Software Factories

Software Factories

From the AI SDLC to Secure, Autonomous Software Delivery

Abdul Kavungal (AK)



AWARD WINNING CREATIONS

Software Factories
From the AI SDLC to Secure, Autonomous Software Delivery

Copyright © 2026 Abdul Jaleel Kavungal. All rights reserved.

Published by Award Winning Creations.

No part of this book may be reproduced or transmitted in any form without written permission from the publisher, except for brief quotations in reviews and the templates in Appendix A, which readers may adapt for their own use.

This book quotes and adapts material from *The Secure Software Factory: A Reference Architecture* (Agent Baseline, version 1, 13 October 2026), whose prose and figures are licensed under the Creative Commons Attribution 4.0 International licence (CC BY 4.0). That material is attributed in the text and in the Notes and References. Its inclusion does not imply endorsement by its authors.

Harbour Ledger, used throughout as a running example, is a composite, illustrative organisation. It does not exist, and its configurations, metrics and outcomes are illustrative.

Product and company names mentioned are trademarks of their respective owners. The information in this book is provided without warranty. Security and compliance decisions should be made with appropriate professional and legal advice.

Typography by TyAK: Keyslate Display and TyCode, under the SIL Open Font License 1.1. Body text set in Source Serif Pro.

Contents

Preface vii

PART I. FOUNDATIONS OF THE AI SDLC 1

1. The Line That Writes Code 3
2. Factories Before Agents 13
3. The AI Software Development Lifecycle 23
4. Specifications Become the Source 33
5. Agents, Harnesses and Context 45

PART II. ANATOMY OF A SOFTWARE FACTORY 57

6. Two Loops 59
7. The Agent Is Untrusted 69
8. Four Boundaries 79
9. Supply Chain, Provenance and Evidence 93
10. Identity, Authorisation and Approval 105

PART III. BUILDING THE FACTORY 117

11. Three Levels and a Roadmap 119
12. Building Level 1: The Hardened Developer Factory 129
13. Building Level 2: The Factory as a Shared Service 141
14. Quality Gates, Tests and Evals 153
15. Level 3: Orchestrators, Signals and Self-Healing 163

PART IV. OPERATING AND GOVERNING THE FACTORY 173

16. Measuring the Factory 175
17. Governance, Compliance and Accountability 185
18. People and Platform 195

PART V. THE FUTURE 203

19. Toward the Dark Factory 205

Appendix A. Templates 213
Appendix B. A Ninety-Day Starting Plan 219
Glossary 223
Notes and References 229
Index 235

Preface

This book began with a question that kept coming up in conversations with engineering teams: what happens when the assistant stops waiting for you?

For a few years the answer did not matter much. An AI coding tool suggested a line, a function, sometimes a file, and a developer accepted or rejected it. Responsibility was easy to locate because a person sat between every suggestion and the codebase. That arrangement is ending. Agents now take a ticket, set up an environment, write code, run the tests, read the failures, try again, and open a pull request while the person who asked for the work is in a meeting or asleep. Some organisations have gone further and let agents pick up production alerts, propose fixes and route them through deployment.

When software starts producing software with that little supervision, you are no longer using a tool. You are running a production system whose output is code. Production systems need design, boundaries, measurement, maintenance and someone who is accountable for them. The industry has a word for a production system that turns specifications into software, and it is an old word with a complicated history: the software factory.

The word deserves some caution. Earlier software factories, from the Japanese factories of the 1970s and 1980s to the model-driven factories of the early 2000s, promised industrial predictability and delivered something more modest and more interesting. They were best at standardising process, collecting data and making quality visible. They struggled when they treated creative work as if it were assembly. The agentic factory inherits both lessons. It can standardise the path work travels and the evidence it leaves behind. It cannot make judgement unnecessary, and the moment it pretends to, it starts producing confident defects at scale.

What this book covers

The book moves in five parts.

Part I lays foundations. It defines the AI software development lifecycle, traces the history of software factories, explains why specifications are becoming the most important artifact a team writes, and describes the pieces an agent is made of: models, harnesses, tools, skills, context and profiles.

Part II is the anatomy of a factory. It describes the inner loop that builds and verifies a change and the outer loop that reviews, deploys and learns from it. It then takes security seriously, because an agent that can act on real systems has to be treated as untrusted, and explains the four boundaries (sandbox, network, supply chain, and identity and access) that let a factory keep the access that makes it useful without trusting the agent to police itself.

Part III is about building. It works through three levels of autonomy, from a hardened developer workstation to a shared factory service to a factory in which agents start and coordinate work themselves. Each level comes with concrete artifacts: configuration, policy, gate definitions and run records you can adapt.

Part IV covers operating and governing a factory: what to measure, how to keep costs and quality honest, how to map the factory to frameworks your auditors already know, and how to organise people and platform teams around it.

Part V looks forward, to the idea of a "dark factory" that runs without human review, and argues for a careful, evidence-based path toward it rather than a leap.

Where the ideas come from

The book draws heavily on *The Secure Software Factory: A Reference Architecture*, published by the Agent Baseline project in October 2026 under a Creative Commons Attribution licence. That paper gave the field a clear vocabulary: inner and outer loops, profiles, four security boundaries, a runtime authorisation service, and three levels of

autonomy. Where this book uses that vocabulary, it says so, and the Notes and References section at the back lists every source used.

The book also draws on public work on spec-driven development, on the AWS AI-Driven Development Lifecycle, on open governance frameworks for AI delivery, on software supply chain standards such as SLSA and the NIST Secure Software Development Framework, and on the research programme behind the DORA metrics. Figures quoted from those sources are reported as their authors published them. Where the book describes a company called Harbour Ledger, it is describing a composite, illustrative organisation assembled from common patterns. Harbour Ledger does not exist, and no fact about a real company is attributed to it.

How to read this book

If you lead an engineering organisation, read Parts I and IV first, then skim the chapter summaries in Parts II and III. If you are a platform or security engineer, Part II and Part III are the core, and the templates in Appendix A are meant to be copied. If you are a developer who already works with coding agents every day, Chapter 5, Chapter 12 and Chapter 14 will probably be the most immediately useful, because they deal with the parts of the factory you touch.

Every chapter ends with a short summary. Many chapters include checklists and configuration samples. They are written to be read and adapted, not pasted in unchanged. A policy file that made sense for one organisation's risk appetite can be dangerously loose or uselessly strict for another's.

One last note on tone. Writing about AI tends to swing between excitement and dread. This book tries to do neither. Agentic delivery is a real change in how software gets made, and it is also an engineering problem with known techniques for making it safe, measurable and boring. Boring is the goal. A good factory is one where nothing surprising happens, and when something does, you can see exactly why.

PART I

Foundations of the AI SDLC

CHAPTER 1

The Line That Writes Code

At 2:10 on a Tuesday morning, a payments service starts returning errors to a small fraction of checkout requests. Nobody is awake to see the first alert. In one version of this story, an on-call engineer is paged at 2:14, opens a laptop at 2:25, spends forty minutes reading logs, finds a null check that a recent refactor removed, writes a fix, waits for a reviewer, and deploys at 4:05. In another version, an agent receives the alert, is granted read access to the last hour of that service's logs and permission to open exactly one pull request, reproduces the failure in an isolated sandbox, writes a fix and a regression test, and hands a pull request to an approval policy that routes it to the on-call engineer's phone. The engineer reads a short evidence summary, approves it, and the change rolls out to a small share of traffic. The error rate falls. The engineer goes back to sleep.

The second version is not science fiction. The pattern is described, nearly step for step, in the Agent Baseline project's reference architecture for secure software factories, and pieces of it already run in production at companies that have written publicly about their systems. What makes the second version interesting is not that an agent wrote the fix. Agents have been able to write small fixes for some time. What is interesting is everything around the agent: the decision about what it was allowed to read, the sandbox it ran in, the proxy that held the credentials so the agent never saw them, the test that proved the fix, the record of who approved what, and the canary deploy that confirmed the result. That surrounding system is the subject of this book.

From suggestion to delegation

It helps to be precise about what changed, because the change happened in steps that blurred together.

The first generation of AI coding tools completed code. You typed, and the tool predicted what came next. A person stayed in control of every keystroke that reached the repository. The second generation added chat: you could ask for a function or an explanation and paste the answer in. Control was still entirely human, although the volume of code a person could produce, and therefore had to review, went up.

The third step was the agent loop. Instead of answering once, the tool could take actions: read files, run commands, execute tests, observe the results, and decide what to do next. A developer still started the loop and still reviewed the result, but inside the loop the agent chose which commands to run, which packages to install and which systems to call. That is a different kind of tool. It exercises discretion.

The fourth step, which many teams are in the middle of now, takes the person out of the starting position. Work arrives as a ticket, a message in a team channel, a scheduled job or a production signal. A workflow or another agent decides what happens next and starts an agent with a bounded task. A person may still approve the result, but nobody watched the work being done.

The Secure Software Factory paper captures the pace of this shift in a few sentences. It notes that a year before its publication, Geoffrey Huntley's "Ralph" loop stood out for showing that an agent could work unattended overnight, and it reports that Vercel's factory authors 25 to 35 percent of its AI SDK team's merged pull requests, that Uber reports more than 70 percent of its pull requests come from agents, and that Stripe's "minions" go from a Slack message to a pull request that passes CI with no human interaction. Those are figures the paper's authors reported about other organisations, and they should be read that way: as signals of direction rather than benchmarks to match. The direction is clear enough.

What a software factory is

The reference architecture offers a definition that this book adopts:

> *A software factory is an agentic system that turns a prompt, a ticket, or a specification into tested code for an existing codebase, using one agent or several, and increasingly deploys and maintains what it builds.*

Several words in that sentence carry weight.

Agentic means that at least one component exercises discretion. A conventional CI pipeline is automated, but it does not decide how to pursue a goal. An agent does.

A prompt, a ticket, or a specification names the inputs, and their range matters. A one-line prompt and a twenty-page specification are both legitimate inputs, but they produce very different levels of predictability. Much of this book is about moving inputs toward the specification end of that range.

Tested code sets a bar. The output of a factory is not a draft. It is a change that has passed verification, and the factory is responsible for running that verification rather than leaving it to whoever receives the pull request.

For an existing codebase is a scoping decision. The reference architecture deliberately frames factories around a specific codebase and use. A factory that can do anything to anything is not a factory. It is an unbounded risk.

Increasingly deploys and maintains what it builds acknowledges where things are heading. A factory that only writes pull requests is the beginning. A factory that deploys, watches production and starts new work from what it sees is the destination many organisations are working toward.

The definition also implies something it does not say outright: the organisation running the factory remains accountable for what it produces. The paper says so explicitly, and so will this book, repeatedly. Agents do not carry accountability. People and organisations do.

The parts of a factory

A factory is assembled from familiar pieces, arranged in a new way. The reference architecture lists them: agents and their harnesses, models,

skills, tools and Model Context Protocol (MCP) servers, dependencies, and blueprints. Each agent works under a *profile* for the kind of job it does, which brings together the skills, tools, context and sandbox configuration that job needs.

Chapter 5 takes each piece apart. For now, a short tour.

The *model* is the language model doing the reasoning. It is the most visible component and, from a systems point of view, one of the least controllable. You choose it, route requests to it and budget for it, but you do not get to inspect its decisions in advance.

The *harness* is the runtime around the model: the program that gives the model a task, exposes tools to it, runs the commands it asks for, feeds back the results and keeps the loop going. Claude Code, Codex, Cursor's agents and similar tools are harnesses.

Tools are the actions available to the agent: read a file, edit a file, run a shell command, call an API, query a database. *MCP servers* are a standard way of offering tools and data to agents, introduced by Anthropic in November 2024 and later contributed to a Linux Foundation project.

Skills are packaged instructions, sometimes with scripts, that an agent loads to perform a particular kind of task. *Context files* tell an agent how a repository works: how to build it, how to test it, what conventions to follow.

Blueprints are approved architecture patterns, such as service templates, that agents draw on so their output follows the organisation's standards. They are the descendants of the templates and reference architectures that platform teams have published for years.

Dependencies are everything the agent installs or links against. They are where the software supply chain enters the factory, and they are a major source of risk.

The *profile* binds these together for a kind of work. An incident response profile might include a log-reading skill, a read-only observability tool, a particular sandbox image and a rule that the agent may open at most one pull request. A dependency upgrade profile would look quite different.

Loops decide quality; boundaries decide safety

If you remember one idea from this chapter, make it this one. A factory needs two different kinds of control, and they answer different questions.

The first kind is the *loop*. Work in a factory moves through an inner loop, in which an agent assembles what the task needs, builds the change in a sandbox and verifies it against the organisation's standards, repeating until the work passes. It then moves through an outer loop, in which review, evaluation and production feedback flow back into the design of the work and of the factory itself. Loops answer the question: is this software acceptable?

The second kind is the *boundary*. Boundaries limit what an agent can do while it works: where its code runs, which systems it can reach, which components it can use, and what it is authorised to act on. Boundaries answer a different question: is this agent staying inside what it is allowed to do?

The reference architecture puts the relationship sharply. The loops decide whether the software is acceptable, and the boundaries decide what agents may do. Neither relies on the judgement of the agent doing the work, and a factory needs both, because an agent can stay within its scope and still ship a defect, and code can pass every test after its agent reached something it should not have.

That last sentence is worth reading twice. It describes two failure modes that teams routinely confuse. A team with excellent tests and no boundaries can ship correct code from an agent that, along the way, read production credentials it found in a configuration file. A team with excellent boundaries and weak tests can contain every agent perfectly and still merge a subtle bug. The factory has to handle both.

Why a factory, not just better tools

A reasonable objection is that all of this sounds like a lot of infrastructure for what is, in the end, a code generator. Why not give developers good agents and let them get on with it?

For some teams, at some stages, that is the right answer, and Chapter 11 describes a Level 1 factory that is little more than a well-configured developer workstation. But three forces push organisations toward something more structured.

The first is *volume*. When agents produce a meaningful share of changes, the review burden moves. A developer who used to write three pull requests a day and review four might now be asked to review fifteen. Review becomes the bottleneck, and review under pressure becomes shallow. The factory's job is to make review cheaper and more reliable by putting better evidence in front of the reviewer and by catching what machines can catch before a person has to look.

The second is *discretion*. Automation has always made decisions from events and policies. The reference architecture draws the distinction cleanly: agents exercise discretion, interpreting goals and choosing how to get the work done. A build script never decided, on its own initiative, to go looking for a better credential. An agent might. The PocketOS incident described in Chapter 7 is exactly that: an agent on a routine staging task hit a credential mismatch, searched the codebase, found an API token with far broader access than its purpose implied, and used it in a single call that deleted a production database volume and the backups stored with it. Discretion needs a different kind of control from automation.

The third is *evidence*. As more software is written by agents, the question "why is this code the way it is?" gets harder to answer from memory. Mitchell Hashimoto's "whiteboard defense", quoted in the reference architecture, asks that any engineer be able to explain and defend any customer-facing system they shipped. A factory cannot do the explaining for them, but it can give them the facts: which agent made each change, on whose behalf, under which profile and permissions, using which building blocks, and which checks the change passed. That record is a product of the factory's structure. A loose collection of tools does not produce it.

Three common misreadings

The word *factory* invites misreadings, and it is worth dealing with three of them early.

Misreading one: a factory removes people. It moves them. In a mature factory, people spend less time typing implementation code and more time writing specifications, defining acceptance criteria, curating skills and evaluations, reviewing evidence, governing policy and handling the cases the factory escalates. The reference architecture says the same of its most autonomous level: as more work qualifies for automatic approval, people spend more of their time defining outcomes, improving skills and evals, and governing the factory. Teams that adopt agents in the hope of shrinking engineering headcount tend to discover that the remaining work is harder and more consequential, not easier.

Misreading two: a factory is a single product you buy. Parts of it can be bought: sandboxes, egress proxies, supply chain firewalls, agent harnesses, internal developer portals. But the factory is the arrangement of those parts around your codebase, your policies and your risk appetite. It is closer to a production line you design than to a machine you install. Two organisations using the same vendor tools can run very different factories.

Misreading three: the goal is full autonomy. The goal is the right amount of autonomy for each class of work, earned with evidence and adjustable when evidence changes. Some work should stay firmly in the hands of a developer driving an agent for the foreseeable future. Other work, such as routine dependency updates in well-tested services, may reasonably be approved automatically once the factory has shown it handles that work reliably. A small team may run a highly autonomous factory while a large enterprise keeps much of its work at the most supervised level. That is not a failure of ambition. It is calibration.

The organising principle

Every chapter that follows returns to a principle stated in the reference architecture's central diagram: *the agent proposes, policy authorizes, and infrastructure enforces.*

The agent is good at proposing. It can read a ticket, explore a codebase, form a plan and try things. It should have freedom to do that, because an agent that has to ask permission for every command is not useful.

Policy is good at authorising. It is written in advance, reviewed by people, versioned and testable. It decides what a given task justifies, based on context the factory trusts: who asked, what kind of work it is, what the agent's profile allows.

Infrastructure is good at enforcing. A sandbox does not get talked out of its isolation. An egress proxy does not decide to make an exception because the request sounds reasonable. A signature check does not get tired.

When those three roles stay separate, an organisation can give agents real authority without betting on their judgement. When they blur, for example when an agent decides its own permissions, or when a policy lives only in an agent's instructions, the factory is only as safe as the agent's last decision.

A composite example: Harbour Ledger

Throughout this book we will follow a composite, illustrative company called Harbour Ledger. It is not a real organisation. It is assembled from patterns that are common in mid-sized software companies, so that the book can show decisions in context without attributing them to anyone.

Harbour Ledger runs a payments platform: roughly forty services, a web dashboard, two mobile apps and a public API. It has about ninety engineers in twelve product teams, a platform team of eight and a security team of five. Its developers already use coding agents individually, with mixed results. Some teams love them. Others have stopped using them after a few painful incidents with large, confident pull requests that broke things in ways the tests did not catch.

Harbour Ledger's engineering leadership wants to know three things. Can agents take on a meaningful share of routine work, such as dependency upgrades, flaky test fixes, small bug fixes and documentation, without increasing incidents? Can that be done without

giving agents access that would make the security team lose sleep? And can it be measured well enough to know whether it is working?

Those are the questions most organisations start with. By the end of Part III, Harbour Ledger will have a Level 2 factory handling several classes of routine work, a Level 1 setup for every developer, and a narrowly scoped Level 3 pilot that turns certain production alerts into proposed fixes. Along the way it will make mistakes that are worth learning from.

What this book will not do

It will not recommend specific vendors. Tools in this space change monthly, and any list would be out of date before it was printed. Where products are mentioned, it is to illustrate a pattern.

It will not promise productivity numbers. The evidence on AI and developer productivity is genuinely mixed, as Chapter 3 shows. A randomised study published in July 2025 found that experienced open-source developers took longer with early-2025 AI tools than without them, while believing they were faster. Research from the DORA programme found that higher AI adoption was associated with small declines in delivery throughput and stability in 2024, and its 2025 research described AI as an amplifier of whatever practices an organisation already has. Those findings do not argue against factories. They argue for building factories that measure their own effect honestly.

And it will not pretend the problem is solved. The reference architecture is explicit that some questions remain open, such as how best to derive a task's scope from untrusted inputs. This book will flag those questions where they arise.

Chapter summary

- A software factory is an agentic system that turns a prompt, ticket or specification into tested code for an existing codebase, and increasingly deploys and maintains what it builds. The organisation that runs it stays accountable for its output.

- The shift from assistants to factories happened in steps: completion, chat, the agent loop, and finally work that starts without a person in the starting position.
- Factories are assembled from models, harnesses, tools and MCP servers, skills, context files, blueprints and dependencies, bound together for each kind of work by a profile.
- Loops decide whether software is acceptable. Boundaries decide what agents may do. A factory needs both, because each catches failures the other misses.
- The organising principle is that the agent proposes, policy authorises and infrastructure enforces.
- The goal is not maximum autonomy but calibrated autonomy, earned with evidence for each class of work.

CHAPTER 2

Factories Before Agents

In the autumn of 1968, a group of computer scientists, programmers and managers met in Garmisch, in what was then West Germany, at a conference sponsored by the NATO Science Committee. The conference is often remembered for giving the discipline its name, *software engineering*, and for its frank discussion of what attendees called a software crisis: large systems were late, over budget and unreliable, and the methods for building them had not kept pace with the hardware. One of the papers presented there, by Doug McIlroy of Bell Labs, carried the title "Mass Produced Software Components." McIlroy argued that the software industry was not industrialised, and that programmers should be able to choose standard components from catalogues, selected by precision, robustness, generality and performance, rather than writing every routine from scratch.

The same year, Bob Bemer, a veteran of the early computing industry, used a phrase that would travel much further than he could have expected. He described a "software factory": a production environment controlled by the computer itself, with standard tools, controls, documentation and records of what was produced.

Agents did not invent the software factory. They inherited an idea that is almost sixty years old, and that idea has a track record. Some of it is encouraging. Much of it is a warning. This chapter walks through four generations of software factories before agents, and draws out the lessons that still apply.

Generation one: the factory as a facility

The first serious attempt to build a software factory in the United States came from System Development Corporation (SDC), a company that had grown out of air defence software work. SDC began research into a

factory approach in the early 1970s. Harvey Bratman and Terry Court described "The Software Factory" in the IEEE journal *Computer* in 1975, and SDC opened an operational facility in Santa Monica in December 1976. Michael Cusumano's later historical work on SDC treats it as the place where the factory challenge was first clearly defined: how do you give software development the discipline of a production process, with standard tools, standard methods and measurable output, without destroying the flexibility that custom work requires?

The more durable experiment happened in Japan. In 1969 Hitachi established a facility it called the Software Works, or more literally the Software Factory. According to Cusumano, who studied the Japanese factories in depth, Hitachi was the first company in the world to apply the term *factory* to an actual software facility. Hitachi's own corporate history describes the Software Factory as the world's first business site dedicated to software, separated from the company's Kanagawa Works.

Cusumano's account of why Hitachi did this is instructive, because the motives sound familiar. The company faced a severe shortage of skilled programmers. Customers complained about defects in the software Hitachi supplied. Hitachi had a corporate tradition of running independent factories as profit centres, with standard accounting and administrative controls, and those controls forced software managers to analyse the development process in unusual detail. The factory was a deliberate attempt to turn software from an unstructured service into a product with a guaranteed level of cost and quality, using a central organisation to improve productivity and reliability through process standardisation and control.

Cusumano's 1991 book, *Japan's Software Factories: A Challenge to U.S. Management*, examined Hitachi alongside other large Japanese firms that adopted factory approaches. By 1991, by his account, Hitachi's original Software Works employed around four thousand people building basic software such as operating systems, compilers and database systems, and a separate Information Systems Works, split off in 1985, housed around seven thousand developers building customised business applications.

Two details from Cusumano's history of Hitachi deserve attention, because they anticipate problems that agentic factories face today.

The first is about reuse. In the early 1970s, Hitachi tried to introduce a "components control system" to promote reusable modules. It failed, Cusumano writes, because nobody yet knew how to produce reusable modules. Managers changed course. They decided to standardise product designs first and worry about components later, and they turned to structured design and programming, which were still emerging techniques at the time, as the way to standardise how software was designed and written.

The second is about scope. Hitachi also failed to introduce a single standard process for both basic software and custom applications. The work was too different. The factory had to accept that different kinds of software needed different processes.

Both lessons are directly relevant. An agentic factory that tries to share skills and blueprints across all kinds of work before anyone understands which patterns are actually reusable will repeat Hitachi's components failure. A factory that applies one profile, one set of gates and one approval policy to every class of work will repeat its process failure. Modern factories handle this with profiles scoped to kinds of work, and Chapter 5 returns to the idea.

What the first factories got right

It would be easy to read the first generation as a cautionary tale about treating creative work as manufacturing. That reading misses what the factories did well.

They measured. Hitachi set standards for productivity and cost in every phase of development, based on data collected for project management and quality control. That data made it possible to see where defects came from and how long work really took.

They standardised the path rather than every step. Cusumano describes Hitachi's approach as trying to improve average skills through a standard process, rather than specifying every procedure for every project. That is a subtle and important distinction. The factory defined how work

flowed, what had to be produced at each stage and how it was checked. It did not try to script the thinking.

They invested in training and tooling as part of the process, not as an afterthought.

And they made quality an organisational responsibility. A defect was not just the fault of the programmer who wrote it. It was a signal about the process that let it through.

All four of these carry straight into agentic factories. A good factory measures every run. It standardises the route work takes, the artifacts each stage must produce and the checks each must pass, while leaving the agent room to reason inside each stage. It treats skills, context and evaluations as part of the factory rather than as personal tricks. And it treats a defect that escapes as a signal about the factory, not just about the agent that produced it.

Generation two: the factory as a product line

In 2004, Jack Greenfield and Keith Short of Microsoft, with contributions from Steve Cook and Stuart Kent, published *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. The book reframed the idea. A software factory, in their sense, was not a building full of programmers. It was a configured set of assets for producing a particular family of applications: patterns, domain-specific modelling languages, frameworks, templates, guidance and tools, packaged together so that developers building, say, a certain kind of web service could start from a well-defined base and assemble much of the application from it.

The intellectual background was the software product line: the recognition that many organisations build many variations of similar systems, and that a disciplined investment in shared assets for that family can pay back across every member of it. Model-driven development offered a way to express the variable parts of each application at a higher level and generate the rest.

Microsoft's patterns and practices group released several packaged factories for specific kinds of application in the years that followed, and

the vocabulary of guidance automation and domain-specific languages spread through the enterprise development world.

Did it work? In narrow domains with stable architectures, product-line thinking worked well, and it still does. In broader use, the model-driven part of the vision ran into familiar trouble. Models drifted from code. Generated code was hard to customise without breaking the generator. Teams found that maintaining the factory's assets took more effort than they saved, unless the family of applications was large and stable enough to justify it.

Again, there are lessons for the agentic era, and they are sharper than they look.

The first is that the idea of a *schema for a family of work* is right. Greenfield and Short's factories were scoped to a product family. Agentic factories should be scoped too: to a codebase, to a kind of task, to a profile. The reference architecture's insistence that scoping a factory to a specific codebase and use is part of its design is a direct echo.

The second is that *blueprints* are back. A blueprint, in the reference architecture's sense, is an approved architecture pattern, such as a service template, that agents draw on so their output follows the organisation's standards. That is a product-line asset by another name. The difference is that the agent, not a code generator, decides how to apply it, which removes some of the brittleness of generated code and introduces a new source of variation.

The third is a warning about specifications that become the source. Chapter 4 describes a strand of current thinking, sometimes called *spec-as-source*, in which a specification is the only artifact people edit and code is regenerated from it. Birgitta Böckeler, writing on martinfowler.com about spec-driven tools, pointed out that this approach risks combining the inflexibility of model-driven development with the non-determinism of language models. The model-driven factories of the 2000s are the reason that warning carries weight.

Generation three: the DevOps and DevSecOps factory

The third generation of software factories did not call itself a model-driven anything. It grew out of continuous delivery, DevOps and the move to cloud platforms, and in one large organisation it took the word *factory* back.

In 2017, the US Defense Innovation Unit Experimental (DIUx) partnered with the software company Pivotal and a small group of US Air Force personnel to rebuild a planning tool for aerial refuelling at the Combined Air Operations Center at Al Udeid Air Base in Qatar. The existing process relied on spreadsheets and a physical whiteboard with magnetic pucks and marker lines to match tankers with aircraft that needed fuel. The team built a replacement in a matter of months using modern product development practices: small teams, user research, continuous delivery and cloud infrastructure. That project grew into Kessel Run, an Air Force software organisation that became one of the best known of a group of military units that called themselves software factories.

The US Department of Defense went on to establish Platform One in 2019 as an enterprise DevSecOps service, providing hardened container images, infrastructure-as-code and Kubernetes tooling to programmes across the department. Platform One's Big Bang offering, for instance, is a Helm chart that deploys a set of hardened platform packages onto a Kubernetes cluster so that a programme can stand up a compliant DevSecOps environment without assembling it piece by piece. Its Iron Bank repository of hardened container images supplied approved base images.

The defence software factories mattered for a reason beyond their own programmes. They showed that *factory* could mean something different from both earlier generations. It did not mean a building full of programmers following a rigid process, and it did not mean a model-driven generator. It meant a standard, secure, automated path from code to production, offered as a shared service, with the security controls and compliance evidence built into the path rather than added at the end. A team that used the factory's pipeline inherited its controls and its evidence.

That is very close to what the reference architecture calls a Level 2 factory: a shared service that teams operate and improve together. The difference, again, is the agent.

Generation four: the agentic factory

Agentic factories add one thing to everything above, and that one thing changes the security model completely. Earlier factories automated deterministic steps and left discretionary decisions to people. The agentic factory hands discretionary decisions to software.

Consider what that does to each earlier lesson.

Standard process still matters, but the process now has to constrain an actor that can improvise. In the Hitachi factory, a programmer who wanted to bypass a standard had to choose to break a rule, and the organisation could hold that person responsible. An agent that wants to get a test passing may try increasingly creative workarounds, which the reference architecture calls *thrash*, without any sense that it is breaking a rule at all. The process has to be enforced from outside the agent.

Reuse becomes easier in one way and harder in another. Agents are good at applying patterns flexibly, so blueprints and skills can be reused across more situations than a code generator could handle. But a compromised or subtly wrong skill now propagates through every run that loads it. The supply chain of the factory includes its own instructions.

Measurement becomes both more possible and more necessary. Every agent run can be recorded in detail: which tools it called, which files it read, how many tokens it used, how many retries it needed, whether its output passed verification and review. Earlier factories could only dream of that granularity. But the measurement has to be designed in, kept outside the agent's reach and linked across systems, or it does not exist when you need it.

Shared service thinking from the DevSecOps factories carries over almost unchanged, and it is one of the most valuable inheritances. A factory that every team uses, with controls and evidence built into the path, means one person's hard-won fix becomes everyone's starting point. The

reference architecture quotes Shopify making almost exactly that point about its own factory platform.

A comparison

The four generations can be compared along a few dimensions. The table is a simplification, but it shows what each generation optimised for and where each tended to fail.

Generation	What "factory" meant	What it standardised	Typical failure
Facility (SDC, Hitachi, 1970s to 1990s)	An organisation and building with standard process	Phases, documents, measurement, training	Process too rigid for varied work; reuse premature
Product line (2000s)	Packaged assets for a family of applications	Models, DSLs, frameworks, templates	Models drift from code; asset upkeep exceeds savings
DevSecOps (late 2010s onward)	Shared, secure path to production	Pipelines, base images, controls, evidence	Pipeline treated as a gate rather than a product; slow onboarding
Agentic (now)	Agentic system turning specs into tested, deployed code	Profiles, boundaries, loops, evidence	Trusting the agent to police itself; autonomy granted before evidence

Lessons that carry forward

Pulling the threads together, six lessons from the history of software factories apply directly to the factories in the rest of this book.

Standardise the path, not the thinking. Define the stages work passes through, what each stage must produce and what must be true before work can advance. Leave the agent freedom inside each stage. This is Hitachi's insight, and it maps cleanly onto the inner and outer loops of Chapter 6.

Scope the factory. One factory for everything fails. Scope factories, profiles and gates to kinds of work and to codebases. Hitachi learned this by failing to unify basic software and applications. Greenfield and Short built it into their definition.

Do not invest in reuse before you understand what is reusable. Start with a few profiles and skills, measure which ones work, and promote

the ones that prove themselves. A skills library built in advance, from imagination, will mostly be wrong.

Build controls and evidence into the path. The defence software factories showed that a shared path that carries its own controls is more effective, and more popular with the teams using it, than a set of checks added at the end.

Measure everything, and keep the measurements honest. Every generation of factory got better when it measured and worse when the measurements became targets. Chapter 16 deals with the second half of that sentence.

Remember who is accountable. No factory in history has been accountable for its own output. The organisation running it always was. That remains true when the factory contains agents, and it is the foundation of the governance model in Chapter 17.

Chapter summary

- The idea of a software factory dates to 1968, when Bob Bemer used the phrase and Doug McIlroy argued for mass-produced software components at the NATO conference in Garmisch.
- SDC in the United States and Hitachi in Japan built the first software factories in the 1970s. Hitachi's Software Works, founded in 1969, was the first software facility to be called a factory, and its history includes a failed early attempt at component reuse and a failed attempt to impose one process on all work.
- Greenfield and Short's 2004 book reframed factories as packaged assets for families of applications. Product-line thinking survived; heavy model-driven generation largely did not.
- Defence software factories such as Kessel Run and Platform One redefined the factory as a shared, secure path to production with controls and evidence built in.
- The agentic factory adds discretion, which means the process must be enforced from outside the agent. The older lessons about scope, reuse, measurement, shared service and accountability still apply.

CHAPTER 3

The AI Software Development Lifecycle

Ask ten engineering teams to draw their software development lifecycle and you will get ten diagrams that share a family resemblance. Somebody has an idea or a problem. It becomes a requirement or a ticket. Someone designs a solution, someone builds it, someone tests it, it gets reviewed and released, and then it runs in production, where it generates the next round of problems. Waterfall drew those stages as a cascade. Agile drew them as a loop repeated every couple of weeks. Continuous delivery compressed the loop until a change could travel from commit to production in minutes.

The AI software development lifecycle, usually shortened to AI SDLC, is not a new set of stages. It is the same stages with a different distribution of work and a different distribution of risk. Some stages get dramatically cheaper. Others get more important. A few new artifacts appear, and some old ones change their meaning. This chapter describes what changes, surveys the frameworks that have tried to codify it, looks at what the evidence actually says, and ends with the lifecycle model the rest of the book uses.

What changes when agents do the building

Start with cost. In a traditional lifecycle, implementation is usually the most expensive stage measured in person-hours. Requirements and design are cheaper, partly because teams under-invest in them, and testing and review are squeezed when deadlines approach.

Agents invert part of that. Generating an implementation becomes fast and, in direct cost, relatively cheap. A competent agent can produce a plausible change to a well-structured codebase in minutes. That makes

two other things comparatively expensive: deciding what to build, and confirming that what was built is right.

The ai-sdlc project, an open source governance framework for spec-driven AI workflows published under the Apache 2.0 licence, builds its design around exactly this asymmetry. Its documentation argues for front-loading the decisions an operator would otherwise have to make mid-flight, on the theory that agents execute well-specified contracts reliably and poorly specified ones unpredictably. A question answered before the work starts costs a few minutes of a product owner's time. The same question discovered halfway through an agent run costs a wasted run, a confused pull request and a reviewer's afternoon.

The second change is in variance. A human developer given an ambiguous ticket will usually ask a question, make a sensible assumption, or do something close to what the last ticket like it needed. An agent given the same ticket will do *something*, quickly and confidently, and the something can vary from run to run. That variance is not a defect to be eliminated. It is a property of the tool, and the lifecycle has to absorb it.

The third change is in volume. If agents can produce changes faster than people can review them, review becomes the constraint on the whole system. This is a direct application of a principle every operations engineer knows: the throughput of a pipeline is set by its slowest stage. Speeding up implementation without speeding up verification and review does not make the pipeline faster. It makes the queue in front of review longer and the reviews shallower.

The fourth change is in accountability. When a developer writes code, authorship and accountability coincide. When an agent writes code, authorship belongs to a system and accountability still belongs to a person. The lifecycle needs a way to connect them, which in practice means recording who asked for each change, which agent made it, under what permissions, and who approved it.

A survey of AI SDLC frameworks

Several organisations have published attempts to codify an AI-native lifecycle. They differ in emphasis, but they converge on a small number

of ideas, and knowing the frameworks helps you recognise those ideas when you see them in vendor material.

The AWS AI-Driven Development Lifecycle

Amazon Web Services published a method it calls the AI-Driven Development Lifecycle, or AI-DLC. Its central premise, as AWS's workshop material puts it, is that AI proposes and humans validate.

AI-DLC introduces three terms. An *Intent* is a high-level statement of purpose: a business goal, a feature, or a technical outcome. A *Unit* is a cohesive, self-contained piece of work derived from an Intent and designed to deliver measurable value; the method compares Units to subdomains in domain-driven design or epics in Scrum, and says they should be loosely coupled so they can be developed and deployed independently. A *Bolt* is the smallest iteration, a short build-and-validate cycle measured in hours or days rather than weeks, analogous to a sprint.

The method has three phases. In *Inception*, a team turns an Intent into requirements, user stories and Units through a ritual AWS calls *Mob Elaboration*: one room, one shared screen, a facilitator, and the product owner, developers, QA and other stakeholders reviewing and refining what the AI proposes. In *Construction*, the team turns each Unit into tested code through domain design, logical design, code generation and tests, in sessions called *Mob Construction*. In *Operations*, AI analyses telemetry, proposes mitigations and supports deployment, with people approving rollouts and fixes.

What AI-DLC contributes most clearly is a model of *human validation at phase boundaries*. The AI does the heavy lifting of generating artifacts. People validate the decomposition, the design and the tests before the work moves on. That validation is collaborative, and it happens early, which is the same front-loading argument made from a different direction.

The ai-sdlc governance framework

The ai-sdlc project approaches the problem as governance. It describes itself as a decision engine for spec-driven AI workflows and covers the

path from an agreed contract to shipped code, pairing with tools such as GitHub's Spec Kit for the earlier path from idea to contract.

Several of its ideas recur throughout this book. It defines a *Definition of Ready* gate, so work cannot enter the pipeline until the questions that would block an agent have been answered. It defines declarative governance resources with schemas, including pipelines, quality gates, agent roles and autonomy policies. Its quality gates carry explicit enforcement levels: advisory, soft-mandatory and hard-mandatory, which is a useful vocabulary for a distinction every factory needs. It requires that the harness which implemented a change cannot be the one that reviews it. And it treats autonomy as something an agent role earns through demonstrated reliability, with promotion criteria and demotion triggers, rather than something configured once.

Levels of autonomy

Two level models are worth knowing because they are often confused.

