

How we build software **with agents**, and what keeps it under control

This is how delivery works on Oxagile projects: agents do the implementation work, and people own intent, architecture and every sign-off. What follows is the mechanism in enough detail to judge it - the artifacts, the control points, the rules that run automatically, and what is left for each team to decide. **It fixes the process, not the toolchain**: requirements name capabilities, and the last page lists today's defaults.

Two pillars

Everything in this document hangs off two questions. The first decides what every agent session has to produce; the second, what every session is allowed to do while producing it. Neither is answered once: a single change passes through many sessions, and both hold for each of them.

What are we building?

Spec-driven development

Intent is written down before code exists, as files the next stage consumes. An agent does not ask what you meant: it takes the spec at face value and fills any gap with its own assumption, silently, in code.

How do we build it safely?

Policy as code

The rules that matter are executable checks in the agent loop, committed with the code and reviewed like it. Documented rules bend under deadline pressure; enforced rules do not.

SPEC-DRIVEN DEVELOPMENT · WHAT EACH SESSION MUST PRODUCE

one artifact in, the next artifact out - the files are the contract between sessions



At each gate a named human role reads the artifact before the next session may start from it.

POLICY AS CODE · WHAT EVERY SESSION IS ALLOWED TO DO

the same hooks, permissions, checks and log apply to each session above - none of them is exempt, none of them is trusted on its own

Without the left rail an agent builds the wrong thing, quickly.

Without the right rail: the right thing, with no proof, no trail and no brakes.

Spec-driven development: four phases, four gates, then release and watch

Every feature and change request runs through four phases, and a gate closes each one. Each phase produces an artifact, the gate is a named human role reading that artifact, and the next phase starts only from what the gate let through. Nothing advances on a verbal agreement.



PHASE 01 Requirements

spec.md - functional requirements and the business rules behind them, acceptance criteria in verifiable GIVEN / WHEN / THEN form, input and output examples for the main path and the worked-through edge cases, non-functional constraints, and an explicit out-of-scope. Backlog refinement meetings are recorded as a matter of practice, and the transcript is what the spec agent drafts from - the BA edits a draft rather than re-typing a meeting. Before any plan exists the spec is interrogated, not implemented: what is ambiguous, missing or contradictory comes back as questions for the BA and, where the answer is technical, the Architect (SpecKit's **/clarify**, then **/checklist** as the unit tests of the requirements) and the answers land in the spec. Skipping this moves the questions into the code as assumptions nobody reviewed.



1 PO/BA, QA LEAD, CLIENT

Requirements are complete and match the business goal; QA confirms the criteria are verifiable and the edge cases are named.



PHASE 02 Design

plan.md - interfaces, data model, dependencies, the non-functional constraints the change has to hold, and the alternatives that were rejected. On an existing codebase this is also where invariants are named: the behaviour that must not change.



PHASE 03 Decomposition

tasks.md - atomic units, ordered, each with its own completion criterion. One unit is **30 to 60 minutes of agent work**: enough to finish inside a single clean context, small enough that a wrong turn costs one step rather than a day.



PHASE 04 Implementation

Code, tests and a pull request, from the same session. **Tests come from the criteria, not from the code**: they are written from the acceptance criteria agreed at Phase 01, and a separate session and the QA lead check them for the patterns that make a suite hollow. A unit without green tests does not exist for the next step. One vertical slice: back end and front end are implemented in parallel in their own branches against the same spec; once both are deployed to a feature environment, test automation writes and runs the end-to-end suite there, and only then does the slice go to the merge gate.



2 DEV LEAD, ARCHITECT

Decisions are justified, risks and NFRs accounted for; QA reads the plan for testability - injection points, mock boundaries.



3 ENGINEER

Dependencies are correct, nothing is lost between units, nothing is oversized.



4 ENGINEER, QA

Criteria met, pipeline green, no regressions, changes reversible, and the spec back in sync with the code that was actually written.

A failed gate returns the work to the phase that produced the defect - to clarification, planning, decomposition, or a fix and recheck. No phase is skipped forward, and a gate closed verbally did not happen: there is nothing to audit and nothing to point at when a defect is traced back.

Policy as code: what actually runs

Each rule below is a check wired into the agent loop at the point where it can still prevent something. The configuration is committed, so it holds for everyone, on every machine that clones the project - rules configured on one engineer's laptop are not policy.



