port

AI Software Factory

How to evaluate a software factory platform

The platform capabilities to check, and three assembly lines to build end to end in a proof of concept.

Table of Contents

01 What is a software factory? 3

02 What breaks when you scale a software factory? 4

03 Capabilities breakdown: what does the platform need to do?

3.1 Orchestration 5

3.1.1 Orchestration: how do you build assembly lines that fit your existing way of work? 5

3.1.2 Human in the loop: how do you stay in control of what agents do? 6

3.1.3 Agents, skills, and MCP: how do you keep track of what agents use? 7

3.2 Context: how do you give agents one reliable source of context? 8

3.2.1 Context ingestion: how to populate the context layer and keep it accurate 8

3.2.2 Context retrieval: how do agents get the context they need? 9

3.2.3 Semantic layer and enrichment: how does the context fit your organization and improve over time? 10

3.3 Control 11

3.3.1 Standards: how do you keep your software factory up to your standards? 11

3.3.2 Permissions: how do you control what each agent can see and do? 12

3.3.3 Audit: how do you know what people and agents did, and why? 13

3.3.4 Measurement: how do you know the factory is working? 14

3.4 General platform requirements 15

3.4.1 Interfaces: how do you get every team working in the same place? 15

3.4.2 Administration: how do you run the platform in your environment? 16

3.4.3 Agnostic: how do you avoid a rebuild when your stack changes? 17

04 What should you build in the POC? 18

05 What are the seven must-haves, and what happens without them? 23

1. What is a software factory?

A software factory is a system in software engineering that starts work from events and moves it through structured engineering workflows run by coordinated agents. Agents read and write the artifacts your teams already use, such as tickets, specs, pull requests, and alerts, and people approve the decisions that matter. One or even many coding agents working on many tasks is not a factory. The factory is the system around the agents: what runs the work, what context the agents get, and what keeps them in bounds.

To work efficiently, every factory needs these 3 main pillars to operate:

Pillar	What it does
Orchestration	Events from your tracker, repositories, pipelines, and alerts become work intake. Afterwards, workflows define the steps the work takes, their order, and the conditions between them. Workflows can run the same way every time or be coordinated by agents. Work inside the steps is usually done by agents, with or without humans in the loop, or defined as deterministic workflow operations.
Context	Agents get the ownership, dependencies, recent changes, and anything else relevant to the work item as context. After the agents do their work, their decisions, actions, and lessons are logged and fed back into the context, so each run starts better informed than the last.
Control	Engineering standards and automated checks decide what proceeds, access controls limit what each agent can reach, and human-in-the-loop controls keep people accountable for the decisions that matter.

A pilot needs to cover all three, but platforms differ in what happens to them at scale, which the next section covers.

2. What breaks when you scale a software factory?

Building the first line of a software factory is within reach for most teams. If you already run CI/CD, GitOps, gradual deployments, and observability, weaving agents into them takes just weeks. However, scaling it across the whole organization is the hard part, and that is where factories break. They break in four common ways, and each one needs something specific from the platform.

Pillar	The problem, and what it takes to solve it
Governance	Agents spread from the bottom up, and that is how it should work. An engineer writes a skill that helps, points an agent at a system that has an MCP server, and ships. Repeat that across a few hundred engineers and you have skills and agents nobody owns, no production-ready bar on any of them, services deployed to the wrong environment, agents running on a developer's credentials with access to every tool that grants, and no audit log of who decided what. **Governed** means the platform blocks the action instead of reporting it later, holds agents to the same standards as their output, gives every agent its own identity and its own tools, puts an owner on every agent and skill, and logs people and agents in one place.
Unique needs and constraints	You cannot automate the whole AI-SDLC on a platform you cannot extend. You may not realize it on the first line, because getting one running is straightforward. Once you start on the ones your business actually depends on, and they start to cross teams, tools, and stages, you end up with entities, sources, and steps nobody shipped for you. If the platform can't reach a certain area, a team will just build it on its own, and you end up with ten AI-SDLCs instead of one. Therefore, **flexible** means your engineers can extend the data model, add sources, and build workflows themselves. **It must cover all AI-SDLC stages,** from intake all the way through security runs.
Quality	At demo scale you can still read every agent decision. At full scale, nobody can. In addition, nothing tells you how good those decisions were. When an agent gets it wrong, the data is usually why: ownership lives in three systems, the context is scattered and stale, and the agent draws the wrong conclusion. Organized context also cuts how much the agent has to reason, which cuts the bill. **Measurable** means you keep what each agent read and decided, mark the calls that were wrong, and add the missing context back to the context layer so the next run is better.
Operational	A software factory is a living project. It needs more workflows, skills, agents, and context every week, and most of it is wanted by teams you do not sit with. Push standards down from the top and teams will find the faster path around them, because shipping is what they are measured on. Meanwhile the factory moves at the speed of its slowest queue, usually pull request review, so it runs at human speed and not agent speed. Your job is to be the platform they pick: agnostic to their tools, their models, and other vendors' agents, contributable and open to every team, so a product team can add a workflow or a skill inside your standards without waiting on you, and so can the people who never write code. Then they all work in one place, you keep control, and they go as fast as the technology allows.

3. Capabilities breakdown: what does the platform need to do?

3.1 Orchestration

3.1.1 Orchestration: how do you build assembly lines that fit your existing way of work?

Capability	What it means	Why it matters
Build multi-step workflows	You build multi-step workflows the vendor did not ship, with branching and calls to your own systems	The workflows that matter are specific to you. If you cannot build them, teams script the work elsewhere, where it is not gated, logged, or measured by the platform.
Cover every AI-SDLC stage	Intake, build, deploy, run, and security all use the same triggers, conditions, and approvals.	Any stage the platform does not cover gets automated by whoever needs it first, and you end up with several incompatible factories to govern.
Expose workflows as MCP tools	Workflows are published to agents as tools over MCP, so an agent can list the ones it is allowed to run, read the inputs each needs, and run them.	The alternative is handing agents your credentials directly, and then the limit on what they can do is whatever the token allows rather than whatever your policy says.
Build forms from context layer	Form fields are filled by the context layer and the user, with validation and fields that appear only when relevant.	Free-text fields are where people and agents pick the wrong environment, owner, or target, and those mistakes reach production.
Invoke any agent as a workflow step	A coding agent or another vendor's agent runs as a node in your workflow and returns its result to the next step.	You will change agents over time. If the agent is hard-coded into the workflow, every workflow that names it has to be edited before anyone can switch.
Retry and roll back failed runs	A run that fails part way through can be retried and rolled back.	Multi-step work fails sometimes, and a half-finished run leaves systems in a state that someone has to untangle by hand at the worst moment.
Test workflows before publishing	You can test a workflow before publishing it, and follow each run step by step with logs.	The factory changes constantly. If changes cannot be tested, one bad edit breaks everyone's work and people stop trusting the platform.
Let teams own and ship workflows	Workflows have an owning team, and teams outside the platform team can build and ship their own.	Inner sourcing the factory is what keeps it one factory. If only the platform team can add workflows, other teams build their own on the side, and those automations are covered by neither your standards nor your audit log.

