Universal Library of Engineering Technology Monograph

ISSN: 3064-996X | Volume 3, Issue 4

Open Access | PP: 36-82

DOI: https://doi.org/10.70315/uloap.ulete.2026.0304006

Contract-Bounded Agents - A Practical Method for Building LLM Agents over Distributed Operational Pipelines

Sai Sruthi Puchakayala

Software Engineer, MS in Computer Science, University of North Texas.

Abstract

An investigation into why a unit of business work did not finish in a distributed operational pipeline crosses several systems of record: schedulers, orchestration engines, warehouses, code repositories and configuration stores. This monograph formulates Contract-Bounded Agent Design (CBAD), a reproducible engineering method for building language-model agents that conduct such investigations within fixed boundaries and can be carried from one pipeline to the next. The method rests on four structural decisions: one narrow, read-only tool per external source, whose description serves as a semantic contract; safety contracts enforced in code, including statement guards, pinned query scopes, payload ceilings and truncation reported with true totals; an explicit failure taxonomy that classifies by meaning on two orthogonal axes, the business stage at which work stopped and the concern that owns the fix; and a named separation between the current state of the work and its historical occurrence. Around the decisions the book sets an ordered investigation runbook, an answer discipline that limits each answer to what the tools returned, and four named design rules: liveness check before trust, best-effort secondary write, partial success by construction and routing by identifier shape. The decisions were derived from two systems developed separately in the author's own practice as a software engineer, an incident-investigation agent over a workflow orchestration engine and a multi-purpose assistant over a scheduled data pipeline, and the graded convergence of their designs is the principal evidence for the method. The method is positioned against research on tool-using agents, AIOps and failure management, natural-language interfaces to data, software architecture and technical debt. Its procedures are given as checklists, decision tables and rule cards, worked through on a constructed claims-processing pipeline, and the final chapter adds a transfer checklist and a staged adoption path. The book addresses software engineers, site reliability engineers and architects who build, review or adopt agents over production workflow and data systems.

Keywords: *Language-Model Agents; Tool Design; Safety Contracts; Failure Taxonomy; Incident Investigation; Operational Pipelines; AIOps; Model Context Protocol.*

PREFACE

This book is about a narrow problem that many engineering teams now face at once. An organisation has operational pipelines whose state is spread across schedulers, warehouses, repositories and configuration stores, and it wants a language-model agent that can answer questions about them: what failed, why, where the rule that rejected a record lives, and what to do next. Building such an agent once is no longer difficult. Building it so that it stays within its boundaries when the model errs, gives answers a business reader can act on, and can be carried to the next pipeline without starting over, is the subject of what follows.

The method in these pages did not begin as a method. It began as two working systems, built at different times for different teams and different kinds of pipeline, and as the observation that the second system, which reused none of the first one's code, had arrived at much of the same structure. The book states that structure as four decisions, four named rules, a runbook and an answer discipline, and it shows how far that structure recurred.

The book is written for practitioners. Every chapter that builds part of the method ends with something to use: a procedure, a checklist, a decision table or a rule card. The worked examples follow a claims-processing pipeline from intake to payment, so that each procedure can be followed step by step and adapted to a reader's own systems. The literature is cited where it supports a design choice or measures a risk, and the reference list is meant to be read as a map of the fields the method draws on.

Readers who build agents will find the order of construction

Citation: Sai Sruthi Puchakayala, "Contract-Bounded Agents - A Practical Method for Building LLM Agents over Distributed Operational Pipelines", Universal Library of Engineering Technology, 2026; 3(4): 36-82. DOI: https://doi.org/10.70315/uloap.ulete.2026.0304006.

in Chapters 2 to 5. Readers who review them will find the checklists and rule cards most useful. Readers who decide whether to adopt such systems will find the conditions of transfer and the staged adoption path in Chapter 7. All three groups share one premise of the book: the reliability of an agent placed in front of operational systems is a property of its design, and design can be written down, reviewed and repeated.

Contents

List of Abbreviations ---- 40
Introduction ---- 41
Relevance ---- 41
Aim and objectives ---- 41
Novelty and practical significance ---- 42
Structure of the monograph ---- 42
Chapter 1. Distributed Operational Pipelines and the Limits of Single-Source Investigation ---- 42
1.1 The operational pipeline as a multi-source system ---- 42
1.1.1 Business meaning as a property of the joint state ---- 42
1.1.2 Why the number of sources grows with architectural maturity ---- 43
1.1.3 The on-call engineer's working situation ---- 43
1.2 Why conventional observability stops short ---- 43
1.2.1 Alerting coupled to technical names ---- 44
1.2.2 Dashboards present, they do not investigate ---- 44
1.3 Natural-language interfaces and their boundary ---- 44
1.3.1 Text-to-SQL and the single-source assumption ---- 44
1.3.2 Retrieval-augmented generation and the multi-step enquiry ---- 44
1.3.3 Answering from one store versus conducting an ordered enquiry ---- 45
1.4 Agents with tools: new capability and new failure surface ---- 45
1.4.1 Reasoning and acting ---- 45
1.4.2 Standardised tool exposure ---- 45
1.4.3 The failure surface ---- 45
1.5 The gap this monograph addresses ---- 45
1. 5.1 Implementation reports versus a design method ---- 45
1.5.2 Four structural decisions ---- 46
Chapter 2. The Tool Layer: Narrow Sources, Explicit Contracts ---- 47
2.1 One narrow tool per external source ---- 47
2.1.1 The alternative and why it fails ---- 47
2.1.2 Scoping rules ---- 47
2.1.3 How the tool count should grow ---- 47
2.1.4 What narrow tools cost ---- 47
2.2 The tool description as a semantic contract ---- 48
2.2.1 What a description must carry ---- 48
2.2.2 Routing intelligence belongs in descriptions ---- 49
2.2.3 Writing for a non-deterministic reader ---- 49
2.3 Tool description debt ---- 49
2.3.1 The four forms and their symptoms ---- 49
2.3.2 Why the debt accumulates silently ---- 50
2.3.3 Remedies ---- 50
2.4 Composite templates as investigation primitives ---- 50

2.4.1 Which sequences deserve a template ---- 51
2.5 Procedure: designing a tool layer for your own system ---- 51
2.5.1 From an inventory of sources to a tool catalogue ---- 51
2.5.2 Review checklist for a tool layer ---- 51
Chapter 3. Safety Contracts Enforced in Code ---- 51
3.1 The principle: never delegate a boundary to the prompt ---- 51
3.1.1 Why prompt-level instruction is insufficient ---- 51
3.1.2 The division of responsibility ---- 52
3.1.3 What code enforcement costs ---- 52
3.2 Statement and scope guards ---- 52
3.2.1 Restricting operations to a permitted class ---- 52
3.2.2 Pinning scope ---- 53
3.2.3 Layering independent guards ---- 53
3.3 Explicit ceilings on every external payload ---- 54
3.3.1 Caps set in code ---- 54
3.3.2 The missing timeout as a design defect ---- 54
3.3.3 Unbounded collection endpoints ---- 54
3.4 Truncation honesty ---- 54
3.5 Liveness and identity resolution before action ---- 55
3.5.1 Confirming that the entity is still live ---- 55
3.5.2 Resolving schema and identity before composing a query ---- 55
3.6 Claimed resilience is not implemented resilience ---- 55
3.6.1 The audit procedure ---- 55
3.6.2 Why documentation and implementation diverge ---- 55
3.7 Validating an agentic system ---- 56
3.7.1 What unit tests cover when the model is in the loop ---- 56
3.7.2 The evaluation gap ---- 56
3.7.3 Declaring known gaps ---- 56
Chapter 4. Failure Classification: Two Orthogonal Taxonomies ---- 57
4.1 Why exception class is the wrong axis ---- 57
4.1.1 Technical signal names drift ---- 57
4.1.2 One exception, many causes and owners ---- 57
4.2 Classification by business milestone ---- 57
4.2.1 The translation table as a first-class artefact ---- 57
4.2.2 Sizing the taxonomy ---- 58
4.2.3 The explicit unknown bucket ---- 58
4.3 Classification by owning concern ---- 58
4.3.1 Classifying by who owns the fix ---- 58
4.3.2 Ordered evaluation, most specific first ----59
4.4 The two axes together ---- 59
4.4.1 Where the work died and who fixes it ---- 59
4.4.2 Routing hand-offs with the classification attached ---- 60
4.4.3 What the two axes cost ---- 60
4.5 Genuine failure versus dependency failure ---- 60
4.6 Current state versus historical occurrence ---- 60

4.6.1 Two semantics behind one phrasing ---- 60
4.6.2 An explicit trigger test, and a safe default ---- 60
4.7 Procedure: building a taxonomy for your own pipeline ---- 61
4.7.1 Derivation steps ---- 61
4.7.2 Sizing guidance and maintenance rules ---- 61
4.7.3 A worked illustrative example ---- 61
Chapter 5. The Diagnostic Runbook and Answer Discipline ---- 62
5.1 Entry by business key ---- 62
5.2 The ordered sequence and its short circuits ---- 62
5.2.1 Terminal-state gate first ---- 62
5.2.2 Ordering the remaining checks ---- 62
5.2.3 Structured collection before reasoning ---- 63
5.2.4 A worked trace ---- 63
5.3 Evidence extraction ---- 64
5.3.1 A bounded window around the evidence ---- 64
5.3.2 Choosing window size and match limits ---- 64
5.4 From evidence to recommendation ---- 64
5.4.1 Mapping signatures to suggestions ---- 64
5.4.2 Retry semantics as a deliberate decision ---- 65
5.4.3 Engine-specific routing ---- 65
5.5 Answer discipline ---- 65
5.5.1 Call a tool, never guess ---- 65
5.5.2 Absence of data is a valid answer ---- 65
5.5.3 Form of the answer ---- 65
5.5.4 Always return a verification path ---- 65
5.6 The runbook as a human checklist ---- 66
Chapter 6. Named Design Rules ---- 66
6.1 Liveness check before trust ---- 67
6.2 Best-effort secondary write ---- 67
6.3 Partial success by construction ---- 68
6.3.1 Degradation rules ---- 68
6.4 Routing by identifier shape ---- 69
6.4.1 Boundaries of the rule ---- 69
6.5 Naming implicit rules in your own codebase ---- 69
6.5.1 How the four rules interact ---- 70
Chapter 7. Convergence, Transfer, and Adoption ---- 70
7.1 Two systems, two domains, independent construction ---- 70
7.1.1 System A ---- 70
7.1.2 System B ---- 70
7.1.3 Why the comparison is meaningful ---- 70
7.2 The convergence claim ---- 71
7.2.1 Point-by-point comparison ---- 71
7.2.2 What the comparison shows ---- 71
7.2.3 What this evidence is and is not ---- 72
7.3 Where the systems differed, and why ---- 72

7.3.1 Variation attributable to domain ---- 72
7.3.2 Variation attributable to sequence ---- 72
7.4 Conditions of transfer ---- 72
7.4.1 Properties of a target environment ---- 72
7.4.2 Where the conditions hold, and where they do not ---- 73
7.4.3 A worked transfer check ---- 73
7.5 Adoption path ---- 73
7.5.1 From a bespoke assistant to a reusable tool layer ---- 73
7.5.2 What to build first and what to defer ---- 74
7.5.3 Review gates ---- 74
Conclusion ---- 75
The method as a whole ---- 75
What the convergence supports ---- 75
Open problems ---- 75
Boundaries of the claims ---- 76
Guidance for applying the method ---- 76
Declarations ---- 76
Glossary ---- 76
References ---- 77

LIST OF ABBREVIATIONS

AI artificial intelligence

AIOps artificial intelligence for IT operations

API application programming interface

CBAD Contract-Bounded Agent Design

CPU central processing unit

DNS Domain Name System

DORA DevOps Research and Assessment

HTTP Hypertext Transfer Protocol

IT information technology

JSON JavaScript Object Notation

LLM large language model

MCP Model Context Protocol

ML machine learning

OWASP Open Worldwide Application Security Project

RFC Request for Comments

SQL Structured Query Language

TLS Transport Layer Security

UTC Coordinated Universal Time

UUID universally unique identifier

W3C World Wide Web Consortium

INTRODUCTION

Relevance

Large language model agents are moving from experimental assistants to components placed in front of production systems. LangChain (2024), a vendor of agent frameworks, surveyed more than 1,300 professionals in 2024, and about 51% of its respondents reported using agents in production. Adoption is uneven and trust is limited: in the 2025 Stack Overflow developer survey, a majority of respondents (52%) either did not use agents or used only simpler AI tools (Stack Overflow, 2025), and in the 2024 DORA survey of technology professionals, 39.2% of respondents reported little or no trust in the quality of AI-generated code, while 81% reported that their organisations had shifted priorities to increase the incorporation of AI into their applications (DORA, 2024). The pressure to place agents in front of operational systems is real, and so is the doubt about whether they can be trusted there.

The change in risk profile is specific. An assistant that drafts text can be wrong without consequence until a person reads it. An agent that queries a warehouse, reads a scheduler, retries a failed process or reports what is broken acts on live systems and on the decisions of people who rely on its answers. Its errors reach production before a person reviews them.

The operational reality that such agents are asked to address is an investigation across several systems. An engineer asked why a unit of business work did not finish must consult a scheduler for the run, a warehouse for the data, a repository for the rule that rejected a record and a configuration store for the setting in force, each with its own schema and vocabulary. No single system holds the answer, because the condition that matters to the business is defined over several of them together.

The problem grows with architectural maturity. Service decomposition and event-driven designs distribute an execution path that once sat in one process across many owners and stores, so the number of sources an investigation must consult rises as architectures mature. In an incident study of one large cloud service, most high-severity incidents had root causes other than code bugs, in infrastructure, deployments and service dependencies (Ghosh et al., 2022).

The published work on agents does not yet answer the question an engineering team faces when it decides to build one over its own pipeline. Chapter 1 reviews that work and locates the missing piece: a procedure for building an agent over heterogeneous operational sources that keeps the agent's boundaries intact when the model misbehaves.

AIM AND OBJECTIVES

The aim of this monograph is to formulate a reproducible engineering method for designing agent systems over distributed operational pipelines. The method is called Contract-Bounded Agent Design (CBAD) and rests on four structural decisions: narrow read-only tools, one per external source; safety contracts enforced in code; an explicit failure taxonomy that classifies by meaning on two orthogonal axes; and a named separation between current state and historical occurrence. The decisions were derived from two systems developed separately and without shared code, both designed and built in the author's own practice as a software engineer: System A, an incident-investigation agent over a workflow orchestration engine, and System B, a multi-purpose assistant over a scheduled data pipeline. The graded convergence of the two designs is the principal evidence for the method: the tool layer recurs in the same form, the safety contracts are explicit in System B and partial in System A, the failure taxonomy appears as two complementary axes, one in each system, and the separation of current state from historical occurrence is explicit in System B only.

The derivation proceeded from structure to principle. The designs of both systems were read decision by decision, separating what each system's code enforces from what its documentation states, and every recurring structure was abstracted to the level of a class of problem: a tool per source, a guard before a request, a table that translates a signal, a check before an action. The abstracted structures were then positioned against five groups of literature: agent architecture and tool use; AIOps, observability and failure management; natural-language interfaces to data; software architecture foundations, including information hiding, the separation of mechanism from policy and ports and adapters; and technical debt. Standards and vendor documentation for the protocols, platforms and security practices that the method touches complete the reference base. Candidate works were found by searches of the OpenAlex and arXiv indexes for each group, supplemented by the standards bodies' and vendors' own documentation, and every work on which a specific statement rests was read in full. The worked examples use a single claims-processing pipeline throughout the book.

The objectives, each met by one chapter, are:

1. to establish the problem space and the limits of existing approaches, from observability to natural-language interfaces and tool-using agents (Chapter 1);

2. to define the tool layer and the contracts that make it maintainable, including the debt that accumulates in tool descriptions (Chapter 2);

3. to specify safety enforcement that does not depend on model behaviour, and the practices that verify it exists (Chapter 3);

4. to construct the failure classification model as two orthogonal taxonomies, together with the separation of current state from historical occurrence (Chapter 4);

5. to formalise the diagnostic runbook and the discipline that governs what an answer may say (Chapter 5);

6. to name the recurring design rules that the systems follow without having stated them (Chapter 6);

7. to examine how the decisions recur across the two systems, state what that recurrence supports, and set out the conditions of transfer and a staged adoption path (Chapter 7).

Novelty and Practical Significance

The novelty of the work lies in integration and in naming. The tools, guards, classifications and runbook steps each have precedents; the method composes them into a procedure with a fixed order of decisions and supports it with the convergence of two systems built separately, in different domains and for different teams. Four design rules that practitioners follow implicitly are stated as named rules with conditions of application: liveness check before trust (confirming that an entity is still live before acting on a stored record about it), best-effort secondary write (letting an auxiliary write-back fail without failing the user's action), partial success by construction (reporting per-item outcomes of bulk operations) and routing by identifier shape (dispatching requests by the form of their identifiers). Failure classification by business milestone and by owning concern is presented as a pair of orthogonal axes over one set of failures. The treatment of tool description debt, the gap that opens when a tool's description drifts from its behaviour, as a maintenance liability extends a concept from earlier published work (Puchakayala, 2026c).

The practical significance lies in the form of the method. It is set out as procedures, checklists, decision tables and rule cards that an engineer can apply to a pipeline and that a reviewer can use in design review: a tool-layer checklist (Chapter 2), a minimum safety contract and a decision table for payload ceilings (Chapter 3), translation and signature tables with a derivation procedure (Chapter 4), an investigation flow and an answer checklist (Chapter 5), four rule cards (Chapter 6) and a transfer checklist with an adoption path (Chapter 7).

Structure of the Monograph

Chapter 1 describes the operational pipeline as a multi-source system and shows why metrics, logs, traces, dashboards, text-to-SQL, retrieval-augmented generation and unconstrained tool-using agents each stop short of a cross-source investigation; it ends with the four structural decisions. Chapter 2 develops the first decision: one narrow tool per external source, the tool description as a semantic contract, tool description debt and composite templates, which capture a recurring sequence of tool calls as one named tool. Chapter 3 develops the second: guards that hold regardless of the model, payload ceilings, truncation honesty, liveness and identity checks, the audit of claimed resilience and the validation of an agentic system. Chapter 4 develops the third and fourth decisions: classification by business milestone and by owning concern, the separation of genuine from dependency failures, and the two semantics of a failure question. Chapter 5 formalises the investigation sequence and the answer discipline, and restates both for human operators. Chapter 6 states the four named rules in a fixed card format and gives a procedure for finding such rules in other codebases. Chapter 7 presents the convergence evidence, the differences between the two systems, the conditions of transfer and the adoption path. The Conclusion draws the method together and names the problems it leaves open.

CHAPTER 1. DISTRIBUTED OPERATIONAL PIPELINES AND THE LIMITS OF SINGLE-SOURCE INVESTIGATION

An engineer who is asked why a piece of business work did not finish overnight rarely finds the answer in one place. The scheduler knows that a task failed, the warehouse knows which records arrived and which did not, the source repository knows where the business rule that rejected them lives, and the versioned configuration knows which threshold was in force that night. Each system answers its own question well. None of them answers the question the engineer was asked. The gap has a structure, it widens as architectures mature, and the established families of tools, natural-language interfaces and tool-using language model agents included, each stop short of closing it.

1.1 The Operational Pipeline as a Multi-Source System

An operational pipeline, in the sense used throughout this book, is a chain of stages that carries a unit of business work from intake to a finished state. The stages are owned by different services, triggered by one or more schedulers or workflow engines, and maintained by different teams. A unit of work leaves traces in the systems it touches: a run record in the scheduler, rows in one or more warehouse tables, log lines in each executing container, a version of the configuration that parameterised the run, and the source code of every business rule applied to it.

Data-system design treats the coexistence of relational stores, batch and stream processors and message brokers inside one application as the normal case, each with its own trade-offs of consistency, scalability and fault tolerance (Kleppmann, 2017). A useful distinction separates a system of record, which holds the authoritative version of a fact, from derived systems that hold transformed or cached copies of it. An operational pipeline has several systems of record at once. The scheduler is authoritative for whether a run happened, the warehouse for what data the run produced, the repository for what logic the run executed. A question about the work is a question about how these authorities relate at a given moment.

1.1.1 Business Meaning as a Property of the Joint State

The conditions that matter to a business reader are defined over several of these systems together. A formalisation from earlier work treats such a condition as a k-ary cross-system

invariant: a business predicate whose evaluation needs projections from at least k systems, so that no observer confined to a single system can decide it once k is two or more (Puchakayala, 2026c). A constructed example makes the point concrete. In a claims-intake pipeline, the statement "every claim accepted at intake yesterday has a matching entry in the payment ledger" needs the intake system's acceptance records, the ledger's entries and the scheduler's record of whether the reconciliation job ran. Each system can report that it is healthy while the predicate is false.

The data-pipeline literature describes the same structure from the side of defects. Foidl et al. (2024) derived a taxonomy of 41 factors that influence whether a data pipeline delivers quality data, validated through expert interviews, and then mined GitHub issues from 11 projects: in a random sample of 400 issues, only about 10% were data-related, and a further sample of 200, drawn after issues matching non-data keywords had been filtered out, brought the set to 100 data-related issues. Among those 100, incorrect data types were the most frequent root cause (33%), and the issues arose mostly in the cleaning (35%) and ingestion (34%) stages. Of 400 sampled Stack Overflow posts, 99 concerned data-pipeline processing, and in those, data integration and ingestion were the most frequent topics, at nearly half (47%) of the questions. Integration and ingestion are the stages where one system's output becomes another system's input, which is where a joint-state condition first breaks.

1.1.2 Why the Number of Sources Grows with Architectural Maturity

Service decomposition distributes an execution path that used to sit inside one process across many owners. Microservice architecture, which grew out of service-oriented computing (Dragoni et al., 2017), moves coordination out of a shared memory and into the network and the operational tooling around it. A review of monolith-to-microservices migration concluded that migration redistributes complexity across service boundaries and coordination mechanisms, and that coordination overhead grows with the number of services (Puchakayala, 2026b). Event-driven designs add asynchronous hand-offs, where the state of a unit of work at any moment is spread across producers, brokers and consumers, each with its own retention and ordering guarantees (Puchakayala, 2026a).

