

Log Design A Visual Guide

Practical Fundamentals for Troubleshooting and Reliable Operations

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with log 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

This is a beginner-friendly book focused on the essential fundamentals of log design. **Read the chapters in order** the first time through. Each chapter builds on terminology, decisions, and examples introduced earlier, so the sequence helps you develop the necessary background knowledge step by step.

At the beginning of each chapter, review **What You'll Be Able to Do** before reading the main explanation. These outcomes show which practical abilities the chapter is intended to develop and help you distinguish the central ideas from supporting detail.

Use the diagrams to understand **relationships, processes, boundaries, and structures** that are difficult to picture from prose alone. Read the surrounding text as well: each diagram supports the explanation but does not replace it.

Try the code, command, query, and hands-on examples yourself whenever your environment allows it. Reproducing a small example and changing one value at a time is often the fastest way to connect the design principle to system behavior.

After each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you can explain the fundamentals in your own words. Return to the relevant section when an answer depends on memorized wording rather than a clear mental model.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as **signposts** for techniques, risks, and vocabulary that you may need when a future project introduces more demanding operational requirements.

Table of Contents

Chapter 1 — The Big Picture: Logs Are Evidence of Events

[Events and records are not the same thing](#)

[Logs are one observability signal](#)

[Different logs serve different purposes](#)

[The qualities of a useful design](#)

[Why more logs can reduce visibility](#)

Chapter 2 — Start with Use Cases and Requirements

[Identify the people and systems that consume logs](#)

[Turn a scenario into answerable questions](#)

[Define requirements that can be tested](#)

[A requirement worksheet for one use case](#)

[Assign ownership before records become permanent](#)

[Balance value, risk, volume, and cost](#)

[Resolve conflicts explicitly](#)

[The central mistake to avoid](#)

Chapter 1 — The Big Picture: Logs Are Evidence of Events

What You'll Be Able to Do

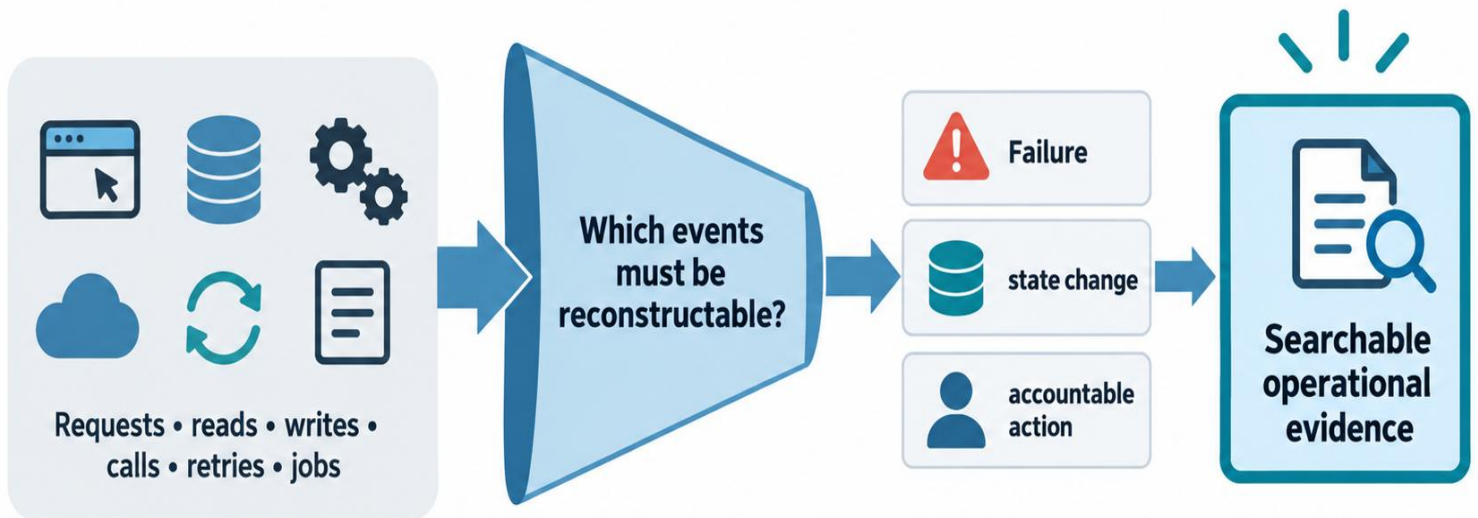
- Distinguish a real system event from the log record that represents it.
- Explain how logs complement metrics and traces during operations.
- Judge whether a proposed record provides relevant, searchable, safe, and proportionate evidence.

Good log design begins with the questions that people must answer during operation. It does not begin with deciding where to place print statements.

A production system continually performs work: it accepts requests, reads and writes data, calls other systems, starts scheduled jobs, retries failed operations, and makes access decisions. Some of those actions must later be reconstructed. An engineer may need to determine why a request failed, a service desk analyst may need to confirm what happened to one customer transaction, or an operator may need to identify which dependency became slow. A **dependency** is another service, system, or component that the current operation relies on.

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-0 From system activity to operational evidence



001

Figure 1-1: From system activity to operational evidence

A useful log provides evidence for those investigations. It should help answer questions such as:

- What happened?
- When did it happen?
- Which system or component was involved?
- Which request, job, message, or business object was affected?
- What was the result?

- Why did the system reach that result, as far as it could reliably determine?

This is similar to a ship's logbook. A useful logbook does not record every movement of every crew member. It records events that matter to navigation, safety, and later review, together with enough context to understand them. System logs should follow the same principle.

Events and records are not the same thing

A **log event** is a meaningful occurrence in the system, such as a request being rejected, a batch job completing, or a dependency call timing out. A **batch job** is a program that processes a defined collection of work, often on a schedule and without an interactive user waiting. A **log record** is the stored representation of an event.

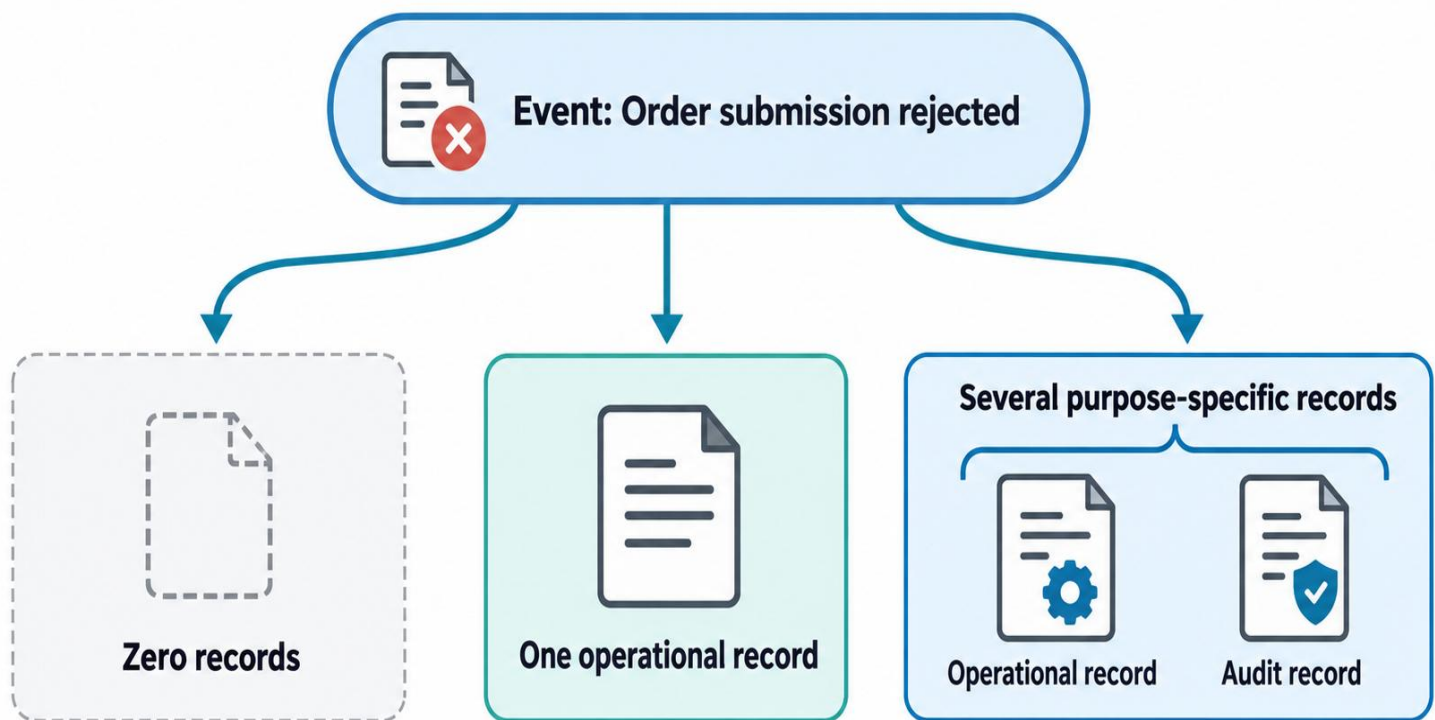
The distinction matters because the event exists independently of the text written about it. For example:

