How to Build an Agentic Software Factory

A software factory should turn an approved request into a reviewer-ready PR with evidence.

Start with one type of change and define the verified PR you expect back. Agents need the right context, a way to coordinate, a sandbox to test the complete change, and clear limits on access and decisions. Add automation and reviewed learning once that path works.

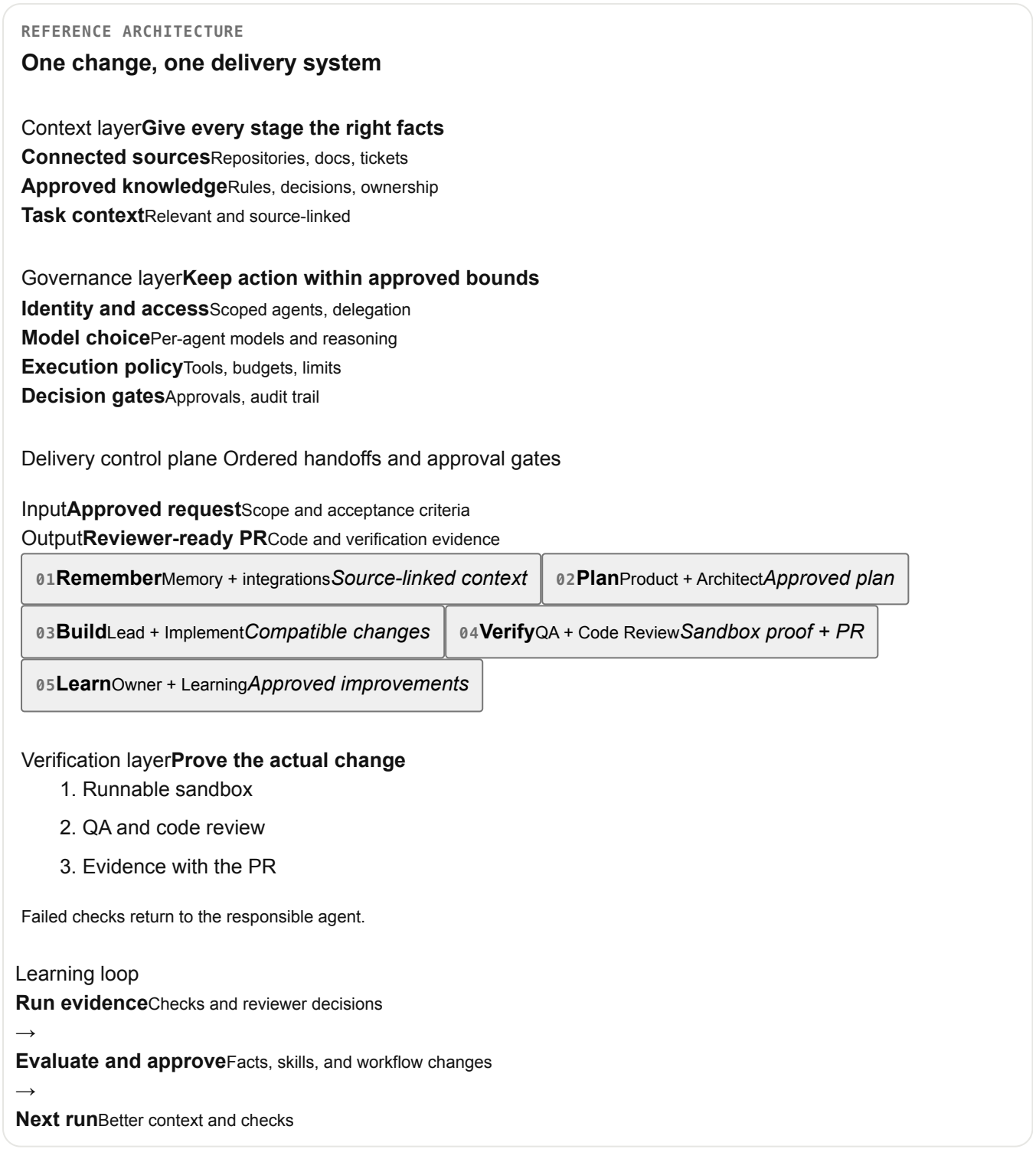
The setup depends on the work, not just company size. A small fix may need only a short review path, while a change across teams or repositories needs explicit owners and handoffs. A consequential change needs access controls and approval even in a small team.

I think about the setup as five layers. They are responsibilities to cover, not five services you must build separately.

