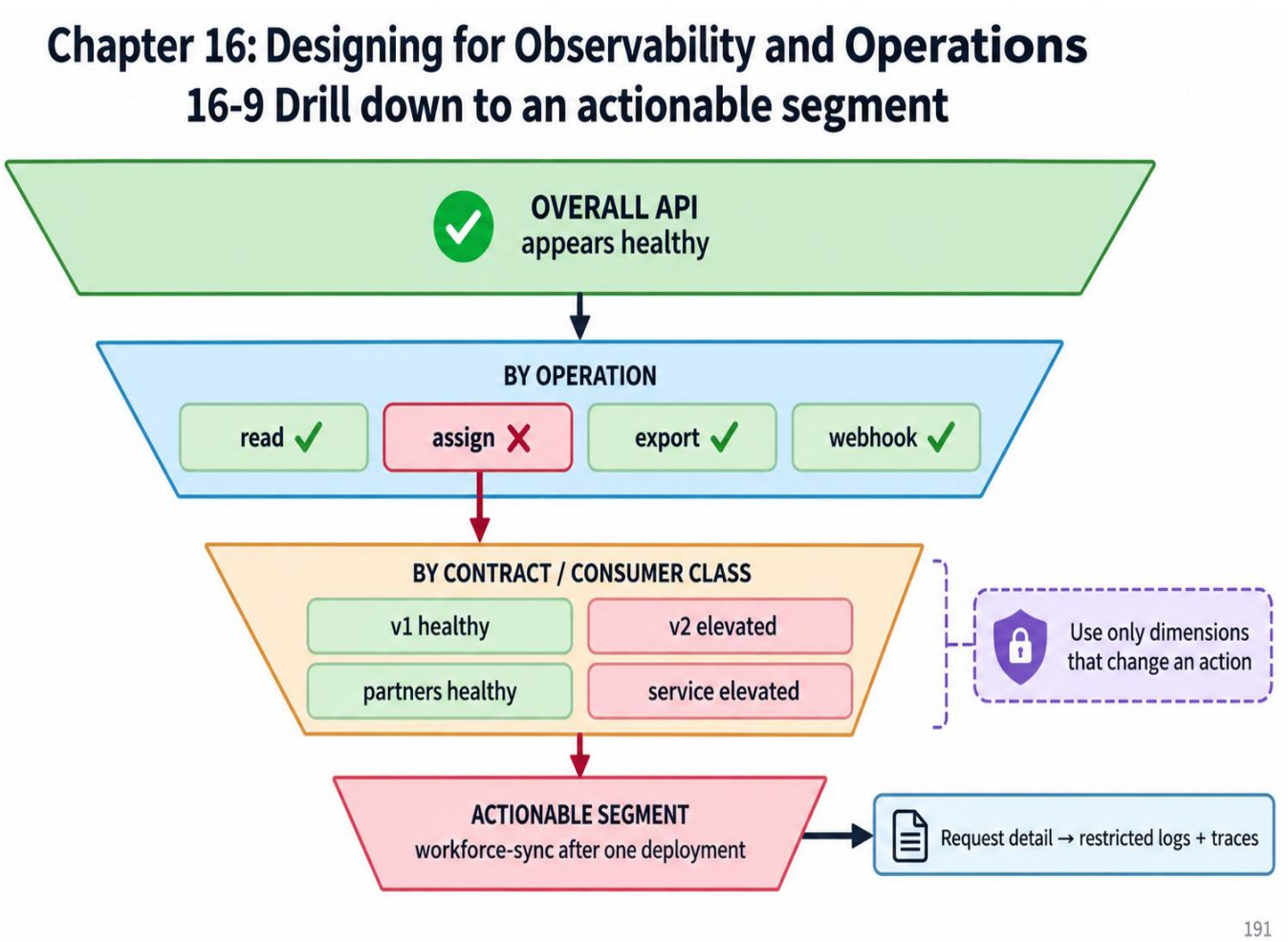



*Figure 16-7: Drilling from service health to operation and consumer segment*

Consumer-level monitoring must respect privacy and metric-cardinality constraints. A controlled consumer identifier or class may be appropriate for partner and service integrations. Individual end-user identifiers usually belong in restricted logs or audit records rather than broad metrics.

A dimension is useful only when the team has a reason to act on it. Collecting every possible label creates cost without improving response.

## Distributed traces follow work across components

A **distributed trace** records the path and timing of one request or workflow across multiple components. A **span** represents one timed unit of work within that trace, such as gateway processing, a database query, or a downstream API call.

A trace for ticket assignment might contain:

1. Gateway span: authenticate and route.
2. Ticket-service span: load ticket and check policy.
3. Workforce-service span: verify assignee eligibility.
4. Database span: update assignment and state.
5. Event-publication span: record notification for delivery.

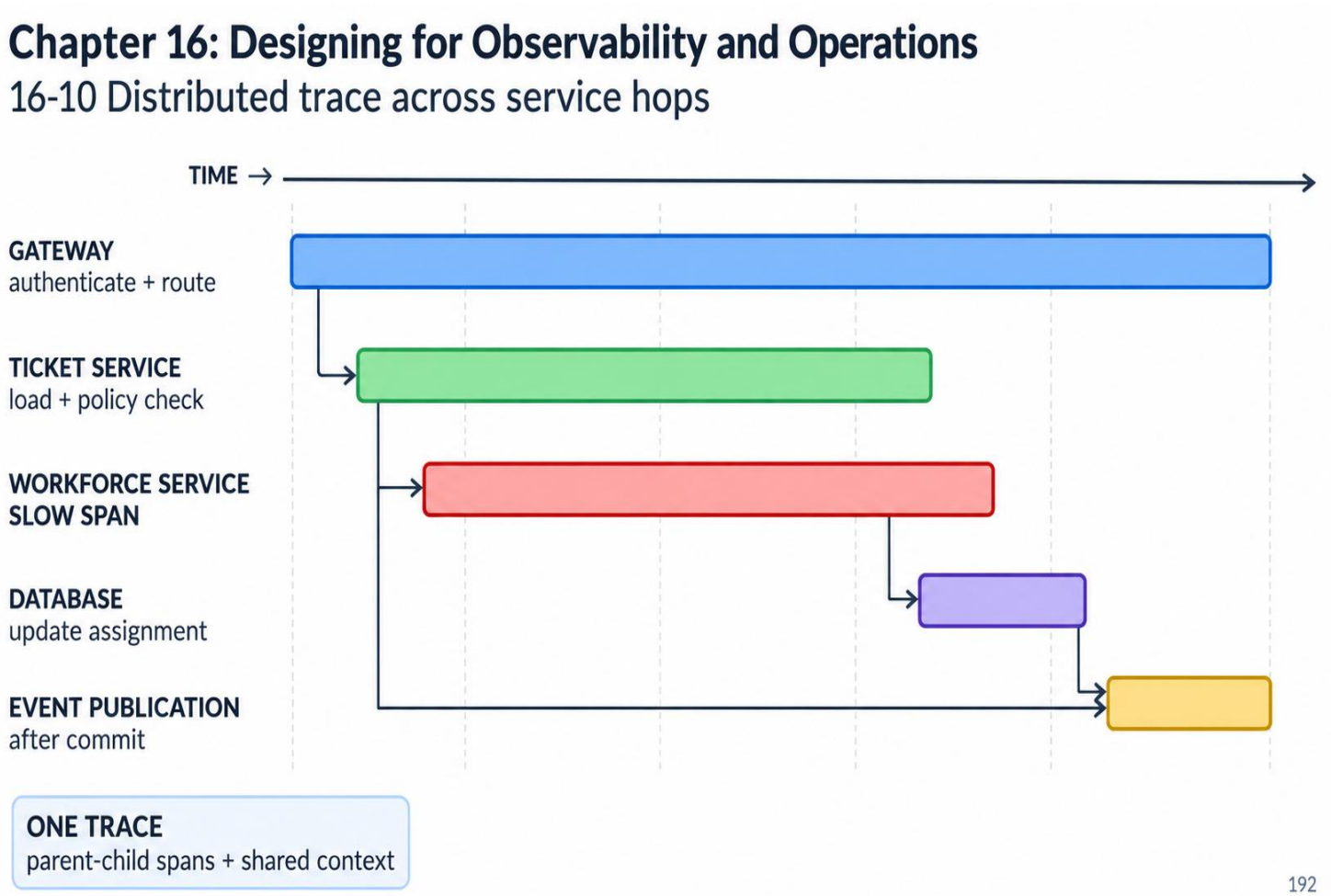



*Figure 16-8: Distributed trace across gateway, service, dependency, database, and publication*

The trace helps answer whether latency came from authentication, policy evaluation, a dependency, storage, or publication. It should record operation names and safe attributes, not entire message bodies.

Trace sampling keeps only a proportion or selected class of traces to control volume. Sampling policies should retain enough errors and slow requests for diagnosis while protecting sensitive data.

Trace context can cross HTTP requests and messages. The provider should use established propagation libraries and standards rather than inventing an incompatible header format.

## Audit records support accountability

An **audit log** is a controlled record of security-sensitive or business-significant actions, designed to support accountability and investigation. It is not merely a verbose application log.

For a ticket assignment, an audit record may include:

- Trusted actor identity.
- Action performed.
- Target ticket identifier.
- Previous and resulting assignee identifiers where policy permits.
- Time of decision and completion.
- Outcome and stable reason code.
- Request and correlation identifiers.
- Source application or credential identity.

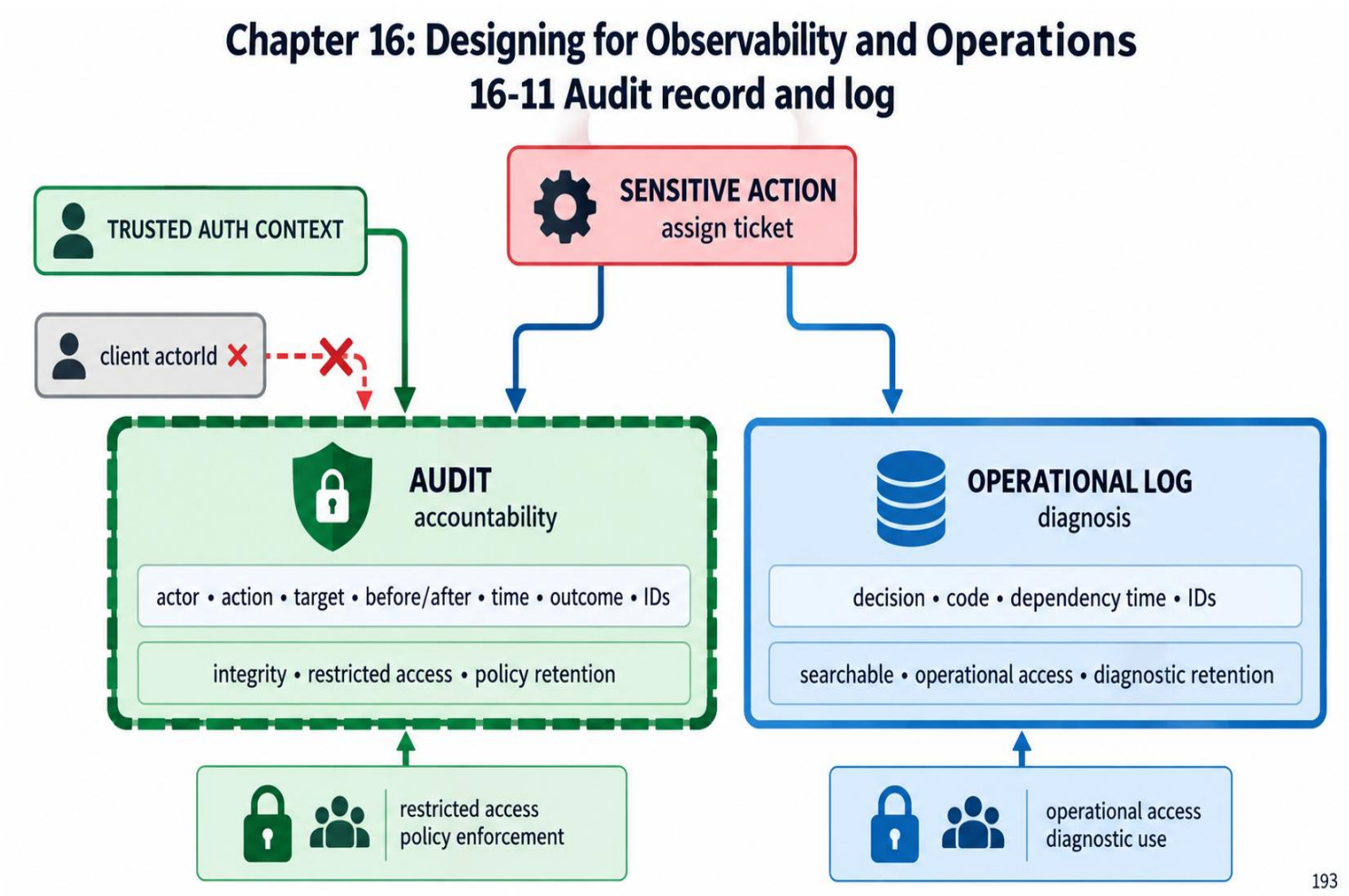



*Figure 16-9: Audit record versus operational log*

Audit records often need stronger access control, integrity protection, and retention than ordinary logs. They should be resistant to casual alteration by the same application path that performs the action.

Do not assume every read needs a permanent audit record. Audit scope should follow security, legal, privacy, and business requirements. Excessive audit data can itself become a sensitive asset.

The actor in an audit record must come from trusted authentication context, not a client-supplied `actorId` field.

## Separate client errors from internal evidence

A client-visible error should provide enough information to take the next safe action and to identify the request during support. Internal telemetry can contain more operational detail, subject to data policy.

For example, the client may receive:

```
{
  "type": "https://api.example.com/problems/assignment-temporarily-unavailable",
  "title": "Assignment is temporarily unavailable",
  "status": 503,
  "code": "ASSIGNMENT_TEMPORARILY_UNAVAILABLE",
  "detail": "The assignment could not be completed. Retry later.",
  "requestId": "req_71d302"
}
```

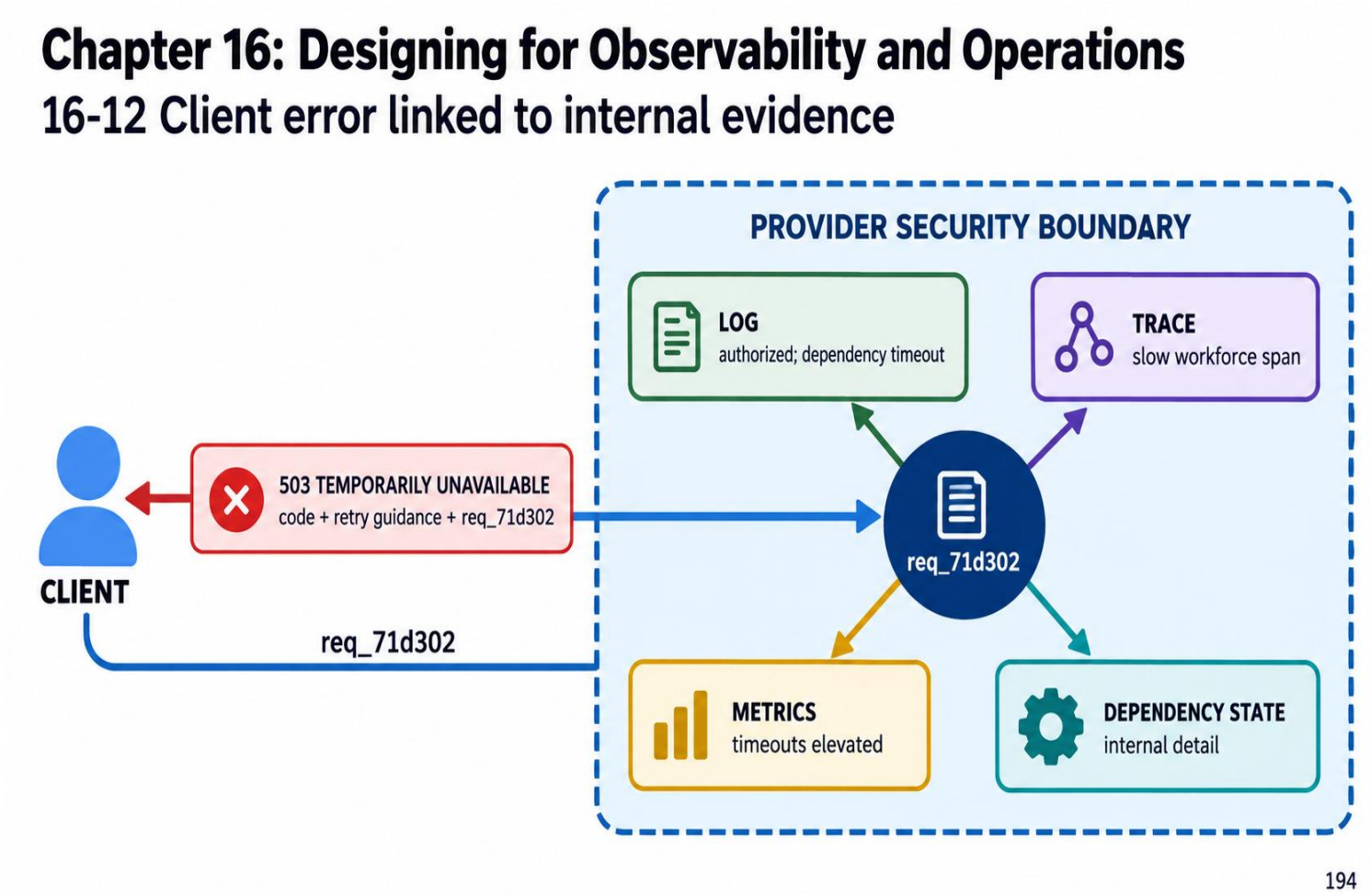



*Figure 16-10: Simple client error linked to rich internal evidence*

Internal evidence may record that the workforce dependency exceeded a 300-millisecond timeout and that the circuit breaker changed state. The client does not need the dependency host name, internal stack, or network topology.

This separation lets support teams diagnose problems without turning error responses into an information leak.

## Health and readiness are operational interfaces

A **health endpoint** reports a bounded view of whether a process or service is functioning. A **readiness endpoint** reports whether an instance is currently prepared to receive normal traffic.

These endpoints should be separate from business operations and kept simple. Possible checks include:

- Process is running and its event loop or main execution path responds.
- Required configuration loaded successfully.
- The instance can reach a critical dependency within a short bound.
- Startup migration or initialization completed.

A readiness failure can remove an instance from traffic while allowing the process to remain available for diagnosis. A basic process-health response should not become unavailable merely because one optional downstream service is slow.

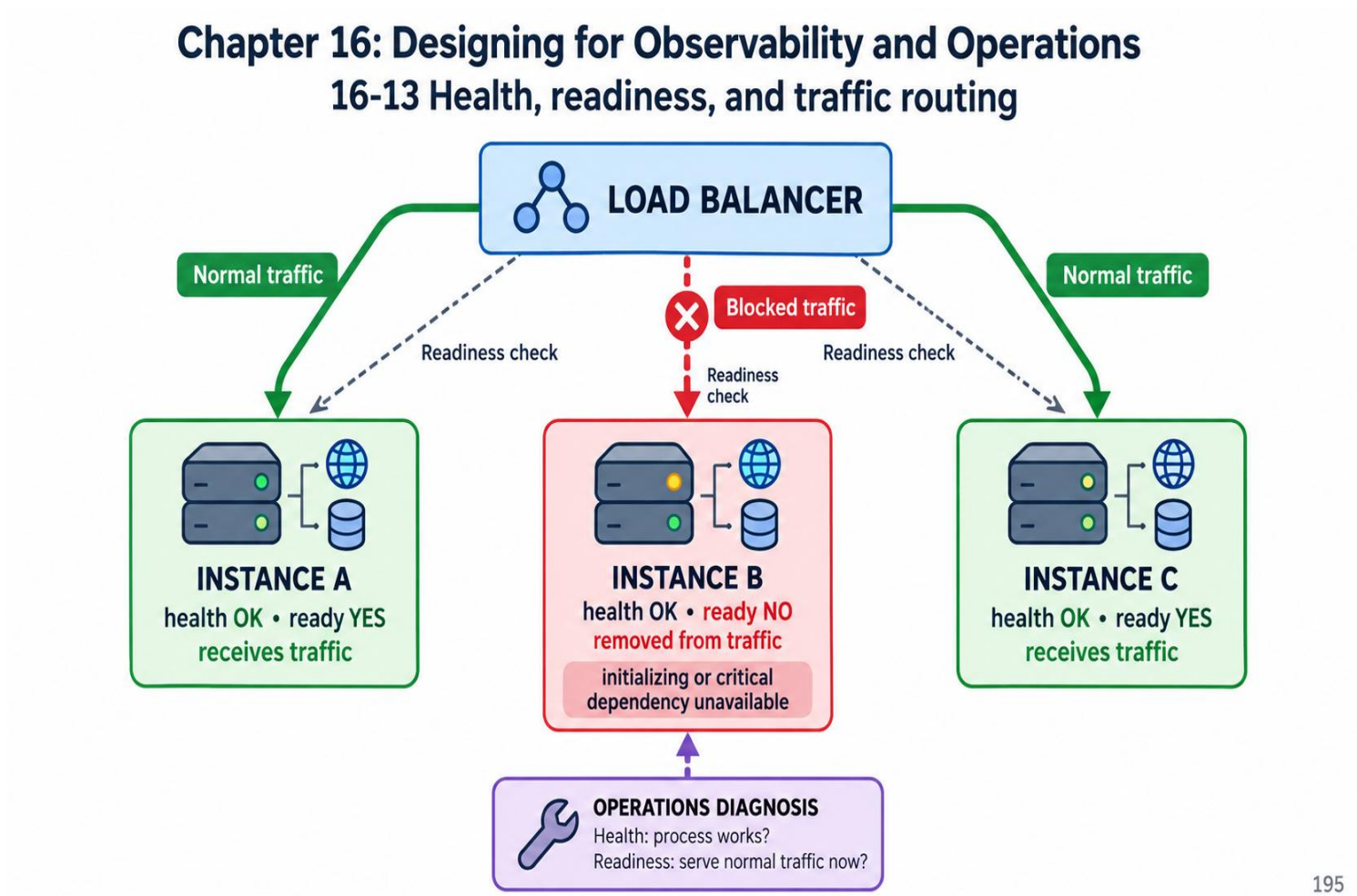



*Figure 16-11: Health versus readiness and traffic routing*

Health endpoints require security and exposure decisions. An external response may reveal only healthy or unavailable, while internal operations tooling can access more detail. Avoid returning credentials, internal host names, version details that create unnecessary exposure, or a full dependency map.

## Health does not prove business correctness

A service can pass its health check while ticket assignment fails because of a permission configuration error, a particular tenant's data issue, or a downstream business rule. Conversely, one optional dependency can be unavailable while core ticket retrieval remains useful.

Complement health checks with:

- Synthetic requests that exercise selected business paths.
- Operation-level error and latency metrics.
- Contract conformance monitoring.
- Queue and asynchronous-operation age alerts.
- Webhook delivery monitoring.

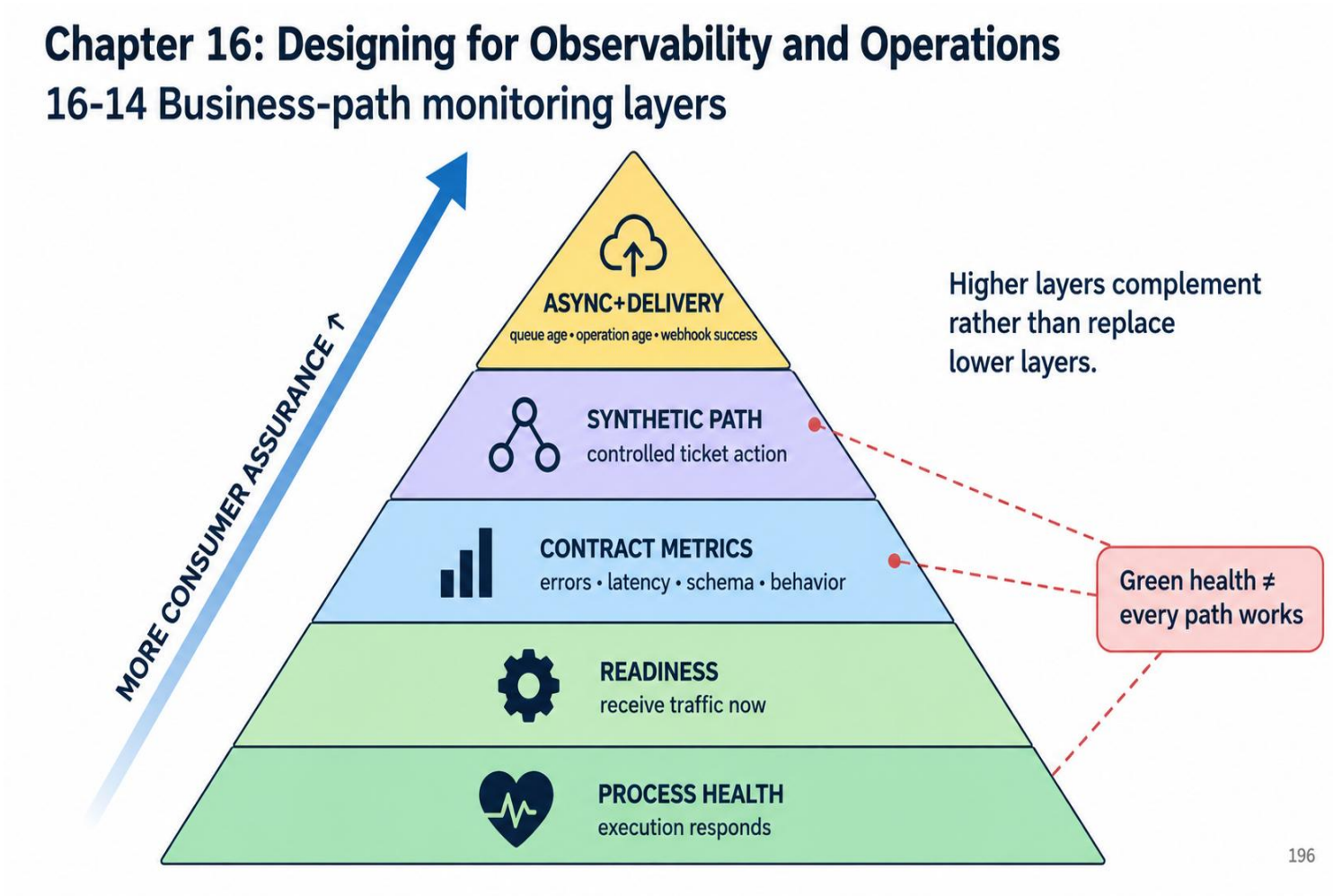



*Figure 16-12: Health checks versus business-path monitoring*

A **synthetic request** is an automated probe that performs a controlled interaction to verify a consumer-visible path. It should use test data and permissions designed for monitoring, not real customer records.