- Event: an order submission is rejected because one item is unavailable.
- Record: a timestamped entry containing the event name, order identifier, outcome, and reason category.

A system may produce one record for an event, several records for different aspects of it, or no record at all. Log design decides which events deserve records and what those records must contain.

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-1 Event, record, and representation are different layers



002

Figure 1-2: Event, record, and representation are different layers

A log record normally has two complementary parts:

- A **message** gives a concise explanation for a person reading the log.
- **Fields** store named values that software can search, filter, group, and compare.

Consider this record:

Something failed

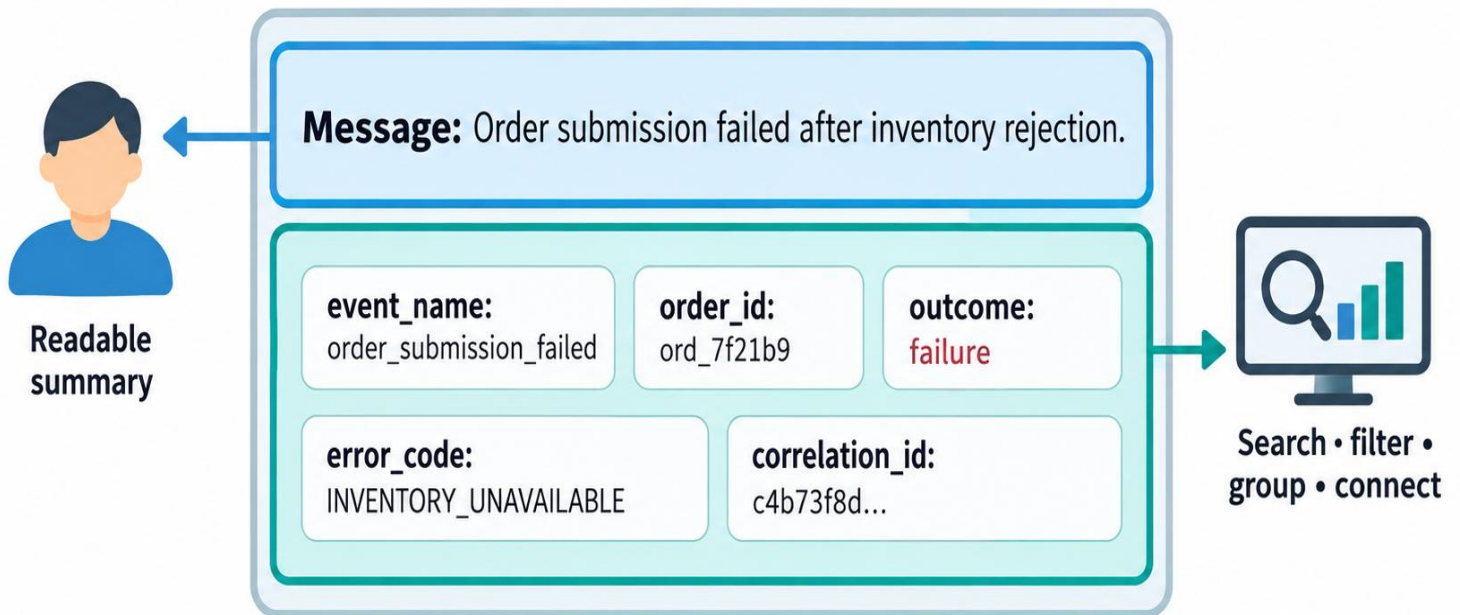
It indicates that a problem occurred, but almost every useful fact is missing. An investigator cannot tell which operation failed, whether the failure affected a user, or how to find related records.

A more useful representation is:

```
{
  "timestamp": "2026-09-18T04:21:36.417Z",
  "severity": "ERROR",
  "event_name": "order_submission_failed",
  "message": "Order submission failed after inventory reservation was rejected.",
  "service": "order-api",
  "environment": "production",
  "outcome": "failure",
  "order_id": "ord_7f21b9",
  "error_code": "INVENTORY_UNAVAILABLE",
  "correlation_id": "c4b73f8d2a6e4a52"
}
```

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-1 Anatomy of a useful log record



003

Figure 1-3: Anatomy of a useful log record

The message gives the human-readable summary. The fields make the event searchable and connect it to a service, an order, a stable error category, and related work. A **timestamp** records when the event occurred, and an **event name** gives the event type a stable label. **Severity** indicates the event's operational significance, an **error code** gives the failure a stable machine-readable category, and a **correlation identifier** connects records produced by related work. Later chapters explain how to choose each field. The important point here is that **the record describes a specific event in a form that remains useful after the original developer is unavailable.**