- **Delivery control plane** keeps agent definitions, task state, handoffs, artifacts, and human decisions together.
- **Verification layer** gives QA a runnable sandbox and records checks against the code revision being reviewed.
- **Context layer** supplies business rules, architecture, contracts, ownership, and prior decisions with their sources.
- **Governance layer** enforces permissions, model and tool policy, budgets, approvals, and an audit trail.
- **Learning layer** reviews run evidence and decisions, then promotes approved knowledge, checks, and workflow improvements for the next run.

An approved request enters the control plane, agents use context while working, and verification checks the resulting revision. Governance applies throughout. The sections start with coordination and verification because you can test them with one change. They are not the order in which agents encounter the layers.

The reference architecture shows how those capabilities support one change as it moves from request to a verified PR. Its five delivery stages describe the workflow, while the layers describe the responsibilities that support it.



1. Delivery control plane layer - Set up the team and move work forward

The control plane keeps the approved agent setup, run state, handoffs, and human decisions in one place. It starts the next task only when its dependencies and required approvals are satisfied.

Factory setup - Configure the agent team and shared workspace

Start with versioned files and a simple interface if that meets your needs. The team should be able to inspect the setup each run actually uses.

Component	What to build
Agent catalog	Create and version agent definitions with a responsibility, inputs, outputs, model configuration, allowed tools, and assigned skills.
Skill library	Create, edit, review, and version reusable procedures. Let several agents use the same skill without maintaining separate copies.
Session and run workspace	Show the conversation, current work, owners, artifacts, approvals, and evidence. Let authorized teammates join, steer, or continue the work.
Harness configuration	Bring agent definitions, skill versions, workflow rules, context sources, and tool policies into an approved configuration that new runs can load.

An authorized teammate should be able to join a session, review an artifact, or steer the work without rebuilding its history. Record the agent and skill versions used by each run, and do not let a later edit silently change an active run. Sharing a session must not grant everyone the original operator's permissions.

Orchestration - Coordinate work and recover interrupted runs

The system needs to know which agent owns the next step, what it must receive, and where a failure or review finding goes back. Agents can suggest the next task, but orchestration checks dependencies, permissions, and approvals before starting it.

1. **Record who owns the work and what it needs.** Store the run, task, accountable agent, input artifact revisions, expected outputs, allowed tools, and attempt identity. Give completed, blocked, waiting for approval, and failed distinct outcomes.
2. **Make the handoff specific.** A review finding should reference the requirement, code revision, evidence, and requested correction. Return it to the responsible implementation task. Preserve the exchange so later reviewers can understand why the code changed.
3. **Pause at human decision gates.** For a material change, send the requirements or design revision, the open question, and the agent's evidence to a named owner before implementation moves on. The owner can approve it, request a revision, redirect the work, or stop it. Record the decision against that artifact revision and return feedback to the responsible agent. If the artifact changes materially, reopen the approval. A bounded fix can pass without an intermediate human gate when the team has configured that path, but it still needs the required review and verification.
4. **Handle retries and interrupted work.** Persist state outside chat history. Give dispatch commands idempotency keys so duplicate events do not create duplicate work. Reject stale completions from earlier attempts and expose an explicit resume, retry, repair, or stop action.

In a full workflow, Product records the request and acceptance criteria. Architect identifies the affected systems and contracts. Lead turns the approved design into implementation tasks. Implement returns code revisions, Code Review sends findings back to the responsible task, and Verify tests the corrected revision. Each handoff carries the artifact, decision, owner, and revision. An agent should not have to reconstruct that exchange from separate chats.

Across repositories, keep compatible code revisions and combined verification evidence visible in the same workspace.

The Coordination Backbone explains work contracts, typed outcomes, an agent registry, and transition policies. Use it when connecting shared agent definitions to durable, authorized work.

Ready when a teammate can inspect a run, a material decision pauses for its owner, and feedback reaches the right agent before work continues. Interrupt and resume a test run. Duplicate events must not create duplicate tasks, and stale results must not replace the current attempt.

2. Verification layer - Build a reproducible sandbox

For me, verification is the most important part. A QA agent needs a runnable environment, not just a detailed prompt. Separate repository tests can pass while the complete change fails at an API, event, or worker boundary. Run the affected components together in a sandbox.