## Runbooks connect signals to action

A **runbook** is an operational guide for diagnosing and responding to a known class of service condition. API design should provide the information a runbook needs, such as:

- Expected operation latency and timeouts.
- Stable error codes and their likely categories.
- Retry and rate-limit behavior.
- Dependency and fallback behavior.
- Request and correlation identifier locations.
- Ownership and escalation contacts.
- Safe ways to verify or replay asynchronous work.
- Rules for credential compromise or webhook suspension.

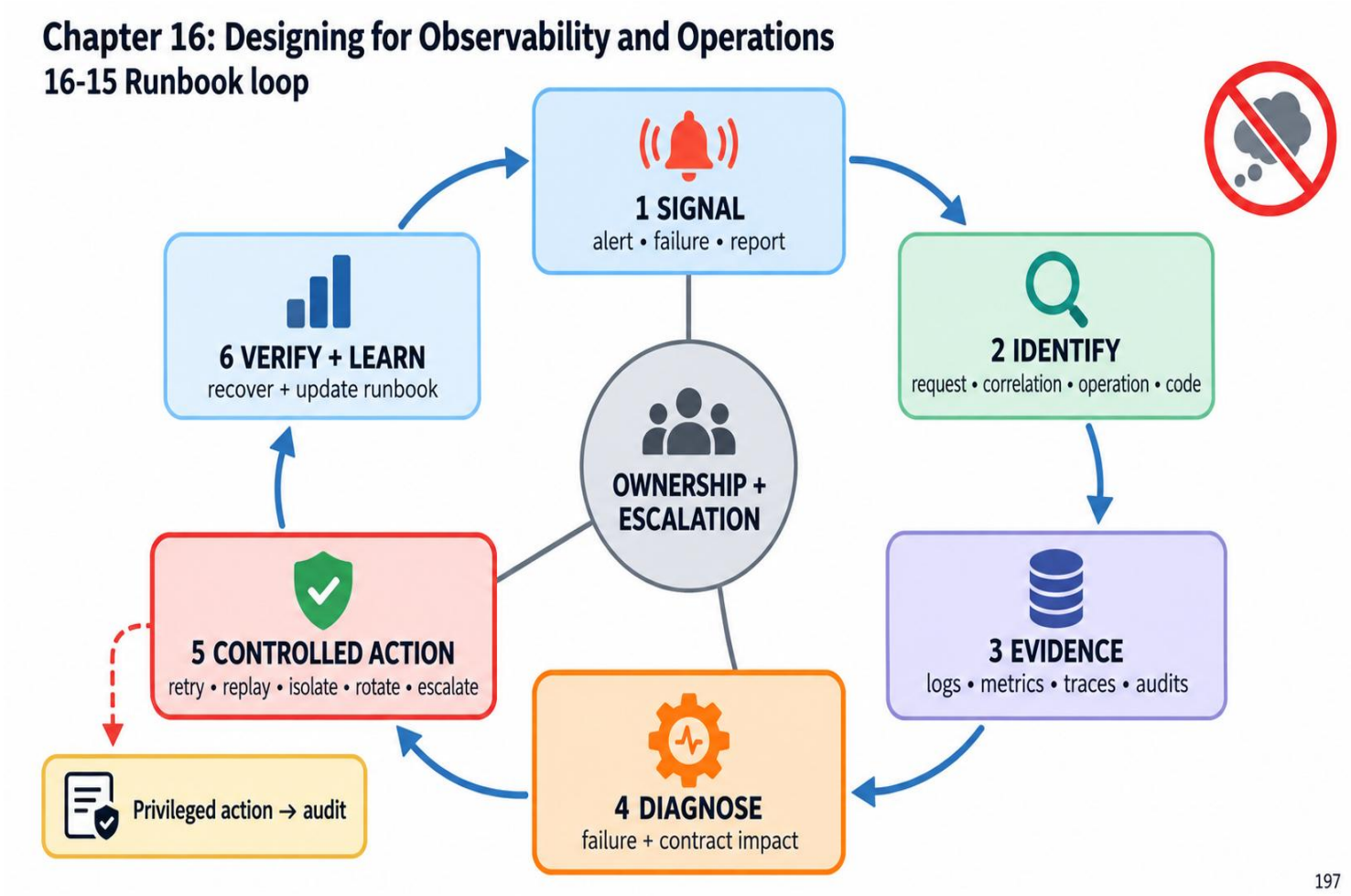



*Figure 16-13: Runbook loop from signal to diagnosis to safe action*

A runbook should not depend on undocumented database edits or guesswork. When recovery requires a privileged operation, that capability should be controlled, audited, and designed deliberately.

## Trace a failed assignment end to end

Suppose an agent receives `503 ASSIGNMENT_TEMPORARILY_UNAVAILABLE` with request ID `req_71d302`.

The support path can be:

1. **Client evidence:** The user interface records the operation, status, request ID, and time without storing the access token.
2. **Gateway access log:** Search `req_71d302` to confirm the request arrived, the caller authenticated, and the gateway returned `503` after 362 milliseconds.
3. **Ticket-service structured log:** The same request ID shows that authorization passed and dependency `assignee-eligibility` timed out before any state change.
4. **Trace:** The workforce-service span consumed 300 milliseconds and ended with a deadline-exceeded result.
5. **Metrics:** Assignment dependency timeouts increased across several tenants after a workforce-service deployment.
6. **Audit evidence:** No successful assignment audit record exists for this request, confirming that the protected state change did not commit.
7. **Runbook action:** The on-call engineer follows the dependency-timeout procedure, verifies recovery, and watches assignment success metrics.

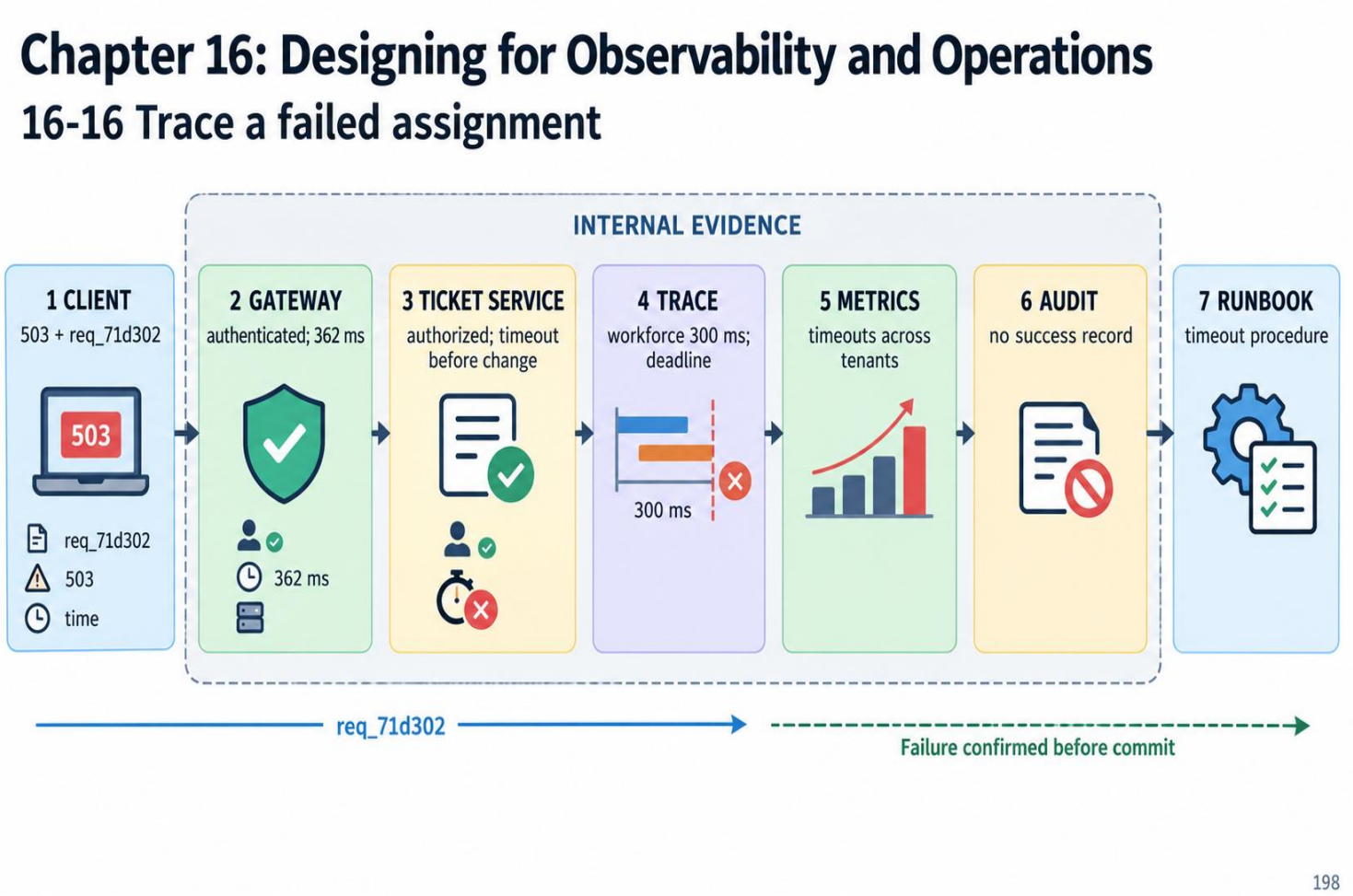





*Figure 16-14: Tracing a failed assignment end to end*

The client does not see internal service names or trace details. The request ID connects the simple client error to the richer operational evidence.

## Two observability mistakes

### "Logging every request body provides the best diagnostics"

Full bodies create security, privacy, retention, and cost risks while making important decisions harder to find. Log stable identifiers, outcomes, timings, and reason codes; capture sensitive content only through narrowly controlled exceptional procedures.

### "A health endpoint proves every business operation is functioning correctly"

A health endpoint verifies a limited operational condition. Business operations can fail because of policy, data, tenant-specific configuration, or downstream behavior that a basic health check does not exercise.

## Operational readiness review

Before releasing an API operation, verify that:

- Every request receives a safe, searchable identifier.
- Correlation can continue across important downstream calls and asynchronous messages.
- Access, application, audit, metric, and trace signals have distinct purposes.
- Structured logs use stable fields and route templates.
- Credentials, sensitive bodies, and unnecessary personal data are excluded from telemetry.
- Metrics cover volume, latency, errors, saturation, retries, limits, and asynchronous backlog where relevant.
- Metric dimensions have controlled cardinality.
- Important operations and consumer classes can be monitored separately.

- Traces reveal dependency timing without recording sensitive payloads.
- Security-significant actions produce trustworthy audit records.
- Client errors include useful correlation without internal disclosure.
- Health and readiness endpoints are bounded and separated from business contracts.
- Runbooks can connect a consumer symptom to evidence and safe action.

**An operable API makes failure explainable. It gives consumers a reference they can report, gives engineers signals they can connect, and preserves the security boundaries that diagnosis must not weaken.**

## Chapter Summary

- Observability connects operational signals to explanations of system behavior.
- Identifiers and structured context allow one consumer-visible failure to be followed across components and messages.
- Health, readiness, dashboards, alerts, audit evidence, and runbooks are operational interfaces with explicit audiences and limits.

## Review Questions

1. When should a correlation identifier differ from a request identifier?
2. Why is logging every request body unsafe and often operationally unhelpful?
3. What can a readiness check prove that a liveness check cannot?

# Part VI — Choosing and Applying an API Style

# Chapter 17 — Choosing Among API Styles

**No API style is best for every interaction.** Choose a style by examining who the consumers are, how they communicate, which guarantees they need, and which operational costs the organization can support.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Compare resource-oriented HTTP, RPC, gRPC, GraphQL, event-driven, and SOAP-based APIs from consistent perspectives.
- Match API style to client diversity, control, typing, streaming, caching, and operational constraints.
- Recognize when several API styles should coexist around one domain.

## Start with the interaction, not the technology name

An API style shapes how consumers express requests, how messages are typed, how calls are transported, and how change is managed. Before choosing one, ask:

- Is the interaction request-response, streaming, or asynchronous publication?
- Are consumers browsers, mobile applications, external partners, or controlled internal services?
- Do consumers need stable resources, named commands, flexible queries, or event notifications?
- How important are human readability and simple HTTP tooling?
- Is strong generated typing valuable across the supported languages?
- Does the interaction need low latency, high throughput, or bidirectional streaming?
- Can the organization operate gateways, event-schema management, brokers, and specialized tooling?
- How will authentication, authorization, compatibility, and observability work?

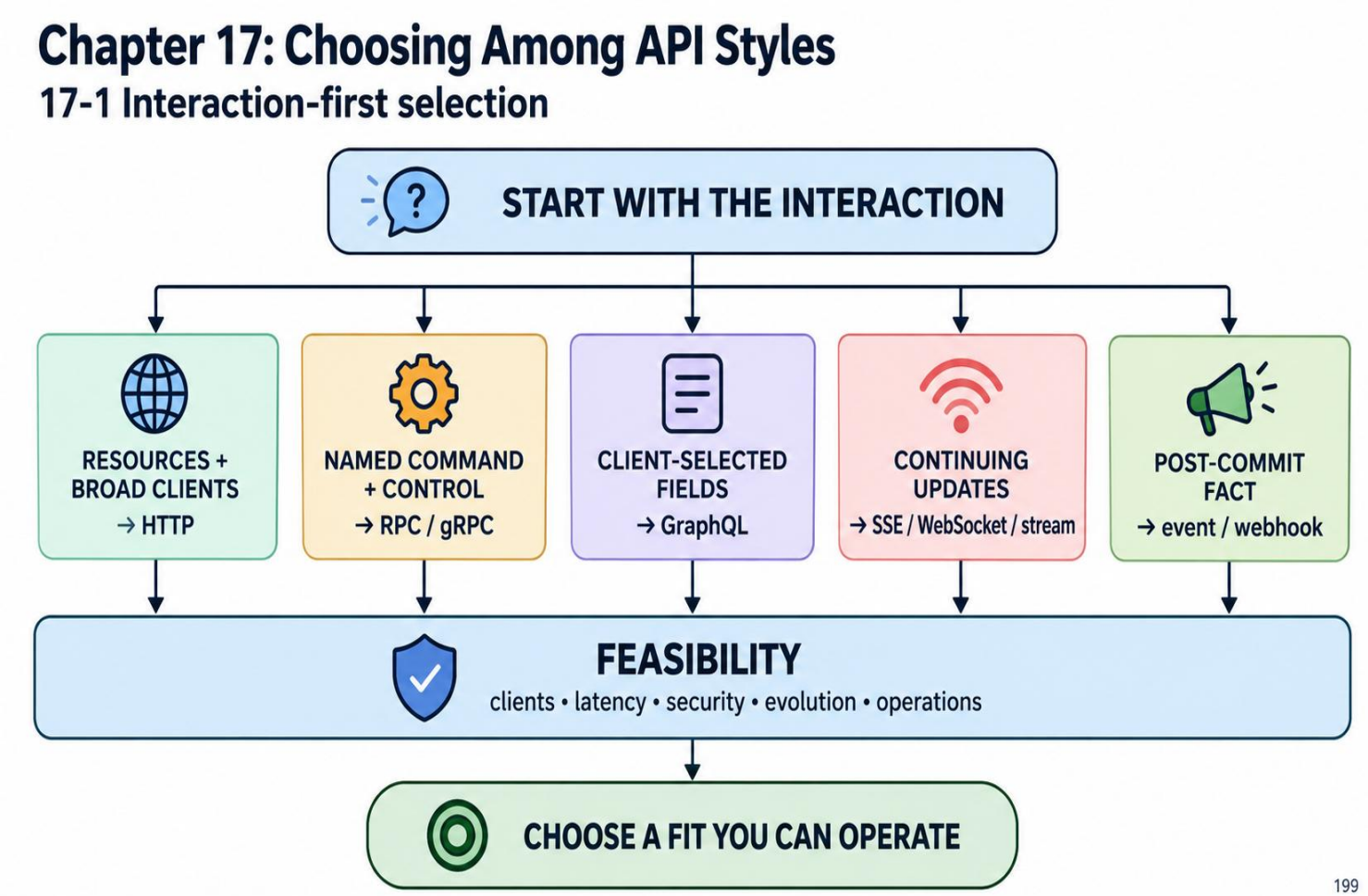



*Figure 17-1: Interaction-first API-style selection*

A familiar tool can still be a poor fit. A theoretically efficient protocol can still be a poor choice when consumers cannot use it safely or the team cannot operate it reliably.

## Resource-oriented HTTP APIs fit broad request-response use

A **resource-oriented HTTP API** models addressable domain concepts and uses HTTP methods and status semantics to operate on them. It is often called a REST-style API when it applies resource orientation and HTTP conventions, although strict REST theory includes additional constraints beyond the practical scope here.

This style fits well when:

- Many different client technologies need access.
- Browser and command-line compatibility matter.
- Resources have stable identifiers and representations.
- Standard HTTP caching, authentication, intermediaries, and diagnostics are useful.
- Human-readable JSON is acceptable.
- Public or partner consumers need approachable tooling.

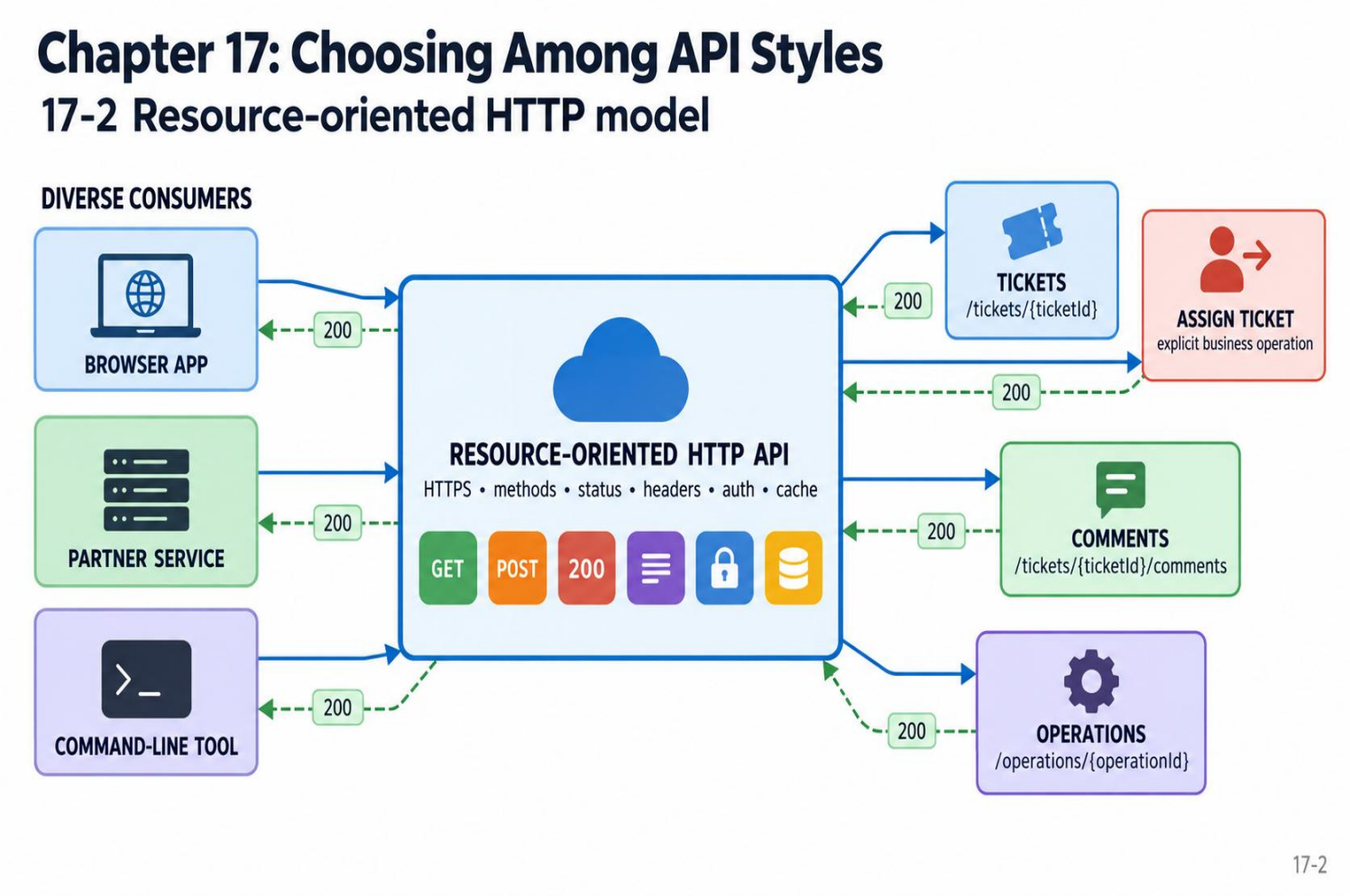



*Figure 17-2: Resource-oriented HTTP interaction model*

Its weaknesses appear when designers force every business operation into misleading CRUD, require very high-performance binary communication, or need complex streaming. Resource-oriented design also requires discipline around representation size and client round trips.

For ticket lookup, creation, comments, and bounded search, a JSON HTTP API is a natural fit because consumers understand resources and standard HTTP behavior.

## RPC exposes named operations

**Remote procedure call (RPC)** is an API style in which a client invokes a named operation on a remote system as though calling a procedure or method. The contract centers on commands such as `AssignTicket`, `CalculateRouting`, or `CloseIncident` rather than primarily on resource representations.

RPC fits when:

- The domain is naturally expressed as commands.
- Provider and consumers share controlled tooling.
- Operations need strongly specified input and output messages.
- Resource-oriented HTTP semantics would obscure the real business action.

RPC can be implemented over several transports and encodings, including JSON over HTTP. The defining feature is the operation-oriented contract, not one particular protocol.

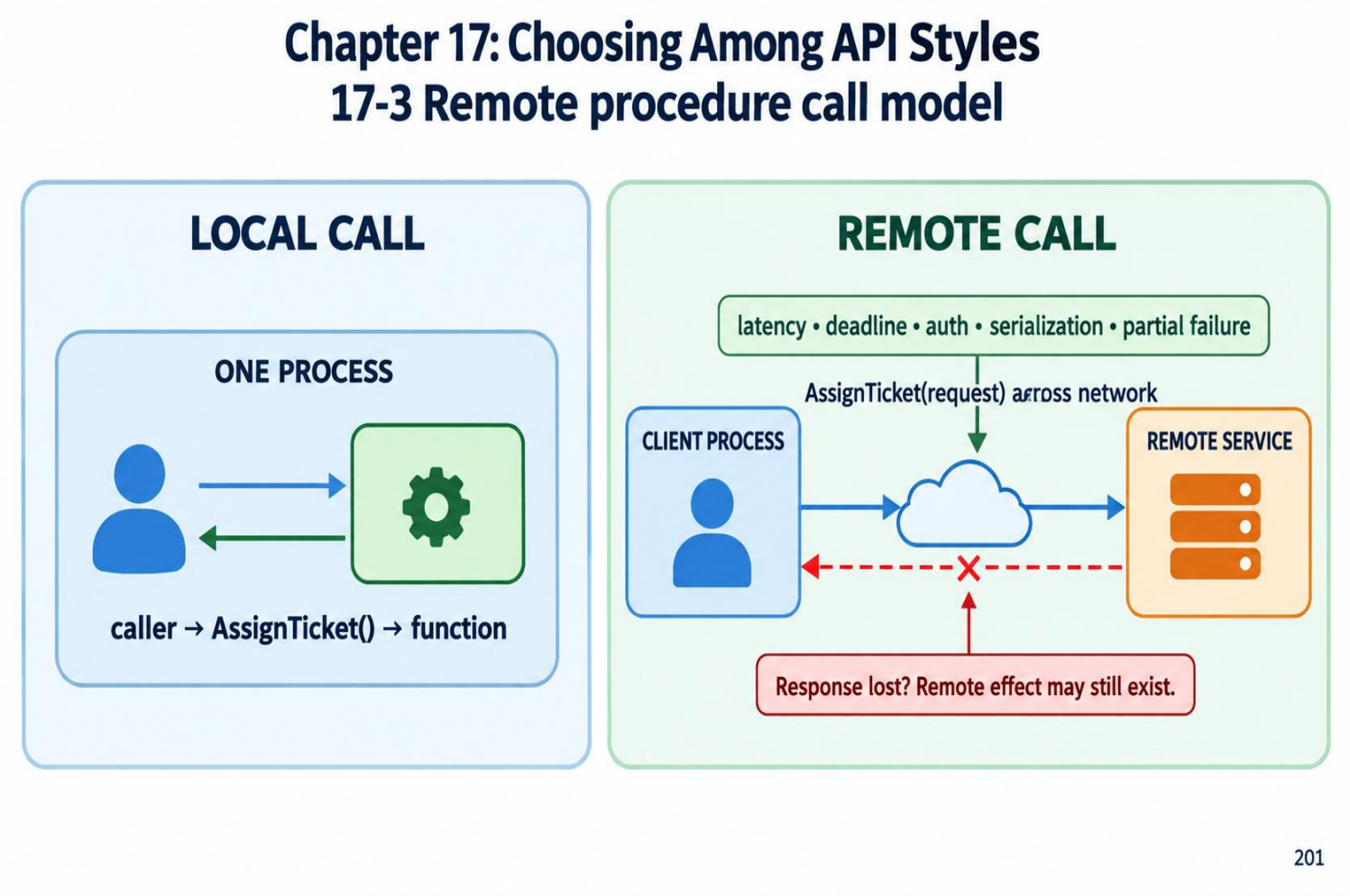



*Figure 17-3: RPC named operation across a network boundary*

The risks are method proliferation, tight coupling to service boundaries, and treating remote calls like cheap local function calls. A remote call can time out, cross a security boundary, and partially fail; RPC syntax does not remove distributed-systems behavior.