Logs are one observability signal

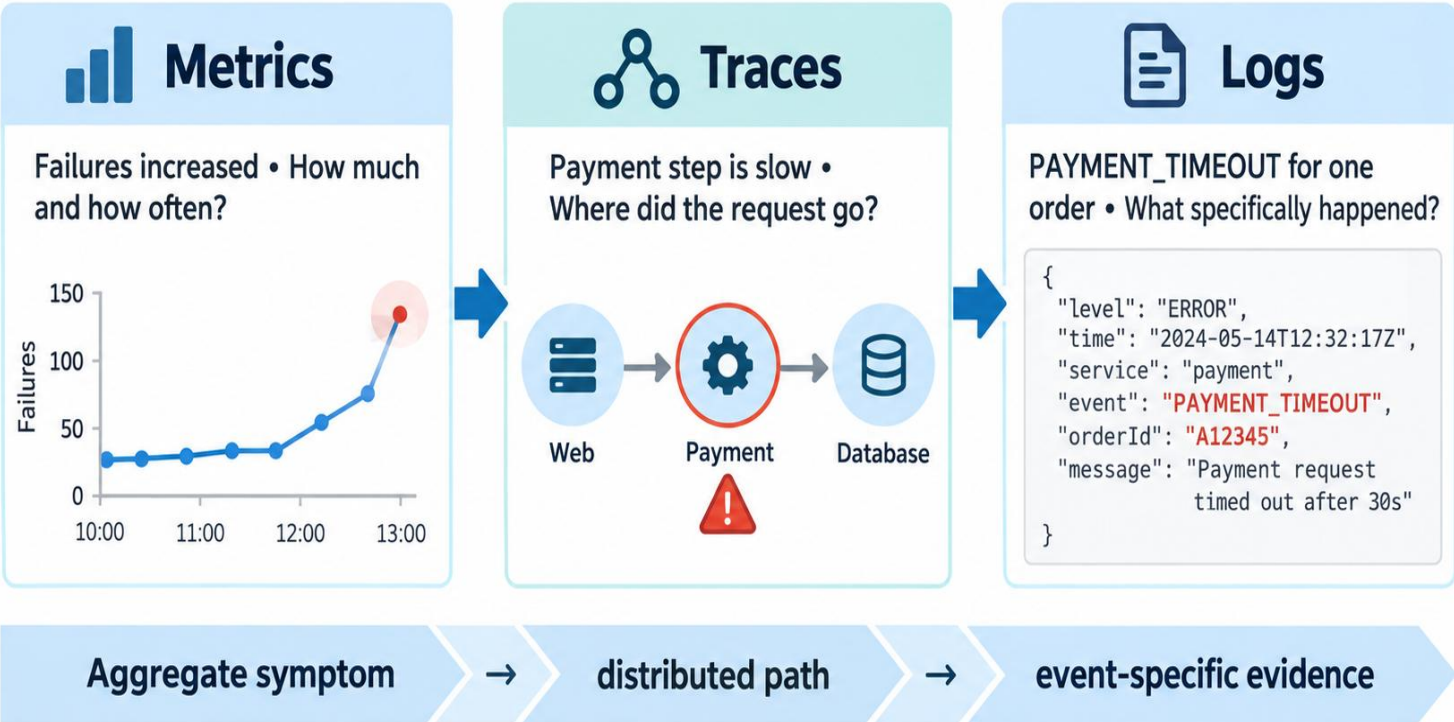
Observability is the ability to understand a system's internal condition by examining the outputs it produces. Logs are one of the main observability signals, together with metrics and traces.

A **metric** is a numeric measurement collected over time, such as request count, error rate, queue depth, or processor utilization. A **trace** is a connected representation of how one request or unit of work travels through multiple operations or services.

Signal	Best at answering	Typical example
Logs	What specifically happened in a particular event?	Why did order ord_7f21b9 fail?
Metrics	How much, how often, or how fast over time?	What percentage of order requests failed in the last 15 minutes?
Traces	Where did one request spend time across components?	Which service made this checkout request slow?

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-2 How logs, metrics, and traces cooperate during an investigation



004

Figure 1-4: How logs, metrics, and traces cooperate during an investigation

These signals complement one another. A metric may show that failures increased. A trace may identify the slow or failing step. Logs may then show the exact error category and operational context for an affected request.

Logs should not be forced to perform every observability task. Recording one log entry every second merely to calculate processor utilization is usually less efficient than using a metric. Likewise, a trace can show the path of a distributed request more naturally than a collection of unrelated messages. Good design uses each signal for the questions it answers well.

Different logs serve different purposes

The word “log” covers several types of evidence. Their purposes overlap, but they are not interchangeable.

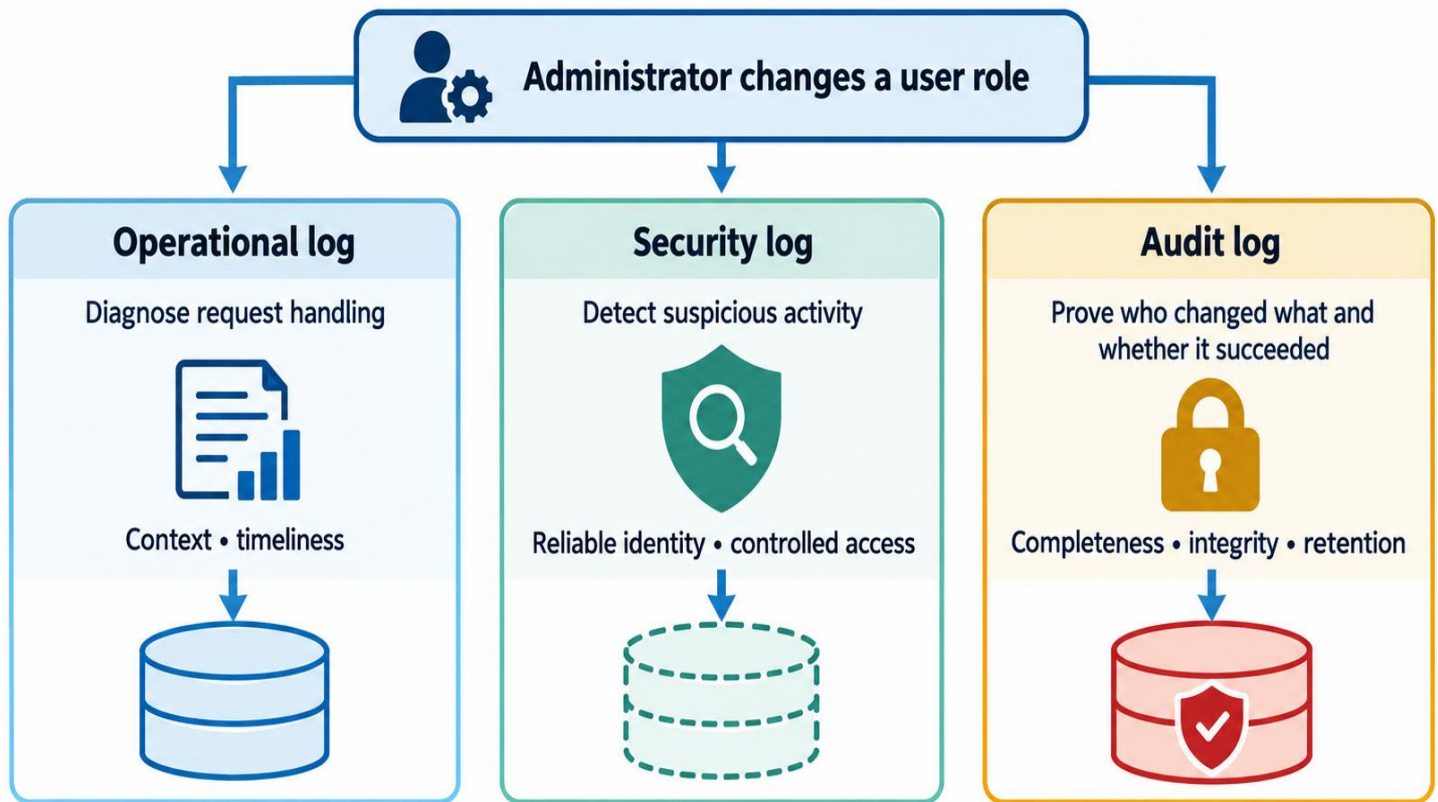
Log type	Primary purpose	Typical event	Design emphasis
Operational log	Diagnose and operate software	Dependency timeout, job completion, invalid configuration	Context, searchability, timeliness
Security log	Detect or investigate security-relevant activity	Repeated authentication failures, denied access, suspicious token use	Reliable identity, controlled access, useful detection fields
Audit log	Establish who performed an accountable action and its result	Administrator changes a user's role	Completeness, integrity, restricted access, appropriate retention

An **audit log** is a record intended to support accountability for significant actions. It often requires stronger completeness and **tamper protection**—controls that prevent or reveal unauthorized changes—than an ordinary application log because it may be used to demonstrate who changed a resource, when the change occurred, and whether it succeeded.