1. **Make the environment repeatable.** Record the repository revisions, dependencies, configuration, startup commands, and health checks. Put setup commands and test fixtures in the repository, with isolated test data and credentials. An agent should be able to start the app without asking you how.
2. **Give QA the data and tools to test.** Seed the states required by the acceptance criteria. Give Verify the browser, API access, logs, and safe test integrations it needs to inspect the result. Record which dependencies are simulated and which boundaries remain untested.
3. **Close the feedback loop.** Have Verify return the failing scenario, tested revision, expected behavior, observed result, and evidence to Implement. After a fix, rerun the affected checks against the new revision. Let that loop run more than once if needed. Set a limit and make it clear when the agent needs to ask for help.

Choose checks that match the change. Focused tests catch a rule failure quickly; API, browser, and integration checks cover the boundaries the change touches.

Measure both time to detect a failure and time to locate and fix it. A long test ending with “flow failed” leaves the agent a large search space. Record the input, expected behavior, observed result, tested revision, and relevant logs so it can identify which boundary broke. My testing feedback-loop post explains how to use those two measures when choosing between focused tests and broader integration checks.

The PR should show the requirement, tested revision, environment, checks, and remaining gaps. If the sandbox cannot start, mark verification blocked rather than passing. Fix missing setup or test data in the repository and rerun. A pass from an earlier revision cannot validate code changed afterward.

My April software-factory essay goes deeper on why a runnable integration environment matters. A passing check in one repository cannot prove that another service will consume its output correctly.

Ready when an agent can start a fresh sandbox, reproduce an intentionally introduced failure, pass an actionable finding to Implement, and verify the corrected revision with inspectable evidence. Run this exercise in a disposable test branch. Keep production rollout checks as a separate responsibility.

3. Context layer - Create a shared baseline for the team

You want agents and humans to work from the same baseline. Otherwise, each session starts with a different understanding of the product and someone has to keep correcting it.

1. **Write down the context agents keep needing.** Keep domain rules, architecture, service contracts, ownership, and accepted decisions with their sources. Agents should find the current rule and the reason behind it before changing code.
2. **Keep the source and owner with the context.** Keep its source, owner, applicable system, and freshness visible. Define which source is authoritative when records conflict. Retrieve relevant context within the requester’s permissions instead of loading every conversation into every prompt.

3. **Attach context to the work.** Link the sources behind a requirement or design decision to the change record. The control plane manages the workflow; this layer supplies the facts agents use.

Start with versioned documents and a decision log if they answer your retrieval needs. As questions span more systems, structured facts and relationships can help connect rules, owners, services, and historical decisions.

Keep business context separate from instructions. Product rules belong in domain knowledge, executable checks in repository tests, and reusable methods in skills. Do not copy the same rule into every prompt; keep one authoritative source and link to it.

My Prinevo Memory architecture article goes deeper on connected facts and changing decisions. The context-tiering article helps separate orientation, skills, and reference material.

Ready when a fresh agent session can retrieve the applicable domain rule and current contract with their sources. Give it an outdated decision as well and check that the conflict is surfaced or resolved against the authoritative record. The resulting plan should show which sources it used.

4. Governance layer - Scope models, tools, and environment access

The harness is the setup around a model, including its instructions, skills, tools, context access, and execution rules. Make that setup explicit and versioned.

Treat agents as first-class actors. The system should know which agent did the work, who asked it to act, and what authority it used. Do not hide every agent action behind one person's account or a shared token.

1. **Give each agent an identity and role.** List the repositories, tools, environments, and operations it needs. QA may need sandbox access and logs without permission to change production. Implement may need code-write access without the ability to approve its own work. Keep those roles distinct unless a separately authorized task requires more.
2. **Support delegated and direct access.** When an agent acts for a person, use the integration's supported delegation flow and limit the grant to that person's allowed actions, the task, and its lifetime. If Implement delegates a verification task to QA, give QA a new grant for that task rather than passing along Implement's code-write access. When an agent performs independently authorized work, use its own scoped identity and permissions. Record the agent, requesting person or team, authority used, and approval in each case. If an integration cannot enforce delegation safely, do not substitute a broad personal token.
3. **Enforce the boundary.** Use scoped credentials and permissions in the underlying services. As integrations grow, a tool registry or MCP gateway can centralize allowed tools, restrictions, and audit records. Ensure shell access or another integration cannot bypass the restriction. The MCP authorization specification provides a reference for scoped authorization.
4. **Configure model choice and operating limits.** Select models by task, record model and harness versions, and set time, retry, and cost limits. Route exceptions to an owner. Keep a way to stop a run and revoke its access.