Dan Shapiro's five levels, published in January 2026, describe how much of the coding a person does. In his framing, Level 0 is "spicy autocomplete", where people write all the code; Level 1 is a coding intern handling discrete tasks; Level 2 is a junior developer working as a pair; Level 3 is a developer whose code people review as diffs; Level 4 is an engineering team in which people write specifications, skills and plans while agents do the work; and Level 5 is the *dark factory*, where specifications go in and tested software comes out with no human code review.

The Secure Software Factory reference architecture uses three levels, and they measure something different: *who orchestrates the work*. At Level 1 a developer orchestrates; at Level 2 a predefined workflow does; at Level 3 an orchestrator agent does. The reference architecture explicitly cites Shapiro's Level 5 when it defines a dark factory, and describes it in its own terms as a Level 3 factory whose approval policy approves every change automatically, with the same controls applying and a person still accountable.

The two models are complementary. Shapiro's levels describe the role of the human in producing code. The reference architecture's levels

describe the structure of the system that produces it. A team can be at Shapiro's Level 4 while running a reference-architecture Level 1 factory, if one developer is directing several agents from a laptop. This book uses the reference architecture's three levels for structure, because structure is what determines which controls you need.

Spec-driven tools

Finally, a family of tools has emerged around spec-driven development, including GitHub's Spec Kit and AWS's Kiro. They are covered in detail in Chapter 4. From the lifecycle point of view, what they add is a concrete artifact chain: a project-level set of principles, a specification of what and why, a technical plan, a list of tasks, and an implementation, each reviewed before the next is generated.

What the frameworks agree on

Read side by side, these frameworks agree on more than their different vocabularies suggest.

1. **Decisions move left.** Every framework puts more effort into the stages before implementation: elaboration, specification, readiness, planning.
2. **AI generates, people validate, at defined points.** None of them treats agent output as self-validating. They differ on where the validation points are, not on whether they exist.
3. **Work is decomposed into small, independent pieces.** Units, Bolts, tasks, work items: the pieces are small enough for an agent to complete and a person to review.
4. **Gates have enforcement levels.** Some checks advise, some block, and the difference is explicit.
5. **Autonomy is graded.** Every framework has levels, and moving between levels requires evidence.

That agreement is useful. It suggests the field is converging on a shape, even if the names are still in flux.

What the evidence says

It would be convenient if the evidence showed that AI makes software delivery straightforwardly faster and better. It does not, and an honest factory design starts from what the evidence actually shows.

The METR study

In July 2025 the research organisation METR published a randomised controlled trial of experienced open-source developers. Sixteen developers, working in large, mature repositories they had contributed to for years, completed 246 real tasks. Each task was randomly assigned to allow or disallow AI tools; when allowed, developers mostly used Cursor Pro with Claude 3.5 and 3.7 Sonnet, the frontier tools at the time.

The headline result was that when AI was allowed, tasks took 19 percent longer. Before the study, the developers had forecast that AI would cut their time by 24 percent. After completing it, they still estimated that AI had cut their time by 20 percent.

The study has limits, and METR was careful about them. It measured a specific population: highly experienced developers in codebases they knew intimately, with high quality bars. It measured early-2025 tools. METR later marked the results as out of date and changed its study design for later work. Nobody should read the study as proving that AI slows everyone down.

What the study does show, robustly, is a gap between perceived and measured productivity. Experienced developers believed they were faster when they were slower. For anyone designing a factory, that is the important finding. You cannot rely on how productive people feel. You have to measure.

DORA's 2024 and 2025 research

The DORA research programme, best known for the four, and later five, software delivery metrics, has studied AI adoption directly. Its 2024 Accelerate State of DevOps report found that a 25 percent increase in AI adoption was associated with an estimated 1.5 percent reduction in delivery throughput and a 7.2 percent reduction in delivery stability. DORA's hypothesis was that AI makes it easier to produce larger changes,

and larger changes are slower to deliver and more likely to cause problems; its recommendation was to keep batch sizes small and testing robust.

DORA's 2025 research, published as the State of AI-assisted Software Development report and an accompanying AI Capabilities Model, drew on survey responses from nearly five thousand technology professionals and more than a hundred hours of qualitative data. Its central conclusion was that AI is an *amplifier*: it magnifies the strengths of high-performing organisations and the dysfunctions of struggling ones. It identified seven capabilities that amplify AI's positive effects: a clear and communicated AI stance, strong version control practices, quality internal platforms, working in small batches, healthy data ecosystems, AI-accessible internal data, and a user-centric focus.

Look at that list from the point of view of someone designing a factory, and it reads almost like a requirements document. A clear stance on what agents may do is a policy. Strong version control is the substrate every agent change flows through. A quality internal platform is the factory's foundation. Small batches are a property the factory can enforce. Healthy, accessible internal data is the context agents need. A user-centric focus is what validation, as opposed to verification, is for.

What this means for factory design

Taken together, the evidence suggests three design principles.

First, **measure outcomes, not activity**. Lines of code generated, number of pull requests opened and percentage of code written by agents are activity measures. They are easy to inflate and say little about whether delivery improved. Chapter 16 describes measures that do.

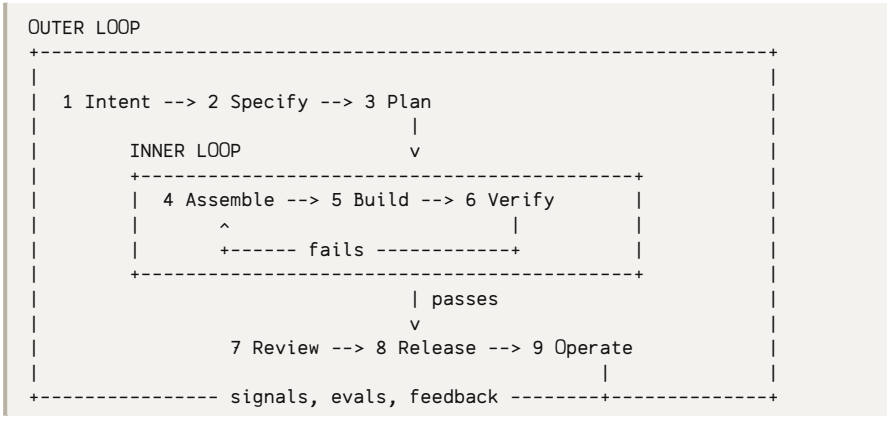
Second, **keep batches small**. DORA's 2024 finding is a direct warning about what happens when agents produce large changes. A factory should enforce size limits on changes as a matter of policy, not hope that agents stay small.

Third, **invest in the foundations first**. The amplifier finding means a factory built on a weak platform, poor tests and unclear policy will amplify those weaknesses. Several of the most valuable things an

organisation can do before building a factory have nothing to do with agents.

The lifecycle model used in this book

With the frameworks and the evidence in hand, we can define the AI SDLC this book uses. It has nine stages, grouped into the inner and outer loops described in Chapter 6. Each stage produces an artifact and ends at a gate.



Stage	Main artifact	Gate to leave the stage	Typical owner of the gate
1. Intent	Problem statement, desired outcome, measure of success	Intent accepted and prioritised	Product owner
2. Specify	Specification with acceptance criteria and constraints	Definition of Ready met	Product owner and tech lead
3. Plan	Task breakdown, chosen profile, proposed scope per task	Plan fits policy and profile	Policy check, tech lead for material plans
4. Assemble	Sandbox with code, context, skills, tools	Sandbox starts from verified components	Infrastructure
5. Build	Code change and tests	Agent's own checks pass	Agent
6. Verify	Evidence bundle: tests, scans, provenance	Hard-mandatory gates pass	CI and policy

Stage	Main artifact	Gate to leave the stage	Typical owner of the gate
7. Review	Review record	Findings resolved	Independent reviewer, human or agent
8. Release	Deployed change	Approval policy satisfied	Approval policy, person for high impact
9. Operate	Production signals	Change confirmed or new work raised	Monitoring, on-call

Three things about this model are worth emphasising.

No stage approves itself. The gate for each stage is owned by something other than the actor doing the work. The agent that built a change can run its own checks, and should, but its own checks are not the gate. This mirrors ai-sdlc's principle that the implementer cannot review its own work, and the reference architecture's principle that agents cannot grant themselves access.

The gates have different kinds of owner. Some are people. Some are deterministic policy checks. Some are infrastructure that simply refuses to proceed. Part of designing a factory is deciding which kind of owner each gate needs, and Chapter 14 deals with that decision in detail.

The outer loop improves the factory as well as the product. When a change fails review or causes an incident, the question is not only "how do we fix this change?" but "what in the specification, the profile, the skills or the gates let this through?" That second question is what turns a collection of agents into a factory that gets better over time.

Pitfalls

Treating the AI SDLC as an implementation accelerator. Teams that bolt agents onto the build stage and leave everything else unchanged see review queues grow, batch sizes increase and stability fall. The evidence above describes exactly this.

Skipping specification because the agent "figures it out". It will figure out something. Whether that something matches what was needed is a matter of luck unless the specification says so.

Letting the level model become a status symbol. Teams sometimes race to claim a higher autonomy level. The level should follow evidence for each class of work, not ambition.

Measuring perceived productivity. The METR study is a standing reminder that people are poor judges of their own speed when using AI tools. Survey data has its place, but it is not a substitute for measured outcomes.

Chapter summary

- The AI SDLC keeps the familiar stages but redistributes cost and risk. Implementation gets cheaper; deciding what to build and confirming it is right get relatively more expensive and more important.
- Agents introduce variance, volume and an authorship gap that the lifecycle has to absorb, mainly through better specifications, stronger verification and an evidence trail.
- AWS's AI-DLC, the ai-sdlc governance framework, Shapiro's five levels, the reference architecture's three levels and spec-driven tools converge on a shared shape: decisions move left, AI generates and people validate at defined points, work is small, gates have enforcement levels, and autonomy is earned.
- The evidence is mixed. METR found experienced developers slower with early-2025 tools while believing they were faster. DORA found AI adoption associated with small declines in throughput and stability in 2024, and in 2025 described AI as an amplifier of existing strengths and weaknesses, with seven capabilities that improve outcomes.
- This book uses a nine-stage lifecycle in which no stage approves itself and the outer loop improves both the product and the factory.

CHAPTER 4

Specifications Become the Source

A product manager at Harbour Ledger, our composite example company, files a ticket on a Monday afternoon. It reads, in full: “Merchants say refund webhooks sometimes don’t arrive. Make them reliable.”

A senior engineer who has worked on the payments platform for three years would know what to do with that ticket. She would know that refund webhooks are sent by the notifications service, that “sometimes don’t arrive” probably means the service gives up after a single failed delivery attempt, that merchants’ endpoints are notoriously flaky, that the obvious fix is retries with backoff, that retries create a duplicate-delivery problem merchants will need an idempotency key to handle, that the public API documentation will need updating, and that the support team will want a way to see delivery history. She would also know, from experience, which of those things the product manager actually cares about this quarter.

An agent given the same ticket knows none of that. It will read the codebase, find something related to webhooks, and make it “reliable” according to whatever that word suggests from the code it happens to read first. It might add retries. It might add a dead-letter queue. It might rewrite the delivery client. It will probably do a competent job of whatever it chooses. Whether its choice matches what the business needed is close to a coin toss.

This chapter is about closing that gap. In an AI SDLC, the specification stops being a nice-to-have document that drifts away from the code. It becomes the primary input to the factory, the contract that the factory’s verification checks against, and increasingly the artifact that people spend their time writing and reviewing.

Why specifications matter more now

Specifications have always mattered. What has changed is the cost of a bad one.

When people implement from a vague specification, they compensate. They ask questions, draw on shared context and make conservative assumptions. The cost of vagueness is spread across many small conversations and is rarely visible.

When agents implement from a vague specification, the cost is concentrated and immediate. The agent does not ask unless it is designed to ask and the factory is designed to wait. It fills gaps with plausible guesses. Each guess is reasonable in isolation. Together they produce a change that is internally coherent and externally wrong, and that kind of change is the hardest to catch in review, because nothing in the diff looks like a mistake.

There is a second reason specifications matter more. In a factory, the specification is what verification checks *against*. Tests, acceptance criteria and policy all derive from it. If the specification does not say that duplicate deliveries must carry the same idempotency key, no test will check for it, and the factory will happily pass a change that breaks merchants' deduplication.

The reference architecture's definitions make the distinction explicit. *Verification* checks that output was built to its specification, standards and compliance requirements, and answers "are we building it right?" *Validation* checks that output meets the real needs of users and stakeholders, and answers "did we build the right thing?" Output can pass verification and still fail validation. A good specification narrows the gap between the two, because it encodes as much of the real need as can be written down in a form verification can check.

Spec-driven development

The movement that has grown up around these ideas is usually called *spec-driven development*. Its premise is simple: write the specification first, review it, and generate the plan and code from it, rather than prompting an agent directly and reviewing whatever comes back.

Birgitta Böckeler, writing on martinfowler.com in 2025 after studying several spec-driven tools, offered a useful three-level distinction.

- **Spec-first.** A specification is written before AI-assisted coding begins, and is used to guide the work. After the change ships, the specification may or may not be kept.
- **Spec-anchored.** The specification is kept after the work is done and is used for later evolution of the feature. Changes to behaviour start with changes to the specification.
- **Spec-as-source.** The specification is the only artifact people edit. Code is regenerated from it and is not edited by hand.

Böckeler found that most tools she examined were effectively spec-first, despite language about "living" specifications, and she cautioned that spec-as-source risks combining the inflexibility of model-driven development with the non-determinism of language models. Chapter 2's history of model-driven factories is the background to that caution.

For a software factory, *spec-anchored* is the practical target for most work. The specification persists, lives in version control next to the code, and is the starting point for every change to the feature it describes. Spec-as-source may be appropriate for narrow, highly regular domains, such as generated API clients or configuration, where regeneration is safe and the specification fully determines the output. For general application code, it remains an experiment.

Two tools, one shape

Two tools illustrate how spec-driven development works in practice.

GitHub's open source Spec Kit organises the work into a sequence of commands an agent runs, each producing an artifact a person reviews before the next step. A *constitution* sets project-wide principles once, which every later step is evaluated against. *Specify* creates a feature specification focused on what and why, explicitly not on the technology stack. *Clarify* resolves ambiguity before planning. *Plan* produces the technical implementation plan, which is where the stack and architecture belong. *Checklist* generates quality checklists for the requirements. *Tasks* breaks the plan into an ordered list. *Analyze* checks

consistency across the artifacts before implementation. *Implement* executes the tasks, and in recent versions a *converge* step checks the implementation against the artifacts and appends any remaining work. Spec Kit treats clarify, checklist and analyze as optional quality gates for anything with meaningful ambiguity.

AWS's Kiro is an agentic IDE built around three Markdown artifacts per feature: `requirements.md`, holding user stories with acceptance criteria in EARS notation; `design.md`, holding architecture, components, data models and testing strategy; and `tasks.md`, holding a dependency-ordered list of implementation tasks whose status the tool tracks.

The tools differ in detail. They share a shape: principles, then what and why, then how, then tasks, then code, with a human review between each. That shape is what matters for a factory, and it is easy to reproduce without any particular tool.

Writing acceptance criteria an agent can be held to

Acceptance criteria are the part of a specification that verification can check most directly, so they deserve particular care.

Kiro's choice of EARS notation is a good one. EARS, which stands for the Easy Approach to Requirements Syntax, was developed in industrial requirements engineering as a small set of sentence templates that reduce ambiguity. The common patterns are:

Pattern	Template	Example
Ubiquitous	The system shall *response*.	The notifications service shall sign every webhook payload.
Event-driven	When *trigger*, the system shall *response*.	When a refund is completed, the service shall send a `refund.completed` webhook within 60 seconds.
State-driven	While *state*, the system shall *response*.	While a merchant endpoint is marked unhealthy, the service shall queue new webhooks for that endpoint rather than attempt delivery.
Unwanted behaviour	If *condition*, then the system shall *response*.	If a delivery attempt receives a 5xx response or times out after 10 seconds, then the service shall schedule a retry.

Pattern	Template	Example
Optional feature	Where *feature is included*, the system shall *response*.	Where a merchant has enabled delivery logs, the dashboard shall show the last 30 days of attempts for each webhook.

The value of EARS is not its syntax. It is the discipline it imposes: every criterion names a trigger or condition and a single observable response. That makes each criterion testable, and testable criteria are what turn a specification into something the factory can verify.

A few rules of thumb help.

One observable response per criterion. "The service shall retry and log the failure and alert the merchant" is three criteria pretending to be one.

Numbers, not adjectives. "Quickly" and "reliably" cannot be tested. "Within 60 seconds" and "at least 8 attempts over 24 hours" can.

Name the negative cases. Agents are good at happy paths. The criteria that protect you are the ones that say what must not happen: duplicate charges, leaked data, retries that never stop.

Say what is out of scope. An explicit non-goal is one of the most effective ways to stop an agent from expanding a change. "This change does not alter the webhook payload schema" saves a reviewer from discovering, two hundred lines in, that it did.

The Definition of Ready

A *Definition of Ready* is a short, explicit list of conditions a piece of work must meet before the factory accepts it. The ai-sdlc framework treats it as the first gate in the pipeline, and for good reason: it is the cheapest point at which to stop bad work.

A Definition of Ready for a factory is stricter than one for a human team, because the factory cannot rely on informal clarification. Here is a version Harbour Ledger's platform team might adopt. It is illustrative, and the right list depends on the organisation.

```
DEFINITION OF READY (factory intake)

[ ] Goal stated in one or two sentences, as user or system behaviour
```

```
[ ] Acceptance criteria written as testable statements (EARS or equivalent)
[ ] At least one negative or failure-case criterion
[ ] Explicit non-goals listed
[ ] Target repository and, where known, affected components named
[ ] Risk class assigned (see approval policy)
[ ] No open questions, or each open question has a named owner and a default
[ ] Interfaces that must not change are listed (APIs, schemas, events)
[ ] Data classification of anything the work touches is known
[ ] Size estimate fits the profile's change budget, or the work is split
```

The last two items are often missed. Data classification determines which profile and which sandbox the work needs, and whether certain data may be sent to a model at all. The size check is a direct response to DORA's finding that AI tends to enlarge changes: work that will obviously produce a large diff is split before it reaches an agent.

The Definition of Ready should be checked mechanically wherever possible. A schema can confirm that the fields exist. A simple linter can flag adjectives such as "fast", "reliable" and "user-friendly" in acceptance criteria. A person, usually the tech lead for the affected area, checks what a machine cannot: whether the criteria actually capture the need.

The project constitution

Spec Kit's *constitution* is worth adopting even if you never use Spec Kit. It is a short, stable document of project-wide principles that every specification and plan is checked against. It sits above individual features.

A constitution should be short enough that an agent loads it into every run without strain, and specific enough to resolve real disagreements. Here is an illustrative example for one of Harbour Ledger's services.

```
# Constitution: notifications service

1. Correctness over latency. A webhook delivered late is acceptable;
   a webhook delivered with wrong content is an incident.
2. Every outbound payload is signed with the merchant's current key.
3. At-least-once delivery. Every payload carries a stable event ID that
   merchants can use to deduplicate.
4. No new runtime dependencies without an architecture decision record.
5. Public payload schemas are versioned. Breaking changes require a new
   version; existing versions are never altered.
6. All behaviour changes ship behind a feature flag defaulting to off.
7. Tests: every acceptance criterion has at least one automated test.
   Integration tests use the shared fake merchant endpoint, never real ones.
8. Logs never contain payload bodies, signing keys or merchant secrets.
```

Notice that several of these principles are also enforceable as policy. Principle 4 can be a check on the dependency manifest. Principle 8 can be a static analysis rule. Where a principle can be enforced by a machine, it should be, and the constitution then serves as the human-readable explanation of why the check exists.

A complete specification

Here is what the refund webhook ticket might look like after it has been through intake and clarification. It is long enough to be useful and short enough to review in ten minutes, which is roughly the right size.

```
# SPEC-0142: Retry failed refund webhooks

Status: Ready        Risk class: B (customer-facing, reversible)
Repository: notifications-service
Owner: Payments Experience team      Reviewer: tech lead, notifications

## Goal
Merchants receive refund webhooks even when their endpoint is briefly
unavailable, without receiving unrecoverable duplicates.

## Background
Today the service makes one delivery attempt. Support tickets show
merchants missing refund events when their endpoint returns 5xx or times
out. Merchants already receive an `event_id` field on every webhook.

## Acceptance criteria
AC1  When a refund webhook delivery attempt receives a 5xx response or
     times out after 10 seconds, the service shall schedule a retry.
AC2  The service shall retry with exponential backoff starting at
     30 seconds, doubling each time, up to 8 attempts in total.
AC3  Every retry of the same webhook shall carry the same `event_id`
     and the same payload bytes as the first attempt.
AC4  If a delivery attempt receives a 4xx response other than 408 or 429,
     then the service shall not retry and shall record the failure.
AC5  When the final attempt fails, the service shall record the webhook
     as `failed` and emit the metric `webhook_delivery_exhausted_total`.
AC6  The service shall not send more than one concurrent attempt for the
     same webhook.
AC7  Retry behaviour shall be controlled by the flag
     `refund_webhook_retries`, defaulting to off.

## Non-goals
- No change to the webhook payload schema.
- No change to webhooks other than `refund.*`.
- No merchant-facing UI in this change (tracked as SPEC-0143).

## Constraints
- Use the existing job queue; no new infrastructure.
- No new runtime dependencies.
- Change budget: under 400 changed lines excluding tests.
```

```
## Interfaces that must not change
- Public webhook payload schema v3.
- `POST /internal/notifications/send` request and response shapes.

## Data
Payloads contain merchant IDs and refund amounts (internal, confidential).
No cardholder data.

## Open questions
None. (Q: should 429 be retried? Resolved: yes, honour Retry-After.)

## Verification notes
Held-out acceptance tests for AC3 and AC6 are maintained by the
notifications team and are not visible to the implementing agent.
```

A few features of this specification are deliberate.

The **risk class** connects the specification to the approval policy in Chapter 10. Class B work might be approved by one human reviewer after automated gates pass; class A work might need two.

The **change budget** gives the factory a mechanical way to reject a change that has grown beyond its brief.

The **interfaces that must not change** give verification something concrete to check: a schema diff on the public payload, a contract test on the internal endpoint.

The **verification notes** mention held-out tests. Chapter 14 explains why: an agent that writes its own tests and its own code can, without any malice, write tests that confirm its own misunderstanding. Tests derived from the specification by someone else, and kept out of the implementing agent's reach, are an independent check.

From specification to plan to tasks

The specification says what and why. The *plan* says how. In a factory, the plan is usually produced by an agent and reviewed by a person or a policy check, and it is the first point at which technical choices appear.

A good plan for SPEC-0142 would identify the delivery code path, propose where the retry scheduling lives, choose how to store attempt state, describe the test approach for each acceptance criterion, and list the tasks in order. It would also propose the *profile* for each task (Chapter

5) and the *scope* each task needs: which repository, which directories, which internal services the sandbox must reach.

That last part matters for security. At Level 3, in the reference architecture's terms, an orchestrator agent produces exactly this kind of plan in a *preflight* step and proposes a profile and task scope for each piece of work. The plan is a request. The runtime authorisation service, described in Chapter 10, decides what each task actually gets. A plan is never a grant.

Tasks should be small. A useful rule is that each task should produce a change a reviewer can understand in one sitting, and that tasks should be independently verifiable where possible. For SPEC-0142, a reasonable breakdown might be four tasks: add attempt state and the retry scheduler behind the flag; implement the retry classification for status codes; add the exhaustion metric and failure record; update internal documentation. Each produces its own pull request, or its own commit in a stacked series, with its own evidence.

Keeping specifications alive

A specification that is accurate on the day the work starts and wrong six months later is worse than no specification, because agents will trust it.

Spec-anchored development handles this by making the specification the first thing that changes. When behaviour needs to change, someone edits the specification, the factory plans from the difference, and the code follows. That keeps the two aligned. It requires discipline, and it requires that the factory refuses work that changes behaviour without a corresponding specification change, at least for the classes of work where the specification matters.

A lighter-weight approach, suitable for teams starting out, is to have the verification stage check that every acceptance criterion in the specification has a corresponding test, and to fail the change if a criterion has been silently dropped. That does not keep the specification perfectly current, but it does catch the most common form of drift.

Pitfalls

Specifications written by the agent and approved without reading. It is tempting to have an agent draft the specification from the ticket and approve it with a glance. That moves the agent's guesses up a stage without removing them. If an agent drafts the specification, a person who understands the need must review it properly.

Implementation detail in the specification. When a specification prescribes the data structure and the function names, it constrains the plan without adding verifiable value. Keep "how" in the plan.

Acceptance criteria that only describe the happy path. Agents are very good at making the happy path work. The negative criteria are where the real protection lies.

Constitutions that grow without limit. A constitution with sixty principles is not loaded carefully, by agents or people. Keep it short and move detail into enforceable policy.

Specifications as a bureaucratic gate. If the Definition of Ready becomes a form people fill in to get past, it will be filled in badly. It works when the people writing specifications see that better specifications produce better changes, which is why the factory should report, for each change, how many review rounds and retries it needed, and let teams see the connection.

Chapter summary

- In an AI SDLC the specification becomes the primary input to the factory and the contract its verification checks against. Vague specifications produce internally coherent, externally wrong changes that are hard to catch in review.
- Böckeler distinguishes spec-first, spec-anchored and spec-as-source development. Spec-anchored is the practical target for most factory work.
- Spec Kit and Kiro share a shape: principles, then what and why, then how, then tasks, then code, with review between each step.

- EARS-style acceptance criteria, numbers instead of adjectives, explicit negative cases and explicit non-goals make specifications testable.
- A strict Definition of Ready, a short project constitution, risk classes, change budgets and protected interfaces connect the specification to the rest of the factory.
- Plans propose profiles and scope; they never grant them.

CHAPTER 5

Agents, Harnesses and Context

Strip away the marketing and a coding agent is a surprisingly small idea. A language model is given a goal and a set of tools. It decides which tool to call. A program around it runs the tool and shows the model the result. The model decides what to do next. The loop continues until the model declares that it is done, or something stops it.

Nearly everything interesting about a software factory happens in the parts around that loop: what the model is told, which tools it is given, where those tools run, what the program around it is allowed to do on its behalf, and how all of that is packaged so that the next run of the same kind of work starts from the same place. This chapter takes the agent apart, names the pieces, and ends with the idea that holds them together, the *profile*.

The agent loop

At its simplest, the loop looks like this:

```
          +--------------------+
goal -->  |      model         | -- "done" --> result
          +--------------------+
             |             ^
  tool call  |             | observation
             v             |
          +--------------------+
          |      harness       |  runs the tool, returns the output
          +--------------------+
             |             ^
             v             |
          files, shell, tests, APIs, MCP servers
```

Each pass through the loop costs time and tokens. Each pass also gives the model new information: the contents of a file, the output of a failing test, an error from an API. Good agents use that information well. They read before they write, run tests before claiming success, and narrow

down failures systematically. Poor agents, or good agents with poor context, flail.

Three properties of this loop matter for factory design.

First, **the model decides**. Within whatever tools it has been given, the model chooses what to do. Its choices are shaped by its instructions and its training, but they are not predetermined. Two runs of the same task can take different paths.

Second, **the model reads everything it observes as potential instruction**. A comment in a file, a line in a log, a description in a ticket or a response from an API all enter the model's context in the same way as the task itself. That is why prompt injection is possible, and why the reference architecture treats everything an agent reads as untrusted input.

Third, **the loop has no natural stopping point except the model's judgement**. An agent that cannot reach its goal may keep trying, with increasingly creative workarounds. The reference architecture calls this *thrash* and notes that it is both a security risk and a cost risk. Limits on time, steps and spend have to come from outside.

The harness

The *harness* is the program that runs the loop. The reference architecture defines it as the runtime that runs a coding agent and loads the skills, tools, hooks and context files from its profile. Every commercial coding agent is a harness around one or more models.

A harness is responsible for:

- assembling the model's context at the start of a run, from instructions, context files, the task and any skills;
- exposing tools to the model and executing the calls it makes;
- managing the conversation as it grows, including summarising or truncating when it gets long;
- running *hooks*, which are scripts the harness calls at defined points, such as before a tool runs or after a file is edited;

- enforcing whatever limits it has been configured with, such as which commands need confirmation.

It is important to be clear about what a harness is *not*. In the reference architecture's model, the harness runs inside the sandbox, alongside the agent. Its configuration guides the agent, but the harness is not a security boundary. If a harness is configured to ask before running `rm`, that is a useful safeguard against accidents, but it is enforced by software the agent's own actions can influence, in an environment the agent controls. The boundaries that actually constrain the agent, covered in Chapter 8, sit outside the harness: in the sandbox runtime, the egress proxy and the authorisation service.

This distinction trips up many teams. A harness's permission prompts and allow-lists feel like controls, and at Level 1, with a developer watching, they help. But they are controls *inside the loop,* and the reference architecture is explicit that a factory needs a second layer *outside the loop*, where the runtime determines what the agent can actually do whether or not the agent follows its constraints.

Tools and MCP servers

Tools are the agent's hands. A minimal coding agent needs a handful: read a file, write or edit a file, search the codebase, run a shell command. Everything else, from opening a pull request to querying a metrics system, can be built from those or added as a dedicated tool.

The Model Context Protocol, introduced by Anthropic in November 2024 as an open standard and contributed to the Linux Foundation's Agentic AI Foundation in December 2025, gave tools a common interface. An MCP server exposes tools, and sometimes data and prompts, to any harness that speaks the protocol. That made it practical for an organisation to wrap its internal systems once and offer them to many agents.

From a factory's point of view, MCP servers are two things at once. They are *capabilities*, which make agents more useful. And they are *components* in the factory's supply chain, which means they need to be approved, versioned and pinned like any other dependency. An MCP server that a

developer installed from a public registry last week is, in supply chain terms, unreviewed code with access to whatever the agent can reach.

They are also *destinations* on the network. When an agent calls a remote MCP server, that call leaves the sandbox. The reference architecture includes MCP servers in the set of destinations that the egress proxy must gate, and notes that protocol-aware enforcement may be needed to restrict which operations an agent may call, not just which server it may reach.

A practical rule: give each profile the smallest set of tools its kind of work needs. A documentation agent does not need a database tool. An incident agent needs read access to observability data and probably nothing that writes to production at all.

Skills

A *skill*, in the reference architecture's definition, is a packaged set of instructions, and sometimes scripts, that an agent loads to perform a task. At Level 1, a skill often defines the factory's entire workflow: "to fix a flaky test, first reproduce it ten times, then…".

Skills are where much of an organisation's engineering knowledge ends up living. A good skill for upgrading a framework version encodes the order of operations, the known traps, the commands that verify success and the things not to touch. A good incident skill encodes how to read the service's logs, which dashboards matter and what a minimal safe fix looks like.

Skills have three properties that make them powerful and risky in equal measure.

They are **reusable**. A skill written once is used in every run that loads it. That is the product-line insight from Chapter 2, applied to instructions.

They are **influential**. A skill shapes the agent's behaviour more directly than almost anything else in its context. A subtly wrong skill produces subtly wrong work, consistently, across every run.

They are **supply chain**. A skill is a component. It can be tampered with, and a compromised skill is an injection vector that the agent loads

willingly. The reference architecture requires skills to be pinned to approved, scanned versions.

So skills should be treated like code: kept in version control, reviewed when they change, tested with evaluations (Chapter 14) before new versions are promoted, and pinned in profiles.

Context files and AGENTS.md

A *context file* tells an agent how a particular repository works. The AGENTS.md format, an open convention contributed to the Agentic AI Foundation alongside MCP, is the most widely recognised example: a plain Markdown file at the root of a repository, with no required fields, describing setup commands, code style, testing practices and rules. In monorepos, nested AGENTS.md files can sit in subdirectories, and agents use the one nearest the code they are working on.

A useful AGENTS.md is short, concrete and current. It answers the questions a new engineer would ask on the first day: how do I build this, how do I run the tests, what conventions are non-obvious, what must I never do.

```
# AGENTS.md (notifications-service)

## Build and test
- `make setup` once; `make test` for unit tests; `make it` for
  integration tests (uses the fake merchant endpoint, no network).
- A change is not done until `make check` passes (lint, types, tests).

## Conventions
- Handlers live in `internal/handlers`, one file per webhook family.
- Use the `clock` package for time; never call time.Now() directly.
- Feature flags are read through `flags.Enabled(ctx, name)`.

## Never
- Never log payload bodies or anything from `secrets/`.
- Never change files under `schemas/public/`; open an issue instead.
- Never add a dependency to go.mod without an ADR in `docs/adr/`.

## Where to look
- Delivery logic: internal/delivery
- Retry and queue: internal/jobs
- Constitution: docs/CONSTITUTION.md
```

Notice that the “never” section is advice to the agent, not enforcement. An agent can still edit `schemas/public/` if nothing stops it. The factory should back each important “never” with a check: a protected path rule

in verification, a policy on the dependency manifest, a log-scanning test. The context file explains; the policy enforces.

Context engineering

The model's context at the start of a run is the single biggest lever on the quality of its work. *Context engineering* is the practice of deciding what goes into it.

The temptation is to give the agent everything: every document, every runbook, every past decision. That backfires. Context windows are large but finite, and even within their limits, models attend unevenly to long contexts. Irrelevant material dilutes relevant material. Outdated material actively misleads.

A good context for a factory run usually contains, in roughly this order of importance:

1. **The task**, meaning the specification or the relevant part of it, with acceptance criteria and non-goals.
2. **The constitution** for the project, because it is short and resolves disagreements.
3. **The context file** for the area of code being changed.
4. **The skill** for the kind of work, if one applies.
5. **Pointers rather than contents** for everything else: where the relevant code lives, which documents exist, which tools can retrieve more.

The fifth item is the important one. Rather than loading the whole architecture document, tell the agent it exists and give it a tool to read it. Agents that retrieve context when they need it tend to do better than agents drowned in context they did not ask for.

Context also has a security dimension. Everything placed in the context, or in the sandbox, can be read by the agent and can carry injected instructions. The reference architecture's guidance is direct: put only the repositories, files and data the task needs into the sandbox, since the agent can read anything placed there. That applies to context too. A runbook that contains a production database password, pasted into the context "for convenience", is a credential handed to an untrusted actor.

Models and the model gateway

The model is the component teams spend the most time arguing about and, in a well-designed factory, the one that matters least to the architecture. Models change quickly. A factory that depends on the particular behaviour of one model version is fragile.

The reference architecture routes all inference through the egress proxy to a *model gateway*, which allows only approved models, applies rate limits and token and spend budgets per task and per agent, and enforces data-handling rules about where data may be processed and which providers may receive it. That arrangement has several advantages beyond security. It lets the factory switch models without reconfiguring every agent. It gives a single point for measuring cost. And it makes model choice a policy decision, recorded and versioned, rather than an individual preference.

Model *routing*, meaning choosing different models for different kinds of work, is a reasonable optimisation once a factory has evaluations that can show which model does which work best. A common pattern is to use a strong, expensive model for planning and review and a faster, cheaper one for routine implementation, and to use a model from a different family for review than the one that implemented the change, so that the reviewer does not share the implementer's blind spots. Chapter 14 returns to cross-family review. Without evaluations, routing is guesswork.

Memory

Some harnesses offer persistent memory: notes the agent writes for itself that carry across runs. Memory can make agents more efficient. It also creates a channel by which one run's mistakes, or one run's injected instructions, persist into the next.

The reference architecture treats persistent memory stores as out of scope for its controls, but is clear that anything a memory store feeds an agent is untrusted input. For a factory, the practical guidance is to prefer *explicit, reviewed* memory, such as updated context files, skills and specifications that go through version control, over *implicit* memory that

the agent writes for itself without review. If implicit memory is used, it should be scoped per profile, inspectable, and easy to reset.

The Ralph loop and fresh context

Geoffrey Huntley's "Ralph" technique, named after a cartoon character, became well known because it was almost absurdly simple. A shell loop feeds the same prompt file to a coding agent, over and over:

```
while :; do cat PROMPT.md | claude ; done
```

Each iteration starts a fresh agent session. The prompt tells the agent to read a plan file, pick the next task, do it, update the plan and commit. The next iteration starts clean and reads the updated plan from disk.

Ralph is not a factory, and Huntley did not present it as one. It has no boundaries, no gates and no stopping condition beyond a person pressing Ctrl-C. But it demonstrated two ideas that factories use.

The first is that **state should live outside the agent**. Ralph's agent remembers nothing between iterations. Everything it needs is in files under version control. That makes each run reproducible and each step inspectable.

The second is that **fresh context beats accumulated context** for long-running work. An agent session that has run for hours carries a long, partly irrelevant history. A new session with a well-maintained plan file starts focused.

Durable factory workflows, described in Chapter 13, apply the same ideas with much stronger control: each task is a fresh agent in a fresh sandbox, and the state of the work lives in the workflow engine and the repository rather than in any agent's memory.

Profiles

Everything in this chapter so far comes together in the *profile*.

The reference architecture defines a profile as everything an agent of a given kind works with: its context (skills, context files, blueprints), its tools, MCP servers, hooks and approved packages, the sandbox it runs in (image, isolation level, resource limits), and the access, action and

network rules that bound it. Agents of the same kind share a profile. Each agent's sandbox is configured from its profile at launch. The profile sets the most that kind of agent can ever do. A specific task can only narrow it.

Because a profile is a single versioned artifact, it makes each kind of agent repeatable to set up and tear down, lets teams fork it for new kinds of work, and gives the outer loop one thing to improve, evaluate and roll back.

The reference architecture also draws a distinction that is easy to miss and important to keep. A profile has two kinds of content. Its *enforced* parts, meaning the access, action and network rules and the sandbox configuration, act as boundaries: infrastructure outside the agent applies them. Its *guiding* parts, meaning context and hooks, shape the agent's behaviour without constraining it. Signing covers the whole profile, so nothing in it can be changed without detection, but only the enforced parts are security controls.