Retention is the period for which records are kept before archival or deletion. One event can have more than one operational meaning. An administrator changing a role may produce an operational event for troubleshooting and an audit event for accountability. Those records may have different fields, access rules, and retention periods. They should not be treated as identical merely because they describe the same underlying action.

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-3 Operational, security, and audit evidence from one underlying action



005

Figure 1-5: Operational, security, and audit evidence from one underlying action

The qualities of a useful design

A useful logging design balances several qualities:

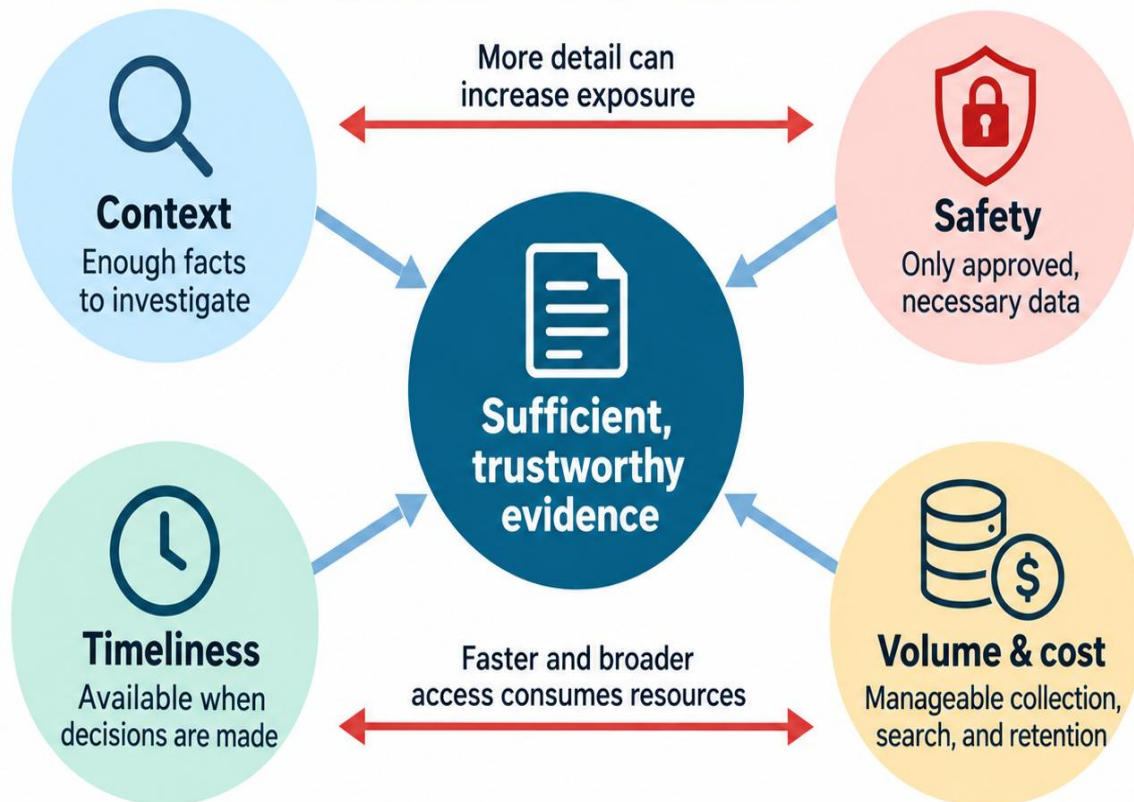
- **Relevance:** Each event supports a real operational, security, support, or accountability need.
- **Consistency:** Similar events use the same naming, field meanings, severity rules, and time format.
- **Context:** The record contains enough information to understand the event and connect it to related work.
- **Searchability:** Important facts are stored in stable fields rather than hidden in changing prose.

- **Safety:** Sensitive values are not recorded merely because they are available in memory.
- **Timeliness:** Records reach the people and systems that need them soon enough to be useful.
- **Manageable cost:** Volume, storage, **indexing**—preparing selected fields for efficient search—and query costs remain proportionate to the value of the data.

These qualities can conflict. Adding full request and response bodies might appear to improve diagnosis, but it can expose sensitive data, create excessive volume, and make useful events harder to find. Removing too much context reduces risk and cost but may make failures impossible to investigate.

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-4 Balancing context, safety, timeliness, and cost



006

Figure 1-6: Balancing context, safety, timeliness, and cost

The goal is not maximum logging. The goal is sufficient, trustworthy evidence for defined questions.

Why more logs can reduce visibility

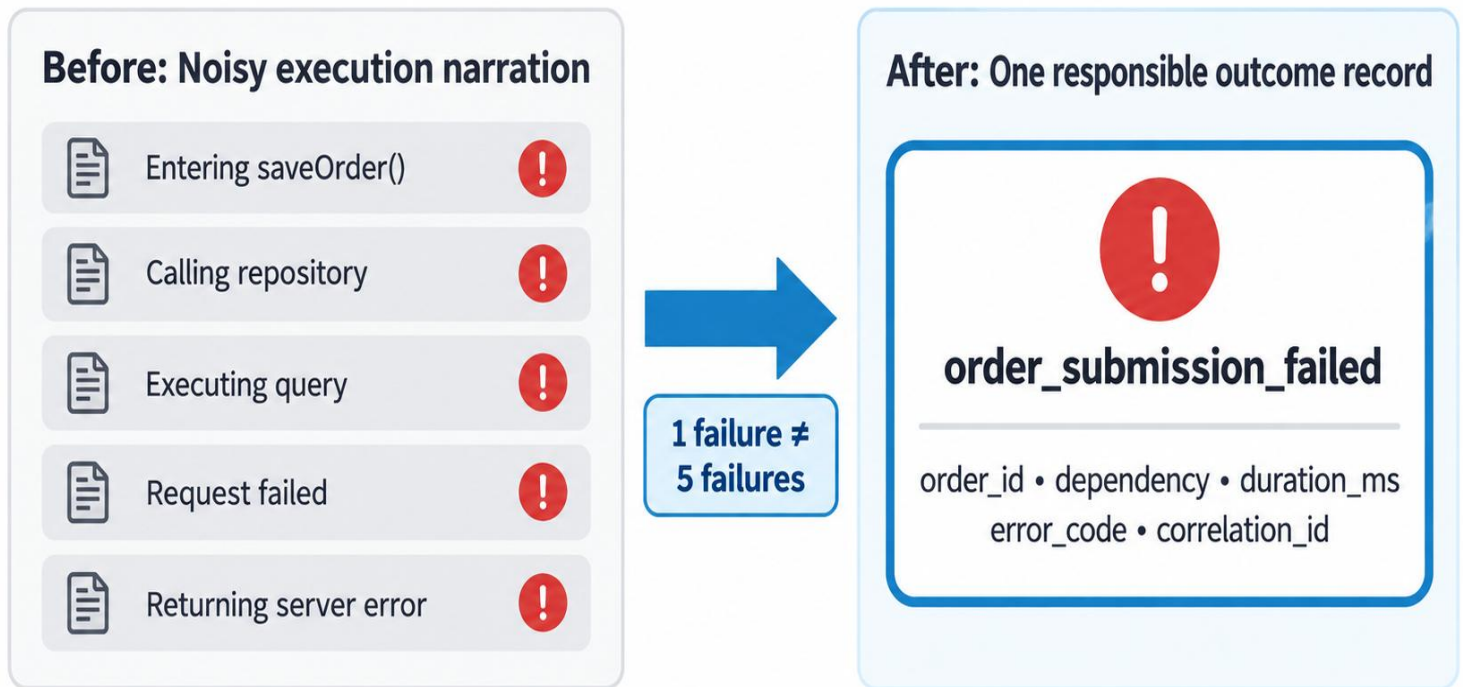
A common beginner assumption is that more records always make a system easier to understand. In practice, a high volume of repetitive or inconsistent records can hide the events that matter.