Models will keep changing. Keep context and handoff rules independent of model choice so you can compare quality, reviewer rework, and total cost before switching.

Keep secrets out of shared memory. Treat retrieved documents and tool responses as data that cannot grant themselves additional permissions.

Ready when a delegated agent can perform only the actions its requester could authorize, an independently authorized agent can use only its own scoped access, and both paths identify the agent and authority in the audit record. An unauthorized operation must be denied, and revoked access must stop working. Test denied operations against a controlled fixture rather than attempting a destructive production action.

5. Learning layer - Improve the setup with evaluation and rollback

Learning has two parts. Keep approved facts and decisions available to the next run, and improve the way agents work when evidence shows a recurring problem.

1. **Save the facts and decisions you want to reuse.** Capture a learned domain rule or decision with its source, scope, owner, and evidence. Review it before promoting it to shared context. When a rule changes, supersede the old record while preserving the history.
2. **Propose changes to the harness separately.** If agents repeatedly miss background consumers, propose an update to the impact-analysis skill, a regression check, or a handoff rule. Version that proposal and explain which observed failure it addresses.
3. **Evaluate before making the change shared.** Compare the proposed setup with the previous version on representative tasks. Check whether it fixes the original failure without hurting other workflows. Review the results, promote accepted changes, and retain a rollback path.

Review the run trace, not only the final PR. Look for lost findings, repeated failures, and implementation that started before approval. Decide whether the cause was missing knowledge, a setup defect, a skill gap, or orchestration before changing the system. Keep rejected proposals visible so the team knows what it already tried.

For a pilot, agree on what a completed run means and record a baseline. Count reviewer-ready PRs with their checks and evidence, but also track human intervention, rework, blocked runs, time, and compute cost. That tells you whether the next setup actually improves delivery, not just whether agents produced more code.

The compounding-systems essay explains this loop in more depth. Observe a repeated failure, choose where the correction belongs, evaluate it, and update the shared setup only after review.

Ready when a later session uses an approved fact without manual prompting, superseded knowledge is handled correctly, and a proposed skill change can be evaluated, accepted or rejected, and rolled back. Claim reduced recurrence only after observing subsequent comparable runs.

6. Automation entry points - Start the workflow from Slack or Linear

Let people start work from familiar tools, such as Linear or Slack. Each authorized request should enter the same delivery system with its context, permissions, and workflow attached.

Choose a workflow that fits the change. A small fix does not need every planning stage, and an existing PR may need only Code Review or Verify.

Workflow	Use it for	Configure
Quick Lane	Bounded changes with agreed acceptance criteria	Implement, Code Review, Verify, and required action permissions
Full Lane	Features needing product decisions, design, or coordination across systems	Requirements, architecture, planning, approvals, implementation, review, and verification
Individual agent	A focused review, investigation, or verification task	Relevant context, scoped tools, expected evidence, and a completion condition

Map the request to its project, requester, workflow, and allowed scope. Preserve the acceptance criteria, return duplicate events to the existing run, and show the current owner and blocker. Full Lane can pause for product or design decisions; Quick Lane can skip intermediate gates when policy permits. Both still need the required review,

verification, and permissions. Link the verified PR and evidence back to the original request. Merge and production release remain under the team's release policy.

Ready when an authorized ticket or Slack request starts the configured workflow and reaches a PR with evidence without someone manually calling each agent. Check that approvals still pause the run, blockers return to the requester, a repeated event resolves to the existing run, and an individual-agent request stays within scope. The final handoff should link back to the original request.

Connect the components into one delivery system

Get one path working first. Can agents coordinate, run the app, fix what fails, and return a PR with evidence that a person can review?

The architecture walkthrough above shows how the layers support the work from request and planning through building, verification, and reviewed learning.

The control plane moves the work. Context informs it, verification checks it, and governance limits it. Learning improves the next run.

Choose the stages and gates your workflow needs. A small fix can take a shorter path while keeping the context, permissions, and verification it requires.

Prinevo.ai gives you these foundations out of the box. Your agents work with shared context and skills, coordinate across repositories, verify changes in a sandbox, and feed reviewed learning back into the setup. You choose the models, access, owners, and gates, and stay in control of the work.

Connect your repositories and tools, establish the domain context and runnable environment, and choose the owners and approval rules that fit your team. Then evaluate one representative change from your backlog, including what the reviewer receives and what the next run can reuse.