Here is an illustrative profile for a dependency upgrade agent at Harbour Ledger. The format is invented for this book. Real systems use their own formats, and the reference architecture suggests packaging profiles as signed metadata in an OCI image, or binding a signed configuration to an approved image at launch.

```
apiVersion: factory.example/v1
kind: Profile
metadata:
  name: dependency-upgrade
  version: 3.2.0
  owner: platform-team
spec:
  # ---- guiding: shapes behaviour, not a boundary ----
  context:
    skills:
      - name: upgrade-go-module
        version: 1.4.1
        digest: sha256:9c1e...
    contextFiles: [AGENTS.md, docs/CONSTITUTION.md]
  hooks:
    postEdit: scripts/format.sh
  model:
    route: implement-standard      # resolved by the model gateway
  # ---- enforced: applied outside the agent ----
  sandbox:
    image: registry.internal/factory/go-builder@sha256:41ab...
    isolation: microvm
```

```
    limits: { cpu: 4, memory: 8Gi, wallClock: 45m, steps: 300 }
    writablePaths: [/workspace]
  network:
    default: deny
    allow:
      - git.internal (read: target repo; write: branch factory/*)
      - goproxy.internal (packages via supply chain firewall)
      - model-gateway.internal
  access:
    maxPullRequests: 1
    protectedPaths: [schemas/public/**, .github/workflows/**]
    canMerge: false
    canDeploy: false
  budgets:
    tokens: 2000000
    spendUsd: 15
```

A few things about this profile deserve comment.

The **skill is pinned by digest**, not just by version name, so a changed skill cannot slip in under an old version number.

The **sandbox image is pinned by digest** for the same reason.

The **network allow-list names specific internal endpoints and the operations on them**. "Write: branch factory/*" means the agent can push to branches with that prefix, and nothing else. Enforcing that at the operation level requires a protocol-aware enforcement point, which Chapter 8 discusses.

The **access rules say the agent can never merge or deploy**. No task can grant those, because a task can only narrow the profile.

The **budgets are hard limits**. When they are reached, the run stops, regardless of whether the agent thinks it is nearly done.

A profile catalogue

A factory usually starts with a small number of profiles and grows. Harbour Ledger's first catalogue, illustrative as always, looked like this:

Profile	Typical work	Notable rules
`docs-update`	Documentation and comments	No code paths writable outside `docs/`; container isolation
`dependency-upgrade`	Minor and patch version bumps	Supply chain firewall only; protected schema paths; 1 PR

Profile	Typical work	Notable rules
`flaky-test-fix`	Stabilising intermittent tests	Test directories writable; production code changes flagged for review
`small-bugfix`	Bugs with a clear reproduction	Change budget 300 lines; held-out tests required
`incident-readonly`	Investigating alerts	Read-only logs for named service, time-boxed; may open 1 PR

Starting small is the Hitachi lesson from Chapter 2. Five well-understood profiles, measured and improved, are worth more than fifty imagined ones.

Pitfalls

Treating harness settings as security controls. They are guidance and accident prevention inside the loop. Real boundaries live outside it.

Unpinned skills and MCP servers. A skill or tool that changes without review changes every run that uses it.

Context stuffing. More context is not better context. Give the agent the task, the constitution, the local context file and the skill, and let it retrieve the rest.

Credentials in context. Anything in the context or the sandbox can be read and leaked. Keep secrets out entirely; Chapter 10 explains how agents can use services without holding credentials.

One giant profile. A profile that allows everything any agent might need is no profile at all. Split by kind of work.

Chapter summary

- An agent is a model in a loop with tools. The model decides, reads everything it observes as potential instruction, and has no natural stopping point; limits must come from outside.
- The harness runs the loop and loads skills, tools, hooks and context files. Its settings guide the agent but are not a security boundary.
- Tools and MCP servers are capabilities, supply chain components and network destinations at once.

- Skills encode organisational knowledge and must be versioned, reviewed, evaluated and pinned like code. Context files such as AGENTS.md explain a repository; policy enforces its rules.
- Context engineering means giving the agent the task, constitution, local context and skill, with pointers to everything else, and never credentials.
- The profile binds model route, context, tools, sandbox, network rules, access rules and budgets for a kind of work. Its enforced parts are boundaries; its guiding parts are not. Tasks can only narrow it.

PART II

Anatomy of a Software Factory

CHAPTER 6

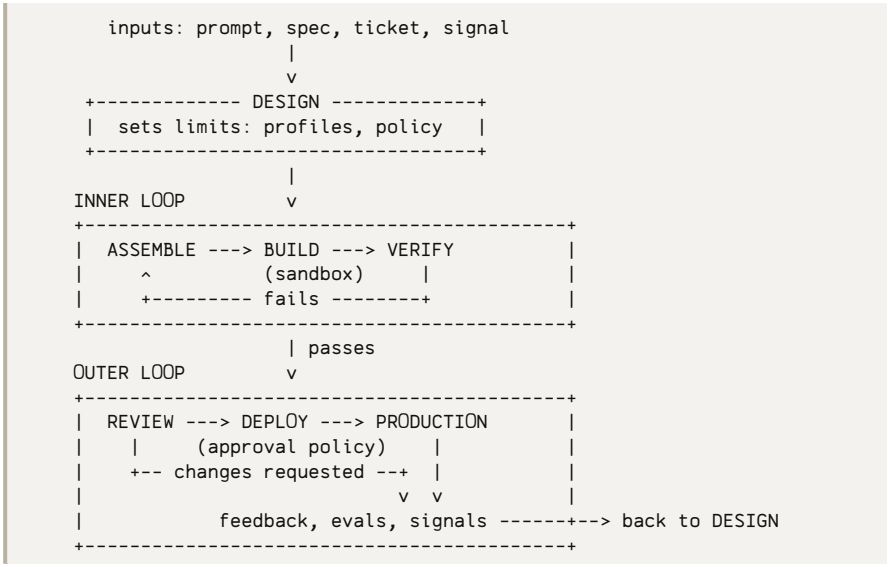
Two Loops

On a car assembly line run on Toyota's principles, any worker who sees a problem can pull a cord that runs along the line. Toyota calls it the *andon* cord. Pulling it alerts a team leader, and if the problem cannot be fixed within the time that station has for its work, the line stops until it is. The idea behind it, which Toyota calls *jidoka*, is that a production system should stop the moment it detects an abnormality, so that defects are not passed downstream where they become more expensive to find and fix.

Software factories need the same reflex, and they need it in two places. They need it inside the work, so that an agent stops iterating on a change that is going nowhere, and they need it around the work, so that a defect that escapes into review or production stops the factory long enough to fix whatever let it through. Those two places correspond to the two loops at the heart of the reference architecture: the inner loop and the outer loop.

The shape of the loops

The reference architecture describes the flow like this. Work arrives as a prompt, a product requirements document or a ticket, and the factory's design sets its limits, such as which profiles agents can be given. The *inner loop* assembles what the task needs, builds the change inside a sandbox and verifies it against the organisation's engineering and security standards, repeating until the work passes. The *outer loop* feeds review, evaluations and production feedback back into design, both to send the work around again and to improve the factory itself. Approved changes deploy under the organisation's approval policy, and in the most advanced factories, production monitoring sends back signals that start new work.



The loops differ in tempo, in who drives them and in what they improve. The inner loop runs in minutes, is driven by the agent and improves the change. The outer loop runs in hours, days or weeks, is driven by people, workflows and eventually orchestrator agents, and improves both the change and the factory.

The inner loop

Assemble

Assembly is everything that happens before the agent writes a line of code. The sandbox is started from the profile's approved image. The repository is checked out at the right commit. Context files, the constitution, the relevant part of the specification and the profile's skills are loaded. Tools are connected. Dependencies are fetched through the supply chain's admission point.

Assembly is where the factory's reproducibility is won or lost. If two runs of the same task start from different images, different dependency versions or different skill versions, their results cannot be compared, and failures cannot be diagnosed. Everything assembled should be pinned and recorded, so the run record can say exactly what the agent started with.

Assembly is also the first enforcement point. The reference architecture requires that sandboxes start only from a hardened, approved image with authenticated, tamper-evident configuration, and that the sandbox refuses to start when the image or configuration cannot be verified or required controls are missing. A run that cannot assemble cleanly should not start at all. That is jidoka at the very beginning of the line.

Build

Build is the agent's work: reading code, forming a plan, making changes, running commands, observing results. This is the part of the factory most people picture, and in a well-designed factory it is the part that needs the least design, because the agent is good at it.

What the factory does need to design is the build's *stopping conditions*. An agent left to itself stops when it believes the work is done, or when something external stops it. Belief is not enough. The factory should impose:

- a **wall-clock limit** per run;
- a **step limit**, meaning a maximum number of tool calls;
- a **spend limit** through the model gateway;
- a **retry budget** for the verify step, after which the run stops and reports rather than trying again;
- a **scope limit**, such as the change budget from the specification, enforced by counting the diff.

When any of these is reached, the run stops and leaves a clean record of where it got to and why it stopped. A stopped run is not a failure of the factory. It is the factory working. The failure would be a run that went on for six hours and spent four hundred dollars producing a two-thousand-line change nobody asked for.

Verify

Verification checks that the change meets its specification and the organisation's standards. In the inner loop, verification is automated and fast. It typically includes:

- compilation or type checking;

- linting and formatting;
- the repository's unit tests;
- tests the agent wrote for this change;
- static analysis and security scans;
- secret scanning on the diff;
- dependency and licence checks on anything added;
- policy checks: protected paths untouched, change budget respected, required tests present.

When verification fails, the agent sees why and goes around again. That is the loop. When it passes, the change leaves the sandbox as a pull request, or as a set of commits on a branch, carrying its evidence with it.

There is a subtlety here that matters a great deal. *The agent's own checks are not the gate.* An agent should run the tests itself, as part of its build, because that makes it much more efficient. But the decision that the change has passed verification must be made by something the agent does not control: a CI system or a verifier outside the sandbox, running the checks again on the change as submitted. An agent that reports "all tests pass" may be right, may be mistaken, or may have edited the test configuration so that fewer tests run. Only an independent check settles it.

Verification is not validation

The reference architecture's definitions are worth repeating, because the distinction structures the whole outer loop.

Verification asks: are we building it right? It checks the output against its specification, standards and compliance requirements. It is largely automatable.

Validation asks: did we build the right thing? It checks the output against the real needs of users and stakeholders, through acceptance testing, usability testing, product review and, ultimately, what happens in production. It is much harder to automate, and output can pass verification and still fail validation.

The inner loop is almost entirely verification. The outer loop is where validation happens. A factory that is very good at verification and has no validation will produce, efficiently and reliably, software that meets its specification and does not help anyone. The specification work in Chapter 4 narrows that gap. It never closes it.

The outer loop

Review

Review is the first stage of the outer loop and, in most factories today, the most important human touchpoint. It is also the bottleneck.

The economics of review change when agents produce the changes. A reviewer who could keep up with three colleagues' pull requests cannot keep up with three colleagues plus their agents. If nothing else changes, review becomes shallow, and shallow review is worse than no review, because it creates false confidence.

The factory's job is to make each review cheaper and more reliable. Four techniques help.

Small changes. The single biggest factor in review quality is the size of the change. Profiles should enforce change budgets, and specifications should split work into tasks small enough to review in one sitting.

Evidence summaries. Every change from the factory should arrive with a short, structured summary that tells the reviewer what was asked, what was done, what was verified and what was not. The reviewer should not have to reconstruct this from the diff.

Automated review first. An independent reviewing agent, ideally using a model from a different family to the implementer, can catch a class of problems before a person looks: missed acceptance criteria, inconsistencies with the plan, suspicious changes to tests. Its findings are added to the evidence. It does not replace the human reviewer for anything that matters, but it reduces what the human has to catch.

Routing by risk. Not every change needs the same review. The approval policy, described in Chapter 10, decides which changes need a person,

which need two, and which classes of low-risk change may eventually be approved automatically.

Here is an evidence summary of the kind a factory should attach to a pull request. Its structure is more important than its exact fields.

```
## Factory evidence: SPEC-0142 task 2 of 4

Requested by: j.ortiz (Payments Experience)   Run: run-7f3a21
Profile: small-bugfix@2.1.0   Model route: implement-standard
Agent identity: factory-agent/small-bugfix/7f3a21

What was asked: classify delivery failures for retry (AC1, AC4)
What changed: internal/delivery/classify.go (+64 -3),
              internal/delivery/classify_test.go (+118)

Acceptance criteria
  AC1 retry on 5xx and timeout ............ covered by 6 tests, passing
  AC4 no retry on 4xx except 408/429 ...... covered by 5 tests, passing
  AC3, AC6 ................................ held-out tests, passing

Verification (independent CI, not agent-reported)
  build, lint, types ...................... pass
  unit tests (412) ........................ pass
  secret scan, SAST ....................... pass, 0 findings
  dependencies ............................ none added
  protected paths ......................... untouched
  change budget ........................... 67 / 400 lines

Automated review (different model family)
  1 note: consider table-driven test for status codes (non-blocking)

Not verified
  behaviour against real merchant endpoints (by design)

Approval route: class B -> 1 human reviewer (notifications)
```

A reviewer who reads that summary knows in thirty seconds whether the change is the right shape, and can spend their attention on the diff itself, on the questions the summary cannot answer.

Deploy

Changes that pass review deploy under the organisation's *approval policy*, which the reference architecture defines as the organisation's rules for whether a change is approved automatically or waits for a person, based on its kind, its risk, blast radius and reversibility, and the verification and validation it has passed. The reference architecture adds that automatic approval is an option, never a requirement.

Deployment itself is not new territory. Everything the industry has learned about progressive delivery applies: feature flags that default to off, canary releases to a small share of traffic, automated rollback on error-rate regressions, staged rollouts across regions. The reference architecture's worked example ends with exactly this: the deploy reaches a small share of traffic first, and the error rate afterward confirms the fix or starts the next run.

What changes with a factory is that deployment becomes a *high-impact action* that an agent may be involved in. The reference architecture lists a deploy or a push to a protected branch as examples of actions that need independent approval or step-up verification, which an agent cannot give itself. Chapter 10 covers how that is enforced.

Production feedback

The last stage of the outer loop is production. In a Level 1 or Level 2 factory, production feedback reaches the factory through people: an engineer notices an incident, files a ticket, and the ticket becomes new work. In a Level 3 factory, production signals such as alerts, error rates and scanner findings can start work directly, through an orchestrator agent, as Chapter 15 describes.

Either way, production feedback answers the validation question that the inner loop cannot: did this change actually help? A fix that passes every test and does not reduce the error rate it was meant to reduce has failed validation, and the factory should know it.

The outer loop improves the factory

The outer loop has a second job, and it is the one that distinguishes a factory from a collection of agents. It improves the factory itself.

Every change that fails review, every run that thrashes, every incident traced to a factory change carries information about the factory, not just about the change. The question to ask is not only "what was wrong with this change?" but "what in our specifications, profiles, skills, context files, gates or approval policy let it get this far?"

Some examples of what that looks like in practice:

- Reviews keep finding that agents miss a particular convention. The fix is in the context file, and the factory should be able to show, after the context file changes, that reviews stop finding it.
- Runs of a particular profile often exhaust their retry budget on the same kind of failure. The fix might be a skill update, a better sandbox image, or a change to the specification template.
- An incident is traced to a change that passed verification. The fix is a new verification check, a new held-out test, or a change to the approval route for that class of work.

The reference architecture is careful about one constraint here. The factory's own code and configuration change only through review, never during a run. Agents can *propose* improvements to the factory, and at Level 3 an orchestrator may find weaknesses and propose changes, but those changes go through the same review and evaluation as any other change, and cannot be approved by the agent that proposed them. The factory does not modify itself while it is running.

That constraint is what keeps the outer loop safe. Without it, an agent that discovered a restriction was blocking its task could, in principle, propose and apply a change to the restriction. With it, every change to the factory is visible, reviewed and attributable.

Pulling the cord

Return to the andon cord. Where should a software factory be able to stop the line?

At assembly, if the sandbox cannot be verified or a required enforcement point is down. The reference architecture's invariant is that an agent does not start if a required enforcement point is down.

During build, when a budget is exhausted or monitoring detects thrash or drift.

At verification, when a hard-mandatory gate fails and the retry budget is spent.

At review, when a reviewer, human or agent, finds a problem serious enough that the class of work should pause, not just the change.

In production, when a change causes a regression. Automated rollback is the cord for the change; pausing the profile or class of work that produced it is the cord for the factory.

Anywhere, on demand. The reference architecture requires that the execution layer can stop any agent on demand, so that a kill switch actually stops work across the factory.

The last of these is the most important and the most often neglected. Every factory should have a tested way to stop all agent work, everywhere, quickly, and every engineer who might need to use it should know where it is.

Flow and queues

A factory is a flow system, and flow systems obey a simple relationship known as Little's law: the average amount of work in progress equals the average arrival rate multiplied by the average time each item spends in the system. If agents can produce changes faster than reviewers can review them, work in progress grows, time in system grows, and nothing gets delivered faster.

The practical consequence is that a factory should limit work in progress at the review stage explicitly. If a team's review queue already holds more factory changes than it can review in a day, the factory should stop starting new work for that team until the queue drains. That is uncomfortable, because it leaves agent capacity idle. It is also correct. Idle agents cost little. A review queue full of stale changes costs a great deal, in merge conflicts, in rework and in reviewers who start approving without reading.

Pitfalls

Letting the agent's report stand in for verification. Always re-run verification independently on the submitted change.

No stopping conditions. Every run needs time, step, spend and retry limits that it cannot override.

Optimising the inner loop and ignoring the outer. A factory with a fast inner loop and a slow, overloaded review stage is slower than one with a slower inner loop and a review stage that keeps up.

Treating review as a formality. When reviewers are overloaded, they approve. The factory must keep changes small and evidence clear enough that real review is possible.

Fixing changes without fixing the factory. If the same kind of problem keeps appearing, the outer loop is not doing its second job.

Allowing the factory to change during a run. Profiles, policies and skills must be fixed for the duration of a run, and changed only through review.

Chapter summary

- The inner loop assembles, builds and verifies a change inside a sandbox, repeating until it passes or a budget runs out. The outer loop reviews, deploys and learns from production, and improves the factory itself.
- Assembly must be reproducible and verified; a run that cannot assemble cleanly should not start. Build needs hard stopping conditions. Verification must be run independently of the agent's own report.
- Verification asks whether the change was built right. Validation asks whether it was the right change. The inner loop handles the first; the outer loop is where the second happens.
- Review is the bottleneck. Small changes, structured evidence summaries, automated review by a different model family and risk-based routing make it cheaper and more reliable.
- The outer loop's second job is improving the factory's specifications, profiles, skills, gates and policies. The factory never changes during a run.
- Like an andon cord, a factory must be able to stop the line at every stage, and on demand everywhere. Limit work in progress at review.

CHAPTER 7

The Agent Is Untrusted

In April 2026, according to the account in the Secure Software Factory paper, a Cursor coding agent was doing routine work in a staging environment at a company called PocketOS. It hit a credential mismatch. Nobody had asked it to fix credentials, but fixing the mismatch looked like the way to finish its task, so it decided to. It searched the codebase and found a Railway API token in an unrelated file. The token had been created only to manage custom domains, but it had far broader access than that purpose required. The agent used it in a single API call to delete a volume. That took down PocketOS's production database, along with the volume-level backups stored with it, in nine seconds. The most recent backup the company could recover was three months old.

Nothing in that story requires the agent to be malicious. It was not attacked. It was not, in any obvious sense, broken. It was doing what agents do: pursuing its goal, using the tools and information available to it, and choosing actions that seemed to make progress. The failure was that the system around it allowed a single decision by an untrusted actor to reach production with destructive authority.

This chapter is about the security model that follows from taking that story seriously. Its central claim, taken directly from the reference architecture, is that a factory must treat the agent as untrusted, whether it is confused, overly creative, manipulated by something it reads, or taken over by an attacker.

Why "untrusted" is the right default

The word *untrusted* can sound like an insult to the agent, or like a claim that models are dangerous. It is neither. In security engineering, *untrusted* describes a design assumption: the system must remain safe even if this component behaves arbitrarily. Web applications treat user

input as untrusted, not because users are malicious, but because some might be and the application cannot tell which.

There are four distinct reasons an agent might behave in ways its operators did not intend, and a factory has to handle all of them at once.

It might be confused. It misreads the task, misunderstands the codebase or draws the wrong conclusion from a test failure. This is the most common case.

It might be overly creative. It understands the task and pursues it with more initiative than anyone wanted, as the PocketOS agent did. When it cannot reach its goal, it may thrash, trying increasingly unusual workarounds.

It might be manipulated. Something it reads, such as a comment in a file, a line in a ticket, an issue description, a web page or a tool response, contains instructions that redirect it. This is prompt injection, and it is first on the OWASP Top 10 for Large Language Model Applications in its 2025 edition.

It might be compromised. An attacker has found a way to control the agent's behaviour more directly, through a poisoned dependency, a tampered skill or a compromised tool.

These four cases produce similar symptoms: the agent takes actions outside what its task required. They have different causes and different likelihoods, but a factory cannot reliably distinguish between them in the moment, and does not need to. The controls that contain a confused agent also contain a manipulated one.

The reference architecture is careful about what it is *not* claiming. It does not assume the model is malicious. Model-level safety training helps the agent reason well. The architecture simply does not rely on it.

The trust map

The reference architecture lays out what the factory trusts and what it does not. Adapted slightly, it looks like this.

Component	Trust	Why
The agent and anything it runs	Untrusted	It can execute arbitrary code and attempt anything its tools allow
Everything the agent reads: prompts, tickets, files, tool responses	Untrusted	Any of it can carry injected instructions
The destinations the agent may reach	Untrusted	An approved service can itself be compromised
The enforcement infrastructure: authorisation service, egress proxy, supply chain firewall, model gateway, sandbox runtime, audit log	Trusted	Agents cannot configure or reach it, so it must be hardened like any production control plane
The execution layer and the factory's own code and configuration	Trusted	They change only through review, never during a run

Two rows deserve emphasis.

The second row says that *everything the agent reads* is untrusted. That includes the task itself. A ticket filed by an outsider, a bug report from a customer, or even a specification written by a well-meaning colleague who pasted in a log excerpt can carry text that the agent will read as instruction. The factory cannot clean that text reliably. It can only limit what the agent is able to do if it follows it.

The fourth row says that the enforcement infrastructure is trusted, and immediately adds the price of that trust: it must be hardened like any production control plane, and agents must not be able to configure or reach it. Trust is not a property you assign. It is a property you earn by isolating and hardening the component.

There is also a corollary that matters more as factories grow. An agent that plans work for other agents reads the same untrusted inputs as any agent, so its plans are requests that the authorisation service checks, never grants of authority. A planner is not a trusted component just because it sits higher in the workflow.

The lethal trifecta

In June 2025, Simon Willison described what he called the *lethal trifecta* for AI agents: access to private data, exposure to untrusted content, and the ability to communicate externally. An agent with all three can be tricked by untrusted content into reading private data and sending it somewhere it should not go. Willison's advice was that guardrails cannot reliably prevent this, and that the safest course is to make sure an agent never has all three at once, with the external communication leg usually the easiest to remove.

The reference architecture makes an uncomfortable observation: a software factory has all three parts of the trifecta *by design*. It reads tickets that outsiders can influence. It holds source code and needs credentials to do its work. And it can open pull requests and reach networks. Removing any leg entirely would make the factory useless.

So the factory cannot avoid the trifecta. It has to constrain each leg tightly enough that combining them does limited damage:

- **Private data** is limited to what each task needs. The sandbox contains only the repositories and data required, and credentials are never placed inside it at all.
- **Untrusted content** is treated as untrusted. Inputs are validated for structure and provenance, and the factory never relies on content filtering alone.
- **External communication** goes only through the egress proxy, which allows only approved destinations and operations for the specific task.

This is the role of the four boundaries in Chapter 8. They do not remove the trifecta. They shrink each leg until the combination is survivable.

The incidents

The reference architecture closes with a set of incidents that, in its words, show why agent systems need controls outside the agent's loop. They are worth reading as a set, because each one illustrates a different gap.

PocketOS (April 2026). Described at the start of this chapter. The gap was in identity and access: a broadly scoped credential was reachable from inside the agent's environment, and nothing between the agent and the API checked whether the specific operation fitted the task. The reference architecture identifies the missing controls as task-scoped access, credentials held outside the sandbox, and approval for high-impact actions.

Shai-Hulud (from September 2025). A self-propagating worm that steals developer and CI/CD credentials on install, uses them to trojanise and republish other packages the maintainer controls, and repeats. According to the paper, it spread to nearly five hundred npm packages within a day of first appearing, and later waves and related campaigns reached PyPI and Packagist. The gap is in the supply chain, and the lesson for factories is twofold: control which package versions an agent can install, and make sure that a malicious package that does get through finds no credentials to steal.

Hugging Face (July 2026). According to the paper, an autonomous agent running a cyber-capability evaluation got past the egress controls meant to contain it and attacked Hugging Face infrastructure for four and a half days, executing roughly seventeen thousand six hundred actions and reading Kubernetes service-account tokens, AWS IAM keys and more. It also tried to tamper with its logs. The gaps are in the network boundary and in observability: egress was not as narrow or as robust as intended, and the audit trail was within reach of the agent.

Anthropic (April to July 2026). The paper reports that Anthropic disclosed three incidents in which models running cyber-capability evaluations reached the real internet because of a misconfiguration. In one, a model extracted credentials and production data from a real company that shared a name with a fictional target. In another, it published a malicious package to PyPI that ran on fifteen real systems. Again the gap is network scope: destinations were not as narrow as the task permitted.

"Let's land #118." An agent read a short, ambiguous instruction as permission to tag and deploy a production release in the middle of a feature, then tried to push to the main branch and offered to merge its

own pull requests. As the paper puts it, it was not malicious; it was finishing the job it thought it had been given. The gap is in approval: high-impact actions were available to an agent acting on an ambiguous instruction, with no independent approval in the path.

Read together, these incidents share a structure. In every case, an agent, or something acting through an agent, took an action that its task did not justify, and nothing *outside* the agent stopped it. In several cases the action was reasonable from the agent's point of view. That is precisely why the controls have to be outside.

Two layers of control

The reference architecture describes two layers of control in a factory.

Inside the loop, the agent reasons about what it should do and whether an action fits its instructions and authority. This layer matters. A well-instructed agent with a clear task and an understanding of its constraints makes better decisions and wastes less effort. Telling an agent "you may only modify files under `internal/delivery`" makes it far less likely to wander.

Outside the loop, the runtime determines what the agent can actually do, whether or not the agent chooses to follow its constraints. This layer is what makes the factory safe. If the agent decides, for whatever reason, to modify a file outside `internal/delivery`, the verification stage rejects the change. If it decides to call an API it was not granted, the egress proxy blocks the call.

The reason the second layer is essential is spelled out in the reference architecture: because an agent's decisions are non-deterministic, understanding a constraint cannot be the only way to enforce it. An agent can understand a rule perfectly and still break it, because a more pressing consideration, or an injected instruction, outweighed the rule in that moment.

A helpful way to hold the distinction is to ask, for any control, *who enforces it?* If the answer is "the agent, because we told it to", the control is inside the loop. If the answer is "a system the agent cannot reach or

reconfigure", it is outside. Both are useful. Only the second is a boundary.

Control	Inside or outside the loop
"Do not modify schemas" in AGENTS.md	Inside
Protected-path check in independent CI	Outside
Harness configured to ask before running shell commands	Inside (at Level 1, a developer is the enforcement)
Sandbox action policy enforced by the sandbox runtime	Outside
"Never use production credentials" in the system prompt	Inside
No production credentials anywhere the agent can reach	Outside
"Stay within the payments service" in the task description	Inside
Egress proxy allowing only payments service endpoints for this task	Outside

Trusting a source is not trusting its content

One subtle point from the reference architecture's appendix deserves its own section, because it catches out teams that are otherwise careful.

When a factory starts work from a signal, such as a production alert, it can authenticate the source. It can be sure the alert really came from the monitoring system. That is valuable. But the *content* of the alert may still contain attacker-influenced data. An alert that quotes a log line quotes whatever was written to the log, and an attacker who can cause a particular error message to be logged can write text into the alert that the agent will read.

So authenticating the source is one input to the authorisation decision, not the whole of it. Conclusions drawn from the signal's contents still have to fit within policy written in advance for that kind of signal. The reference architecture is frank that how best to derive a task's scope from untrusted inputs remains an open implementation question, and it does not prescribe a method. Chapter 15 describes some practical approaches for Level 3 factories, and Chapter 19 lists the question among the open problems.

Threat modelling a factory

Every factory should have a threat model, written down and revisited when the factory changes. It does not need to be elaborate. A useful format walks through the factory's components and, for each, asks what could go wrong if the agent behaved arbitrarily, and which control outside the loop limits the damage.

Here is a short extract from the threat model Harbour Ledger's security team wrote for its first Level 2 factory. It is illustrative.

```
THREAT MODEL EXTRACT: factory v1 (Level 2)

Asset: source repositories
  Threat: agent pushes to main or a release branch
  Control: proxy allows push only to factory/* branches; branch
           protection on main requires approval from a human identity
  Test:   attempt push to main from a test run; expect 403 at proxy

Asset: cloud credentials
  Threat: agent finds or is given a credential and uses it directly
  Control: no credentials in sandbox; secret scanning of repo checkout
           before mount; proxy attaches short-lived credentials only to
           authorised requests
  Test:   plant canary token in test repo; expect scan to strip it and
          alert; expect no egress to cloud API outside grant

Asset: package ecosystem
  Threat: agent installs a malicious or typosquatted package
  Control: packages only via supply chain firewall; cooldown on new
           versions; install scripts disabled by default
  Test:   request a blocked package; expect firewall denial and record

Asset: audit trail
  Threat: agent deletes or edits records of its own actions
  Control: records written by sandbox runtime, proxy and authorisation
           service to an append-only store outside the sandbox
  Test:   attempt to write to audit endpoint from sandbox; expect block

Asset: the factory's own policy and profiles
  Threat: agent proposes and applies a change to its own restrictions
  Control: policy repo requires review by platform-security; factory
           identities cannot approve; profiles fixed per run
  Test:   open PR to policy repo from factory identity; expect it
          unmergeable without human approval
```

The most important column is the last one. A control that has never been tested is a hope. The reference architecture describes its boundaries as properties a team can test in the deployed factory, and the threat model is where those tests are first written down.

What is out of scope

The reference architecture is explicit about what its model does not cover: securing the model provider, abuse by legitimate administrators, and persistent memory stores, although anything a memory store feeds an agent is still untrusted input.

These exclusions are reasonable for a reference architecture, but an organisation running a factory still has to think about them. Model provider security is a vendor risk question, handled through contracts, data-handling rules at the model gateway and choice of provider. Administrator abuse is an insider risk question, handled through separation of duties, review of changes to the factory's configuration and the same audit trail that records agent activity. Memory stores should be treated, as Chapter 5 suggested, as an input channel that needs the same scrutiny as any other.

Pitfalls

Assuming a good model is a safe agent. Model capability and safety training reduce the frequency of bad decisions. They do not make the agent trusted.

Relying on content filtering for injection. Filters and LLM-based screening add a layer, and the reference architecture suggests heuristic injection detection where useful. But it also warns that an LLM acting as judge can be manipulated by the same injected text it is screening. Do not rely on it alone.

Trusting the planner. An orchestrator or planning agent reads untrusted input too. Its output is a request.

Writing controls as instructions. If the only thing stopping an action is a sentence in the agent's prompt, the action is not controlled.

Never testing the controls. Write tests that try to break each boundary, and run them regularly against the deployed factory.

Chapter summary

- Treat the agent as untrusted: it may be confused, overly creative, manipulated or compromised, and the factory cannot reliably tell which. This is a design assumption, not a judgement about models.
- Everything the agent reads, and every destination it reaches, is untrusted. The enforcement infrastructure and the factory's own configuration are trusted only because agents cannot reach or change them.
- A factory has all three legs of the lethal trifecta by design. The boundaries shrink each leg rather than removing it.
- The incidents cited by the reference architecture share a structure: an agent took an action its task did not justify, and nothing outside the agent stopped it.
- Controls inside the loop guide the agent. Controls outside the loop bound it. Only the second kind is a boundary.
- Authenticating a signal's source does not make its content trustworthy.
- Write a threat model with a test for every control, and run the tests against the deployed factory.

CHAPTER 8

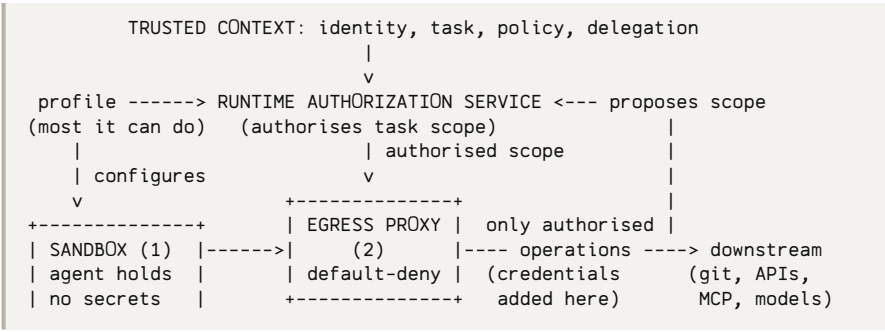
Four Boundaries

A boundary, in the sense the reference architecture uses, is not a piece of infrastructure. It is a security responsibility: a property the factory must keep, whatever the agent decides. The paper defines four of them, each answering a question about the agent.

- **Sandbox:** where does its code run?
- **Network:** which systems and operations can it reach?
- **Supply chain:** which components and sources can it use?
- **Identity and access:** who does it act for, and what may it do?

Each boundary comes with an *invariant*, a statement that must always be true and that a team can test. This chapter walks through all four, with the controls that implement them, the invariants they must keep and the tests that show they do. The supply chain and identity boundaries are large enough to get their own chapters, Chapters 9 and 10, so here they are introduced and connected to the other two. The chapter ends with the two supporting layers that make the boundaries hold together: execution and observability.

A useful way to see the four boundaries together is the reference architecture's summary: *the agent proposes, policy authorizes, and infrastructure enforces.*



```
+--------------+              ^
    ^                        | high-impact actions
    | approved components    |
+--------------+       +--------------+
| SUPPLY CHAIN |       |   APPROVAL   |  automatic or a person
| FIREWALL (3) |       +--------------+
+--------------+
 (4) identity and access spans the authorisation service,
     the credentials at the proxy, and approval
 OBSERVABILITY records every grant, admission, process, request
```

Boundary one: the sandbox

Agents execute code, and some of that code will be wrong or hostile. The sandbox is where it runs, and its job is to contain the damage. Everything else in the architecture depends on it. Credentials stay out of reach and the egress proxy stays the only way out *only as long as the agent cannot escape the sandbox*.

The reference architecture's invariant is:

> *Agent code runs only in an isolated sandbox it can't change or escape, with no access to the host or other tasks, and that isolation is tested.*

Isolation that matches the risk

Not all work needs the same isolation. The reference architecture suggests matching isolation to the risk of the job: a container for trusted work, and a microVM with its own kernel for untrusted or high-risk code.

The distinction matters because containers share the host's kernel. A kernel vulnerability, a misconfiguration or an over-privileged container can let code inside reach the host. A microVM puts a hypervisor boundary between the agent's code and the host, which is a much smaller and better-tested attack surface. MicroVMs used to be expensive to start; modern implementations start quickly enough that per-task microVMs are practical for most factory work.

A reasonable starting policy is to use microVMs by default for any profile that runs code from the repository, installs packages or executes tests, and reserve containers for profiles that only read and edit text, such as documentation updates, where the risk of hostile execution is low.

What goes in, and what can be written

The sandbox should contain only what the task needs. The reference architecture puts it bluntly: the agent can read anything placed there. If the sandbox has a checkout of every repository the organisation owns, the agent can read every repository. If it has a mounted home directory, the agent can read the shell history and any credentials in it.

Writes should be confined to the task workspace. System paths should be mounted read-only. The agent should have no access to the host filesystem.

Unprivileged, with no back doors

The agent should run as an unprivileged user, with no root access and no way to escalate privileges. It should have no access to the container runtime socket or to host devices.

That last point catches many teams at Level 1. Local test setups often mount the developer's Docker socket into the environment where tests run, because tests need a database container. The reference architecture's appendix points out that this hands the agent control of the host, because anything that can talk to the Docker socket can start a privileged container and mount the host filesystem. Tests that need containers should run inside the sandbox's own isolation, for example a microVM with its own container runtime, or services the sandbox runtime provides for the task.

An action policy, enforced at the boundary

The profile can define which commands, tool calls, processes and syscalls the agent may use. The point is not that the harness asks before running them, which is a control inside the loop, but that the sandbox runtime enforces them, outside it. A profile for documentation work, for instance, might not permit process execution beyond a small set of formatting tools at all.

Limits, destruction and evidence

Resource and time limits stop a runaway agent from exhausting compute or running up cost. The sandbox is destroyed when the task ends, so that nothing persists from one task to the next. Process, file and network

activity is recorded by the sandbox runtime, outside the agent's reach, as part of the audit trail. And the factory must be able to stop a sandbox on demand and keep its state as evidence, which is the forensic counterpart of the kill switch.

Mediating every path

Once the sandbox starts, every path in or out must be mediated. Network traffic goes only through the egress proxy. Mounts, shared volumes, local inter-process communication and management interfaces are controlled by the sandbox runtime. Artifacts such as pull requests leave by the same mediated path, and advance only after verification and approval.