Imagine a service that writes all of the following for one failed database operation:

```
Entering saveOrder()
Calling repository
Executing query
Database call failed
saveOrder failed
Request failed
Returning server error response
```

Chapter 1: The Big Picture: Logs Are Evidence of Events

1-5 Duplicate execution narration versus one responsible outcome record



007

Figure 1-7: Duplicate execution narration versus one responsible outcome record

The sequence contains several statements but little reliable context. If each layer also records the same technical failure details, one failure may generate dozens of nearly identical entries. During a large incident, those duplicates can overwhelm search results and alerts. An **alert** is an automated notification that a defined condition requires attention.

A better design identifies the meaningful boundary. For example, the component responsible for the order operation may record one failure with the order identifier, safe error category, dependency name, duration, and correlation identifier. Lower-level diagnostic detail can be enabled when needed, but routine production evidence stays focused.

The first design question is therefore not “Where can we add a log?” It is “**Which later decision or investigation will this event support?**”

Chapter Summary

- Logs are durable evidence for questions that must be answered after an event occurs.
- Operational, security, and audit records may describe the same action but serve different purposes and controls.
- More data is not automatically better; useful logging balances context, safety, timeliness, and cost.

Review Questions

1. What is the difference between a log event and a log record?
2. Why can a large volume of repetitive records reduce visibility?
3. Which question is usually answered better by a metric or a trace than by a log?

Chapter 2 — Start with Use Cases and Requirements

What You'll Be Able to Do

- Identify the people and automated systems that will consume a log event.
- Translate an operational scenario into answerable questions, required events, and fields.
- Define testable requirements, ownership, access, retention, and trade-offs before implementation.

Logging requirements should be derived from the situations in which the records will be used. This prevents teams from recording whatever is easiest to emit while omitting the facts an investigator actually needs.

A requirement such as “add sufficient logging” cannot be tested. Two engineers may interpret it in completely different ways. A useful requirement names the scenario, the question to answer, the evidence needed, and any access or retention constraints.

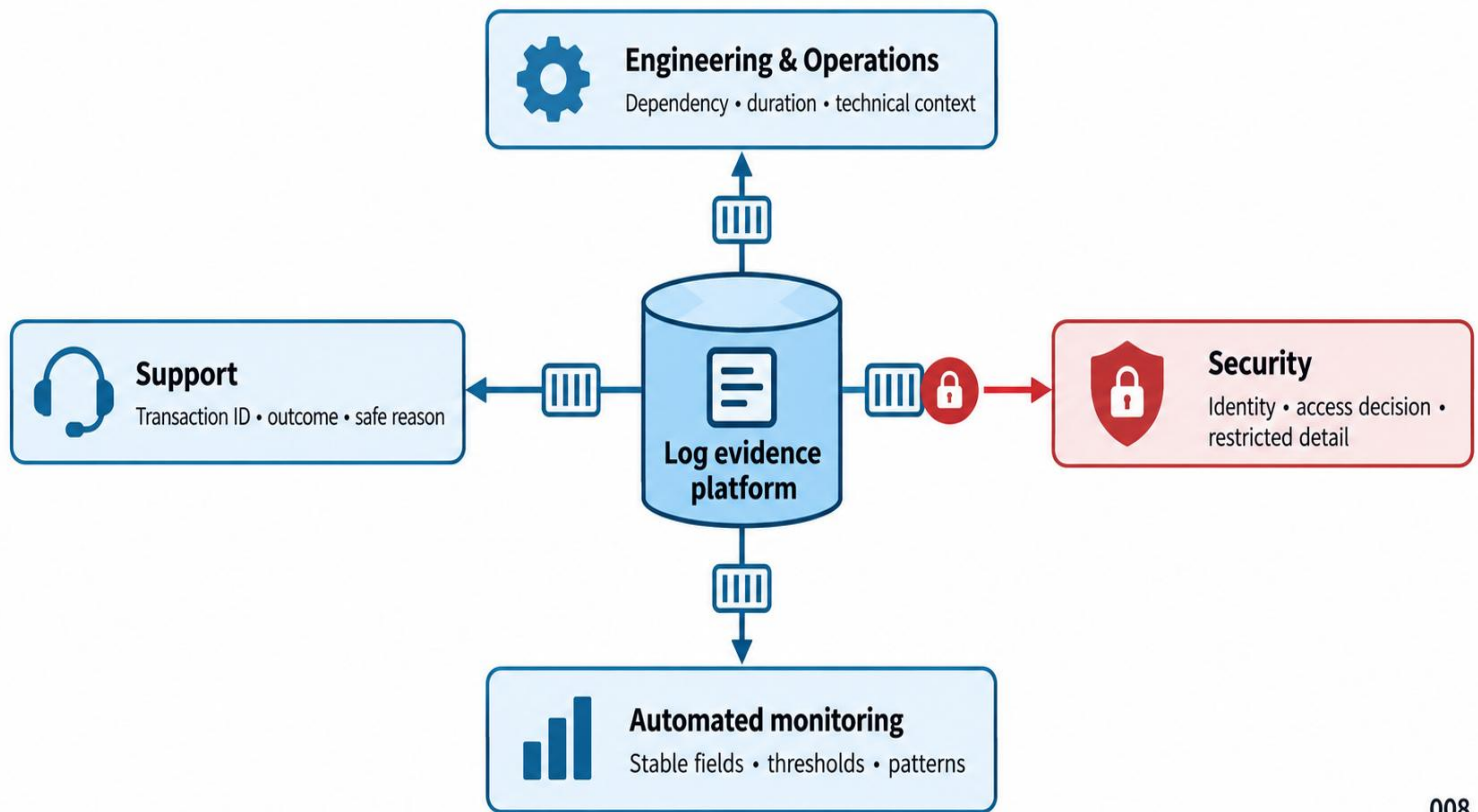
Identify the people and systems that consume logs

A **log consumer** is a person or automated process that reads or acts on log data. Common consumers include:

- Developers diagnosing defects or verifying new behavior.
- Operators investigating incidents and checking service health.
- Service desk staff researching a reported user problem.
- Security teams detecting and investigating suspicious activity.
- Auditors reviewing accountable changes.
- Monitoring systems that evaluate fields and create alerts.

Chapter 2: Start with Use Cases and Requirements

2-1 Log consumers, information needs, and access boundaries



008

Figure 2-1: Log consumers, information needs, and access boundaries

These consumers do not always need the same information. A developer may need detailed technical failure information. A service desk analyst may need a transaction identifier and a clear outcome but should not necessarily see internal security details. An automated monitor needs stable fields because changing prose is difficult to evaluate reliably.

Identifying consumers also clarifies access. A record is not automatically appropriate for every employee merely because it is useful to one team. Access should follow the purpose of the data.