Incident studies show the consequence. Ghosh et al. (2022) analysed 152 high-severity production incidents of one large cloud service and categorised their root causes: code bugs were the largest single category (27.0%), while infrastructure, deployment and service dependencies together accounted for a majority (60%). Dependency failures alone made up 16.4% of the incidents in that study. A failure whose cause sits in a neighbouring service is, by construction, a failure that the failing service's own telemetry cannot explain. Gunawi et al. (2016) reached a related observation from public material: across 1,247 news reports and post-mortems describing 597 unplanned outages of 32 internet services between 2009 and 2015, 355 of the 597 outages were reported without a stated root cause. Public reports are a weak proxy for internal investigation, but the gap between "an outage occurred" and "we know why" is visible even there.

1.1.3 The On-Call Engineer's Working Situation

For the engineer on call, the result is a translation problem before it is a diagnostic one. The same unit of work carries a business key in the warehouse, a run identifier in the scheduler, a task name in the logs and a path in the repository. Systems built at different times by different teams use different words for the same stage of work and the same word for different things. Investigation starts with cross-referencing: resolving a name the business user knows into the identifiers each system needs, and then reading each system in an order that narrows the question. That ordered, multi-source enquiry is the activity the rest of this book designs for.

1.2 Why Conventional Observability Stops Short

Modern observability rests on three signal types. The OpenTelemetry project defines traces, metrics and logs as its core signals and standardises how they are produced and exported (OpenTelemetry Authors, n.d.). Each answers a different question well.

Metrics answer "is the system within its normal envelope?" The site reliability engineering literature recommends four golden signals for user-facing systems (latency, traffic, errors and saturation) and argues that alerts should fire on symptoms that users experience, with causes left to investigation (Ewaschuk, 2016). That division is sound, and it is also the point where metrics stop: an error-rate alert says that something is wrong with a service and leaves open what happened to a particular unit of work.

Logs answer "what did this process say while it ran?" They carry the detail an investigation needs, at a volume no person reads in full. The automated log-analysis literature has developed parsing, anomaly detection and failure prediction to make that volume usable (He et al., 2021), and log parsers convert free text into templates that can be counted and compared (Zhu et al., 2019). These techniques operate on one log stream at a time.

Traces answer "which services did this request pass through, and where did it spend its time?" Dapper, the tracing system that shaped the field, propagates a trace context through instrumented remote procedure calls and collects the resulting spans centrally with low overhead (Sigelman et al., 2010). The W3C Trace Context Recommendation standardises how that context crosses service boundaries in HTTP headers (W3C, 2021). Tracing assumes a request that carries its context from hop to hop. A nightly batch run triggered by a scheduler, reading a table that another job

wrote six hours earlier, breaks that assumption: the causal link between the two jobs lives in the scheduler's dependency graph and in the data, where no header travels.

1.2.1 Alerting Coupled to Technical Names

Rule-based alerting turns signals into notifications by matching names and thresholds. Alert rules written against low-level task names, topic names or exception classes inherit every rename and refactoring of the code beneath them. Volume compounds the problem. Zhao et al. (2020) studied three years of alert data from the online service systems of a large commercial bank: in 96.87% of minutes the number of alerts was at most ten, while in 0.17% of minutes hundreds to thousands of alerts arrived at once, and 166 such alert storms were identified from historical failure tickets. An alert storm is the moment when the on-call engineer most needs a single explanation and receives the most fragments.

Ghosh et al. (2022) found that existing monitors detected roughly half of the 152 incidents in their study, and that simple measures such as additional telemetry, fixes to existing monitors, adjusted thresholds or new monitors on existing telemetry could raise the effectiveness of automated monitoring by 50%. The finding concerns detection. For the step after it, the authors observed that incidents in some root-cause categories are mitigated quickly once their category is known, and they pointed to tools that, given an incident, quickly identify its coarse-grained root-cause category.

1.2.2 Dashboards Present, they do not Investigate

Dashboards assemble metrics and log queries into a view. They show that a value moved and when it moved. They do not decide which system to consult next, and they cannot resolve the gap that Huang et al. (2017) call gray failure: a form of differential observability in which at least one application observes that a system is unhealthy while the system's own monitoring observes that it is healthy. Gray failure is a precise description of the operational question this book addresses, because the "observer" in a pipeline is each component's local view and the "application" is the business process that depends on all of them.

Research on artificial intelligence for IT operations (AIOps) has built methods on top of these signals. One survey of AIOps for failure management sorted 100 solutions into 5 categories and 14 subcategories by when they intervene and which problem they solve (Notaro et al., 2021). The arrival of large language models has extended that work toward incident summarisation and diagnosis (Zhang et al., 2025). Most of these methods take one signal family as input and produce a ranking, a score or a summary. The investigation that connects a business question to several systems of record stays outside their scope.

1.3 Natural-Language Interfaces and their Boundary

Natural-language interfaces promise to remove the translation step by letting a person ask in ordinary words. Two families dominate: text-to-SQL, which compiles a question into a query over one database, and retrieval-augmented generation, which grounds a model's answer in retrieved documents.

1.3.1 Text-to-SQL and the Single-Source Assumption

Text-to-SQL research has measured progress against benchmarks of increasing realism. Spider introduced a large cross-domain benchmark of questions over many databases (Yu et al., 2018). BIRD moved closer to practice with 12,751 questions over 95 databases totalling 33.4 GB across 37 professional domains; on it, GPT-4 reached 54.89% execution accuracy against a human result of 92.96% (Li et al., 2023). In the authors' analysis of ChatGPT's failures on BIRD, wrong schema linking accounted for 41.6% of the errors, meaning that the model understood the database structure and still attached the question to the wrong tables or columns.

Benchmarks built from enterprise workloads lower these numbers further. Spider 2.0 collects 632 text-to-SQL workflow problems from enterprise use cases, with schemas averaging 812 columns; an agent framework built on o1-preview solved 21.3% of its tasks, compared with 91.2% on the original Spider and 73.0% on BIRD (Lei et al., 2025). BEAVER, assembled from real query logs, contains 9,128 question-SQL pairs over enterprise databases averaging 101.5 tables and 869.4 columns; agentic frameworks using GPT-5.2 achieved 10.8% accuracy, rising to 30.1% when the annotated subtasks were supplied as oracle hints (Chen et al., 2026).

These results concern questions that have an answer inside one database. The joint-state question of a claims-intake pipeline does not: whether a reconciliation job ran is a fact in the scheduler, and whether the rule that rejected a record changed last week is a fact in the repository. Text-to-SQL assumes a single source by design. Inside that source it struggles with schema scale; across sources it has nothing to compile into.

1.3.2 Retrieval-Augmented Generation and the Multi-Step Enquiry

Retrieval-augmented generation pairs a generator with a retriever over a document index, so that output is conditioned on retrieved evidence as well as on what the model memorised in training (Lewis et al., 2021); later work has catalogued many refinements of retrieval, generation and augmentation (Gao et al., 2024). Grounding reduces unsupported statements. It does not perform an investigation, because a single retrieval step fetches passages similar to the question, while an investigation needs the answer to its first query to decide its second.

Trivedi et al. (2023) state the multi-step condition: what to retrieve depends on what has already been derived, which in turn may depend on what was previously retrieved. Their

method interleaves retrieval with chain-of-thought steps and improved retrieval recall by 11 to 21 points over one-step, question-based retrieval on four multi-hop question-answering datasets. The gain shows that ordering matters. It was obtained over document collections, where every step queries the same kind of store. An operational enquiry queries different kinds of systems, each with its own interface, cost and failure behaviour.

1.3.3 Answering from One Store versus Conducting an Ordered Enquiry

The boundary of both families can be stated as a pair of requirements. An operational enquiry crosses several systems of record, and it is ordered: the result of each step decides whether the next step is needed and which system it consults. Text-to-SQL meets neither requirement. Retrieval-augmented generation can be made iterative, but its steps act on an index; run state, configuration and code reside in live systems that the index only copies, if it holds them at all.

1.4 Agents with Tools: New Capability and New Failure Surface

1.4.1 Reasoning and Acting

Language model agents meet the ordering requirement. ReAct interleaves reasoning traces with actions, so that the model decides which tool to call, observes the result and plans the next step from it (Yao et al., 2023). Toolformer showed that a model can learn when and how to call external tools from self-generated examples (Schick et al., 2023). Gorilla connected a model to a large collection of machine-learning APIs and measured its tendency to invent API calls (Patil et al., 2024). ToolLLM built an instruction set over 16,464 real-world APIs to train and evaluate tool use at that scale (Qin et al., 2023). Surveys of language model agents organise this work around planning, memory and action modules (Wang et al., 2024a; Xi et al., 2025), and surveys of tool learning trace the stages of task planning, tool selection, tool calling and response generation (Qu et al., 2025).

Operations research has applied these agents to incident investigation. Roy et al. (2024) evaluated a ReAct agent for root-cause analysis on incidents from a large cloud provider, on a random sample of 500 incidents from the 3,000-incident test split of a collection of 107,000. RCACopilot combined automated diagnostic workflows with a model that predicts the root-cause category, and reached a micro-F1 of 0.766 on a one-year dataset of 653 incidents from one service (Chen et al., 2024). AIOpsLab built a framework of 48 problems for evaluating operations agents; across 288 evaluated cases, the best agent reached 59.32% accuracy with a limit of 20 steps, and the weakest configuration 15.25% (Chen et al., 2025). Riddell et al. (2026) executed 48,000 simulated failure scenarios and produced a labelled taxonomy of 16 recurring reasoning failures of models performing root-cause analysis.

1.4.2 Standardised Tool Exposure

The Model Context Protocol (MCP) standardises how an application exposes tools to a model: a server declares each tool with a name, a natural-language description and a JSON schema for its input, and clients discover and call tools through a common interface (Model Context Protocol, 2025b). The protocol's revision of March 2025 added tool annotations that describe behaviour, such as whether a tool is read-only or destructive (Model Context Protocol, 2025a). Standardisation changes maintainability in one important respect: a tool written once can serve several consumers, so that the design of the tool becomes the unit of engineering, and each consumer inherits it.

1.4.3 The Failure Surface

The same capability opens a new failure surface, which the remaining chapters treat one part at a time.

1. **Non-deterministic tool selection.** The model chooses tools from their descriptions; two runs of the same question can take different paths, and a vague or overlapping description shifts the choice (Chapter 2).

2. **Fabricated output.** Models produce fluent statements unsupported by their input, a behaviour surveyed under the name hallucination (Huang et al., 2025). In an investigation, a fabricated field or an invented status is worse than no answer (Chapter 5).

3. **Unbounded payloads.** A tool that returns whatever the upstream system returns can flood the context window or the upstream system itself (Chapter 3).

4. **Actions on stale state.** A record retrieved from a secondary store can describe a process that has since finished or been cancelled; acting on it repeats or contradicts work (Chapters 3 and 6).

The protocol ecosystem already shows these risks at scale. Hou et al. (2026) analysed the MCP server lifecycle and identified 16 threat scenarios across four types of attacker. Hasan et al. (2026) evaluated 1,899 open-source MCP servers and found general vulnerabilities in 7.2% of them, MCP-specific tool poisoning in 5.5%, critical or blocker-level code smells in 66% and critical or blocker-level bugs in 14.4%. Tool poisoning, in which a tool's description carries instructions aimed at the model, is a direct consequence of who reads a description: a model that follows instructions found anywhere in its input.

1.5 The Gap this Monograph Addresses

1.5.1 Implementation Reports versus a Design Method

The literature reviewed above contains two kinds of work. Implementation reports and benchmarks describe particular agents, often for root-cause analysis in large cloud services, and measure them against datasets or simulated failures

(Chen et al., 2024; Pei et al., 2025; Roy et al., 2024; Wang et al., 2024b). Architecture surveys classify the internal components of agents: planning, memory, tool use, reflection (Masterman et al., 2024; Sumers et al., 2024). Practitioner guidance recommends composing simple, well-documented patterns before reaching for autonomous agents (Anthropic, 2024).

None of these treats the integrity of an agent placed in front of operational systems as its subject: how the tools it calls are scoped, which boundaries hold when the model misbehaves, how failures are classified so that the answer means something to the reader, and how the question's semantics are fixed before any data is fetched. A procedure an engineer could follow to build such an agent over their own pipeline, and to review someone else's, was not identified in the reviewed literature.

1.5.2 Four Structural Decisions

The method developed in this book, Contract-Bounded Agent Design (CBAD), rests on four structural decisions. Each is stated here and developed in its own chapter.

1. **Narrow tools, one per external source.** The agent reaches each external system through a small, read-only, stateless tool scoped to that system, and every consumer shares the same tool layer. The number of tools tracks the number of systems, and new questions are answered by composing existing tools (Chapter 2).

2. **Safety contracts enforced in code.** Every boundary that must hold (which operations are permitted, which environment is in scope, how much data may be fetched, how long a call may take) is checked by code before or after the model acts, and never left to the wording of a prompt (Chapter 3).

3. **An explicit failure taxonomy that classifies by meaning.** Failures are classified on two orthogonal axes, the business stage at which work stopped and the concern that owns the fix, instead of by the exception class that happened to be thrown (Chapter 4).

4. **A named separation of current state and historical occurrence.** The two semantics that one surface question can express, "what is failing now" and "what failed during a period", are named, selected by an explicit test and never allowed to default to the broader one (Chapter 4).

Table 1.1 places the families of approaches discussed in this chapter against the requirements of a cross-source investigation. The final row anticipates what the four decisions add.

Table 1.1. Approach families against the requirements of cross-source operational investigation

Approach family	Crosses several systems of record	Ordered multi-step enquiry	Business meaning in the answer	Bounded cost and load	Holds when the model misbehaves	Answer a human can verify
Metrics and rule-based alerting	No: one service's envelope	No	No: technical names	Yes	Not applicable	Partial: a chart
Log search and log analytics	No: one stream at a time	No	No	Partial: query limits	Not applicable	Yes: raw lines
Distributed tracing	Partial: request paths only	No	No	Yes: sampling	Not applicable	Yes: the trace
Dashboards	Partial: side-by-side panels	No	Partial: curated labels	Yes	Not applicable	Partial
Text-to-SQL	No: one database	No	Partial: depends on schema names	Partial	No: generated query runs as written	Partial: the query
Retrieval-augmented generation	Partial: one index	Partial: iterative variants	Partial	Partial	No	Partial: cited passages
Tool-using agent without contracts	Yes	Yes	Depends on prompt	No	No	Depends on prompt
Contract-bounded agent (design intent of this book)	Yes: one tool per source	Yes: encoded runbook	Yes: two-axis taxonomy	Yes: ceilings in code	Yes: guards in code	Yes:verification path

Source: compiled from the literature discussed in this chapter. The final row states the properties the method is designed to provide (Chapters 2–5).

The comparison in the table is qualitative and coarse. Its purpose is to locate the gap: the families that cross systems and order their steps are the families whose integrity depends on the model, and the families whose integrity does not depend on a model cannot conduct an enquiry. The four decisions are the conditions under which an agent can do the first without inheriting the weakness of the second.

CHAPTER 2. THE TOOL LAYER: NARROW SOURCES, EXPLICIT CONTRACTS

The first structural decision concerns the surface through which an agent touches the world. Every external system the agent may consult is reached through a tool, and the shape of those tools decides how much of the agent's behaviour can be reasoned about, tested and reused. The decision has two parts: how tools are scoped, and what each tool promises about itself.

2.1 One Narrow Tool per External Source

2.1.1 The Alternative and why it fails

The default way to give a new consumer access to operational data is a bespoke interface: an endpoint or a query written for the questions that consumer asks. The first assistant over a pipeline gets an API shaped like its prompts. The second consumer, a dashboard or a different assistant, gets another. Each interface embeds assumptions about the backends it reads, so that every change in a backend schema or behaviour has to be propagated to every consumer that wrapped it. The coupling grows with the product of consumers and backends.

The architecture literature has named the alternative for decades. Parnas (1972) argued that the criteria used to divide a system into modules decide how flexible and comprehensible it is, and showed that a decomposition chosen by a different criterion than the order of processing steps had clear advantages for change. Ports and adapters, the pattern also known as hexagonal architecture, states the goal for an application's boundary: to let the application be driven equally by users, programs, automated tests or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases (Cockburn, 2005). Applied to an agent, the tool layer is the set of adapters, and the agent is one driver among several.

2.1.2 Scoping Rules

Three scoping rules follow.

1. **One tool answers one kind of question about one source.** A run-status tool reads the scheduler, a warehouse-query tool reads the warehouse, a repository tool reads source control. A tool that joins two sources hides a cross-source decision inside an implementation where the agent cannot see it and the reviewer cannot test it.

2. **Read operations and act operations are separate tools.** Reading run state and retrying a run are different tools with different contracts, different guards and, where possible, different permissions. The separation lets the read path be broad and the act path be narrow, audited and gated.

3. **Each tool is stateless.** A call carries everything the tool needs; the tool keeps no session. State that the investigation accumulates lives with the agent and is visible in its trace.

Both systems on which this book draws were built to these rules. System A, an incident-investigation agent over a workflow orchestration engine, exposes narrow tools for operations such as finding process instances by a business variable and fetching the error detail of an instance, and keeps the operation that retries an instance as a separate act tool. System B, a multi-purpose assistant over a data-pipeline scheduler, reaches each of its external systems (run status, warehouse, source repository, versioned configuration and documentation) through its own read-only tool. The tool layer let the reasoning component of System A be reused instead of being wired to one workflow, and it let System B grow from failure investigation into general enquiry without new interfaces per question.

2.1.3 How the Tool Count Should Grow

The count of tools should grow with the number of external systems and the number of distinct question kinds per system, and never with the number of questions users ask. A request for a new kind of answer is first met by composing existing tools, through composite templates or the investigation runbook, and only then, if the composition needs data no tool reads, by a new tool for that source. The stages of tool learning noted in Chapter 1 (Qu et al., 2025) make the same point from the agent's side: a stable, narrow tool set keeps the selection stage tractable.

2.1.4 What Narrow Tools Cost

The decision has costs, and they grow with the number of tools. Each narrow tool is one more choice the model must make correctly, and selection accuracy falls as tool catalogues grow (Guo et al., 2026). An investigation that a broad tool would answer in one call takes several narrow calls, each with its own latency. Orchestration that a broad tool would hide moves into the runbook and the templates, where it has to be designed and tested.

The realistic alternative is one generic tool per source, such as a free-form query tool over the warehouse. It answers questions no narrow tool anticipated, which makes it attractive for exploratory analysis by a trusted user. It also carries no contract beyond "a query", so its guards must cover every statement a model might write, and its descriptions cannot route by question kind. The narrow layout wins where the consumers are many, the questions recur and the boundaries matter; a generic tool remains the better choice for a single expert user exploring one store, and a layer can offer both, with the generic tool behind the strictest guards.

Figure 2.1 shows the resulting structure. Consumers share one tool layer; each tool reaches one external system; the single act tool is marked as gated.

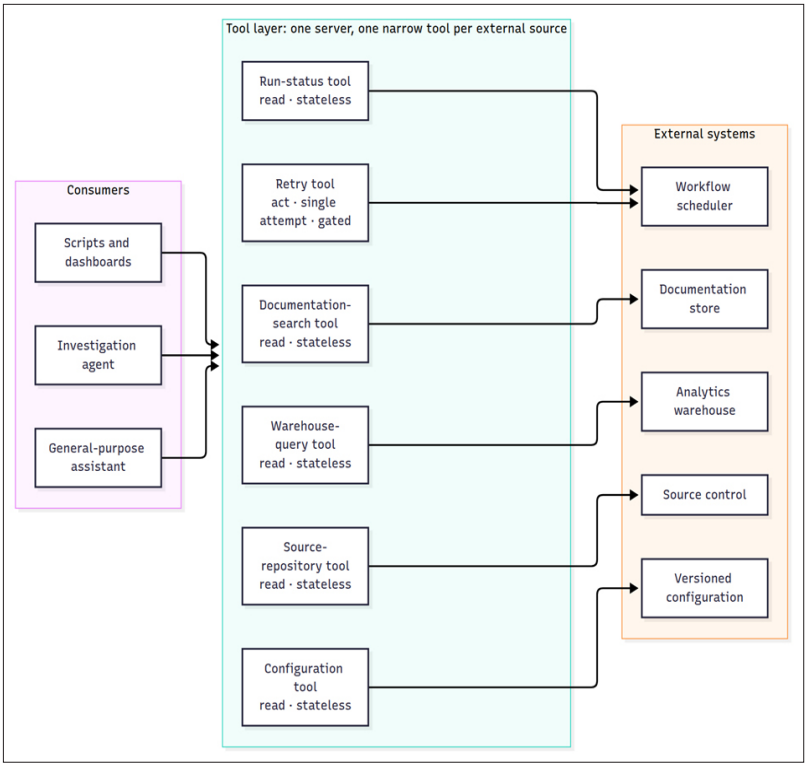


Figure 2.1. Tool-layer schema: consumers share one server that exposes one narrow, stateless tool per external system, with read and act operations separated. *Source: generalised from the tool layers of System A and System B.*

2.2 The Tool Description as a Semantic Contract

A tool reaches the model as a name, a natural-language description and a schema for its input, and in recent protocol revisions an optional schema for its output and annotations about its behaviour (Model Context Protocol, 2025b). The model selects tools by reading these fields. The description is therefore not documentation in the ordinary sense. It is the contract on which routing decisions are made by a reader that does not ask clarifying questions.

2.2.1 What a Description Must Carry

Earlier work on a two-layer investigation architecture defined a tool by its name, a natural-language description of what it does, when it applies and what shape its result takes, input and output schemas, an applicability declaration, an idempotency declaration and a side-effect annotation, and observed that the description carries most of the discriminative power in tool selection (Puchakayala, 2026c). A tool contract is the set of these fields treated as one reviewable unit, and Table 2.1 consolidates them into a template.

Table 2.1. Anatomy of a tool contract

Field	What it states	Example (illustrative)	Who relies on it
Name	One verb and one object, unique across the layer	get_run_status	Model (selection), logs, reviewers
Purpose	The one question the tool answers, about which source	"Returns the state of one scheduled run in the pipeline scheduler"	Model (selection)
Applicability	When to call it, and when not to	"Call when the user names a run or asks about the latest run; do not call for history across days"	Model (selection)
Authority	Which source is authoritative for which phrasing when two could answer	"Configured threshold: read the configuration tool; threshold used at run time: read this tool"	Model (selection), reviewers
Input schema	Required and optional parameters, types, allowed values	run_id (string) or pipeline_key + latest=true	Client validation, model (arguments)
Result shape	Fields returned, units, time zone, and whether the result can be truncated	"Returns state, start, end (UTC), task counts; truncated and total when capped"	Model (answer), answer layer
Side effects	Read-only, or the exact change it makes	"Read-only"	Guards (Chapter 3), reviewers

Idempotency	Whether repeating the call changes the outcome	"Safe to repeat" / "Each call is one retry attempt"	Retry logic, human decision points
Failure contract	Typed errors the caller can branch on	NOT_FOUND, UPSTREAM_UNAVAILABLE, SCOPE_REFUSED	Model (next step), answer layer

Source: compiled by the author from the tool-definition fields in Puchakayala (2026c), the MCP tool specification (Model Context Protocol, 2025b) and the tool layers of System A and System B; field values are constructed for illustration.