ON READ, WRITE OR COMMIT Secret protection

Default-deny on secret paths - **.env***, ***.key**, ***.pem**, ***.p12**, credential and service-account files, private keys. The call is refused before the file opens: the agent receives a denial, never the contents.



ON EVERY PROMPT Context control

What enters the model's context is checked against the project's data rules. Credential-shaped strings and personal data are caught here rather than apologised for later.



ON SHELL COMMANDS Action safety

Recursive deletes outside the project tree, force pushes to protected branches and destructive schema statements are refused outright, whatever the agent believes it is doing.



ON EDIT, THEN BEFORE COMMIT Static analysis and tests

Lint and type errors come straight back to the agent as work after each edit; the tests touching the change run before a commit or pull request, so a broken branch never becomes one.



ON EVERY TOOL CALL Action log

Files touched and commands run are appended to a log kept with the project, used in review, in post-mortems, and whenever someone asks what the agent actually did.



ON EVERY TURN Session cap

Each session runs under a turn and spend cap and stops after the same failure repeats, so a looping agent halts itself, with the reason in the log, before it reaches the bill.

Stages 5 and 6: release and watch

The four gates close the build. Two stages follow the merge, and what stage 6 finds goes back into phase 1 as scope.

STAGE 05 Release

A release gate the agent cannot pass. The client's release manager authorises the deploy, and the record shows who did it and when.

STAGE 06 Watch

A deterministic watcher compares production metrics with the client's thresholds and calls the agent when one breaks. The agent diagnoses read-only, and a person decides the fix.

Mapped to Anthropic's AI-native SDLC playbook: Plan to phase 1, Design to phase 2, Build to phase 3, Test to phase 4, Deploy to stage 5, Maintain to stage 6.

WHAT RUNS UNATTENDED, AND WHAT NEVER RUNS

- ✓

Auto-allow

Reading inside the repository, running tests and linters, editing the working tree of a feature branch, committing its own steps.
- 👤

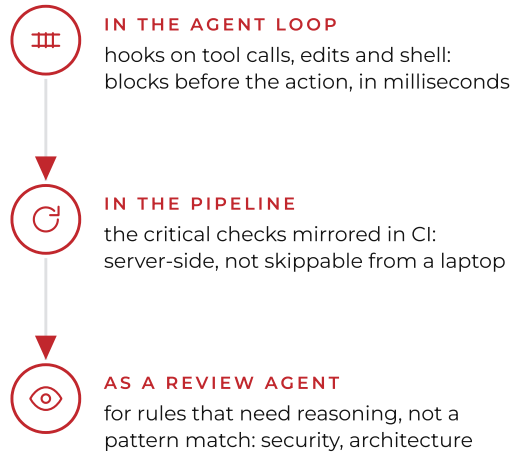
Confirm first

Adding a dependency, touching CI or infrastructure config, writing outside the project, calling an external service.
- 🚫

Blocked

Secret paths, destructive commands, force push to a protected branch, anything touching production credentials.