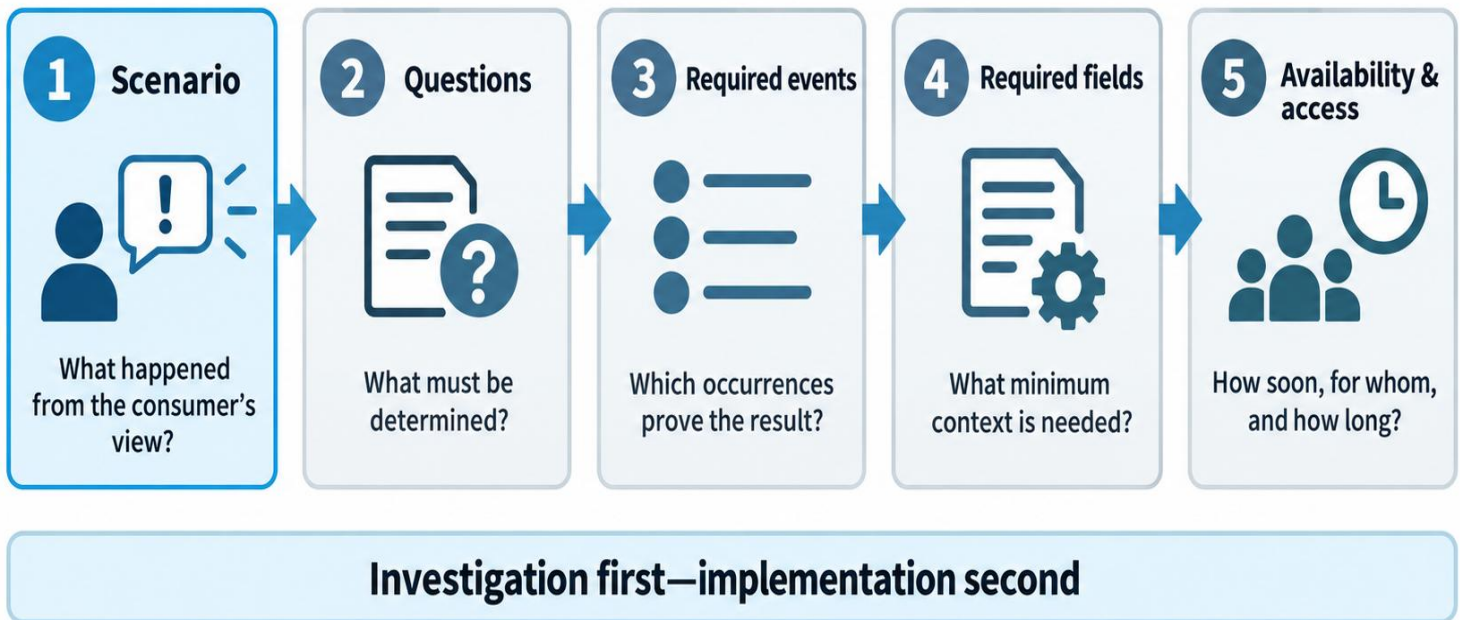
Turn a scenario into answerable questions

A practical method is to move through five steps:

1. **Describe the scenario.** State what has happened from the consumer's point of view.
2. **List the questions.** Identify what the consumer must determine.
3. **Choose the required events.** Decide which system occurrences would prove or explain the result.
4. **Choose the required fields.** Record the minimum context needed to find and interpret those events.
5. **Set availability needs.** Decide how quickly the records must be searchable, who may access them, and how long they remain useful.

Chapter 2: Start with Use Cases and Requirements

2-2 Investigation-first requirement design in five steps



009

Figure 2-2: Investigation-first requirement design in five steps

Consider the scenario: “A customer reports that an order submission failed yesterday afternoon.”

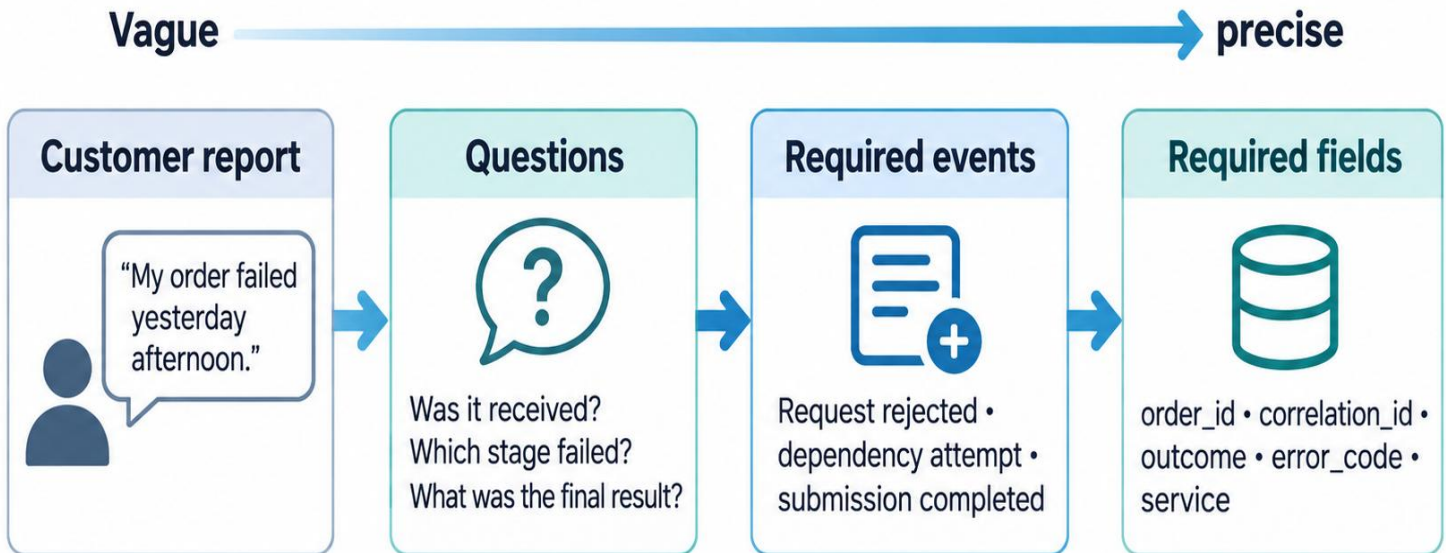
The support or engineering team may need to answer:

- Was a request received for the customer's order?
- Which order or request identifier corresponds to the report?
- Did validation reject the request, or did an internal operation fail?
- Was inventory reserved?
- Was payment attempted?
- Did the client receive a failure even though processing later completed?
- Which service produced the final outcome?

Those questions lead to candidate events such as request rejection, order submission completion, inventory reservation result, and payment result. They also reveal necessary fields: timestamp, order identifier, request or correlation identifier, service, operation, outcome, safe error code, and perhaps duration.

Chapter 2: Start with Use Cases and Requirements

2-2 From a customer report to questions, events, and fields



010

Figure 2-3: From a customer report to questions, events, and fields

The process is deliberately investigation-first. It avoids a common failure mode in which a developer records the variables close at hand but omits the business or operational identifier that someone later uses to search.