## gRPC provides strongly typed RPC and streaming

**gRPC** is an RPC framework that commonly uses Protocol Buffers to define services and messages and uses HTTP/2-based transport in typical deployments. **Protocol Buffers** is a schema language and binary serialization format that supports generated code in multiple languages.

A service definition can express unary calls, where one request receives one response, and streaming calls, where clients or servers exchange sequences of messages.

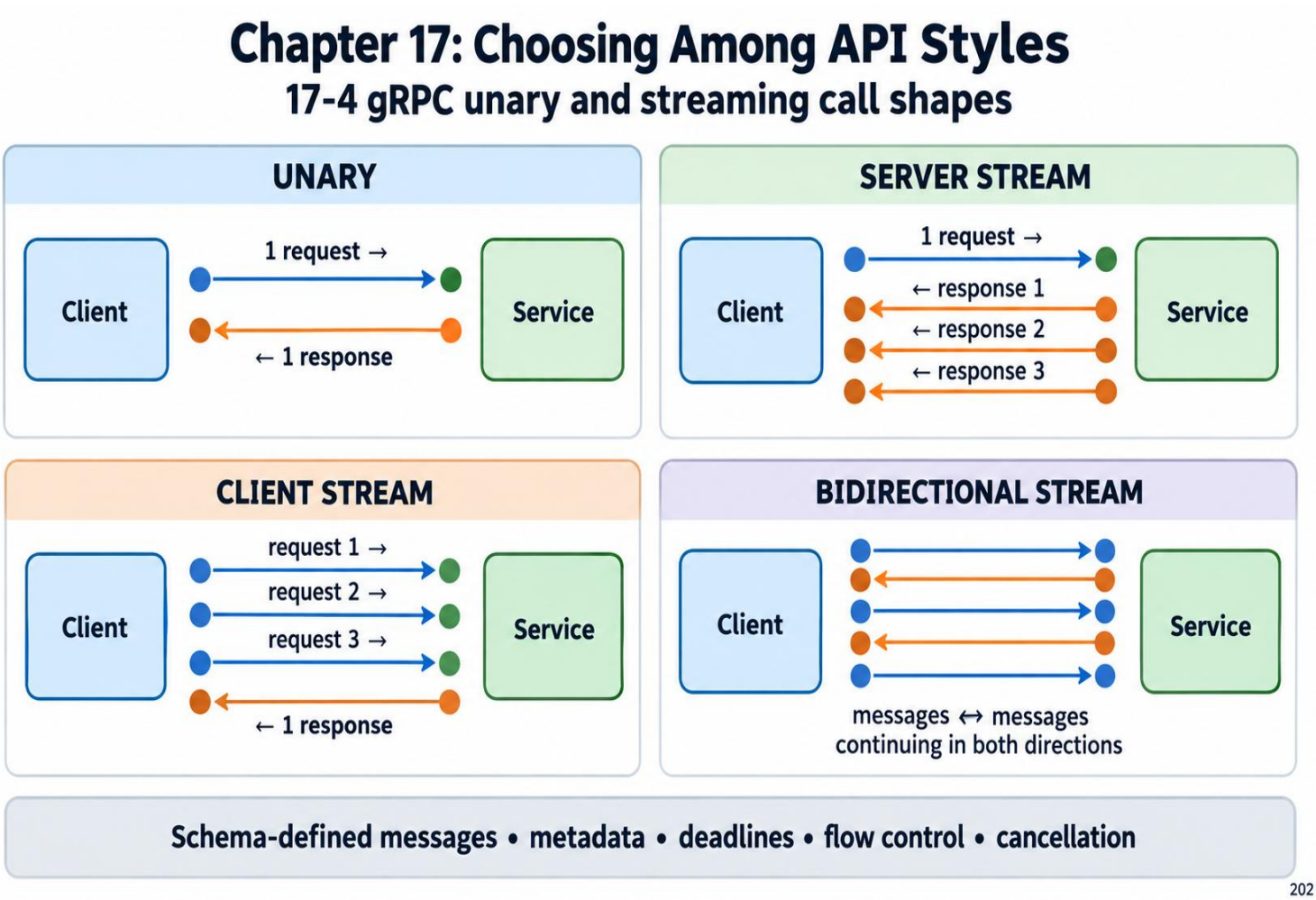



*Figure 17-4: gRPC unary and streaming call shapes*

At an overview level, gRPC offers:

- Strong schema-based messages and code generation.
- Efficient binary serialization.
- Standard support for deadlines, metadata, and streaming.
- Good fit for controlled service-to-service environments.

Its trade-offs include:

- Less direct readability in ordinary browser or command-line tools.
- Browser access that may require additional proxy or web-specific support.
- Schema-evolution rules that teams must understand.
- Generated-client and language-runtime coordination.
- Infrastructure that must correctly support the transport and observability model.

gRPC is not automatically faster for every workload. Payload size, network distance, processing, fan-out, and query behavior still dominate many systems.

## GraphQL lets clients select fields from a schema

**GraphQL** is a query-oriented API style in which clients request fields from a typed schema and receive data shaped according to the query. A schema defines object types, fields, arguments, and operations such as queries and mutations.

GraphQL fits when:

- Different clients need different combinations of related data.
- The domain can be presented as a coherent graph.
- Reducing repeated client round trips has high value.
- The organization can govern schema evolution, the performance of **resolvers** that supply field values, and field-level authorization.

A client might request a ticket, selected requester data, and assignment summary in one query rather than call several resource endpoints. This flexibility shifts responsibility to the provider, which must control query depth, cost, fan-out, authorization, and caching.

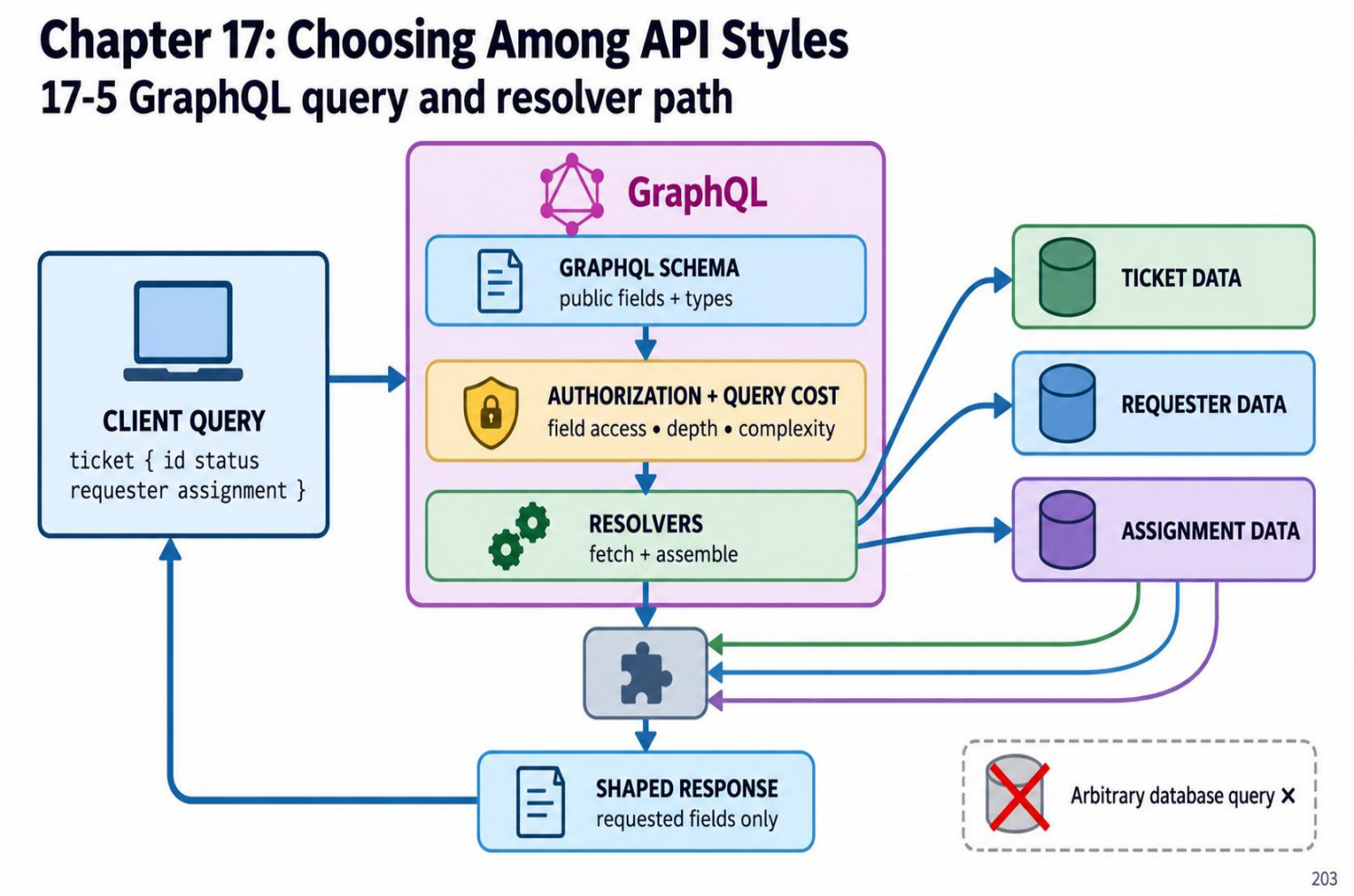



*Figure 17-5: GraphQL query through schema and resolvers to data sources*

GraphQL does not mean “any query against the database.” The schema is a designed public contract. Those resolvers must enforce authorization and avoid inefficient repeated data access.

Because many GraphQL APIs use one HTTP endpoint and return application-level errors within a response, teams need deliberate monitoring and error conventions. Standard HTTP infrastructure still matters, but operation names and field behavior often become more useful monitoring dimensions than the URL alone.

## Event-driven APIs publish facts asynchronously

An **event-driven API** allows producers to publish events and consumers to subscribe or otherwise receive them asynchronously. It fits when:

- Several consumers react independently to the same completed fact.
- The producer should not wait for every consumer.
- Temporary disconnection and later processing are acceptable.
- Eventual consistency matches the workflow.
- Replay or historical streams provide real value.

Its costs include duplicate handling, ordering limits, schema evolution, broker or delivery operations, difficult end-to-end debugging, and delayed consistency. It is not a replacement for request-response interactions that require an immediate decision.

Ticket-change notifications fit an event-driven style because assignment can complete in the ticket service before analytics, workforce, and notification consumers react.

## SOAP and WSDL remain relevant in some environments

**SOAP** is an XML-based messaging protocol with standardized conventions for envelopes, faults, and extensions. **Web Services Description Language (WSDL)** is an XML format that describes SOAP service operations and messages.

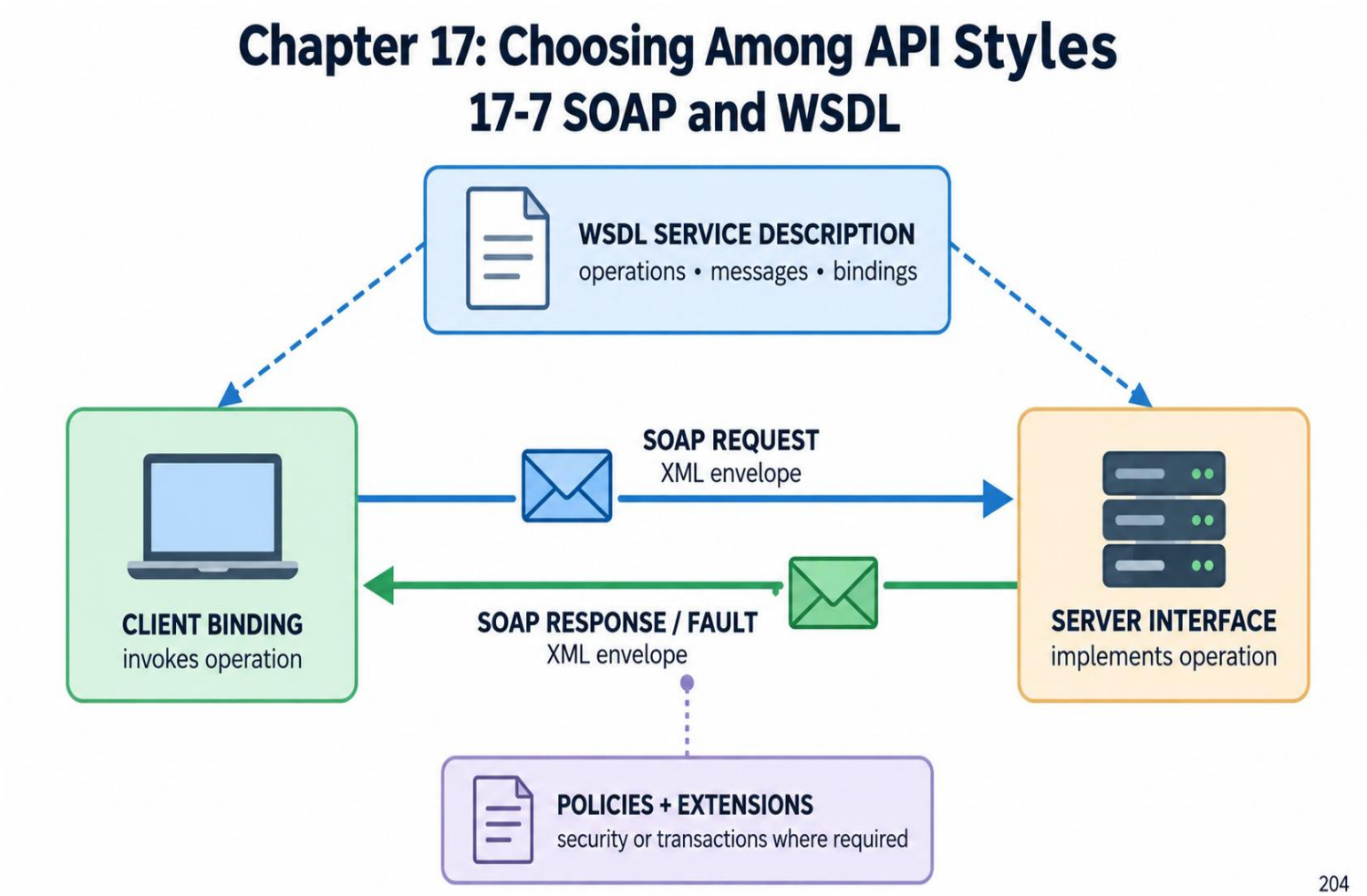



*Figure 17-6: SOAP and WSDL relationship*

Systems engineers may encounter SOAP in enterprise, financial, government, and long-lived integration environments. Its strengths can include mature tooling, explicit service descriptions, and standardized extensions for certain security and transaction needs. Its costs include verbose XML, complex specifications, and tooling expectations that differ from modern JSON HTTP APIs.

Do not dismiss a working SOAP integration merely because it is older, and do not select SOAP for a new simple API solely because existing enterprise tooling supports it. Evaluate consumer requirements, governance, and migration cost.

This guide does not cover detailed SOAP message construction or its extension specifications; recognition and comparison are sufficient for foundational API-style decisions.

## Compare styles from the same perspectives

| Style | Primary interaction model | Common strengths | Common costs | Typical fit |
|---|---|---|---|---|
| Resource-oriented HTTP | Request-response around resources | Broad interoperability, readable messages, HTTP semantics and caching | Can require several calls; poor designs force CRUD or oversized representations | Public, partner, browser, and general service APIs |
| RPC | Named remote operations | Direct expression of commands; explicit inputs and outputs | Method coupling; remote calls can be mistaken for local calls | Controlled command-oriented integrations |
| gRPC | Strongly typed RPC, including streams | Efficient binary messages, code generation, deadlines, streaming | Specialized tooling and browser/infrastructure constraints | Internal low-latency or streaming service communication |
| GraphQL | Client-selected query and mutation fields | Flexible data selection; coherent graph for varied clients | Query-cost, caching, resolver, and authorization complexity | Product surfaces with diverse data-shape needs |
| Event-driven | Asynchronous publication and subscription | Decoupled timing, multiple consumers, replay possibilities | Eventual consistency, duplicates, ordering and operational complexity | Notifications, integration facts, background reactions |
| SOAP/WSDL | Operation-oriented XML messaging | Mature enterprise contracts and extension ecosystem | Verbosity and specification/tooling complexity | Existing enterprise platforms and regulated integrations |

The table describes tendencies, not strict boundaries. GraphQL commonly uses HTTP. RPC can use JSON. Events can be delivered through webhooks. Technology layers can be combined, but their contract meanings remain different.

## Consider client diversity and control

The more diverse and independent the consumers, the more important accessible standards, stable compatibility, and low tooling barriers become.

A public API used by unknown languages benefits from conventional HTTPS, JSON, and documented HTTP behavior. A tightly controlled internal platform may gain more from generated gRPC clients. A browser application can use GraphQL effectively when the provider supplies the necessary gateway, security, and query governance.

“Internal” does not automatically mean “controlled.” If dozens of teams deploy on different schedules and use several languages, the compatibility problem resembles an external ecosystem even when all consumers belong to one organization.

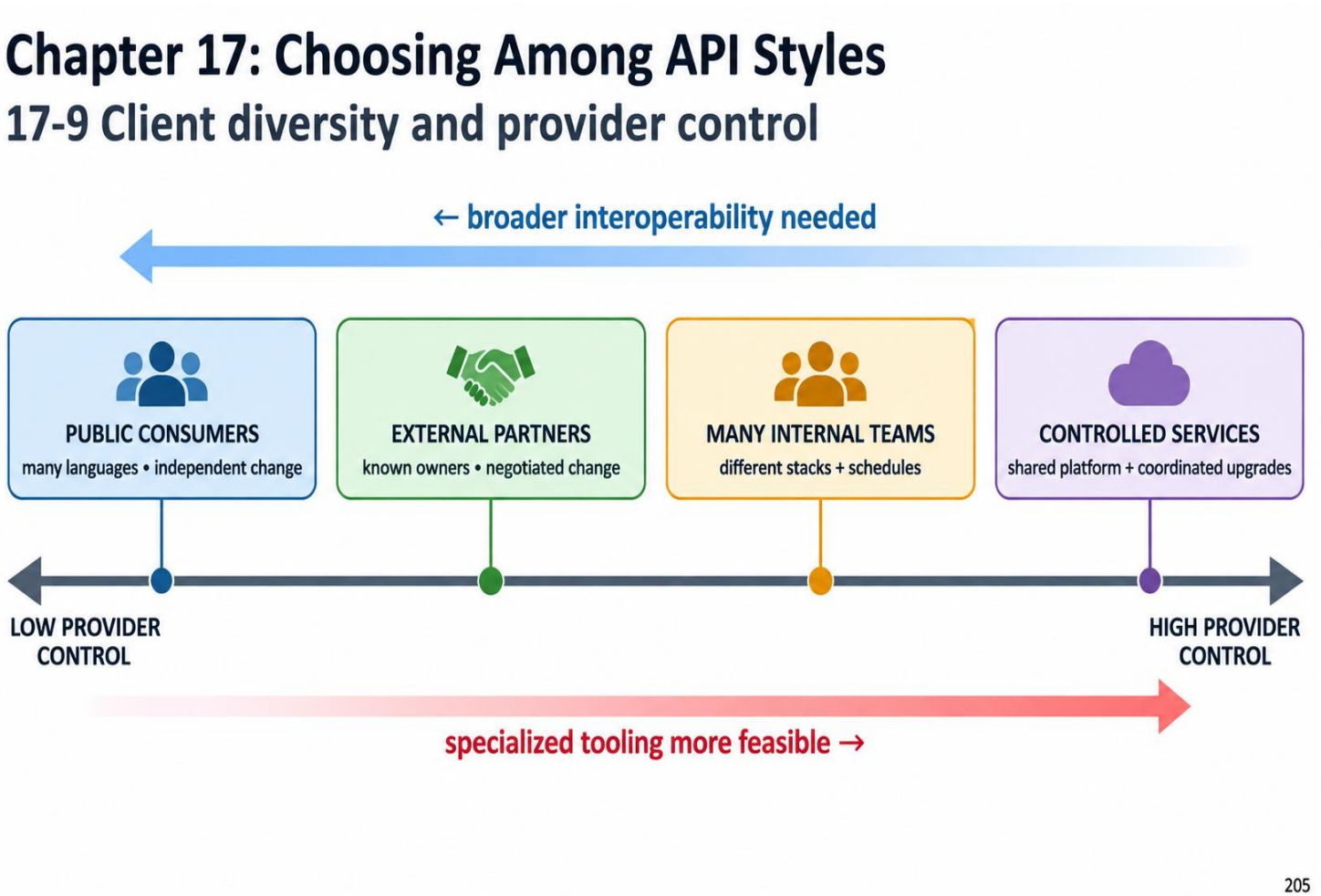



*Figure 17-7: Client diversity and provider control spectrum*

## Type systems help, but they do not define business truth

Strong schema tooling can detect missing fields, wrong scalar types, and incompatible message changes. It cannot by itself prove:

- Whether a caller may access a ticket.
- Whether assignment is valid in the current state.
- Whether a retry duplicates a business effect.
- Whether a field name communicates the right concept.
- Whether an event arrived out of order.

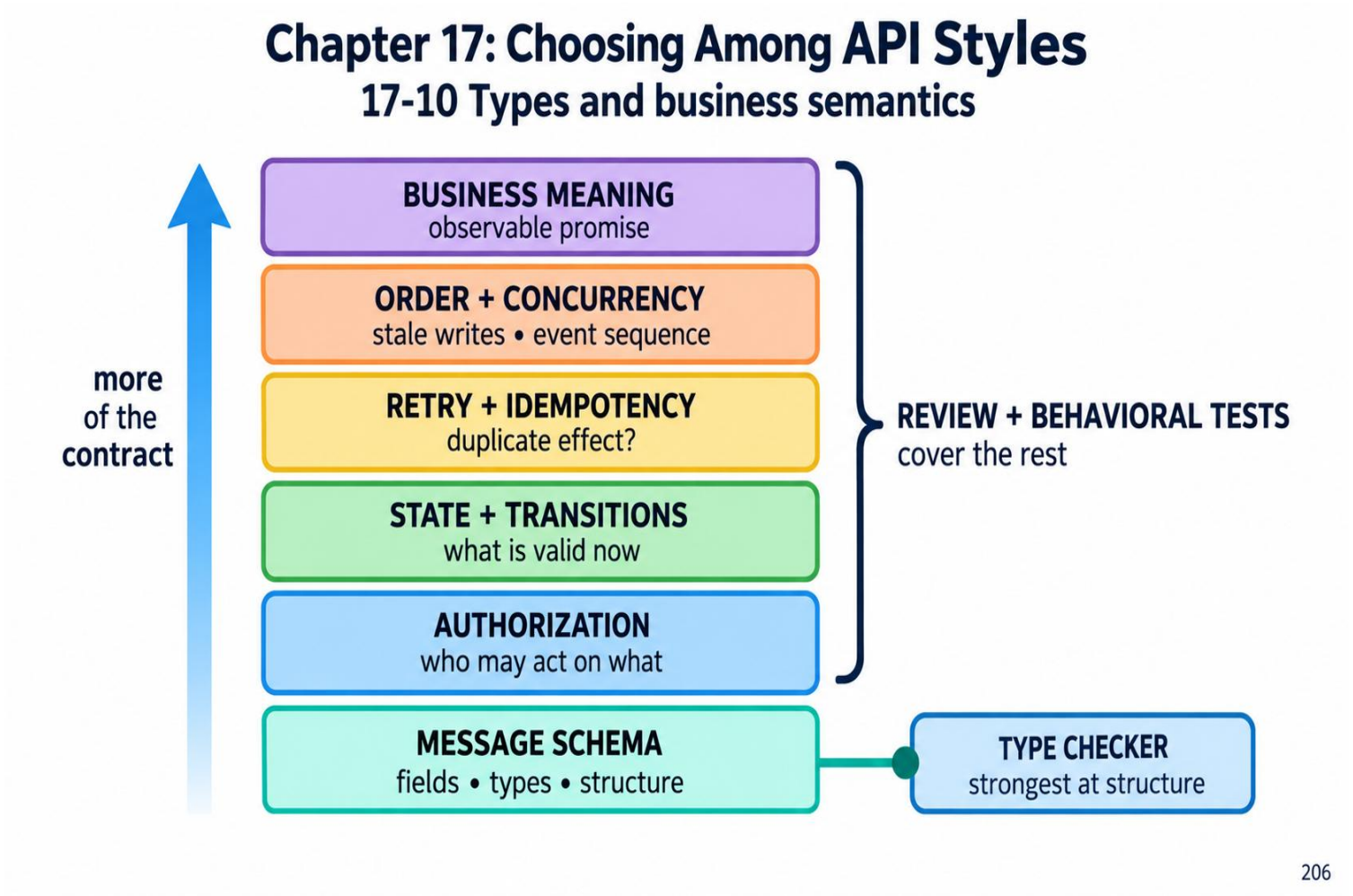



*Figure 17-8: Type system coverage versus business semantics*

JSON HTTP APIs can also have strong schemas through OpenAPI and generated types, while binary RPC contracts can still have vague semantics. Compare both structural typing and behavioral documentation.