Two fields deserve emphasis because they are often missing. The authority field resolves a conflict the model cannot resolve alone: when a static configuration value and a value known only at run time can both answer a question, the contract states which one to prefer for which phrasing. System B states this preference in its tool descriptions instead of letting the model guess. The side-effect and idempotency fields matter because the protocol itself treats behavioural annotations as hints; the MCP specification requires that, for trust and safety, clients consider tool annotations to be untrusted unless they come from trusted servers (Model Context Protocol, 2025b). A declaration in a description informs the model. Enforcement belongs to code.

2.2.2 Routing Intelligence belongs in Descriptions

The instructions that tell a model when to call which tool can live in two places: the system prompt or the tool descriptions. When they live in the prompt, every consumer that reuses the tool layer has to copy them, and the copies drift. When they live in the descriptions, they travel with the tool. Both systems keep routing knowledge in the descriptions as the single source of truth and refer to it from prompts instead of restating it.

The research on tool learning supports placing this weight on documentation. Hsieh et al. (2023) found that zero-shot prompts containing only tool documentation were sufficient to elicit proper tool usage, with performance on par with few-shot prompts built from demonstrations, and they recommended documentation over demonstrations because demonstrations are hard to obtain and bias usage toward the examples chosen. EASYTOOL condensed verbose tool documentation into concise, structured instructions; on ToolBench the transformation reduced the token cost of tool documentation by 70.43% and improved task performance across models (Yuan et al., 2025). One model vendor's guidance points the same way: Anthropic's documentation recommends aiming for at least 3–4 sentences per tool description, more for complex tools (Anthropic, n.d.), and its engineering guidance suggests namespacing related tools under common prefixes to delineate boundaries between them (Anthropic, 2025b).

2.2.3 Writing for a Non-Deterministic Reader

Developer documentation is written for a reader who can ask a colleague, read the source, or notice that an example contradicts the prose. A model reading a tool description can do none of these. Three consequences follow for the writer of a description.

First, conditions must be stated in words: "call when the user names a run" is a usable condition, while "useful for run questions" is not. Second, negative conditions are as important as positive ones, because two tools whose descriptions both claim a question will split traffic between them from one run to the next. Third, the description must not carry volatile operational detail such as a current threshold or a list of today's job names; such detail goes stale silently and is better returned at run time as data. Uddin and Robillard (2015), surveying 323 software professionals about API documentation, found ambiguity, incompleteness and incorrectness to be the three severest documentation problems. A model meets the same three problems in a tool description and has no way of recognising them.

2.3 Tool Description Debt

Technical debt names the future cost created by expedient design and implementation choices (Cunningham, 1992), and the concept has been extended from code to design, architecture and documentation (Kruchten et al., 2012). Machine-learning systems add their own forms, because behaviour depends on data and configuration as much as on code (Sculley et al., 2015). Tool description debt, a category defined in earlier work for language-model-based observability systems, is the gap that opens when the natural-language description of a tool diverges from the tool's actual behaviour, so that the agent's tool-selection step weakens without any visible error (Puchakayala, 2026c).

2.3.1 The Four Forms and their Symptoms

The earlier taxonomy distinguishes four forms of the debt (Puchakayala, 2026c). Table 2.2 states each form with its symptom in an investigation and its remedy.

Table 2.2. Taxonomy of tool description debt

Form	Definition	Symptom in an investigation	Remedy
Vague	The description underspecifies when the tool applies	The tool is called when its result will be empty or noisy; investigations take extra steps	Sharpen the purpose; add applicability conditions and counter-examples
Overlapping	Several tools claim similar semantics	Selection between tools becomes unstable; the same question takes different paths on different runs	Disambiguate scope; merge duplicates; state authority per phrasing
Stale	The description survived a change in the tool's behaviour or in the business logic behind it	The tool is chosen for questions it no longer answers, or skipped for questions it now answers	Couple description review to code change; test selection behaviour on every release
Over-specified	The description embeds operational detail that changes	Frequent edits; descriptions contradict the data the tool returns	Move volatile detail into runtime metadata; keep the description on intent

Source: forms and remedies from Puchakayala (2026c); symptom column compiled by the author.

2.3.2 Why the Debt Accumulates Silently

No rule fires when a description drifts away from behaviour. A wrong description does not fail a build, raise an exception or trip a monitor; the only observable signal is a gradual decline in the quality of investigations, and nothing in the decline points to the description as its cause (Puchakayala, 2026c). The documentation literature catalogues the forms the underlying drift takes. Wen et al. (2019) mined the complete change history of 1,500 Java systems, 1.3 billion AST-level changes, and built from 500 manually analysed commits a taxonomy of the code-comment inconsistencies that developers fix. Aghajani et al. (2019) categorised 878 documentation-related artefacts from mailing lists, Stack Overflow, issue trackers and pull requests into a taxonomy of 162 types of documentation issues.

Prompt text, the closest relative of a tool description, drifts in the same way and is documented less. Tafreshipour et al. (2025) analysed 1,262 prompt changes across 243 GitHub repositories and found that only 21.9% of the changes were explicitly documented in commit messages. A study of self-admitted technical debt in 93,142 Python files using language-model APIs identified prompt design as the primary source of debt specific to language models, at 6.61% of the debt found (Aljohani & Do, 2025). The maintainability findings for open-source MCP servers reported in Chapter 1 point the same way. None of these studies measured tool descriptions directly; together they show that the texts surrounding language-model integrations change often, are rarely recorded when they change, and accumulate the maintenance problems of ordinary code.

Recent work confirms that selection quality depends on description quality at a fine grain. DOCSCHISEL found that adding or removing a single information field in a tool's documentation changed the task success rate of agents by 6.34 percentage points on average (Lu et al., 2026). Work on rewriting tool descriptions reported that improved descriptions reduced accuracy degradation by 29.23% as tool catalogues scaled to more than 150 candidates (Guo et al., 2026). Both results come from benchmarks, and production tool layers have not been measured this way; both point to the description as an engineering artefact whose quality can be measured and lost.

The description is also an attack surface. Tool poisoning, in which a description carries instructions aimed at the model, appeared among the open-source MCP servers surveyed in Chapter 1 (Hasan et al., 2026), and Dabas et al. (2026) scanned 6,062 tools across 288 verified MCP servers and flagged 608 with parameters susceptible to drift induced by an agent's memory. A layer that treats descriptions as reviewed code is easier to defend than one that treats them as free text.

2.3.3 Remedies

The remedies follow from treating descriptions as first-class engineering artefacts.

1. **Couple description review to code change.** A change to a tool's implementation, its upstream system or the business rule behind it opens a review of its description in the same change request.

2. **Test selection, not only execution.** A description test pairs a set of user questions with the tool that should be selected and the tools that should not; the suite runs on every release of the tool layer.

3. **Disambiguate scope.** Overlapping tools are either merged or given mutually exclusive applicability conditions and an authority statement.

4. **Move volatile detail to runtime metadata.** Thresholds, job lists and environment names are returned by tools as data, never written into descriptions.

2.4 Composite Templates as Investigation Primitives

Some multi-tool sequences recur so often that leaving them to the model's planning wastes steps and invites variation. A composite template is a recurring multi-tool sequence captured as one named tool with its own contract. The earlier

architecture captured such sequences as analysis templates: recurring composite queries encapsulated as reusable investigation primitives, which reduce the reasoning burden on the agent (Puchakayala, 2026c). A constructed example in a neutral domain is "list the claims that have stayed in one processing stage for more than D days, with their last error", which joins a status read, a filter and an error-detail read into one call that the model selects by name.

Practitioner guidance on agent design makes a related distinction between workflows, in which language models and tools are orchestrated through predefined code paths, and agents, in which the model directs its own process and tool usage (Anthropic, 2024). A template is a small workflow placed inside an agent's tool set.

2.4.1 Which Sequences Deserve a Template

A sequence deserves a template when four conditions hold.

1. It recurs across many investigations with the same order of steps.

2. Its intermediate results are not interesting to the user; only the final result is.

3. Its steps read from sources that a single tool layer already reaches, so the template adds no new access.

4. Its result can be stated as a contract in the same form as Table 2.1.

A sequence should stay composed at run time when its order depends on intermediate results, when users ask to see its intermediate steps, or when it is new and its stable form is not yet known. Freezing an unstable sequence into a template creates a stale template, which is tool description debt in a new form.

2.5 Procedure: Designing a Tool Layer for Your Own System

2.5.1 From an Inventory of Sources to a Tool Catalogue

1. **Inventory the systems of record.** List every system that is authoritative for some fact an investigation needs: scheduler or orchestrator, warehouse, logs, source control, configuration, documentation, ticketing. For each, note the access path, the authentication model and the read cost.

2. **List the question kinds per system.** For each system, write the few kinds of question an investigation asks of it ("state of a run", "rows for a business key", "where is this rule defined"). Merge kinds that differ only in parameters.

3. **Draft one read tool per question kind per system.** Name it with a verb and an object; write purpose, applicability, authority and result shape (Table 2.1).

4. **Separate act operations.** Every operation that changes state (retry, cancel, annotate) becomes its own tool with a single-attempt contract and its own guard (Chapter 3).

5. **Resolve overlaps.** Where two tools could answer one phrasing, write the authority statement into both descriptions.

6. **Identify template candidates.** Apply the four conditions for templates to the sequences that recur in past investigations.

7. **Write description tests.** For each tool, list questions that must select it and questions that must not.

2.5.2 Review Checklist for a Tool Layer

Before a tool layer is placed in front of an agent, a reviewer confirms:

- every tool reads or acts on exactly one external system;
- read and act operations are separate tools;
- no tool keeps state between calls;
- every description states purpose, applicability, authority where needed, result shape, side effects and idempotency;
- no description contains thresholds, job lists, environment names or other volatile operational detail;
- the result shape states whether results can be truncated and how the true total is reported;
- every act tool has a single-attempt contract and is gated (Chapter 3);
- description tests exist and run with every change to the layer;
- routing instructions live in descriptions and the system prompt refers to them without restating them.

A tool layer that passes this checklist is the precondition for everything that follows. Guards attach to tools, failure taxonomies classify what tools return, and an investigation runbook is a sequence of tool calls.

CHAPTER 3. SAFETY CONTRACTS ENFORCED IN CODE

The second structural decision moves every boundary the system must respect out of the model's instructions and into code that runs whether or not the model follows them. The principle needs guards to implement it and a discipline that keeps a team honest about which guards exist: auditing implementation against claims, and validating a system whose central component is non-deterministic.

3.1 The Principle: Never Delegate a Boundary to the Prompt

3.1.1 Why Prompt-Level Instruction is Insufficient

A prompt can ask a model to leave data unmodified, to stay out of production and to fetch only what it needs. The request is honoured most of the time. Security research on language-model applications shows why "most of the time" is the wrong standard for a boundary. Greshake et al. (2023)

demonstrated indirect prompt injection: instructions planted in data that an application retrieves, such as a web page or a document, are processed by the model as instructions, so that whoever controls the data can steer the application. In an operational agent, log lines, ticket comments, configuration files and error messages are all data that an attacker or an accident can shape.

Benchmarks quantify the exposure. InjecAgent evaluated 30 agents on 1,054 test cases of indirect injection through tool outputs and found a ReAct-prompted GPT-4 agent vulnerable 24% of the time (Zhan et al., 2024). AgentDojo provides 97 realistic tasks and 629 security test cases for agents with tools (Debenedetti et al., 2024). Defences that act on the model's input or behaviour reduce these rates without closing them: Zhan et al. (2025) evaluated eight defences against indirect injection and bypassed all of them with adaptive attacks, with attack success rates above 50%. Training a model to rank system instructions above data, the instruction hierarchy, raised robustness by up to 63% on its authors' evaluations (Wallace et al., 2024). A substantial gain from training is still a probability, and a boundary that holds with high probability is a boundary that will eventually fail.

The specific case of databases makes the point concrete. Pedro et al. (2025) showed that applications built on a popular middleware let an adversary with access to a chatbot interface inject arbitrary SQL, and that restrictions written into the prompt, such as "do not write to the database", could be bypassed for both writes and reads. A prompt that forbids a statement does not forbid the statement; only a check between the model and the database does.

3.1.2 The Division of Responsibility

The OWASP project lists excessive agency among the principal risks of language-model applications, defining it as the vulnerability that enables damaging actions in response to unexpected, ambiguous or manipulated model outputs, whatever causes the model to malfunction, and traces its root causes to excessive functionality, excessive permissions and excessive autonomy (OWASP Foundation, 2025a). The definition is useful because it does not depend on why the model misbehaved. The remedy it implies is architectural.

Operating-systems design offers the template. In Hydra, the separation of mechanism from policy was a basic design principle: the kernel was built almost entirely of mechanisms, and policies that used them were left to user-level software (Levin et al., 1975). An agent inverts the familiar arrangement but keeps the separation. The model chooses the policy (which tool to call next, with which arguments, in pursuit of which question) and code supplies the mechanism, which includes the refusals. Research on securing agents reaches the same structure from the attack side. Debenedetti et al. (2025) separated control flow from untrusted data so that injected content cannot change which tools are called, and their system solved 77% of AgentDojo tasks with provable security against 84% for an undefended system. Beurer-Kellner et al. (2025) catalogued design patterns, among them action selection from a fixed list and planning before reading untrusted data, whose common principle is that once an agent has read untrusted input, its possible actions are constrained by construction.

The division can be stated as one sentence for design reviews: **the model decides what to ask; the code decides what may be done.**

3.1.3 What Code Enforcement Costs

Guards are code, and code has costs. Each guard has to be written, tested against inputs designed to defeat it and maintained as the upstream systems change. Guards also refuse legitimate requests: a statement guard can reject a valid read written in an unfamiliar form, and a ceiling can cut off an answer that needed one more page. Those refusals are the price of a boundary that fails closed, and they are visible, because each returns a typed error that the model reports and an engineer can inspect.

Not every instruction to a model is a boundary. Preferences about answer format, tone or which of two valid tools to try first can stay in the prompt, because a lapse costs quality and leaves the boundaries intact. The test for whether an instruction belongs in code is the consequence of a single violation: if one lapse can change data, reach the wrong environment, exhaust a quota or mislead a reader about completeness, the instruction is a boundary and moves into code.

3.2 Statement and Scope Guards

3.2.1 Restricting Operations to a Permitted Class

A statement guard is a check that inspects a request before it leaves the tool and refuses anything outside a permitted class. For a warehouse-query tool, the permitted class is read statements. The guard is a deterministic check in the tool's code, applied to the final text that will be sent. System B's warehouse tool works this way: the statement class is checked in code before a query leaves the tool, a statement outside the permitted class is refused with a typed error the model can read, and the model's instructions play no part in the decision. Such a guard can be built as a list of forbidden statement keywords or as an allowlist of read forms, and the principles below favour the second.

Saltzer and Schroeder (1975) named the principles such a guard follows. The principle of fail-safe defaults bases access decisions on permission rather than exclusion; for a statement guard it recommends an allowlist that accepts only recognised read forms, which fails closed on a statement type its authors did not foresee, where a blocklist fails open. A blocklist guard is therefore always paired with the next two principles. The principle of complete mediation requires that every access to every object be checked for authority; the

guard therefore sits in the one code path every query takes. The principle of least privilege has every program operate with the least set of privileges necessary to complete the job; the credentials behind such a tool should be read-only even when the guard already refuses writes.

3.2.2 Pinning Scope

A scope guard is a check that fixes, in code, which environment, project, dataset or tenant a tool may address. A runtime input, whether from the user, the model or a retrieved document, can then supply a filter within that scope but never redirect the query to a different environment. The environment identifier is a constant of the deployment, and no tool parameter carries it. System B pins its warehouse scope this way, so that no argument the model produces can point a query at another project.

3.2.3 Layering Independent Guards

No single guard is perfect. A statement parser can be confused by an unusual dialect; a credential can be broader than intended; a ceiling can be misconfigured. Reason (2000) described the defences of complex systems as slices of Swiss cheese whose holes continually open, shut and shift, so that a hole in any one slice does not normally cause a bad outcome; harm requires holes in many slices to line up. The design implication is independence: guards should fail for different reasons. A statement allowlist, a pinned scope, a read-only credential and a payload ceiling share no failure mode. System B's warehouse tool combines three such guards (statement class, pinned scope and a cap on data scanned per query), none of which depends on the model behaving.

Figure 3.1 shows the layering as a request passes through it.

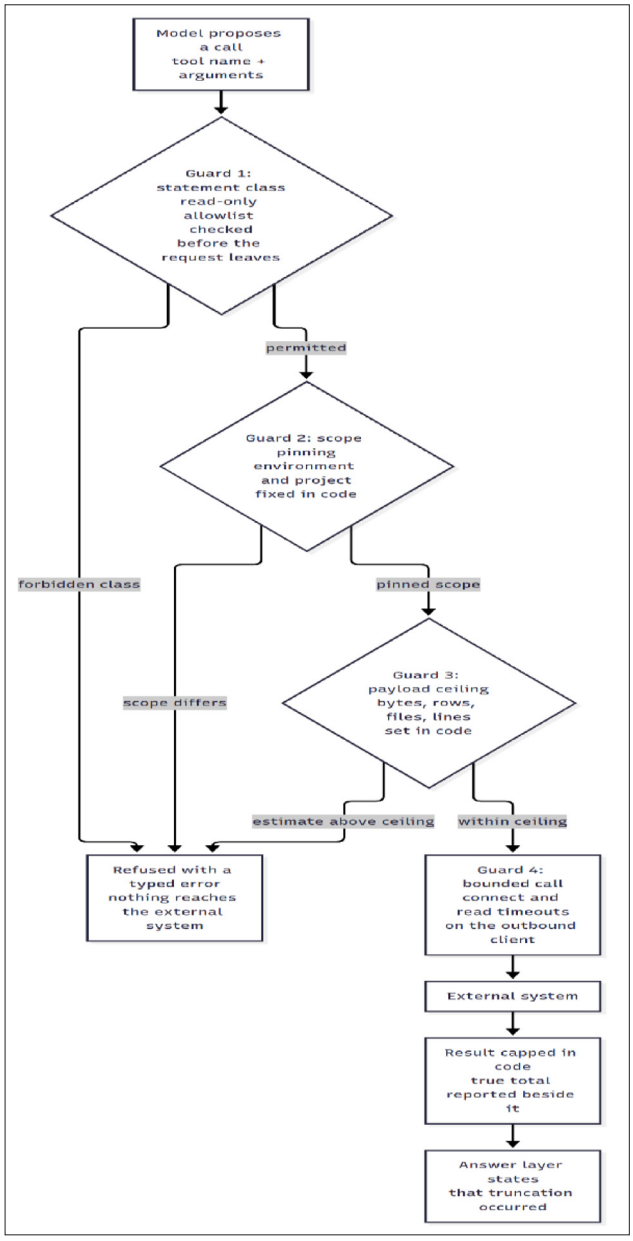


Figure 3.1. Layered guards between the model and an external system. Each guard is enforced in code and fails for a different reason; a refusal returns a typed error and nothing reaches the external system. *Source: generalised from the guard layers of System B.*

3.3 Explicit Ceilings on Every External Payload

3.3.1 Caps Set in Code

Every tool that reads from an external system has a ceiling on how much it reads: bytes scanned, rows returned, files fetched, characters of a file, log lines extracted, items listed. The ceiling is set in the tool's code, independently of what the upstream system is willing to return. OWASP classifies the absence of such limits as unbounded consumption and recommends input validation against reasonable size limits, rate limiting and quotas (OWASP Foundation, 2025b). MITRE's weakness catalogue describes the same defect for ordinary software as allocation of resources without limits or throttling (MITRE, n.d.).

Some upstream systems provide a native ceiling, and a tool should use it in addition to its own. BigQuery's maximum-bytes-billed setting estimates the bytes a query will read before execution and fails the query without charge when the estimate exceeds the limit (Google Cloud, n.d.). Agent platforms apply the same logic to tool output: Anthropic's guidance recommends pagination, range selection, filtering or truncation with sensible defaults for any tool response that could use up a large share of the context, and reports restricting tool responses in one of its own products to 25,000 tokens by default (Anthropic, 2025b).

Table 3.1 is a decision table for setting ceilings. The values are constructed for illustration; a team sets its own from the cost of a read, the size of the model's context and the size of a useful answer.

Table 3.1. Decision table for payload ceilings (illustrative values)

Payload	Ceiling enforced in code	Illustrative value	What happens at the ceiling	Native mechanism to combine with
Warehouse query	Data scanned per query	2 GB	Query refused before execution, typed error	Bytes-billed limit, dry-run estimate
Warehouse result	Rows returned	200 rows	Result truncated; true row count returned	LIMIT added by the tool, not by the model
Source file	Characters read per file	64 KB	File truncated with a marker and the true size	Range reads
Run listing	Runs fetched per call	500 runs	Listing truncated; total and warning returned	Server-side page size
Log extraction	Lines returned per failed step	300 lines	Earliest and most specific windows kept	Server-side grep or filter
Search results	Hits returned	25 hits	List truncated; total returned	Query filters
Outbound call	Connect and read time	2 s / 20 s	Call aborted, typed UPSTREAM_TIMEOUT	Client timeout settings

Source: compiled by the author; values constructed for illustration.

3.3.2 The Missing Timeout as a Design Defect

An outbound client without a configured connect and read timeout waits as long as the remote side allows. Under normal load the omission is invisible; under a slow dependency it turns every caller into a held thread or connection, and the slowness spreads. Amazon's engineering guidance treats timeouts as a deliberate choice, picking an acceptable rate of false timeouts (such as 0.1%) and setting the timeout at the corresponding latency percentile of the downstream service (Amazon Web Services, n.d.). The same guidance warns that retries multiply load across layers: in a call stack five services deep with three retries at each layer, the load on the database at the bottom rises 243-fold if each layer retries independently, making recovery unlikely (Amazon Web Services, n.d.). A missing timeout and an unbounded retry loop are therefore design defects, to be fixed before a system goes live. Nygard (2018) presents timeouts and circuit breakers among the stability patterns that keep a failure in one component from cascading, and a circuit breaker, in Fowler's description, wraps a protected call and, after failures reach a threshold, returns errors at once instead of calling the failing service (Fowler, 2014).