3.1.2 Human in the loop: how do you stay in control of what agents do?

Capability	What it means	Why it matters
Human-input, not just approvals, as a first-class capability	Human input is an integral part of the factory and impacts its stages. An input can be reviewing the agent's work, adding context mid-workflow, or deciding where the workflow should be routed next.	Approvals only let a person say yes or no, but many of the decisions that shape a run are about direction, not permission. If a reviewer cannot add context mid-workflow or route the run elsewhere, they either pass the output through or stop the run entirely. The judgment that would have made the next step right never reaches the agent, and the factory repeats the same mistake.
Insert approval steps anywhere	A run can pause more than once for human input, and each pause has its own people, its own form, and its own rules. Some pauses are approvals, which gate whether the run proceeds.	Approving a plan and approving a change are different decisions, and so is giving the agent context or choosing a route. With one input point for everything, you either review too late to matter or block work that never needed it.
Set who can provide human input	Policies decide who can answer each input request, from simple rules, like the work item owner reviews the agent plan, to advanced rules, like the team lead decides when a change affects a mission-critical service.	Input from the wrong person is worse than no input. If anyone can approve a change to a critical service, the review is a formality, and when the change fails, nobody can say who was entitled to decide.
Attach context to human input requests	The person providing input sees open incidents, recent deployments, the relevant metrics, and what the agent already tried.	With nothing to judge the change against, people click through, and you get a sign-off rather than a review.
Set when human input is required, in advance	Rules about tier, blast radius, timing, and confidence decide whether an action runs on its own, needs approval, or is blocked	Decided during an incident, the same action is allowed at 2am and blocked at noon depending on who is responding, and nobody can say afterwards which was correct.
Re-evaluate the human input policy at execution	The rules are checked again when the action runs, not only when it was proposed.	Conditions change between the plan and the action, and acting on an old approval is how a safe change becomes an outage.
Human input from chat (ChatOps)	Approvals reach people in chat or the tools they already use, and can be answered there.	If people have to open another tool, input requests wait, and so does all the work behind them.
Track pending human inputs in one place	All pending decisions appear in one place, showing which ones are stuck and who owns them.	Approvals spread across chat threads stall for days without anyone noticing, and the run behind each one waits that long.

3.1.3 Agents, skills, and MCP: how do you keep track of what agents use?

Capability	What it means	Why it matters
Execute third-party and in-house agents	Code-first agents (built with SDKs or cloud agents like GitHub Copilot, Cursor and Claude), cloud-managed agents (Bedrock, Azure AI Foundry, Cursor Cloud), and agents built natively in the platform all register and run under the same rules.	You will replace agents and models constantly. If the platform supports only one, each change becomes a migration and your engineers go back to working outside it.
Maintain an agent registry	One list of every agent you run, with its owner, model, status, and environment.	You cannot govern agents you cannot see and measure. Without one, you learn about an agent when it breaks something, and two teams build the same agent.
Publish skills and prompts to a shared registry	Skills and prompts are published with an owner, reviewed before release, and discoverable to the agents and IDEs each team uses, scoped by team.	Without a shared version, the same task comes out differently depending on who wrote the prompt. A prompt that lives on one laptop can't be reused, reviewed, or improved by anyone else
Govern MCP servers with a registry and gateway	Approved MCP servers are listed with a way to request new ones. The platform connects to each server once and acts as a gateway, so every team or agent sees only the tools it should, including a way to customize them and adjust their inputs.	Connecting to a server should not unlock every tool on it. Without per-tool control, one connection gives anyone who plugs in broad write access
Connect AI assets to the rest of your engineering context	Agents, their runs, skills, and MCP servers live in the same context layer as your services, repos, and work items, and are related to them. You can see which skills a service relies on, which agent runs touched a ticket or PR, and which team owns each one.	A standalone list tells you what exists, not where it's used or what it changed. When AI is connected to your services and work, you can trace an incident back to the agent run behind it, set permissions by ownership, and trigger workflows from AI activity like any other change.
Distribute AI coding rules to repositories	AI coding rules are maintained in one place and synced into the rules file each repository uses, such as AGENTS.md, where IDE agents read them. Each repository shows whether it's compliant and how to improve it.	Without it, each repository drifts, agents follow different rules depending on the repo, and reviewers catch the differences by hand.
Provision agents with identity and scope	A new agent is requested through a golden path and arrives with a non-human identity, a scope, and approved tools already set.	When agents share a service account or run on a developer's personal token, you can't tell which agent took an action or revoke one without breaking the others. An agent with its own identity and scope can be traced, limited, and shut off on its own.

3.1.3 Agents, skills, and MCP: how do you keep track of what agents use? (continued)

Capability	What it means	Why it matters
Certify, measure, and retire assets	Skills, prompts, agents, and MCP servers are approved and evaluated before use, measured while in use, then deprecated with their consumers notified.	A skill many teams use but nobody maintains spreads the same mistake across the organization for months before anyone looks at it.