## Streaming is a distinct requirement

**Streaming** allows a sequence of messages to be exchanged over a continuing logical interaction rather than one complete response per request. It is valuable for continuous updates, large progressive results, or bidirectional communication.

Before selecting a streaming technology, define:

- Who starts and ends the stream.
- How flow control prevents a fast sender from overwhelming a slow receiver.
- How authentication and authorization are refreshed or rechecked.
- What happens after disconnection.
- Whether messages can be resumed or replayed.
- How ordering and duplicates are handled.
- How operations are observed and limited.

A ticket lookup does not need streaming. A live agent dashboard with frequent queue updates might. **Server-sent events** provide one-way updates from a server to a browser over HTTP, while **WebSockets** provide a persistent bidirectional connection; both differ from gRPC streaming and broker subscriptions in browser support and operational behavior. Select only after the update pattern is understood.

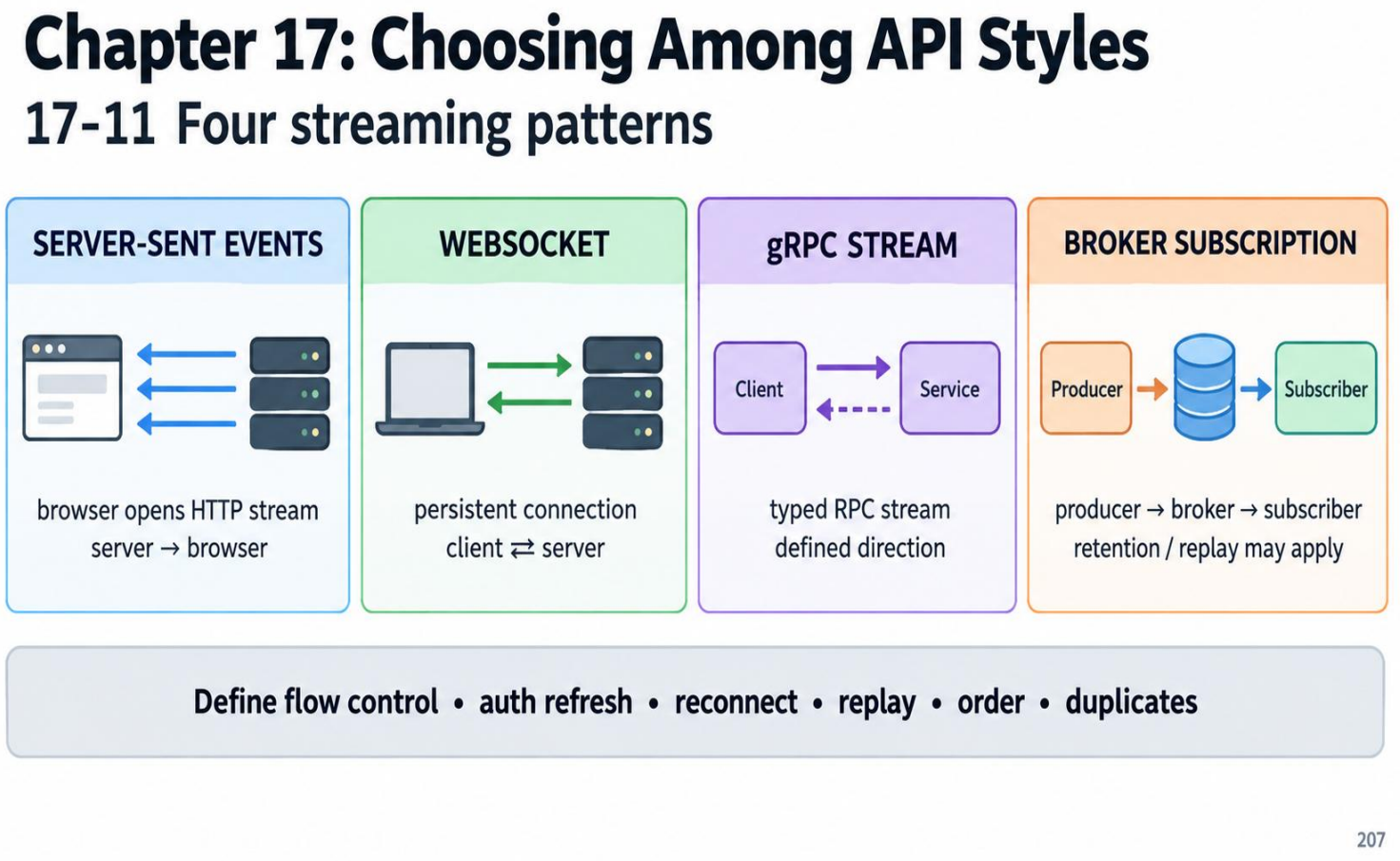



*Figure 17-9: Streaming patterns: SSE, WebSocket, gRPC stream, and broker subscription*

## Caching differs significantly by style

Resource-oriented HTTP can use standardized cache controls and validators naturally for suitable reads. GraphQL caching often needs operation-aware or entity-aware strategies because many queries share one endpoint and select different shapes. gRPC and RPC commonly rely on application or intermediary-specific caching rather than browser-style HTTP caches. Events are retained or replayed rather than cached as responses.

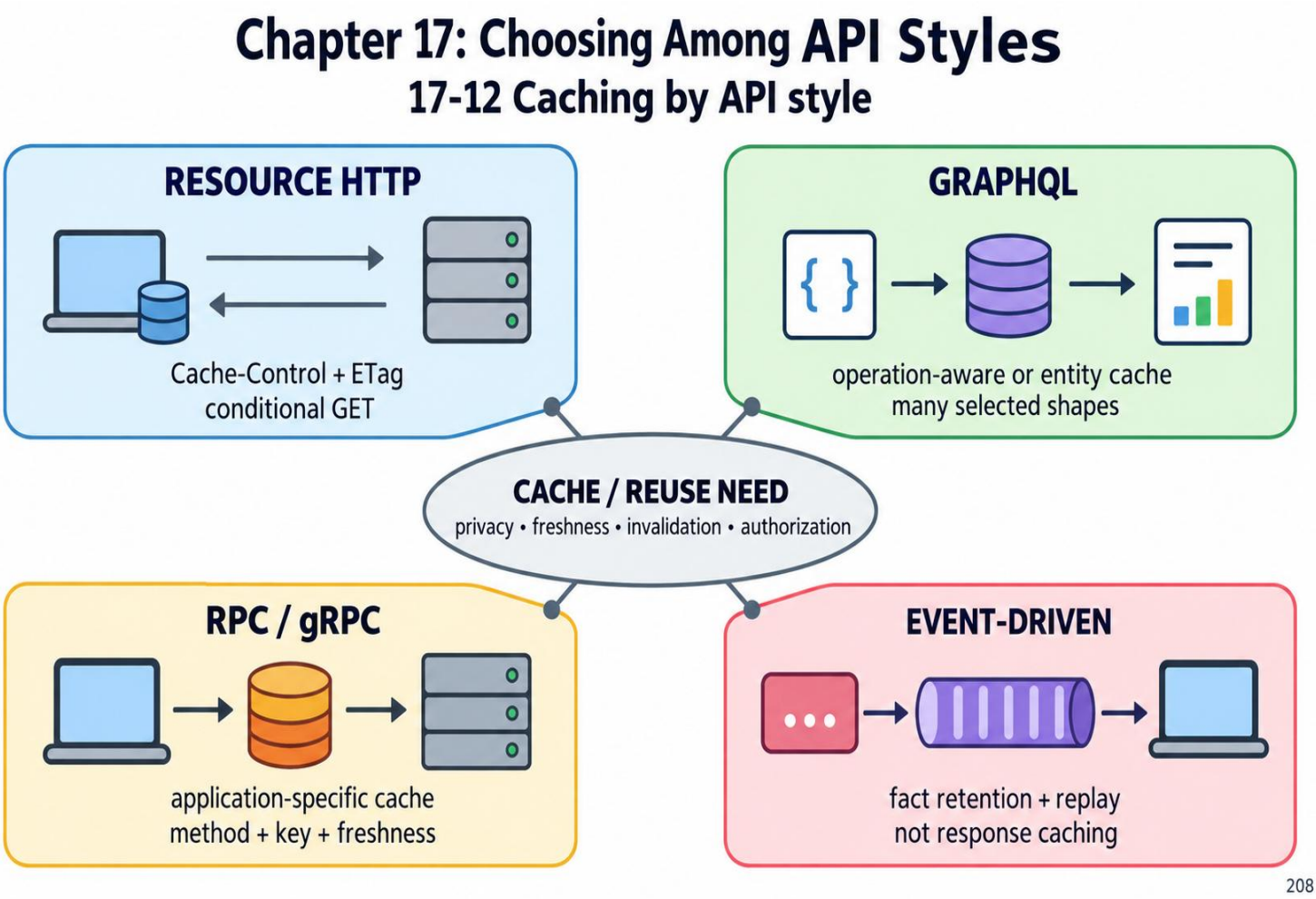



*Figure 17-10: Caching implications by API style*

If shared caching is important for public reference data, that requirement favors a style and representation that intermediaries can handle safely. If every response is user-specific and rapidly changing, standardized shared caching may carry less weight.

Caching should be one criterion, not the only criterion.

## Operational complexity belongs in the decision

Every style needs authentication, authorization, compatibility, rate control, telemetry, and incident ownership. Specialized styles add specific needs:

- gRPC needs supported proxies, client runtimes, deadline propagation, and trace integration.
- GraphQL needs query-cost controls, resolver observability, schema governance, and field-level security.
- Event-driven systems need broker or webhook operations, replay, **dead-letter handling** that isolates messages that still fail after retries, and **consumer-lag monitoring** that measures how far event processing trails publication.
- SOAP may need certificate, policy, XML schema, and enterprise middleware expertise.

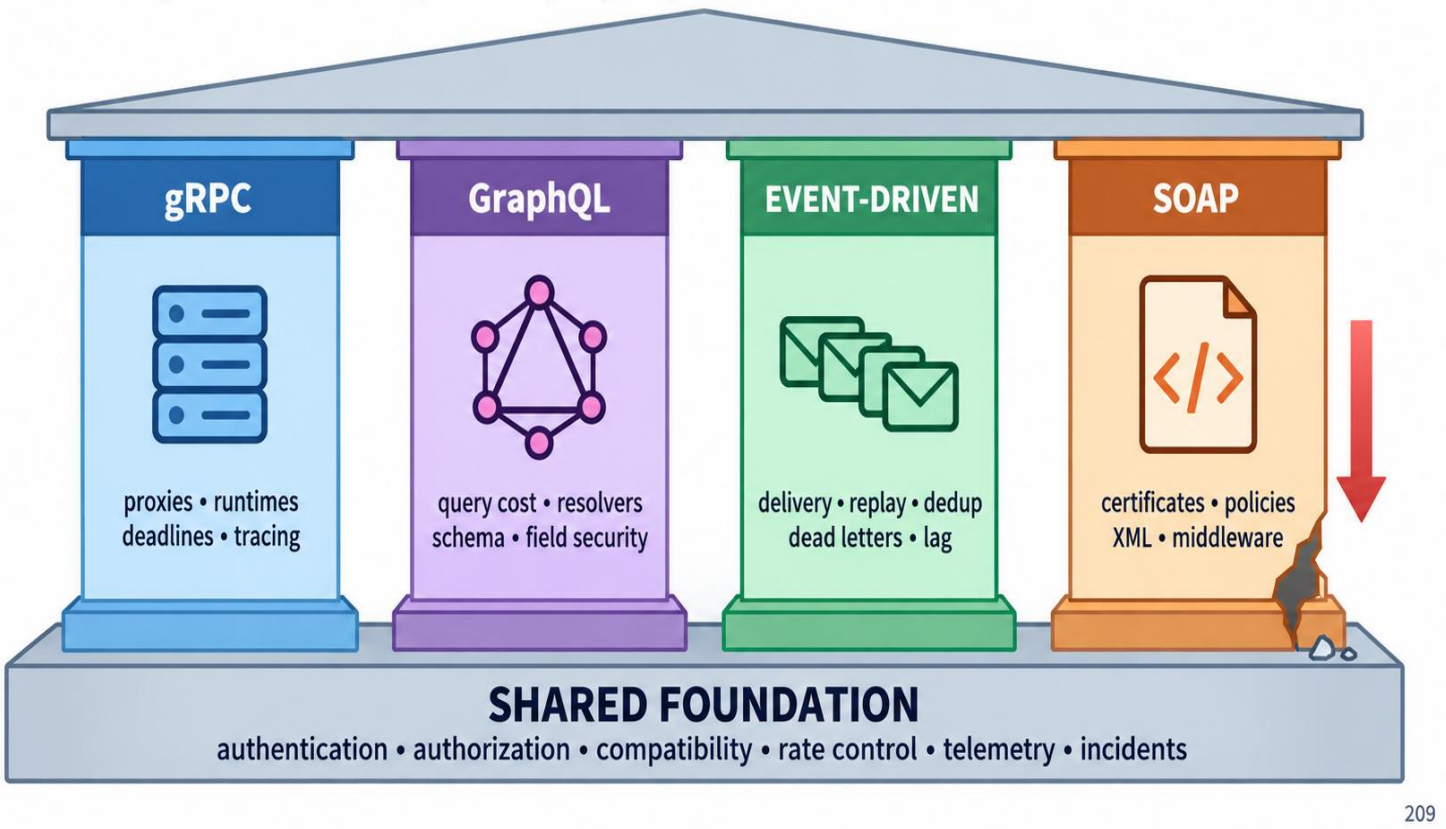



*Figure 17-11: Operational capabilities required by each style*

A design is not complete when a proof of concept sends one message. The organization must be able to operate the chosen style during overload, security incidents, schema migration, and consumer failure.

## Styles can coexist within one system

Using one style everywhere can create more complexity than deliberately combining styles. The boundary should follow use case and ownership, not fashion.

A ticket platform might use:

- Resource-oriented JSON HTTP for public and partner ticket operations.
- gRPC for a controlled, low-latency internal routing service.
- GraphQL for an internal agent workspace that combines ticket and customer views.
- Events for ticket-change notifications and analytics ingestion.

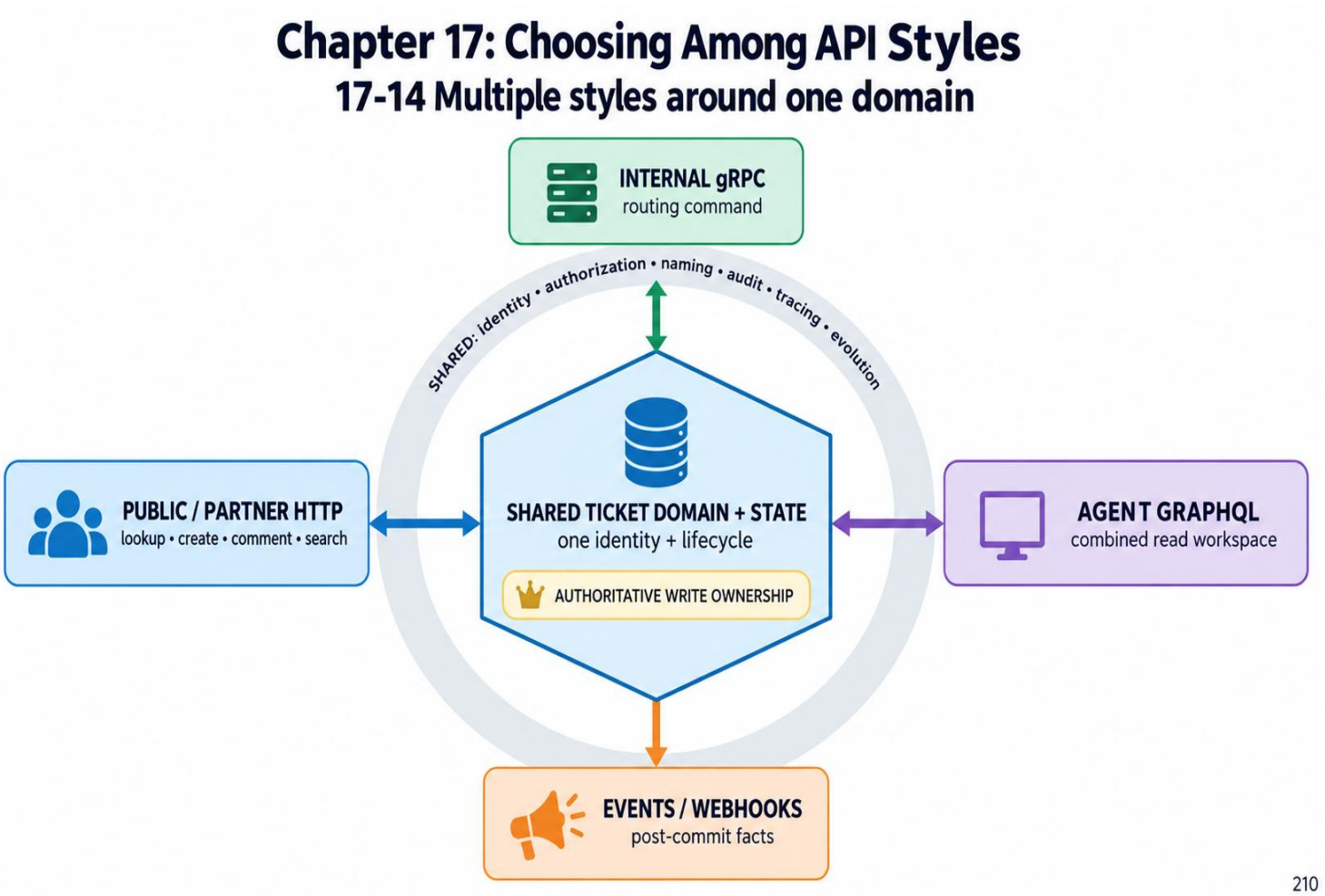



*Figure 17-12: Several API styles coexisting around one ticket domain*

Coexistence needs consistency in identity, authorization, naming, audit, tracing, and ownership. The same ticket identifier should not mean different entities in each style. Teams should avoid exposing four redundant ways to perform the same state change without a clear reason.

## Apply the criteria to four ticket needs

The following choices are plausible for the example environment, not universal recommendations.

### Public ticket lookup

A resource-oriented HTTP API is a strong fit because external consumers need broad interoperability, understandable request-response behavior, standard authorization, and straightforward diagnostics. Cache controls can apply to suitable public reference data, while protected ticket responses remain access-controlled.

### Low-latency service-to-service assignment

A gRPC operation such as `AssignTicket` can fit when both services are controlled, supported languages have mature tooling, and the infrastructure handles deadlines and telemetry. The operation is naturally command-oriented and can use generated request and response types.

A conventional HTTP assignment operation would also be valid, especially if operational simplicity and cross-language accessibility matter more than binary efficiency. The decision should use measured latency and organizational capability, not assumption.

### Flexible analytics queries

GraphQL may fit an interactive analytics or agent-analysis surface where users need varied fields and relationships. The provider must impose query-cost limits and authorize every requested field.

For large historical aggregation, a dedicated analytical store or asynchronous report API may be better than either GraphQL or the transactional ticket API. “Flexible query” does not justify running arbitrary expensive analysis against the operational service.

### Ticket-change notifications

An event-driven API fits because multiple consumers can react after the ticket change commits. The contract needs event IDs, schema compatibility, duplicate handling, and ordering scope. A webhook can deliver selected events to external consumers that operate HTTPS endpoints; an internal broker can serve controlled subscribers.

The important choice is asynchronous fact publication, not the brand of broker.

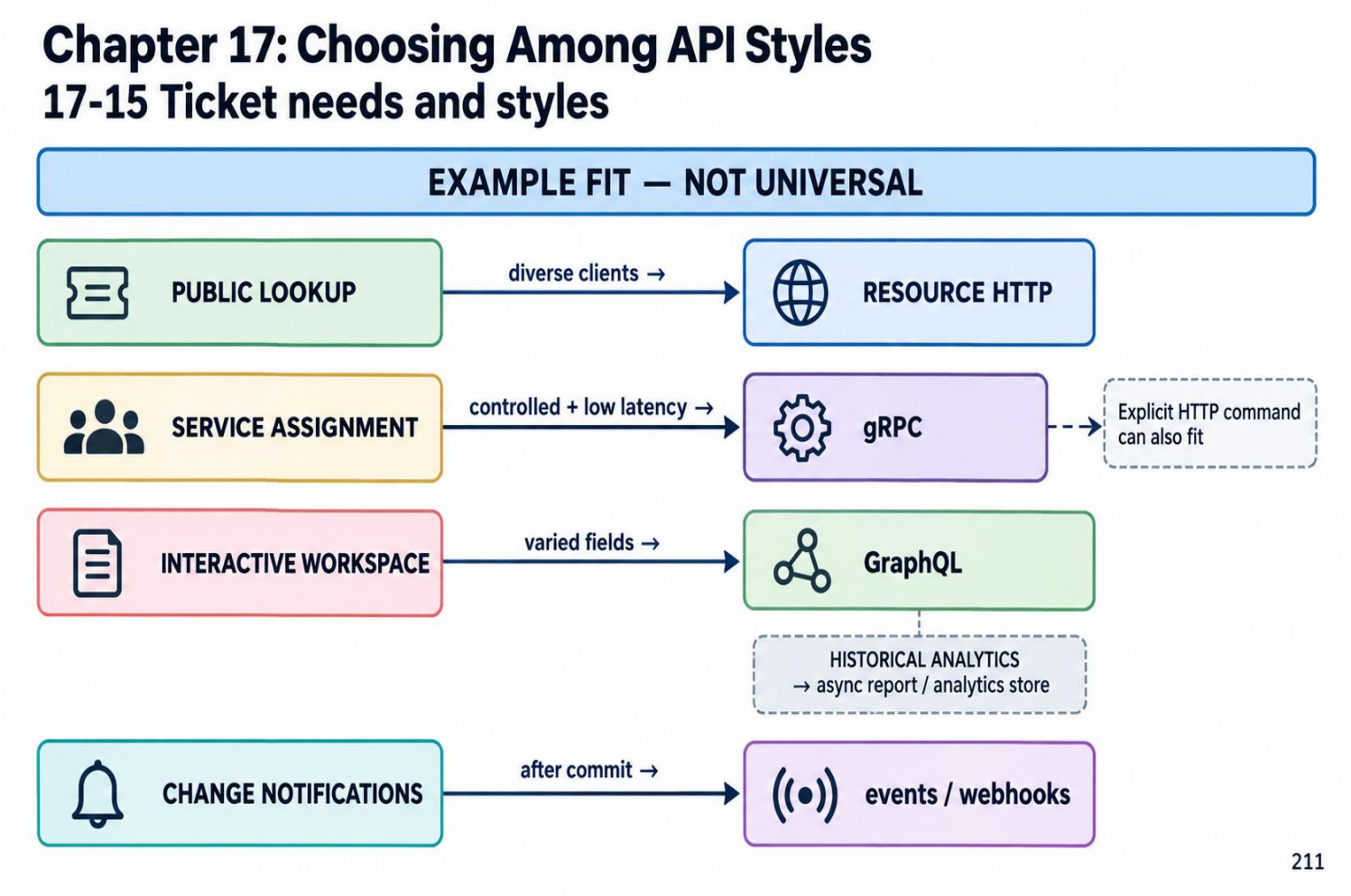



*Figure 17-13: Four ticket needs mapped to suitable interaction styles*

## Avoid technology selection by fashion or familiarity alone

A team may choose GraphQL because a previous project used it, gRPC because performance sounds important, or events because loose coupling sounds desirable. Familiarity reduces implementation risk, but it does not replace fit analysis.

Record the decision in terms of:

- Consumer types and control.
- Interaction and consistency needs.
- Message and query patterns.
- Performance evidence.
- Compatibility approach.
- Security model.
- Operational tooling and skills.
- Alternatives considered and their consequences.

A small proof of concept can validate browser support, gateway behavior, generated clients, or throughput. It should test the risky assumption rather than merely demonstrate that a “hello world” call works.

## Two style-selection misconceptions

### “REST, GraphQL, and gRPC are interchangeable transport formats”

They organize contracts differently. Resource-oriented HTTP centers on resources and HTTP semantics, GraphQL centers on a typed query graph, and gRPC centers on typed remote procedures and streams. Changing style changes client behavior, tooling, and operational concerns, not only message encoding.

### “A modern API should use one style everywhere”

Uniformity has value, but forcing every interaction into one style can produce awkward contracts. Deliberate coexistence is reasonable when each style solves a distinct need and shared security, identity, observability, and ownership remain coherent.

## API-style selection questions

Before choosing a style, answer:

- What interaction model does the consumer actually need?
- How diverse, independent, and controlled are the clients?
- Are resources, commands, flexible queries, streams, or events the central concept?
- Which latency, throughput, and payload characteristics are measured requirements?
- How will the style handle authentication and resource-level authorization?
- What compatibility rules and tooling will consumers use?
- How will failures, retries, deadlines, and rate limits appear?
- Can the organization monitor and operate the required infrastructure?
- Does the choice duplicate another interface without clear value?
- Which assumption should be validated with a focused prototype?

**Choose the style that makes the required interaction clear and sustainable for its consumers and operators. A fashionable protocol cannot compensate for a mismatched communication model or an unsupported operational burden.**

## Chapter Summary

- API style should follow the interaction pattern and operating environment rather than fashion or team familiarity alone.
- Strong type systems improve structural clarity but do not define authorization, business invariants, or compatibility policy.
- Different interactions in one system may appropriately use different styles while sharing domain meaning and governance.