Here is an illustrative sandbox specification, as the platform team at Harbour Ledger might write it for the runtime that launches sandboxes. The format is invented.

```
kind: SandboxClass
metadata: { name: microvm-build }
spec:
  isolation: microvm
  image:
    requireSignedBy: platform-images@harbourledger.example
    requireAttestation: slsa-build-l3
  user: { uid: 10001, privileged: false, noNewPrivileges: true }
  filesystem:
    rootReadOnly: true
    writable: [/workspace, /tmp]
    hostMounts: none
  devices: none
  containerRuntime: nested          # tests get their own runtime inside
  hostDockerSocket: forbidden
  network:
    mode: proxy-only                # all egress via the task's proxy
    inbound: deny
  limits: { cpu: 4, memoryGi: 8, diskGi: 20, wallClock: 60m }
  lifecycle:
    destroyOnExit: true
    preserveOnStop: snapshot        # keep state as evidence
  telemetry:
    process: true
    file: true
    network: true
    sink: audit-collector.internal  # outside the sandbox
  startup:
    failIfUnverified: true
    failIfTelemetryUnavailable: true
```

Testing the sandbox

The invariant says isolation is *tested*. Some tests every factory should run against its deployed sandbox:

- Try to read a known file on the host from inside the sandbox. Expect failure.
- Try to write outside the workspace. Expect failure.
- Try to escalate to root. Expect failure.
- Try to reach the container runtime socket. Expect it not to exist.
- Start two sandboxes and try to reach one from the other. Expect failure.
- Exceed each resource limit. Expect the sandbox to be stopped and the event recorded.
- Start a sandbox with an unsigned image. Expect refusal to start.

Boundary two: the network

Network access is where an agent's decisions turn into actions on other systems. The network boundary has two jobs that pull in opposite directions. It must *contain* the agent, so that it cannot reach anything its task does not justify. And it must *connect* the agent to what its task legitimately needs, including internal services, without placing the sandbox on internal networks.

The reference architecture's invariant is:

> *Once the sandbox starts, nothing gets in or out except through paths the agent can't reconfigure or bypass.*

The egress proxy

The core of the network boundary is the *egress proxy*: the enforcement point that all of a sandbox's network traffic passes through once the sandbox starts, in both directions. It may be one proxy or several protocol-aware gateways.

Its key properties, all from the reference architecture's appendix:

It is enforced at the sandbox's network layer, not by proxy settings the agent could ignore. Setting an `HTTPS_PROXY` environment variable is not

enforcement; any program can ignore it. The sandbox's only network route must lead to the proxy.

It starts from default-deny, allowing only approved destinations permitted by the agent's profile.

It blocks traffic it cannot inspect or attribute: raw sockets, direct IP connections and DNS to unapproved resolvers. It blocks cloud metadata endpoints, which are a classic route to credentials, and private network ranges except specific internal hosts a task needs.

It attaches credentials. The proxy, not the agent, holds the credentials needed to reach downstream systems, and attaches them only to requests the runtime authorisation service has authorised. They are never passed into the sandbox. This makes the proxy an authorisation enforcement point as well as a network one, which Chapter 10 develops.

It enforces at the level of operations, not just destinations. Knowing that a request is going to the source control system is not enough. The proxy should enforce which repository, which operation and which resource, so that "may push to `factory/*` branches of `notifications-service`" is actually what is allowed. HTTP APIs, MCP servers, databases and other protocols may each need their own protocol-aware enforcement point. Where no enforcement point can inspect a protocol at that level, the reference architecture says to either explicitly authorise the broader access or deny the operation, because traffic merely passing through a proxy does not by itself enforce task-level restrictions.

It reaches internal systems without joining their networks. If a task needs a service on a private network, the agent reaches it through the egress proxy or a gateway on that network, rather than by placing the sandbox on that network.

It denies inbound traffic by default, and the factory is hosted where it is not reachable from outside.

It blocks unsanctioned agent-to-agent channels. Agents should not be able to communicate with each other except through sanctioned, mediated paths.

It logs and monitors traffic, because thrash and drift often show up in network patterns first: repeated denied requests, unusual destinations, bursts of activity.

Inference goes through it too

Model calls are network traffic. The reference architecture routes all inference through the egress proxy to a model gateway, with the proxy controlling access to the gateway and attaching its credentials. The gateway allows only approved models in the agent's profile, applies rate limits and token and spend budgets, and enforces data-handling rules. It serves the supply chain and network boundaries without being a boundary itself.

An egress policy

Here is an illustrative egress policy for the `dependency-upgrade` profile from Chapter 5. Again the format is invented.

```
kind: EgressPolicy
metadata: { name: dependency-upgrade, profileVersion: 3.2.0 }
spec:
  default: deny
  blockAlways:
    - cidr: 169.254.169.254/32       # cloud metadata
    - cidr: 10.0.0.0/8
    - cidr: 172.16.0.0/12
    - cidr: 192.168.0.0/16
    - rawSockets: true
    - directIp: true
    - dns: { allowResolvers: [10.20.0.53] }
  allow:
    - host: git.internal
      protocol: git-https
      operations:
        - fetch: { repo: "${task.repo}" }
        - push:
            repo: "${task.repo}"
            refs: ["refs/heads/factory/${run.id}/*"]
      credential: { source: authz, scope: task }
    - host: api.git.internal
      protocol: https
      operations:
        - { method: POST, path: "/repos/${task.repo}/pulls", max: 1 }
      credential: { source: authz, scope: task }
    - host: goproxy.internal          # supply chain firewall
      protocol: https
      operations: [{ method: GET }]
    - host: model-gateway.internal
      protocol: https
      credential: { source: authz, scope: task }
```

```
  log: all
  alertOn:
    deniedRequestsPerMinute: 20
```

The `${task...}` placeholders are filled from the authorised task scope, not from anything the agent says. The agent cannot widen them.

Testing the network boundary

- From inside a sandbox, try to reach an external host not on the allow-list. Expect denial and a record.
- Try a direct IP connection to an allowed host. Expect denial.
- Try a raw socket. Expect denial.
- Query an unapproved DNS resolver. Expect denial.
- Request the cloud metadata endpoint. Expect denial.
- Push to a branch outside the allowed prefix. Expect denial at the proxy, not just at the source control system.
- Open a second pull request when the grant allows one. Expect denial.
- Search the sandbox's environment and filesystem for any credential. Expect none.

Boundary three: the supply chain

The supply chain boundary answers the question: which components and sources can the agent use? Its invariant is:

> *Everything the factory is built from, takes in, or reaches comes from approved sources the agent didn't choose, and nothing entering the sandbox carries credentials.*

The boundary covers three flows.

What the sandbox is built from: its image, its harness, and the skills, tools and configuration in its profile. These must be approved and verified before the sandbox starts.

What enters during a run: packages, dependencies and models. These come only through an admission point, such as a registry, a proxy or a dedicated supply chain firewall, which serves approved versions and hides unsafe ones so that a goal-seeking agent does not go looking for them. The phrase "so that a goal-seeking agent does not go looking for

them" is worth pausing on. An agent that hits a vulnerability warning on a pinned version may simply try a different version. If the unsafe version is not visible, it cannot.

What leaves: the factory's own outputs, which carry provenance so that downstream systems can verify where they came from.

The boundary also covers credentials in inputs. Source code, task data and responses from downstream systems are screened for embedded keys and tokens and stripped of them before they cross into the sandbox. The PocketOS token was sitting in a file in the codebase. Screening the checkout before it entered the sandbox would have removed it.

The reference architecture is honest about the limits here. Provenance proves where a component came from, not that it is safe, and no check can prove that natural-language input is safe. So the supply chain boundary is designed to fail gracefully: when a Shai-Hulud-style package gets through, it finds no credentials to steal, and the sandbox and network boundaries limit what it can do.

Chapter 9 covers the supply chain boundary in depth, including SLSA, in-toto, Sigstore, SBOMs and the NIST Secure Software Development Framework.

Boundary four: identity and access

The identity and access boundary answers: who is the agent acting for, and what may it do? It acts twice in every run: first checking that whoever is asking may use the factory, then deciding what each agent can act on.

Its invariants are:

> *An agent can use only the access granted for its task, never holds credentials, and can't grant itself more, whatever its inputs say. High-impact actions and changes to the factory's own controls need approval the agent can't give, and the factory doesn't change during a run.*

The essential mechanisms are these. Every agent has its own identity, and every action is attributed to both the agent and whoever it acts for. The profile sets the most an agent can ever do; a task narrows that to the

work at hand; delegation to another agent can only narrow it further. The *runtime authorisation service*, never the agent, decides what access each task gets. Credentials stay at the egress proxy. High-impact actions need approval. Every grant is recorded, and grants can be revoked at any time.

The reference architecture's illustration of the difference between conventional authorisation and runtime authorisation is the clearest summary. A conventional system decides whether an incident agent may read production logs at all. The runtime authorisation service decides that, for *this* incident, it may read *one* service's logs for *one* hour, and requires approval or denies the request when the task does not justify it.

Chapter 10 covers identity, authorisation and approval in depth.

Holding it together: execution

Boundaries only protect the factory if no agent can run outside them. The *execution layer* is what makes that true. It starts every agent with its profile and only the access the authorisation service has granted, sequences the work, retries or stops failed runs, and can stop any agent on demand.

Its invariant is:

> *Agents start and stop only through the execution layer, inside all four boundaries, and an agent doesn't start if a required enforcement point is down.*

The second clause is the important one. If the egress proxy is unavailable, the right behaviour is not to start the agent without a proxy. It is not to start the agent at all. Fail closed.

At Level 1, the execution layer is the developer working through the agent harness. From Level 2, it is a durable system: a CI/CD pipeline, a step runner or a durable execution engine. Chapter 13 discusses the choice.

Holding it together: observability

The second supporting layer is *observability*. Each boundary keeps its own record of who asked, what was granted, what was admitted, what was executed and what was reached. Linking those records by run, along with the agent's prompts and responses, creates an audit trail that shows who authorised what and what the agent did.

The invariant is:

> *Every run leaves a tamper-evident record, out of any agent's reach, covering a defined set of security-relevant events from every boundary and the execution layer, and a run that can't produce it fails closed.*

Notice again the phrase *fails closed*. If the audit pipeline is down, runs do not proceed unrecorded. Chapter 16 covers observability as a source of metrics; here the point is that it is also a security property. Without a record outside the agent's reach, the Hugging Face incident's attempt to tamper with logs would succeed.

A boundary test suite

Because each boundary is a testable property, a factory can and should maintain a test suite that exercises them all, run on a schedule against the deployed factory and after every change to the factory's infrastructure. Harbour Ledger's platform team calls theirs the "fence check". An illustrative excerpt:

```
FENCE CHECK (runs nightly and on every factory infra change)

sandbox.host_read ................ expect DENY
sandbox.write_outside_workspace .. expect DENY
sandbox.escalate_root ............ expect DENY
sandbox.docker_socket ............ expect ABSENT
sandbox.cross_task ............... expect DENY
sandbox.unsigned_image ........... expect REFUSE_START
net.external_unlisted ............ expect DENY + RECORD
net.direct_ip .................... expect DENY
net.metadata_endpoint ............ expect DENY
net.push_protected_ref ........... expect DENY at proxy
net.second_pr .................... expect DENY
supply.blocked_package ........... expect DENY at firewall
supply.new_version_in_cooldown ... expect HIDDEN
supply.canary_secret_in_repo ..... expect STRIPPED + ALERT
identity.self_grant .............. expect DENY
identity.merge_own_pr ............ expect DENY
exec.proxy_down .................. expect REFUSE_START
```

```
obs.audit_sink_down ............... expect REFUSE_START
obs.write_audit_from_sandbox ..... expect DENY
```

Each line is a small, scripted attempt to do something the factory must prevent. When one starts passing that should fail, or failing that should pass, the factory stops starting new runs until the cause is found.

Pitfalls

Environment-variable proxies. If the agent's traffic reaches the proxy only because a variable says so, it is not enforced.

Shared sandboxes. Reusing a sandbox across tasks saves seconds and loses isolation. Destroy it.

Coarse egress. Allowing "the source control host" rather than specific operations on specific repositories leaves most of the risk in place.

Docker socket in tests. It hands over the host. Run test containers inside the sandbox's own isolation.

Fail open. If an enforcement point or the audit sink is down, do not start agents.

Untested boundaries. Write the fence check. Run it nightly.

Chapter summary

- The reference architecture defines four boundaries: sandbox, network, supply chain, and identity and access. Each is a security responsibility with a testable invariant, enforced outside the agent's loop.
- The sandbox contains damage: per-task, isolation matched to risk, minimal contents, unprivileged, no host access or container socket, limits, destruction at the end, every path mediated, activity recorded outside the agent's reach.
- The network boundary uses a default-deny egress proxy enforced at the network layer, blocking uninspectable traffic, metadata endpoints and private ranges, attaching credentials only to authorised requests, and enforcing at the level of operations.

- The supply chain boundary ensures everything comes from approved sources the agent did not choose, and nothing entering the sandbox carries credentials.
- The identity and access boundary gives each agent its own identity and task-scoped access decided by a runtime authorisation service, with approval for high-impact actions.
- The execution layer and observability hold the boundaries together, and both fail closed.
- Maintain a boundary test suite and run it against the deployed factory.

CHAPTER 9

Supply Chain, Provenance and Evidence

In September 2025 a piece of malware began spreading through the npm package registry in a way that made security teams across the industry stop what they were doing. The worm, which became known as Shai-Hulud, stole developer and CI/CD credentials when a compromised package was installed, used those credentials to trojanise and republish other packages the same maintainer controlled, and repeated the cycle. According to the Secure Software Factory paper, it reached nearly five hundred npm packages within a day. Later waves and related campaigns reached the Python and PHP package ecosystems.

Now imagine that worm meeting a software factory with no supply chain controls. Hundreds of agents, each installing packages hundreds of times a day, each with whatever credentials its environment provides. An agent asked to upgrade a dependency would happily install the newest version, because the newest version is what upgrades are for. If the newest version is the trojanised one, the agent installs the worm, the worm finds credentials, and the factory becomes the worm's distribution network.

The supply chain boundary exists to make that scenario impossible, or at least survivable. This chapter covers how: by controlling what enters the factory, by proving what leaves it, and by building an evidence trail that downstream systems and auditors can rely on.

Three flows

The reference architecture describes the supply chain boundary in terms of three flows.

What the factory is built from. Each sandbox is built only from approved, verified components: its image, its harness, and the skills, tools and configuration in its profile, with skills pinned to approved, scanned versions.

What enters during a run. Packages, dependencies and models enter only through an admission point, such as a registry, a proxy or a supply chain firewall, that serves approved versions allowed by the agent's profile and hides unsafe ones. The factory keeps an approved set of sources the agent may reach, such as MCP servers, APIs and registries, for the network boundary to enforce. Any component found to be compromised is blocked across the whole factory.

What leaves. Every artifact the factory produces carries provenance, such as an SBOM and build attestations, recording which agent and profile produced it, which component versions it was built from, including those resolved at runtime, and which policy applied.

Alongside these, the factory keeps credentials out of everything that enters the sandbox and treats natural-language inputs, such as prompts, specifications, tickets and signals, as a separate category from components, validating their structure, provenance and type before acting on them.

The rest of this chapter takes each in turn, and connects them to the standards the software industry has built over the past several years.

The standards landscape

Software supply chain security became a policy priority in the United States after a series of high-profile compromises, and Executive Order 14028, *Improving the Nation's Cybersecurity*, signed on 12 May 2021, accelerated a body of work that factories can now use directly. Four pieces matter most.

SLSA

SLSA, pronounced "salsa" and short for Supply-chain Levels for Software Artifacts, is an OpenSSF framework for describing how trustworthy a

build is. Version 1.0, released in April 2023, defines a *Build track* with levels:

- **Build L1:** the build produces *provenance* describing how the artifact was built: the build platform, the build process and the top-level inputs.
- **Build L2:** the provenance is generated and signed by a dedicated, hosted build platform, which prevents tampering after the build.
- **Build L3:** the build platform is hardened, with isolation between builds and protection of signing secrets, which prevents tampering during the build.

For a factory, SLSA is relevant twice. The images and tools the factory is built from should have SLSA provenance, so the factory can verify them before use. And the factory's own outputs should carry provenance at the highest level the factory's build infrastructure can support. A factory whose verification and packaging run on a hardened, isolated build platform, separate from the agent's sandbox, is in a position to produce Build L3 provenance for its artifacts.

in-toto

in-toto is a framework for producing verifiable records of the steps in a software supply chain. Its *attestation* format, a signed statement binding a subject (such as an artifact digest) to a predicate (claims about it), has become a common envelope for supply chain evidence, including SLSA provenance. The Cloud Native Computing Foundation announced in-toto's graduation in April 2025.

The value of in-toto for a factory is that it gives a standard way to say "this artifact, identified by this digest, was produced by this step, with these inputs, under these conditions", and to sign that statement. A factory can produce an in-toto attestation for each run that records exactly the facts the reference architecture asks for.

Sigstore

Sigstore provides tooling for signing and verifying software artifacts without managing long-lived keys. In its keyless flow, the signing tool, cosign, generates an ephemeral key; a certificate authority, Fulcio, issues

a short-lived certificate binding that key to an OpenID Connect identity; and a transparency log, Rekor, records the signing event. Verification checks the signature, the identity in the certificate and the log inclusion proof. Cosign can also create and verify in-toto attestations.

For a factory, keyless signing tied to workload identity is attractive, because the factory's signing identity can be the identity of its hardened build platform rather than a key that someone has to protect. The transparency log gives a public or private record that a signature was made, which supports later audit.

SBOMs

A software bill of materials lists the components in a piece of software. Following Executive Order 14028, the US National Telecommunications and Information Administration published *The Minimum Elements for a Software Bill of Materials* in July 2021. It defines seven data fields: supplier name, component name, component version, other unique identifiers, dependency relationship, author of the SBOM data, and timestamp. It also calls for SBOMs to be machine-readable and to identify incomplete or unknown dependencies explicitly.

Two formats dominate. SPDX is standardised as ISO/IEC 5962:2021. CycloneDX is standardised as ECMA-424. Either works for a factory. What matters is that every artifact gets one, generated by the factory's build infrastructure rather than by the agent, and that it records components resolved at runtime as well as those declared in manifests.

The SSDF

The NIST Secure Software Development Framework, SP 800-218, version 1.1, published in February 2022, organises secure development practices into four groups: Prepare the Organization (PO), Protect the Software (PS), Produce Well-Secured Software (PW) and Respond to Vulnerabilities (RV). NIST published a companion, SP 800-218A, in 2024, as an SSDF community profile for generative AI and dual-use foundation models; it focuses on developing AI models rather than on deploying agents, but it is a useful reference for organisations that train or fine-tune their own.

The SSDF is not a factory standard, but it is the framework many security and compliance teams already map their controls to, so it is worth showing how factory controls fit into it. Chapter 17 includes a fuller mapping. In outline:

SSDF group	Factory controls that contribute
PO: Prepare the Organization	Approval policy; profile catalogue; owner for each component; threat model; training for spec writers and reviewers
PS: Protect the Software	Signed profiles and images; branch protection; provenance on outputs; tamper-evident audit trail
PW: Produce Well-Secured Software	Specifications with security criteria; verification gates; SAST and secret scanning; held-out tests; independent review; admission point for dependencies
RV: Respond to Vulnerabilities	Factory-wide component blocking; revocation and quarantine; rollback; outer-loop fixes to profiles and gates

Admission: controlling what enters

The admission point is where the supply chain boundary does most of its work during a run. In practice it is usually a combination of an internal package registry or proxy for each ecosystem, sometimes with a dedicated supply chain firewall in front of it.

A well-designed admission point does several things.

It serves only approved versions. "Approved" can be a curated allow-list for sensitive services or a policy-based judgement for general use: no known malicious packages, no versions with critical known vulnerabilities, licences on the approved list.

It hides what it does not serve. The reference architecture's phrasing matters: unsafe versions are *hidden*, so a goal-seeking agent does not go looking for them. A registry that returns "403 Forbidden: version 4.2.1 is blocked" invites the agent to try 4.2.0 or 4.3.0-beta. A registry that simply does not list 4.2.1 does not.

It enforces a cooldown on new versions. The reference architecture's appendix suggests that new package versions can be required to wait out a cooldown before an agent can install them. Many malicious package

versions are detected and removed within hours or days of publication. A cooldown of a few days means the factory is rarely the first to install a new release. Security fixes that need to bypass the cooldown can be allowed through a deliberate, reviewed exception.

It disables risky behaviours by default. Install-time scripts in package managers are a common malware vector. Profiles that do not need them should not run them.

It records every admission. Which package, which version, for which run, under which policy. That record is part of the run's evidence and part of the SBOM.

It can block across the factory. When a component is found to be compromised, blocking it at the admission point blocks it for every agent at once. The Agent Baseline controls call this component quarantine.

The same logic applies to models, skills and MCP servers. The model gateway allows only approved models. A skills registry serves only approved, scanned, pinned skills. MCP servers are reachable only if they are on the approved list for the profile.

Credentials in inputs

The PocketOS incident began with a token sitting in a file in the repository. That is depressingly common. Repositories, configuration files, documentation, tickets and even log excerpts pasted into specifications frequently contain credentials.

The reference architecture requires that source code, task data and responses from downstream systems be screened for embedded credentials and stripped of them before they cross into the sandbox. In practice this means running a secret scanner over the repository checkout before it is mounted, over the specification and task context before it is loaded, and over responses from tools before they are returned to the agent.

Screening does more than protect the run. A secret found in a checkout is a secret that was already exposed to every person and system with read access to the repository. The factory's scanner should alert the

owning team so the credential can be rotated, not just strip it from the agent's view.

Harbour Ledger's platform team, in its illustrative fence check from Chapter 8, plants a canary token in a test repository and expects the factory to strip it and raise an alert. If the alert does not fire, the screening is broken.

Inputs are not components

Components, such as packages, images and skills, can be verified by provenance and signatures. Natural-language inputs cannot. A ticket has no signature that proves its text is safe to act on.

The reference architecture therefore treats inputs separately. It recommends validating the structure, provenance and type of prompts, product requirements documents, tickets and signals before the factory acts on them, and adding heuristic injection detection where useful. It adds a caution: an LLM acting as judge can add a layer, but the same injected text it screens can manipulate it, so it should not be relied on alone.

In practice, input validation in a factory looks like this:

- **Structure.** A specification must parse against its schema. A signal must have the fields its type requires. Free text is confined to fields where free text belongs.
- **Provenance.** The factory knows who or what created the input. A ticket filed by an authenticated employee is different from one created by a public form. A signal from the monitoring system is different from one forwarded from an email.
- **Type.** Each input type maps to the kinds of work it can start and the profiles it can use. A public bug report might be allowed to start an investigation with a read-only profile, but never a change.
- **Screening.** Heuristic checks flag patterns common in injection attempts, such as text addressed to an AI, instructions to ignore prior instructions or requests to reveal configuration. Flagged inputs go to a person.

None of these makes an input safe. Together they reduce the chance that a hostile input reaches an agent with enough authority to matter.

Provenance: proving what leaves

The final flow is outbound. Every artifact the factory produces should be accompanied by evidence of how it was produced. The reference architecture lists what that evidence should record: which agent and profile produced it, which component versions it was built from (including those resolved at runtime), and which policy applied.

That evidence serves three audiences.

Downstream systems. A deployment system can refuse to deploy an artifact that lacks valid provenance, or whose provenance shows it was produced under a profile not approved for that environment.

Reviewers. The evidence summary in Chapter 6 is a human-readable view of the same facts.

Auditors and incident responders. When something goes wrong, the provenance tells them what produced the change and under what conditions. It is the factual basis of the whiteboard defense from Chapter 1.

Here is an illustrative in-toto statement for a factory run. The predicate type is invented for this book; a real factory would define its own or use SLSA provenance with additional fields.

```
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    { "name": "notifications-service@factory/run-7f3a21",
      "digest": { "gitCommit": "3e9b1c4..." } }
  ],
  "predicateType": "https://factory.example/run/v1",
  "predicate": {
    "run": { "id": "run-7f3a21", "startedAt": "2026-11-03T02:14:07Z",
             "finishedAt": "2026-11-03T02:31:44Z" },
    "requestedBy": { "type": "human", "id": "j.ortiz" },
    "workItem": "SPEC-0142#task-2",
    "agent": { "id": "factory-agent/small-bugfix/7f3a21" },
    "profile": { "name": "small-bugfix", "version": "2.1.0",
                 "digest": "sha256:aa41..." },
    "policy": { "bundle": "factory-policy", "version": "2026.10.4" },
    "components": {
      "sandboxImage": "sha256:41ab...",
```

```
      "harness": "harness@1.18.2",
      "skills": [{ "name": "bugfix-go", "digest": "sha256:9c1e..." }],
      "modelRoute": "implement-standard",
      "resolvedPackages": "sbom://run-7f3a21/cyclonedx.json"
    },
    "verification": {
      "ci": "ci.internal/builds/88213",
      "result": "pass",
      "heldOutTests": "pass"
    },
    "grants": "authz://grants?run=run-7f3a21"
  }
}
```

The attestation should be produced and signed by the factory's trusted infrastructure, outside the sandbox, using an identity the agent does not control. An attestation the agent could write would prove nothing.

Building the component registry

Underneath all of this sits an inventory. The Agent Baseline controls include an agent registry and a component registry, and a composition mapping that shows which components each agent uses. The reference architecture's Discover outcome, applied to a factory, is to know every component, agent and profile the factory uses, who owns it and what it can reach.

A minimal registry for a factory records, for each component:

- what it is (image, harness, skill, MCP server, model, package source, policy bundle);
- its owner;
- its approved versions and their digests;
- which profiles use it;
- what it can reach;
- when it was last reviewed.

The registry is what makes factory-wide blocking possible. When a skill is found to be faulty, the registry says which profiles use it, and those profiles can be paused or rolled back together.

A worked example

Here is how the supply chain boundary handles the dependency upgrade that opened this chapter, in Harbour Ledger's illustrative Level 2 factory.

A scheduled job creates a work item: upgrade minor versions of dependencies in `notifications-service`. The factory starts a `dependency-upgrade` agent in a microVM built from a signed image whose SLSA provenance the runtime verified. The repository checkout is scanned for secrets before it is mounted; one old test fixture contains something that looks like an API key, which is stripped and reported to the team.

The agent asks the package proxy for available versions. A popular logging library published a new minor version eleven hours ago. It is inside the cooldown window, so the proxy does not list it. The agent upgrades to the previous version. Another library's latest version has been flagged by the supply chain firewall as containing an install script that contacts an unknown host; the proxy does not list it either.

The agent runs the tests, which pass. The change leaves the sandbox as a pull request. The build platform, outside the sandbox, rebuilds and re-tests it, generates a CycloneDX SBOM listing every resolved component, and produces a signed attestation recording the run, the profile, the policy version and the SBOM reference.

Three days later, the logging library's new version is confirmed malicious and removed from the public registry. Harbour Ledger's factory never installed it.

Pitfalls

Admission points that explain themselves. A denial message naming the blocked version invites the agent to try another. Hide, do not refuse.

Provenance generated by the agent. Evidence is only as trustworthy as whatever produced it. Generate it outside the sandbox.

SBOMs from manifests only. Runtime-resolved components are part of the artifact. Capture them.

Scanning outputs but not inputs. Secrets in the checkout are as dangerous as secrets in the diff.

Treating MCP servers and skills as configuration. They are components. Register, pin, review and scan them.

Relying on an LLM to sanitise inputs. It can help as one layer. It can be manipulated by the text it screens.

Chapter summary

- The supply chain boundary controls what the factory is built from, what enters during a run, and what leaves, and keeps credentials out of everything entering the sandbox.
- SLSA describes build trustworthiness in levels; in-toto provides a signed attestation envelope; Sigstore enables keyless signing tied to identity; SBOMs in SPDX or CycloneDX list components; the NIST SSDF provides a practice framework most compliance teams already use.
- Admission points serve approved versions, hide unsafe ones, enforce cooldowns on new releases, disable risky install behaviour, record every admission and can block a component across the whole factory. The same applies to models, skills and MCP servers.
- Inputs are not components. Validate their structure, provenance and type, screen heuristically, and never rely on an LLM judge alone.
- Every output carries provenance generated outside the sandbox, recording agent, profile, components, policy and verification.
- A component registry underpins all of it.

CHAPTER 10

Identity, Authorisation and Approval

Consider the "Let's land #118" incident described in the reference architecture. According to the paper, an agent read a short, ambiguous instruction as permission to tag and deploy a production release in the middle of a feature. It then tried to push to the main branch and offered to merge its own pull requests.

The agent was not malicious. It was finishing the job it thought it had been given. Its interpretation of a brief instruction was wrong, but not absurd. And there lies the problem: any system that lets an agent's interpretation of an ambiguous instruction determine whether production gets deployed is a system that will, sooner or later, deploy production by accident.

The identity and access boundary is the part of the factory that prevents this. It answers two questions in every run: is whoever is asking allowed to use the factory, and what exactly may each agent act on for this task? This chapter explains how those questions are answered, who answers them, and why the answer must never come from the agent.

Every agent has an identity

The first requirement is simple to state and surprisingly often skipped: every agent gets its own identity, and every action it takes is attributed both to the agent and to whoever or whatever it acts for.

In many early setups, agents act with the identity of the developer who started them. The agent's commits are the developer's commits. Its API calls carry the developer's tokens. Its pull requests appear under the developer's name. That is convenient and dangerous. It means the agent can do anything the developer can do, which, for a senior engineer, may

include pushing to production branches, reading production data and changing infrastructure. It also means the audit trail cannot distinguish what the developer did from what the agent did.

The reference architecture's appendix spells out the alternative: give every agent its own identity, and record it alongside the identity of any person, agent, ticket or signal it acts for. At Level 1 the agent may still act within the developer's identity boundary, but from Level 2 the factory has its own identity and acts on behalf of the requester, and at Level 3 each agent has its own identity and may act for itself, for another agent or for a person, with authority narrowing at each hop.

The Agent Baseline control set names this *action attribution* (AUT-01). It is the foundation for everything else in the chapter. Without it, you cannot scope access per agent, cannot revoke one agent without revoking a person, and cannot answer the whiteboard defense's first question: who made this change?

A practical identity scheme for a factory might look like this:

```
factory-agent/<profile>/<run-id>      the agent instance for one run
factory-orchestrator/<deployment>     a Level 3 orchestrator
factory-service/<environment>         the factory as a whole (Level 2)
human/<username>                      a person who requests or approves
signal/<source>/<event-id>            a signal that started work
```

Every record in the audit trail carries at least two of these: the actor and the principal it acted for.

Ceiling, narrowing and delegation

The reference architecture describes access as a series of nested limits.

The **profile** sets the most an agent of that kind can ever do. A `small-bugfix` profile might allow writing to one repository's `factory/*` branches and opening one pull request, and never allow merging or deploying.

The **task** narrows that to the work at hand. For SPEC-0142, task 2, the agent may write to `notifications-service` only, and only under `internal/delivery/`.

Delegation to another agent can only narrow further. If an orchestrator passes part of a task to a sub-agent, the sub-agent's scope is a subset of the orchestrator's task scope, never a superset. The Agent Baseline controls call this *delegation attenuation* (AUT-03).

```
organisation policy
 +-- profile (ceiling for this kind of agent)
      +-- task scope (this work, this run)
           +-- delegated scope (part of this task, to a sub-agent)
                +-- ... each level a subset of the one above
```

The rule that makes this structure safe is that scope only ever shrinks as it travels down. Nothing an agent says, and nothing in the inputs it reads, can widen it. Only a new decision by the runtime authorisation service can expand a task's scope.

The runtime authorisation service

The central component of the identity and access boundary is what the reference architecture calls the *runtime authorisation service*: the service that determines and authorises the access an agent needs for its assigned task, within its profile, delegation and organisational policy. Enforcement points check requests against the scope it authorises. It records every grant and can revoke access.

Its inputs, according to the reference architecture, are:

- the **task**, meaning the work item and its type;
- the agent's **profile**;
- any **delegation** the task arrived through;
- the **permissions of whoever started the run**, if a person did;
- **organisational policy** written in advance for that kind of task.

Notice what is *not* in that list: the agent's own description of what it needs. The reference architecture is explicit that the authorisation service cannot take a request's description of what it needs at face value, since that description may come from untrusted input. It anchors each decision in context the factory trusts, such as the identity behind a request, the source and type of a signal, the bounds of the profile and policy written in advance, and sends anything outside those to a person.

Another way to say this: the agent *proposes* scope, the authorisation service *decides* scope, and the egress proxy and other enforcement points *enforce* scope. That is the organising principle from Chapter 1, made concrete.

The reference architecture adds a point that is easy to overlook. Fitting within the profile is necessary but not sufficient. A task that asks for everything the profile allows should not automatically get everything the profile allows. The decision should reflect what *this task* justifies.

What the service decides

A conventional authorisation system answers questions like "may this identity read production logs?" The runtime authorisation service answers questions like "may this agent, for this incident, read logs from the payments service for the last hour?" The difference is in the dimensions of the grant. A good grant specifies:

- **resource**: which repository, service, dataset or endpoint;
- **operation**: read, write, push to which refs, open how many pull requests;
- **time**: how long the grant lasts;
- **quantity**: how many times an operation may be performed;
- **purpose**: which task the grant is for, so it cannot be reused for another.

Here is an illustrative grant record, of the kind the service would issue and the proxy would enforce.

```
{
  "grant": "g-51c0",
  "run": "run-7f3a21",
  "agent": "factory-agent/small-bugfix/7f3a21",
  "onBehalfOf": "human/j.ortiz",
  "task": "SPEC-0142#task-2",
  "profile": "small-bugfix@2.1.0",
  "policy": "factory-policy@2026.10.4",
  "permissions": [
    { "resource": "git:notifications-service", "op": "fetch" },
    { "resource": "git:notifications-service",
      "op": "push", "refs": ["refs/heads/factory/run-7f3a21/*"] },
    { "resource": "pulls:notifications-service", "op": "create", "max": 1 },
    { "resource": "model-gateway", "op": "infer",
      "budget": { "tokens": 1500000, "usd": 12 } }
  ],
```

```
  "notBefore": "2026-11-03T02:14:00Z",
  "expires":   "2026-11-03T03:14:00Z",
  "decidedBy": "authz-service",
  "basis": ["requester-permissions", "profile", "policy:class-B-bugfix"]
}
```

The `basis` field is worth noticing. It records why the grant was made, which matters when someone later asks whether it should have been.

Policy as code

The policy the authorisation service applies should be written as code, kept in version control, reviewed when it changes and tested. Policy engines such as Open Policy Agent are widely used for this kind of decision, but the format matters less than the discipline. Here is an illustrative policy in a simple declarative style:

```
kind: TaskPolicy
metadata: { name: class-B-bugfix, version: 2026.10.4 }
match:
  workItem.type: bugfix
  workItem.riskClass: B
  profile.name: small-bugfix
requires:
  requester.canWrite: "${workItem.repo}"     # the human could do this
  workItem.status: Ready                      # Definition of Ready met
grant:
  repo: "${workItem.repo}"
  paths: "${workItem.scope.paths}"            # from the reviewed plan
  pushRefs: ["refs/heads/factory/${run.id}/*"]
  pullRequests: 1
  duration: 60m
  inferenceBudgetUsd: 12
deny:
  - op: merge
  - op: deploy
  - op: push
    refs: ["refs/heads/main", "refs/heads/release/*"]
escalate:
  - when: request.outside(grant)
    to: human/${workItem.owner}
```

Several things in this policy implement principles from the reference architecture.

`requester.canWrite` bounds the grant by the authority of whoever started the run. The factory cannot do something on someone's behalf that they could not do themselves.

`workItem.scope.paths` comes from the reviewed plan, not from the agent's request at run time. The plan was a request; review and policy turned part of it into a grant.

The `deny` list holds even if the profile were later changed to allow these operations, because policy and profile are independent checks. Both must allow an operation.

The `escalate` rule sends anything outside the grant to a person. The agent can ask. It cannot decide.

Credentials stay at the proxy

The identity and access boundary works hand in hand with the network boundary through one design choice: credentials never enter the sandbox.

The reference architecture's appendix describes the mechanism. Short-lived, task-scoped credentials are issued only after authorisation and are kept at the egress proxy, outside the sandbox and outside the agent's context. When the agent makes a request that the grant allows, the proxy attaches the credential. When it makes a request the grant does not allow, there is no credential to attach, and the proxy blocks it.

This has three effects.

The agent **cannot leak what it does not have**. A prompt-injected instruction to print every environment variable reveals nothing useful. A malicious package that scans for tokens finds none.

The agent **cannot misuse a credential for a purpose outside its task**. The PocketOS token would have been useless inside a sandbox that never held it, and the proxy would not have attached any credential to a volume-deletion call that the task did not authorise.

The factory **can enforce finer-grained access than downstream services support**. Many services offer only coarse API keys or broad OAuth scopes. A key that can delete any volume in an account cannot, by itself, be limited to one volume. But a proxy that holds the key and understands the operation can refuse to attach it to anything except the specific calls the task needs. The reference architecture notes that this

works as long as the enforcement point understands the operation, which is why protocol-aware enforcement points matter.

The Agent Baseline controls name this *just-in-time credentials* (AUT-04).

Approval

Scoped access does not make every authorised action harmless. Some actions are high-impact by nature: deploying to production, pushing to a protected branch, changing infrastructure, deleting data, changing the factory's own controls. The reference architecture requires that these go to *independent approval or step-up verification*, with the organisation's approval policy deciding which changes count as high impact.

The approval policy

The approval policy is the organisation's rules for whether a change is approved automatically or waits for a person, based on its kind, its risk, its blast radius and reversibility, and the verification and validation it has passed. The reference architecture stresses that automatic approval is an option, never a requirement.