3.2 Context: how do you give agents one reliable source of context?

3.2.1 Context ingestion: how to populate the context layer and keep it accurate

Capability	What it means	Why it matters
Model everything you operate in one place	One graph holds the objects you operate, in the shape you define: services, deployments, environments, incidents, work items, cloud resources, and teams.	If the model does not match how you work, agents fall back on data that is scattered and stale across many systems, and they decide from the wrong picture. That is how changes land on the wrong service and answers stop being trusted.
Ingest from the core toolchain and custom sources	Ready-made integrations cover the core toolchain: git, CI/CD, cloud, observability, on-call, incident management, and security scanning. Anything else can arrive through API, webhooks, a custom integration, or infrastructure as code.	Without the core tools, agents cannot answer what changed, what is running, who owns it, or what is at risk. A tool you cannot connect is invisible to every rule and every agent.
Sync on live events	Live events push changes as they happen and incremental syncs fill the gaps, and you can see when a sync fails or the data no longer matches reality.	Agents act on what the context layer says. When it lags they roll back the wrong deployment or page the wrong team, and people stop trusting the factory.
Auto-discover resources and relations	The platform suggests resources and relations it finds in your connected tools. Reconciliation makes sure there are no drifts or duplications.	Modeling everything by hand takes months, and for those months the factory covers a fraction of your estate.
Link unstructured knowledge to entities	Markdown context files, runbooks, ADRs, and postmortems are linked to the entities they describe, and agents read them under the same permissions.	Much of what explains a system is written down, not modeled. Without it, agents miss the runbook or the AGENTS.md that had the answer.

3.2.1 Context ingestion: how to populate the context layer and keep it accurate

Capability	What it means	Why it matters
Relate entities to actions, events, and decisions	Deployments, agent and workflow runs, incidents, approvals, and exceptions are linked to the entities they touched, with who acted and when.	Current state shows what is true now, not how and why it got there. These links answer what changed before something broke, and why.
Link factory identities to tool identities	Users, teams and agents are entities in the context layer, related to their identities in git, on-call, and chat.	A git handle, an on-call user, and a chat account are separate records. Until they are linked, the platform cannot say who owns a service, so work is routed and approved by the wrong people.
Relate entities to factory concepts	Users, teams, agents, workflows, skills, and MCP servers link to the services and work items they own or apply to.	Agents know which workflow, skill, and MCP tools apply to a service, and when. Without the links, they guess.

3.2.2 Context retrieval: how do agents get the context they need?

Capability	What it means	Why it matters
Search with precise filters	The API filters on any property or relation, combines conditions, sorts, aggregates, and returns only the fields asked for, whether the caller is a person, a workflow, or an agent.	Agents that cannot filter precisely pull whole lists and sort through them in the prompt. That costs tokens, and a long context is where the agent misses the record that mattered.
Expose the full API as MCP tools	Everything the API can do, reads and writes, is available to agents as MCP tools, under the same permissions and audit log as any other caller.	When agents get only a curated subset, teams fill the gaps with scripts and raw API tokens, and that work happens outside your permissions and your audit log.
Traverse the graph in one query	One query can follow relations across several hops, such as from an incident to the service, its dependencies, their owning teams, and who is on call, and return one answer.	Ownership, blast radius, and root cause are all questions that span relations. If each hop is a separate call, the agent stitches the answer together itself, and that is where it gets the join wrong.
Query unmodeled data over MCP	Agents reach data you have not modeled, such as a code diff or a chat thread, through MCP connectors to those tools at the moment they need it.	If agents can only reach what you modeled in advance, they reason without the diff, the thread, or the log line that explains the problem, and they reach confident conclusions that are wrong.
Track entity history	You can see how an entity and its properties changed over time, not only their current state.	Most investigations start with what changed before this broke. If the context layer holds only today's state, people and agents guess, and recovery takes longer than it should.

3.2.2 Context retrieval: how do agents get the context they need? (continued)

Capability	What it means	Why it matters
Centralize RBAC for context access	You grant access to the data once, in the context layer, instead of in every source tool, and every read is recorded.	Managing agent access tool by tool does not survive many agents. You end up with agents reading more than they should and no way to prove what any of them saw.
Use one query syntax everywhere	The same search and filter syntax is used by the entire platform.	The rule, the report, and the agent return the same list, so nobody spends the review meeting arguing about whose data is right.

3.2.3 Semantic layer and enrichment: how does the context fit your organization and improve over time?

Capability	What it means	Why it matters
Customize the semantic layer	You add and change types, properties, and relations from the UI, the API, or as code, without asking the vendor.	Teams define words like service and rollback differently. If the model cannot represent how a team delivers, that team has no entities in the context layer, so no standard, workflow, or agent can reach their work.
Add semantic descriptions	Types, properties, and relations carry a description of what they mean and where the value comes from.	Agents read these descriptions to decide what to do. Without them an agent guesses what a field means, and a wrong guess about tier or environment becomes a wrong action in production.
Build organizational memory	Knowledge enrichment flows learn continuously from incidents, fixes, rejected changes, and postmortems, and write the patterns back onto the related entities.	Without memory, every run starts from zero. Agents repeat mistakes the team already learned from, and that knowledge leaves with the people who hold it.
Define relations between entities	An entity can have many relations, and can show values from related entities, such as a service showing its team's on-call.	Ownership, dependencies, and blast radius are all relation questions. A flat model cannot answer them, so agents cannot judge how risky a change is before making it.
Use calculated and aggregated properties	The platform can calculate values for you, such as open incidents per service or how many dependencies are out of date.	Everyone works from the same number instead of each team and each agent computing it their own way, so you are not reconciling three versions of it in a meeting.
Extend the platform's own concepts	Teams, users, scorecards, agents, factory lines, workflows and more are themselves entities you can add properties and relations to, such as an owner, a category, or an SLA.	If the parts the vendor built cannot carry your fields, you end up tracking owners, due dates, and SLAs in a spreadsheet beside the platform, and the two drift apart.
Manage the model as code	The model is managed as code through GitOps or Terraform and promoted between environments.	Once the model drives what agents do, an untracked change to it changes behavior everywhere. Review and rollback are how you avoid finding that out in production.