3.3.3 Unbounded Collection Endpoints

A feature checklist asks whether an operation exists. It does not ask how large its input can be. An endpoint that accepts an array of items and processes all of them, or a query that lists records with no limit and no pagination, works in tests with small inputs and fails when an input arrives that is far larger than the test data. Google's API design guidance requires list methods to support pagination with a default page size and a documented maximum, coercing larger requests to the maximum (Google, 2019a). Two review questions find these risks before load does: "what is the largest input this operation accepts?" and "what is the largest result this query can return?" An answer of "unbounded" to either is a defect.

3.4 Truncation Honesty

A ceiling creates a new risk: a truncated result that looks complete. A model given the first 200 rows of a result has no way to know that 40,000 exist unless the tool says so, and an

answer built on the visible rows will understate counts, miss failures and misstate trends.

The rule is that a tool never truncates silently. Every capped result carries two fields beside the data: a flag that truncation occurred and the true total, or a lower bound on it when counting is itself too expensive. System B applies this rule to its capped listings, returning the true total beside the capped result and an explicit warning whenever the total exceeds what was fetched.

The second half of the rule belongs to the answer layer. The system prompt requires the model to state, in its answer, that the result was truncated and what the true total was, and the answer discipline of Chapter 5 includes the same requirement as a checklist item. A truncation that the tool reports and the answer omits is still a silent truncation from the reader's point of view.

3.5 Liveness and Identity Resolution before Action

Two guards protect the agent from acting on a picture of the world that is no longer true. Both are checks performed by code before an action or a query, and both close the fourth failure mode of Chapter 1: actions on stale state.

3.5.1 Confirming that the Entity is Still Live

An operational agent may read from a secondary store, such as an incident table in a warehouse or a cache of recent failures, because the secondary store is cheaper to query and easier to filter than the system of record. A record in a secondary store describes the world at the moment it was written. Before an agent acts on such a record, by retrying a process, annotating it or reporting it as open, code confirms against the system of record that the underlying entity is still live. If it is not, the tool does not act; it reports that the record is stale and, where the design permits, removes or marks the stale record.

System A applies this check before acting on an incident row retrieved from its warehouse: it confirms that the process instance is still active in the orchestration engine, and if the instance has finished or been cancelled, it cleans up and says so instead of acting. The check is a rule in its own right, with a cost and conditions of application.

The term needs one clarification. In container orchestration, a liveness probe determines when to restart a container (Kubernetes Authors, n.d.). The liveness check of this chapter asks a different question, about a business entity, and its outcome is a decision not to act; it never restarts anything.

3.5.2 Resolving Schema and Identity before Composing a Query

The same discipline applies to identifiers and schemas. Before the agent composes an ad-hoc query, a dedicated tool confirms the real field names and the latest available snapshot, so that the agent never guesses a column or queries a partition that does not exist yet. System B requires this order in its tool contracts: a schema-and-snapshot tool is called before any ad-hoc warehouse query is written. The order is a correctness discipline carried by the contracts; where a wrong field name would have a safety consequence, the query tool can also refuse, in code, a query that names fields the resolution step did not return. The schema-linking errors of text-to-SQL models described in Chapter 1 (Li et al., 2023) are errors of guessing, and a resolution step replaces the guess with a lookup.

3.6 Claimed Resilience is Not Implemented Resilience

3.6.1 The Audit Procedure

Projects document their resilience features: feature matrices, architecture pages, checklists marked "done". The documentation describes intent. Whether a timeout, retry policy, circuit breaker, ceiling or pagination rule exists in the running code is a separate question, and the answer can only come from the code and its configuration.

The claimed-versus-implemented audit is a review that reads the claims against the implementation, one claim at a time.

1. **List the claims.** Collect every resilience feature that the project's documents mark as present: timeouts, retries with backoff, circuit breakers, bulkheads, rate limits, ceilings, pagination, idempotency keys, dead-letter queues that hold the messages a consumer cannot process.

2. **Locate each mechanism.** For each claim, find the code or configuration that implements it: the timeout setting on each outbound client, the retry annotation or library call, the breaker wrapper, the LIMIT or page-size parameter on each list query.

3. **Check each call path.** A timeout configured on one client does not protect another. List every outbound call and every collection endpoint and mark each one covered or not covered.

4. **Exercise the mechanism.** For each mechanism found, write or run a test that triggers it: a slow stub for a timeout, an oversized input for a ceiling, repeated failures for a breaker.

5. **Record the result as a table of claim, location, coverage and test.** Claims without a location are reported as gaps.

A constructed example from the claims-processing pipeline shows the typical finding. The design page of the payment hand-off service states that a claim the payment system rejects is moved to a dead-letter queue for manual review. The audit finds the queue defined in the messaging configuration, while the consumer catches every rejection, writes it to the log and acknowledges the message, so no rejected claim ever reaches the queue. The documented feature exists in the configuration and in the design page. It does not exist on the path a rejected claim takes.

3.6.2 Why Documentation and Implementation Diverge

Documentation and code are maintained by different acts. A design page is written when a feature is planned; the code is

changed when a bug is fixed or a deadline approaches; nothing ties the two. The drift between code and its accompanying text described for comments and tool descriptions in Chapter 2 has the same cause.

The consequences concentrate in error handling, the code path that runs least often. Yuan et al. (2014) studied 198 randomly selected, user-reported failures in Cassandra, HBase, the Hadoop Distributed File System, Hadoop MapReduce and Redis, and found that, of the 48 catastrophic failures in that sample (those affecting all or most users), 92% resulted from incorrect handling of non-fatal errors explicitly signalled in the software, and 35% were caused by trivial mistakes in error-handling logic. Resilience features are error-handling code by definition. They are the code most likely to be claimed, least often exercised and wrong.

During review, the divergence is detected by refusing to accept a claim without a location. A reviewer who reads the feature matrix and approves it has reviewed the documentation. A reviewer who asks “show me the line that sends a rejected claim to the dead-letter queue” has reviewed the system.

3.7 Validating an Agentic System

3.7.1 What Unit Tests Cover When the Model is in the Loop

The deterministic parts of an agentic system are tested like any other code: tools, guards, ceilings, parsers, classification tables and answer formatters. The boundary for mocking is the external system on one side and the model on the other. Tools are tested against stubs of the systems they read. In four software systems chosen because they use mock objects routinely, Spadini et al. (2017) analysed 4,419 test units and found that 25.39% contained at least one mock object. Guards are tested with inputs designed to violate them. The classification tables of Chapter 4 are tested with known signals and their expected classes.

What unit tests cannot cover is the model’s choice. In an interview study with 26 professional software engineers who build product copilots, the engineers reported pain points at every step of the engineering process and challenges that strained existing development practices (Parnin et al., 2025). A study of testing in agent applications labelled 2,572 test methods from 240 modules and found that 61.8% tested tools or agents in isolation, 34.6% targeted interaction-level behaviour and 3.5% evaluated multi-tool scenarios (Pan et al., 2026). The tests concentrate where determinism is available.

3.7.2 The Evaluation Gap

An automated harness for the reasoning step of an operational agent needs a golden-answer set: a fixed collection of investigation questions, each with the expected tool sequence or the set of acceptable sequences, the expected classification on both axes of Chapter 4, the expected answer fields, and the verification path a human would follow. It also needs a reliability measure across repeated runs, since a non-deterministic system that answers correctly once may not answer correctly twice. The pass^k metric introduced with τ-bench measures this property: the probability that an agent succeeds on all of k trials of the same task, and its authors found state-of-the-art function-calling agents to be inconsistent across trials (Yao et al., 2024).

Evaluation criteria are themselves a moving target. Shankar et al. (2024) observed, in a study of language-model-assisted evaluation, that people refine their grading criteria while grading outputs, so that criteria cannot be fully fixed before the outputs are seen. A golden-answer set is therefore maintained like the tool descriptions of Chapter 2: reviewed when behaviour changes and extended when a new kind of question appears.

3.7.3 Declaring Known Gaps

The ML Test Score rubric presents 28 specific tests and monitoring needs for production machine-learning systems, and its authors found, in a survey of several dozen teams at Google, that none of the tests was implemented by more than 80% of teams (Breck et al., 2017). The value of such a rubric lies as much in the list of what is missing as in the score. An agentic system benefits from the same practice: a short, maintained statement of which guards, tests and evaluations exist, which are planned and which are absent. A declared gap can be scheduled, reviewed and accepted as a risk. An undeclared gap is discovered by an incident.

Table 3.2 collects the guards and practices of this chapter into a minimum safety contract that can be checked during design review.

Table 3.2. Minimum safety contract checklist

#	Contract item	Enforced by	Check during review
1	Read tools hold read-only credentials	Credential scope	Inspect the credential’s grants
2	Statements outside the permitted class are refused before leaving the tool	Permitted-class check in tool code (allowlist preferred to a keyword blocklist)	Test with forbidden and unforeseen statement types
3	Environment, project and tenant are fixed by the deployment	Constant in tool code	Try to override through arguments
4	Every external read has a ceiling	Tool code, plus native limits	Find the ceiling for each tool
5	Every outbound client has connect and read timeouts	Client configuration	Find the setting on each client

6	Every capped result reports truncation and the true total	Tool result schema	Inspect the result fields
7	The answer states truncation when it occurred	System prompt and answer checklist	Review traces with capped results
8	Liveness is confirmed before any action on a record from a secondary store	Act tool code	Test with a finished entity
9	Schema and latest snapshot are resolved before ad-hoc queries	Tool ordering in code or contract	Review traces
10	Act tools make a single attempt per call	Act tool code	Inspect retry configuration
11	Every refusal returns a typed error	Tool error contract	Trigger each guard
12	Claimed resilience features have a code location and a test	Audit table	Read the audit
13	Known gaps are declared	Maintained gap statement	Read the statement

Source: compiled by the author from the guards and practices of this chapter.

CHAPTER 4. FAILURE CLASSIFICATION: TWO ORTHOGONAL TAXONOMIES

An investigation ends in a classification, whether or not anyone names it. The answer "the job failed with a timeout" classifies the failure by its exception; the answer "the claims stopped before adjudication because the enrichment service could not reach the address registry" classifies it by business stage and by owner. The second answer is the one a business reader can act on and the one a downstream team can route. The third structural decision is an explicit failure taxonomy that classifies by meaning, built on two axes that answer two different questions. The fourth decision, the separation of current state from historical occurrence, belongs with it, because the two semantics decide which failures enter the classification at all.

4.1 Why Exception Class is the Wrong Axis

4.1.1 Technical Signal Names Drift

The signals a pipeline emits on failure are written by developers for developers: exception classes, task names, topic names, log templates, error codes. They change whenever the code changes. A task is renamed in a refactoring, a library upgrade changes an exception's class, a log message gains a parameter. Log-parsing research exists because the same event appears under many surface forms; parsers such as Drain extract templates from raw log lines online, so that messages differing only in parameters can be grouped (He et al., 2017). Grouping by template helps with volume. It does not stop the templates themselves from changing between releases.

Alerting and reporting logic that keys on these names inherits every change silently. A renamed task stops matching its alert rule, which fails without an error; a new exception class lands in a default branch and is reported as "other". The degradation is invisible until someone notices that a known failure was not reported.

4.1.2 One Exception, Many Causes and Owners

The same exception class can arise from unrelated causes that belong to different owners. A timeout raised by a data-loading step can mean that the upstream service is slow (its owners fix it), that the network path is broken (the platform team fixes it), that the caller's quota was throttled (whoever owns the quota fixes it) or that the query was too large (the step's authors fix it). Classifying by the exception class puts all four in one bucket and hands the bucket to whoever owns the step.

Empirical studies of incidents classify the other way. Ghosh et al. (2022), in the incident study cited in Chapter 1, sorted root causes into categories that correspond to who fixes them: code bugs, dependency failures, infrastructure, deployment errors, configuration bugs, database and network problems, and authentication failures. Protocol designers have moved in the same direction at the interface level. RFC 9457 defines a "problem detail" that carries machine-readable details of errors in HTTP responses, so that clients can tell error types apart without parsing human-readable text (Nottingham et al., 2023), and Google's API guidance requires each error to carry a machine-readable reason in a constrained format (Google, 2019b). System A follows the same convention: its mutating operations return a typed error string beside the HTTP status, so that a caller can branch on the cause instead of on the status code alone.

Typed errors at the interface are necessary and not sufficient. They name what went wrong in the vocabulary of the component that failed. An investigation needs two further translations: into the vocabulary of the business process (where did the work stop?) and into the vocabulary of the organisation (who fixes it?).

4.2 Classification by Business Milestone

4.2.1 The Translation Table as a First-Class Artefact

A milestone taxonomy is a classification that names the business stages that a unit of work passes through, independently of any engine's vocabulary, and maps every technical failure signal to the stage at which the work stopped. The mapping is built once, as a translation table from signal to stage, and maintained as code: versioned with the pipeline, reviewed when the pipeline changes, and tested

with known signals. Alerting and reporting read the stage, never the raw signal.

System A was built this way. Its system-prompt module names the business milestones of the pipeline in business terms, and a classification enumeration maps the engine's failure topics to error stages, with an explicit fallback for signals it does not recognise. When a task is renamed, one row of the table changes; every report, alert and answer that depends on the stage stays correct.

Table 4.1 shows a translation table for a constructed claims-processing pipeline with five stages: intake, validation, enrichment, adjudication and payment hand-off.

Table 4.1. Signal-to-stage translation table for a constructed claims-processing pipeline (illustrative)

Technical signal (as emitted)	Emitted by	Business stage where work stopped	Stage reported to the business
claim_ingest_parse_error	Intake service	Intake	Claim not received in a usable form
intake.dedupe.conflict	Intake service	Intake	Duplicate submission held
schema_reject_v2	Validation job	Validation	Claim failed validation rules
rules_engine_unavailable	Validation job	Validation	Validation could not run
enrich_lookup_404	Enrichment service	Enrichment	Reference data missing for the claim
enrich.registry.timeout	Enrichment service	Enrichment	Reference data could not be fetched
adj_model_score_fail	Adjudication task	Adjudication	Claim not assessed
ledger_post_conflict	Hand-off task	Payment hand-off	Payment record not created
handoff.retry.exhausted	Hand-off task	Payment hand-off	Payment record not created after retries
any signal not in this table	any	Unknown	Failure not yet classified; under review

Source: constructed by the author for illustration; signal names and stages do not describe any production system.

4.2.2 Sizing the Taxonomy

The signal vocabulary of a mature pipeline runs to dozens or hundreds of names. The stage vocabulary should fit on one line of a status report. A useful rule is that the number of stages should match the number of hand-offs a business reader recognises, and that each stage should correspond to a change in what the business can do with the unit of work: before intake the organisation has nothing, after validation it has an accepted claim, after adjudication it has a decision. Stages that a business reader cannot tell apart should be merged, however different their technical signals.

Two further distinctions earn a place in the taxonomy when the business acts on them differently. One is a terminal non-failure state such as "cancelled", which must not be counted as a failure. The other is the separation of an error at a stage from an error just after it, when the work at the stage was completed but its result was not handed on; the two call for different repairs.

4.2.3 The Explicit Unknown Bucket

A translation table never covers every signal. The rule for unrecognised signals is that they go to an explicit unknown stage, are counted and reported, and are never dropped. A signal dropped by a classifier is a failure that disappears from every report built on the classification.

No published figure gives the share of unrecognised signals in a translation table, but a neighbouring classification problem shows how often a scheme meets cases it was not built for. In the RCACopilot incident dataset described in Chapter 1, incidents whose root-cause category had not been seen before accounted for 24.96% of all incidents (163 of 653) (Chen et al., 2024). Root-cause categories are not failure signals, and the figure does not transfer as a rate; it shows that a classification scheme should expect new cases as a matter of routine. The unknown bucket is the place where they appear first; a periodic review of its contents is how the translation table grows.

4.3 Classification by Owning Concern

4.3.1 Classifying by who Owns the Fix

The owner-concern taxonomy is the second axis: it classifies an infrastructure failure by the concern that owns its fix, regardless of the exception thrown. The categories are defined by ownership, and they are reusable across pipelines because the owners recur: every platform has someone responsible for compute sizing and someone responsible for permissions. Examples of owner-defined categories are:

- **compute sizing**: memory or CPU requests too small for the work;
- **scheduling**: the platform could not place the work on available capacity;
- **trust configuration**: certificates, trust stores and TLS settings;
- **permissions**: identities, roles and grants;
- **quota**: rate limits and allocation limits imposed by a provider;
- **missing upstream data**: an input table, partition or file did not exist when the step ran;
- **network reachability**: a host could not be resolved or reached;
- **user-code exit**: the step's own code exited with an error;

- **packaging**: a dependency, class or module was missing from the deployed artefact;
- **provisioning**: a cluster or environment could not be created.

System B classifies failures this way. An ordered table of failure signatures maps each signature to an owning concern and to a remediation suggestion, and the owner is the unit of the classification. Knowledge for such tables accumulates in past investigations: incident reports written by engineers contain the mapping from symptoms to causes in natural language, and research has extracted structured knowledge from them, from root-cause information in problem-review documents (Saha & Hoi, 2022) to entities and attributes in incident descriptions (Shetty et al., 2021).

4.3.2 Ordered Evaluation, Most Specific First

The signature table is evaluated in order and the first match wins. Order matters because signatures overlap. A process killed with exit code 137 has also exited with a non-zero code; if the generic "non-zero exit" signature is evaluated first, a memory problem will be reported to the authors of the step as a bug in their code. The rule is to place the most specific signature first and the most generic last. A further rule caps the number of suggestions returned, so that an extract that matches many signatures produces a short, ranked answer instead of a list of every possibility.

Table 4.2 shows an ordered signature table. The patterns are simplified for illustration.

Table 4.2. Ordered signature table with owner mapping (illustrative; first match wins)

Order	Signature (simplified pattern)	Owning concern	Suggested first action
1	exit code 137 or OOMKilled or Java heap space	Compute sizing	Raise the memory request for the step; check input size
2	certificate verify failed or PKIX path building failed	Trust configuration	Check trust store and certificate expiry for the target host
3	Access Denied or permission denied or 403 from a cloud API	Permissions	Check the role bound to the step's identity
4	429 or quota exceeded or rate limit	Quota	Check provider quota; reschedule or request increase
5	table not found or partition does not exist or No such file on an input path	Missing upstream data	Check whether the upstream job ran and published
6	Connection refused or UnknownHost or connect timed out	Network reachability	Check endpoint, DNS and network policy
7	Class Not Found Exception or Module Not Found Error	Packaging	Rebuild the artefact; check dependency versions
8	Unschedulable or Insufficient cpu	Scheduling	Check capacity and placement constraints
9	cluster creation failed	Provisioning	Check environment templates and provider status
10	exit code [1-9][0-9]* (any other non-zero exit)	User-code exit	Read the step's own error output
Last	no signature matched	Unknown	Return the log extract unclassified for review

Source: constructed by the author for illustration. At most three suggestions are returned per failure.

4.4 The Two Axes Together

4.4.1 Where the Work Died and Who Fixes it

The milestone axis answers where in the pipeline the work stopped. The owner axis answers who fixes the cause. Neither substitutes for the other. A failure at the enrichment stage can belong to network reachability (the registry was unreachable), to missing upstream data (the reference table was not published) or to user code (the enrichment logic threw). A permissions failure can stop work at intake, at enrichment or at payment hand-off. The two classifications are orthogonal: knowing one does not determine the other.

Table 4.3 lays the two axes out as a matrix for the constructed pipeline. Each cell names what the combination means for the reader of an investigation's answer.

Table 4.3. Two-axis classification matrix for a constructed claims-processing pipeline (illustrative)

Owning concern ↓ / Business stage →	Intake	Validation	Enrichment	Adjudication	Payment hand-off
Missing upstream data	Source feed absent; nothing received	Rule tables not published	Reference data absent	Model inputs absent	Ledger extract absent
Network reachability	Submission channel unreachable	Rules engine unreachable	Registry unreachable	Scoring service unreachable	Ledger unreachable

Permissions	Intake identity lacks write grant	Job cannot read rules	Service cannot read registry	Task cannot read features	Task cannot post to ledger
Compute sizing	Large batch exceeds memory	Rule evaluation exceeds memory	Join exceeds memory	Scoring exceeds memory	Rare: hand-off is small
User-code exit	Parser bug	Rule logic error	Mapping logic error	Scoring logic error	Hand-off logic error
Unknown	Unclassified, review	Unclassified, review	Unclassified, review	Unclassified, review	Unclassified, review

Source: constructed by the author for illustration. Six of the owner categories are shown; the others follow the same pattern.

4.4.2 Routing Hand-Offs with the Classification Attached

The practical value of the two axes appears at the hand-off. When a failure needs a team other than the one investigating, the hand-off carries the stage, the owning concern, the evidence extract and any existing ticket references. System A attaches its milestone classification and the ticket references already logged against an instance when it routes a failure to another team, so that the receiving team does not re-derive where the work stopped or duplicate an investigation already in progress. A hand-off without classification restarts the investigation; a hand-off with classification continues it.

4.4.3 What the Two Axes Cost

Two classifications mean two tables to maintain. The translation table changes with every renamed signal, and the signature table with every new failure mode; both need tests and a periodic review of the unknown bucket. The owner list follows the organisation, so a reorganisation can invalidate it without a line of pipeline code changing. A team that cannot staff both tables does better with one, chosen by its readers: the milestone axis when the readers are business users who need to know where work stopped, the owner axis when the readers are engineers routing fixes. The second axis is added when hand-offs between teams begin to restart investigations, which is the cost the pair exists to remove.

4.5 Genuine Failure versus Dependency Failure

A pipeline step that ran and failed and a step that never ran because an upstream step failed are different events. The first has a cause in its own code, data or environment; the second has no cause of its own, and its remedy is to fix the upstream step and rerun. Mature schedulers encode the distinction in their state models. Apache Airflow, for example, gives a task the state upstream_failed when an upstream task failed and the trigger rule says the task needed it, separately from the state of a task that ran and failed (Apache Software Foundation, n.d.).

Conflating the two produces two kinds of damage. Reports overstate the number of failures, because one root failure appears as many; a run in which one early step failed can show a dozen failed steps. Remediation is misdirected, because the owners of the downstream steps are paged for failures they did not cause. System B separates the two at the sub-task level: it pulls every sub-task's state for a failed run and reports genuine failures and dependency failures in different groups, with the dependency failures pointing to the upstream step. The milestone and owner classifications of the previous sections apply only to genuine failures; a dependency failure inherits its classification from the upstream step that caused it.