## Review Questions

1. Which requirements make streaming a first-class design concern?
2. Why are REST-style HTTP, GraphQL, and gRPC not interchangeable transport formats?
3. What operational capabilities must be considered before adopting an event-driven interface?

# Chapter 18 — Applying the Design Process on a Team

**Effective API design is a repeatable team process that connects consumer needs, contract decisions, verification, release, operation, and retirement.** The contract is not finished when implementation begins; it remains a managed product boundary for as long as consumers depend on it.

## What You'll Be Able to Do

After completing this chapter, you will be able to:

- Apply the full API design lifecycle from discovery through retirement within a team.
- Connect ownership, examples, formal contracts, reviews, tests, release plans, catalogs, governance, and operational readiness.
- Use the complete support-ticket case study and final design-review model to evaluate a practical API.

## Keep the work connected from discovery to retirement

Weak API processes produce isolated artifacts: requirements in one document, an OpenAPI file generated later, tests based on the code, and operational behavior discovered after release. A strong process keeps these views connected.

A practical sequence is:

1. Establish ownership, scope, consumers, and desired outcomes.
2. Write concrete use cases and example interactions.
3. Model resources, operations, data, errors, permissions, and lifecycle behavior.
4. Examine failure, concurrency, compatibility, performance, and security boundaries.
5. Record significant decisions and rejected alternatives.
6. Express the structured HTTP contract in OpenAPI where applicable.
7. Review with consumer, domain, security, privacy, and operations perspectives.
8. Implement and verify the contract through layered tests.
9. Release with telemetry, runbooks, migration information, and rollback planning.
10. Observe real use, evolve compatibly, deprecate deliberately, and retire responsibly.

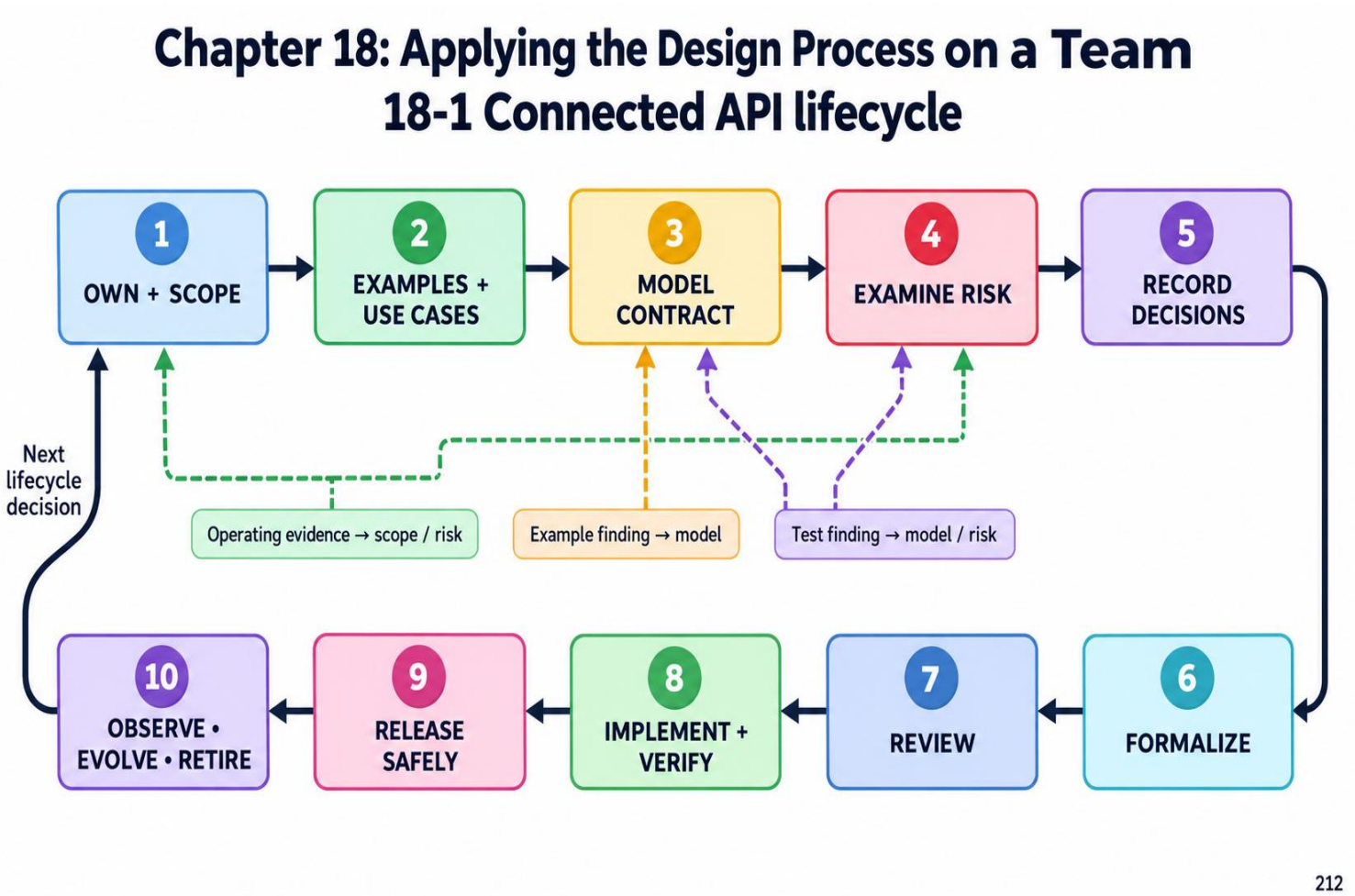



*Figure 18-1: Connected API lifecycle from discovery to retirement*

The steps can overlap. Example messages may reveal a missing domain rule, and a test may reveal that an error contract is incomplete. Iteration is expected. What matters is that a change in one view is reflected in the others.

## Assign clear ownership

An API needs an accountable owner who can make and maintain contract decisions. Ownership does not mean that one person decides everything; it means that consumers know which team is responsible for behavior, support, compatibility, and lifecycle.

Useful responsibilities include:

- **API owner:** Accountable for the contract, roadmap, compatibility, and consumer communication.
- **Domain representative:** Confirms business language, rules, and lifecycle behavior.
- **Consumer representative:** Verifies that real tasks can be completed safely and efficiently.
- **Security and privacy reviewers:** Examine identity, authorization, data exposure, retention, and abuse cases.
- **Operations representative:** Reviews SLOs, limits, telemetry, failure behavior, and runbooks.
- **Implementation team:** Confirms feasibility and implements the agreed behavior without allowing internal structure to define it accidentally.

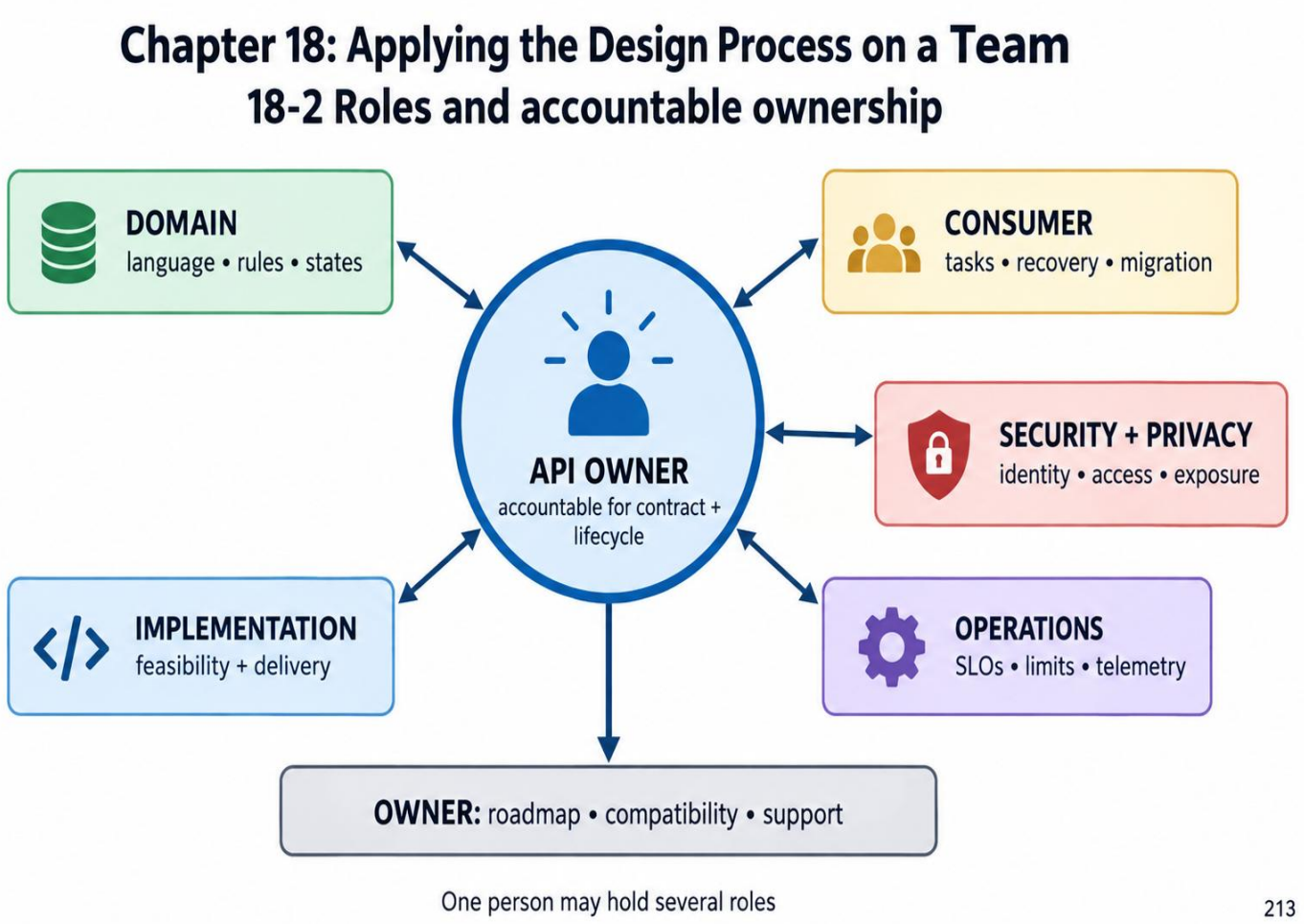



*Figure 18-2: Team roles and accountable API ownership*

One person can hold several roles in a small project. The important point is that the perspectives are considered and the final owner is not ambiguous.

Ownership metadata should remain current after release. An abandoned API with no reachable owner is difficult to patch, migrate, or retire safely.

## Define outcomes and constraints before endpoint details

A concise design brief should answer:

- Which consumer problem is being solved?
- Which consumers and caller types are in scope?
- Which business outcomes define success?
- Which capabilities are explicitly out of scope for the first release?
- What data sensitivity and tenant boundaries apply?
- What traffic, latency, availability, and retention expectations matter?
- Which existing systems and organizational standards constrain the design?
- How will adoption and service quality be measured?

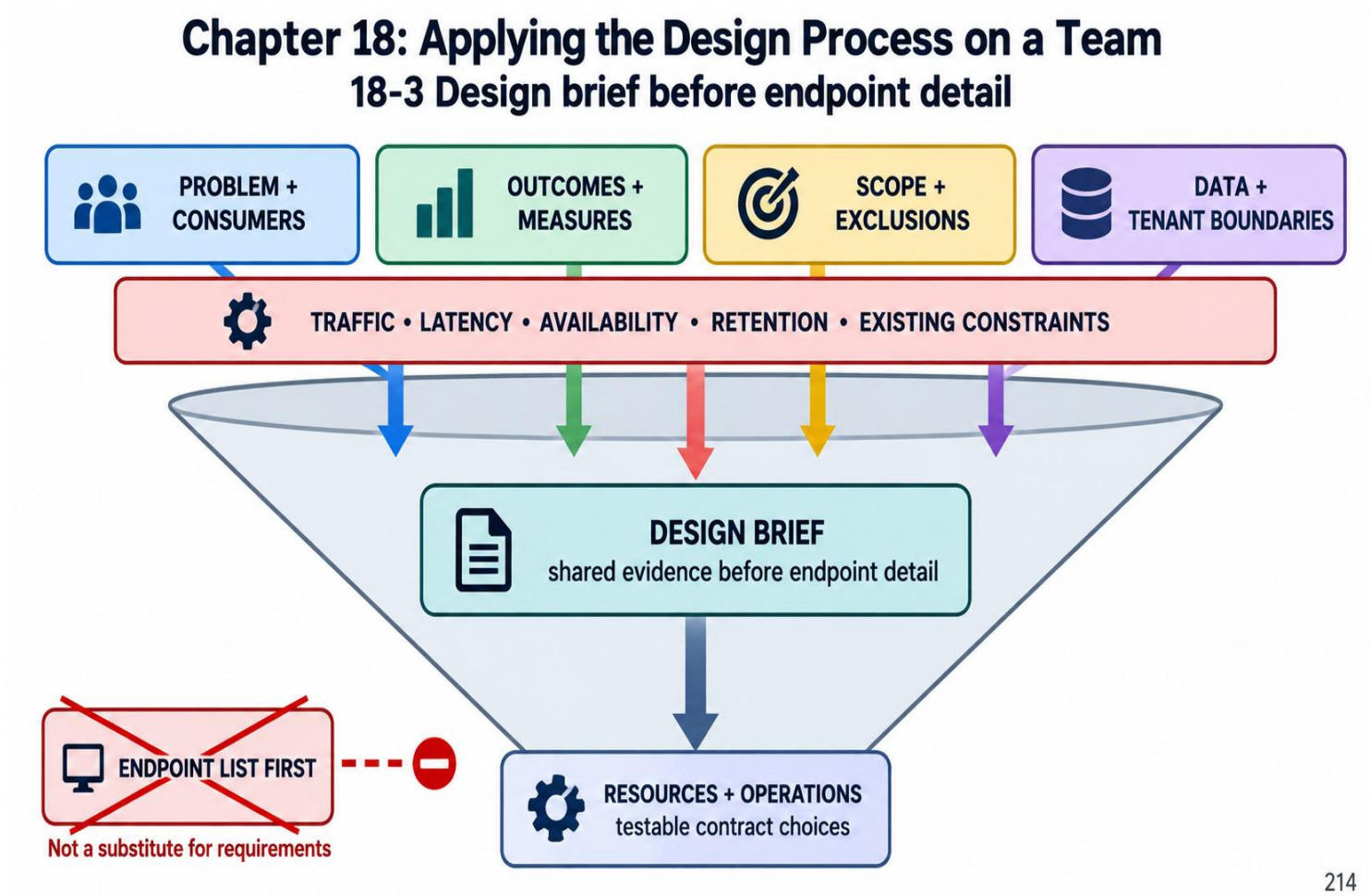



*Figure 18-3: Design brief before endpoint detail*

This prevents an endpoint list from becoming a substitute for requirements. “Build a ticket API” is not enough. “Allow requesters to submit and track their own issues, allow authorized agents to assign and resolve tickets, and allow automation to retry creation safely” leads to testable design decisions.

Constraints should be stated as evidence or current project requirements, not universal truths. A browser-based agent application creates different needs from a controlled batch integration. A multi-tenant service creates authorization obligations that a single-user local tool may not have.

## Use example interactions before completing the schema

Examples force abstract decisions into observable form. Before building a complete description, write representative interactions for:

- A successful operation.
- A validation failure.
- A permission denial.
- A state conflict.
- A retry after uncertain completion.
- A collection page and continuation.
- An asynchronous acceptance and terminal result where applicable.

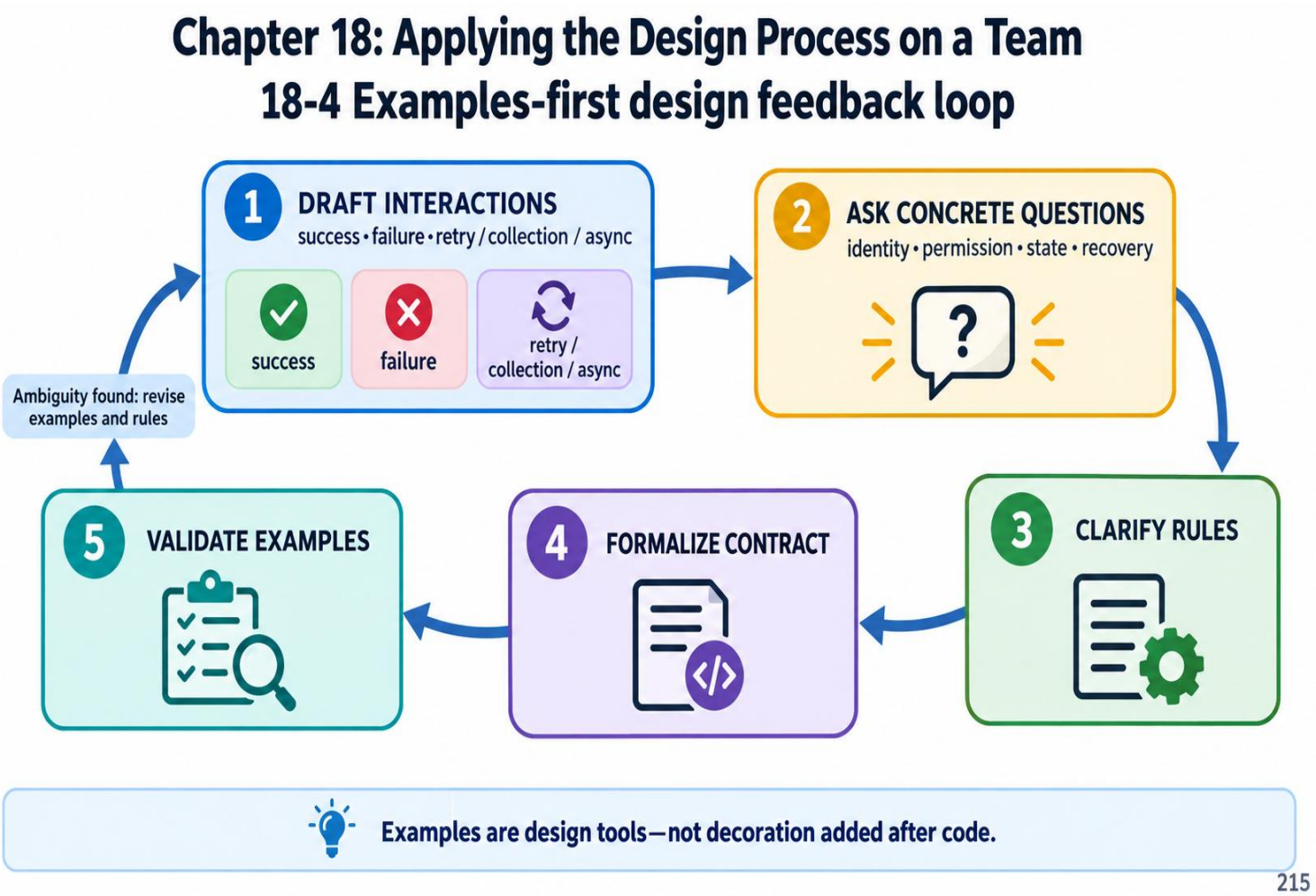



*Figure 18-4: Examples-first design feedback loop*

Reviewers can then ask concrete questions: Is the identifier stable? Who supplies the status? Can the requester see this field? Does the retry create another ticket? How does the client recover from the conflict?

Examples are design tools, not merely documentation added after code. Once the design is agreed, validate the examples against the formal schemas so that they remain trustworthy.

## Produce a small set of connected artifacts

A practical API design package does not need dozens of documents. It should include enough material to preserve important decisions:

- A brief statement of purpose, ownership, consumers, and scope.
- Use cases and representative request-response examples.
- The machine-readable contract where suitable.
- Permission and lifecycle rules that the schema cannot express.
- A small set of decision records for consequential trade-offs.
- A test matrix derived from the contract.
- Release, migration, and deprecation information.
- Operational ownership, dashboards, alerts, and runbooks.

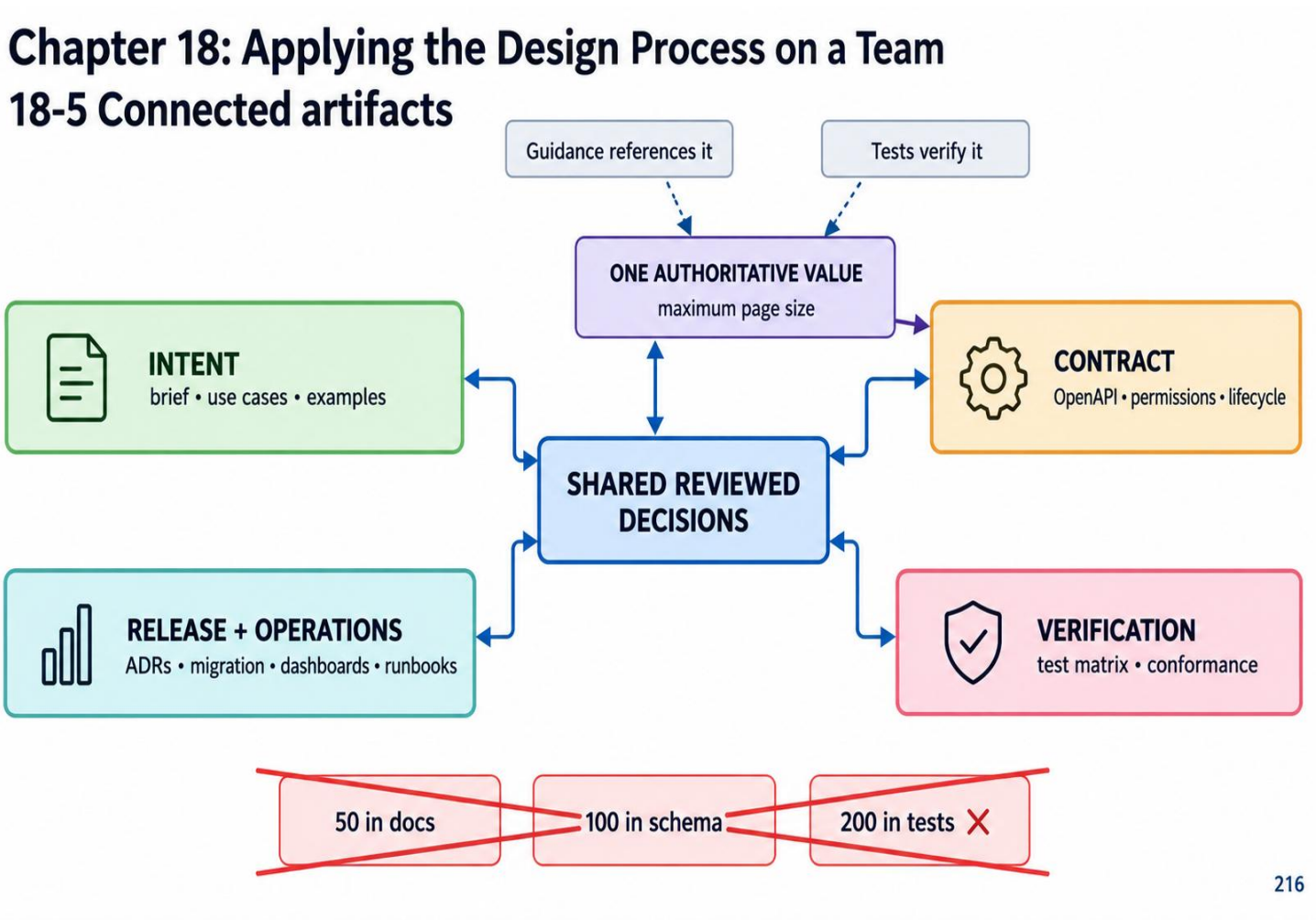



*Figure 18-5: Connected API design artifacts*

These can live together in one repository or managed documentation system. The format matters less than discoverability, review history, and alignment.

Avoid duplicating the same rule in many places with no source of authority. For example, define the maximum page size in the contract and reference it from user guidance and tests rather than manually maintaining three unrelated values.

## Record significant decisions and alternatives

An **architecture decision record (ADR)** is a short record of an important design choice, its context, considered alternatives, and consequences. It preserves reasoning that may otherwise disappear after a meeting.

A useful ADR has four parts:

- **Context:** The problem, requirements, and constraints.
- **Decision:** The chosen approach.
- **Alternatives:** Serious options considered and why they were not selected.
- **Consequences:** Benefits, costs, risks, and follow-up obligations.

For cursor pagination, an ADR might state:

- Context: Ticket data changes frequently, and the automation service processes large result sets.
- Decision: Use opaque cursor pagination ordered by `createdAt` and `id`.
- Alternatives: Offset pagination was simpler but produced unstable continuation under frequent inserts.
- Consequences: The provider must encode and validate cursors, consumers cannot jump directly to page numbers, and compatibility tests must protect cursor semantics.

The record should explain the decision, not reproduce an entire chapter on pagination.

Record decisions that affect multiple teams, long-term compatibility, security, or operational cost. Do not create an ADR for every field name.

## Review with the people who bear the consequences

A design review should include enough perspectives to expose missing assumptions without turning every minor change into a large committee meeting.