3.3 Control

3.3.1 Standards: how do you keep your software factory up to your standards?

Capability	What it means	Why it matters
Turn failed standards into fixed work	When a service falls below a standard, the platform opens a work item that enters a factory line, where an agent fixes it and the standard is checked again to confirm it passed. Where a fix is not safe to automate, it notifies the owner or opens a ticket for a person.	Otherwise the gap sits on a dashboard until it causes the incident the standard was written to prevent. Closing the loop makes a failing standard work the factory does, not a report people have to act on.
Set standards on the factory itself	Standards also apply to how the factory runs, with goals on measures such as agent PR acceptance rate, lead time, and time to recovery. When a goal is missed, an agent can find the reason and open work items to improve the line.	A factory that is only measured gets reported on, and one with goals gets better. Without standards for its own performance, a line that slowly gets worse, such as rising rework or slower recovery, goes unnoticed until someone reviews a dashboard, and the fix waits for a person with time to act.
Block actions that fail a standard	A standard can stop an action from happening, not only record that it failed.	Reporting only works if someone reads it in time. With agents producing changes all day nobody does, and the first sign of a problem is an incident.
Give agents the standards before they act	Agents read the standards that apply to a service, such as its tier and the rules it must meet, as context before they plan or change anything. They can check their own work against them before opening a pull request.	Blocking after the fact catches the mistake but wastes the run. An agent that knows the bar up front gets it right the first time, so fewer runs fail the gate and fewer changes need rework.
Grade maturity in levels	Services are graded across levels, such as bronze, silver, and gold, instead of compliant or not.	Teams can see which rule to fix next and which level it moves them to, instead of a single red mark that does not say where to start.
Apply the same standard to agent changes	A change made by an agent has to meet the same standard as a change made by a person.	Agents will make most of the changes. If standards only cover people, your quality bar stops applying to most of your software.
Certify agents, skills, and MCP servers	Agents, skills, prompts, and MCP servers are checked against a standard, the same way a service is.	These reach production systems. An agent nobody checked, with wide access, can turn a helpful automation into an outage.
Query scorecards, rules, and results	Scorecards, rules, and results are entities you can search, chart, and build workflows on.	A failing rule becomes a list of work with owners, so the gaps get assigned and closed instead of accumulating in a report.
Track standards over time	Every rule result is an entity with history, so you can chart progress per service, per team, and per initiative.	Without history you only see today's compliance, so you cannot tell leadership whether the program is working or show an auditor a trend.

3.3.1 Standards: how do you keep your software factory up to your standards? (continued)

Capability	What it means	Why it matters
Show each owner their failing rules	A view of unmet rules filtered to the entities that person or team owns.	People act on a list of services they own. An organization-wide compliance number gets read and ignored, and nothing gets fixed.
Scope rules per team and grant exceptions	Different teams can have different definitions of done, and an exception, or waiver, carries an owner and an expiry date.	A rule that does not fit a team's stack fails for reasons they cannot act on, so they stop looking at the scorecard. An exception with no end date is never revisited, so it becomes the standard by default.

3.3.2 Permissions: how do you control what each agent can see and do?

Capability	What it means	Why it matters
Give agents their own service accounts	Each agent gets its own non-human identity, a service account and token limited by environment and by action, usable from CI and from an IDE.	If an agent acts as a person, the audit trail is wrong and its permissions are far wider than the task, so a single prompt can reach everything that person can.
Cap agent permissions at the user's own	An agent working on someone's behalf inherits least privilege: it can never do more than that person is allowed to do.	Without this, anyone can use an agent to do what they are not allowed to do, and the permission model you already built stops meaning anything.
Use attribute-based access control	Rules are checked against current context layer data, such as only the owning team may deploy a tier-1 service during a change window.	Ownership changes often. Fixed name lists are out of date within weeks, and then either the wrong people have access or the right ones are blocked from doing their job.
Restrict tool access per agent	You choose which tools each agent can call, and tools that change things require approval by default.	A few tools cause most of the damage. Without per-tool control, any agent that connects can call those tools too.
Scope agent read access	What an agent can see is set separately from what it can do.	An agent that can read everything is a data exposure even if it never changes anything, and that is the finding that stops the rollout.
Simulate permissions before granting them	You can dry-run a permission check for a person or an agent and see both the answer and the reason.	Testing in production is slow and risky, and every unexplained denial becomes a support ticket for your team.
Revoke a single agent	You can disable one agent immediately without affecting the others.	During an incident you need to stop one agent without stopping the factory. With all or nothing you either leave it running or halt everyone's work.

3.3.2 Permissions: how do you control what each agent can see and do? (continued)

Capability	What it means	Why it matters
Let teams build their own lines, with central oversight	A central platform team oversees all factory lines and owns the shared standards, policies, and registries. Permissions let other teams create, edit, publish, and run their own workflows, agents, and skills, scoped by team and environment, and keep changes to shared standards and policies with the platform team.	An AI-native organization cannot wait on one team to build every line. If only the platform team can build, other teams work around it, and their automations sit outside your standards and audit log. If everyone can change everything, one edit to a shared standard changes every team's work at once. Scoped permissions let the teams closest to the work build while the platform team keeps oversight of all of it.

3.3.3 Audit: how do you know what people and agents did, and why?

Capability	What it means	Why it matters
Keep one audit log for people and agents	Actions by people and by agents are in the same log, each linked to the identity behind it and the system it came from.	After an incident you need to know who changed what and who approved it. Two logs cannot answer that, and neither can an auditor.
Log every agent invocation	An invocation log records, for each agent run, the data it read, the tools it called, and what it decided.	When output is wrong the cause is usually the input. Without both you cannot tell a weak model from weak context, so you spend weeks fixing the wrong thing.
Capture feedback on agent decisions	You can record whether a decision was right, and compare agents and skills over time.	Otherwise quality stays an opinion, and you cannot tell whether changing a skill or a model made things better or worse.
Trace failures back to missing context	When an agent gets something wrong, you can find the missing context, add it to the context layer, and run again.	This is how the factory improves. Without it the same mistake repeats, and the only fix anyone tries is rewording the prompt.
Produce audit evidence per release	The full history of one release, including approvals, comes from a single view.	Audits and regulators ask for this, and rebuilding it by hand costs days of engineering time for every request.
Retain and export logs	Logs are kept for as long as your policy requires and can be forwarded to your own SIEM.	Security teams need these logs in their SIEM beside everything else they monitor, and a trail they cannot retain is one they will not accept.