4.6 Current State versus Historical Occurrence

4.6.1 Two Semantics behind One Phrasing

The question "what failed today?" has two answers, and both are correct for some reader.

- **Current-state semantics** is the reading that takes the latest run of each unit of work and reports the units whose latest run failed. A unit that failed in the morning and succeeded on a rerun in the afternoon is not reported. This answers "what is broken now?"
- **Historical-occurrence semantics** is the reading that reports every failed run in the window, including runs whose unit later recovered. This answers "how often did things fail, and where?"

A constructed example shows the size of the difference. Suppose 40 units of work ran today, 12 of them failed once each, and 9 of those 12 recovered on a single rerun. Under current-state semantics the answer is 3 failures; under historical-occurrence semantics it is 12. Neither number is wrong. An answer that gives one when the reader asked for the other is wrong, and nothing in the number tells the reader which semantics produced it.

4.6.2 An Explicit Trigger Test, and a Safe Default

System B names the two modes in its tools and its system prompt and selects between them by an explicit test on the wording of the question. Phrasings that ask about a period ("how many times", "every failure", "history", "at any point since") select the occurrence mode. Phrasings that ask about the present ("what is failing", "which are broken", "is it still down") select the current-state mode. When the wording is ambiguous, the current-state mode is used, the answer states which mode was used, and the other view is offered. The broader mode is never the default, because it reports recovered work as failed, and a reader who acts on it reruns work that has already succeeded.

When a question names a single run ("what failed in last night's run?"), the two semantics coincide for that run, and the test is not needed.

The safe default has a cost. A pipeline that fails and recovers repeatedly, a flapping pipeline, looks healthy under current-state semantics whenever its latest run succeeded. The answer discipline limits the risk: every answer names the mode it used and offers the other view, and a current-state answer adds a one-line note when a unit it reports as healthy failed more than once in the window. Figure 4.1 shows the test as a decision tree.

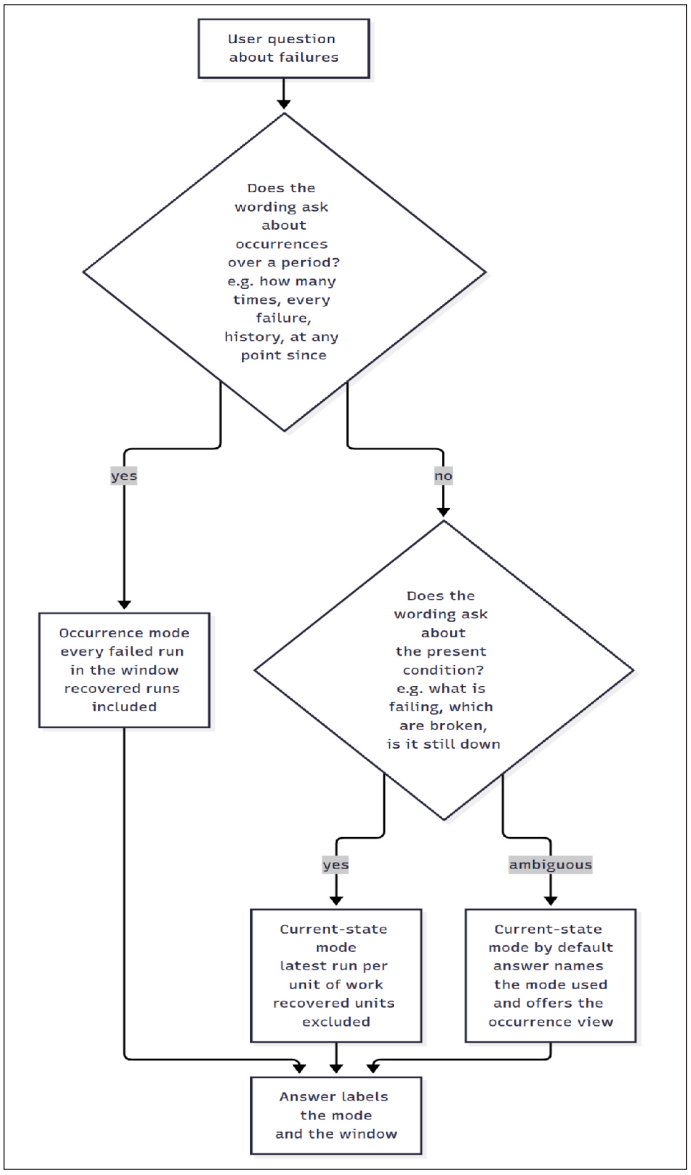


Figure 4.1. Decision tree for selecting current-state or historical-occurrence semantics. The broader occurrence mode is selected only by wording that asks about a period; ambiguous wording falls to the current-state mode, and every answer names the mode it used. *Source: generalised from the selection rule of System B.*

4.7 Procedure: Building a Taxonomy for Your Own Pipeline

4.7.1 Derivation Steps

1. **Name the business stages.** Write the stages a unit of work passes through in the words of the business, one line each, stating what the business can do with the work after each stage. Add terminal non-failure states such as "cancelled".

2. **Collect the signal vocabulary.** Gather the failure signals the pipeline emits: task names, topic names, exception classes, error codes and log templates, from configuration, code and past incidents.

3. **Map signals to stages.** Build the translation table (Table 4.1). Each signal maps to exactly one stage. Add the explicit unknown row.

4. **List the owners.** Write the owning concerns that exist in the organisation, using the owner categories listed earlier in this chapter as a starting set. Merge concerns owned by the same team; split concerns owned by different teams.

5. **Write the signature table.** For each owner, write the signatures that identify its failures in log extracts. Order the table from most specific to most generic (Table 4.2). Set a cap on suggestions.

6. **Separate dependency failures.** Confirm that the scheduler's state model, or the tool that reads it, distinguishes steps that ran and failed from steps that never ran.

7. **Name the two semantics.** Write the trigger test for current state and historical occurrence, with the safe default, into the tool descriptions and the system prompt.

8. **Test the tables.** For each row of each table, keep at least one known signal or log extract with its expected classification, and run the set with every change.

4.7.2 Sizing Guidance and Maintenance Rules

The stage set stays small and stable; the signal table grows. A stage set that changes often is a sign that stages are being defined by technical events instead of business hand-offs. The signature table is reviewed whenever the unknown bucket grows, and every new signature enters with a test. Signatures that have not matched anything for a long period are candidates for removal, because an obsolete signature placed early in the order can capture failures that belong to a later row.

4.7.3 A Worked Illustrative Example

In the constructed claims pipeline, the enrichment service fails one night. The scheduler reports the enrichment task as failed and four downstream tasks as upstream_failed. The investigation proceeds through the tables.

- The dependency split removes the four downstream tasks from the count of genuine failures and points them at the enrichment task.
- The translation table maps the enrichment task's signal, enrich.registry.timeout, to the enrichment stage: reference data could not be fetched.
- The bounded log extract matches signature 6 of Table 4.2, connect timed out, so the owning concern is network reachability.

- The question names one run ("what failed in last night's run?"), so the two semantics coincide. The answer reports one genuine failure, at enrichment, owned by network reachability, with four dependent tasks.
- The hand-off to the platform team carries the stage, the owner, the extract and the existing ticket reference.

The reader of the answer learns where the claims stopped and who is fixing it. The platform team receives a classified failure instead of five raw ones.

CHAPTER 5. THE DIAGNOSTIC RUNBOOK AND ANSWER DISCIPLINE

The tools of Chapter 2, the guards of Chapter 3 and the taxonomies of Chapter 4 are parts. An investigation is the order in which they are used and the rules that govern what the system may say at the end. The order is a runbook, an ordered sequence of checks with defined points at which the investigation stops; the rules are an answer discipline, a short set of constraints on what an answer may contain. Both bind a human operator as much as an agent.

5.1 Entry by Business Key

Investigations start from the identifier the person asking knows, and that is almost never the identifier the system needs. A business user knows a claim number, a region, a report identifier or a customer reference; the scheduler knows a run identifier, the orchestration engine a process-instance identifier, the warehouse a surrogate key. A runbook that begins by asking the user for an internal identifier has already failed the user.

The first step therefore resolves the loose identifier into the exact one. System A looks up a process instance by a known business key before it uses the process-instance identifier, because business users know the former and not the latter. System B resolves the user's informal name for a unit of work into the exact identifier the scheduler needs before it fetches any run. Resolution is a tool call like any other, with its own contract: it returns either one exact identifier, a short list of candidates with the attributes that distinguish them, or an explicit "not found". It never returns a guess.

Two rules make the step safe. When resolution returns several candidates, the agent asks which one is meant or reports all of them; it does not pick the most plausible. When resolution returns nothing, the investigation ends with that finding, stated as such, because every later step would be an investigation of the wrong thing.

Business keys are rarely unique in time. A claim number processed again after a correction, a region whose batch runs every night, or a report name shared by a daily and a monthly run all resolve to several technical identifiers. The resolution tool therefore takes a time window as part of the key, defaulting to the window the question implies ("last night", "this week"), and returns the candidates inside it with the attributes a person would use to choose: start time, pipeline, environment. The vocabulary mismatch described in Chapter 1 is resolved here, once, at the start of the investigation, instead of being rediscovered by each later step.

5.2 The Ordered Sequence and its Short Circuits

5.2.1 Terminal-State Gate First

The cheapest question with the largest effect on the rest of the investigation is whether there is anything to investigate. Once the run is identified, its terminal state is checked first. The terminal-state gate is the first check of the runbook: if the run succeeded, the investigation ends: the answer reports success, with a link to the run in the scheduler's own interface. If the run is still running, the investigation ends with a report of progress. Only a failed terminal state leads to further calls. System B applies this gate before any deeper analysis; a successful or running state short-circuits the whole investigation.

The gate does three things. It saves the cost of reading logs and tables for runs that need no diagnosis. It prevents a class of false answers in which the agent, having been asked "why did it fail?", finds something that looks like an error in the logs of a run that succeeded. And it fixes the semantics of the question early. Under current-state semantics the gate is applied to the latest run of each unit, so a unit whose latest run succeeded leaves the investigation. Under historical-occurrence semantics it is applied to each run in the window, so a run that failed is investigated even if a later rerun of the same unit succeeded.

5.2.2 Ordering the Remaining Checks

After the gate, the sequence is ordered so that each step either resolves the question or narrows it. The site reliability engineering literature describes troubleshooting as an application of the hypothetico-deductive method: given observations of a system and a theoretical basis for its behaviour, the investigator iterates between hypothesis and test (Jones, 2016). A runbook fixes the order of the first, most informative tests so that the iteration starts in the right place.

The order below is a synthesis of the two systems' runbooks, which begin at different points (Table 7.1 records the difference):

1. **Existing annotations and ticket references.** Before new investigation, read what is already known about the failure. System A reads the comments attached to a failed process instance, which surface any ticket numbers already logged against it, before it considers any further action. A failure under active investigation by another engineer needs a reference to that investigation; a second one would duplicate it.

2. **Sub-task states.** For a failed run, read the state of every sub-task, separate genuine failures from dependency failures (Chapter 4) and continue only with the genuine ones.

3. **Evidence extraction.** For each genuine failure, extract bounded evidence from its logs, as described below.

4. **Classification and recommendation.** Classify on both axes and map the evidence to suggestions through the ordered signature table.

5. **Answer.** Compose the answer under the answer discipline, with a verification path.

Encoding the sequence pays off in measurable ways in research settings. Flow-of-Action guided a multi-agent system with standard operating procedures for root-cause analysis and, on a testbed of 90 incidents covering 9 types of injected anomaly, raised accuracy to 64.01% against 35.50% for a ReAct agent without procedures (Pei et al., 2025). The comparison concerns one testbed and one procedure set; it shows that an explicit procedure gives a model's reasoning a structure that unconstrained planning does not find on its own.

5.2.3 Structured Collection before Reasoning

The sequence front-loads deterministic collection. Each of the first steps is a tool call with a defined result, and the model reasons over collected evidence instead of deciding, at each turn, what to look at next. RCACopilot, a research system for root-cause analysis of cloud incidents, uses the same arrangement: it matches incoming incidents to handlers that operators construct for each alert type and that collect the relevant diagnostic information automatically, and only then passes the collected information to a language model for root-cause prediction (Chen et al., 2024). A composite template of Chapter 2 plays the same role inside a runbook: a recurring collection, such as the stuck-claims template described there, becomes one step.

5.2.4 A Worked Trace

A constructed trace in the claims-processing pipeline of Chapter 4 shows the sequence end to end. A business user asks: "Why didn't yesterday's batch for the northern region finish?"

1. **Resolve.** The resolution tool receives "northern region batch" and the window "yesterday", and returns one run identifier. Had it returned two (a scheduled run and a manual rerun), the agent would have listed both.

2. **Gate.** The run-status tool reports the run as failed. The investigation continues.

3. **Known context.** The annotation tool returns one comment with a ticket reference opened that morning. The agent reports the reference and, because the user asked "why", continues without duplicating the ticket's work.

4. **Sub-tasks.** Of six sub-tasks, one failed and three show the state for "upstream failed". The three are reported as dependent on the one.

5. **Evidence.** The extraction tool returns two windows from the failed sub-task's log, 41 lines in total, with two signature matches.

6. **Classify.** The first match in table order is a missing-partition signature: owning concern, missing upstream data. The translation table maps the sub-task's signal to the enrichment stage.

7. **Answer.** The answer states, in a short table, that the batch stopped at enrichment because a reference partition was not published, that three later steps did not run as a result, that a ticket already exists, and that the suggested first action is to check whether the upstream publishing job ran. It ends with links to the run and to the log lines.

Five tool calls and two table lookups produce an answer the business user can read and the owning team can act on. None of the steps required the model to guess an identifier, a field or a cause.

Figure 5.1 shows the full sequence with its short-circuit points.

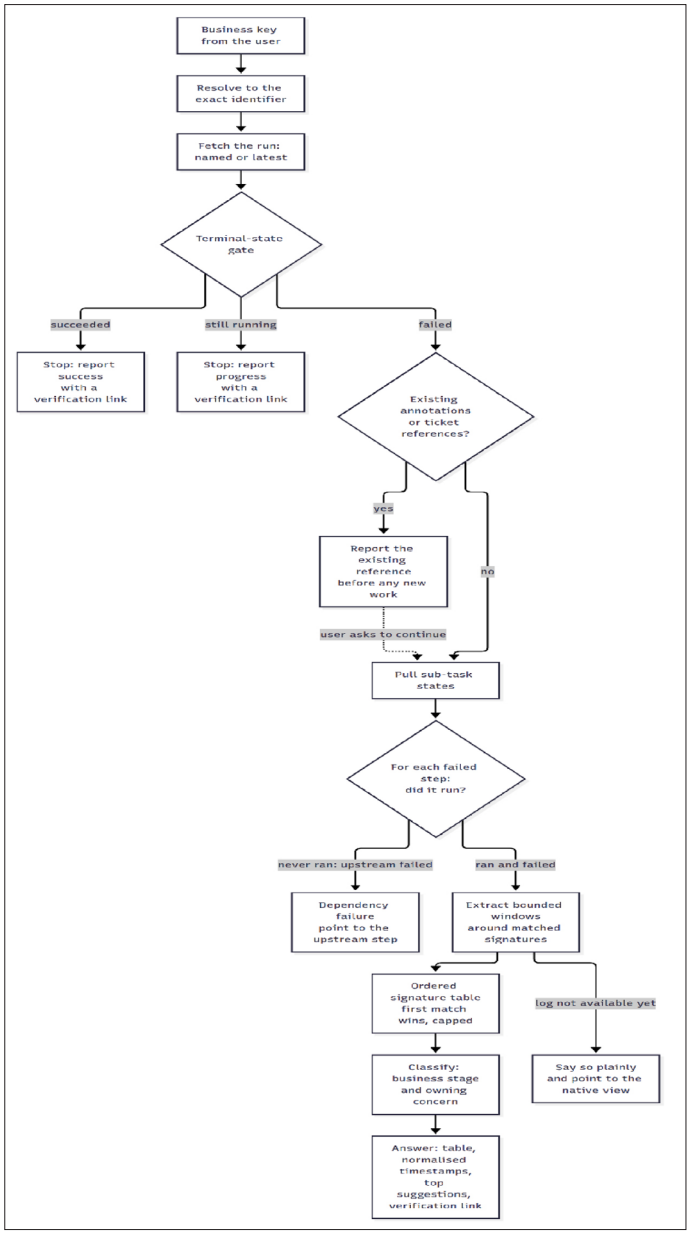


Figure 5.1. Investigation flow with short-circuit points: entry by business key, terminal-state gate, existing annotations, dependency split, bounded evidence, ordered signatures and a verified answer. *Source: generalised from the runbooks of System A and System B.*

5.3 Evidence Extraction

5.3.1 A Bounded Window around the Evidence

The raw log of a failed step can be too large to hand to a model and, when it fits, too large to be read well. Models use long inputs unevenly. Liu et al. (2024) found a U-shaped performance curve: models are better at using relevant information at the very beginning or end of their input context, and performance degrades when the relevant information sits in the middle. In their multi-document question-answering experiments, performance with relevant information in the middle could fall below the performance the model achieved with no documents at all. Levy et al. (2024) held the reasoning task fixed and varied only the length of the input, and found that accuracy averaged over the tested models fell from 0.92 to 0.68 at an input of 3,000 tokens, far below the models' technical limits. Context, in the words of one practitioner guide, is a finite resource with diminishing marginal returns (Anthropic, 2025a).

The extraction step therefore hands the model a bounded evidence window instead of the log. A bounded evidence window is an extract of the log around each line that matches a failure signature, merged and capped in code. System B extracts, for each genuine failure, a fixed number of lines before and after each line that matches an error signature, under a ceiling on the total number of lines. The extract keeps the evidence and drops the noise, and its size is predictable for every failure.

Figure 5.2 shows the extraction schema.

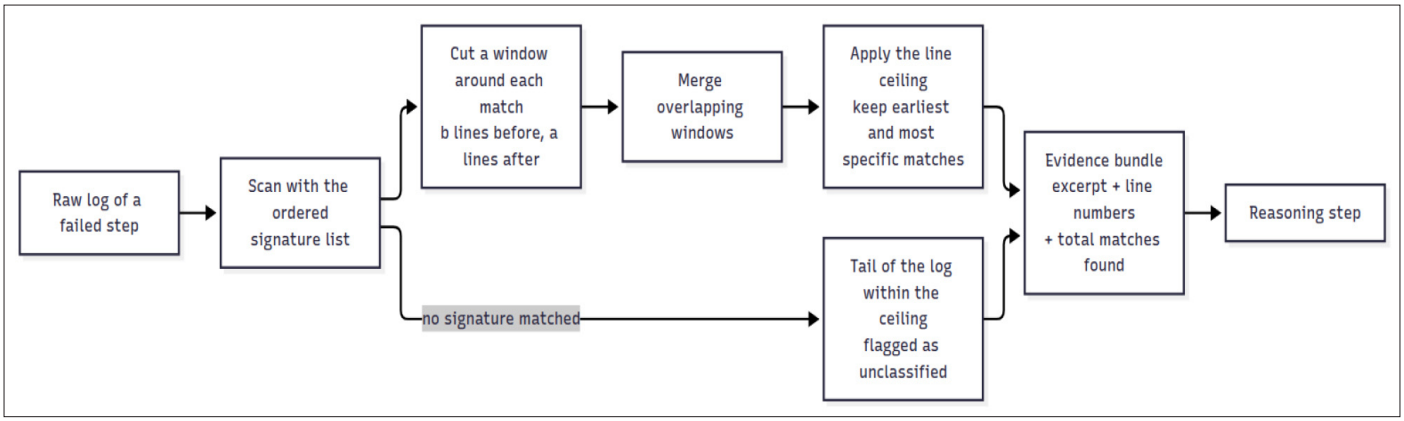


Figure 5.2. Evidence extraction schema: the raw log is scanned with the ordered signature list, a window is cut around each match, overlapping windows are merged, the line ceiling is applied, and the bundle carries line numbers and the total number of matches. *Source: generalised from the extraction step of System B.*

5.3.2 Choosing Window Size and Match Limits

Three parameters define the extraction, and each is a trade-off.

Lines before a match. Enough to show what the step was doing when the error appeared: the operation, its inputs, the last successful step. Ten lines is a reasonable starting value for application logs, and fewer suffice for structured logs where one line carries the full context.

Lines after a match. More than before, because stack traces, retries and the consequences of the error follow the match. Thirty lines is a reasonable starting value.

Total ceiling. The sum of all windows after merging, kept well inside the share of the context the answer can afford. A ceiling of 300 lines is a reasonable starting value; when the ceiling cuts windows, the earliest windows and those matching the most specific signatures are kept, and the bundle states how many matches were dropped.

The values above are constructed for illustration. A team derives its own from its logs: draw a sample of past failures whose cause is known, measure how many lines before and after the matching line an engineer needed to see the cause, take a value that covers most of the sample, and set the total ceiling from the share of the model's context that an answer can afford. The values are settings of the tool, never parameters the model chooses, and they are revisited when the log format or the model changes.

When no signature matches, the extract falls back to the tail of the log within the ceiling and is flagged as unclassified, so that the failure enters the unknown bucket of Chapter 4 instead of being reported as having no evidence. Log-analysis research supplies the tools to make signatures better over time: log parsers that turn messages into templates (Zhu et al., 2019) and the wider body of automated log analysis for reliability engineering (He et al., 2021).

5.4 From Evidence to Recommendation

5.4.1 Mapping Signatures to Suggestions

The ordered signature table of Chapter 4 does double duty at this step. The first matching signature classifies the failure by owning concern and supplies a suggested first action, and the cap on suggestions keeps the answer short. The table turns the extraction step's evidence into a recommendation without asking the model to invent one; the model's role is to explain the match in the context of the user's question and to report when nothing matched.

5.4.2 Retry Semantics as a Deliberate Decision

Retry is a tempting remediation to automate and a risky one. A retry of a non-idempotent operation repeats its side effects; a retry loop at several layers multiplies load on the failing component, as the retry amplification discussed in Chapter 3 shows. Helland (2012) states the condition that makes repetition safe with an example: withdrawing a billion dollars is not idempotent, while processing withdrawal XYZ for a billion dollars, if not already processed, is idempotent. The difference lies in the identifier and the check; the amount is the same in both.

The runbook makes retry a single attempt per action. System A's retry tool performs one attempt per request, fixed in code and not configurable by the model, and the decision to try again rests with the human or with the agent's next explicit step, never with a loop. The human decision point sits after the classification: a failure classified as missing upstream data or as a permissions problem will fail again on retry, and the classification tells the person deciding not to waste the attempt.