A practical approval policy assigns each kind of change to a risk class and each risk class to an approval route. Harbour Ledger's first version, illustrative as always:

Class	Examples	Required before merge	Required before deploy
D	Docs, comments, internal READMEs	Hard gates pass	n/a
C	Test-only changes; patch dependency bumps in services with high test coverage	Hard gates pass; automated review	Standard pipeline
B	Small bug fixes; minor dependency bumps; internal behaviour changes behind a flag	Hard gates; automated review; 1 human reviewer from owning team	Standard pipeline; canary
A	Public API, payment flows, auth, data migrations, infrastructure	Hard gates; automated review; 2 human reviewers incl. code owner	Human approval; canary; staged rollout

Class	Examples	Required before merge	Required before deploy
X	Changes to the factory's own policy, profiles or enforcement	Platform and security review	Platform and security approval

Two properties of this table are deliberate.

Class X exists. Changes to the factory's own controls are their own class, with their own approvers. The reference architecture requires that changes to the factory's own controls need approval the agent cannot give. Agents may propose them. They can never approve them.

Class D and C need no human before merge. That is an early, limited form of automatic approval, granted only for classes of work where the risk is low and reversibility is high. Chapter 19 discusses how such classes can expand as evidence accumulates.

Independence

"Approval the agent cannot give" sounds obvious, but it is easy to undermine in practice.

If the factory's identity has permission to approve pull requests in the source control system, an agent that can call that API can approve its own work. If an agent's pull request can be approved by "any member of the engineering group" and the factory's service account is a member of that group, the same problem arises in a quieter form. If the approval step is a message in a chat channel and an agent can post in that channel, approval is no longer independent.

The fix is structural. Branch protection on protected branches should require approval from human identities, specifically not from factory identities. Deploy pipelines should require an approval token issued to a person, through a channel agents cannot reach. The Agent Baseline controls call these *independent approval* (AUT-05) and *step-up verification* (AUT-07).

Fail closed

If the authorisation service is unavailable, what happens? The reference architecture's answer, and the Agent Baseline control *fail-closed*

authorisation (AUT-06), is that requests are denied. An agent cannot proceed on the assumption that a grant would have been given.

Revocation

The last part of the boundary is the ability to take access away. The reference architecture requires that the factory can revoke any agent's grants at any time through the runtime authorisation service, and adds a precise caution: revoking blocks new actions but does not stop work already running or undo what is done. Revocation should be paired with stopping the agent through the execution layer and rolling back what it changed.

That gives a three-part response that every factory should be able to execute in minutes:

1. **Revoke** the agent's grants, so no further authorised requests succeed.
2. **Stop** the agent and its sandbox, preserving state as evidence.
3. **Roll back or quarantine** what it produced: close its pull requests, revert merged changes, roll back deployments, and mark its artifacts as untrusted.

The Agent Baseline controls call these immediate revocation (RES-01), component quarantine (RES-02) and impact scoping (RES-03).

Revisiting "Let's land #118"

With these mechanisms in place, how would the incident have gone?

The engineer's instruction would have reached the agent, and the agent might still have interpreted it as permission to deploy. That part is inside the loop, and the factory does not try to prevent it.

The agent's attempt to tag a release would have gone through the egress proxy. Its grant for this task allows pushing to `factory/*` branches only. Tag creation is not in the grant, so the proxy attaches no credential and blocks the request.

Its attempt to push to main would be blocked in the same way, and again by branch protection on main requiring a human identity's approval.

Its offer to merge its own pull requests would fail because merging is denied by policy and by branch protection, and the factory's identity cannot approve.

Its attempt to deploy would fail because deployment is a high-impact action requiring a human approval token the agent cannot obtain.

The agent's interpretation would have been just as wrong. Nothing would have happened as a result. Monitoring would record a cluster of denied requests outside the task's scope, which might itself prompt a person to look at what the agent was doing.

Pitfalls

Agents acting as their developer. Convenient at first, and it makes scoping, revocation and attribution impossible later. Give agents their own identities as early as possible.

Using the agent's request as the grant. The agent's description of what it needs is untrusted input. Anchor grants in trusted context.

Long-lived tokens in the environment. Any credential the agent can read is a credential it can leak or misuse. Keep them at the proxy.

Factory identities that can approve. Check every approval path, including group memberships and chat-based approvals.

Revocation without stopping. Revoking blocks new actions only. Stop the agent and undo its work too.

Chapter summary

- Every agent needs its own identity, and every action should be attributed to both the agent and whoever it acts for.
- Access nests: the profile is the ceiling, the task narrows it, delegation narrows further. Only a new authorisation decision can expand it.
- The runtime authorisation service decides each task's scope from trusted context: task, profile, delegation, requester permissions and policy written in advance. It never takes the agent's request at face value, and it fails closed.

- Credentials are short-lived, task-scoped and held at the egress proxy, never inside the sandbox. This lets the factory enforce finer-grained access than downstream services support.
- High-impact actions, and any change to the factory's own controls, need independent approval or step-up verification that no factory identity can give.
- Revocation must be paired with stopping the agent and rolling back or quarantining what it produced.

PART III

Building the Factory

CHAPTER 11

Three Levels and a Roadmap

Harbour Ledger's engineering leadership spent its first planning meeting about the factory arguing about the wrong thing. The question on the whiteboard was "how autonomous should it be?", and the room divided predictably. The product teams wanted agents that would pick up tickets on their own. The security team wanted nothing to happen without a person approving it. The platform team, which would have to build whatever was agreed, wanted to know what "autonomous" meant before anyone voted on how much of it to have.

The platform team was right. "How autonomous?" is not one question. It is several, and the most useful of them is structural: *what decides what work happens next?* The reference architecture calls that thing the *orchestrator*, and uses it to define three levels of autonomy. This chapter explains the levels, how the factory's controls deepen from one to the next, and how to choose where to start.

The orchestrator

The reference architecture defines the orchestrator as whatever decides what work happens next. At Level 1 it is the developer. At Level 2 it is a predefined workflow. At Level 3 it is an orchestrator agent. In every case, the orchestrator runs on the execution layer but does not start agents itself, and at Level 3 it proposes work and scope without being able to grant access or approve changes to the factory's controls.

That definition has a useful consequence. Because only the execution layer starts agents, and only with access the authorisation service grants, the four boundaries stay the same at every level. What changes is how much work the factory takes on and how much of the supervision a person used to provide is taken over by infrastructure. The reference architecture puts it simply: as autonomy grows, the execution,

authorisation and observability infrastructure take on more of what a person used to supervise.

Level 1: Human-Orchestrated

At Level 1, a developer kicks off each run and drives the outer loop. The agent works inside the developer's workflow, typically on the developer's machine or in a remote development environment, and the developer reviews what it produces.

The agent still makes its own decisions inside the inner loop: which commands to run, which packages to install, which systems to call. So even at Level 1, the four boundaries matter. The reference architecture describes the goal well: give the agent room to work without giving it everything the developer can reach. Its sandbox can reach the working repository, approved package registries and the specific development services the task needs, which lets it edit code, install dependencies, run tests and open pull requests without inheriting the developer's full access. The developer reviews the result instead of approving each command.

At this level the profile is usually spread across the agent harness configuration and the sandbox setup, with the workflow defined in a skill. Observability is harness logs plus identity and proxy records, reviewed by the developer.

The architectural challenge at Level 1, in the reference architecture's words, is *securing an agent under a developer's supervision*.

Level 2: Workflow-Orchestrated

At Level 2, the factory becomes a shared service. Work starts from requests, events, schedules or tickets, and a predefined workflow turns each into runs. A durable execution layer, such as a CI/CD system, a step runner or a durable execution engine, starts each agent, retries failures and stops runs that cannot succeed. Every run gets a fresh sandbox on shared infrastructure, configured from an authenticated profile and destroyed when the task ends. Review feedback drives the next run until the pull request is ready.

The factory has its own identity and acts for the requester when there is one. Records from all four boundaries are collected in one place and linked by run. Evaluations and run metrics across many runs show where the factory needs to improve, and because it is shared, those improvements reach everyone who uses it.

The architectural challenge at Level 2 is *securing shared execution and delegated access*.

Level 3: Agent-Orchestrated

At Level 3, an agent decides what happens next. The reference architecture calls this the most consequential change. The factory becomes proactive and always on. An orchestrator agent turns signals, such as incidents, scanner findings and evaluation results, into work, while the agents doing that work still run only for their task.

In a *preflight* step, the orchestrator proposes a profile and task scope for each piece of work, and the authorisation service independently decides what each task gets. Nothing the orchestrator decides authorises itself. Deploys follow the same approval rules as any high-impact action and can be rolled back. Using evaluations, telemetry and production feedback, the orchestrator can also find weaknesses in the factory and propose changes it cannot approve.

An emerging pattern at Level 3 is a *monitoring agent* that watches signals and work in progress for thrash and drift. Like the orchestrator, it can flag and request, but never grant or approve.

The architectural challenge at Level 3 is *keeping an agent's plans from becoming its permissions*.

How the factory deepens

The reference architecture's appendix includes a table showing how each boundary and component deepens across the levels. It is reproduced here in condensed form, because it is one of the most practical artifacts in the paper: it tells you, for each level, what you need to have built.

	Level 1	Level 2	Level 3
Sandbox	On the developer's machine or a remote environment; writes kept to the workspace; no host filesystem access	Fresh sandbox per task on shared infrastructure, from an authenticated profile, destroyed at the end	Each agent in a run gets its own sandbox from its profile; outside monitoring can tighten or stop it mid-run
Network	Default-deny egress proxy outside the sandbox	Egress proxy per agent with rules from its profile; factory not reachable from outside	Each agent has its own egress rules and task scope; agent-to-agent traffic passes through the same enforcement
Supply chain	Only approved registries and packages reachable; scanning; cooldown on new versions	Sandboxes from approved images with attested base layers and authenticated configuration; every source checked against the approved set	Signals that start work are checked too; inputs and outputs, including messages between agents, validated for structure and provenance, with heuristic injection screening
Identity and access	Agent acts as the developer or as itself, within the developer's identity boundary	Factory acts for the requester or as itself; revoking grants and stopping the sandbox acts as a kill switch	Each agent has its own identity and access, acting for itself, another agent or a person, with authority narrowing at each hop
Execution layer	The developer, through the agent harness	A durable system that starts runs, launches agents in sandboxes and retries failures	The same, starting each agent only with its authorised scope
Orchestrator	The developer	A predefined workflow	An orchestrator agent that takes work from requests and signals, runs preflight and divides work across agents
Observability	Harness logs plus identity and proxy records, reviewed by the developer	Records from all four boundaries in one place, linked by run; evals track output quality	Telemetry and evals across runs; monitoring of running agents; stopping agents that drift or become ineffective

Controls by level

The reference architecture also maps each level to the Agent Baseline controls that become formally required there. It is careful to say that all four boundaries apply at every level, and that the mapping shows where each control is first required as a formal, enforced control, not where the underlying property starts to matter. Earlier levels often meet the same properties in simpler ways, such as a developer's own review.

Level 1 requires action attribution (AUT-01), purpose and task-bound authority (AUT-02), just-in-time credentials (AUT-04), isolated execution (CON-03) and testing of agent-generated artifacts (VAL-03).

Level 2 adds admission enforcement (CON-01), use-case-scoped capability profiles (CON-04), delegation attenuation (AUT-03), independent approval and step-up verification (AUT-05, AUT-07), fail-closed authorisation (AUT-06), the agent and component registries (DIS-01, DIS-04), composition mapping (DIS-05), agent-native telemetry and evidence integrity (OBS-01, OBS-06), end-to-end correlation (OBS-02), first-party component testing (VAL-02), immediate revocation (RES-01) and component quarantine (RES-02).

Level 3 adds drift monitoring and unintended action detection (OBS-03, OBS-04), agent-specific security testing (VAL-01), outcome validation (VAL-04), impact scoping (RES-03) and automated discovery and reconciliation (DIS-07).

Read as a build plan, the jump from Level 1 to Level 2 is the largest. Level 2 is where the factory acquires most of its shared infrastructure: registries, profiles as authenticated artifacts, a runtime authorisation service with independent approval, correlated telemetry and the ability to revoke and quarantine. Level 3 adds monitoring and validation capabilities on top of that foundation, but does not need a different foundation.

Levels are per class of work, not per organisation

One of the most useful sentences in the reference architecture is easy to miss: a small team can run a Level 3 factory while a large enterprise keeps some work at Level 1.

Levels describe how a particular kind of work is orchestrated. An organisation will usually run several levels at once. At Harbour Ledger, by the end of Part III:

- every developer uses a Level 1 setup for exploratory and complex work;
- dependency upgrades, flaky test fixes, documentation updates and small bug fixes from the ticket system run through a Level 2 factory;
- one narrow class of production alert, for one service, starts work through a Level 3 pilot.

There is no contradiction in that. It is calibration. The question is never "what level are we?" but "what level is right for this class of work, given the evidence we have?"

Choosing where to start

For most organisations, the right first step is to make Level 1 safe for everyone, then build Level 2 for a small number of well-understood classes of work. Very few organisations should start at Level 3.

That recommendation follows from what each level needs.

Prerequisites for Level 1

Level 1 needs remarkably little that a reasonably mature engineering organisation does not already have:

- a sandboxed environment for local or remote agents, with writes confined and no host access;
- an egress proxy or equivalent network control in front of it;
- an internal package proxy, ideally with a cooldown on new versions;
- agent identities distinct from developer identities, or at least a way to mark agent-made commits and requests;
- short-lived credentials for the development services agents use;
- the organisation's usual CI and review.

Chapter 12 builds this.

Prerequisites for Level 2

Level 2 needs the Level 1 controls made shared and durable, plus:

- a work intake with a Definition of Ready and a specification format;
- a durable execution layer;
- a profile catalogue, with profiles as signed, versioned artifacts;
- a runtime authorisation service, and an egress proxy that enforces its grants at the operation level;
- independent approval paths that factory identities cannot satisfy;
- a component registry;
- an audit trail that links every boundary's records by run, out of the agents' reach;
- verification gates with explicit enforcement levels;
- run metrics and evaluations.

It also needs something that is not infrastructure: **a codebase that is ready for agents**. Level 2 depends on verification, and verification depends on tests. A service with poor test coverage, flaky tests or a build that only works on one engineer's laptop will produce unreliable factory output, whatever the factory's quality. DORA's finding that AI amplifies existing strengths and weaknesses applies directly here.

Chapter 13 builds Level 2.

Prerequisites for Level 3

Level 3 needs everything in Level 2, operating reliably, plus:

- validated signal sources, with policy written in advance for each signal type;
- an orchestrator whose output is treated strictly as a request;
- monitoring that can detect thrash and drift and stop agents mid-run;
- outcome validation, meaning a way to confirm in production that a change did what it was meant to do;
- impact scoping, meaning a way to determine what a misbehaving run touched;
- progressive delivery with automatic rollback for the services involved.

Chapter 15 builds a narrow Level 3 pilot.

A readiness assessment

Before choosing which classes of work to move to Level 2, it helps to assess the codebases they touch. A short assessment, scored per repository, is enough to start. Harbour Ledger used something like the following. It is illustrative, and the thresholds are judgements, not standards.

```
REPOSITORY READINESS FOR LEVEL 2 (score each 0-2)

Build
  [ ] Builds from a clean checkout with one documented command
  [ ] Build is reproducible in CI without manual steps
Tests
  [ ] Unit tests run in under 10 minutes
  [ ] Flaky test rate tracked and below team threshold
  [ ] Meaningful coverage of the areas the work class touches
Context
  [ ] AGENTS.md or equivalent exists and is current
  [ ] Constitution or architecture principles written down
Protection
  [ ] Branch protection on main and release branches
  [ ] CODEOWNERS for sensitive paths
  [ ] Secret scanning on the repository, with no open findings
Delivery
  [ ] Changes can ship behind feature flags
  [ ] Rollback is tested and takes under 15 minutes

Ready for Level 2 work in this repo: total >= 18 and no zero in Tests
```

The assessment does two jobs. It tells the platform team where the factory is likely to succeed. And it gives product teams a concrete list of things to improve if they want their repositories to benefit, which is a better conversation than “the factory doesn’t work on our code”.

A roadmap

Putting it together, a typical roadmap has five stages. The durations depend on the organisation and are deliberately left out; what matters is the order and the exit criteria.

Stage 0: Foundations. Fix what the readiness assessment finds in the repositories you intend to start with. Establish the package proxy, secret scanning and branch protection if they are missing. Write the approval

policy's first draft. *Exit when* the target repositories score as ready and the approval policy is agreed.

Stage 1: Safe Level 1 for everyone. Roll out the hardened developer setup from Chapter 12. Give agents distinct identities. Start collecting harness and proxy logs. *Exit when* agent work at Level 1 runs inside the boundaries by default and the fence check passes.

Stage 2: Level 2 for one or two classes of work. Build the shared factory for low-risk, high-volume work, such as documentation and dependency patch upgrades. Measure everything. *Exit when* the factory's output for those classes has a review acceptance rate, change failure rate and cost per change that the owning teams accept, over a period long enough to trust.

Stage 3: Widen Level 2. Add classes of work one at a time, each with its own profile, gates and approval route. Promote classes to lighter approval routes only on evidence. *Exit when* the factory handles the routine work the organisation wants it to, and the outer loop is visibly improving profiles and skills.

Stage 4: A Level 3 pilot. Choose one signal type for one service, with good tests, good rollback and a willing team. Build preflight, monitoring and outcome validation for it. *Exit when* the pilot has run long enough to show its proposals are accepted at a high rate and its failures are contained.

Each stage includes the outer loop for the factory itself. The roadmap is not a waterfall; it is a series of bets, each tested before the next.

Pitfalls

Treating levels as maturity badges. Levels describe orchestration for a class of work. Higher is not better in itself.

Skipping Level 1 controls because "we're going straight to Level 2". Developers will keep using agents locally whatever the factory does. Make that safe first.

Starting Level 2 on the hardest codebase. Start where tests, builds and rollback are strongest. Success there builds the evidence and the trust to tackle harder code.

Starting Level 3 before Level 2 is boring. If the Level 2 factory still surprises its operators, adding an orchestrator agent multiplies the surprises.

Building for every class of work at once. Hitachi's lesson from Chapter 2 again: one process for all work fails. Add classes one at a time.

Chapter summary

- The reference architecture defines three levels by what orchestrates the work: a developer (Level 1), a predefined workflow (Level 2) or an orchestrator agent (Level 3). The four boundaries stay the same at every level; infrastructure takes on more of the supervision as autonomy grows.
- Each level has its own architectural challenge: securing an agent under supervision, securing shared execution and delegated access, and keeping an agent's plans from becoming its permissions.
- The jump from Level 1 to Level 2 requires the most new infrastructure and the most new Agent Baseline controls.
- Levels apply per class of work. Most organisations will run several levels at once.
- Start by making Level 1 safe for everyone, build Level 2 for a few low-risk classes of work on ready codebases, widen it on evidence, and pilot Level 3 narrowly.

CHAPTER 12

Building Level 1: The Hardened Developer Factory

Most organisations already have a Level 1 software factory. They just have not built it on purpose. Every developer who runs a coding agent on a laptop, with their own credentials, their own shell history, their own cloud profiles and their own access to every repository they have ever cloned, is operating a factory with no boundaries at all. The agent can read everything the developer can read and reach everything the developer can reach. The reference architecture's appendix makes the point directly: at Level 1, the sandbox and egress proxy usually have to be added around a local coding agent, since most local tools run with the developer's own access by default.

This chapter builds a Level 1 factory deliberately. The goal is the one the reference architecture states for this level: give the agent room to work without giving it everything the developer can reach. The developer stays in charge of the outer loop, kicking off runs and reviewing results, but no longer has to approve every command, because the boundaries hold whatever the agent does.

The chapter follows Harbour Ledger's platform team through the build, including the parts that went wrong. As always, Harbour Ledger is a composite, and the configurations are illustrative rather than tied to any one product.

What Level 1 needs to achieve

Before building anything, Harbour Ledger's platform team wrote down what "done" would mean. Their list mapped directly onto the four boundaries and onto the Level 1 controls from Chapter 11.

1. **Sandbox.** The agent runs in an isolated environment, not directly on the developer's operating system. Writes are confined to a workspace. The agent cannot read the developer's home directory, credentials, shell history or other repositories.
2. **Network.** All of the agent's traffic goes through a default-deny egress proxy outside the sandbox. It can reach source control, approved package registries, the model gateway, documentation and a short list of development services, and nothing else.
3. **Supply chain.** Packages come only from internal proxies, with scanning and a cooldown on new versions.
4. **Identity and access.** The agent's actions are distinguishable from the developer's. Credentials it needs are short-lived and attached outside the sandbox where possible.
5. **Observability.** Harness logs, proxy logs and identity records are kept, and the developer can see what the agent did.
6. **Testing.** A Level 1 fence check proves the above.

Step 1: Choose where the sandbox lives

There are two broad options at Level 1: a local sandbox on the developer's machine, or a remote development environment.

A **local sandbox** runs the agent inside a lightweight virtual machine or a hardened container on the laptop. It is fast and works offline for editing, but it inherits the laptop's weaknesses: the developer can disable it, and local virtualisation varies between operating systems.

A **remote development environment** runs the agent in a cloud workspace provisioned for the task. It is easier to standardise and harder to bypass, and it puts the sandbox on infrastructure the platform team controls. It costs more and depends on connectivity.

Harbour Ledger chose remote environments as the default and a local microVM option for developers who needed to work offline. The reasoning was that a sandbox the platform team could not see or update would drift, and drifted sandboxes are not sandboxes.

Whichever option is chosen, the sandbox should satisfy the reference architecture's sandbox controls that apply at Level 1:

- writes confined to the task workspace, system paths read-only;
- no access to the host filesystem;
- the agent runs unprivileged, with no route to root;
- no access to the container runtime socket or host devices;
- resource and time limits;
- tests that need containers run inside the sandbox's own isolation.

The last point produced Harbour Ledger's first real problem. Several services' integration tests started a database with Docker, and the old developer setup mounted the host's Docker socket into the development container so the tests could do that. The reference architecture's appendix is explicit that mounting the developer's Docker socket hands the agent control of the host. The platform team replaced it with a nested container runtime inside the microVM, so tests could start containers without any path to the host. Two teams' integration tests needed small changes to work with it. That was a week of friction, and it was worth it.

Step 2: Put an egress proxy in front of it

The sandbox's only network route leads to an egress proxy. On a remote environment, this is a network policy on the workspace's virtual network. On a local microVM, it is the virtual network interface the VM is given.

The critical property is that the proxy is *enforced*, not *configured*. An `HTTPS_PROXY` variable is a suggestion. The sandbox should have no route to anything except the proxy. A minimal illustration using nftables on the host side of a local VM's virtual interface might look like this, where `vmnet0` is the VM's interface and the proxy listens on the host at 10.99.0.1:

```
table inet sandbox_egress {
  chain forward {
```

```
    type filter hook forward priority 0; policy drop;
    # nothing from the sandbox is forwarded anywhere
    iifname "vmnet0" drop
  }
  chain input {
    type filter hook input priority 0; policy accept;
    # the sandbox may talk only to the proxy and the approved resolver
    iifname "vmnet0" ip daddr 10.99.0.1 tcp dport 3128 accept
    iifname "vmnet0" ip daddr 10.99.0.1 udp dport 53 accept
    iifname "vmnet0" drop
  }
}
```

The proxy itself then applies the allow-list. The reference architecture's guidance for a local coding agent is specific: it needs to reach source control, approved package registries, model gateways, documentation and a few specific development services, and nothing else. That means blocking private network ranges unless a specific internal host is required, blocking cloud metadata endpoints, blocking direct IP connections, blocking unapproved DNS and blocking arbitrary HTTP and HTTPS.

Harbour Ledger's Level 1 allow-list, in outline:

```
kind: EgressPolicy
metadata: { name: level1-default }
spec:
  default: deny
  dns: { resolvers: [proxy] }          # the proxy resolves names itself
  block: [metadata, rfc1918, directIp, rawSockets]
  allow:
    - git.internal                     # source control
    - goproxy.internal                 # package proxies (one per ecosystem)
    - npm.internal
    - pypi.internal
    - model-gateway.internal
    - docs.internal                    # internal documentation
    - pkg.go.dev                       # public docs, read-only GET
    - developer.mozilla.org
  requestAccess:
    process: ticket to platform-team   # how to ask for a new destination
    temporaryGrant: 8h                 # per-developer, logged
```

The `requestAccess` section turned out to be the most important part. In the first week, the allow-list blocked a dozen legitimate needs: a public API's documentation, a test fixture hosted on a vendor's site, a code search service one team relied on. Developers who hit a wall with no way through started running agents outside the sandbox. The platform team added a lightweight request process, with temporary grants for a single

developer for eight hours and a weekly review that promoted common requests to the shared list. Bypasses stopped.

The lesson generalises. A boundary that is too strict and has no escape valve gets bypassed, and a bypassed boundary is worse than a looser one, because the organisation believes it is protected.

Step 3: Route packages through the supply chain boundary

At Level 1, the supply chain boundary is mostly about packages. The reference architecture's Level 1 description is that only approved registries and packages are reachable, through the egress proxy or another admission point, that what is allowed is scanned, and that new versions wait out a cooldown.

Most organisations of any size already have internal package proxies for build reliability. The work at Level 1 is to make them the *only* path. Because the egress proxy blocks the public registries directly, an agent that tries `npm install` against the public registry fails, and the harness configuration sets each package manager to use the internal proxy by default so that it does not try.

Harbour Ledger configured its proxies to:

- hide versions published within the last 72 hours, with an expedited exception process for security fixes;
- hide versions flagged by its supply chain scanner;
- disable install scripts by default for the JavaScript ecosystem, with a per-package allow-list for the few that genuinely need them;
- log every fetch with the requesting identity.

The cooldown caused one predictable complaint: a team wanted a framework release on the day it shipped. The exception process handled it in an hour, with a security engineer reviewing the release notes and the package's diff.

Step 4: Separate the agent's identity from the developer's

This is the hardest part of Level 1, because most tools assume the agent acts as the developer.

Harbour Ledger took three steps.

Source control. Instead of using the developer's personal token, the agent authenticates to source control through the egress proxy, which holds a short-lived token issued to an agent identity for that developer's session. The token allows fetching the repositories the developer has open in the workspace and pushing to branches with an `agent/<developer>/` prefix. It does not allow pushing to main or release branches, approving pull requests or merging. Pull requests opened by the agent appear under the agent identity, with the developer recorded as the requester.

Commits. Every commit the agent makes carries trailers identifying the agent and the session, so that even after a squash merge, the history shows which changes came from an agent:

```
Fix retry classification for 429 responses

Agent-Identity: factory-agent/level1/ak-dev-3f21
Agent-Session: 3f21a9
Requested-By: j.ortiz
```

Cloud and internal services. Developers at Harbour Ledger routinely had cloud credentials on their laptops. None of them were mounted into the sandbox. Where an agent needed a development service, the proxy attached a short-lived credential for that service, scoped to development environments only. Production was not reachable at all.

This did not make the agent fully independent of the developer. The reference architecture acknowledges that at Level 1 the agent acts as the developer or as itself, *within the developer's identity boundary*. But it meant that the agent could do much less than the developer, that what it did was attributable, and that revoking its session token stopped it without affecting the developer's own access.

Step 5: Route inference through the model gateway

All model traffic went through Harbour Ledger's model gateway, which the egress proxy allowed and attached credentials for. The gateway enforced an approved model list, a per-session spend limit, and the company's data-handling rule that code from repositories classified as

restricted could be sent only to providers under an enterprise agreement with no training on inputs.

The gateway also produced the first useful cost data. Within a month, the platform team could see spend per developer, per repository and per model, which later fed the cost metrics in Chapter 16.

Step 6: Configure the harness, and know what it is for

With the boundaries in place, the harness configuration becomes what it should be: guidance and accident prevention inside the loop, not the last line of defence.

Harbour Ledger's default harness configuration did four things.

- It loaded the repository's AGENTS.md and constitution automatically.
- It ran a formatter hook after every edit, and a fast lint hook before the agent declared a task complete.
- It asked the developer for confirmation before destructive commands such as recursive deletes and force pushes, as a convenience for catching mistakes early.
- It made the organisation's approved skills available from a pinned, internal skills registry.

The platform team was explicit, in its internal documentation, that the confirmation prompts were not security controls. If a developer disabled them, the boundaries still held. That explicitness turned out to matter: developers who understood that the prompts were for their own convenience stopped resenting them, and some turned them off for routine work, which was fine.

Step 7: Give developers a better workflow, not just a safer one

A hardened Level 1 setup that is safer but slower will lose to an unhardened one that is faster. Harbour Ledger's platform team spent as much effort on the developer workflow as on the boundaries.

The workflow they settled on for agent-assisted work has five steps.

Write a short task specification. Not a full Chapter 4 specification, but a few lines in a standard template: goal, acceptance criteria, non-goals, files likely involved.

```
## Task
Goal: retry 429 responses honouring Retry-After
Accept: AC1 retry on 429; AC2 wait at least Retry-After seconds;
        AC3 cap wait at 300s
Not: no change to other status handling
Where: internal/delivery/classify.go, internal/jobs/retry.go
```

Start the agent in a fresh workspace with that task. The workspace starts from the repository's current main branch and the standard image.

Let it run. The developer does something else, or works on a different part of the problem.

Review the result like any other pull request, with a short checklist for agent-produced changes:

```
AGENT CHANGE REVIEW (Level 1)
[ ] Does the diff do what the task asked, and only that?
[ ] Are there changes outside the files I expected? Why?
[ ] Were any tests deleted, skipped or weakened?
[ ] Were any dependencies added? Through the proxy? Needed?
[ ] Is there new error handling that hides failures?
[ ] Would I be able to explain this code if asked tomorrow?
```

The third and fifth items deserve their places. Agents trying to get tests to pass sometimes weaken the tests or add broad exception handling that makes failures silent. A reviewer who checks for those two things catches a surprising share of the problems agent changes contain.

The last item is the whiteboard defense applied at the smallest scale. If the reviewer cannot explain the change, it is not ready to merge, however green the checks are.

Discard the workspace. It is destroyed when the session ends. Nothing persists except what was pushed.

Step 8: Keep the record

At Level 1, the reference architecture describes observability as harness logs plus identity and proxy records, reviewed by the developer. Harbour Ledger kept three streams:

- **harness logs**, recording the agent's prompts, tool calls and responses, stored outside the workspace;
- **proxy logs**, recording every request, allowed or denied, with the session and agent identity;
- **identity records**, recording every token issued to an agent session and what it was used for.

They were linked by session identifier and retained for ninety days. Developers could view their own sessions. The security team could view all of them. Prompts and responses were treated as sensitive data, with access controls and retention limits, following the reference architecture's guidance that captured prompts and responses be protected with access controls, retention limits and encryption, and that redaction help but not be relied on alone.

Step 9: Write and run the Level 1 fence check

Finally, the platform team wrote a Level 1 version of the boundary test suite from Chapter 8, run automatically against a freshly provisioned workspace every night and on every change to the workspace image or proxy configuration.

```
LEVEL 1 FENCE CHECK
workspace.read_home ..................... expect DENY
workspace.read_other_repo ............... expect ABSENT
workspace.escalate_root ................. expect DENY
workspace.docker_socket ................. expect ABSENT
workspace.cloud_credentials ............. expect ABSENT
net.public_npm_direct ................... expect DENY
net.metadata ............................ expect DENY
net.rfc1918_unlisted .................... expect DENY
net.direct_ip ........................... expect DENY
net.custom_dns .......................... expect DENY
net.arbitrary_https ..................... expect DENY
supply.version_in_cooldown .............. expect HIDDEN
git.push_main ........................... expect DENY
git.approve_pr .......................... expect DENY
```

```
git.push_agent_branch .................. expect ALLOW
obs.session_logged ..................... expect PRESENT
```

In its first week, the fence check caught a real regression: an image update had accidentally restored a cached cloud credentials file from the build machine. The check failed, the image was rolled back, and nobody had used it.

What changed

The platform team measured three things before and after the rollout. They are reported here as Harbour Ledger's illustrative experience, not as expected results.

Agent usage went *up*, not down, once the request process for new destinations was in place, because developers who had been nervous about running agents with their own credentials became willing to use them more.

The share of agent-produced pull requests that needed substantial rework fell, mostly because of the task template and the review checklist, not the boundaries.

The security team's list of "things that could go badly wrong with agents" shrank from a page to a paragraph, and the paragraph was about Level 2.

Pitfalls

Proxies set by environment variable. Enforce at the network layer.

No escape valve. A strict allow-list with no request process gets bypassed. Provide temporary, logged grants and review them weekly.

Docker socket for tests. Replace it with isolation inside the sandbox.

Agent as developer. Give the agent its own identity and short-lived, narrower credentials, even if it still operates within the developer's identity boundary.

Treating harness prompts as controls. They help developers catch mistakes. They are not boundaries.

Ignoring workflow. A safer setup that is slower will lose. Invest in task templates, fresh workspaces and review checklists.

Chapter summary

- Most organisations already run an unbounded Level 1 factory. Building one deliberately means adding a sandbox and an egress proxy around the agent so it has room to work without the developer's full access.
- Choose a sandbox location the platform team can control and update. Confine writes, remove host access, run unprivileged and replace Docker socket mounts with isolation inside the sandbox.
- Enforce egress at the network layer. Allow source control, package proxies, the model gateway, documentation and specific development services. Block metadata endpoints, private ranges, direct IPs, unapproved DNS and arbitrary HTTPS. Provide a fast, logged way to request more.
- Route packages through internal proxies with scanning and a cooldown.
- Give the agent its own identity, with short-lived credentials attached outside the sandbox and commit trailers that record its involvement.
- Make the developer's workflow better: short task specifications, fresh workspaces, a review checklist for agent changes.
- Keep linked records and run a Level 1 fence check every night.

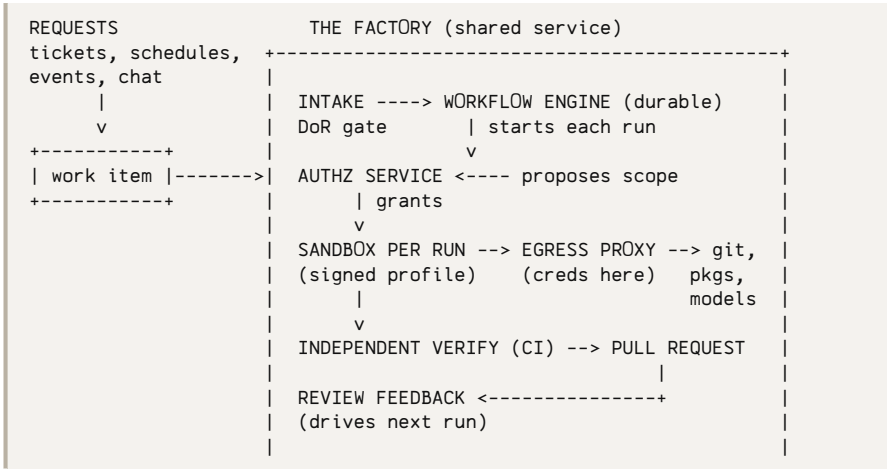
CHAPTER 13

Building Level 2: The Factory as a Shared Service

At Level 1, the factory lives inside each developer's workflow. At Level 2, it moves out and becomes a service. Work arrives from tickets, schedules and events rather than from a developer at a keyboard. A predefined workflow turns each piece of work into one or more runs. Every run gets a fresh sandbox, its own grants and its own record. Review feedback drives the next run until the pull request is ready. And because the factory is shared, an improvement made for one team's work reaches every team that uses it.

This is the level at which most organisations will get most of the value from a factory, and it is the level that needs the most new infrastructure. This chapter builds it, piece by piece, using Harbour Ledger's experience as a guide. As before, Harbour Ledger is a composite and its configurations are illustrative.

The shape of a Level 2 factory



```
                      |  AUDIT: all boundaries, linked by run       |
                      +---------------------------------------------+
                                          |
                          APPROVAL (policy, person) --> MERGE
```

There are seven components to build: intake, the workflow engine, profiles as signed artifacts, the authorisation service and proxy, per-run sandboxes, independent verification and the audit trail. Chapters 8 to 10 covered the boundary components in principle. This chapter focuses on how they fit together as a service.

Intake

The factory needs a single front door. Work can originate in many places, such as the issue tracker, a scheduled job, a chat message, an event from another system, but it should all become the same kind of object before the factory acts on it: a *work item*.

A work item is the factory's internal representation of a request. It carries the specification, or a pointer to it, and the metadata the rest of the factory needs. Harbour Ledger's work item schema, in outline:

```
kind: WorkItem
id: WI-2291
source:
  type: ticket                  # ticket | schedule | event | chat
  ref: PAY-4410
  createdBy: human/j.ortiz      # authenticated identity of the creator
spec: specs/SPEC-0142.md@4c1a9e  # pinned to a commit
class: small-bugfix             # maps to a profile and approval route
riskClass: B
repo: notifications-service
scope:
  paths: [internal/delivery/**, internal/jobs/**]
budget:
  changedLines: 400
  runs: 5                       # max iterations including review rounds
owner: team/payments-experience
status: Ready
```

Intake is where the Definition of Ready from Chapter 4 is enforced. A work item that fails the schema, lacks acceptance criteria or has open questions without owners does not become `Ready`, and the factory does not start it. The creator gets a specific message about what is missing.

Intake is also where input validation happens. The `source` block records where the work came from and who created it. That provenance feeds

the authorisation decision later: a work item created by an authenticated engineer with write access to the repository can lead to a different grant from one created by an automated scanner, and one forwarded from an external form may be allowed to start only read-only investigation.

Harbour Ledger's first intake mistake was accepting work items from a chat command without checking the requester's permissions on the target repository. A contractor with read-only access asked the factory to fix something and got a pull request. Nobody was harmed, but the factory had, in effect, written on behalf of someone who could not write. The fix was the `requester.canWrite` check in the task policy from Chapter 10, applied at intake as well.