3.3.4 Measurement: how do you know the factory is working?

Capability	What it means	Why it matters
Velocity flow Are we delivering faster?	Time from request to PR and production, deployment frequency, PR throughput, completed work, flow metrics and waits across reviews, approvals, and handoffs.	Agents can write code faster while delivery stays slow. Without measuring the whole flow, you cannot tell whether the software factory actually improved.
Quality and reliability Are we maintaining and improving quality and reliability?	Production defects, change failures, security findings, rework, and recovery time.	Volume on its own hides rework. The factory can look productive while people fix what it ships and customers experience more failures.
Autonomy Which tasks are human-driven and which are autonomous?	Tasks led by people, with or without AI assistance, versus tasks completed autonomously by agents across the whole workflow. Track where people approve, correct, or take over agent work.	More agent activity does not mean more autonomy. You need to understand who drives the work and where human oversight is still needed before expanding agent autonomy.
Cost efficiency Are we reducing the cost of delivery?	AI spend and human effort per completed work item, including retries, reviews, and corrections.	Agent spend grows with use. Without counting the full cost, you cannot tell which workflows justify their spend.
Agent readiness **Do agents have what they need, and are their decisions improving?**	Context availability and accuracy, gaps in tests and tooling, and whether agent decisions lead to fewer corrections and better outcomes over time.	Agents repeat mistakes when context is missing or findings are lost. Measuring both helps you check whether the factory learns from previous work.
Velocity flow Are we delivering faster?	Time from request to PR and production, deployment frequency, PR throughput, completed work, flow metrics and waits across reviews, approvals, and handoffs.	Agents can write code faster while delivery stays slow. Without measuring the whole flow, you cannot tell whether the software factory actually improved.
Quality and reliability Are we maintaining and improving quality and reliability?	Production defects, change failures, security findings, rework, and recovery time.	Volume on its own hides rework. The factory can look productive while people fix what it ships and customers experience more failures.

3.4 General platform requirements

3.4.1 Interfaces: how do you get every team working in the same place?

Capability	What it means	Why it matters
Build role-based views	Developers, security, SRE, managers, and people who do not write code each get pages limited to what they may see and do.	Several groups operate the factory. Anyone it does not serve keeps working in their own tools, and what they do there is never gated or recorded by the platform.
Embed actions in the view	The workflows people need are buttons on the page they are already looking at, running the same governed logic.	If people have to switch tools to act, they find a faster route that skips your controls.
Build initiative dashboards	The person running an initiative can see progress by team, what is blocked, and why.	Without it every status update is assembled by hand, and the reporting cost grows with each initiative you run.
Build custom UI components	You can build custom views that run inside the platform, using its data, permissions, and login.	Any view you cannot build yourself waits for a vendor release, and the team that needed it builds something outside the platform in the meantime.
Embed a scoped AI assistant	An assistant sits on the page, answers from the context layer, and can only run what that user is allowed to run.	Not everyone will learn a query language, and they should not have to. Without this they ask a person, and your team becomes the interface.
Follow and steer a running agent	A person can watch a run as it happens, see what the agent has read and done so far, and pause it, add context, redirect it, or stop it.	Reviewing the result after the run is too late to change it. If people cannot step in mid-run, they let a wrong run finish or cancel work that was mostly right.
Operate from the IDE	Developers can start workflows, see their work items, and answer human input requests from the editor they already use.	Developers will not switch tools for every request. If they have to, the request waits or gets answered somewhere the platform does not see.
Operate from chat	Questions, notifications, and human input requests work from your chat tool, such as Slack or Teams, routed to the right channel for each event.	That is where engineering already works. Approvals and questions that live somewhere else either wait or get answered off the record.

3.4.2 Administration: how do you run the platform in your environment?

Capability	What it means	Why it matters
Connect SSO and SCIM	Single sign-on over SAML or OIDC, automatic user creation and removal through SCIM, group filters, and roles taken from your directory.	Managing users by hand does not scale, and accounts that are never removed are the access risk auditors find first.
Separate development and production environments	A development environment where you change standards and workflows, and a production one they are promoted into.	You need to test a standard or a workflow before it applies to production, or a single edit blocks every team at once.
Manage the whole factory as code	Everything that defines the factory, not only the data model, can be managed as code: standards, workflows, agents, skills, permissions, and views. It works through GitOps and CI/CD pipelines or infrastructure as code tools such as Terraform. Changes go through pull request review and are promoted between environments by pipeline, and drift between the code and what is running is detected.	The factory changes every week. A change made in the UI leaves no review, no rollback, and no record of why, and once these definitions drive what agents do, an untracked edit changes behavior everywhere. As code, they get the same review and rollback as the software they govern, and teams that work in pipelines are not forced into a UI.
Choose region and deployment model	A clear answer on data residency: where data is stored, which regions are available, and what can run inside your own network.	This is the question that stops rollouts at the last stage, and answering it after the POC means running the evaluation again.
Manage secrets outside agent context	Secrets are encrypted, cannot be read back after they are saved, and are referenced by name in a workflow rather than pasted into agent context.	Anything that reaches a prompt should be treated as stored somewhere you do not control, and one leaked credential is the incident that ends the program.
Use your own model provider	The platform can use your model provider, your cloud's model service, or your own gateway, with a different model per use case.	Models differ in cost and quality by task, and some work cannot leave your infrastructure. Without the choice you either overpay or cannot run the workload at all.
Enforce rate limits and quotas	Published limits on concurrent runs and API calls, and protection for your source systems when agents run hot.	Agent volume grows quickly. Without limits the first symptom is your own production systems slowing down under your automation.
Export everything, with no vendor lock-in	Your model, standards, workflows, and assets can be exported with their relations intact.	What you build here is the valuable part. If it cannot leave, every renewal is a negotiation you cannot walk away from.

3.4.3 Agnostic: how do you avoid a rebuild when your stack changes?

One checklist for the agnostic gate. For each row, ask what in the factory would have to be rebuilt if you replaced that tool next quarter.