5.4.3 Engine-Specific Routing

When more than one execution engine is in play, for instance two incompatible major versions of a workflow engine running side by side during a migration, retry and status semantics differ by engine. The routing decision is made in code from properties of the identifier the investigation already holds, so that each engine receives its own sequence. The rule card for routing by identifier shape gives the conditions under which such dispatch holds and the point at which it becomes a liability.

5.5 Answer Discipline

5.5.1 Call a Tool, Never Guess

The first rule of the answer is that every fact in it comes from a tool result in the current investigation. A field the tools did not return is not inferred, however plausible. A status that was not read is not stated. The rule targets the failure mode surveyed as hallucination (Huang et al., 2025), and its justification is sharpened by an analysis of why models produce it: Kalai et al. (2025) argue that language models hallucinate because training and evaluation procedures reward guessing over acknowledging uncertainty. An operational agent has to be built against that incentive.

5.5.2 Absence of Data is a Valid Answer

The second rule makes "no data found" a final answer. When the log a question depends on is not yet available, or the table has no rows for the key, the answer says so plainly and points to the native interface where a human can look; it does not keep searching for something to report, and it does not compose a plausible report from what it has. System B's runbook ends this way when the underlying log or data is not yet available. Research on abstention treats a model's refusal to answer as a means of mitigating hallucination and improving safety, and organises the field around when a query, a model or human values call for it (Wen et al., 2025). In an operational answer the condition is concrete: the evidence does not exist yet.

5.5.3 Form of the Answer

Four further rules govern form. They come from System A's compact operating standard for its agent and apply unchanged to System B.

1. **Normalise timestamps.** Convert epoch values and engine-local times to one stated time zone.

2. **Use tables for multi-field data.** Several runs, several sub-tasks or several classifications go into a table with one row per item.

3. **Answer only what was asked.** A question about one run gets an answer about one run; additional findings are offered as a follow-up.

4. **Remember the key identifiers.** Within a conversation, the resolved identifiers are reused; the user is not asked for them twice.

5.5.4 Always Return a Verification Path

Every answer includes the path a human would follow to confirm it: a link to the run in the scheduler's interface, the query that produced a count, the line numbers of the log extract. System B includes a link to the native interface with every structured report, so that a person can verify without re-deriving the location. The verification path is what turns an answer from a claim into evidence, and it is the cheapest safeguard against a fabricated answer, because a fabricated answer has no path to return.

Table 5.1 collects the discipline into a checklist that can be applied to any answer, by an agent before it replies or by a reviewer reading a trace.

Table 5.1. Answer discipline checklist

#	Check	Fails when
1	Every fact traces to a tool result in this investigation	A field, status or count appears that no tool returned
2	No absent field is inferred	A missing value is filled from context or plausibility
3	Absence of data is stated as the answer	The answer reports a finding the evidence does not contain
4	The semantics (current state or historical occurrence) and the window are named	A count is given without saying which semantics produced it
5	Truncation is stated with the true total	A capped result is presented as complete

6	Genuine and dependency failures are reported separately	Downstream steps are counted as failures
7	Classification on both axes is given for genuine failures	Only an exception class is reported
8	Timestamps are normalised to one stated time zone	Mixed epoch and local times appear
9	Multi-field data is in a table	Several items are described in running prose
10	Only the question asked is answered	Unrequested findings are appended
11	A verification path is included	The reader cannot confirm the answer without re-deriving it

Source: compiled by the author from the answer rules of System A and System B and Chapters 3–5.

5.6 The Runbook as a Human Checklist

Nothing in the runbook or the answer discipline depends on the reader being a model. Resolve the identifier; check the terminal state; read what is already known; separate the failures that happened from the ones that were caused; read a bounded window of evidence; classify where the work stopped and who fixes it; answer only what was asked, with a path to check it. Each step is a sound instruction for a new engineer on their first on-call rotation.

The checklist literature explains why this matters. Gawande (2010) documented the use of ordinary checklists in medicine, aviation, construction and other fields where experienced practitioners work under pressure. The site reliability engineering literature makes the same point for operations: a systematic troubleshooting method beats intuition under pressure (Jones, 2016).

The equivalence also runs in the other direction. A procedure that an agent can follow has been written down in enough detail that a human can follow it too, and a procedure that a new engineer cannot follow is unlikely to be followed reliably by a model. Encoding the runbook in tool descriptions and a system prompt is therefore also a way of documenting the team's investigative practice. The agent becomes one reader of a procedure that the team owns, reviews and improves, and the procedure stays useful on the day the agent is unavailable.

CHAPTER 6. NAMED DESIGN RULES

Four rules recur in the code of the two systems without appearing in their documentation as rules: one in both systems, and three in System A, the system that performs the operations they govern. They are visible only as repeated structures in the code: a read before every action, a warning instead of an exception after a particular write, a loop that catches per item, a dispatch on the form of an identifier. Named, each becomes a tool for design review: a reviewer can ask whether a change honours "liveness check before trust" in the time it takes to say the name. A formulation of these four as named rules was not identified in the reviewed literature. Each is stated below in a fixed card format, one per section, and a procedure for finding such rules in other codebases follows the four cards.

Pattern writing has a long tradition of fixed forms. Alexander et al. (1977) presented their pattern language as a set of named, interlinked solutions to recurring problems, and Gamma et al. (1994) brought the practice into software by systematically naming, explaining, evaluating and cataloguing recurring designs. Meszaros and Doble (1997) collected the conventions of pattern writing itself into a pattern language, in which every pattern has a set of mandatory elements, among them a problem, a context and the forces that bear on the solution, and in which evocative names help readers grasp the essence of a solution. Architecture decision records apply a similar discipline to individual decisions, recording each with its context, decision, status and consequences (Nygard, 2011).

A rule card is the fixed format in which each rule of this chapter is stated. The card, shown in Table 6.1, borrows from both traditions. It is shorter than a full pattern and more prescriptive than a decision record, because its purpose is to be checked during review.

Table 6.1. Rule card template

Element	Content
Name	A short imperative or noun phrase that can be used in conversation
Statement	The rule in one sentence, testable against code
Rationale	Why the rule holds: the failure it prevents
Failure mode when violated	What goes wrong, and how it shows up in operation
Cost	What the rule costs in latency, load, complexity or consistency
Conditions of application	When the rule applies
When not to apply	The conditions under which following the rule is wrong
Evidence	Where the rule was observed: System A, System B, or both

Source: compiled by the author, adapting the mandatory pattern elements of Meszaros and Doble (1997) and the decision-record sections of Nygard (2011).

6.1 Liveness Check before Trust

The mechanism of this rule was described with the guards of Chapter 3. As a rule, it governs every place where code acts on information it did not just read from the authority.

Table 6.2. Rule card: liveness check before trust

Element	Content
Name	Liveness check before trust
Statement	Before acting on a record obtained from a secondary store, confirm against the system of record that the entity the record describes is still live; if it is not, do not act, and report the record as stale.
Rationale	Secondary stores (warehouse tables, caches, incident lists) describe the world at the moment they were written. An action taken on that description repeats or contradicts work the system of record has already completed or cancelled.
Failure mode when violated	Retries of finished processes, annotations on cancelled work, reports that list resolved failures as open. The errors look like correct actions on correct-looking data, so they are found late.
Cost	One additional read against the system of record per action; added latency; load on the authority in proportion to the action rate. A window remains between check and action, so the rule narrows the race without closing it.
Conditions of application	The action changes state or is reported to a person as current; the record came from a store other than the authority; the authority can be queried cheaply enough to do so per action.
When not to apply	The secondary store is itself the authority for the fact in question; the action is a read-only report labelled with its as-of time; the authority is unavailable, in which case the correct behaviour is to refuse the action and keep the check.
Evidence	System A confirms that a process instance is still active in the orchestration engine before acting on an incident row from the warehouse, and cleans up the row when the instance is no longer active. System B applies the same discipline to schemas and snapshots: the real field names and the latest snapshot are confirmed before any ad-hoc query.

Source: compiled by the author from the designs of System A and System B.

The cost entry deserves a remark. The check does not make the action atomic with the check; a process can finish between the read and the retry. The rule's value lies in removing the large class of stale records that are hours or days old; a race of milliseconds remains, and the rule leaves it to the next measure. Where the race itself matters, the action needs a conditional form at the authority, such as an idempotent operation keyed on the state it expects.

In review. The reviewer lists every act tool and, for each, looks for a read of the system of record between the moment the record was obtained and the moment the action is sent. The signal in code is a status read whose result gates the action; its absence is the violation. A second question asks what the tool does when the authority cannot be reached. The only acceptable answer is a refusal with a typed error, because an action taken without the check is the very action the rule exists to prevent.

6.2 Best-Effort Secondary Write

User-facing actions in operational systems often write twice: once to a durable record, such as an audit or incident table, and once back to the live system, such as a comment or a suspension flag on the running process. The two writes do not have the same standing.

Table 6.3. Rule card: best-effort secondary write

Element	Content
Name	Best-effort secondary write
Statement	When a user-facing action writes both an authoritative durable record and a write-back to a live system, the authoritative write is always attempted and decides the outcome; the write-back is best effort, and its failure is logged as a warning without failing the action.
Rationale	The durable record is what later investigation, audit and reporting rely on. The write-back is a convenience for people looking at the live system. Coupling the user's action to the convenience lets a failure in an auxiliary system block the record that matters.
Failure mode when violated	Actions fail because a live system was briefly unreachable; users retry and create duplicates; or, worse, code that writes the live system first skips the durable record when the live write throws.
Cost	The durable record and the live system can disagree. The disagreement must be visible (the warning is monitored) and repairable (a reconciliation job, or a retry of the write-back that is safe to repeat).

Conditions of application	The two writes have different authority; the write-back can be repeated safely; a missed write-back has a bounded, recoverable effect.
When not to apply	The second write is required for correctness, for example a suspension that prevents duplicate processing. Such a write is not secondary: it must succeed with the primary write, or the primary write must be compensated.
Evidence	System A treats its warehouse write as authoritative and always attempted, and makes the write-back to the workflow engine (a comment or a suspension) best effort, logging a warning on failure instead of failing the user-facing action.

Source: compiled by the author from the design of System A.

The rule has respectable ancestors. The end-to-end argument holds that a function can be implemented completely and correctly only with the knowledge and help of the application at the end points of a system (Saltzer et al., 1984); the durable record is the end point that knows whether the action happened, and the write-back is an intermediate convenience. Sagas address the converse case, when several writes must all take effect: a long-lived transaction is written as a sequence of transactions, and either all complete or compensating transactions are run to amend a partial execution (Garcia-Molina & Salem, 1987). The "when not to apply" entry of the card marks the boundary between the two: a write that can be lost and repaired is best effort, and a write that cannot be lost belongs in a saga or a transaction. The write-back must also be safe to repeat for the repair to work, which is the property that Hohpe and Woolf (2003) name the idempotent receiver: a receiver that can safely receive the same message multiple times.

In review. The reviewer finds each user-facing action that writes to more than one place and asks two questions: which write is the authority, and what happens to the user's action if the other write fails. The signal in code is the order of the writes and the handling around the second one: the authoritative write comes first and its failure fails the action; the secondary write is wrapped so that its failure produces a warning that someone monitors. A secondary write whose failure is swallowed without a warning violates the rule as much as one whose failure aborts the action, because the divergence it creates can no longer be repaired.

6.3 Partial Success by Construction

Operations on many items at once, such as retrying a list of failed runs, refreshing a set of incident records or annotating a batch, raise the question of what the operation returns when some items succeed and others fail.

Table 6.4. Rule card: partial success by construction

Element	Content
Name	Partial success by construction
Statement	A bulk operation processes each item independently, catches failures per item, and returns a per-item outcome with totals (succeeded, skipped, failed); the batch as a whole is reported as failed only when every item failed.
Rationale	Items in an operational batch are independent units of work. An all-or-nothing result hides which items succeeded, invites a retry of the whole batch and repeats work that was already done.
Failure mode when violated	One bad item aborts the batch; or the batch reports failure while most items succeeded, and the caller retries everything; or the batch reports success while some items failed, and the failures disappear.
Cost	A richer result shape that every caller must read; per-item error handling; a definition of which outcome classes count toward failure.
Conditions of application	Items are independent; each item's operation is safe to repeat; callers can act on per-item results.
When not to apply	The items form a unit that must succeed or fail together, such as the steps of one financial transfer; then the operation needs transactional or saga semantics, not partial success.
Evidence	System A's bulk operations iterate items independently, catch per item and return per-item counts with a total, and degrade the whole batch to a failure status only when every item failed.

Source: compiled by the author from the design of System A.

6.3.1 Degradation Rules

The rule needs an explicit statement of when a batch counts as failed overall, and the simplest useful statement is the one in the card: only when every item failed. Two refinements are common. An operation can report failure overall when the share of failures exceeds a threshold that the caller sets, which suits callers who treat a mostly failed batch as a signal to stop. An operation can distinguish items that failed from items that were skipped, for example because the liveness check found them already finished; skipped items are neither successes nor failures and are reported as a third class.

Partial success interacts with the ceilings of Chapter 3. A bulk operation that accepts an unbounded list of items is an unbounded collection endpoint; partial success makes the operation correct for any input, and the ceiling makes it safe for large ones. Both are needed.

In review. The reviewer finds each operation that accepts a collection and reads its loop. The signal is a per-item try-block whose catch records the item's outcome and continues, and a result type that carries per-item outcomes and totals. Two further questions complete the check: what the operation returns when every item fails, and what the largest accepted collection is. An operation that returns success when every item failed hides a total outage; an operation with no maximum is the unbounded endpoint of Chapter 3.

6.4 Routing by Identifier Shape

During migrations, two versions of an execution engine often run side by side, each with its own incident model, its own retry semantics and its own form of identifier. Code that handles an identifier must send it to the right engine.

Table 6.5. Rule card: routing by identifier shape

Element	Content
Name	Routing by identifier shape
Statement	When incompatible backends coexist and their identifiers have disjoint forms, dispatch each request to the backend whose identifier form it matches, and give each backend its own operation sequence.
Rationale	The identifier is the one piece of information every request carries. When its form already distinguishes the backends, routing on the form needs no lookup and keeps the agent from applying one backend's semantics to the other's entities.
Failure mode when violated	A retry sequence written for one engine is applied to the other's instance and fails or, worse, partially succeeds; the agent must be told which engine to use and sometimes guesses.
Cost	A hidden coupling between identifier format and backend; the routing table must change when a backend changes its identifier scheme.
Conditions of application	The set of backends is small and closed; their identifier forms are disjoint by construction (for example a numeric sequence and a universally unique identifier in its standard textual form); the form is checked strictly, not by loose resemblance.
When not to apply	A third backend adopts a form that overlaps an existing one; identifiers are migrated between backends; the routing decision has security consequences, in which case an explicit lookup or an explicit backend tag replaces inference from form.
Evidence	System A routes by identifier form between two coexisting major versions of its workflow engine, one of which issues numeric identifiers and the other string or UUID identifiers, and gives each its own retry sequence.

Source: compiled by the author from the design of System A.

6.4.1 Boundaries of the Rule

The rule rests on a property of the identifier schemes, and the engines inherit it. A universally unique identifier is 128 bits long and has a standard textual representation defined by its specification (Davis et al., 2024), and a numeric sequence has none of its structure, so a strict check can tell the two apart without error. The rule becomes a liability as soon as that property weakens. Engines that switch to a new identifier scheme in a later version, migration tools that carry identifiers from one engine to another, and string identifiers that happen to be numeric all break the inference silently: the request is routed, the wrong semantics are applied, and nothing in the request reveals the mistake.

The rule is therefore a migration-period rule. It is the cheapest correct design while two engines coexist and their identifier forms are disjoint, and it should be replaced by an explicit backend tag, carried with the identifier from the moment it is first resolved, once a third engine or a migration of identifiers is planned.

In review. The reviewer finds the dispatch and checks that the shape test is strict (a full pattern match, not a test such as "contains a hyphen"), that every branch ends in a backend-specific sequence, and that an identifier matching no known form is refused instead of sent to a default backend. A default branch is the first violation to look for, because it turns an unrecognised identifier into a silent wrong route.

6.5 Naming Implicit Rules in Your Own Codebase

The four rules were found in working code before they were stated. Each was visible as a repeated structure in working code before it had a name. The procedure below finds such rules in other codebases.

1. **Look for repeated defensive structures.** Reads that precede writes, try-blocks inside loops, writes whose exceptions are caught and logged, dispatches on string format, guard clauses that refuse before calling out. Each repetition is a candidate.

2. **Ask why the structure is there.** For each candidate, find

the incident, review comment or design discussion that introduced it. A structure with no recoverable reason may be accidental; a structure with a reason is a rule waiting for a name.

3. **Look for counter-examples.** Find the places in the same codebase where the structure is absent. If the absences are bugs, the rule is real; if they are deliberate, the conditions of application need refining.

4. **Write the card.** Fill every element of Table 6.1. A rule that cannot be given a "when not to apply" entry is not yet understood.

5. **Name it.** Choose a name short enough to be used in a review comment and specific enough not to collide with established terms; an evocative name, in the pattern-writing tradition, lets readers grasp the essence of the solution (Meszaros & Doble, 1997). "Liveness check before trust" says what is checked and when.

6. **Record and reference it.** Store the card beside the code it governs, and link it from the architecture decision records that depend on it (Nygard, 2011), so that the rule is found by people who change the code.

A constructed example shows the procedure at work. A reviewer reading the act tools of an operational service notices that three of them begin with the same call: a status read against the orchestration engine whose result decides whether the tool continues. Asking why, the reviewer finds a review comment on the first of the three, written after an incident in which a stale record led to the retry of a finished process. Searching for counter-examples, the reviewer finds a fourth act tool, added later by a different engineer, that does not perform the read; its history contains a bug report about annotations on cancelled work. The repeated structure has a reason and its absence produces a bug, so the rule is real. The card is written, the fourth tool is fixed, and the rule gets a name that the next review can use.

Naming has two returns in review. It makes a rule checkable: a reviewer can ask whether a change honours a named rule, and a missing structure becomes visible as a violation instead of an omission. It makes design knowledge transferable: a rule that a second team can name is a rule that a second team can adopt. Treating an architecture as a set of design decisions, each with its rationale, is an established way of keeping such knowledge from evaporating as teams change (Jansen & Bosch, 2005). The four cards of this chapter are a small instance of that practice.

6.5.1 How the Four Rules Interact

The rules are stated separately, and they work together. The liveness check produces the "skipped" outcome that partial success must report as a third class; without the check, stale items are either retried wrongly or counted as failures. The best-effort secondary write depends on the idempotency that also makes partial batches safe to repeat; a write-back that cannot be repeated cannot be repaired, and a batch whose items cannot be repeated cannot be retried for its failed subset. Routing by identifier shape decides which authority the liveness check must ask; a check sent to the wrong engine reports an entity as missing and turns a live process into a skipped item. A design review that checks one rule should therefore check its neighbours, and the cards' conditions of application are where the dependencies are stated.

CHAPTER 7. CONVERGENCE, TRANSFER, AND ADOPTION

Two systems, built in different domains and team contexts without shared code, exhibit the method's structure in graded forms: some decisions recur unchanged, some are explicit in one system and partial in the other, and three of the named rules appear only where the operations they govern exist. What that recurrence can show, which explanations compete with "method", where the method transfers and how a team adopts it all follow from that grading.

7.1 Two Systems, Two Domains, Independent Construction

7.1.1 System A

System A is an incident-investigation agent over a workflow orchestration engine and its associated stores. Its users are engineers and business users who need to know why a long-running business process stopped. It reaches the orchestration engine and a warehouse of incident records through a small tool layer, investigates a failed process instance by a business key, classifies the failure by the business stage at which the process stopped, and supports a single-attempt retry under human decision. It was built first, in one team, for one family of business workflows.

7.1.2 System B

System B is a multi-purpose assistant over a scheduled data pipeline. Its users are the engineers, leads and business stakeholders of the pipeline, and it answers questions across four areas: runtime status of scheduled runs, source code, versioned configuration and warehouse data, with documentation search alongside. Failure investigation is one of its capabilities; it also locates where a business rule is applied for a given pipeline and which code repositories implement it. It was built later, in a different team, over a different execution platform.

7.1.3 Why the Comparison is Meaningful

Both systems come from the author's own practice as a software engineer. "Independent construction" in this chapter has a precise and limited meaning: the systems were built separately, in different domains, for different teams, without code reuse. The domains differ: one system serves business-process orchestration, the other batch data

processing, with different failure vocabularies and different users. The team contexts differ: different conventions and different reviewers. And no tool, guard or table of System B was copied from System A. What the comparison examines is whether a design persisted when its domain, team and code base changed.

7.2 The Convergence Claim

7.2.1 Point-by-Point Comparison

Table 7.1 compares the systems decision by decision and rule by rule, and grades each row by the form in which the structure appears.

Table 7.1. Convergence comparison across System A and System B

Structure	System A (incident investigation over a process orchestrator)	System B (multi-purpose assistant over a data-pipeline scheduler)	Form of recurrence
D1. Narrow tool per external source	Narrow, stateless tools over the orchestration engine and the incident warehouse; act operation separate	One read-only tool per external system: run status, warehouse, source repository, configuration, documentation	Same form in both
D1. Routing knowledge in descriptions	Tool descriptions as the single source of truth for when to call each tool	Same; authority between overlapping sources stated per phrasing	Same form in both
D2. Safety contracts in code	Some boundaries held by convention at the time of construction	Statement guard, pinned scope and payload caps in tool code; truncation reported with true totals	Explicit in B, partial in A
D3. Failure taxonomy by meaning	Business-milestone axis: signals translated to business stages, explicit unknown bucket	Owner-concern axis: ordered signature table, first match wins, capped suggestions	Complementary: one axis in each system; the pair is a synthesis
D3. Genuine vs dependency failure	Not a separate classification	Explicit split at sub-task level	B only
D4. Current state vs historical occurrence	Not named as a rule	Named modes, trigger test, safe default	B only
Runbook	Business-key entry; open incidents; existing comments and tickets first; classification; single-attempt retry	Name resolution; terminal-state gate; sub-task split; bounded log windows; signature table; verification link	Same pattern, different first steps; the order of Chapter 5 is a synthesis
Answer discipline	Tool not guess; no inferred fields; normalised timestamps; tables; answer only what was asked	Same, plus plain "not available yet" and a link to the native interface	Same form in both
Rule: liveness check before trust	Process confirmed active before acting on a warehouse incident record	Schema and latest snapshot confirmed before any ad-hoc query	In both, on different objects
Rule: best-effort secondary write	Durable record authoritative; write-back to the engine best effort	No live write-backs in its scope	A only (operation absent in B)
Rule: partial success by construction	Bulk operations return per-item outcomes and totals	No bulk act operations in its scope	A only (operation absent in B)
Rule: routing by identifier shape	Two engine versions dispatched by identifier form	One execution platform	A only (operation absent in B)

Source: compiled by the author from the designs of System A and System B.