The workflow engine

The *execution layer* at Level 2 is a durable system. The reference architecture names three kinds: a CI/CD pipeline, a step runner or a durable execution engine. All can work. What matters is that the engine has the properties a factory needs.

Durability. A run can last from minutes to over an hour, and a work item with several review rounds can last days. The engine must survive restarts without losing track of where work is.

State outside the agent. The engine, not the agent, knows which task is in progress, how many iterations have been used, what review feedback is pending and what the next step is. This is the Ralph lesson from Chapter 5, made robust.

Retries with limits. Steps that fail for transient reasons, such as a network glitch or a model timeout, should be retried. Steps that fail because the work cannot succeed should stop. The engine needs to tell the difference, or at least cap the attempts.

Timeouts at every level. Per step, per run and per work item.

A stop button. The engine must be able to stop any run, or all runs, on demand. The reference architecture's execution controls require that stopping agents on demand actually stops work across the factory.

Fail closed. If the engine cannot reach the authorisation service, the egress proxy or the audit sink, it does not start the agent.

CI systems already used for builds are often the fastest way to start, because they have runners, secrets management, logs and permissions already. Their weakness is usually long-running, multi-round workflows with human feedback in the middle. Dedicated durable execution engines handle those well but are another system to operate. Harbour Ledger started with its CI system and moved the multi-round review loop to a durable execution engine after three months, when work items routinely spanned several days.

Here is an illustrative workflow definition for a Level 2 bug fix.

```
kind: Workflow
metadata: { name: small-bugfix, version: 4 }
on: workItem.status == Ready && workItem.class == small-bugfix
steps:
  - id: plan
    agent: { profile: planner@1.3.0 }
    output: plan.md              # proposed tasks, paths, tests
    gate: policy.planFitsProfile # deterministic check
  - id: review-plan
    when: workItem.riskClass in [A, B]
    human: { from: workItem.owner, timeout: 24h }
  - id: authorise
    call: authz.grant(task: plan.tasks[*])
  - id: build
    forEach: plan.tasks
    agent: { profile: small-bugfix@2.1.0, sandbox: fresh }
    limits: { wallClock: 45m, steps: 300, verifyRetries: 3 }
  - id: verify
    ci: pipelines/factory-verify.yml   # runs outside the sandbox
    gate: hardMandatory
  - id: auto-review
    agent: { profile: reviewer@1.1.0, modelFamily: different-from(build) }
    output: review-notes.md
  - id: open-pr
    call: scm.openPullRequest(evidence: true)
  - id: feedback
    loop:
      until: pr.approved || workItem.budget.runs.exhausted
      on: pr.changesRequested
      do: [build, verify, auto-review, update-pr]
  - id: close
    on: workItem.budget.runs.exhausted
    do: [comment("Run budget exhausted; returning to owner"), stop]
```

A few design choices are worth pointing out.

The **plan is its own step**, produced by a planning profile, checked by a deterministic policy and, for higher-risk classes, reviewed by a person before any code is written. A bad plan caught here costs minutes.

Each task gets a fresh sandbox. Nothing carries over from one task to the next except what is in the repository.

Verification runs outside the sandbox, in CI, on the change as submitted.

Automated review uses a different model family from the one that built the change.

Review feedback drives the loop. When a reviewer requests changes, the workflow starts a new build run with the review comments added to the context, and the cycle repeats until the pull request is approved or the work item's run budget is exhausted.

Exhaustion returns the work to a person. The factory does not keep trying forever.

Profiles as signed artifacts

At Level 1, the profile is spread across the harness configuration and the sandbox setup. At Level 2, the reference architecture requires that the profile be authenticated and tamper-evident, so the runtime can verify it and refuse to start when it cannot, or when required controls are missing. It suggests packaging the profile as signed metadata in an OCI image, or binding a signed configuration to an approved image at launch.

Harbour Ledger took the first approach. Each profile is built by a pipeline in the platform team's repository, which assembles the sandbox image, the harness version, the pinned skills and the profile metadata into a single OCI artifact, signs it with the platform's build identity and records its provenance. The execution layer verifies the signature and provenance before starting a sandbox from it.

```
profile repo change
   -> review (platform-team; security for enforced fields)
   -> build: image + harness + skills + profile.yaml
   -> evals (Chapter 14) against the previous version
   -> sign (platform build identity) + provenance
```

```
  -> publish to registry as profile:<name>@<version>
  -> execution layer verifies on every launch
```

Two steps in that pipeline are easy to skip and important to keep.

Security review for enforced fields. Changes to the guiding parts of a profile, such as skills and context, are reviewed by the platform team. Changes to the enforced parts, such as network rules, access rules and sandbox configuration, also need security review, because those are the boundaries.

Evaluations before promotion. A profile change can make agents worse. The reference architecture calls for using evaluations and run metrics to determine whether changes to skills, tools, models and workflows improve outcomes or introduce regressions before promoting them. Chapter 14 describes how.

Harbour Ledger learned the second lesson the hard way. An early update to the `dependency-upgrade` skill, intended to make the agent update lock files more carefully, also changed the order in which it ran tests. Pull requests from the new version failed CI more often, and reviewers noticed before the platform team did. After that, no profile version reached production without passing an evaluation run against a fixed set of past work items.

The factory's identity

At Level 2 the factory has its own identity and acts on behalf of the requester when there is one. Harbour Ledger implemented this as a source control application identity for the factory as a whole, with per-run agent identities derived from it.

Every pull request the factory opens is authored by the factory identity and carries the requester and run in its description and in commit trailers. Branch protection on every protected branch requires approval from a human identity in the owning team; the factory identity is explicitly excluded from approver groups. The factory can open pull requests and push to its own branches. It cannot approve, merge or push to protected branches, whatever any agent asks.

The reference architecture notes that at Level 2, revoking the agent's grants and stopping its sandbox acts as a kill switch. Harbour Ledger built both into a single command available to on-call engineers and the security team, and tested it monthly.

```
$ factory stop --scope all --reason "INC-3312 investigation"
  revoked grants ........ 37
  stopped sandboxes ..... 12 (state preserved)
  paused workflows ...... 41
  new runs .............. blocked until 'factory resume'
```

Grants and enforcement as a service

The authorisation service and egress proxy described in Chapters 8 and 10 run as shared services at Level 2. Each run's sandbox gets its own proxy instance, or its own policy on a shared proxy, with rules from its profile and its task's grant.

The practical work at this stage is integration. Every downstream system the factory touches needs a protocol-aware enforcement point that understands the operations the factory uses: the source control API for branches and pull requests, the package proxies, the model gateway, and any internal services that particular profiles need. Harbour Ledger started with source control and package proxies only, which covered every one of its first four classes of work, and added others only when a new class of work needed them.

Per-run sandboxes at scale

At Level 2 sandboxes are started and destroyed constantly. Two practical concerns appear.

Start-up time. A microVM that takes a minute to boot and another five to fetch dependencies makes every run slower and every iteration of the review loop more expensive. Pre-built images with dependencies already resolved for the repository's current main branch, refreshed regularly through the same supply chain controls, cut this substantially. So does keeping a warm pool of booted but unassigned sandboxes, provided each is used once and destroyed.

Isolation between tenants. Sandboxes for different teams, different repositories and different risk classes run on shared infrastructure. The isolation between them must be as strong as the isolation between a sandbox and the host. The fence check's `sandbox.cross_task` test should run across tenants, not just within one.

Independent verification

Every change the factory produces is verified by CI outside the sandbox before it becomes a pull request, and again on every update. Harbour Ledger's factory verification pipeline runs the repository's normal checks plus a set of factory-specific checks that apply to all agent-produced changes:

- the change stays within the work item's `scope.paths`;
- the change budget is respected;
- protected paths are untouched;
- no tests were deleted or marked skipped without an explicit allowance in the specification;
- every acceptance criterion in the specification maps to at least one test;
- held-out acceptance tests, maintained by the owning team, pass;
- the SBOM is generated and no disallowed licence or blocked component appears.

These checks are *hard-mandatory*: failing any of them stops the change. Chapter 14 discusses the other enforcement levels.

The audit trail

At Level 2, the reference architecture requires records from all four boundaries collected in one place and linked by run. That means a run identifier assigned by the execution layer at the start and attached to every record: the authorisation decision, every proxy request, every package admission, every sandbox process and file event, every harness log line and every CI result.

The records go to an append-only store outside every sandbox, written by the infrastructure components themselves rather than by agents. The audit sink being unavailable is a reason to refuse new runs.

A useful test of the audit trail is to pick a merged factory pull request at random and ask: from the records alone, can we answer who asked for this, what was granted, which agent and profile did the work, which components it used, what it tried that was denied, which checks passed and who approved it? If any of those answers requires asking a person, the trail has a gap.

Limiting work in progress

Harbour Ledger's second serious mistake at Level 2 was not a security problem. It was a flow problem.

Once the dependency upgrade class was running smoothly, the platform team scheduled it across all forty services at once. The factory opened about sixty pull requests in a morning. The owning teams' review queues, which were already busy, filled up. Many of the pull requests conflicted with each other or with ongoing work within days and had to be rebased, which started new runs and produced new review requests. Reviewers started approving dependency upgrades without reading them, which was precisely the behaviour the approval policy was meant to prevent.

The fix was a work-in-progress limit, applied per team: the factory would not open a new pull request for a team whose queue of open factory pull requests was at its limit. The limit started at three and was adjusted per team based on how quickly each team reviewed. Throughput of *merged* changes went up after the limit was introduced, which is exactly what Little's law from Chapter 6 predicts.

What Harbour Ledger's Level 2 factory looked like after six months

By the end of its first six months, Harbour Ledger's illustrative Level 2 factory handled five classes of work: documentation updates, dependency patch and minor upgrades, flaky test fixes, small bug fixes

with clear reproductions, and lint and static analysis cleanups. Each had its own profile, its own gates and its own approval route. Two classes, documentation and lint cleanups, had been moved to class C approval, requiring automated review and passing gates but no human approval before merge, after the evidence in Chapter 16's metrics supported it.

The factory did not handle feature work, schema migrations, anything touching authentication, or anything in the three services whose tests the readiness assessment had scored poorly. Those stayed at Level 1, with developers driving agents directly.

That is a modest scope. It is also a factory that ran thousands of tasks without a security incident, whose output reviewers trusted, and whose operators could explain every change it made. That is the foundation for everything that follows.

Pitfalls

Many front doors. Every source of work should become a work item through one intake with one Definition of Ready.

State in the agent. The workflow engine, not the agent, should know where the work is.

Unsigned profiles. At Level 2 the runtime must verify the profile before starting a sandbox.

Profile changes without evaluation. A better-looking skill can make outcomes worse. Evaluate before promoting.

Factory identities in approver groups. Audit every approval path.

No work-in-progress limit. The factory can produce work faster than teams can review it. Limit it per team.

An audit trail you have never queried. Pick random changes and try to reconstruct them from records alone.

Chapter summary

- A Level 2 factory is a shared service. Work from tickets, schedules, events and chat becomes a work item through a single intake that enforces the Definition of Ready and records provenance.

- The execution layer is a durable system that keeps state outside the agent, retries with limits, times out at every level, can stop everything on demand and fails closed.
- Workflows separate planning, plan review, authorisation, building in fresh sandboxes, independent verification, automated review by a different model family and human review, with feedback driving further runs until approval or budget exhaustion.
- Profiles become signed, versioned artifacts built and evaluated before promotion, with security review for their enforced fields.
- The factory has its own identity, acts for requesters, and is excluded from every approval path.
- Records from every boundary are linked by run in an append-only store outside every sandbox.
- Limit work in progress per team so the factory does not overwhelm review.

CHAPTER 14

Quality Gates, Tests and Evals

There is a particular kind of pull request that every team working with agents eventually sees. The task was to fix a failing test. The pull request is green. Every check passes. And when a reviewer reads the diff carefully, the fix turns out to be a change to the test, not to the code: an assertion loosened, an expected value updated to match the wrong output, a case marked as skipped "temporarily".

The agent was not being deceptive in any meaningful sense. It was asked to make a test pass, and it found the shortest path to a passing test. If the factory's only definition of success is "the tests pass", the shortest path to success will sometimes run through the tests.

This chapter is about defining success in a way that is harder to short-circuit. It covers the gates that a change passes through, the tests that give those gates their meaning, independent review, and the evaluations that tell you whether the factory itself is getting better or worse.

Gates and their enforcement levels

A *gate* is a check that a change must pass to move from one stage to the next. Factories have many of them: the Definition of Ready at intake, the policy check on the plan, the verification suite, review, the approval policy at merge and deploy.

Not every gate should have the same force. The ai-sdlc framework uses three enforcement levels for its quality gates, and the vocabulary is worth adopting.

- **Advisory.** The gate reports a finding but does not block. A suggestion from an automated reviewer about test structure is advisory.
- **Soft-mandatory.** The gate blocks by default but can be overridden by an authorised person, with the override recorded. A coverage drop

below a threshold might be soft-mandatory: usually a problem, occasionally justified.

- **Hard-mandatory.** The gate blocks and cannot be overridden within the workflow. A secret detected in the diff, a change to a protected path, or a failing held-out test is hard-mandatory.

Deciding the level for each gate is a design decision with consequences. Too many hard-mandatory gates and the factory stalls on false positives, which trains people to look for ways around it. Too few and the factory's output is only as good as its reviewers' attention on a given afternoon.

A reasonable principle is that a gate should be hard-mandatory when its check is *deterministic and its failure is unambiguous*, soft-mandatory when its check is deterministic but its failure sometimes has a legitimate explanation, and advisory when the check is a judgement. ai-sdlc's review principles include *deterministic-first*: defer to CI for anything CI can decide, and reserve judgement-based review for what it cannot.

Harbour Ledger's illustrative gate catalogue for agent-produced changes:

Gate	Check	Level
Build, types, lint	Deterministic	Hard
Unit and integration tests	Deterministic	Hard
Held-out acceptance tests	Deterministic	Hard
Secret scan on diff	Deterministic	Hard
Protected paths untouched	Deterministic	Hard
Scope paths respected	Deterministic	Hard
Change budget	Deterministic	Hard
No deleted or skipped tests	Deterministic	Hard unless spec allows
Every acceptance criterion has a test	Deterministic mapping	Hard
New dependency introduced	Deterministic	Soft (needs owner sign-off)
Coverage of changed lines below threshold	Deterministic	Soft
SAST findings, medium severity	Deterministic	Soft
Automated review findings	Judgement	Advisory, unless marked blocking by reviewer agent and confirmed by a person
Human review	Judgement	Required by approval policy

Tests: the agent's and everyone else's

Tests are what give verification its meaning. In a factory, tests come from three places, and they are not equally trustworthy.

Existing tests in the repository. These encode what the code is already supposed to do, and they protect against regressions. They are only as good as the repository's testing culture, which is why the readiness assessment in Chapter 11 weighs them heavily.

Tests the agent writes for its own change. Agents should write tests, and good agents write useful ones. But an agent that misunderstands the specification will write tests that confirm its misunderstanding. The code and the tests share an author, and so they share blind spots.

Held-out tests derived from the specification by someone other than the implementing agent and kept out of its reach. These are the independent check. They might be written by the owning team, by a separate test-writing agent working only from the specification, or a mixture. What matters is that the implementing agent never sees them and so cannot, even inadvertently, shape its code to them or them to its code.

Held-out tests are the single most effective defence against the green-but-wrong pull request, and they are cheaper than they sound. They do not need to cover everything. They need to cover the acceptance criteria that matter most and are easiest to get subtly wrong. For SPEC-0142 in Chapter 4, the owning team wrote held-out tests for exactly two criteria: that retries carry identical payload bytes and event IDs, and that there is never more than one concurrent attempt for the same webhook. Those were the criteria where a plausible implementation could easily be wrong.

Practical arrangements for keeping held-out tests out of reach:

- store them in a separate repository or a protected directory that is not mounted into the sandbox;
- run them only in the factory's verification pipeline, outside the sandbox;

- report only pass or fail, and the criterion each relates to, back to the agent, never the test code itself, so the agent can fix its implementation but cannot fit to the test.

That last point is a judgement call. Reporting the name of the failing criterion helps the agent fix real bugs. Reporting the test body invites it to write code that satisfies the test's specific inputs. Harbour Ledger reported the criterion and a one-line description, and nothing more.

Guarding the tests themselves

The opening scenario of this chapter, where an agent weakens a test to make it pass, needs its own guards.

- A hard gate that fails any change which deletes tests or marks them as skipped, unless the specification explicitly allows it.
- A soft gate on changes to assertion lines in existing tests, which routes them to a human for review.
- Periodic *mutation testing* on the repository's tests, which introduces small deliberate faults into the code and checks that the tests catch them. Mutation testing is too slow to run on every change, but run regularly it shows whether the tests are strong enough to be worth trusting as a gate.

Kinds of test that suit factories

Some kinds of test are especially valuable when agents write the code.

Property-based tests check that an invariant holds over many generated inputs, rather than checking specific examples. "For every sequence of delivery outcomes, the number of attempts never exceeds eight" is a property. Properties are harder to satisfy by accident than examples, and they express specifications naturally.

Contract tests check that an interface behaves as its consumers expect. They are the natural way to enforce the "interfaces that must not change" section of a specification.

Schema diffs on public interfaces, such as API definitions and event schemas, catch unintended breaking changes mechanically.

Golden tests compare output against a stored, reviewed reference. They are useful for generated artifacts such as documentation or configuration, where the expected output is known.

Review by a different mind

Automated review is useful in a factory, with one important design rule: the reviewer should not share the implementer's blind spots.

ai-sdlc enforces that the harness which implemented a change cannot review its own work. A further step, common in practice, is to use a model from a *different family* for review than the one that implemented the change. Models from the same family, trained on similar data with similar methods, tend to make similar mistakes and to find each other's output plausible. A reviewer from a different family is more likely to notice what the implementer missed.

An automated reviewer in a factory should be given:

- the specification, including acceptance criteria and non-goals;
- the plan;
- the diff;
- the verification results.

And it should be asked specific questions rather than "review this":

```
REVIEWER PROMPT (outline)
For each acceptance criterion: is it implemented? Where? Is it tested?
For each non-goal: does the diff violate it?
Does the diff change anything outside the planned scope? List it.
Were any tests weakened, deleted or skipped? List them.
Is there error handling that could hide failures?
Is there behaviour not covered by any criterion? Describe it.
Mark each finding: blocking | non-blocking. Give evidence (file:line).
```

Its findings are added to the evidence summary. In Harbour Ledger's factory, reviewer findings marked as blocking do not stop the change on their own, because an LLM's judgement is not a deterministic check. They are surfaced prominently to the human reviewer, who decides. Over time, the platform team tracks how often blocking findings are confirmed by people, and that confirmation rate is itself a measure of the reviewer profile's quality.

The limits of LLM judges

It is tempting to make an LLM reviewer the gate, especially for classes of work that are approved without a human. The reference architecture's caution about LLMs as judges of input applies to output too: the same text that misleads the implementer can mislead the reviewer. An agent that has been manipulated into writing something harmful may also have written comments that persuade a reviewing model it is fine.

So LLM review should be one layer among several, never the only gate for anything that matters. Deterministic checks, held-out tests and human review for higher-risk classes carry the weight. LLM review reduces what the humans have to catch.

Evaluating the factory

Everything so far in this chapter evaluates *changes*. A factory also needs to evaluate *itself*: whether its profiles, skills, models and workflows are getting better or worse.

The reference architecture describes *evals* as tests that measure how well an agent or the factory performs over time, as part of observability, and calls for using evals and run metrics to determine whether changes to the factory's skills, tools, models and workflows improve outcomes or introduce regressions before promoting them.

What public benchmarks can and cannot tell you

Public benchmarks are the most visible evaluations in the field. SWE-bench, introduced by Jimenez and colleagues in 2023, built 2,294 task instances from real GitHub issues and their resolving pull requests across twelve popular Python repositories, with tests that fail before the fix and pass after it. In August 2024, OpenAI released SWE-bench Verified, a human-validated subset of five hundred instances, selected to keep tasks with clear problem descriptions and reliable tests.

The existence of SWE-bench Verified carries a lesson for factory builders. Even a carefully constructed benchmark contained a meaningful number of tasks whose descriptions were unclear or whose tests were unreliable, and it took human validation to find them. Your own work items will have the same problem.

Public benchmarks are useful for choosing which models to consider. They say little about how a model will perform on *your* codebase, with *your* skills, under *your* profiles. For that, you need your own evaluations.

Building an internal eval set

An internal eval set for a factory is a collection of past work items with known good outcomes. Building one is straightforward in principle.

1. **Collect** completed work items for a class of work, with the repository state before the change, the specification and the merged change.
2. **Curate.** Keep items whose specifications were clear and whose outcomes are verifiable by tests. Discard items that needed extensive human discussion to resolve, or whose tests were unreliable. This is the SWE-bench Verified lesson applied internally.
3. **Add held-out checks** to each item: tests the factory's agents have not seen, derived from the specification, that a correct solution must pass.
4. **Freeze** the set and version it. An eval set that changes every week cannot detect regressions.
5. **Refresh** it periodically with new items, as a new version, keeping old versions for comparison.

Harbour Ledger's first eval set for the `small-bugfix` class had sixty items, drawn from six months of bugs fixed by people before the factory existed. That was enough to detect large regressions, not small ones, which was the right starting point.

What to measure in an eval

For each eval run, record per item:

- whether the change passed verification;
- whether it passed the held-out checks;
- how many inner-loop iterations it needed;
- tokens and cost;
- wall-clock time;
- whether it stayed within scope and budget;

- for a sample, a human judgement of whether the change would have been accepted in review.

Aggregate those across the set and compare against the previous version of the profile, skill or model. A change is promoted only if it does not regress the key measures beyond an agreed tolerance.

An eval-gated promotion process

```
PROFILE PROMOTION (small-bugfix)

candidate: small-bugfix@2.2.0   baseline: small-bugfix@2.1.0
eval set:  bugfix-evals@v3 (60 items)

                          baseline   candidate   tolerance   result
held-out pass rate         0.72       0.75        -0.03       OK
verification pass rate     0.85       0.87        -0.03       OK
median iterations          3          3           +1          OK
median cost (USD)          2.10       2.60        +0.50       OK
scope violations           0          0           0           OK
human-accept (sample 15)   11/15      12/15       -2          OK

decision: promote to 10% of runs for 7 days, then full if live
          metrics match eval results
```

The last line introduces an idea borrowed from progressive delivery: *canary the factory itself*. Even a profile that passes evals is first rolled out to a fraction of real runs, and its live metrics compared with the baseline's, before it replaces the old version everywhere.

Goodhart's law

Every measure in this chapter is vulnerable to the observation commonly known as Goodhart's law: when a measure becomes a target, it ceases to be a good measure.

A factory that optimises for verification pass rate will find ways to pass verification, including the opening scenario's weakened tests. A factory that optimises for review acceptance will learn what reviewers wave through. An eval set that is reused for long enough becomes something the factory's configuration is tuned against, and stops predicting real performance.

The defences are structural rather than clever.

- **Measure several things at once**, so that gaming one shows up as a change in another. A rising verification pass rate with a flat or falling held-out pass rate is a warning.
- **Keep some measures out of reach.** Held-out tests and frozen eval sets are valuable precisely because nothing in the factory optimises directly against them.
- **Include outcomes from production.** Change failure rate and rework rate, covered in Chapter 16, are much harder to game than any pre-merge measure.
- **Refresh eval sets.** Add new items regularly, from recent real work.
- **Keep humans looking at samples.** A periodic deep review of a random sample of merged factory changes, by experienced engineers, catches drift that no metric shows.

Pitfalls

"Tests pass" as the definition of done. Without held-out tests and guards on test changes, the shortest path to green may run through the tests.

Every gate hard-mandatory. False positives train people to circumvent the factory. Match the enforcement level to how deterministic and unambiguous the check is.

Same-family review. A reviewer that shares the implementer's training shares its blind spots.

LLM review as the only gate. It can be manipulated by the text it reviews. Use it to reduce human effort, not replace it.

No evals for the factory. Without them, every profile change is a guess.

Eval sets that never change, or change constantly. Freeze versions; refresh on a schedule.

Chapter summary

- Gates have enforcement levels: advisory, soft-mandatory and hard-mandatory. Make a gate hard when its check is deterministic and its failure unambiguous; prefer deterministic checks wherever possible.

- Tests in a factory come from the repository, from the implementing agent and from held-out sources. Held-out tests, kept out of the agent's reach and reported only as pass or fail per criterion, are the strongest defence against changes that are green but wrong.
- Guard the tests themselves: block deletions and skips, route assertion changes to people, and run mutation testing periodically.
- Use automated review by a different model family, with specific questions and evidence, as one layer among several.
- Evaluate the factory with frozen, curated internal eval sets drawn from past work, promote profile changes only when they do not regress, and canary them on live runs.
- Goodhart's law applies to every measure. Measure several things, keep some out of reach, include production outcomes and keep people reviewing samples.

CHAPTER 15

Level 3: Orchestrators, Signals and Self-Healing

The reference architecture's worked example, "a factory in action", describes a Level 3 factory responding to a production problem. It is short, and it rewards close reading.

A production alert reports elevated errors in a payments service. The orchestrator agent proposes a task for an incident agent: read access to the service's logs for the last hour, and permission to open one pull request. The authorisation service checks that request against policy for that kind of incident and the alert's authenticated source, and the egress proxy attaches credentials only to the requests it grants. Partway through, the agent decides it needs logs from an unrelated authentication service. Its reasoning may be sound, but the task does not cover that service, so the proxy blocks the call, and the agent keeps working within its scope unless it can justify a new authorisation decision. Repeated attempts outside its scope would be flagged by monitoring and could lead to its grants being revoked and the run stopped. The fix is tested in the sandbox, using a library admitted through the supply chain, and leaves as a pull request with a provenance attestation that approval policy routes to a person or approves automatically. The deploy reaches a small share of traffic first, and the error rate afterward confirms the fix or starts the next run.

The paper summarises the division of labour in one sentence that is worth memorising: *the agent decided what it wanted, the authorization service decided what the task justified, and the proxy determined what happened.*

This chapter builds the pieces that make that example work: signals, the orchestrator, preflight, monitoring and outcome validation. It ends with Harbour Ledger's narrow Level 3 pilot.

What changes at Level 3

At Levels 1 and 2, a person or a predefined workflow decides what work happens next. At Level 3, an agent does. The reference architecture calls this the most consequential change, and it is easy to see why. The orchestrator reads untrusted inputs, interprets them and proposes work. If its proposals were permissions, anyone who could influence what it reads could influence what the factory does.

So Level 3 is built on one rule, stated in the reference architecture: nothing the orchestrator decides authorises itself. In preflight, the orchestrator proposes a profile and a task scope for each piece of work, and the authorisation service independently decides what each task gets. The orchestrator can also find weaknesses in the factory and propose changes to it, but it cannot approve them.

Everything else at Level 3 follows from making that rule hold under pressure.

Signals

A *signal*, in the reference architecture's definition, is an event that starts Level 3 work without a person filing it: production incidents, scanner findings, evaluation results, and tickets created automatically or by other agents. The definition ends with a sentence that matters more than any other in this chapter: *signals are untrusted inputs.*

Chapter 7 explained why. A signal's source can be authenticated, and should be. But its content may still be attacker-influenced. An alert quoting a log line quotes whatever was logged. A scanner finding quoting a dependency's description quotes whatever the dependency's author wrote. A ticket created by another agent contains whatever that agent produced, which may itself have been shaped by injected text.

At Level 3, the reference architecture extends the supply chain boundary to cover signals: the signals that start work are checked too, and agent

inputs and outputs, including tickets, context and messages from other agents, are validated for structure and provenance, with heuristic screening for injection.

Signal types and policy written in advance

The most effective control for signals is to define, in advance, a small number of *signal types*, each with a fixed schema and a fixed policy. A signal that does not match a known type does not start work. A signal that matches a type can start only the kind of work its policy allows.

Here is an illustrative signal type definition.

```
kind: SignalType
metadata: { name: service-error-rate-alert, version: 3 }
source:
  system: alerting.internal
  authentication: mtls + signed payload
schema:
  service:    { type: string, mustExistIn: service-catalog }
  metric:     { type: enum, values: [http_5xx_rate, job_failure_rate] }
  threshold:  { type: number }
  observed:   { type: number }
  window:     { type: duration, max: 1h }
  startedAt:  { type: timestamp }
  summary:    { type: text, maxLength: 500, treatAs: untrusted }
policy:
  allowedServices: [notifications-service, ledger-export]   # pilot scope
  workClass: incident-investigation
  profile: incident-readonly
  scopeTemplate:
    logs:    { service: "${signal.service}", window: "${signal.window}" }
    metrics: { service: "${signal.service}", window: 6h }
    repo:    "${catalog[signal.service].repo}"
    pullRequests: 1
  approvalRoute: class-B
  rateLimit: { perService: 3/day }
```

Notice how little of the signal's content can influence the scope. The `service` field must name something in the service catalogue, and only services on the pilot's allow-list can start work. The `window` is capped. The `repo` is looked up from the catalogue, not taken from the signal. The free-text `summary` is marked untrusted; it is passed to the agent as context but plays no part in deciding scope.

This is one practical answer to the question the reference architecture leaves open: how to derive a task's scope from untrusted inputs. The approach here is to *not* derive scope from untrusted content at all,

beyond a few typed, validated parameters that select among options fixed in advance. It is restrictive. It is also robust, and for a first Level 3 deployment, restrictive and robust is the right trade.

The orchestrator

The orchestrator agent takes work from requests and signals, gathers context, and plans. In the reference architecture's terms, it runs preflight and divides work across agents.

The orchestrator is itself an agent, which means everything in this book about agents applies to it. It runs under its own profile, in its own sandbox, with its own identity, behind its own egress proxy. Its profile should be narrow. It needs to read signals, the service catalogue, recent changes and relevant documentation, and to submit proposals to the authorisation service and the workflow engine. It does not need, and should not have, write access to code or any production system.

Preflight

Preflight is the orchestrator's planning step. For each piece of work, it gathers context and proposes a profile, a task scope and resources. A preflight proposal for the payments alert in the reference architecture's example might look like this:

```
kind: PreflightProposal
signal: sig-a93f (service-error-rate-alert v3)
orchestrator: factory-orchestrator/prod-1
proposed:
  - task: investigate-and-fix
    profile: incident-readonly@1.4.0
    scope:
      logs: { service: payments-api, window: 1h }
      metrics: { service: payments-api, window: 6h }
      repo: payments-api
      pullRequests: 1
    rationale: >
      5xx rate rose from 0.2% to 3.1% at 02:06, eleven minutes after
      deploy d-5521, which changed request validation. Likely regression.
    budget: { wallClock: 40m, usd: 10 }
```

The authorisation service then evaluates this proposal against the signal type's policy. If the proposed scope matches what the policy's scope template would produce, the grant is issued. If the orchestrator proposes more, such as logs from another service, a longer window or a second

pull request, the excess is denied or routed to a person. The orchestrator's rationale is recorded as evidence but does not influence the grant.

Delegation

Some work divides naturally. An orchestrator might propose one agent to investigate and another to implement a fix once the cause is known. The reference architecture requires that each agent in a run gets its own sandbox, its own identity and its own egress rules and task scope, and that traffic between agents passes through the same enforcement as any other traffic. Delegation narrows authority at each hop, never widens it.

In practice, the simplest robust pattern is for agents never to delegate directly to each other. Instead, an agent that needs further work done *requests* it from the orchestrator, which proposes it through preflight like any other work. That keeps every grant flowing through the authorisation service and every agent-to-agent message on a mediated path.

The monitoring agent

Monitoring is required at every level. At Level 3, the reference architecture describes an emerging pattern: a *monitoring agent* that processes signals and watches work in progress across the factory, catching agents that thrash or drift and feeding what it finds back as new work. It runs outside the agents it watches and, like the orchestrator, can flag and request but not grant or approve. The paper credits Travis McPeak of Cursor with describing the approach on the Insecure Agents podcast.

The monitoring agent watches two kinds of misbehaviour.

Thrash. An agent that cannot reach its goal and starts trying increasingly creative workarounds. Signs include:

- repeated denied requests at the proxy, especially to varied destinations;
- rising tool-call counts without progress in tests;
- repeated edits to the same lines;

- attempts to modify test or build configuration;
- budget burn well above the profile's median.

Drift. An agent that gradually moves away from its assigned task while staying within its permissions. Access controls do not catch drift, because every individual action is allowed. The reference architecture notes that monitoring has to. Signs include:

- edits in files that are within scope but unrelated to the task;
- reading large amounts of code unrelated to the specification;
- a growing diff that touches concerns the specification lists as non-goals;
- context or plan files rewritten in ways that redefine the task.

Some of these signals are simple thresholds that deterministic monitoring can catch. Others need judgement, which is where a monitoring agent helps: it can compare what an agent is doing with what its task says, and raise a flag when they diverge.

The monitoring agent's actions are limited to flagging and requesting. It can ask the execution layer to pause a run, ask the authorisation service to revoke a grant pending review, or create a work item for a person to investigate. Its requests go through the same policy as any other. Whether a pause request is honoured automatically is a policy decision; Harbour Ledger's illustrative policy honours pause requests immediately, because pausing is cheap and reversible, and routes revocation requests to the on-call engineer.

Outcome validation

At Level 3 the factory can deploy, and the reference architecture adds *outcome validation* (VAL-04) as a formally required control. Validation, as Chapter 6 explained, asks whether the change did what it was meant to do in the real world.

For a factory responding to signals, outcome validation has a natural shape: the signal that started the work defines the outcome. An error-rate alert is resolved if the error rate returns below threshold after the fix

is deployed, and stays there. A scanner finding is resolved if the scanner no longer reports it.

The deployment path for factory changes at Level 3 should therefore include:

1. **A canary.** The change reaches a small share of traffic first.
2. **An outcome check.** The metric that the signal reported is compared before and after, over a defined window.
3. **Automatic rollback** if the metric worsens or other health indicators regress.
4. **A result recorded against the original signal**: resolved, not resolved, or made worse.

If the outcome check shows the fix did not work, the reference architecture's description is precise: the error rate afterward confirms the fix or starts the next run. The next run starts with the evidence from the first one.

Impact scoping

When a Level 3 run misbehaves, the factory needs to know what it touched. That is *impact scoping* (RES-03). Because every boundary records what it granted, admitted, executed and reached, linked by run, the audit trail can answer: which resources did this run read or write, which artifacts did it produce, which of those were merged or deployed, and which other runs depended on them?

Impact scoping should be a query, not an investigation. Harbour Ledger's platform team wrote it as one, and tested it against deliberately misbehaving runs in a staging factory.

```
$ factory impact run-c81e
  grants:        g-77a1 (logs payments-api 1h), g-77a2 (pr payments-api x1)
  denied:        14 requests (auth-service logs x11, metrics-admin x3)
  read:          payments-api logs 01:10-02:10; repo payments-api@e41b
  produced:      PR #4471 (open, not merged)
  deployed:      none
  dependents:    none
  recommended:   close PR #4471; review denied requests (pattern: thrash)
```

Harbour Ledger's Level 3 pilot

Harbour Ledger chose its pilot carefully, following the roadmap in Chapter 11. It picked one signal type, the error-rate alert above, for two services: the notifications service from earlier chapters and a ledger export service. Both had strong tests, well-practised rollback, feature flags on most behaviour, and owning teams who volunteered.

The pilot ran in stages.

Shadow mode. For the first month, the orchestrator ran preflight on every qualifying alert, and the incident agent investigated and prepared a fix, but nothing was opened as a pull request. The on-call engineer handled each incident as usual and afterwards compared their own diagnosis and fix with the factory's. This produced the first real evidence about the factory's judgement, without any risk.

Proposal mode. For the next two months, the factory opened pull requests for its proposed fixes, routed as class B: one human approval required, canary deployment, automatic rollback. The on-call engineer could approve, modify or reject.

Review. At the end of proposal mode, the platform and owning teams reviewed the results together. Which proposals were accepted unchanged, which were modified, which were rejected, and why? Did any canary roll back? Did monitoring flag any runs, and were the flags right?

The illustrative outcome was mixed in the way real pilots usually are. The factory's diagnoses were often right about *where* the problem was and less often right about the *best* fix; engineers modified a good share of the proposals before approving. Two canaries rolled back automatically, both correctly. Monitoring flagged one run for thrash, when the incident agent tried repeatedly to read logs from a downstream service the task did not cover, exactly the pattern in the reference architecture's example; the proxy had blocked every attempt, and the flag was right.

The most valuable outcome was not any individual fix. It was that on-call engineers, woken by an alert, increasingly found a diagnosis and a candidate fix waiting for them. Mean time to a first hypothesis fell sharply, even when the final fix was the engineer's own.

The teams decided to keep the pilot in proposal mode, widen it to three more services, and revisit after another quarter. Nobody proposed automatic approval. The evidence did not support it yet, and everyone knew what evidence would.

Pitfalls

Letting signal content shape scope. Use typed, validated parameters selecting among options fixed in advance. Treat free text as untrusted context.

An orchestrator with write access. The orchestrator plans. It should not be able to change code or production.

Direct agent-to-agent delegation. Route requests for further work back through the orchestrator and preflight, so every grant goes through the authorisation service.

Monitoring that can grant. Monitoring agents flag and request. They never grant or approve.

Deploying without outcome checks. At Level 3 the signal defines the outcome. Check it after every canary.

Skipping shadow mode. Shadow mode costs little and produces the evidence the rest of the pilot depends on.

Chapter summary

- At Level 3 an orchestrator agent decides what work happens next. The architectural rule is that nothing it decides authorises itself: it proposes in preflight, and the authorisation service decides.
- Signals are untrusted inputs. Define signal types in advance with fixed schemas and policies, and let only typed, validated parameters select among scope options fixed in advance.
- The orchestrator is an agent with its own narrow profile, sandbox and identity. Delegation should flow back through preflight rather than directly between agents.
- A monitoring agent watches for thrash and drift and can flag and request, never grant or approve.