Capability	What it means	Why it matters
Work with any git provider	GitHub, GitLab, Bitbucket, Azure DevOps, and self-hosted installations are all first-class, including more than one at the same time.	Large organizations usually run two or more after acquisitions. A platform tied to one leaves those teams with no context layer, no standards, and no workflows.
Work with any CI/ CD system	Workflows can be triggered by your pipelines and can trigger them back, whichever system a team uses.	Your pipelines are not moving because of a platform purchase. If the factory only drives one of them, every other team keeps its own scripts outside the governed path.
Work with any cloud and runtime	The cloud accounts, Kubernetes clusters, and on-premises runtimes you have today all appear in the context layer and can be acted on.	The platform cannot gate a deployment into a runtime it cannot see, so agents working there run with no checks at all.
Work with any observability and on-call stack	Metrics, logs, alerts, and on-call schedules come from the tools you already run.	These tell an agent whether its change worked and who to wake up. Without them it is guessing at both.
Work with any security scanners	Findings from your code, dependency, container, and cloud scanners feed the same context layer and the same standards.	If security data sits outside, you cannot make it a condition of a deployment, and the security team stays outside the factory too.
Work with any issue tracker	Work items from your tracker are entities in the context layer, linked to services and owners, whichever tracker each team uses.	Intake is where the factory starts. If it only reads one tracker, the teams on the others cannot use the line at all.
Run any coding agent	Third-party assistants, cloud-hosted agents, and agents your own team built all register, run, and are governed the same way.	Agent choice changes every few quarters and differs by team. If the platform supports one, either your engineers work outside it or you re-platform when you switch.
Use any LLM provider and model	Your own provider account, your cloud's model service, or a gateway you already run, with the model chosen per use case.	Models differ in cost and quality by task, and some work cannot leave your infrastructure. Without the choice you either overpay or cannot run that workload at all.
Keep skills and prompts portable	Skills and prompts are stored in an open format, in your repositories, and read by agents that are not the vendor's.	These encode how your organization works. If they only exist inside one product, the work you put into them leaves with the contract.

3.4.3 Agnostic: how do you avoid a rebuild when your stack changes? (continued)

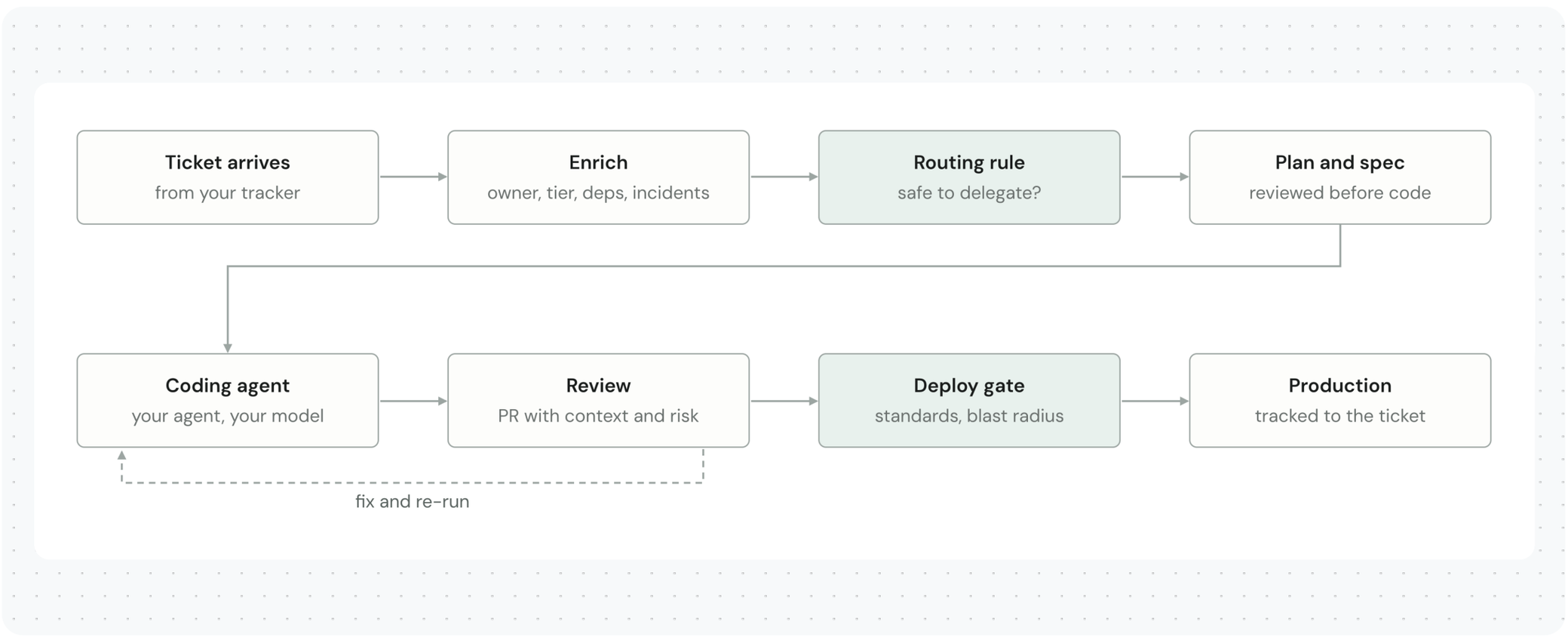
Capability	What it means	Why it matters
Connect any MCP server	External MCP servers can be registered and routed, and you can expose your own internal tools the same way.	It is how the factory reaches tools nobody wrote a connector for. Without it, coverage stops at the vendor's integration list.
Use your identity provider and chat tool	Users, teams, and roles come from your directory, and notifications and approvals run through your existing chat tool.	These decide who can do what and where approvals happen. A second directory or a second inbox means both go stale.
Run agents where you choose	Agents can run in your CI, in your cloud, or hosted by the vendor, without changing how they are governed.	Some workloads cannot run outside your network. If the controls only apply to the vendor-hosted option, those are the workloads running without them.
Avoid vendor lock-in	The model, standards, workflows, skills, and prompts export in a usable format, with their relations intact.	What you build here is the valuable part. If it cannot leave, every renewal is a negotiation you cannot walk away from.

4. What should you build in the POC?

Build these three lines in your own environment, on your own repositories, tickets, and alerts. The steps follow the diagram above each table, in the order you would build them. When a step fails it is usually a pillar giving way rather than one missing feature, and that is what the red flag describes.

Line 1. Idea to production

A real ticket, from intake to a deployed change, where a rule decides how it is handled instead of whoever picked it up.