Read row by row, the table supports a graded statement. The tool-layer decision, its routing discipline and the answer discipline recur in the same form. The safety decision recurs, explicit in System B and partial in System A. The taxonomy decision recurs as two complementary axes, one per system, and the pairing of the axes is this book's synthesis. The separation of semantics is explicit in System B only. One named rule recurs in both systems on different objects; three are observed in System A, the only system that performs the operations they govern, and nothing in System B contradicts them.

7.2.2 What the Comparison Shows

This graded recurrence is the principal evidence the book offers for the method. The structures in the "same form" rows survived a change of domain, team and code base, so they are not peculiar to one setting. Design-science research treats a designed artefact, including a method, as a research contribution whose utility must be demonstrated (Hevner et al., 2004); the graded table is the demonstration offered here, and the transfer checklist and the adoption gates later in this chapter turn the method into checks a team can run on its own pipeline.

Three readings of the recurrence compete with "method", and each changes what it means.

1. **Problem-forced convergence.** The structure may be

what any careful design of an agent over several operational systems arrives at. If so, the recurrence shows that the structure belongs to the problem class, which is the scope at which the method is offered.

2. **The shared protocol.** Both systems expose their tools through the same protocol, and a protocol can impose structure. The Model Context Protocol, however, prescribes the form of a tool definition and leaves its granularity open: a server may expose one tool that answers every question or one per source, and the specification treats behavioural annotations as untrusted hints (Model Context Protocol, 2025b). Nothing in the protocol requires a statement guard, a payload ceiling, a failure taxonomy or a named distinction between current state and historical occurrence. The protocol explains why both systems have tools. The shape of the tools, guards and tables needs another explanation.

3. **Common period and platform.** Both systems were built in the same years, against models with similar tool-use behaviour and on similar cloud platforms, and shared external conditions favour similar designs. The method is accordingly offered for agents over operational systems with the tool-use behaviour of current models.

7.2.3 What this Evidence Is and Is Not

The convergence is structural corroboration: a graded record of which decisions recur, in which form, across two systems built separately. Two further readings compete with "method", carried-over design and post hoc abstraction, and they are also the limits of the evidence, stated here once. Both systems were designed in the author's own practice, and experience with the first informed the second; the comparison cannot separate the method from that carried-over design, and it does not show that other engineers would reach the same decisions. The four decisions were abstracted after both systems existed, and what counts as "the same structure" was judged within the same practice that built them, so the comparison describes recurrence and does not test a prediction; Table 7.1 grades each row to make that judgement inspectable. A case selected in advance to predict similar results, a literal replication in case-study terms, would be the stronger test (Runeson & Höst, 2009), and a third system chosen that way is the next step. The comparison is not a controlled experiment either; case studies do not generate the same results on causal relationships as controlled experiments do (Runeson & Höst, 2009). No measurement of investigation quality, time or cost under matched conditions, no evaluation of either system's answers and no comparison with alternative designs is reported in this book. The systems cannot be inspected by the reader: every statement of the form "System B does X" is an account of a design given at the level of the class of problem the system addresses, and the numbers in the book's tables and examples are constructed for illustration. These are the boundaries within which the method's claims are made.

7.3 Where the Systems Differed, and Why

7.3.1 Variation Attributable to Domain

Three differences trace to the domains.

1. **Number of sources.** System B reaches more external systems than System A, because a data pipeline's behaviour is distributed across code, versioned configuration and warehouse tables as well as the scheduler. Under the first decision, its tool layer is correspondingly wider, and the authority statements between overlapping sources (Chapter 2) appear first in System B because overlapping sources appear first there.

2. **Breadth of the question space.** System A answers one family of questions, why a process stopped and what to do about it. System B answers questions that are not about failures at all, such as where a business rule is applied or which repository implements a step. The same tool layer served both kinds of question without per-question interfaces, which is the property the first decision is meant to deliver.

3. **Failure vocabulary.** A process orchestrator's failures are business-stage events; a data scheduler's failures are infrastructure events in containers and clusters. Each system built the taxonomy axis its domain forced on it first: System A the milestone axis, System B the owner axis.

7.3.2 Variation Attributable to Sequence

The remaining differences trace to the order in which the systems were built. In System B the caps are constants in the tools' code, capped listings report their true totals, and warehouse queries pass a statement guard and a pinned scope; in System A, some of the corresponding boundaries were held by convention at the time of construction. The second system moved into code what the first had left to convention, and the second decision of the method states that move as a rule.

The separation of current state from historical occurrence followed the same path. System B served a user population that asked about failures in both senses every day, and it named the two modes, gave them a trigger test and fixed a safe default. System A's users asked about open incidents, and the distinction had no occasion to become a rule there. The second system made explicit what the first had left implicit.

7.4 Conditions of Transfer

7.4.1 Properties of a Target Environment

The method applies where three properties hold together.

1. **Several systems of record.** The facts an investigation needs are held by more than one system, each authoritative for part of them. With one system of record, a natural-language interface to that system is simpler and sufficient (Chapter 1).

2. **A joint state that carries business meaning.** The conditions that matter to the people asking are defined over several systems together, the k-ary cross-system invariants of Chapter 1. Without such conditions, monitoring each system separately answers the questions that are asked.

3. **Investigations that cross system boundaries in an order.** Answering a typical question requires reading several systems, where the result of one read decides the next. Without ordering, a dashboard or a report is cheaper.

Two further properties are practical preconditions. The systems of record must expose programmatic read access that a tool can call with bounded cost; and a team must own the tool layer, its contracts and its tables, because each artefact of the method decays without an owner.

7.4.2 Where the Conditions hold, and where they do not

The conditions hold in many domains beyond the two cases. Claims and payment processing, continuous integration and delivery pipelines, machine-learning training and deployment pipelines, healthcare data integration and logistics tracking all move units of work through stages owned by different systems, and all face questions whose answers span a scheduler, a store and the code that connects them. A design's transfer to a new context is itself a hypothesis to be checked against that context's conditions, and Table 7.2 turns the conditions into a checklist for that check.

The conditions fail in four kinds of environment. Single-store systems, where one database holds every fact, are served by a text-to-SQL interface without a tool layer. Real-time control loops, where decisions are made in milliseconds, leave no room for an investigative agent in the loop. Systems without programmatic access to their state, where operators read screens, cannot be wrapped in tools without first building that access. And exploratory analytics without a notion of failure or business stage has no use for the taxonomies, though it may still benefit from the tool layer and its guards.

7.4.3 A Worked Transfer Check

A constructed example applies the checklist of Table 7.2 to a continuous-delivery pipeline. Its questions ("why did last night's release not reach staging?") draw on the build system, the artefact registry, the deployment controller and the change history of the configuration repository, so the first condition holds. The business condition "every change merged before the cut-off is deployed to staging by morning" needs the source repository, the build system and the deployment controller together, so the second holds. A typical investigation reads the deployment status first, then the build of the artefact it tried to deploy, then the change that introduced a failing test, an ordered enquiry, so the third holds. Each system exposes an API with rate limits, which satisfies the fourth. Stages can be named in terms a release manager uses (merged, built, verified, deployed to staging, promoted), which meets the fifth condition, and owners can be named (build infrastructure, test authors, platform, security), which meets the sixth. The method transfers, and the adoption path starts at Stage 1 with read tools over the four systems. The one condition that needs a decision is the eighth: deployment controllers offer act operations such as rollback, and a team adopting the method decides at Stage 4 whether the agent may propose them or only recommend them to a person.

Table 7.2. Transfer conditions checklist

#	Condition	How to check	If it fails
1	Facts needed by investigations are held by two or more systems of record	List the systems consulted in the last ten investigations	Use a single-source interface
2	Business conditions are defined over several systems together	Write three business questions; mark which systems each needs	Monitor systems separately
3	Typical investigations read systems in an order that depends on intermediate results	Reconstruct two past investigations step by step	Build a report or dashboard
4	Each system offers programmatic read access with bounded cost	Identify the API and its limits for each system	Build read access first
5	Business stages of the work can be named in business terms	Draft the stage list (Chapter 4)	Start with the owner axis only
6	Owners of failure causes can be named	Draft the owner list (Chapter 4)	Start with the milestone axis only
7	A team will own the tool layer, guards and tables	Name the owner	Defer adoption
8	Act operations, if any, can be made single-attempt and gated	Inspect each act operation	Keep the agent read-only

Source: compiled by the author.

7.5 Adoption Path

7.5.1 From a Bespoke Assistant to a Reusable Tool Layer

The adoption path starts from a bespoke assistant, with one prompt, a few functions written for its questions and one consumer, and converts it into the structure of the method in stages, each ending in a review gate. Industry forecasts give a reason for the gates. Gartner predicts that over 40% of agentic AI projects will be canceled by the end of 2027, due to escalating costs, unclear business value or inadequate risk controls (Gartner, 2025). Each of those three

causes corresponds to a stage that is easy to skip: cost to the ceilings, business value to the taxonomies that make answers meaningful, and risk controls to the guards.

Figure 7.1 shows the stages and gates.

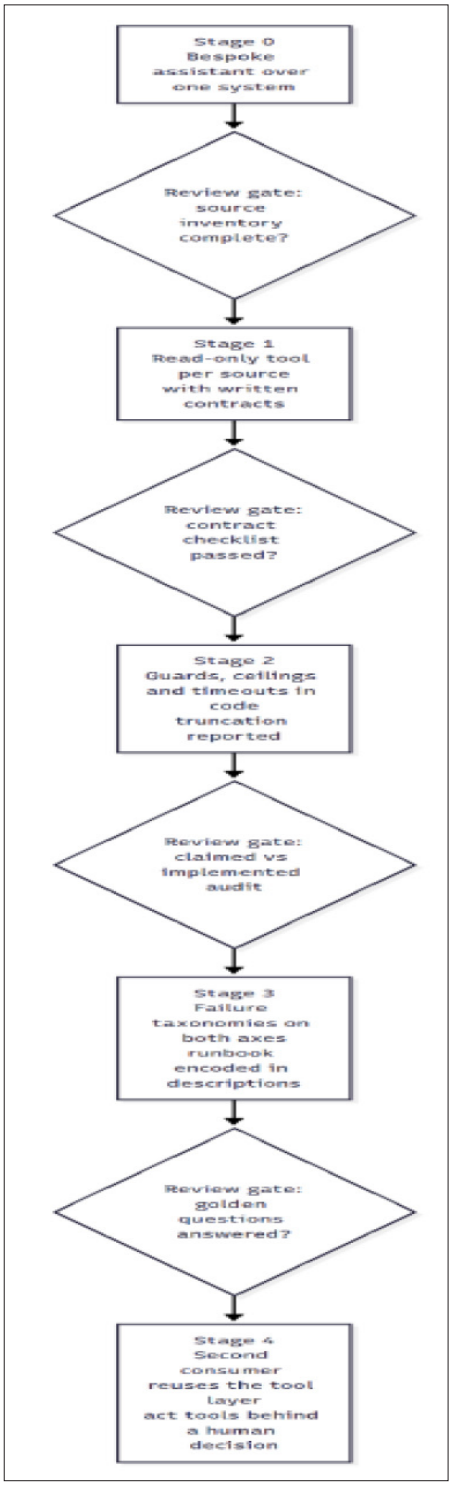


Figure 7.1. Staged adoption roadmap from a bespoke assistant to a reusable tool layer with several consumers, with a review gate between stages. *Source: compiled by the author.*

1. **Stage 0: the bespoke assistant.** The starting point. Its functions and prompt are inventoried; nothing is discarded yet.

2. **Stage 1: read tools with contracts.** One read-only tool per system of record, each with a contract (Table 2.1). The stage is complete when the source inventory is complete and the tool-layer checklist of Chapter 2 passes.

3. **Stage 2: guards in code.** Statement guards, pinned scope, ceilings, timeouts and truncation reporting (Table 3.2). The stage is complete when the claimed-versus-implemented audit of Chapter 3 finds no claim without a code location.

4. **Stage 3: taxonomies and runbook.** Translation table, signature table, dependency split, named semantics, and the runbook encoded in descriptions. The stage is complete when a golden set of past investigations is answered with the expected classifications and verification paths. A workable minimum is a set that covers every business stage and every owner category at least once, labelled by the engineers who resolved the original incidents, run several times per question so that consistency across runs is measured in the pass^k sense of Chapter 3, and passed when the classifications and verification paths match on a share of questions the team fixes in advance. The figures a team chooses (set size, number of runs, pass share) are its own; what the gate requires is that they are written down before the run. Until such a set exists, completion of the stage is a reviewed judgement, recorded as such.

5. **Stage 4: reuse and action.** A second consumer is connected to the same tool layer, and act tools are added behind single-attempt contracts and a human decision point. The stage is complete when the second consumer needs no new interface for its first questions.

7.5.2 What to Build First and what to Defer

Read access with contracts and guards comes first, because every later stage depends on it and because it is valuable on its own: a guarded, documented read layer improves human investigation even before an agent uses it. Taxonomies come before act tools, because an action taken on an unclassified failure repeats the conflations that Chapter 4 removes. Templates are deferred until the runbook has stabilised, because a template frozen around an unstable sequence becomes stale. Act tools are deferred longest, and some teams will rightly never add them: an agent that investigates and recommends, with a human who acts, already removes most of the cross-system work from the engineer's day.

7.5.3 Review Gates

The gates are what distinguish an adoption path from a backlog. Each gate is a review with a fixed question and evidence that answers it: an inventory, a checklist, an audit table, a golden set, a second consumer. A stage whose gate has not been passed is incomplete, however much code it contains. The gates turn the four decisions into checks that a team can pass or fail on its own pipeline, which is the form in which a method can be tested beyond the two systems from which it was drawn.

CONCLUSION

The Method as a Whole

Contract-Bounded Agent Design answers one question: how to build a language-model agent over several operational systems so that it remains reliable as it grows and portable to the next pipeline. The answer is four structural decisions, and their value lies in how they depend on one another.

The tool layer comes first because everything else attaches to it. A narrow tool per external source gives each system of record one entry point, and that entry point is where the guards of the second decision are enforced: a statement guard, a pinned scope, a ceiling and a timeout can only be checked in code that every request passes through. Guards in turn make the third decision trustworthy. A failure taxonomy classifies what the tools return, and a classification computed over a silently truncated result or a stale record is a confident answer to the wrong question; truncation honesty and the liveness check are what let the taxonomies be read at face value. The taxonomies give the answer its meaning: the milestone axis says where the work stopped in terms the business uses, and the owner axis says who fixes it in terms the organisation uses. The fourth decision fixes which failures enter the classification at all, by naming whether the question concerns the present state of the work or its history over a period.

Each decision depends on the others. Without narrow tools, guards have no single place to live. Without guards, the taxonomies classify unreliable evidence. Without the taxonomies, a correct investigation still ends in an answer that names an exception instead of a stage and an owner. Without the named semantics, the same correct investigation can report recovered work as broken. The four decisions are one design seen from four sides.

Around the decisions sit two disciplines and four rules. The runbook orders the investigation so that each step resolves or narrows the question, with a terminal-state gate first and a bounded window of evidence instead of a raw log. The answer discipline restricts what may be said to what the tools returned, makes “no data” a valid answer and requires a verification path. The four named rules (liveness check before trust, best-effort secondary write, partial success by construction and routing by identifier shape) state, with their costs and their limits, defensive structures that the two systems built before anyone named them: one in both systems, three in the system that performs the operations they govern.

What the Convergence Supports

The two systems on which the method rests were built in different domains and team contexts, without shared code, and the convergence of their designs is the book’s principal evidence for the method. Read decision by decision, they show the tool layer, its routing discipline and the answer discipline in the same form; the safety decision explicit in the later system and partial in the earlier one; the two taxonomy axes one in each system; the separation of current state from historical occurrence named in the later system only; and the named rules where the operations they govern exist. Where the domain, the team and the code base changed, the recurring structure persisted.

Open Problems

Five problems remain open, and each marks a direction for further work.

1. **Measuring tool description quality before it degrades investigations.** Tool description debt accumulates silently because no signal fires when a description drifts. Selection tests catch regressions in the questions they contain; a measure of description quality that could be computed on every change, independent of a test set, does not yet exist. Recent work that measures how single information fields in tool documentation shift agents’ success rates points toward one, at benchmark scale (Lu et al., 2026).

2. **Evaluation harnesses for the reasoning step.** Tools, guards and tables can be unit-tested; the model’s choice of the next step cannot. A golden set of investigations with expected tool sequences, classifications and verification paths, scored for reliability across repeated runs, is the shape of the missing harness. Building and maintaining such sets for operational domains, where the correct answer changes as the pipeline changes, is an engineering problem in its own right.

3. **Controlled comparison under matched load.** A comparison of contract-bounded agents with unconstrained agents on the same pipelines, the same questions and the same load, measuring answer correctness, investigation length and operational cost, would test the method’s effect directly.

4. **Agents that act, and agents that share a tool layer.** The method keeps act operations single-attempt and behind a human decision. Extending it to agents permitted to remediate on their own, or to several agents sharing one tool layer and one set of guards, raises questions the two systems did not face: how act tools are rate-limited across agents, how one agent’s action becomes another agent’s stale record, and how the liveness check generalises when the actor and the checker run concurrently.

5. **The description channel as an attack surface.** Tool descriptions are read by the model as instructions, and tool poisoning places hostile instructions there. Within one team’s tool layer, review and selection tests contain the risk. When agents load tools from third-party servers, the description becomes untrusted input on the same footing as a retrieved document, and the guards of Chapter 3 hold only if they are enforced in code that the third party does not control.

Boundaries of the Claims

The evidence for the method is structural corroboration from two systems designed in the author's own practice. It is not the result of a controlled experiment, its four decisions were abstracted after both systems existed, it does not separate the method from design experience carried over from the first system to the second, and the book presents the method with constructed illustrative values and with both systems described at the level of the problem they address. Its claims are therefore claims about design: that the decisions recur under changed constraints in the graded forms of Chapter 7, that the rules hold where their conditions hold, and that the procedures can be followed by an engineer on another pipeline. A third system, chosen in advance and examined against the four decisions, and the controlled comparison named above are the tests that would carry the claims further.

Guidance for Applying the Method

For an engineer applying the method to a pipeline, the order is fixed by the dependencies between the decisions. Inventory the systems of record and give each one a narrow, read-only tool with a written contract. Put every boundary into code before the first consumer depends on it, and audit every claimed resilience feature against the line of code that implements it. Name the business stages and the owners, build the translation and signature tables, separate dependency failures from genuine ones, and decide in writing what "what failed?" means. Encode the runbook in the tool descriptions, hold every answer to the discipline of Chapter 5, and check the design against the four rule cards.

An agent built this way can still be wrong about what to ask next. It is designed so that it cannot fetch without bound, write where it may only read or reach an environment outside its scope, because those boundaries are enforced in tool code; so that a capped result always carries its true total, because the tools return it; and so that its answers report truncation, name the semantics they used and speak in the vocabulary of stages and owners, which the answer discipline and the system prompt require and review checks. Those properties bear on whether an agent can be placed in front of operational systems, and they belong to the design, which a team can write down, review and repeat.

Declarations

Glossary

Answer discipline. The rules that govern what an investigation's answer may contain: every fact traced to a tool result, no inferred fields, absence of data stated as an answer, named semantics, reported truncation, and a verification path (Chapter 5).

Best-effort secondary write. The named rule that, when an action writes an authoritative durable record and a write-back to a live system, the authoritative write decides the outcome and the write-back's failure is logged without failing the action (Chapter 6).

Bounded evidence window. A bounded extract of a log around each line that matches a failure signature, merged and capped in code, handed to the model in place of the raw log (Chapter 5).

Claimed-versus-implemented audit. A review procedure that locates, for every resilience feature a project documents, the code or configuration that implements it on every call path, and reports claims without a location as gaps (Chapter 3).

Composite template. A recurring multi-tool sequence captured as one named tool with its own contract, reducing the planning the model has to do (Chapter 2).

Contract-Bounded Agent Design (CBAD). The method of this book: four structural decisions (narrow tools per source, safety contracts in code, failure taxonomy by meaning, named separation of current state and historical occurrence) with a runbook, an answer discipline and four named rules.

Current-state semantics. The reading of a failure question that takes the latest run of each unit of work and reports the units whose latest run failed; recovered units are excluded (Chapter 4).

Dependency failure. A step that did not run because a step it depends on failed; distinguished from a genuine failure, in which the step ran and failed (Chapter 4).

Historical-occurrence semantics. The reading of a failure question that reports every failed run in a period, including runs whose unit later recovered (Chapter 4).

k-ary cross-system invariant. A business condition whose evaluation needs projections from at least k systems, so that no observer confined to one system can decide it when k is two or more (Chapter 1).

Liveness check before trust. The named rule that, before acting on a record obtained from a secondary store, code confirms against the system of record that the entity is still live and does not act if it is not (Chapters 3 and 6).

Milestone taxonomy. Classification of failures by the business stage at which work stopped, implemented as a maintained translation table from technical signals to stages, with an explicit unknown bucket (Chapter 4).

Owner-concern taxonomy. Classification of failures by the concern that owns the fix, implemented as an ordered signature table evaluated most specific first, with the first match winning and suggestions capped (Chapter 4).

Partial success by construction. The named rule that a bulk operation processes items independently, catches failures per item and returns per-item outcomes with totals; the batch is reported as failed only when every item failed (Chapter 6).

Payload ceiling. A limit, set in a tool's code, on how much the tool reads from an external system: bytes scanned, rows returned, files or characters read, log lines extracted, items listed (Chapter 3).

Routing by identifier shape. The named rule that, when incompatible backends coexist and their identifiers have disjoint forms, each request is dispatched by the form of its identifier to the backend's own sequence (Chapter 6).

Rule card. The fixed format in which a named design rule is stated: name, statement, rationale, failure mode, cost, conditions of application, when not to apply, evidence (Chapter 6).

Scope guard. A guard that fixes in code which environment, project or tenant a tool may address, so that no runtime input can redirect a request (Chapter 3).

Statement guard. A guard that inspects a request before it leaves a tool and refuses anything outside a permitted class, such as non-read statements to a warehouse (Chapter 3).

System of record. The system that holds the authoritative version of a fact, as opposed to systems that hold derived or cached copies of it (Chapter 1).