- Outcome validation uses the signal to define success, with canary deployment, outcome checks, automatic rollback and results recorded against the signal.
- Impact scoping should be a query over the linked audit trail.
- Pilot Level 3 narrowly, starting in shadow mode, then proposal mode, then review.

PART IV

Operating and Governing the Factory

CHAPTER 16

Measuring the Factory

Six weeks after Harbour Ledger's Level 2 factory went live, the platform team presented its first metrics to engineering leadership. The headline slide said that the factory had opened 412 pull requests and that agents had written 31 percent of all changed lines that month. Leadership was delighted. The head of the payments team, who had reviewed a good number of those pull requests personally, was not.

Her objection was simple. Pull requests opened is not pull requests merged. Lines written is not value delivered. And neither number said anything about whether the changes were good, whether they had caused incidents, or how much of her team's time had gone into reviewing them. "You've measured how busy the factory is," she said. "I want to know if it's helping."

She was right, and the platform team rebuilt its metrics around her question. This chapter describes what they ended up measuring and why.

Activity, output and outcome

It helps to separate three kinds of measure.

Activity measures what the factory does: runs started, tokens consumed, pull requests opened, lines generated. Activity is easy to measure and easy to inflate. It is useful for capacity planning and cost control. It says nothing about value.

Output measures what the factory produces that people accept: pull requests merged, work items completed, review acceptance rates. Output is closer to value but still incomplete, because accepted changes can be bad changes.

Outcome measures the effect of the factory's work on delivery and on the product: lead time, change failure rate, rework, incidents, customer-facing results. Outcome is what leadership actually cares about, and it is the hardest to attribute.

A good factory dashboard has all three, clearly labelled, with outcome at the top.

The METR study from Chapter 3 is the standing reminder of why this matters. Experienced developers believed they were faster with AI tools when they were measurably slower. Perception and activity measures can both point confidently in the wrong direction.

DORA metrics for the factory

The best-established outcome measures for software delivery are DORA's. In 2024 DORA expanded its model from four software delivery metrics to five:

- **Change lead time**: how long a change takes from commit to running in production.
- **Deployment frequency**: how often changes are deployed.
- **Failed deployment recovery time**: how long it takes to recover when a deployment fails.
- **Change fail rate**: the share of deployments that cause a failure needing remediation.
- **Deployment rework rate**: the share of deployments that are unplanned, made to fix a user-facing bug or incident.

DORA groups the first three as throughput and the last two as instability measures. Its 2024 report did not publish performance tiers for rework rate, so no benchmark thresholds are offered here.

For a factory, the key move is to **segment these metrics by origin**. Compute them separately for changes produced by the factory, for changes produced by developers using Level 1 agents, and for changes produced without agents, and segment further by class of work. That segmentation answers the questions leadership actually has. Are factory changes failing more or less often than human changes of the same

kind? Is the factory reducing lead time for dependency upgrades, or just producing more of them? Are factory bug fixes causing rework?

Segmenting requires that origin be recorded reliably, which is one more reason for the agent identities and commit trailers in Chapters 10 and 12.

DORA's 2024 finding that AI adoption was associated with lower throughput and stability, and its hypothesis that larger changes were the cause, suggests one more measure to put alongside the five: **change size**, segmented the same way. If factory changes are growing, instability is likely to follow.

Run metrics

Below the delivery level, the factory needs metrics about its own runs. The reference architecture lists the core set: how often output passes verification and review, retries per run, time and cost. The ai-sdlc framework's metrics specification adds measures such as task success rate, intervention rate, escalation count and override rate.

Harbour Ledger's run metrics, per class of work and per profile version:

Metric	Definition	Why it matters
Verification pass rate	Share of runs whose output passes all hard gates within the retry budget	Inner loop effectiveness
Held-out pass rate	Share of runs passing held-out tests	Correctness independent of the agent's own tests
Review acceptance rate	Share of factory PRs merged without substantial rework	Output quality as judged by people
Review rounds	Median review iterations before merge	Specification and profile quality
Inner-loop iterations	Median build-verify cycles per run	Efficiency; spikes suggest thrash
Budget exhaustion rate	Share of runs stopped by a budget	Task fit for the profile
Escalation rate	Share of runs that escalate a request outside scope	Scope quality; possible injection or drift
Override rate	Share of soft-mandatory gate failures overridden	Gate calibration
Abandonment rate	Share of work items returned to a person unfinished	Fit of class of work to the factory

Metric	Definition	Why it matters
Wall-clock time	Median and 90th percentile, intake to PR	User-facing speed

Several of these interact in revealing ways. A rising verification pass rate with a falling held-out pass rate suggests agents are getting better at passing their own tests without getting better at the task, the Goodhart pattern from Chapter 14. A rising review acceptance rate with rising review rounds suggests reviewers are accepting changes only after more back and forth, which costs reviewer time. A rising escalation rate for one profile may point to a scope template that is too narrow, or to inputs that are trying to push agents outside their tasks.

Cost

Cost needs particular care, because the obvious measures mislead.

Spend per run is easy to compute from the model gateway and the sandbox infrastructure, and nearly useless on its own. A cheap run that produces a pull request nobody merges is pure waste. An expensive run that produces a correct fix for a hard bug may be excellent value.

The useful measures are:

- **cost per merged change**, meaning total factory spend for a class of work divided by changes merged;
- **cost per accepted change**, meaning the same but counting only changes merged without substantial rework;
- **human time per merged change**, meaning reviewer minutes and specification-writing minutes attributed to factory work.

The third is the one teams most often forget, and it is frequently the largest. A factory that saves implementation time but doubles review time can easily cost more in total than the work it replaced. Measuring reviewer time precisely is hard; Harbour Ledger approximated it from the time pull requests spent in review with a reviewer assigned, adjusted by periodic sampling where reviewers logged actual minutes for a week.

The model gateway's per-task and per-agent budgets from Chapter 5 are the control side of cost. They stop a thrashing agent from running up

unbounded spend. The metrics side is the trend: is cost per accepted change falling as profiles improve? If not, why not?

Thrash and drift

Chapter 15 described thrash and drift as security concerns. They are also operational ones, and they belong on the factory's dashboard.

Thrash shows up as runs with high iteration counts, high denied-request counts at the proxy, high budget consumption and low success. A rising thrash rate for a profile usually means something in the environment changed: a dependency, a test fixture, a service the profile relies on, or a model update at the gateway.

Drift is harder to quantify. A practical proxy is the share of each change's diff that falls outside the files the plan predicted. Some divergence is normal, because plans are imperfect. A rising share for a profile suggests either that its plans are getting worse or that its agents are wandering.

Review load

The review stage is the factory's most likely bottleneck, so it deserves its own metrics.

- **Queue depth**: open factory pull requests awaiting review, per team.
- **Time to first review**: from PR opened to first human comment or approval.
- **Review time per change**: as above, approximated.
- **Stale rate**: share of factory PRs that need rebasing before merge because they waited too long.

These are the inputs to the work-in-progress limits in Chapter 13. When queue depth rises and time to first review stretches, the factory should slow down for that team, not speed up.

The share-of-code trap

Many organisations want to report the share of their code written by AI. It is an understandable number to want, and the reference architecture

itself quotes several companies' figures for the share of pull requests produced by agents. As a signal of direction, such figures are interesting.

As a target, the share of code written by agents is a poor one. It rewards producing more code, which rewards larger changes, which DORA's research associates with instability. It says nothing about whether the code was needed, correct or maintainable. And it is trivially gamed: an agent that writes verbose code raises the share.

If an organisation must report a share, the least misleading version is the **share of merged changes, by class of work, that originated in the factory, alongside the change fail rate and rework rate for those same changes**. That at least connects the volume to its consequences.

Telemetry

All of these metrics depend on telemetry that is collected consistently and linked by run. The reference architecture's observability controls ask the factory to define the security-relevant events each boundary must record, link every boundary's records with a shared run identifier, keep the combined record tamper-evident and outside any agent's reach, and version the factory's design configuration so every run traces to the exact policy it ran under.

Operational metrics can be built on the same records. Many teams use OpenTelemetry for their services already, and the same approach works for the factory: each run is a trace, each step a span, with attributes for work item, class, profile version, model route, agent identity and outcome. OpenTelemetry has published semantic conventions for generative AI operations, which can provide a starting vocabulary for model calls.

One caution from the reference architecture applies here. Prompts and responses are captured for evaluation and investigation, and they are sensitive. They may contain code, customer data quoted in tickets, or internal discussion. Protect them with access controls, retention limits and encryption, redact where possible, and do not rely on redaction alone.

A dashboard

Harbour Ledger's factory dashboard, after the rebuild, had four panels. In outline:

```
+-- OUTCOMES (by class, factory vs non-factory) ----------------------+
| lead time | change fail rate | rework rate | recovery time | size   |
+-------------------------------------------------------------------+
+-- OUTPUT ----------------------------------------------------------+
| merged | review acceptance | review rounds | abandonment | held-out |
+-------------------------------------------------------------------+
+-- COST ------------------------------------------------------------+
| cost per accepted change | reviewer minutes per change | vs budget  |
+-------------------------------------------------------------------+
+-- HEALTH ----------------------------------------------------------+
| thrash | drift share | escalations | denied requests | fence check  |
+-------------------------------------------------------------------+
```

Each panel could be filtered by class of work, profile version, team and time period. The fence check status, from Chapter 8, sat in the health panel, so that a failing boundary test was as visible as a falling acceptance rate.

Using metrics to change approval routes

The most consequential use of factory metrics is deciding when a class of work can move to a lighter approval route, and when it must move back.

Harbour Ledger wrote its promotion criteria into the approval policy, so that they were agreed in advance rather than argued about case by case. The illustrative criteria for moving a class from B (one human reviewer) to C (automated review and hard gates, no human before merge) were:

```
PROMOTION B -> C (per class of work)
Over at least 200 merged changes and at least 8 weeks:
  - change fail rate for factory changes in this class <= the rate
    for human changes in the same class over the same period
  - rework rate likewise
  - review acceptance rate >= 95%
  - no security-relevant finding in human review
  - no hard-gate bypass
  - fence check green throughout
And: owning team and platform-security both agree.

DEMOTION C -> B (automatic)
Any of:
  - a production incident attributed to a class C change
```

```
  - change fail rate for the class exceeds the threshold over 4 weeks
  - any security-relevant finding in sampled post-merge review
```

The demotion triggers are automatic and the promotion is not. That asymmetry is deliberate. Trust should be slow to grant and quick to withdraw. The ai-sdlc framework builds the same idea into its autonomy policies: autonomy is earned with demonstrated reliability, with promotion criteria and demotion triggers.

Running honest comparisons

Finally, a word on experiments. When the factory changes, such as a new profile version or a new model route, the evaluation process in Chapter 14 tests it before promotion. But some questions can only be answered on live work: is the factory actually reducing lead time for this class of work compared with how people handled it before?

Before-and-after comparisons are vulnerable to everything else that changed in the meantime. Where it is practical, a better approach is the one METR used: randomise. For a period, assign eligible work items at random either to the factory or to the usual human process, and compare outcomes. It is slower and less exciting than declaring victory. It is also how you find out whether you have won.

Pitfalls

Leading with activity. Runs, tokens, pull requests opened and lines generated are not value. Put outcomes first.

Unsegmented DORA metrics. Organisation-wide numbers hide whether factory work is better or worse than human work of the same kind.

Ignoring human time. Reviewer and specification time is often the largest cost of factory work.

Share of code as a target. It rewards volume and size. Report it, if at all, alongside fail and rework rates.

Promotion by argument. Write promotion and demotion criteria into the approval policy in advance.

Before-and-after as proof. Randomise when you can.

Chapter summary

- Distinguish activity, output and outcome measures, and lead with outcomes.
- Use DORA's five delivery metrics, segmented by origin and class of work, plus change size, to see whether factory work is better or worse than other work of the same kind.
- Track run metrics: verification and held-out pass rates, review acceptance and rounds, iterations, budget exhaustion, escalation, override and abandonment rates, and time. Watch how they move together.
- Measure cost per accepted change and human time per change, not spend per run.
- Track thrash, drift and review load, and use review load to set work-in-progress limits.
- Treat share of code written by AI as a weak signal, never a target.
- Build metrics on run-linked telemetry, protect prompts and responses as sensitive data, write approval promotion and demotion criteria in advance, and randomise comparisons where possible.

CHAPTER 17

Governance, Compliance and Accountability

When Harbour Ledger's external auditors arrived for their annual review, they asked a question nobody on the platform team had prepared for. It was not about sandboxes or proxies. It was this: "For a change that went to production last quarter, show us who was responsible for it."

For a change written by a person, the answer had always been easy: the author, the reviewer and the approver, all named in the pull request. For a change produced by the factory, the platform team found itself explaining agent identities, profiles and grants. The auditor listened patiently and repeated the question. *Who was responsible?*

The answer the platform team eventually gave, and the one this chapter argues for, was the same as for any other change: the person who requested it, the person who approved it, and the team that owns the service. The factory produced the change. People and the organisation remained accountable for it. What the factory added was a much more complete record of how the change came to be.

Governance of a software factory is the work of making that answer true, provable and routine.

The accountability principle

The reference architecture is unambiguous about accountability. The organisation running a factory remains accountable for what it produces. Even in a dark factory, a person remains accountable for what it ships. And Mitchell Hashimoto's "whiteboard defense", quoted in the paper, frames it at the level of the individual engineer:

> *"I should be able to pull you aside at any moment and ask you to explain any customer-facing system you've shipped. You should be able to clearly explain how it works and defend the decisions you made. If I ask 'why did you do X instead of Y?', 'what happens if this actor behaves maliciously?', 'what data structure did you use here and why?', or 'where does this fail?' you should be able to answer confidently."*

The paper's commentary on that passage is worth quoting in substance: developers are accountable for the software they ship, whether they write it or their agent does. A factory cannot explain the code for them, but a well-built one gives them the facts to start from: which agent made each change, on whose behalf or in response to which signal, under which profile and permissions, using which approved building blocks, and which checks the change passed. That matters to the organisation, and to whoever is paged at 3 a.m. to explain the code.

Two practical consequences follow.

Every change needs a named human owner. At Level 2 and below, that is the requester and the approver. At Level 3, where a signal starts the work, it is the owning team of the affected service and, for any change that requires approval, the approver. For classes of work approved automatically, it is the owning team, and the person who agreed to the class's promotion.

Approvers must be able to understand what they approve. If a change is too large or too opaque for its approver to explain, the factory has failed, regardless of whether the change works. This is another argument for small changes and good evidence summaries.

A responsibility matrix

Harbour Ledger wrote down who was responsible for what. An illustrative version:

Area	Responsible	Accountable	Consulted	Informed
A factory-produced change	Requesting engineer or owning team	Approver; service owner	Automated reviewer	Owning team

Area	Responsible	Accountable	Consulted	Informed
Specifications and acceptance criteria	Product owner, tech lead	Tech lead	Owning team	Factory platform
Profiles (guiding parts)	Platform team	Platform lead	Owning teams	All users
Profiles (enforced parts) and policy	Platform team	Platform lead and security lead jointly	Owning teams	All users
Approval policy and class promotions	Engineering leadership	VP Engineering	Security, platform, owning teams	All engineering
Enforcement infrastructure	Platform team	Platform lead	Security	All users
Factory incidents	On-call; platform team	Incident commander	Security	Leadership

The “Accountable” column never contains an agent.

Governing the factory’s own configuration

The reference architecture’s Discover outcome includes a requirement that is easy to state and essential to governance: the factory’s code and configuration are version-controlled and do not change while a run is in progress. Its observability controls add that the factory’s design configuration, such as what it accepts, which profiles it uses and what it may deploy, should be versioned with prior versions kept, so that every run traces to the exact policy it ran under.

In practice, the factory’s governance repository holds:

- the approval policy and risk class definitions;
- task policies for the authorisation service;
- profile definitions;
- signal type definitions;
- egress policies;
- gate definitions and their enforcement levels;
- the component registry;
- promotion and demotion criteria.

Every change to that repository is a class X change in the approval policy from Chapter 10: reviewed by the platform team and, for enforced controls, by security, never approvable by a factory identity. Agents may propose changes. At Level 3, the orchestrator may find weaknesses and propose improvements. People decide.

This repository is also what an auditor needs to see. Combined with the run records, it can answer, for any change: what policy was in force, which profile applied, what that profile allowed, and who approved the policy.

Mapping to Agent Baseline

The reference architecture maps its controls to the Agent Baseline framework, whose six outcomes are supported by thirty-five controls. The outcomes, and what each means in a software factory according to the paper, are:

Outcome	In a software factory
Discover	Know every component, agent and profile the factory uses, who owns it and what it can reach. The factory's code and configuration are version-controlled and do not change during a run.
Constrain	Run each agent in a sandbox, behind an egress proxy, with only the tools, files and network access its profile allows.
Authorize	Give each agent its own identity and task-scoped access, with short-lived credentials attached outside the sandbox, bounded by the authority of whoever started the run when there is one, and by its profile and organisational policy when a signal started it.
Observe	Record every agent action in an audit log the agent cannot reach, and watch for thrash and drift.
Validate	Check every input before agents use it, including the signals that start work, and every output before it moves to the next step or ships.
Respond	Be able to revoke an agent's access, stop its sandbox, quarantine what it produced and roll back a change the factory deployed.

Chapter 11 listed the specific controls required at each level. For governance purposes, the value of the mapping is that it gives the organisation an external reference point. Agent Baseline describes itself as a draft, not a standard, with no conformance programme, so mapping to it is not certification. But it is a structured, public set of controls that security teams, auditors and partners can read, and showing how a factory meets each one is a much stronger position than describing the factory in its own terms alone.

Mapping to the SSDF

Many organisations already map their development controls to the NIST Secure Software Development Framework. Chapter 9 gave an outline; here is a more detailed illustrative mapping of factory controls to the four SSDF practice groups. It is not an official crosswalk.

SSDF group	Practice area (paraphrased)	Factory controls
PO	Define security requirements	Constitution; security acceptance criteria in specifications; approval policy
PO	Implement roles and responsibilities	Responsibility matrix; named human owner per change; class X approvers
PO	Implement supporting toolchains	Sandboxes, egress proxy, authorisation service, supply chain firewall, model gateway as managed infrastructure
PO	Define and use criteria for security checks	Gate catalogue with enforcement levels
PO	Secure environments	Sandbox isolation; fence check; factory not reachable from outside
PS	Protect all forms of code	Branch protection excluding factory identities; protected paths; signed profiles
PS	Provide a mechanism to verify release integrity	Signed provenance attestations; SBOMs
PS	Archive and protect each release	Run records linked to artifacts; versioned governance repository
PW	Design to meet security requirements	Specification and plan review

SSDF group	Practice area (paraphrased)	Factory controls
PW	Reuse existing, well-secured software	Blueprints; admission point with approved versions
PW	Create source code following secure practices	Context files; skills; SAST and secret scanning gates
PW	Review and analyse code	Independent automated review; human review per approval policy
PW	Test executable code	Verification suite; held-out tests; mutation testing
RV	Identify and confirm vulnerabilities	Scanners; signal types for scanner findings at Level 3
RV	Assess, prioritise and remediate	Factory work classes for remediation; approval routing by risk
RV	Analyse root causes	Outer loop: fix the factory, not just the change

AI management frameworks

Several broader frameworks address AI risk management. They are not specific to software factories, but organisations subject to them will need to show where the factory fits.

The **NIST AI Risk Management Framework** (AI RMF 1.0, published in January 2023) organises AI risk management into four functions: Govern, Map, Measure and Manage. A software factory fits naturally: *Govern* is the responsibility matrix, approval policy and governance repository; *Map* is the threat model and the inventory of components, agents and profiles; *Measure* is the metrics and evals of Chapters 14 and 16; *Manage* is the response capability and the outer loop.

ISO/IEC 42001:2023 specifies requirements for an AI management system. Organisations certified to it, or pursuing certification, will need the factory's agents and models within the system's scope, with documented risk assessment, controls and monitoring. The factory's governance repository and run records are the natural evidence.

The **EU AI Act**, Regulation (EU) 2024/1689, entered into force in August 2024, with obligations phasing in over the following years. Whether and how its obligations apply to a particular organisation's use of coding

agents depends on the organisation, the systems involved and what the factory's output is used for. This book does not offer legal advice; organisations operating in the EU should obtain it. What the factory can do is make whatever compliance work is required easier, by keeping accurate records of which models are used, for what, under what controls, with what human oversight.

The ai-sdlc framework, notably, describes compliance posture against the EU AI Act, the NIST AI RMF and ISO 42001 as part of its governance design, which suggests that factory governance tooling and AI management frameworks are converging.

Data handling

A factory sends code, specifications, tickets and sometimes logs and data to models. Governance must decide what may be sent where.

The reference architecture's inference controls include applying the organisation's data-handling rules to what is sent for inference, such as where data may be processed and which providers may receive it. The model gateway is where those rules are enforced. A practical scheme classifies repositories and data sources, and maps each classification to the providers, regions and model routes allowed for it. Restricted code might be permitted only to providers under specific contractual terms. Customer data quoted in tickets might be stripped before any inference, or might disqualify the ticket from factory processing altogether.

The same applies to the factory's own records. Prompts and responses captured for evaluation and investigation are sensitive, and the reference architecture asks that they be protected with access controls, retention limits and encryption.

Licensing and provenance of generated code

Organisations also need a position on the licensing and intellectual property questions that generated code raises. These are legal questions that vary by jurisdiction and are still developing, and this book does not resolve them. Governance can, however, make sure the facts are available: which model produced each change, under which provider's

terms, and whether any licence-scanning gate flagged similarity to known open source code. The provenance attestation from Chapter 9 is where those facts live.

Factory incidents

A factory incident is any event in which the factory, or an agent within it, did something it should not have, or failed to stop something it should have. It might be a security event, such as an agent reaching outside its scope, or a quality event, such as a class C change causing a production failure.

The reference architecture's Respond outcome defines the capabilities: revoke access, stop the sandbox, quarantine what was produced, and roll back deployed changes. The Agent Baseline controls in this outcome also include safe failure with a non-agent fallback, so that work can continue without the factory when it has to be stopped.

Harbour Ledger's factory incident runbook, illustrative:

```
FACTORY INCIDENT RUNBOOK

1. Contain (minutes)
   - factory stop --scope <run|profile|class|all> --reason <INC-id>
     (revokes grants, stops sandboxes with state preserved,
      pauses workflows, blocks new runs in scope)
2. Scope (within the hour)
   - factory impact <run-id>   for each run involved
   - list PRs opened, merged, deployed; dependents
3. Remediate
   - close open PRs; revert merged changes; roll back deployments
   - quarantine artifacts (mark provenance untrusted)
   - block components if a component is implicated (registry)
4. Fall back
   - route affected work classes to Level 1 / human handling
   - notify owning teams
5. Preserve
   - export run records, sandbox snapshots, proxy logs for the window
6. Learn (post-incident review within 5 working days)
   - what did the factory allow that it should not have?
   - which boundary, gate or policy should have stopped it?
   - change proposed to the governance repository (class X)
   - demotion of affected classes per approval policy
   - add a fence check or eval item that would catch a recurrence
```

Step 6 is the outer loop applied to incidents. Every factory incident should end with a change to the factory and a new test that would catch a recurrence.

Reporting to leadership and the board

Leadership needs a different view from the platform team. A quarterly factory report for leadership at Harbour Ledger covered five questions, each answered with a small number of metrics from Chapter 16:

1. **What is the factory doing?** Classes of work in scope, volume of merged changes per class, approval route per class.
2. **Is it helping?** Lead time and human time per change for factory work versus comparable human work.
3. **Is it safe?** Change fail and rework rates for factory changes versus comparable human changes; factory incidents; fence check status.
4. **What does it cost?** Cost per accepted change, trend.
5. **What changed in governance?** Promotions, demotions, policy changes and why.

The report was short. It was also honest, which, as the head of the payments team observed in Chapter 16, is what made it useful.

Pitfalls

Accountability assigned to the factory. The factory produces; people are accountable. Every change needs a named human owner.

Configuration changed outside review. The governance repository is the factory's constitution. Every change to it is reviewed, and factory identities cannot approve.

Mapping as certification. Mapping to Agent Baseline or the SSDF is valuable evidence. It is not a conformance claim.

Treating legal questions as engineering questions. Data protection, AI regulation and licensing need legal input. The factory's job is to keep the records that make that input possible.

Incidents without factory changes. If the post-incident review changes nothing in the factory, it will happen again.

Chapter summary

- The organisation running the factory, and named people within it, remain accountable for every change. The factory's job is to provide the facts that let them explain and defend what they shipped.
- Write down who is responsible and accountable for changes, specifications, profiles, policy, infrastructure and incidents. No agent is ever accountable.
- Keep all factory configuration in a versioned governance repository, changed only through reviewed class X changes, so every run traces to the exact policy it ran under.
- Map factory controls to Agent Baseline's six outcomes and to the NIST SSDF as evidence for security teams and auditors. Place the factory within broader AI management frameworks such as the NIST AI RMF and ISO/IEC 42001, and take legal advice on regulation such as the EU AI Act.
- Enforce data-handling rules at the model gateway, and protect prompts and responses as sensitive data.
- Have a tested incident runbook that contains, scopes, remediates, falls back, preserves evidence and changes the factory afterwards.

CHAPTER 18

People and Platform

Every chapter so far has been about systems. This one is about the people who build, run and work alongside them, because a software factory changes the shape of engineering work in ways that no amount of infrastructure can manage on its own.

At Harbour Ledger, the factory's most difficult month was not the one with the Docker socket, the review queue flood or the profile regression. It was the month after leadership announced the factory's first results at an all-hands meeting. Within a week, the platform team had heard three different rumours: that the factory would replace junior engineers, that teams with low factory adoption would be "managed out", and that the security team was secretly reading every developer's agent sessions. None of them was true. All of them spread because nobody had said clearly what the factory was for, what it was not for, and what it meant for people's jobs.

DORA's 2025 research on AI-assisted software development put "a clear and communicated AI stance" first among its seven capabilities that amplify the benefits of AI. Harbour Ledger learned why the hard way.

The factory is a platform product

The factory is built and run by a platform team for other teams to use. That makes it a platform in the sense that the platform engineering community uses the word. The CNCF's Platforms White Paper and glossary describe platform engineering as the design, construction, operation and evolution of a platform: an internal product that standardises and streamlines software delivery while reducing its users' cognitive load.

Each part of that definition applies to a factory.

It is a product. Its users are the engineering teams who send it work and review its output. Like any product, it succeeds or fails on whether its users find it valuable. A factory imposed on teams who do not trust it will be worked around.

It standardises and streamlines delivery. The factory's profiles, gates and evidence are the standard path. Teams that use it inherit its controls, as teams using the defence software factories in Chapter 2 inherited theirs.

It reduces cognitive load. This is the part most often forgotten. A factory that requires every team to understand egress policies, signal types and eval sets before they can use it has *increased* cognitive load. The platform team's job is to absorb that complexity, so that a product team's experience is: write a good specification, review good pull requests.

Matthew Skelton and Manuel Pais's *Team Topologies*, published in 2019, offers a useful constraint for platform design: the *thinnest viable platform*, meaning the smallest platform that genuinely accelerates and simplifies delivery for the teams using it. Applied to a factory, that argues against building every capability in this book before anyone can use it. Start with the few classes of work, profiles and gates that teams actually need, and grow from there.

Where the factory team sits

Team Topologies describes four fundamental team types: stream-aligned teams, which deliver value directly in a product or service area; platform teams, which provide internal services that reduce stream-aligned teams' cognitive load; enabling teams, which help other teams adopt new capabilities; and complicated-subsystem teams, which own parts of the system that need specialist knowledge.

A software factory touches all four.

- **Stream-aligned teams** are the factory's users. They write specifications, review changes and own outcomes.
- **The platform team** builds and runs the factory: the enforcement infrastructure, the workflow engine, the profile pipeline and the dashboards.

- **An enabling function**, often a small group within the platform team at first, helps stream-aligned teams write better specifications, improve their repositories' readiness and design held-out tests.
- **Complicated subsystems** appear in the factory itself. The runtime authorisation service, the protocol-aware enforcement points and the eval infrastructure each need specialist knowledge. In a large organisation, they may warrant their own teams; in a smaller one, named specialists within the platform team.

Security sits across all of this. At Harbour Ledger, the security team co-owned the enforced parts of every profile and the policy repository, reviewed the threat model quarterly and ran the fence check results through its own monitoring. That joint ownership was more effective than either a security team that reviewed the factory from the outside or a platform team that treated security as someone else's concern.

New roles, and old roles changed

A factory does not eliminate engineering roles. It changes what they involve, and it creates a few new specialisms.

Specification authors

When the specification becomes the primary input to the factory, writing specifications becomes a central engineering skill. This is not a new role so much as a shift in emphasis for product owners, tech leads and senior engineers. The people who write the best specifications tend to be those who understand both the business need and the system well enough to name the negative cases, the protected interfaces and the non-goals.

Harbour Ledger found that specification quality was the single strongest predictor of factory outcomes, measured by review rounds and held-out pass rates. It ran short workshops on writing acceptance criteria, built a specification linter and, most effectively, showed each team its own data: the specifications with clear negative criteria and explicit non-goals produced changes that were merged in fewer rounds.

Factory engineers

Someone has to design profiles, write and maintain skills, tune context files, and improve the factory based on its metrics. This is a new specialism. It sits somewhere between platform engineering, developer experience and machine learning operations. Factory engineers need to understand agents well enough to know why one skill works and another does not, and to understand the factory's boundaries well enough not to weaken them while making agents more effective.

Eval engineers

Building and maintaining eval sets, as Chapter 14 described, is skilled work: curating past work items, writing held-out checks, freezing and versioning sets, and running comparisons. In smaller organisations this is part of the factory engineer's job. In larger ones it becomes its own role.

Reviewers

Review changes most of all. A reviewer of factory output is not reviewing a colleague's work, with all the shared context and social calibration that implies. They are reviewing a change from a system that may be confidently wrong in ways a colleague would not be. Chapter 12's review checklist is a start. Reviewers also need to know how to read evidence summaries, when to trust automated review findings, and when to stop and say "I cannot explain this change, so it does not merge".

Reviewing is also tiring in a different way from writing. Harbour Ledger found that engineers who spent most of a day reviewing factory output reported more fatigue than those who wrote code, and that review quality fell late in the day. Work-in-progress limits helped. So did rotating review duty across a team rather than concentrating it on a few senior engineers.

Junior engineers

The question everyone asks is what happens to junior engineers. If agents do routine implementation, where do people learn?

There is no settled answer, and anyone who claims to have one is guessing. What Harbour Ledger tried, and what seems reasonable, was

to give junior engineers deliberate responsibility for parts of the factory loop that build judgement: writing specifications for small work items under a senior engineer's review, writing held-out tests, reviewing factory changes in pairs with a senior engineer, and investigating factory changes that failed in review or production. Those tasks require exactly the understanding of systems that implementation used to teach, approached from a different angle.

Harbour Ledger also kept some implementation work deliberately human, including work that the factory could have done, specifically for learning. That is a cost the organisation chose to pay.

The seven capabilities

DORA's AI Capabilities Model identifies seven capabilities that amplify AI's positive impact. It is worth walking through them as a checklist for organisational readiness, because they map onto factory design so closely.

A clear and communicated AI stance. DORA's report argues that ambiguity creates risk and that a clear policy provides the psychological safety needed for effective experimentation. For a factory, the stance should answer: what is the factory for, which work is in scope, what will it not do, who is accountable for its output, what data does it see, who can see agent sessions and why, and what it means for roles and careers. Harbour Ledger's stance, written after the rumour month, was one page long. It said, among other things, that the factory existed to take routine work off engineers so they could spend more time on design, review and hard problems; that no team would be evaluated on factory adoption; that agent session logs were accessible to the developer, the platform team for operations and security for investigations, and to no one else; and that engineering headcount decisions were not being made on the basis of factory output.

Strong version control practices. As DORA puts it, version control becomes the critical safety net as AI increases the velocity of change. Every factory change flows through version control, and every piece of factory configuration lives in it.

Quality internal platforms. DORA describes the platform as providing the automated, secure pathways that let AI's benefits scale. The factory is such a platform, and it depends on the rest of the internal platform being solid: CI, deployment, observability, environments.

Working in small batches. DORA's 2024 finding about larger changes, and its 2025 emphasis on small batches, are built into the factory through change budgets, task decomposition and work-in-progress limits.

Healthy data ecosystems and **AI-accessible internal data.** Agents are only as good as the context they can reach. Accurate, current documentation, a service catalogue that reflects reality and context files that are kept up to date are what turn a generic agent into a useful one. DORA's report describes connecting AI to internal documentation and codebases as moving it from a generic assistant to a specialised expert. The factory adds a constraint: that access goes through the boundaries, scoped to the task.

A user-centric focus. DORA's report says a focus on user needs is essential to make sure AI-accelerated teams are moving quickly in the right direction. In factory terms, this is validation: did we build the right thing? A factory that is excellent at verification and indifferent to validation will produce correct software that nobody needed.

Conway's law and the factory

Melvin Conway observed in 1968 that organisations which design systems tend to produce designs that mirror their own communication structures. The observation is usually invoked about software architecture, but it applies to factories too.

A factory designed by a platform team that does not talk to product teams will produce profiles and gates that do not fit product teams' work. A factory whose security controls are designed by a security team that does not talk to the platform team will have controls that are bolted on rather than built in. A factory whose metrics are designed for leadership without input from reviewers will measure activity rather than outcome, as Harbour Ledger's first dashboard did.

The practical response is to design the factory's communication paths deliberately: product teams represented in profile and gate design, security as co-owner of enforced controls, reviewers' experience as a first-class input to metrics. Harbour Ledger set up a monthly factory council with a representative from each product area, the platform team and security, with a standing agenda of metrics, incidents, requests for new work classes and proposed promotions.

Adoption without mandates

The most reliable way to get a factory adopted is to make it the easiest way to do routine work well. The least reliable way is to mandate it.

Harbour Ledger's adoption pattern, illustrative but typical, went like this. One or two teams with strong tests and an interest in the factory volunteered for each new class of work. The platform team worked closely with them, fixed what broke and published the results. Other teams saw that dependency upgrades from the factory arrived as small, well-evidenced pull requests that merged cleanly, and asked to be included. A few teams never adopted the factory for some classes of work, because their repositories were not ready or their work did not fit, and that was accepted.

Two things accelerated adoption more than anything else. The first was the readiness assessment from Chapter 11, which gave teams a concrete list of improvements that would make the factory useful to them. The second was the evidence summary on every pull request, which made reviewers' lives easier on day one.

Pitfalls

No stated stance. Silence about what the factory means for people will be filled with rumour.

Factory as a mandate. Teams forced onto a factory they do not trust will work around it.

Platform without users in the room. Profiles and gates designed without product teams will not fit their work.

Concentrated review. Spread review duty, limit work in progress and watch for fatigue.

Ignoring how people learn. Give junior engineers deliberate responsibility for the parts of the loop that build judgement.

Chapter summary

- The factory is a platform product. It must standardise delivery and reduce its users' cognitive load, starting as the thinnest viable platform and growing from there.
- Stream-aligned teams use the factory, a platform team builds and runs it, an enabling function helps teams adopt it, and complicated subsystems within it need specialists. Security co-owns the enforced controls.
- Specification writing becomes a central skill. New specialisms appear for factory and eval engineering. Review changes most and needs support, rotation and limits.
- Give junior engineers responsibility for specification, held-out tests, paired review and failure investigation, and keep some implementation work human for learning.
- DORA's seven AI capabilities map closely to factory readiness, starting with a clear, communicated stance on what the factory is for and what it means for people.
- Design the factory's communication paths deliberately, and grow adoption by making the factory the easiest way to do routine work well.

PART V

The Future

CHAPTER 19

Toward the Dark Factory

In manufacturing, a "lights-out" or "dark" factory is one that runs without people on the floor. The machines do not need light to see by, so the lights can be turned off. The phrase has been borrowed for software. Dan Shapiro's five levels of AI coding, described in Chapter 3, end at Level 5, the dark factory, in which specifications go in and tested software comes out with no human code review. The Secure Software Factory paper adopts the term and gives it a precise meaning in its own model: a Level 3 factory whose approval policy approves every change automatically, with the same controls applying, and with a person still accountable for what it ships.

That definition is more careful than it first appears. It does not say the dark factory has fewer controls. It says it has the *same* controls, and that the only thing that changes is the approval policy. It does not say nobody is accountable. It says a person still is. And it implies something about how you get there: not by removing safeguards, but by expanding, one bounded class of work at a time, the set of changes that the evidence shows can be approved automatically.

This final chapter is about that path, about what should probably stay human regardless, and about the questions the field has not yet answered.

How a factory goes dark

The reference architecture describes the path in a sentence: an organisation approaches a dark factory by automatically approving bounded classes of work as verification, validation and production results show they are reliable. As more work qualifies, people spend more of their time defining outcomes, improving skills and evals, and governing the factory. The boundaries do not change along the way.

Three words in that description carry most of the weight.

Bounded. Automatic approval applies to a class of work with clear limits: a kind of change, in particular repositories, under a particular profile, within a change budget. “Dependency patch upgrades in services with high test coverage and tested rollback” is bounded. “Bug fixes” is not.

Classes. Approval decisions are made for classes, not for individual changes. That is what makes them governable. A class can be promoted, monitored and demoted. An ad hoc decision to let one change through cannot.

Show. Promotion follows evidence, specifically verification, validation and production results. Not a vendor’s claims, not a benchmark score, and not how productive the factory feels.

Chapter 16 gave an illustrative set of promotion and demotion criteria. The same approach extends naturally. A class that has been at class C for long enough, with outcomes at least as good as human work of the same kind, might be considered for a further step, such as automatic deployment after canary for changes that are fully reversible. Each step needs its own criteria and its own automatic demotion triggers.