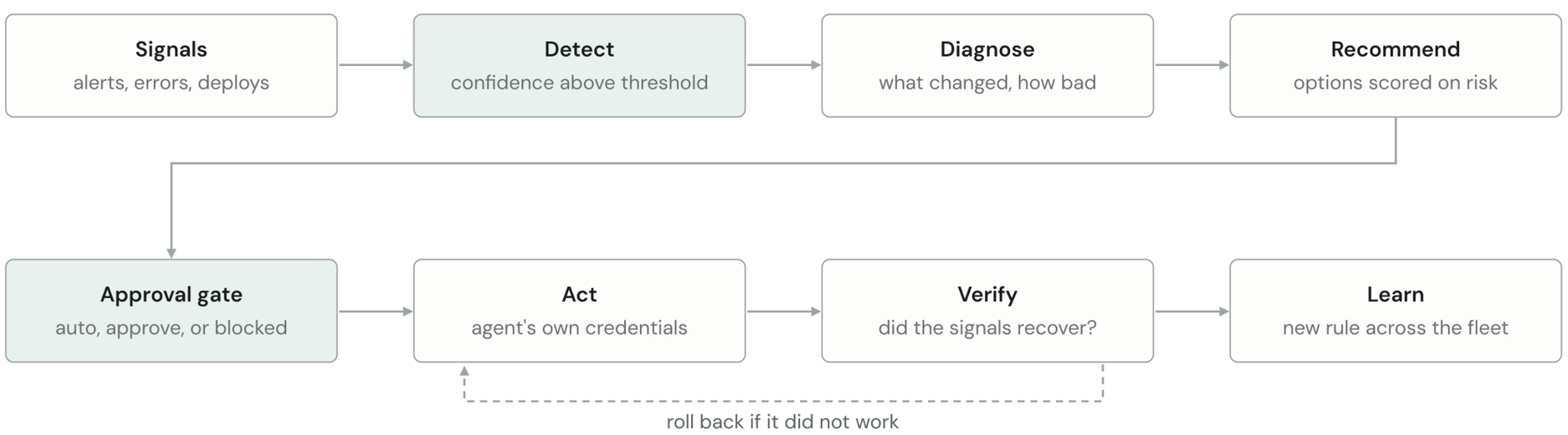


Line 1. Idea to production (continued)

Step	What you build	Red flag
Ticket arrives	Connect your tracker so a new ticket lands in the context layer as a work item linked to a service	The factory starts at the pull request, so intake and planning stay outside it and nothing before the first commit is tracked or governed.
Enrich	Add a step that pulls owner, tier, dependencies, and open incidents onto the work item	The fields the work item needs cannot be added, so every agent starts from a title and guesses the rest.
Routing rule	Write a standard that decides whether the work is safe to delegate, using tier, blast radius, open incidents, and how well it is described	The rule only advises, so the decision is made by whoever is on rotation, differently every time.
Plan and spec	Generate the requirements and the implementation plan from your shared skills, and have a person review them	Nothing is written down before the agent starts, so afterwards you cannot tell whether the agent was wrong or the instruction was.
Coding agent	Hand the work to the coding agent your engineers already use, passing in the context and the constraints	Only the vendor's own agent can be called, so your engineers keep working outside the factory.
Review	Open the pull request with ownership, risk, and the standards that apply, routed to the right reviewer	The owning team and security cannot review here, so review moves back into chat and the record ends.
Deploy gate	Gate promotion on the readiness standard and the blast radius, then let it proceed without a second check	The gate reports instead of blocking, which during a release is a report nobody reads.
Measure the line	Compare against the baseline: time from intake to deployed change, how much ran without a person, rework on agent changes, and cost per item	The only number at the end is tickets handled, which tells you the line ran and nothing about whether it worked.
Trace the line	Follow one requirement from intake to deployed change in a single view, with every approval and agent action attributed	The trail breaks at a tool boundary, so nobody can say who approved the change or on what basis.

Line 2. Self-healing incidents

From alert to remediation to a standard that stops it recurring. Run it on a real service, with a real page.



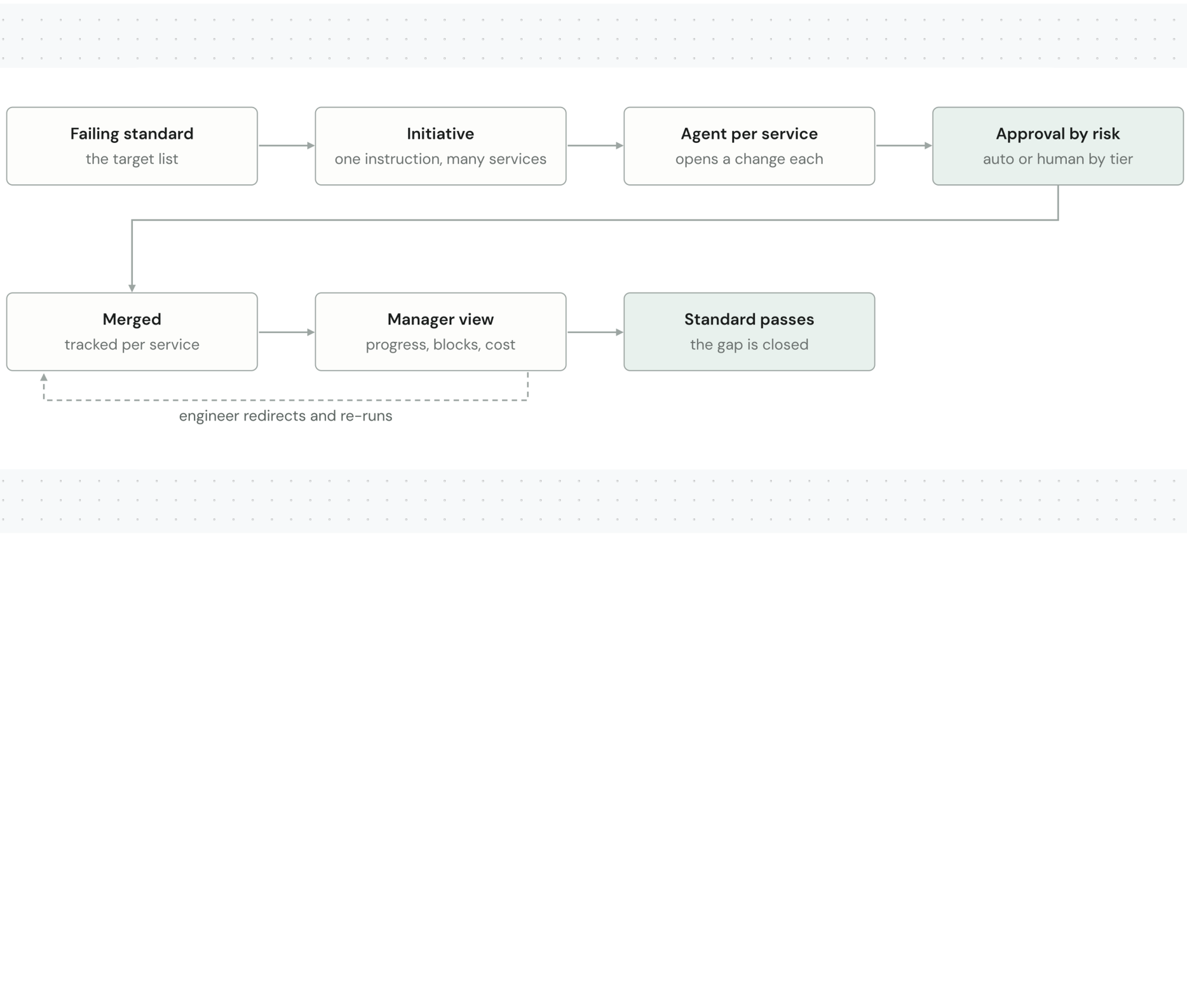
Step	What you build	Red flag
Detect	Set the confidence threshold and let signals open an incident with an owner and a start time	Declaring an incident is manual, so the response starts whenever someone happens to notice.
Diagnose	Give the agent the service's recent deployments, config changes, dependencies, and impact without being asked	The context layer has no record of recent deployments or config changes, so the agent asks the responder to paste them in during the incident.
Recommend	Have the agent score more than one option on effectiveness, risk, reversibility, and time to verify	Only one option is ever put forward, so there is nothing to compare and no way to say the choice was a good one.
Approval gate	Write the policy that returns autonomous, needs approval, or blocked, and attach the context the approver needs	Whether the action may run gets decided during the incident, so it is decided differently every time.
Act	Run the chosen action with the agent's own credentials and its own restricted set of tools	The agent runs on the responder's credentials, so it can reach everything that person can, in the worst possible hour.
Verify and roll back	Check the signals the action was meant to move, and roll back or escalate when they do not recover	Nothing checks the result, so a fix that did nothing looks exactly like one that worked.