Define requirements that can be tested

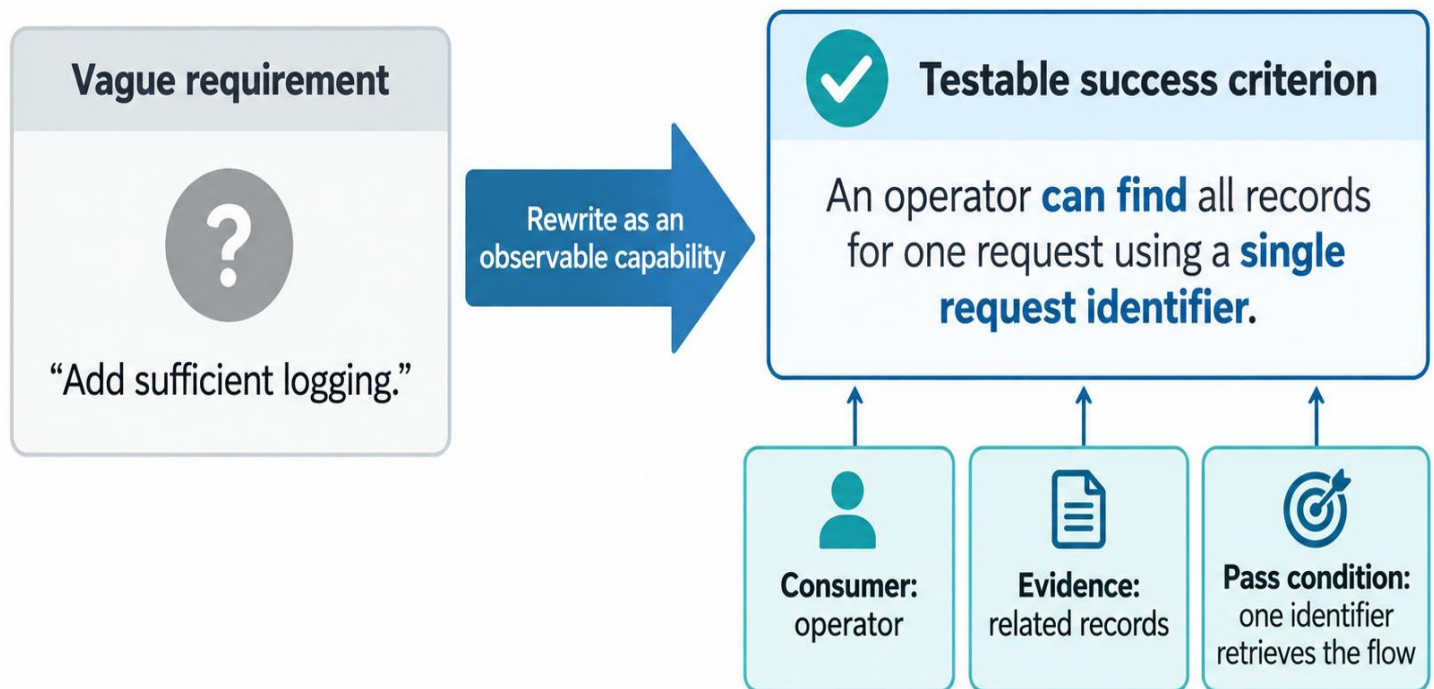
A useful requirement describes an observable capability. For example:

- An operator can find all records for one incoming request using a single request identifier.
- A service desk analyst can distinguish a client validation rejection from an internal service failure.
- A security reviewer can identify which administrator changed a role, which account was affected, and whether the change succeeded.
- A batch operator can determine how many rows succeeded, failed, and were skipped without reading one record per row.
- An engineer can tell whether a dependency timeout recovered through retry or became the final operation outcome.

Each statement can be checked against real records. These testable outcomes are **success criteria** for the logging design. By contrast, "write detailed logs" has no clear success condition.

Chapter 2: Start with Use Cases and Requirements

2-3 Vague logging request versus testable success criterion



011

Figure 2-4: Vague logging request versus testable success criterion

Requirements should also state relevant limitations. For example, "Support staff can search by internal customer identifier, but ordinary operational logs do not contain email addresses." This expresses both diagnostic value and **data minimization**, the practice of recording only the information needed for a defined purpose.

A requirement worksheet for one use case

The following compact format is useful during design reviews:

Requirement element	Order-failure example
Consumer	Service desk analyst, then on-call engineer if escalation is needed
Scenario	Customer reports an order submission failure
Questions	Was it received? Which stage failed? Was retry attempted? What was the final result?
Events	Request rejected; submission completed; dependency attempt; final submission failure
Search keys	Order ID, correlation ID, time range, service, event name
Interpretation fields	Outcome, error code, dependency, attempt number, duration
Availability	Searchable within a few minutes
Access	Support sees operational summary; engineers see technical detail according to role
Retention rationale	Long enough to cover the normal support-reporting window
Owner	Order service team

The exact values differ among organizations. The value of the worksheet is that it makes hidden assumptions visible before implementation.

Assign ownership before records become permanent

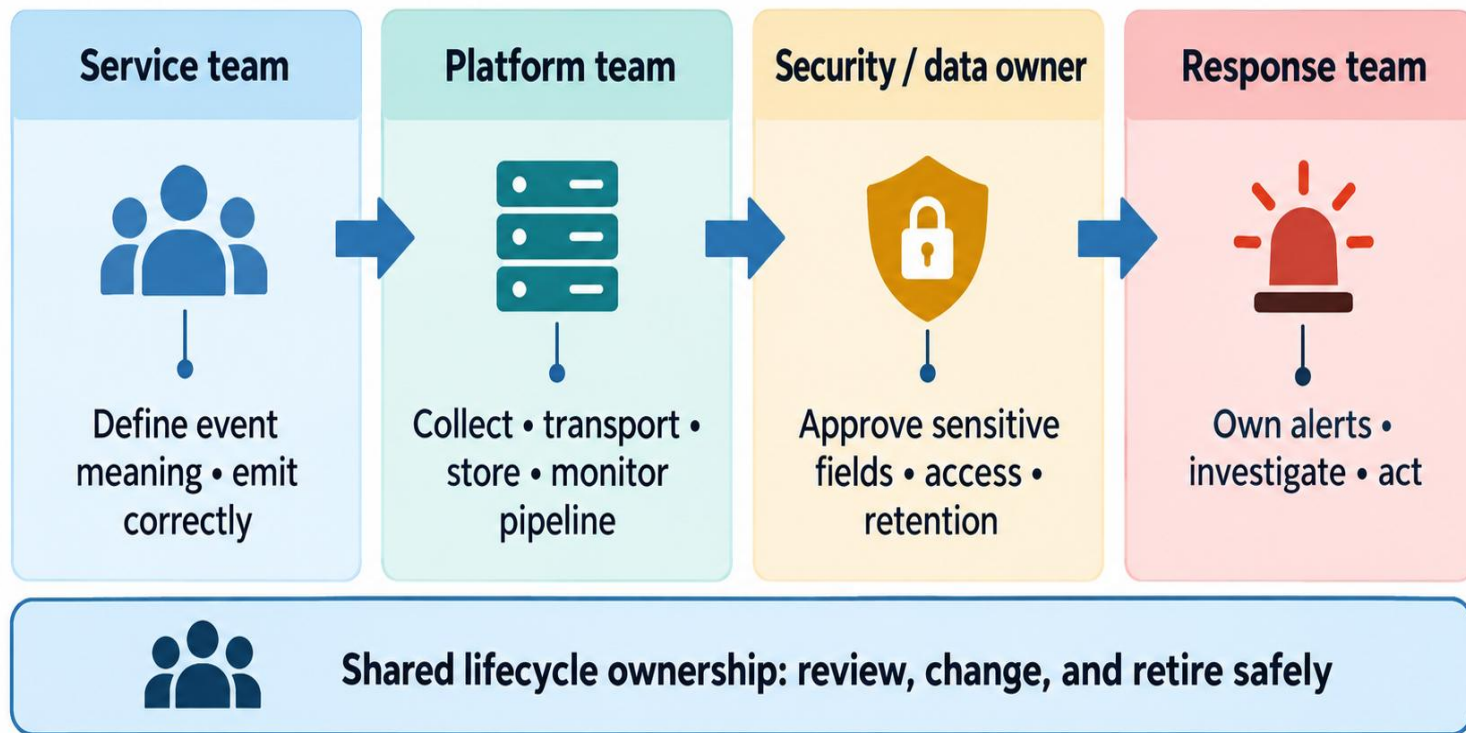
Log records tend to outlive the code change that introduced them. Event names appear in queries, **dashboards**—visual summaries of operational data—alerts, support procedures, and investigation notes. Someone must therefore own their meaning.

Ownership can be divided into several responsibilities:

- The service team defines and emits the event correctly.
- The platform or operations team runs the collection and storage path.
- The security or data owner approves handling rules for sensitive fields.
- The team named in an alert responds when the event pattern occurs.