Different reviewers ask different questions:

| Perspective | Questions it should answer |
|---|---|
| Consumer | Can the task be completed? Are recovery and migration practical? |
| Domain | Do names, states, and transitions represent the business correctly? |
| Security and privacy | Who can access which data? Are tenant, secret, retention, and abuse risks controlled? |
| Operations | Are limits, timeouts, telemetry, failure modes, and ownership defined? |
| Implementation | Can the provider meet the promises atomically and sustainably? |
| Compatibility | Which released expectations could the change affect? |

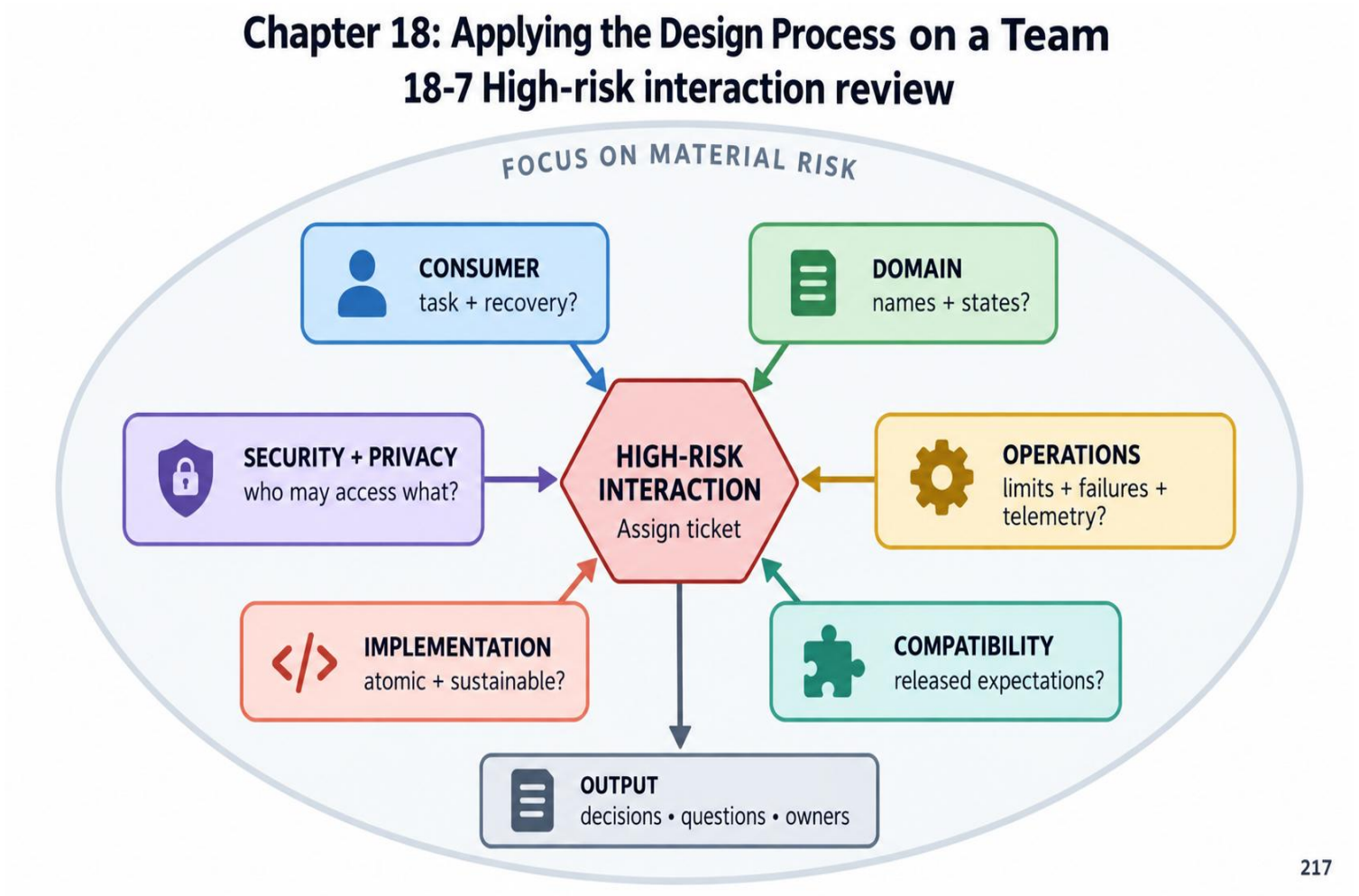



*Figure 18-6: Review perspectives around a high-risk interaction*

Review the high-risk interactions, not only the full schema diff. A one-line type change can be more dangerous than adding a new operation. A permission change may have no OpenAPI diff at all.

Record unresolved questions, owners, and decisions. A meeting that ends with “looks good” but no agreed contract or follow-up does not reduce risk.

## Turn review comments into contract consequences

Good feedback states why a concern matters. Compare:

- Weak: “Use `PUT` instead of `POST`.”

- Strong: “The caller does not know a stable assignment-resource URI before creation, and repeated submissions need one logical result; `POST` with an idempotency key communicates that behavior more accurately.”
- Weak: “This response is too large.”
- Strong: “The collection returns full descriptions and attachment metadata for up to 500 tickets, which conflicts with the interactive latency target and exposes fields the list view does not use.”
- Weak: “Add more security.”
- Strong: “The operation checks the `tickets:read` scope but does not verify that the ticket belongs to the caller’s tenant.”

This style makes review educational and actionable. It also helps the owner decide when an alternative design is equally valid.

## Formalize the contract after the core behavior is understood

For an HTTP API, create or update the OpenAPI description once the use cases, examples, and important behaviors are coherent. The description should not be delayed until release, but neither should schema construction replace domain design.

The formal contract should connect to:

- Generated or validated examples.
- Provider conformance tests.
- Compatibility checks against the previous release.
- Reference documentation.
- Selected client or server scaffolding where useful.

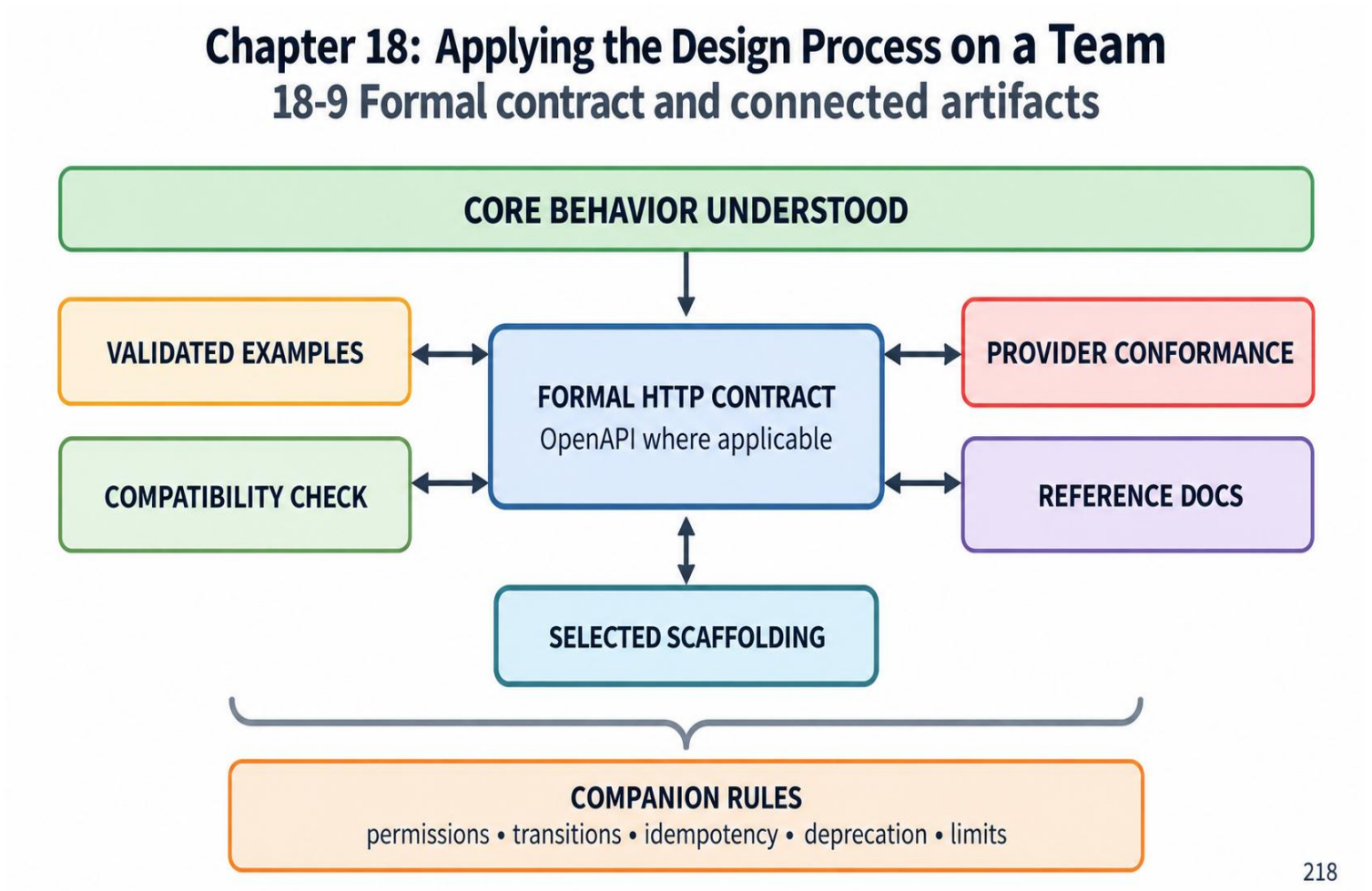



*Figure 18-7: Formal contract connected to examples, tests, compatibility, and documentation*

Behavior outside the schema—permissions, transition rules, idempotency matching, deprecation policy, and operational limits—must remain reviewable and testable through accompanying definitions.

## Plan release before implementation is complete

A release plan answers more than when code will deploy. It includes:

- Which consumers receive the capability first.
- How compatibility with existing consumers is verified.
- Whether a feature can be enabled gradually.
- Which metrics and errors will indicate trouble.
- How the provider can stop or roll back exposure safely.
- Whether new stored data prevents a full rollback.
- Which documentation, examples, and migration notes publish with the release.
- Who owns support and incident response.

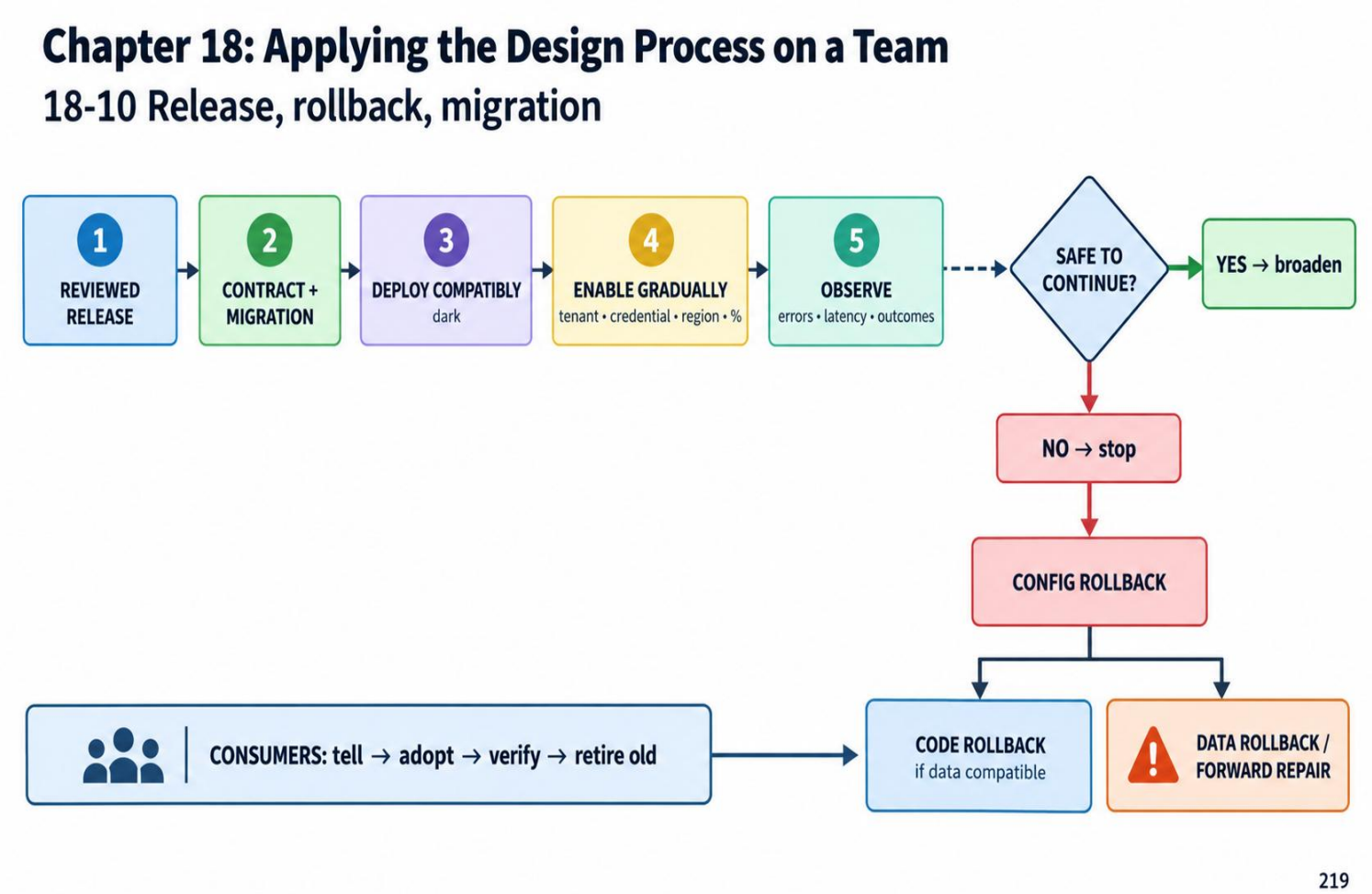



*Figure 18-8: Release plan with rollout, rollback, and migration branches*

Gradual rollout reduces impact only when the rollout unit is meaningful. Enabling by tenant, credential, region, or percentage has different consequences. A **feature flag** is a controlled switch that enables behavior for selected traffic without a separate code deployment; authorization and contract behavior must not become inconsistent merely because such a flag is misapplied.

A rollback plan should distinguish code rollback, configuration rollback, and data rollback. After the provider accepts a new event value or data shape, reverting code may not restore compatibility automatically.

## Style guides create defaults, not automatic answers

An **API style guide** records organization-wide conventions such as naming, error shape, pagination parameters, authentication declarations, identifier formats, and deprecation practices. It reduces needless variation and lets reviewers focus on domain-specific decisions.

A useful style guide:

- Explains the reason behind important rules.
- Distinguishes requirements from recommendations.
- Shows small valid examples.
- Defines an exception process.
- Evolves based on real project evidence.
- Is checked automatically where rules are structural.

A style guide should not force a misleading design merely for uniformity. An explicit action endpoint can be appropriate even if ordinary updates use resource paths. An exception should record why the normal rule does not fit and how consistency will still be protected.

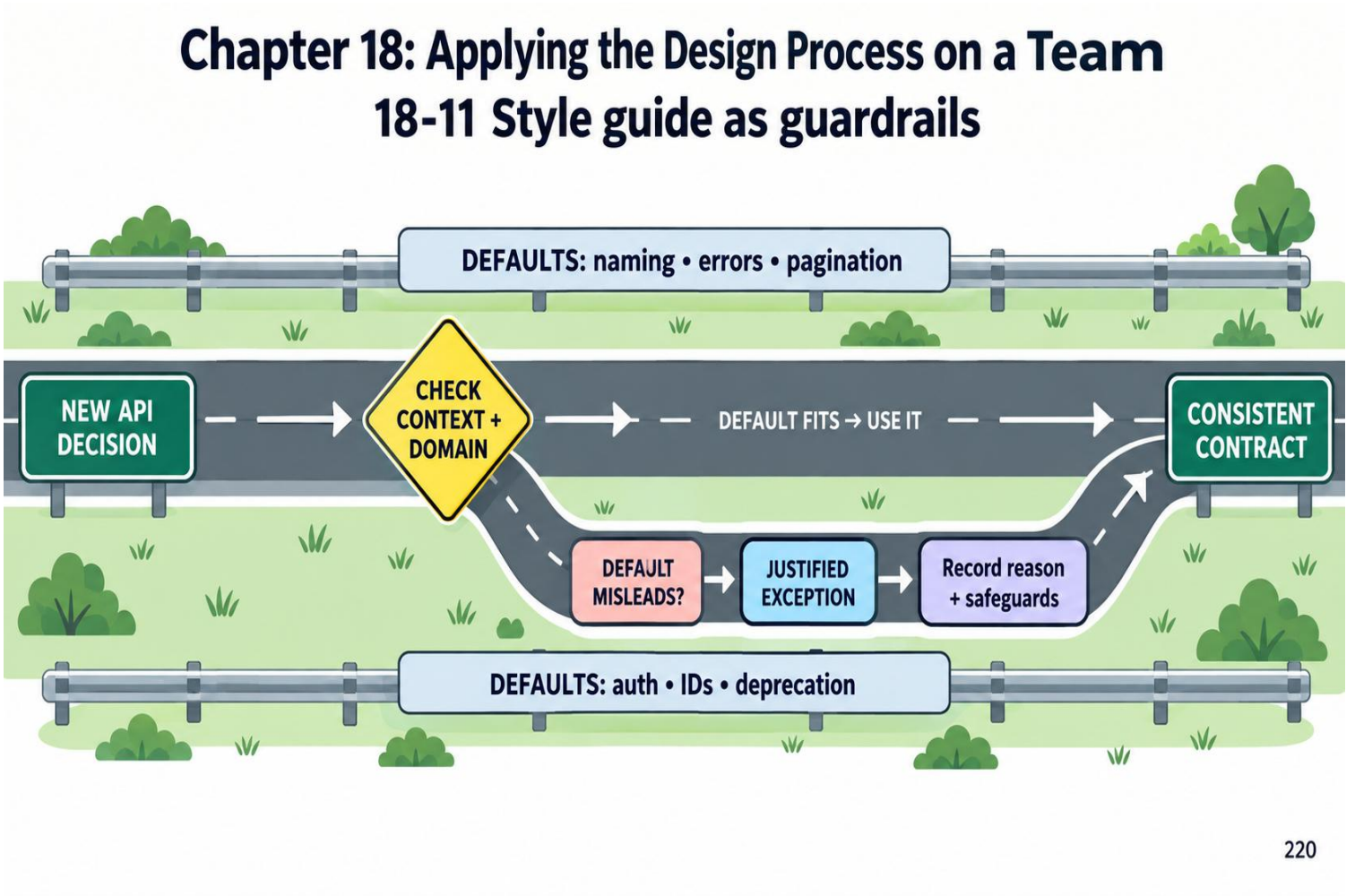



*Figure 18-9: Style guide as guardrails, not an automatic answer*

## Catalogs make APIs discoverable and accountable

An **API catalog** is an inventory of available APIs and their ownership, lifecycle, and contract locations. A useful catalog entry can include:

- API and domain name.
- Owning team and support contact.
- Consumer audience: internal, partner, or public.
- Production and test environments.
- Current contract version and documentation location.

- Authentication model.
- Data sensitivity classification.
- SLO or support tier.
- Lifecycle state: proposed, active, deprecated, or retired.
- Known consumers where appropriate.

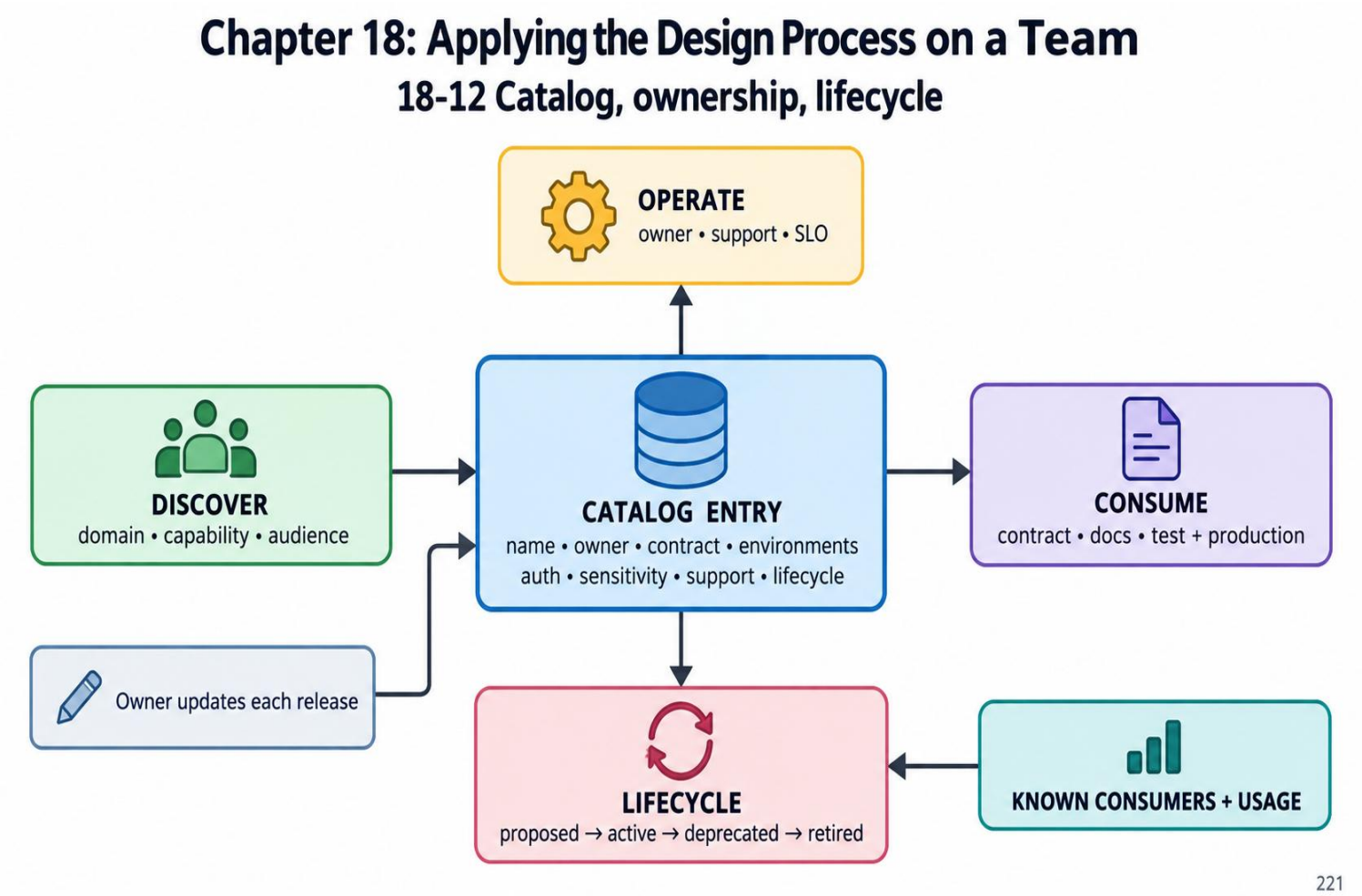



*Figure 18-10: API catalog connecting consumers to ownership and lifecycle*

The catalog prevents teams from unknowingly building duplicate capabilities and helps incident responders find owners. It also supports deprecation by connecting usage to accountable consumers.

A catalog with stale entries is worse than a smaller accurate one. Update ownership and lifecycle through the same release process used for the API.

## Governance should be proportional to risk

**API governance** is the set of policies, review practices, automation, and ownership mechanisms used to maintain quality across an API portfolio. Governance should catch cross-team risks without turning every design choice into central approval.

A proportionate model can use:

- Self-service templates and examples for common designs.
- Automated checks for naming, security declarations, schema validity, and obvious compatibility breaks.
- Peer review within the owning team for low-risk compatible additions.
- Cross-team review for new shared APIs, incompatible changes, sensitive data, or unusual reliability models.
- Specialist review for high-impact security, privacy, legal, performance, or consistency concerns.
- Portfolio review for duplication, ownership gaps, and retirement.

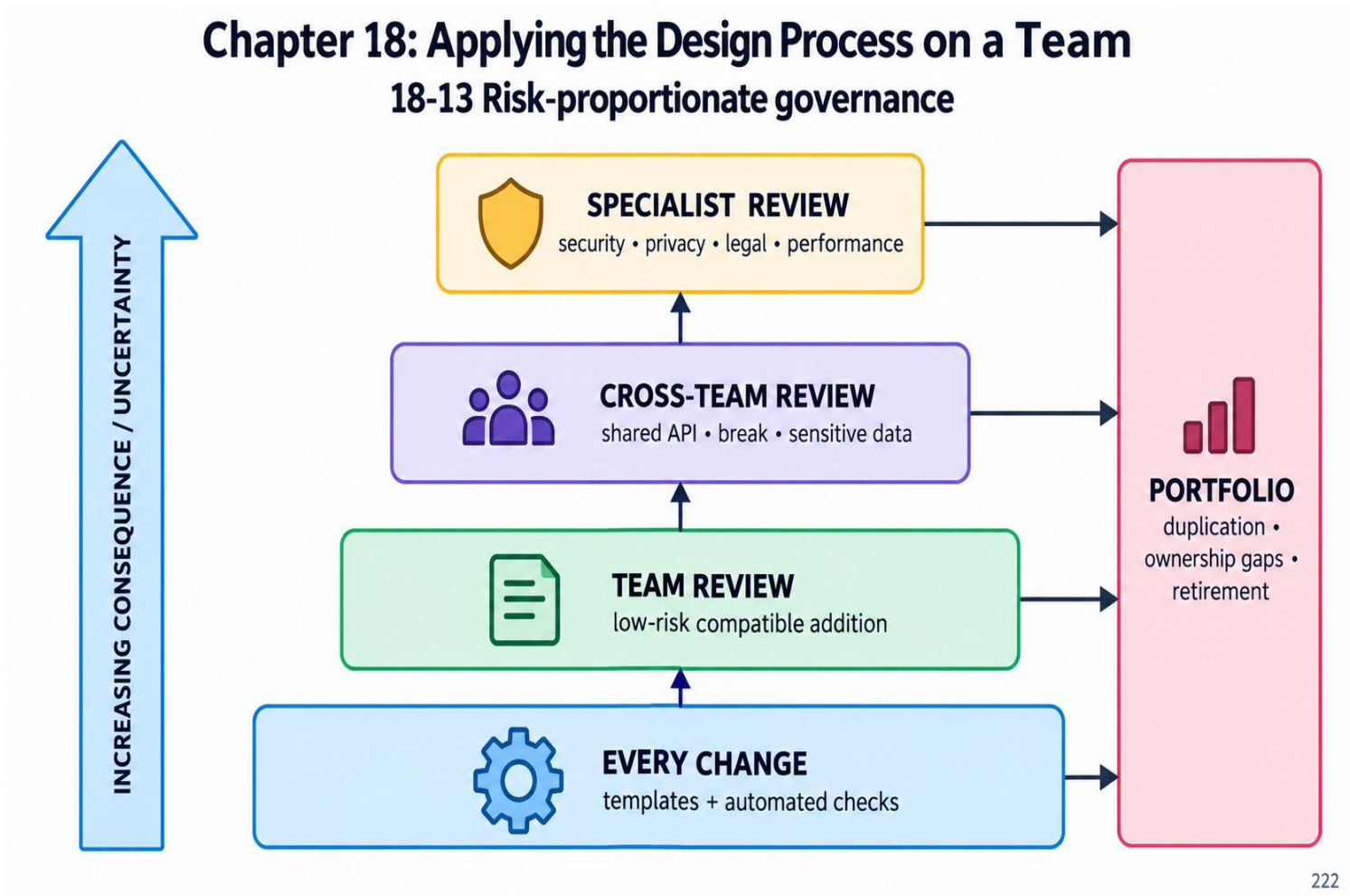



*Figure 18-11: Governance proportional to risk*

Central experts are most valuable when they create reusable guidance and join decisions with material risk. Requiring them to approve every description edit creates delay without improving judgment.