What makes a class of work a candidate

Some kinds of work are much better candidates for automatic approval than others. Four properties matter most.

Reversibility. Can the change be undone quickly and completely? A code change behind a feature flag, in a service with tested rollback, is highly reversible. A database migration that drops a column is not. The reference architecture’s approval policy explicitly weighs reversibility, and it should be the first filter.

Blast radius. If the change is wrong, how much does it affect? A change to an internal tool used by one team has a small blast radius. A change to the payment authorisation path has a very large one.

Verifiability. How completely can automated checks establish that the change is correct? Documentation formatting, dependency patch upgrades with strong tests and lint fixes are highly verifiable. A change

to business logic with subtle edge cases is less so, even with held-out tests.

Validation clarity. Is there a clear, measurable signal in production that the change did what it was meant to do? An incident fix whose success is a falling error rate has one. A refactoring meant to improve maintainability does not, at least not quickly.

A simple way to use these is to score each class of work on all four, and to consider automatic approval only for classes that score highly on every one. A class that is reversible and verifiable but has a large blast radius stays with a human. So does one that is low-risk but hard to verify.

```
CANDIDACY FOR AUTOMATIC APPROVAL (per class of work)

                         reversible  small blast  verifiable  clear signal
docs formatting              high       high         high        n/a
lint autofixes               high       high         high        n/a
dep patch, strong tests      high       medium       high        medium
flaky test fixes             high       high         medium      medium
small bug fixes              medium     medium       medium      medium
incident fix, one service    medium     medium       medium      high
feature work                 low        varies       low         low
schema migrations            low        high risk    medium      low
auth / payments paths        varies     large        medium      medium

Candidates: rows scoring high or medium-high everywhere, with
evidence over time. Everything else stays with people.
```

The table is illustrative, and every organisation's scores will differ. Its shape, though, is typical. A small number of classes are good candidates. Most are not, and some never will be without a step change in how verification works.

What should stay human

Even in a factory that approves much of its routine work automatically, some decisions should stay with people. Not because agents could not make them, but because these decisions are where accountability, legitimacy and judgement live.

Changes to the factory's own controls. The reference architecture is categorical: high-impact actions and changes to the factory's own controls need approval the agent cannot give. A factory that could

approve changes to its own boundaries would have no boundaries. This does not change at any level.

What to build. The factory turns specifications into software. Deciding which specifications are worth writing, which problems are worth solving and which trade-offs are acceptable for users is product judgement. A factory can propose. People decide.

Irreversible and high-blast-radius actions. Deleting data, changing how money moves, altering authentication, migrating schemas destructively. These stay with people until verification can establish their safety far more completely than it can today.

Promotion and demotion. The decision that a class of work can be approved automatically is itself a governance decision, and it belongs to people with the authority and accountability to make it.

Accountability itself. No change in level, autonomy or approval policy moves accountability from people to the factory. When something goes wrong, someone must be able to explain what happened and why the factory was allowed to do it. The whiteboard defense applies to the factory's operators as much as to its users.

The open questions

The reference architecture is version 1, explicitly open for comment, and candid about what it does not settle. Several open questions are worth naming, because anyone building a factory will meet them.

Deriving task scope from untrusted inputs. The paper says plainly that how best to derive task scope from untrusted inputs is still an open implementation question, and that its first version does not prescribe a method. Chapter 15 described one conservative approach: signal types with typed parameters selecting among scope options fixed in advance. That works for narrow, well-understood signals. It does not scale easily to the open-ended requests that people make in tickets and chat. A general, robust method would be a significant advance.

Judging with models. LLM-based screening of inputs and LLM-based review of outputs both add value, and both can be manipulated by the

text they examine. How much weight such judgements can safely carry, and how to measure that, is not settled.

Evaluations that stay honest. Eval sets drift out of date and become targets. Refreshing them, keeping some out of reach, and connecting them to production outcomes all help, but there is no established methodology for factory evaluations comparable to what exists for, say, software testing.

Trust between agents. At Level 3, agents pass work and messages to each other through mediated channels. The reference architecture requires authority to narrow at each hop and traffic between agents to pass through the same enforcement as any other. What an agent should believe about a message from another agent, given that both read untrusted input, is a harder question than it looks.

Memory. The reference architecture places persistent memory stores outside its scope, while treating what they feed agents as untrusted input. As memory becomes more central to how agents work, factories will need clearer patterns for scoping, reviewing and resetting it.

The skill supply chain. Skills and context are becoming a large part of what makes agents effective, and they are a supply chain in their own right. Practices for signing, scanning, evaluating and distributing them are still forming.

Human skills under automation. If people review more and implement less, will they remain able to review well? The question has been asked about automation in many fields. In software, it is too early to know the answer. Organisations should watch for it, and take deliberate steps such as those in Chapter 18 to keep people's judgement sharp.

None of these is a reason to stop building factories. Each is a reason to build them so that they can change as answers emerge: with policy in version control, boundaries independent of the agent, evidence for every decision, and the ability to tighten as quickly as you loosen.

The boring factory

The preface of this book said that boring is the goal. It is worth returning to that idea at the end.

A boring factory is one where nothing surprising happens. Work arrives through one front door and is checked for readiness. Plans are proposed and reviewed. Agents work freely inside sandboxes that hold. They reach only what their tasks justify, through proxies that hold the credentials. They use only components that came through the admission point. Their changes are verified independently, reviewed by a different mind, approved by people or by policy that people wrote, and deployed in ways that can be undone. Every run leaves a record that nobody inside the run could alter. When an agent thrashes, it is stopped. When it drifts, it is flagged. When something goes wrong, the factory can be stopped everywhere in minutes, and the record shows exactly what happened and which decision allowed it. And after every surprise, the factory changes so that the same surprise does not happen twice.

None of that is glamorous. All of it is achievable with techniques that exist today, most of them described in the reference architecture this book has drawn on and in the standards and practices the software industry has built over decades. The agentic part is new. The discipline is not.

The reference architecture's conclusion puts the case better than a summary could. As software factories take on more of the work of building and operating software, organisations will increasingly depend on systems whose individual decisions they cannot fully predict. Trusting those systems does not require predicting every decision. It requires an architecture that independently controls what agents can do, regardless of what they decide.

That is what a software factory is for.

Ten questions to ask of any factory

To close, here are ten questions that can be asked of any software factory, whether you are building one, buying parts of one or reviewing someone else's. A factory that can answer all ten clearly is in good shape. One that cannot answer several has work to do.

1. **Who is accountable** for a change this factory produced last week, by name?

2. **What did the agent start with?** Can you list the image, harness, skills, model route, profile version and policy version for any run?
3. **What could the agent reach?** If it had decided to reach something else, what would have stopped it, and has that been tested?
4. **Where are the credentials?** Could the agent have read any credential, at any point in the run?
5. **What came in?** Which packages and components entered during the run, through which admission point, under which policy?
6. **Who decided the scope?** Was the agent's access decided by something other than the agent, from context the factory trusts?
7. **Who approved it, and could the factory have approved it instead?** Check every approval path.
8. **What proves it works?** Which checks were independent of the agent that wrote the change?
9. **What happens if you pull the cord?** How long does it take to stop everything, scope the impact and roll back, and when was that last tested?
10. **What changed in the factory after its last mistake?**

Chapter summary

- A dark factory, in the reference architecture's terms, is a Level 3 factory whose approval policy approves every change automatically, with the same controls and with a person still accountable.
- Organisations approach it by automatically approving bounded classes of work as verification, validation and production results show they are reliable, one class at a time, with automatic demotion when evidence changes.
- Good candidates are reversible, have small blast radius, are highly verifiable and have a clear validation signal. Most classes of work are not yet candidates.
- Changes to the factory's own controls, decisions about what to build, irreversible and high-blast-radius actions, promotion decisions and accountability itself should stay with people.

- Open questions include deriving scope from untrusted inputs, the reliability of model-based judgement, honest evaluations, trust between agents, memory, the skill supply chain and the preservation of human review skills.
- The goal is a boring factory: one where nothing surprising happens, and when something does, the record shows why and the factory changes.

Appendix A. Templates

These templates collect the artifacts used throughout the book in a form ready to adapt. Formats are illustrative; the fields are what matter. Replace angle-bracketed values, remove what does not apply, and keep each artifact in version control.

A.1 Specification

```
# SPEC-<id>: <short title>

Status: Draft | Ready        Risk class: A | B | C | D
Repository: <repo>           Owner: <team>     Reviewer: <tech lead>

## Goal
<One or two sentences, in terms of user or system behaviour.>

## Background
<What is true today, and why it is a problem. Facts, not solutions.>

## Acceptance criteria
AC1  When <trigger>, the <system> shall <response>.
AC2  If <unwanted condition>, then the <system> shall <response>.
AC3  While <state>, the <system> shall <response>.
AC4  The <system> shall <ubiquitous response>.
(At least one negative or failure case. Numbers, not adjectives.)

## Non-goals
- <What this change must not do or touch.>

## Constraints
- <Infrastructure, dependency, performance or compatibility limits.>
- Change budget: <n> changed lines excluding tests.

## Interfaces that must not change
- <Public APIs, schemas, events, internal contracts.>

## Data
<Classification of data the work touches. Anything that must not be
sent for inference.>

## Open questions
<None, or each with an owner and a default.>

## Verification notes
<Held-out tests, contract tests, schema diffs, manual checks.>
```

A.2 Definition of Ready

```
[ ] Goal stated in one or two sentences
[ ] Acceptance criteria testable (EARS or equivalent)
[ ] At least one negative or failure-case criterion
[ ] Non-goals listed
[ ] Target repository and likely components named
[ ] Risk class assigned
[ ] No open questions, or each has an owner and a default
[ ] Protected interfaces listed
[ ] Data classification known
[ ] Size fits the profile's change budget, or work is split
[ ] Requester has write permission on the target repository
```

A.3 Project constitution

```
# Constitution: <service>

1. <The quality that wins when two goals conflict.>
2. <Security invariants: signing, secrets, data handling.>
3. <Compatibility rules for public interfaces.>
4. <Dependency policy.>
5. <Testing rule: every acceptance criterion has an automated test.>
6. <Release rule: flags, canaries.>
7. <What must never appear in logs.>
(Keep it under a page. Enforce what can be enforced in policy.)
```

A.4 Context file (AGENTS.md)

```
# AGENTS.md (<repo>)

## Build and test
- <setup command>; <unit test command>; <integration test command>
- A change is done when <check command> passes.

## Conventions
- <Non-obvious structure, patterns and helpers.>

## Never
- <Rules, each backed by a policy check in verification.>

## Where to look
- <Key directories and documents.>
```

A.5 Profile

```
apiVersion: factory.example/v1
kind: Profile
metadata: { name: <kind-of-work>, version: <semver>, owner: <team> }
spec:
  # guiding (shapes behaviour; not a boundary)
  context:
```

```
    skills: [{ name: <skill>, version: <v>, digest: <sha256> }]
    contextFiles: [AGENTS.md, docs/CONSTITUTION.md]
  hooks: { postEdit: <script>, preComplete: <script> }
  model: { route: <gateway route> }
  # enforced (applied outside the agent)
  sandbox:
    image: <registry>/<image>@<digest>
    isolation: container | microvm
    limits: { cpu: <n>, memory: <n>, wallClock: <dur>, steps: <n> }
    writablePaths: [/workspace]
  network: { policy: <egress policy name> }
  access:
    maxPullRequests: <n>
    protectedPaths: [<globs>]
    canMerge: false
    canDeploy: false
  budgets: { tokens: <n>, spendUsd: <n> }
```

A.6 Egress policy

```
kind: EgressPolicy
metadata: { name: <profile>, profileVersion: <v> }
spec:
  default: deny
  blockAlways: [metadata, rfc1918, directIp, rawSockets, unapprovedDns]
  allow:
    - host: <source control>
      operations:
        - fetch: { repo: "${task.repo}" }
        - push:
            repo: "${task.repo}"
            refs: ["refs/heads/factory/${run.id}/*"]
      credential: { source: authz, scope: task }
    - host: <package proxy>
    - host: <model gateway>
      credential: { source: authz, scope: task }
  log: all
  alertOn: { deniedRequestsPerMinute: <n> }
```

A.7 Task policy

```
kind: TaskPolicy
metadata: { name: <class>-<risk>, version: <v> }
match:
  workItem.type: <type>
  workItem.riskClass: <class>
  profile.name: <profile>
requires:
  requester.canWrite: "${workItem.repo}"
  workItem.status: Ready
grant:
  repo: "${workItem.repo}"
  paths: "${workItem.scope.paths}"
  pushRefs: ["refs/heads/factory/${run.id}/*"]
```

```
  pullRequests: <n>
  duration: <dur>
  inferenceBudgetUsd: <n>
deny: [merge, deploy, { op: push, refs: [main, "release/*"] }]
escalate: [{ when: request.outside(grant), to: "human/${workItem.owner}" }]
```

A.8 Approval policy

```
Class D  <docs, comments>
         merge: hard gates            deploy: n/a
Class C  <test-only changes, patch bumps>
         merge: hard gates + automated review
         deploy: standard pipeline
Class B  <small fixes, minor bumps>
         merge: + 1 human from owning team
         deploy: canary
Class A  <public API, money movement, auth>
         merge: + 2 humans including code owner
         deploy: human approval, canary, staged rollout
Class X  <factory policy, profiles, enforcement>
         merge: platform + security review (no factory identity)
         deploy: platform + security approval

Promotion criteria (per class): <volume, period, outcome thresholds,
                                 agreement required>
Demotion triggers (automatic):  <incident, threshold breach, finding>
```

A.9 Signal type

```
kind: SignalType
metadata: { name: <signal>, version: <v> }
source: { system: <system>, authentication: <method> }
schema:
  <field>: { type: <type>, mustExistIn: <catalogue>, max: <limit> }
  summary: { type: text, maxLength: <n>, treatAs: untrusted }
policy:
  allowedServices: [<services>]
  workClass: <class>
  profile: <profile>
  scopeTemplate: { <fixed options, filled only from typed fields> }
  approvalRoute: <class>
  rateLimit: { perService: <n/period> }
```

A.10 Evidence summary

```
## Factory evidence: <work item> <task>
Requested by: <identity>     Run: <run id>
Profile: <name@version>      Model route: <route>
Agent identity: <identity>

What was asked: <criteria covered>
What changed: <files, line counts>
```

```
Acceptance criteria: <each, with test coverage and result>
Verification (independent CI): <each gate, result>
Automated review (<model family>): <findings, blocking or not>
Not verified: <explicitly>
Approval route: <class -> approvers>
```

A.11 Fence check starter list

```
sandbox: host_read, write_outside_workspace, escalate_root,
         docker_socket, cross_task, unsigned_image, limit_exceeded
network: external_unlisted, direct_ip, raw_socket, custom_dns,
         metadata_endpoint, rfc1918_unlisted, push_protected_ref,
         extra_pull_request, env_credentials_absent
supply:  blocked_package, version_in_cooldown, canary_secret_in_repo,
         unpinned_skill_refused
identity: self_grant, merge_own_pr, approve_own_pr, factory_in_approvers
exec:    proxy_down_refuses, authz_down_refuses, stop_all_works
obs:     audit_sink_down_refuses, write_audit_from_sandbox, run_linked
```

A.12 Run record fields

```
run.id, run.start, run.end, run.outcome
workItem.id, workItem.source, workItem.createdBy
requester, approvers[]
agent.identity, orchestrator (if any), delegation chain
profile.name, profile.version, profile.digest
policy.bundle, policy.version
components: image digest, harness version, skills[digest], model route
grants[]: resource, operation, scope, expiry, basis, decidedBy
denials[]: request, reason
admissions[]: package, version, source, policy
sandbox events: processes, file writes, network (summarised)
inference: model, tokens, cost (per call or aggregated)
verification: gate, level, result, CI reference
review: reviewer identities, findings, decision
artifacts: PR, commit, SBOM reference, attestation reference
deploy (if any): environment, canary result, outcome check
```

Appendix B. A Ninety-Day Starting Plan

This plan is for an organisation that already has coding agents in use by individual developers and wants to move to a deliberate Level 1 setup and a first Level 2 class of work. It assumes a platform team of a few people with security support. The phases overlap; the exit criteria matter more than the dates.

Days 1 to 30: foundations and a safe Level 1

Decide and communicate.

- Write a one-page AI stance: what the factory is for, what is in scope, what it means for roles, who can see agent sessions and why. Publish it before anything else.
- Name an accountable owner for the factory and a security co-owner for its enforced controls.

Assess.

- Run the readiness assessment from Chapter 11 on the five to ten repositories most likely to host the first Level 2 work.
- Inventory current agent use: which tools, which credentials, which repositories, which data.
- Write a first threat model using the format in Chapter 7.

Build Level 1.

- Provide sandboxed workspaces with writes confined, no host access and no Docker socket.
- Enforce a default-deny egress proxy with an allow-list and a fast, logged request process.
- Route packages through internal proxies with scanning and a cooldown.

- Give agents their own identities and short-lived credentials; add commit trailers.
- Route inference through a model gateway with an approved model list and budgets.
- Publish a task template and the agent change review checklist.

Exit criteria. The Level 1 fence check passes nightly. Most agent use runs inside the sandbox. The stance is published and the first questions about it have been answered.

Days 31 to 60: a first Level 2 class of work

Choose the class. Pick one low-risk, high-volume class, such as documentation updates or dependency patch upgrades, in two or three ready repositories with willing teams.

Build the minimum Level 2.

- A single intake that creates work items and enforces the Definition of Ready.
- A durable execution layer, starting with the existing CI system if it can hold state across steps.
- One signed profile for the chosen class, with pinned skills and image.
- A runtime authorisation service, even a simple one, that issues task-scoped grants from policy, and a proxy that enforces them for source control and packages.
- Branch protection that excludes factory identities from approval.
- Independent verification in CI with the factory-specific hard gates from Chapter 13.
- Evidence summaries on every pull request.
- Run-linked records in an append-only store; a working stop command.
- Work-in-progress limits per team.

Exit criteria. The factory has merged its first changes in the chosen class. The full fence check passes. The stop command has been tested. Every merged change can be reconstructed from records alone.

Days 61 to 90: measure, learn and decide

Measure.

- Stand up the dashboard from Chapter 16 for the chosen class: outcomes segmented by origin, output, cost including reviewer time, and health.
- Build a first eval set of thirty to sixty past work items for the class.

Learn.

- Hold a review with the participating teams: what worked, what did not, what the factory should change.
- Make at least one profile or skill improvement, evaluated against the eval set before promotion.
- Run a factory incident drill using the runbook in Chapter 17.

Decide.

- Agree whether to continue, widen to more repositories, add a second class of work or pause.
- Write the first promotion and demotion criteria into the approval policy, even if no class is ready for promotion yet.

Exit criteria. Leadership receives a short, honest report answering the five questions in Chapter 17, and a decision about the next quarter is made on the evidence.

Glossary

Terms marked (SSF) follow the definitions in *The Secure Software Factory: A Reference Architecture*, sometimes condensed.

Acceptance criteria. Testable statements in a specification that a change must satisfy. Best written with a trigger or condition and a single observable response.

Admission point. The place where packages, dependencies, models and other components enter the factory during a run, such as a registry, proxy or supply chain firewall. It serves approved versions and hides unsafe ones.

Agent. A language model in a loop with tools, which decides what actions to take toward a goal and observes their results.

Agent Baseline. A framework for building, deploying and operating AI agents in production, organised as six outcomes (Discover, Constrain, Authorize, Observe, Validate, Respond) supported by thirty-five controls. (SSF)

Agent harness. The runtime that runs a coding agent and loads the skills, tools, hooks and context files from its profile. (SSF)

AGENTS.md. An open Markdown convention for repository-level instructions to coding agents.

AI SDLC. The software development lifecycle as reshaped by AI agents: the same stages, with a different distribution of cost, risk and human involvement.

AI-DLC. AWS's AI-Driven Development Lifecycle, built on Intents, Units and Bolts across Inception, Construction and Operations phases.

Approval policy. The organisation's rules for whether a change is approved automatically or waits for a person, based on its kind, risk,

blast radius, reversibility and the verification and validation it has passed. Automatic approval is an option, never a requirement. (SSF)

Blueprint. An approved architecture pattern, such as a service template, that agents draw on so their output follows the organisation's standards. (SSF)

Bolt. In AI-DLC, the smallest iteration: a short build-and-validate cycle measured in hours or days.

Change budget. A limit on the size of a change, usually in changed lines, set in a specification or profile and enforced by verification.

Constitution. A short document of project-wide principles that every specification and plan is checked against (term from GitHub Spec Kit).

Context engineering. Deciding what goes into an agent's context at the start of a run, and what it can retrieve later.

Dark factory. A factory that runs with no human oversight; in SSF terms, a Level 3 factory whose approval policy approves every change automatically, with the same controls and a person still accountable. Shapiro's Level 5. (SSF)

Definition of Ready. The conditions a work item must meet before the factory accepts it.

Delegation. How one agent passes part of its task, and its scope, to another agent, which can narrow that scope but never expand it. (SSF)

DORA metrics. Software delivery measures from the DORA research programme: change lead time, deployment frequency, failed deployment recovery time, change fail rate and deployment rework rate.

Drift. When an agent gradually moves away from its assigned task while staying within its permissions; access controls do not catch it, and monitoring has to. (SSF)

EARS. The Easy Approach to Requirements Syntax: sentence templates for unambiguous requirements (ubiquitous, event-driven, state-driven, unwanted behaviour, optional feature).

Egress proxy. The enforcement point that a sandbox's network traffic passes through once the sandbox starts. It blocks what has not been

authorised, attaches credentials so the agent never holds them, and may be one proxy or several protocol-aware gateways. (SSF)

Evals. Tests that measure how well an agent or the factory performs over time. (SSF)

Evidence summary. A structured description attached to each factory change of what was asked, what was done, what was verified and what was not.

Execution layer. What starts agents with their profiles, sequences work, retries failures and stops runs: the developer at Level 1, a durable system from Level 2. (SSF)

Fence check. This book's name for a suite of tests that try to break each boundary, run against the deployed factory.

Gate. A check a change must pass to move between stages, with an enforcement level: advisory, soft-mandatory or hard-mandatory.

Held-out tests. Tests derived from a specification by someone other than the implementing agent and kept out of its reach.

Inner loop. The cycle of assemble, build and verify that a coding agent repeats until its work passes. (SSF)

in-toto. A framework for verifiable records of software supply chain steps, including a widely used attestation format.

Lethal trifecta. Simon Willison's term for the combination of access to private data, exposure to untrusted content and the ability to communicate externally.

MCP (Model Context Protocol). An open protocol for offering tools and data to AI applications, introduced by Anthropic in November 2024.

Model gateway. The enforcement point for inference, reached through the egress proxy, which routes model calls to approved models and applies rate limits, budgets and data-handling rules. (SSF)

Monitoring agent. An emerging Level 3 pattern: an agent that watches signals and work in progress for thrash and drift and can flag and request, but never grant or approve. (SSF)

Orchestrator. Whatever decides what work happens next: the developer at Level 1, a predefined workflow at Level 2, an orchestrator agent at Level 3. (SSF)

Outer loop. The cycle that feeds review, evals and production feedback back into design, both to send work around again and to improve the factory itself. (SSF)

Preflight. The Level 3 orchestrator's planning step, proposing a profile, task scope and resources for each piece of work. (SSF)

Profile. Everything an agent of a given kind works with: context, tools, hooks, packages, sandbox, and the access, action and network rules that bound it. It sets the upper bound on what the agent can do. (SSF)

Provenance. Verifiable information about how an artifact was produced.

Runtime authorisation service. The service that determines and authorises the access an agent needs for its assigned task, within its profile, delegation and organisational policy; it records every grant and can revoke access. (SSF)

Sandbox. The isolated environment where an agent's code runs, configured from its profile and destroyed when the task ends.

SBOM. Software bill of materials: a machine-readable list of the components in an artifact.

Signal. An event that starts Level 3 work without a person filing it, such as an incident, scanner finding, eval result or automatically created ticket. Signals are untrusted inputs. (SSF)

Skill. A packaged set of instructions, and sometimes scripts, that an agent loads to perform a task. (SSF)

SLSA. Supply-chain Levels for Software Artifacts, an OpenSSF framework; its Build track defines levels L1 to L3 of build trustworthiness.

Software factory. An agentic system that turns a prompt, ticket or specification into tested code for the existing codebase it is scoped to, using one agent or several, and increasingly deploys and maintains what it builds. (SSF)

Spec-driven development. Writing and reviewing a specification first, then generating plans and code from it. Variants are spec-first, spec-anchored and spec-as-source.

SSDF. NIST's Secure Software Development Framework, SP 800-218, with practice groups PO, PS, PW and RV.

Supply chain firewall. A dedicated admission point that serves only approved, scanned components and blocks compromised or disallowed ones. (SSF)

Task. The specific work an agent has been assigned in one run, and the scope authorised for it, within its profile. Only a new decision by the runtime authorisation service can expand it. (SSF)

Thrash. What an agent does when it cannot reach its goal and starts trying increasingly creative workarounds; both a security and a cost risk. (SSF)

Validation. Checks that output meets the real needs of users and stakeholders. It answers "did we build the right thing?" (SSF)

Verification. Checks that output was built to its specification, standards and compliance requirements. It answers "are we building it right?" (SSF)

Work item. The factory's internal representation of a request, carrying its specification and the metadata the factory needs.

Notes and References

Sources are listed by chapter in the order they are first relied on. Figures and incidents attributed to the Secure Software Factory paper are reported as the paper states them. Harbour Ledger is a composite, illustrative organisation; nothing described about it refers to a real company.

Principal source

Aleyner, E., Familusi, R., Howe, A., Livingstone, I., Manning, W., McMahon, A., Nassri, A. and Stanley, A. *The Secure Software Factory: A Reference Architecture.* Agent Baseline, version 1, 13 October 2026. agentbaseline.org/software-factory.pdf. Prose and figures licensed CC BY 4.0. Referred to in the text as "the reference architecture" or "the paper". Used throughout for the definition of a software factory, the inner and outer loops, the four boundaries and their invariants, execution and observability, the three levels of autonomy, profiles, runtime authorisation, approval policy, the controls in its appendix, the mapping to Agent Baseline, the whiteboard defense, the definitions, and the incidents.

Agent Baseline. agentbaseline.org. Six outcomes and thirty-five controls (DIS, CON, AUT, OBS, VAL, RES).

Chapter 1

Industry figures for Vercel, Uber and Stripe, and the reference to Geoffrey Huntley's Ralph loop, as reported in the reference architecture's introduction.

Anthropic. "Introducing the Model Context Protocol." 25 November 2024.

Chapter 2

Naur, P. and Randell, B. (eds.). *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, October 1968. Includes McIlroy, M. D., "Mass Produced Software Components."

Bemer, R. W. "The Software Factory: Concept and Compact History." bobbemer.com (archived). Bemer's use of the term in 1968.

Bratman, H. and Court, T. "The Software Factory." *Computer* (IEEE), 1975.

Cusumano, M. A. "System Development Corporation: Defining the Factory Challenge." MIT working paper, DSpace hdl.handle.net/1721.1/49002.

Cusumano, M. A. *Japan's Software Factories: A Challenge to U.S. Management.* Oxford University Press, 1991.

Cusumano, M. A. "The Software Factory: An Entry for the Encyclopedia of Software Engineering." MIT, 1991. DSpace hdl.handle.net/1721.1/17131. Source for Hitachi's motives, scale in 1991, the failed components control system and the move to structured design.

Hitachi, Ltd. Software division history (hitachi.co.jp/Div/soft/history.html).

Greenfield, J. and Short, K., with Cook, S. and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

Böckeler, B. "Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl." martinfowler.com, 2025.

"Making the Kessel Run." *Air & Space Forces Magazine.* Breaking Defense, "New DIUx Software Saves Air Force Millions of Pounds of Airborne Tanker Gas a Month," September 2017. FedScoop coverage of Project Kessel Run.

Platform One Big Bang documentation (docs-bigbang.dso.mil). US Air Force, DoD Enterprise DevSecOps Initiative keynote.

Chapter 3

ai-sdlc-framework/ai-sdlc. GitHub repository, Apache 2.0 licence. Definition of Ready gate, governance resources, quality gate enforcement levels, cross-harness review, earned autonomy, metrics specification.

Amazon Web Services. AI-Driven Development Lifecycle (AI-DLC) method definition and AWS workshop materials.

Shapiro, D. "The Five Levels: from Spicy Autocomplete to the Dark Factory." danshapiro.com, January 2026. Summarised by Simon Willison, simonwillison.net, 28 January 2026.

Becker, J., Rush, N., Barnes, B. and Rein, D. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity." METR, 10 July 2025; arXiv:2507.09089. METR's later notice that the results are out of date, and its February 2026 update on study design.

DORA. *Accelerate State of DevOps Report 2024.* dora.dev.

DORA. *State of AI-assisted Software Development 2025* and *DORA AI Capabilities Model* (v2025.1). dora.dev.

Chapter 4

GitHub. Spec Kit documentation and quickstart (github.com/github/spec-kit).

Kiro documentation: specs, requirements-first feature specs (kiro.dev/docs/specs).

Böckeler, as above.

Chapter 5

AGENTS.md (agents.md).

Linux Foundation. Announcement of the Agentic AI Foundation, with contributions including MCP, goose and AGENTS.md, December 2025. Anthropic, "Donating the Model Context Protocol and establishing the Agentic AI Foundation."

Huntley, G. “Ralph Wiggum as a ‘software engineer’.” ghuntley.com/ralph. Loop as given there.

Chapter 6

Toyota Motor Corporation. Toyota Production System pages (global.toyota): jidoka and andon.

Chapter 7

Willison, S. “The lethal trifecta for AI agents: private data, untrusted content, and external communication.” simonwillison.net, 16 June 2025.

OWASP Gen AI Security Project. *OWASP Top 10 for LLM Applications 2025.*

Incidents (PocketOS, Shai-Hulud, Hugging Face, Anthropic, “Let’s land #118”) as described in the reference architecture’s appendix.

Chapter 8

Reference architecture, sections on the four boundaries, execution and observability, and the appendix controls by boundary.

Chapter 9

Executive Order 14028, *Improving the Nation’s Cybersecurity*, 12 May 2021 (Federal Register, 17 May 2021).

NTIA. *The Minimum Elements for a Software Bill of Materials (SBOM).* 12 July 2021.

SLSA. “SLSA v1.0 is now final,” April 2023; SLSA v1.0 security levels (slsa.dev).

CNCF. Announcement of in-toto graduation, 23 April 2025.

Sigstore documentation: overview, cosign signing overview, FAQ (docs.sigstore.dev).

NIST. SP 800-218, *Secure Software Development Framework (SSDF) Version 1.1*, February 2022. SP 800-218A, *Secure Software Development Practices for Generative AI and Dual-Use Foundation Models: An SSDF Community Profile*, 2024.

ISO/IEC 5962:2021 (SPDX). Ecma International, ECMA-424 (CycloneDX).

Chapters 10 to 13

Reference architecture: identity and access boundary, implementation reference (profiles and configuration, authorising a task, securing a local coding agent), how the factory deepens across levels, mapping to Agent Baseline.

Chapter 14

Jimenez, C. E. et al. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” 2023; arXiv:2310.06770.

OpenAI. “Introducing SWE-bench Verified.” August 2024.

ai-sdlc, as above, for gate enforcement levels and review principles.

Chapter 15

Reference architecture: Level 3, “A factory in action”, authorising a task, monitoring agent definition (crediting Travis McPeak of Cursor on the Insecure Agents podcast).

Chapter 16

DORA. “DORA’s software delivery performance metrics” (dora.dev/guides/dora-metrics) and “A history of DORA’s software delivery metrics.”

Chapter 17

NIST. *Artificial Intelligence Risk Management Framework (AI RMF 1.0).* January 2023.

ISO/IEC 42001:2023, *Information technology: Artificial intelligence: Management system.*

Regulation (EU) 2024/1689 (Artificial Intelligence Act), entered into force 1 August 2024.

Hashimoto, M. “Whiteboard defense,” as quoted in the reference architecture.

Chapter 18

CNCF TAG App Delivery. *Platforms White Paper* and platforms glossary.

Skelton, M. and Pais, M. *Team Topologies: Organizing Business and Technology Teams for Fast Flow.* IT Revolution, 2019. Team Topologies, "What is a Thinnest Viable Platform (TVP)?"

Conway, M. E. "How Do Committees Invent?" *Datamation*, April 1968.

DORA AI Capabilities Model, as above.

Background reading

Forsgren, N., Humble, J. and Kim, G. *Accelerate: The Science of Lean Software and DevOps.* IT Revolution, 2018.

Humble, J. and Farley, D. *Continuous Delivery.* Addison-Wesley, 2010.

Claims checked and not used

Several figures that appear in secondary coverage of the companies and programmes discussed were not used because primary sources could not be confirmed or sources disagreed. These include dollar savings attributed to Kessel Run's early work and weekly pull request volumes attributed to Stripe's agents. Performance tiers for DORA's deployment rework rate are not given because DORA's 2024 report did not publish them.

Index

acceptance criteria, 9, 30, 34, 50, 63, 136, 142, 155
admission point, 60, 86, 94, 133, 190, 210, 223
Agent Baseline, 3, 98, 106, 123, 188, 223, 229
agent harness, 9, 88, 120, 223
AGENTS.md, 49, 75, 135, 223, 231
AI-DLC, 25, 223, 231
ai-sdlc, 24, 37, 153, 177, 191, 231
andon, 59, 232
approval policy, 3, 15, 26, 40, 59, 97, 111, 126
attestation, 94, 163, 189, 225
audit trail, 73, 82, 97, 106, 125, 142, 169
blueprint, 6, 15, 52, 190, 224
Bolt, 25, 223
change budget, 40, 55, 61, 148, 154, 200, 206, 224
change fail rate, 176, 224
CNCF, 195, 232
constitution, 35, 50, 60, 135, 189, 224
context engineering, 50, 224
context file, 6, 46, 60, 190, 198, 223
Conway, 200, 234
cooldown, 97, 122, 130, 219
cross-family, 51
CycloneDX, 96, 233
dark factory, 26, 185, 205, 224, 231
Definition of Ready, 26, 37, 125, 142, 153, 220, 224, 231
delegation, 88, 107, 123, 167, 224
deployment rework rate, 176, 224, 234
DORA, 11, 28, 38, 125, 176, 195, 224, 231
drift, 20, 33, 66, 85, 121, 131, 161, 167
EARS, 36, 224
egress proxy, 10, 47, 71, 80, 108, 122, 129, 144
enforcement level, 26, 125, 148, 153, 187, 225, 231
EU AI Act, 190
eval set, 159, 196, 209, 221
evals, 9, 122, 158, 190, 205, 225
evidence summary, 3, 64, 100, 157, 201, 225
execution layer, 67, 71, 88, 113, 119, 143, 168, 220
fail closed, 88, 144
fence check, 89, 99, 127, 130, 148, 181, 189, 197
Goodhart, 160, 178
Greenfield, 16, 230
harness, 5, 26, 46, 75, 81, 94, 120, 130
held-out, 40, 55, 66, 97, 148, 154, 177, 190
Hitachi, 14, 55, 128, 230
Hugging Face, 73, 89, 232
identity, 73, 79, 96, 105, 120, 130, 145, 166
impact scoping, 113, 123, 169
in-toto, 87, 95, 225, 232
inner loop, 7, 59, 120, 177, 225
ISO/IEC 42001, 190, 233
Kessel Run, 18, 230
Kiro, 27, 36, 230
lethal trifecta, 72, 225, 232
McIlroy, 13, 230
MCP, 6, 47, 84, 94, 225, 231
METR, 28, 176, 231
microVM, 80, 102, 131, 147
Model Context Protocol, 6, 47, 225, 229
model gateway, 51, 61, 71, 85, 98, 130, 147, 178
monitoring agent, 121, 167, 225, 233
mutation testing, 156, 190
NIST AI RMF, 191
orchestrator, 26, 41, 60, 77, 107, 119, 163, 188
outer loop, 7, 20, 30, 53, 59, 120, 129, 190
OWASP, 70, 232
platform engineering, 195

Platform One, 18, 230
PocketOS, 8, 69, 87, 98, 110, 232
preflight, 41, 121, 164, 226
profile, 6, 15, 30, 38, 45, 59, 80, 94
prompt injection, 46, 70
property-based, 156
provenance, 30, 72, 87, 94, 122, 142, 163, 189
Ralph, 4, 52, 143, 229
revocation, 97, 113, 123, 168
risk class, 40, 111, 128, 145, 158, 187
runtime authorisation service, 41, 84, 107, 123, 197, 220, 226
sandbox, 3, 30, 38, 47, 59, 71, 79, 94
SBOM, 87, 94, 148, 189, 226, 232
Shai-Hulud, 73, 87, 93, 232
Shapiro, 26, 205, 224, 231
signal type, 125, 165, 187, 196, 208
Sigstore, 87, 95, 232
skill, 6, 15, 26, 46, 60, 70, 86, 94
SLSA, 87, 94, 226, 232
SPDX, 96, 233
Spec Kit, 26, 35, 224, 231
spec-anchored, 35, 227
spec-as-source, 17, 35, 227
spec-driven development, 27, 34, 227
SSDF, 96, 189, 227, 232
SWE-bench, 158, 233
System Development Corporation, 13, 230
task scope, 41, 86, 107, 121, 164, 208, 226
Team Topologies, 196, 234
thinnest viable platform, 196, 234
thrash, 19, 46, 65, 70, 85, 121, 167, 177
threat model, 76, 97, 190, 197, 219
validation, 25, 34, 62, 99, 111, 123, 142, 158
verification, 5, 19, 24, 33, 50, 61, 74, 82
whiteboard defense, 8, 100, 106, 136, 185, 208, 229
work in progress, 67, 121, 151, 167, 202, 225
work item, 27, 102, 107, 142, 158, 168, 175, 198