Chapter 2: Start with Use Cases and Requirements

2-5 Ownership across the event lifecycle



012

Figure 2-5: Ownership across the event lifecycle

Ownership does not require a complicated governance process for every small application. It means that a team can answer basic questions: Who decides whether this event is still valid? Who changes its field definitions? Who notices if its volume increases unexpectedly? Who updates dependent queries when a field changes?

An event without an owner often becomes misleading. Its message may describe behavior that no longer exists, its severity may be wrong, or its fields may change while dashboards continue to assume the old meaning.

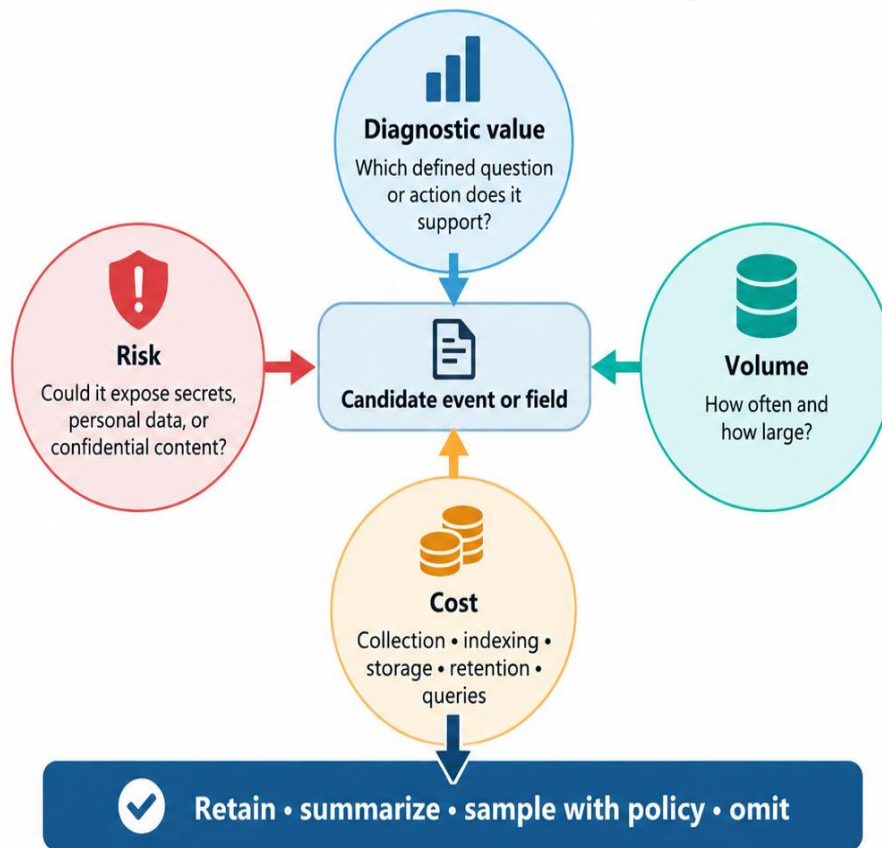
Balance value, risk, volume, and cost

Every proposed field and event has benefits and costs. A practical decision considers four dimensions:

Dimension	Question
Diagnostic value	Does this record help answer a defined question or support a defined action?
Risk	Could the data expose secrets, personal information, or confidential business details?
Volume	How often will this event occur, and how large will each record be?
Cost	What will collection, indexing, storage, retention, and query use consume?

Chapter 2: Start with Use Cases and Requirements

2-6 Value, risk, volume, and cost as a design decision



013

Figure 2-6: Value, risk, volume, and cost as a design decision

An **API (application programming interface)** is a defined way for software components to exchange requests and results. A **health check** is a small request used to verify that a service is available or ready. Suppose an API handles one million successful health-check requests per day. A detailed **INFO** record for every request may add little value because metrics already show availability and **latency**, the time required to complete a request. The same service may still need a record when a health check fails because that event helps diagnose an outage. The correct choice depends on the later question, not on the fact that both events are easy to log.

Some information can be summarized rather than retained in full. A batch job may record a completion event with processed, succeeded, skipped, and failed counts instead of writing an **INFO** event for every successful row. Errors that require item-level investigation can still include an item identifier where justified.

Sampling is another option for high-volume routine events. **Sampling** means retaining only a defined subset of events. It can preserve representative detail, but it must not be used carelessly for evidence that must be complete, such as certain audit events or final failure outcomes.

Resolve conflicts explicitly

Logging requirements sometimes conflict:

- Developers want detailed **payloads**—the data carried by requests, responses, or messages—while security rules require data minimization.
- Operators want long retention, while storage cost is limited.
- A team wants every retry recorded, while a failing dependency could generate millions of repetitive events.
- Support staff need quick access, while some technical details require tighter authorization.

Chapter 2: Start with Use Cases and Requirements

2-7 Resolving conflicting requirements with layered evidence



014

Figure 2-7: Resolving conflicting requirements with layered evidence

Do not resolve these conflicts through accidental implementation choices. State which need takes priority and how the remaining need will be handled. For example, a system might retain a safe error category and identifiers in routine logs while allowing a time-limited diagnostic mode that records additional approved fields.

The central mistake to avoid

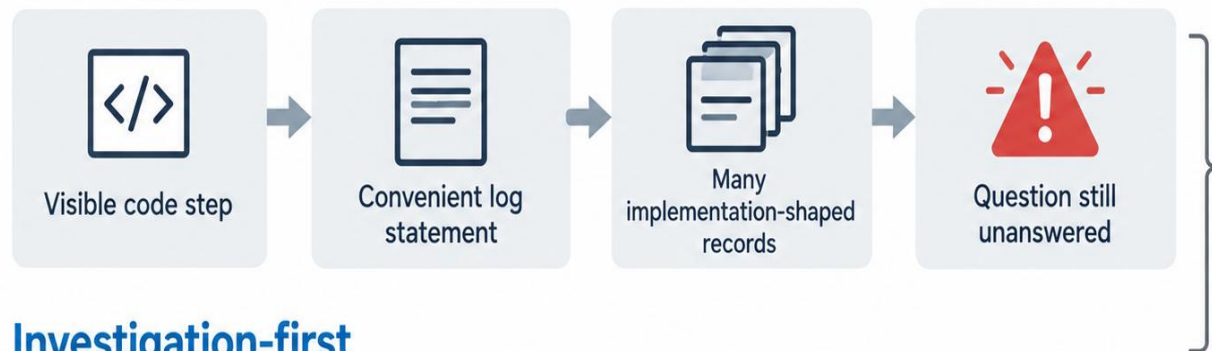
It is easy to design records around the shape of the code:

```
Entered controller  
Calling service  
Repository returned null
```

Chapter 2: Start with Use Cases and Requirements

2-8 Designing backward from a decision instead of forward from code

Code-first



Investigation-first



015

Figure 2-8: Designing backward from a decision instead of forward from code

These statements describe implementation steps, but they may not answer the consumer's question. A support analyst does not usually care which method returned `null`; the analyst needs to know whether the order existed, which operation failed, and what result was returned.

Design from the investigation backward. Start with the decision someone must make, identify the evidence that supports it, and only then decide where the application should emit the record.

Chapter Summary

- Logging requirements begin with operational use cases, not with variables that happen to be available in code.
- A useful requirement states what a consumer must be able to determine and how success will be verified.
- Ownership and explicit trade-offs prevent records from becoming stale, unsafe, or unnecessarily expensive.

Review Questions

1. How do you turn a support report into a logging requirement?
2. What makes a logging requirement testable?
3. Which responsibilities belong to the event producer, platform owner, and operational consumer?