## Know when specialist help is necessary

A systems engineer should recognize boundaries that need deeper expertise. Seek specialist review when the API involves:

- Custom cryptography, credential issuance, or unusual identity federation.
- Highly sensitive personal, health, financial, or regulated data.
- Cross-border data transfer, retention mandates, consent, or legal hold.
- High-value administrative operations or severe abuse risk.
- Hard real-time or unusually strict latency requirements.
- Large-scale streaming, broker partitioning, or complex replay behavior.
- Distributed consistency that affects money, inventory, or irreversible actions.
- Safety-critical decisions or high-assurance verification.

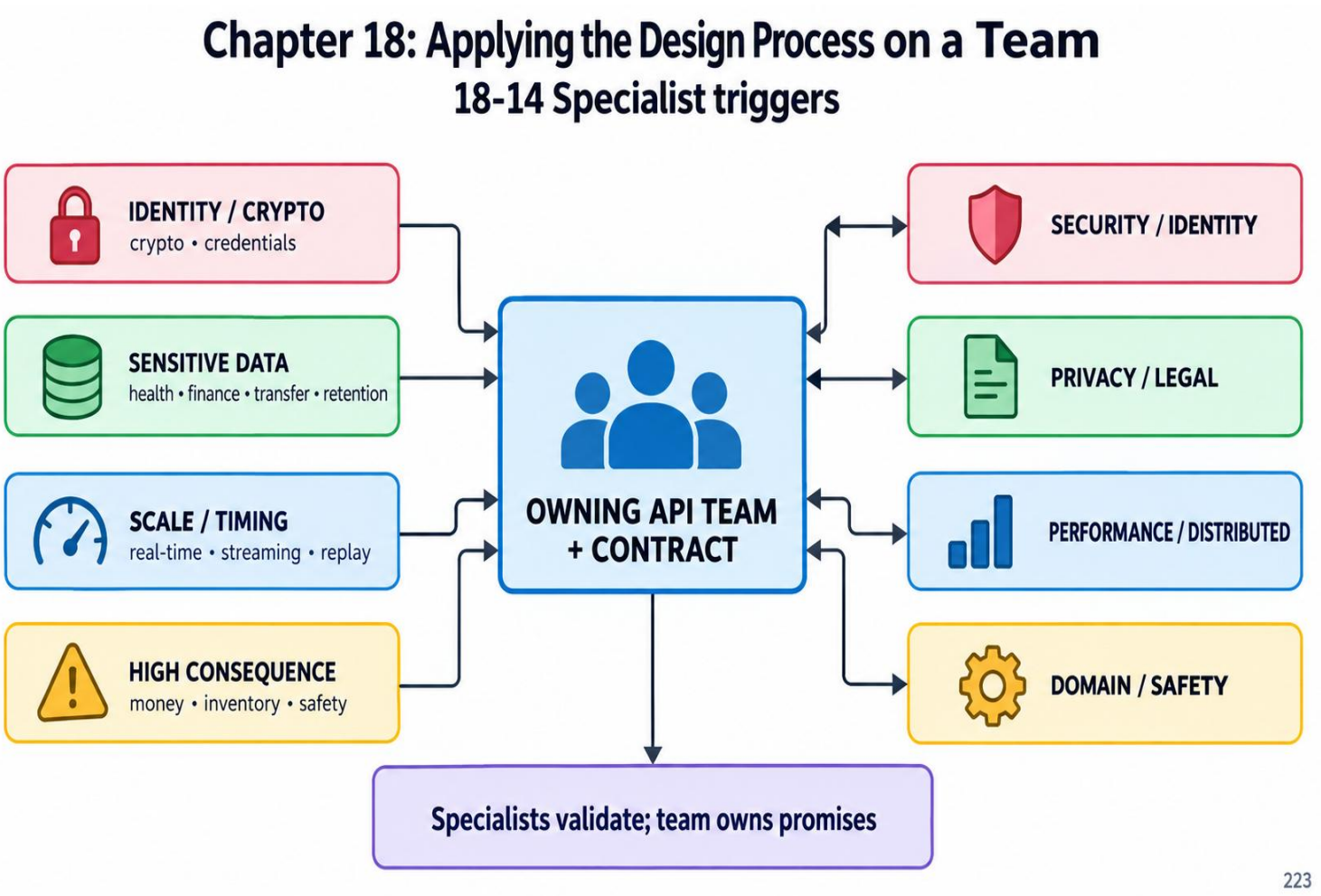



*Figure 18-12: When specialist review is necessary*

The API designer still defines the consumer-visible contract. Specialist involvement helps ensure that the underlying control or guarantee is realistic and compliant.

## End-to-end case study: the support-ticket API

The following design package applies the complete process without repeating every earlier explanation.

### Design brief

**Purpose:** Allow requesters to create and follow support tickets, allow authorized agents to work those tickets, and support automation and reporting without exposing internal storage or unsafe retry behavior.

**Owner:** Support Platform team.

**Primary consumers:**

- Agent web application.
- Requester portal.
- Ticket automation service.
- Reporting service.
- Approved webhook subscribers.

**Important constraints:**

- The service is multi-tenant.
- Ticket descriptions and requester details may contain personal information.
- Consumers deploy independently.
- Automation must retry creation safely.
- Agent updates must not silently overwrite one another.
- Large exports must not block interactive traffic.

- Security-significant changes require audit records.

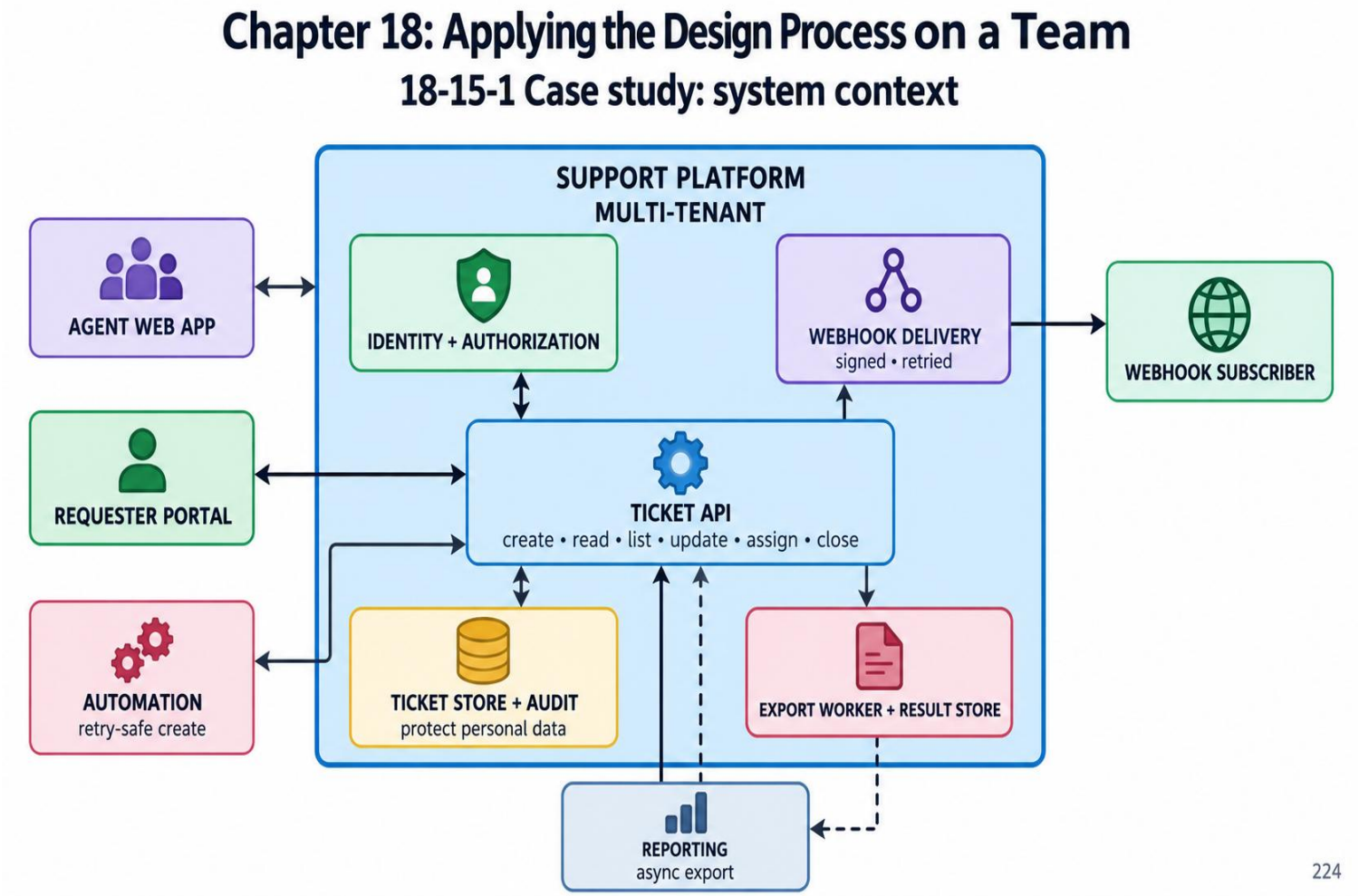



*Figure 18-13: Case-study system context and consumers*

**First-release scope:**

- Create, retrieve, list, update, assign, close, and comment on tickets.
- Filtered cursor-based listing.
- Asynchronous resolved-ticket export.
- Assignment webhook.

**Deferred scope:**

- Bulk state changes.
- Arbitrary analytical queries against the transactional API.
- Cross-ticket workflow orchestration.
- Public attachment sharing.
- Custom authentication mechanisms.

Deferring these capabilities prevents speculative complexity while leaving room for later reviewed additions.

### Representative resource and operation set

| Consumer task | Contract shape | Important behavior |
|---|---|---|
| Create a ticket | POST /tickets | Idempotency key protects retry; returns 201 and stable identifier |
| Retrieve one ticket | GET /tickets/{ticketId} | Resource-level authorization filters access and fields |
| List tickets | GET /tickets | Filters, deterministic order, opaque cursor, bounded page size |
| Update editable fields | PATCH /tickets/{ticketId} | Requires current If-Match value for protected updates |
| Assign a ticket | POST /tickets/{ticketId}/assign | Checks permission, eligibility, state, and concurrency atomically |
| Close a ticket | POST /tickets/{ticketId}/close | Enforces lifecycle transition and records actor |
| Add a comment | POST /tickets/{ticketId}/comments | Public and internal visibility have different permissions |
| Request an export | POST /ticket-exports | Returns 202 and an operation resource |
| Read operation status | GET /operations/{operationId} | Reports terminal result, failure, and retention |
| Receive assignment fact | ticket.assigned webhook event | Signed, retried, and deduplicated by event ID |

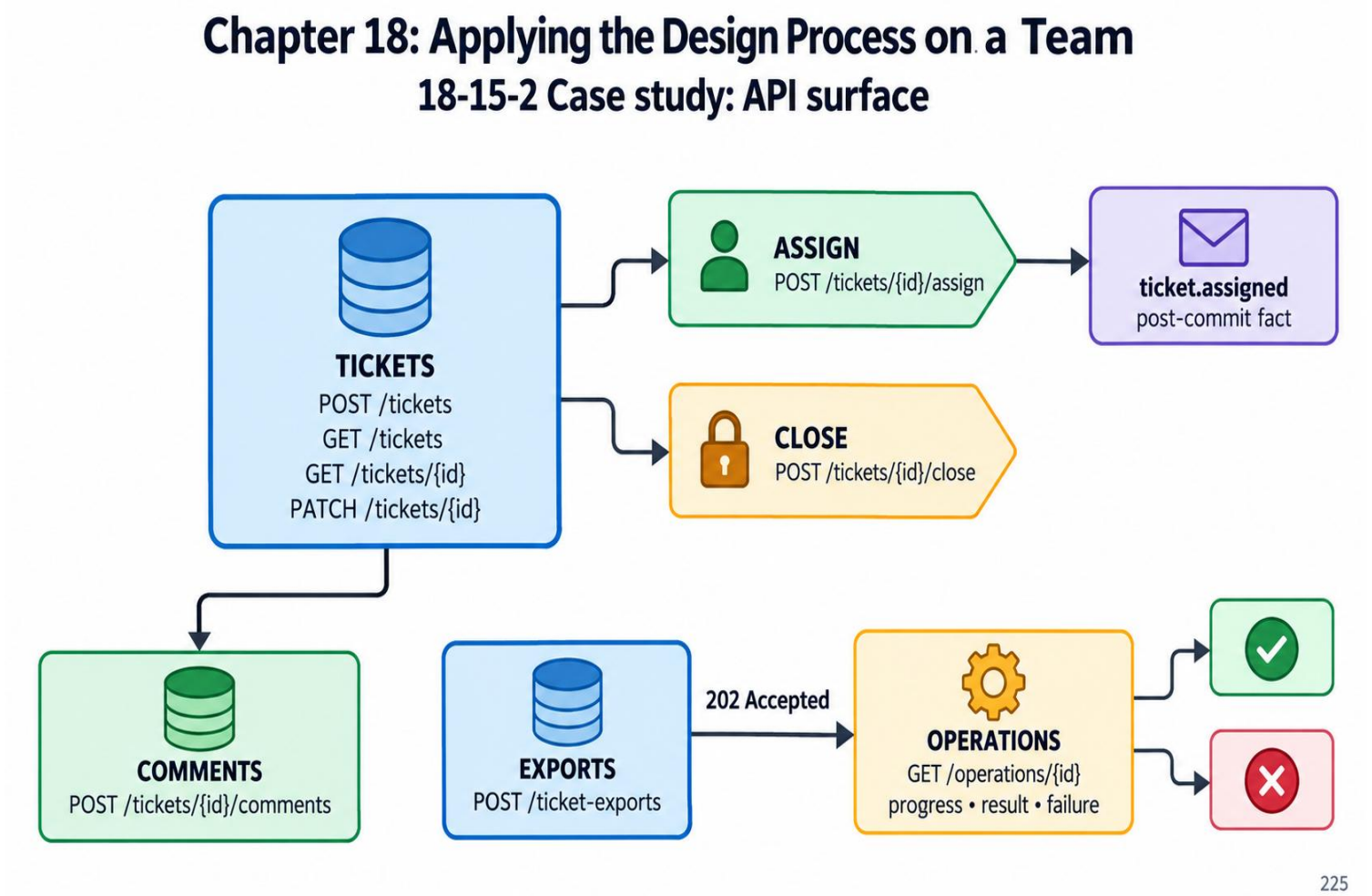



*Figure 18-14: Case-study resource and operation surface*

This surface reflects current consumer tasks. It does not expose database tables, generic query execution, or one endpoint that performs unrelated operations.

### Core ticket representation

A detail response can use:

```
{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "description": "The reset link reports that it has expired.",
  "status": "in_progress",
  "priority": "normal",
  "requesterId": "usr_882",
  "assigneeId": "usr_314",
  "createdAt": "2026-09-18T09:42:31Z",
  "updatedAt": "2026-09-24T08:45:12Z"
}
```

The HTTP response carries an opaque `ETag` for concurrency. Internal comments, audit records, database keys, and credentials are not embedded in this representation. Field visibility can be narrower for requesters and reporting consumers.

Creation and update requests use dedicated schemas so that consumers cannot set provider-controlled fields such as `id`, `status`, timestamps, or trusted actor information.

## Permission summary

| Capability | Requester | Support agent | Administrator | Reporting service |
|---|---|---|---|---|
| Create ticket | Allowed within own tenant | Allowed when acting for permitted channel | Allowed within scope | Denied |
| Read ticket | Own visible tickets | Permitted queues | Administrative scope | Approved reporting fields only |
| Add public comment | Own visible tickets | Permitted queues | Within scope | Denied |
| Add internal comment | Denied | Permitted queues | Within scope | Denied |
| Assign or close | Denied | Permitted queues and states | Within scope | Denied |
| Administrative delete | Denied | Denied | Separate privileged capability | Denied |
| Request export | Own limited data only if offered | Approved scope | Approved scope | Approved reporting scope |

Every permitted entry still requires object-level tenant and resource checks. A role alone does not authorize every ticket.

## Key design decisions

| Decision | Rationale | Consumer obligation |
|---|---|---|
| Opaque string ticket identifiers | Stable across storage changes and safe to treat as nonnumeric | Do not parse or generate identifiers |
| Cursor pagination ordered by `createdAt` and `id` | Stable continuation under changing data | Treat cursor as opaque and follow returned continuation |
| Problem Details-style error body with stable `code` | Supports machine branching and human diagnosis | Branch on status and code, not message text |
| Idempotency key on ticket and export creation | Makes uncertain retries safe | Reuse the same key only for the same logical request |
| `ETag` and `If-Match` on protected changes | Prevents silent lost updates | Refresh and resolve after a stale precondition |
| Bearer-token authentication plus resource authorization | Supports user and service identities without granting universal access | Request only needed scopes and preserve credential secrecy |
| Asynchronous exports | Protects interactive latency and gives long work a lifecycle | Poll at guided intervals and handle terminal failure |
| At-least-once webhook delivery | Supports bounded retry after network uncertainty | Verify signature and deduplicate by event ID |
| Additive evolution within the initial contract where safe | Limits parallel-version cost | Ignore unknown fields and follow documented enum policy |
| Request ID in every response | Connects consumer reports to provider evidence | Record it with failures, never treat it as a secret credential |

Each consequential decision has an owner and a short ADR. The table summarizes outcomes; it does not replace detailed behavior in the contract.

## Representative ticket creation

The automation service submits:

```
POST /tickets HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access-token>
Idempotency-Key: 8f0b7f35-2f37-49e3-8ddb-f7b7f68e4604

{
  "subject": "Unable to reset password",
  "description": "The reset link reports that it has expired.",
  "priority": "normal"
}
```

A successful response is:

```
HTTP/1.1 201 Created
Location: /tickets/tkt_48271
Content-Type: application/json
ETag: "ticket-48271-v1"
X-Request-Id: req_1001

{
  "id": "tkt_48271",
  "subject": "Unable to reset password",
  "description": "The reset link reports that it has expired.",
  "status": "open",
  "priority": "normal",
  "requesterId": "usr_882",
  "assigneeId": null,
  "createdAt": "2026-09-18T09:42:31Z",
  "updatedAt": "2026-09-18T09:42:31Z"
}
```

The body contains the created ticket representation. Repeating the same logical request with the same idempotency key returns the recorded logical result rather than creating another ticket. Reusing the key for changed input produces the documented key-reuse error.

### Representative assignment

An agent that previously retrieved the ticket submits:

```
POST /tickets/tkt_48271/assign HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access-token>
If-Match: "ticket-48271-v7"

{
  "assigneeId": "usr_314"
}
```

Success means that the provider has atomically applied the assignment-related durable state defined by the contract and returns the new ticket representation with a new `ETag`.

The operation can fail predictably because:

- The caller lacks queue or tenant permission.
- The assignee is not eligible.
- The ticket state does not permit assignment.
- The `If-Match` value is stale.
- A required dependency is temporarily unavailable before commit.

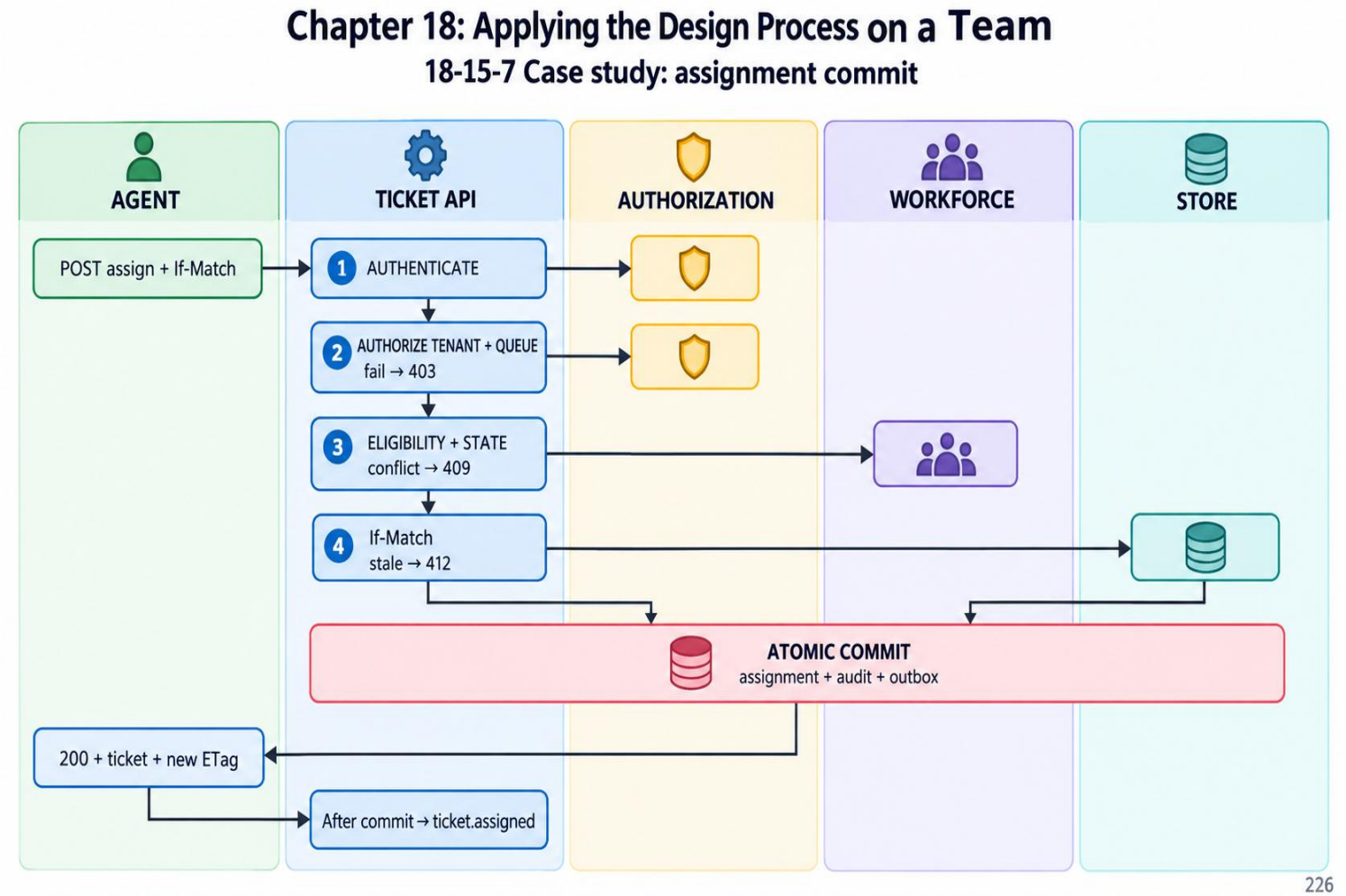



*Figure 18-15: Case-study assignment decision and commit path*

These conditions have distinct status, stable error code, and recovery behavior. The client does not retry a stale write or business conflict as though it were a transient server failure.

### Representative collection retrieval

The agent application can request:

```
GET /tickets?status=open&assigneeId=usr_314&sort=createdAt&limit=50 HTTP/1.1
Authorization: Bearer <access-token>
```

The response contains the current page and an opaque next cursor when more data is available. The provider applies a deterministic secondary order by ticket ID. The contract does not promise an expensive exact total count for every query.

Permission filtering occurs before data is returned. The API must not reveal the existence or count of another tenant's tickets through collection metadata.