Line 2. Self-healing incidents (continued)

Step	What you build	Red flag
Learn	Turn the lesson into a standard applied across the fleet, and add the context the agent was missing to the context layer	The lesson cannot become a rule that applies everywhere, so the same incident comes back next quarter.
Measure the line	Compare against the baseline: time to diagnosis and to recovery, share closed without a person acting, how often the diagnosis matched the real cause, and cost per incident	Only speed is reported, so nobody can tell whether the automation was right or just fast.
Trace the line	Show what the agent read, what it concluded, who approved, whether it was right, and whether it earned more autonomy next time	Only the action is logged, so the judgment behind it cannot be reviewed.

Line 3. Run an initiative

One change that many services need. Agents do the work, engineers run them, and the manager can see where it stands.



Line 3. Run an initiative (continued)

Step	What you build	Red flag
Target list	Use a failing standard to produce the affected services, their owners, and their environments	The standard only reports, so the target list goes back to being a spreadsheet nobody trusts.
Initiative	Model the initiative once and each affected service as its own tracked target underneath it	The initiative and its targets cannot be modeled, so progress lives outside the platform from day one.
Agent per service	Have agents open a change per service, each scoped to that team's repositories	One agent holds write access across the organization, which makes the whole program a single blast radius.
Approval by risk	Set one policy where low-risk services proceed on their own and tier-1 services wait for a person	Every change takes the same path, so either tier-1 changes are unsafe or everything is slow.
Engineers operate	Let an engineer review, redirect, and re-run an agent from the surface they already work in	Corrections go through the platform team, so the program runs at the speed of that queue.
Manager view	Build the view that shows progress by team, what is blocked and why, and the exposure still open	The manager cannot see the program, so people have to stop working to produce a status.
Measure the line	Track targets completed and merged, rework and failed runs, where work queued and for how long, and spend against the engineering time it replaced	A count of pull requests is the only measure, which says the agents ran and not that the gap is closing.
Set a goal on the line	Write a standard on the line's own results, such as agent PR acceptance rate or time from intake to deployed change, and let a missed goal open a work item for an agent to find the cause and propose a fix	Goals can only be viewed on a dashboard, so a line that is getting worse is noticed only when someone looks.
Trace the line	Attribute every change to an agent with a named owner, and close the initiative with the same standard that opened it	Changes arrive from an anonymous account, so nobody is accountable when one of them goes wrong.

A service created after the initiative should fail the same standard without anyone adding it to a list. If it does not, you automated a one-time cleanup rather than closing the gap.

5. What are the seven must-haves, and what happens without them?

Seven things are worth insisting on, whichever vendor you pick.

- **Governed.** Standards block an action rather than report it after the fact, they cover agents as well as people, and both show up in one audit log.
- **Flexible.** Your engineers extend the data model, add sources, and build workflows without waiting on a release.
- **Covers all AI-SDLC stages.** Intake through security, on the same triggers and approvals.
- **Measurable.** You can see what an agent read and decided, and mark the times it was wrong.
- **Agnostic.** Works with your tools, your models, and other vendors' agents.
- **Contributable.** Teams outside the platform team can add workflows, skills, and agents inside your standards.
- **Collaborative.** Agents and humans collaborate on shared tools and context.

None of it matters much in a pilot, which is why it is easy to skip. At ten agents you still review the changes yourself. At a hundred, a scorecard that only reports never gets opened, and you find out during an incident. The first workflow fits whatever shipped in the box, and the tenth needs an entity type or a source that is not there. One uncovered stage is fine in one team, but across dozens each team fills the gap its own way and you are governing several factories. A few runs a week and you notice a bad change yourself, a few thousand and quality slips for months. The rest holds until the day you swap a model or an agent, or until a team that could not add a workflow to your factory quietly builds their own, and their work stops being something you can see.

Template for internal use. Adapt the capabilities and the lines to your program before issuing to vendors.