Terminal-state gate. The first check of the runbook: a run that succeeded or is still running ends the investigation without further calls (Chapter 5).

Tool contract. The fields through which a tool presents itself to the model and to reviewers: name, purpose, applicability, authority, input schema, result shape, side effects, idempotency and failure contract (Chapter 2).

Tool description debt. The gap that opens when a tool's natural-language description diverges from its behaviour, in four forms: vague, overlapping, stale and over-specified (Chapter 2).

Truncation honesty. The rule that a capped result always carries a truncation flag and the true total, and that the answer states the truncation (Chapter 3).

Unknown bucket. The explicit class that receives every failure signal not covered by a taxonomy's tables, so that unrecognised failures are counted and reviewed instead of dropped (Chapter 4).

REFERENCES

1. Aghajani, E., Nagy, C., Vega-Marquez, O. L., Linares-Vasquez, M., Moreno, L., Bavota, G., & Lanza, M. (2019). Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 1199–1210). https://doi.org/10.1109/icse.2019.00122
2. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., & Jacobsohn, N. (1977). *A pattern language: Towns, buildings, construction*. Oxford University Press.
3. Aljohani, A., & Do, H. (2025). PromptDebt: A comprehensive study of technical debt across LLM projects. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering* (pp. 371–382). https://doi.org/10.1145/3756681.3756976
4. Amazon Web Services. (n.d.). *Timeouts, retries, and backoff with jitter* [Amazon Builders' Library]. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/
5. Anthropic. (2024). *Building effective AI agents*. https://www.anthropic.com/engineering/building-effective-agents
6. Anthropic. (2025a). *Effective context engineering for AI agents*. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
7. Anthropic. (2025b). *Writing effective tools for AI agents: Using AI agents*. https://www.anthropic.com/engineering/writing-tools-for-agents
8. Anthropic. (n.d.). *Define tools* [Claude Developer Platform documentation]. https://platform.claude.com/docs/en/agents-and-tools/tool-use/define-tools
9. Apache Software Foundation. (n.d.). *Tasks* [Apache Airflow 3.3.2 documentation]. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html
10. Beurer-Kellner, L., Buesser, B., Creţu, A.-M., Debenedetti, E., Dobos, D., Fabian, D., Fischer, M., Froelicher, D., Grosse, K., Naeff, D., Ozoani, E., Paverd, A., Tramèr, F., & Volhejn, V. (2025). *Design patterns for securing LLM agents against prompt injections* (arXiv:2506.08837v3). arXiv. https://doi.org/10.48550/arXiv.2506.08837
11. Breck, E., Cai, S., Nielsen, E., Salib, M., & Sculley, D. (2017). The ML test score: A rubric for ML production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)* (pp. 1123–1132). https://doi.org/10.1109/bigdata.2017.8258038
12. Chen, Y., Xie, H., Ma, M., Kang, Y., Gao, X., Shi, L., Cao, Y., Gao, X., Fan, H., Wen, M., Zeng, J., Ghosh, S., Zhang, X., Zhang, C., Lin, Q., Rajmohan, S., Zhang, D., & Xu, T. (2024). Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (pp. 674–688). https://doi.org/10.1145/3627703.3629553
13. Chen, Y., Shetty, M., Somashekar, G., Ma, M., Simmhan, Y., Mace, J., Bansal, C., Wang, R., & Rajmohan, S. (2025). *AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds* (arXiv:2501.06706v1). arXiv. https://doi.org/10.48550/arXiv.2501.06706
14. Chen, P. B., Yang, D., Li, W., Wenz, F., Zhang, Y., Tatbul, N., Cafarella, M., Demiralp, Ç., & Stonebraker, M. (2026). *BEAVER: An enterprise benchmark for text-*

to-SQL (arXiv:2409.02038v3). arXiv. https://doi.org/10.48550/arXiv.2409.02038

15. Cockburn, A. (2005). *Hexagonal architecture*. https://alistair.cockburn.us/hexagonal-architecture/
16. Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715
17. Dabas, M., Jeong, J., Jin, M., & Jia, R. (2026). *Memory-induced tool-drift in LLM agents* (arXiv:2605.24941v1). arXiv. https://doi.org/10.48550/arXiv.2605.24941
18. Davis, K., Peabody, B., & Leach, P. (2024). *Universally unique IDentifiers (UUIDs)* (RFC 9562). RFC Editor. https://doi.org/10.17487/RFC9562
19. Debenedetti, E., Zhang, J., Balunovic, M., Beurer-Kellner, L., Fischer, M., & Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems 37* (pp. 82895–82920). https://doi.org/10.52202/079017-2636
20. Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., & Tramèr, F. (2025). *Defeating prompt injections by design* (arXiv:2503.18813v2). arXiv. https://doi.org/10.48550/arXiv.2503.18813
21. DORA. (2024). *Accelerate state of DevOps report 2024*. Google Cloud. https://dora.dev/research/2024/dora-report/
22. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering* (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12
23. Ewaschuk, R. (2016). Monitoring distributed systems. In B. Beyer, C. Jones, J. Petoff, & N. R. Murphy (Eds.), *Site reliability engineering: How Google runs production systems*. O'Reilly Media. https://sre.google/sre-book/monitoring-distributed-systems/
24. Foidl, H., Golendukhina, V., Ramler, R., & Felderer, M. (2024). Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers. *Journal of Systems and Software*, *207*, 111855. https://doi.org/10.1016/j.jss.2023.111855
25. Fowler, M. (2014). *CircuitBreaker*. https://martinfowler.com/bliki/CircuitBreaker.html
26. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
27. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024). *Retrieval-augmented generation for large language models: A survey* (arXiv:2312.10997). arXiv. https://doi.org/10.48550/arXiv.2312.10997
28. Garcia-Molina, H., & Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87* (pp. 249–259). https://doi.org/10.1145/38713.38742
29. Gartner. (2025, June 25). *Gartner predicts over 40% of agentic AI projects will be canceled by end of 2027* [Press release]. https://www.gartner.com/en/newsroom/press-releases/2025-06-25-gartner-predicts-over-40-percent-of-agentic-ai-projects-will-be-canceled-by-end-of-2027
30. Gawande, A. (2010). *The checklist manifesto: How to get things right*. Metropolitan Books.
31. Ghosh, S., Shetty, M., Bansal, C., & Nath, S. (2022). How to fight production incidents? An empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing* (pp. 126–141). https://doi.org/10.1145/3542929.3563482
32. Google. (2019a). *AIP-158: Pagination* [API Improvement Proposals]. https://google.aip.dev/158
33. Google. (2019b). *AIP-193: Errors* [API Improvement Proposals]. https://google.aip.dev/193
34. Google Cloud. (n.d.). *Estimate and control costs* [BigQuery documentation]. https://docs.cloud.google.com/bigquery/docs/best-practices-costs
35. Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security* (pp. 79–90). https://doi.org/10.1145/3605764.3623985
36. Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., & Eliazar, K. J. (2016). Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (pp. 1–16). https://doi.org/10.1145/2987550.2987583
37. Guo, R., Dong, K., Gao, X., & Das, K. (2026). *Learning to rewrite tool descriptions for reliable LLM-agent tool use* (arXiv:2602.20426v2). arXiv. https://doi.org/10.48550/arXiv.2602.20426
38. Hasan, M. M., Li, H., Fallahzadeh, E., Rajbahadur, G. K., Adams, B., & Hassan, A. E. (2026). Model Context Protocol (MCP) at first glance: Studying the security and maintainability of MCP servers. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3814959

39. He, P., Zhu, J., Zheng, Z., & Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)* (pp. 33–40). https://doi.org/10.1109/icws.2017.13

40. He, S., He, P., Chen, Z., Yang, T., Su, Y., & Lyu, M. R. (2021). A survey on automated log analysis for reliability engineering. *ACM Computing Surveys*, *54*(6), 1–37. https://doi.org/10.1145/3460345

41. Helland, P. (2012). Idempotence is not a medical condition. *Communications of the ACM*, *55*(5), 56–65. https://doi.org/10.1145/2160718.2160734

42. Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–106. https://doi.org/10.2307/25148625

43. Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

44. Hou, X., Zhao, Y., Wang, S., & Wang, H. (2026). Model Context Protocol (MCP): Landscape, security threats, and future research directions. *ACM Transactions on Software Engineering and Methodology*, *35*(10), 1–37. https://doi.org/10.1145/3796519

45. Hsieh, C.-Y., Chen, S.-A., Li, C.-L., Fujii, Y., Ratner, A., Lee, C.-Y., Krishna, R., & Pfister, T. (2023). *Tool documentation enables zero-shot tool-usage with large language models* (arXiv:2308.00675v1). arXiv. https://doi.org/10.48550/arXiv.2308.00675

46. Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., & Yao, R. (2017). Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 150–155). https://doi.org/10.1145/3102980.3103005

47. Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., & Liu, T. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, *43*(2), 1–55. https://doi.org/10.1145/3703155

48. Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)* (pp. 109–120). https://doi.org/10.1109/wicsa.2005.61

49. Jones, C. (2016). Effective troubleshooting. In B. Beyer, C. Jones, J. Petoff, & N. R. Murphy (Eds.), *Site reliability engineering: How Google runs production systems*. O'Reilly Media. https://sre.google/sre-book/effective-troubleshooting/

50. Kalai, A. T., Nachum, O., Vempala, S. S., & Zhang, E. (2025). *Why language models hallucinate* (arXiv:2509.04664v1). arXiv. https://doi.org/10.48550/arXiv.2509.04664

51. Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.

52. Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, *29*(6), 18–21. https://doi.org/10.1109/ms.2012.167

53. Kubernetes Authors. (n.d.). *Liveness, readiness, and startup probes* [Kubernetes documentation]. https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/

54. LangChain. (2024). *LangChain state of AI agents report: 2024 trends*. https://www.langchain.com/stateofaiagents

55. Lei, F., Chen, J., Ye, Y., Cao, R., Shin, D., Su, H., Suo, Z., Gao, H., Hu, W., Yin, P., Zhong, V., Xiong, C., Sun, R., Liu, Q., Wang, S., & Yu, T. (2025). *Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows* (arXiv:2411.07763v2). arXiv. https://doi.org/10.48550/arXiv.2411.07763

56. Levin, R., Cohen, E., Corwin, W., Pollack, F., & Wulf, W. (1975). Policy/mechanism separation in Hydra. *ACM SIGOPS Operating Systems Review*, *9*(5), 132–140. https://doi.org/10.1145/1067629.806531

57. Levy, M., Jacoby, A., & Goldberg, Y. (2024). Same task, more tokens: The impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 15339–15353). https://doi.org/10.18653/v1/2024.acl-long.818

58. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2021). *Retrieval-augmented generation for knowledge-intensive NLP tasks* (arXiv:2005.11401v4). arXiv. https://doi.org/10.48550/arXiv.2005.11401

59. Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Cao, R., Geng, R., Huo, N., Zhou, X., Ma, C., Li, G., Chang, K. C. C., Huang, F., Cheng, R., & Li, Y. (2023). *Can LLM already serve as a database interface? A BIg bench for large-scale database grounded text-to-SQLs* (arXiv:2305.03111v3). arXiv. https://doi.org/10.48550/arXiv.2305.03111

60. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, *12*, 157–173. https://doi.org/10.1162/tacl_a_00638

61. Lu, Y., Zhang, K., Chen, B., & Peng, X. (2026). *DOCSCHISEL: Adaptive tool documentation optimization framework*

for LLM agents (arXiv:2608.10037v1). arXiv. https://doi.org/10.48550/arXiv.2608.10037

62. Masterman, T., Besen, S., Sawtell, M., & Chao, A. (2024). *The landscape of emerging AI agent architectures for reasoning, planning, and tool calling: A survey* (arXiv:2404.11584). arXiv. https://doi.org/10.48550/arXiv.2404.11584

63. Meszaros, G., & Doble, J. (1997). *A pattern language for pattern writing*. Addison-Wesley. https://hillside.net/index.php/a-pattern-language-for-pattern-writing

64. MITRE. (n.d.). *CWE-770: Allocation of resources without limits or throttling* [Common Weakness Enumeration]. https://cwe.mitre.org/data/definitions/770.html

65. Model Context Protocol. (2025a). *Key changes* [Specification revision 2025-03-26]. https://modelcontextprotocol.io/specification/2025-03-26/changelog

66. Model Context Protocol. (2025b). *Tools* [Specification revision 2025-06-18]. https://modelcontextprotocol.io/specification/2025-06-18/server/tools

67. Notaro, P., Cardoso, J., & Gerndt, M. (2021). A survey of AIOps methods for failure management. *ACM Transactions on Intelligent Systems and Technology*, *12*(6), 1–45. https://doi.org/10.1145/3483424

68. Nottingham, M., Wilde, E., & Dalal, S. (2023). *Problem details for HTTP APIs* (RFC 9457). RFC Editor. https://doi.org/10.17487/RFC9457

69. Nygard, M. (2011). *Documenting architecture decisions*. Cognitect. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

70. Nygard, M. T. (2018). *Release it! Design and deploy production-ready software* (2nd ed.). Pragmatic Bookshelf.

71. OpenTelemetry Authors. (n.d.). *Signals* [OpenTelemetry documentation]. https://opentelemetry.io/docs/concepts/signals/

72. OWASP Foundation. (2025a). *LLM06:2025 Excessive agency* [OWASP Top 10 for LLM Applications 2025]. https://genai.owasp.org/llmrisk/llm062025-excessive-agency/

73. OWASP Foundation. (2025b). *LLM10:2025 Unbounded consumption* [OWASP Top 10 for LLM Applications 2025]. https://genai.owasp.org/llmrisk/llm102025-unbounded-consumption/

74. Pan, R., Stennett, T., Sankar, D., McGinn, B., Orso, A., Pavuluri, R., Sinha, S., & Vukovic, M. (2026). *Tangent: An empirical study of testing practices for LLM-based agent applications* (arXiv:2608.08413v1). arXiv. https://doi.org/10.48550/arXiv.2608.08413

75. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

76. Parnin, C., Soares, G., Pandita, R., Gulwani, S., Rich, J., & Henley, A. Z. (2025). Building your own product copilot: Challenges, opportunities, and needs. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 338–348). https://doi.org/10.1109/saner64311.2025.00039

77. Patil, S., Zhang, T., Wang, X., & Gonzalez, J. (2024). Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems 37* (pp. 126544–126565). https://doi.org/10.52202/079017-4020

78. Pedro, R., Castro, D., Carreira, P., & Santos, N. (2025). *From prompt injections to SQL injection attacks: How protected is your LLM-integrated web application* (arXiv:2308.01990v4). arXiv. https://doi.org/10.48550/arXiv.2308.01990

79. Pei, C., Wang, Z., Liu, F., Li, Z., Liu, Y., He, X., Kang, R., Zhang, T., Chen, J., Li, J., Xie, G., & Pei, D. (2025). Flow-of-action: SOP enhanced LLM-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025* (pp. 422–431). https://doi.org/10.1145/3701716.3715225

80. Puchakayala, S. S. (2026a). Principles of designing event-driven systems for real-time stream data processing. *Universal Library of Engineering Technology*, *3*(1), 65–69. https://doi.org/10.70315/uloap.ulete.2026.0301011

81. Puchakayala, S. S. (2026b). Review of modern strategies for migrating monolithic enterprise applications to microservices architecture: Challenges, patterns, and best practices. *International Journal of Advanced Engineering, Management and Science*, *12*(3), 53–60. https://doi.org/10.22161/ijaems.123.5

82. Puchakayala, S. S. (2026c). Multi-layer AI architecture for real-time detection of workflow and data integrity failures in distributed retail platforms. *International Journal of Open Information Technologies*.

83. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., & Sun, M. (2023). *ToolLLM: Facilitating large language models to master 16000+ real-world APIs* (arXiv:2307.16789v2). arXiv. https://doi.org/10.48550/arXiv.2307.16789

84. Qu, C., Dai, S., Wei, X., Cai, H., Wang, S., Yin, D., Xu, J., & Wen, J.-r. (2025). Tool learning with large language models: A survey. *Frontiers of Computer Science*, *19*(8). https://doi.org/10.1007/s11704-024-40678-2

85. Reason, J. (2000). Human error: Models and

management. *BMJ*, *320*(7237), 768–770. https://doi.org/10.1136/bmj.320.7237.768

86. Riddell, E., Riddell, J., Sun, G., Antkiewicz, M., & Czarnecki, K. (2026). Stalled, biased, and confused: Uncovering reasoning failures in LLMs for cloud-based root cause analysis. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering* (pp. 172–183). https://doi.org/10.1145/3793655.3793732

87. Roy, D., Zhang, X., Bhave, R., Bansal, C., Las-Casas, P., Fonseca, R., & Rajmohan, S. (2024). Exploring LLM-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (pp. 208–219). https://doi.org/10.1145/3663529.3663841

88. Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164. https://doi.org/10.1007/s10664-008-9102-8

89. Saha, A., & Hoi, S. C. H. (2022). Mining root cause knowledge from cloud service incident investigations for AIOps. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (pp. 197–206). https://doi.org/10.1145/3510457.3513030

90. Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, *63*(9), 1278–1308. https://doi.org/10.1109/proc.1975.9939

91. Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, *2*(4), 277–288. https://doi.org/10.1145/357401.357402

92. Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36* (pp. 68539–68551). https://doi.org/10.52202/075280-2997

93. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28*. https://papers.nips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

94. Shankar, S., Zamfirescu-Pereira, J. D., Hartmann, B., Parameswaran, A., & Arawjo, I. (2024). Who validates the validators? Aligning LLM-assisted evaluation of LLM outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (pp. 1–14). https://doi.org/10.1145/3654777.3676450

95. Shetty, M., Bansal, C., Kumar, S., Rao, N., Nagappan, N., & Zimmermann, T. (2021). Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 218–227). https://doi.org/10.1109/icse-seip52600.2021.00031

96. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a large-scale distributed systems tracing infrastructure* (Technical Report dapper-2010-1). Google. https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf

97. Spadini, D., Aniche, M., Bruntink, M., & Bacchelli, A. (2017). To mock or not to mock? An empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (pp. 402–412). https://doi.org/10.1109/msr.2017.61

98. Stack Overflow. (2025). *2025 Stack Overflow developer survey: AI*. https://survey.stackoverflow.co/2025/ai

99. Sumers, T. R., Yao, S., Narasimhan, K., & Griffiths, T. L. (2024). *Cognitive architectures for language agents* (arXiv:2309.02427). arXiv. https://doi.org/10.48550/arXiv.2309.02427

100. Tafreshipour, M., Imani, A., Huang, E., Almeida, E. S. d., Zimmermann, T., & Ahmed, I. (2025). Prompting in the wild: An empirical study of prompt evolution in software repositories. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)* (pp. 686–698). https://doi.org/10.1109/msr66628.2025.00106

101. Trivedi, H., Balasubramanian, N., Khot, T., & Sabharwal, A. (2023). Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 10014–10037). https://doi.org/10.18653/v1/2023.acl-long.557

102. Uddin, G., & Robillard, M. P. (2015). How API documentation fails. *IEEE Software*, *32*(4), 68–75. https://doi.org/10.1109/ms.2014.80

103. W3C. (2021). *Trace context* (W3C Recommendation, 23 November 2021). https://www.w3.org/TR/trace-context/

104. Wallace, E., Xiao, K., Leike, R., Weng, L., Heidecke, J., & Beutel, A. (2024). *The instruction hierarchy: Training LLMs to prioritize privileged instructions* (arXiv:2404.13208v1). arXiv. https://doi.org/10.48550/arXiv.2404.13208

105. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., & Wen, J. (2024a). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, *18*(6). https://doi.org/10.1007/s11704-024-40231-1

106. Wang, Z., Liu, Z., Zhang, Y., Zhong, A., Wang, J., Yin, F., Fan, L., Wu, L., & Wen, Q. (2024b). RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (pp. 4966–4974). https://doi.org/10.1145/3627673.3680016

107. Wen, F., Nagy, C., Bavota, G., & Lanza, M. (2019). A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (pp. 53–64). https://doi.org/10.1109/icpc.2019.00019

108. Wen, B., Yao, J., Feng, S., Xu, C., Tsvetkov, Y., Howe, B., & Wang, L. L. (2025). Know your limits: A survey of abstention in large language models. *Transactions of the Association for Computational Linguistics*, *13*, 529–556. https://doi.org/10.1162/tacl_a_00754

109. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., Zheng, R., Fan, X., Wang, X., Xiong, L., Zhou, Y., Wang, W., Jiang, C., Zou, Y., Liu, X., . . . Gui, T. (2025). The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, *68*(2). https://doi.org/10.1007/s11432-024-4222-0

110. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing reasoning and acting in language models* (arXiv:2210.03629v3). arXiv. https://doi.org/10.48550/arXiv.2210.03629

111. Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2024). *τ-bench: A benchmark for tool-agent-user interaction in real-world domains* (arXiv:2406.12045). arXiv. https://doi.org/10.48550/arXiv.2406.12045

112. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (pp. 3911–3921). https://doi.org/10.18653/v1/d18-1425

113. Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., & Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

114. Yuan, S., Song, K., Chen, J., Tan, X., Shen, Y., Ren, K., Li, D., & Yang, D. (2025). EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)* (pp. 951–972). https://doi.org/10.18653/v1/2025.naacl-long.44

115. Zhan, Q., Liang, Z., Ying, Z., & Kang, D. (2024). InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics ACL 2024* (pp. 10471–10506). https://doi.org/10.18653/v1/2024.findings-acl.624

116. Zhan, Q., Fang, R., Panchal, H. S., & Kang, D. (2025). Adaptive attacks break defenses against indirect prompt injection attacks on LLM agents. In *Findings of the Association for Computational Linguistics: NAACL 2025* (pp. 7101–7117). https://doi.org/10.18653/v1/2025.findings-naacl.395

117. Zhang, L., Jia, T., Jia, M., Wu, Y., Liu, A., Yang, Y., Wu, Z., Hu, X., Yu, P., & Li, Y. (2025). A survey of AIOps in the era of large language models. *ACM Computing Surveys*, *58*(2), 1–35. https://doi.org/10.1145/3746635

118. Zhao, N., Chen, J., Peng, X., Wang, H., Wu, X., Zhang, Y., Chen, Z., Zheng, X., Nie, X., Wang, G., Wu, Y., Zhou, F., Zhang, W., Sui, K., & Pei, D. (2020). Understanding and handling alert storm for online service systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (pp. 162–171). https://doi.org/10.1145/3377813.3381363

119. Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., & Lyu, M. R. (2019). Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 121–130). https://doi.org/10.1109/icse-seip.2019.00021

Copyright: © 2026 The Author(s). This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.