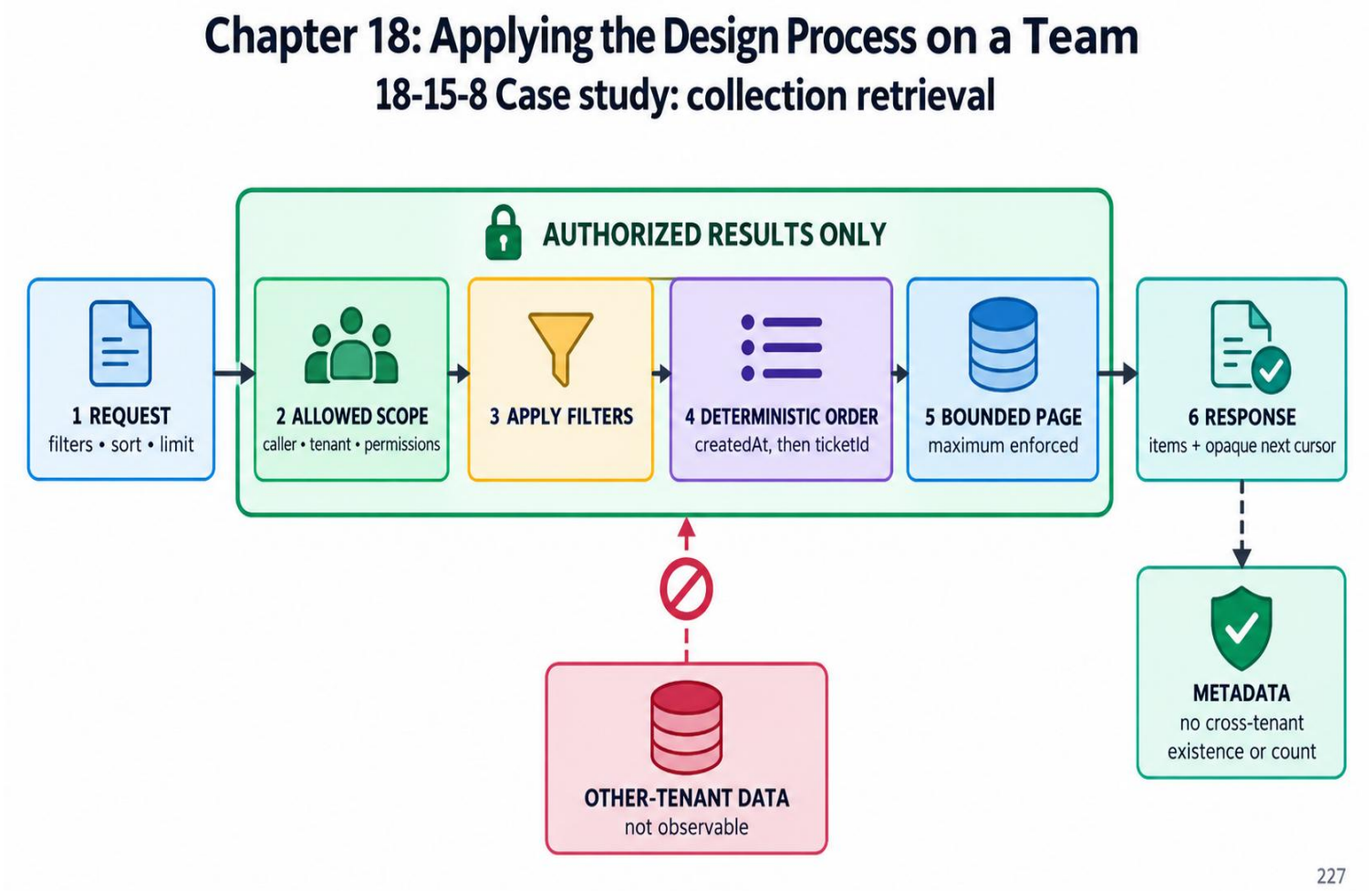



*Figure 18-16: Case-study collection retrieval*

### Asynchronous export and webhook

The reporting service creates an export with `POST /ticket-exports`. The `202` response identifies an operation resource. Success provides a time-limited result link; failure provides a stable error and request reference. Operation and file retention are documented separately.

After an assignment commits, the provider creates a `ticket.assigned` event. A webhook delivery includes an event ID, occurrence time, ticket identity, assignment data, signature, and replay-protection timestamp. The receiver stores the event durably, deduplicates by ID, and acknowledges promptly.

The export and webhook reuse the general error, security, idempotency, compatibility, and observability conventions of the API rather than inventing independent rules.

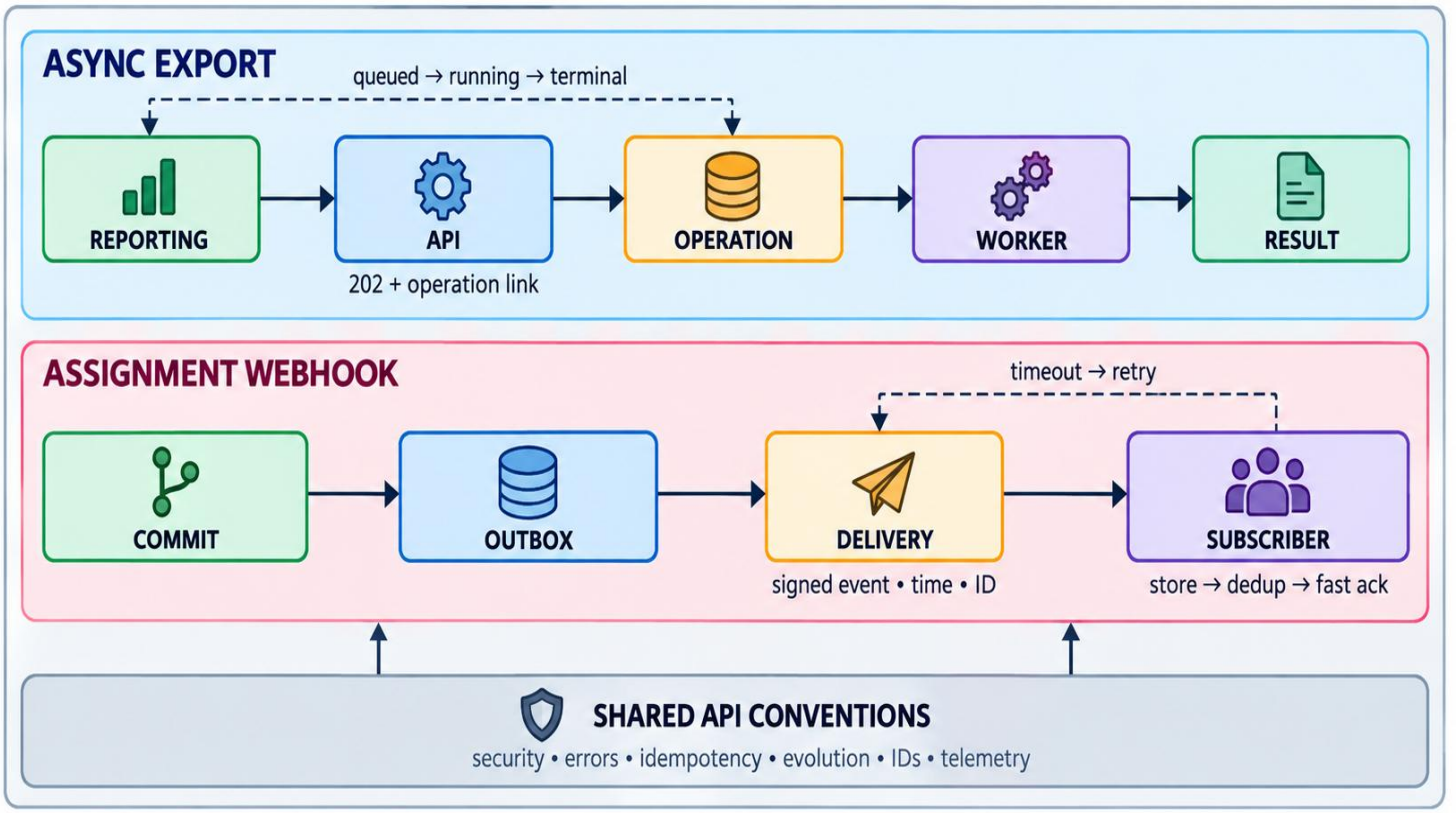





*Figure 18-17: Case-study asynchronous export and webhook architecture*

### OpenAPI excerpt

The structured description for creation can include:

```
/tickets:
  post:
    operationId: createTicket
    summary: Create a support ticket
    security:
      - bearerAuth: []
    parameters:
      - name: Idempotency-Key
        in: header
        required: true
        schema:
          type: string
          maxLength: 200
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/TicketCreateRequest"
    responses:
      "201":
        description: Ticket created or prior result replayed.
        headers:
          Location:
            schema:
              type: string
          ETag:
            schema:
              type: string
```

```
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/TicketDetail"
      "400":
        $ref: "#/components/responses/InvalidRequest"
      "403":
        $ref: "#/components/responses/Forbidden"
      "409":
        $ref: "#/components/responses/Conflict"
```

The excerpt gives tools the operation, security requirement, idempotency header, request schema, and response structures. Accompanying prose still defines key scope, request matching, retention, in-progress duplication, permission policy, and conflict recovery.

## Verification plan

The release test matrix includes:

| Area | Representative verification |
|---|---|
| Creation | Valid creation, missing field, boundary length, duplicate same-key request, changed-input key reuse |
| Authorization | Own-tenant access, peer-resource denial, cross-tenant denial, field-level visibility |
| Assignment | Allowed transition, ineligible assignee, closed-ticket conflict, stale If-Match, dependency timeout before commit |
| Collections | Filter correctness, deterministic tie order, cursor continuation, maximum limit, permission-filtered metadata |
| Compatibility | Previous examples and consumer contracts pass against the new provider |
| Export | 202 acceptance, state progression, failure result, cancellation policy, result expiration |
| Webhook | Valid signature, invalid signature, replay window, duplicate event, retry after receiver timeout |
| Operations | Request-ID propagation, safe logs, metrics, trace continuity, audit record on successful assignment |

Schema validation, provider conformance, integration tests, consumer-driven contracts, and controlled fault injection each cover different rows. No single test layer is expected to prove the whole contract.

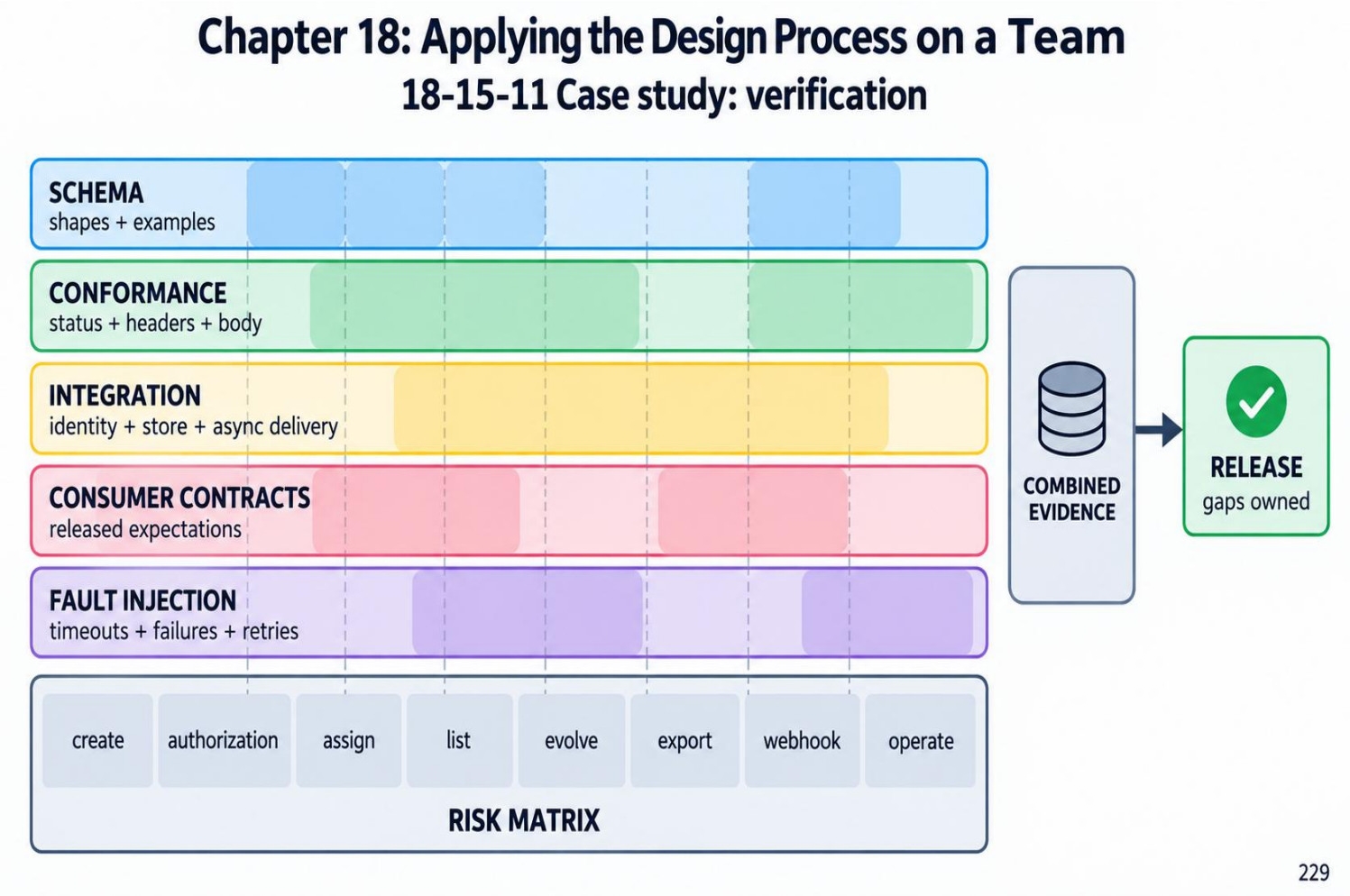



*Figure 18-18: Verification layers over the case-study test matrix*

## Release notes

A useful initial release note is concise and action-oriented:

> **Ticket API 1.0** introduces ticket creation, retrieval, cursor-based listing, assignment, closure, comments, asynchronous exports, and assignment webhooks. Ticket and export creation require an `Idempotency-Key`. Protected ticket changes require the latest `ETag` through `If-Match`. Consumers must ignore unknown response fields and follow the documented policy for future enumeration values. Default and maximum page sizes, rate-limit behavior, operation retention, webhook retry behavior, and support contacts are published with the contract.

Later releases should state compatibility, affected operations, adoption work, and deprecation timelines rather than listing internal implementation changes.

### Operational readiness

Before general release, the owner confirms:

- Dashboards show request volume, latency, errors, saturation, retries, rate limits, export age, and webhook failures.
- Alerts distinguish widespread provider failure from one consumer's invalid traffic.
- Request and correlation identifiers propagate through gateway, service, export, and webhook paths.
- Audit records cover assignment, closure, administrative deletion, and credential-sensitive changes as required.
- Logs redact tokens, ticket descriptions, and personal data by policy.
- Rate limits, page limits, payload limits, timeouts, and retry guidance are configured and tested.
- Runbooks cover dependency failure, stuck exports, webhook delivery failure, credential compromise, and rollback.
- Credential and webhook-signing key rotation have owners and tested procedures.
- The API catalog entry, ownership, contract, and support path are current.
- SLO measurements exclude defined client errors consistently and reflect consumer-visible operations.

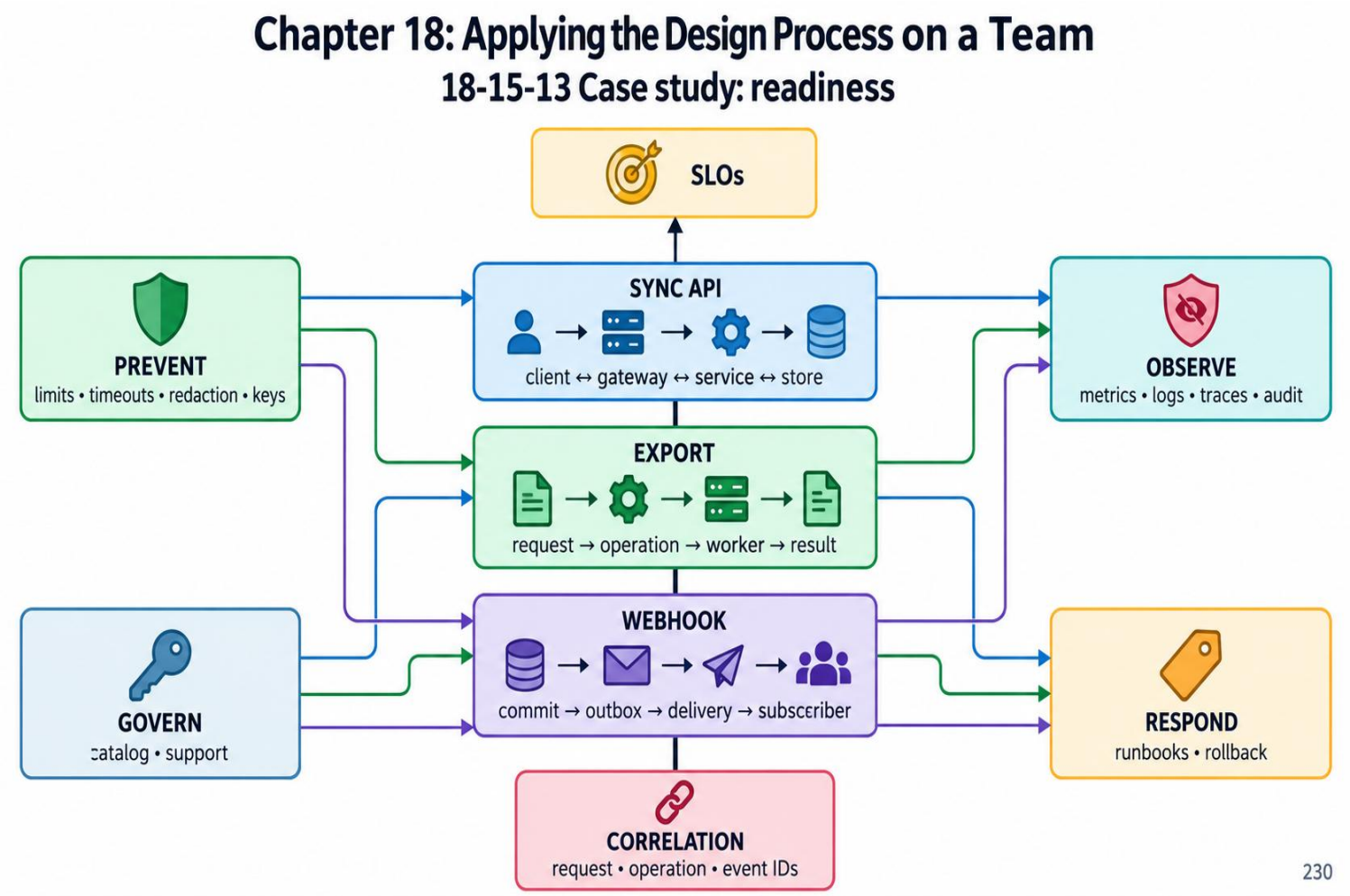



*Figure 18-19: Operational readiness across synchronous and asynchronous paths*

Operational readiness is part of release acceptance, not work postponed until the first incident.

## Use a practical design-review checklist

A final review can ask:

### Consumer and domain

- Are consumers, tasks, scope, and ownership explicit?
- Do resource names and state transitions match shared domain language?
- Can each task be completed without hidden implementation knowledge or excessive calls?

### Contract behavior

- Are method, path, inputs, outputs, errors, limits, and success guarantees defined?
- Are omission, null, default, ordering, and pagination semantics clear?
- Are duplicate requests, concurrency, transactions, and asynchronous outcomes safe?

### Security and privacy

- Are authentication and resource-level authorization separate and complete?
- Are tenant, field, secret, and sensitive-data boundaries enforced?
- Are credential lifecycle, abuse controls, audit needs, and telemetry safety planned?

### Compatibility and lifecycle

- Can likely additions be made without breaking supported consumers?
- Are enum, unknown-field, versioning, deprecation, migration, and retirement policies explicit?
- Are consumers and real usage discoverable?

### Verification and operation

- Do examples, OpenAPI, implementation, and tests agree?
- Are important failures, retries, limits, and partial dependencies tested?
- Can a consumer-visible error be traced through logs, metrics, traces, and audit evidence?
- Are SLOs, dashboards, alerts, runbooks, and support ownership ready?

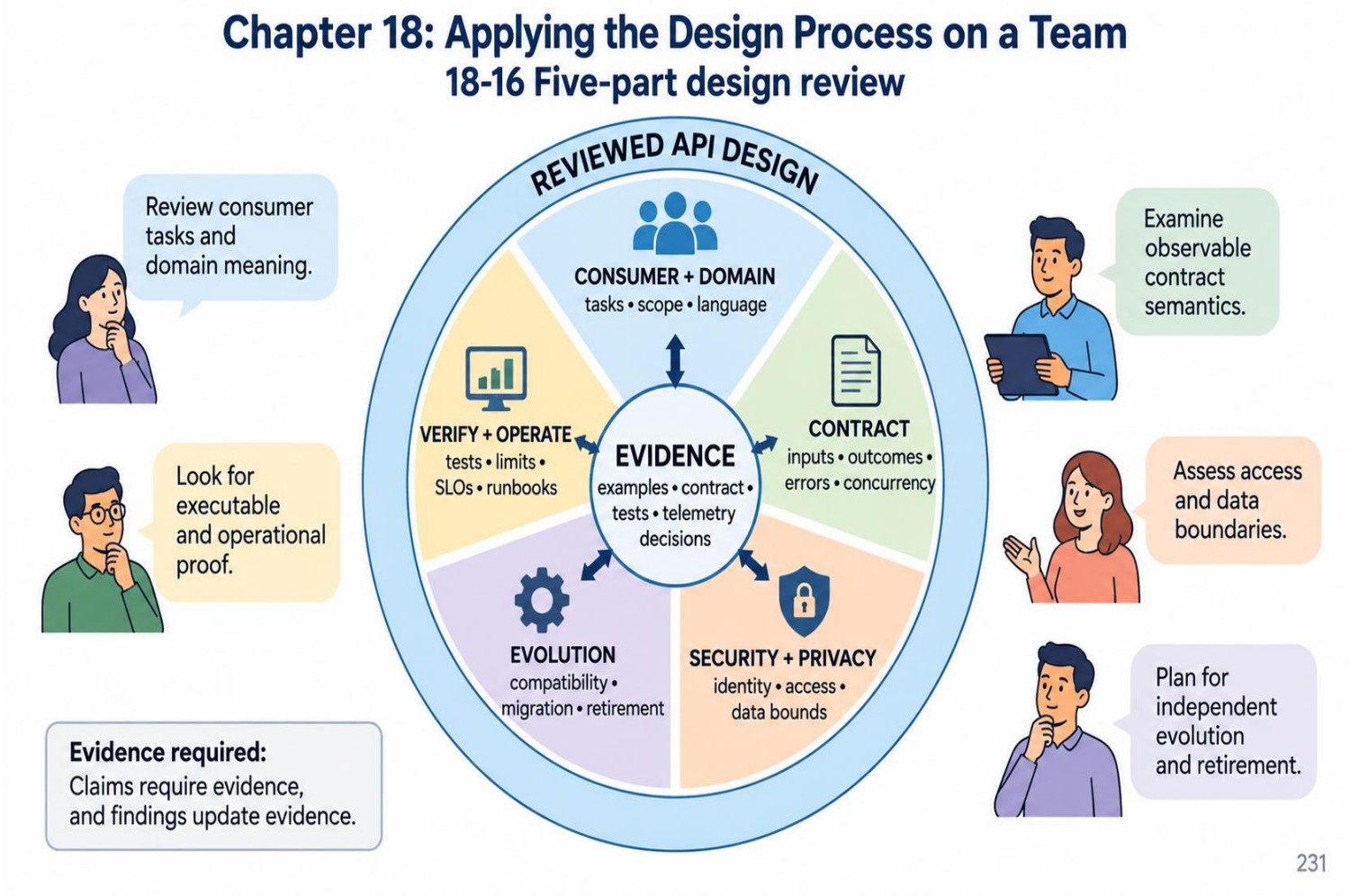



*Figure 18-20: Five-part design-review checklist*

A checklist supports judgment; it does not replace it. A checked box is meaningful only when evidence exists.

## Two team-process misconceptions

### "Governance means a central team must approve every small decision"

Effective governance creates clear defaults, automated checks, discoverable ownership, and focused expert review for material risk. Central approval of every edit can slow delivery while encouraging teams to treat compliance as a substitute for design reasoning.

### "The contract is finished when implementation begins"

Implementation, testing, release, observed use, compatibility work, and retirement all reveal or create contract obligations. The owner must manage the API for its full consumer lifecycle.

## The durable working model

A systems engineer can approach any API design with the following model:

1. **Contract:** Define exactly what consumers may rely on.
2. **Boundary:** Hide internal implementation while enforcing identity, permission, and data ownership.
3. **Semantics:** Use operations, messages, errors, and state transitions that communicate business meaning.
4. **Safety:** Account for invalid input, duplicate delivery, stale state, partial failure, and overload.
5. **Verification:** Turn important promises into examples, descriptions, reviews, and executable tests.
6. **Evolution:** Assume provider and consumers change independently.
7. **Operation:** Make behavior bounded, measurable, traceable, and supportable.
8. **Ownership:** Keep decisions, releases, incidents, migration, and retirement accountable.

**Good API design allows independently changing systems and teams to cooperate without relying on hidden knowledge.** The result is not merely an interface that works today, but a contract that remains understandable, safe, and manageable as consumers, implementations, and operational conditions change.

## Chapter Summary

- API design is a managed team process, not a one-time endpoint specification exercise.
- Connected artifacts and proportional review keep consumer needs, implementation, compatibility, release, and operation aligned.
- The durable working model combines contract, boundary, semantics, safety, verification, evolution, operation, and ownership.

## Review Questions

1. Which decisions require explicit ownership throughout an API's lifecycle?
2. How do examples, OpenAPI, tests, release plans, and runbooks reinforce one another?
3. Which parts of the durable working model would you inspect first when reviewing a high-risk state-changing operation?

# Closing Notes

No part of this publication may be reproduced, republished, distributed, or transmitted without prior permission. This work may not be transferred to a third party without authorization, whether for payment or free of charge.

The content of this book is based on information that was available and could be verified at the time of writing. Technologies, product specifications, and service details may change. Before applying anything in practice, consult the latest information on the relevant official website or in the official documentation.

The author assumes no responsibility for any loss, damage, or disadvantage arising from the use of the information contained in this book.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.