WHERE A RULE RUNS

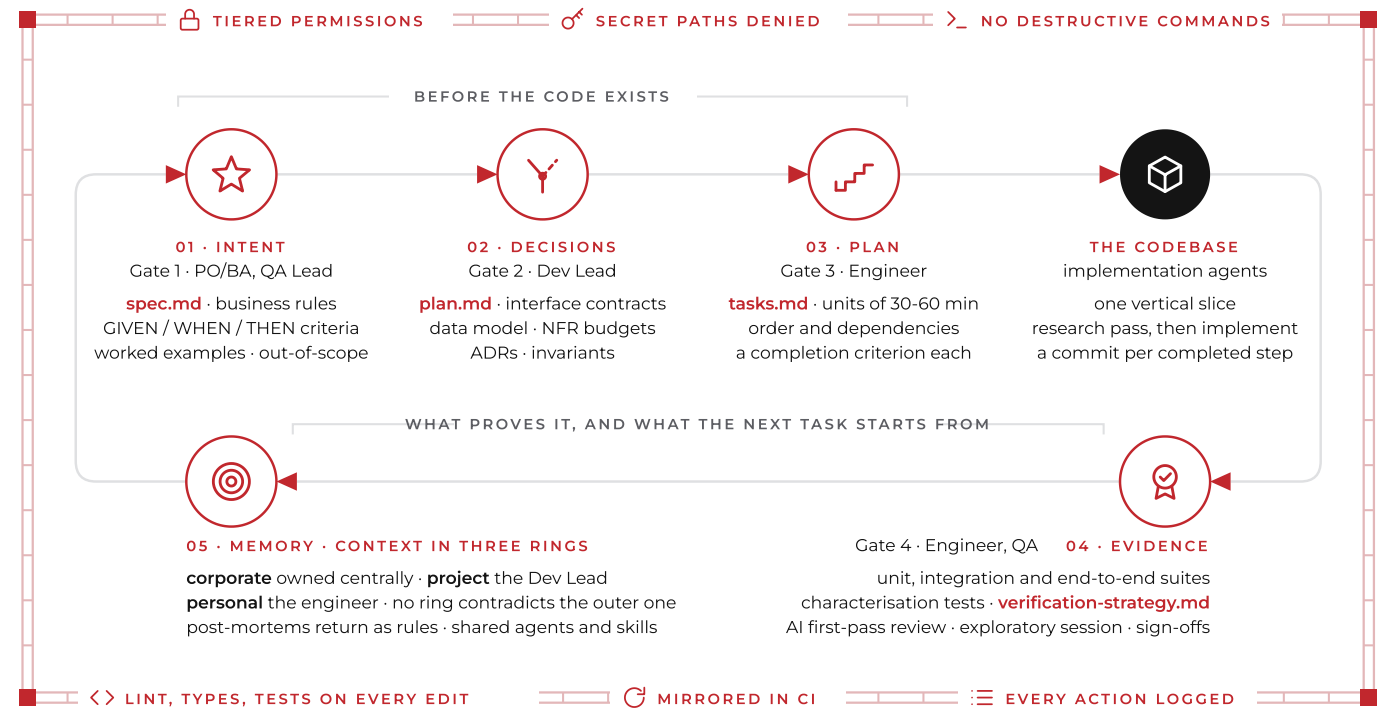


A refused action always comes back with its reason. Swallow it silently and the agent simply looks for another route to the same place.

The project harness: the infrastructure of trust around the code

The project harness is the full set of artifacts around the code - specifications, standards, tests, schemas, decision records, context, hooks and agent configuration - against which an agent checks its own work. The more complete the harness, the less has to be inspected by hand, and the more of the result can be trusted on evidence rather than on reading every line. Each station has a named human owner; the context the agent starts every task from is one of them.

III GUARDRAILS · ALWAYS ON, AROUND EVERYTHING INSIDE



A station is rarely one file: it is the set of versioned artifacts kept with the code - specs and plans, test suites, hook and agent configuration, decision records - each with a named human owner. Only the paths and the commands differ between frameworks: with SpecKit the top edge runs `/specify` · `/clarify` · `/checklist` · `/plan` · `/tasks` · `/analyze` · `/implement`, and each gate closes on the artifact the command before it produced.

SEVERAL REPOSITORIES · HUB AND SPOKES

One product, one source of truth for what it does

When a product spans several repositories - three front ends and four back ends is ordinary - the product-level truth lives once, in a **hub repository**: the business specs, epics and stories, architectural decisions, the domain glossary, the constitution. Each service repository is a **spoke**: its own context file links back to the hub, and it keeps only what is specific to that service - the technical plan and the tasks derived from the hub's spec. The spec stays technology-neutral in the hub; plans and tasks are technology-specific in the spokes, and the agent working in a spoke reads the spec from the hub through that link.

A business rule then changes in one commit, in one place, and no spoke is touched; the tracker holds a status and a link to the spec, never the requirement itself. Shared agents and skills are distributed the same way, hub to spokes, so an improved agent reaches every repository at once.

Who does what: people, and the agents they run

The roles on the team do not change; their centre of gravity does. Each role moves from producing an artifact by hand to owning the intent behind it and validating the result - and each runs its own agents, with their own instructions, context and tools. **Agents are not job titles**: they are configured executors for one stage, whose prompts live in the repository and are reviewed like code.



OWNS THE WHAT
PO / BA

Then: free-form stories in a tracker, clarified over chat. **Now**: writes the structured, testable spec that the whole chain consumes, works at epic level and leaves decomposition to the engineer. Owns Gate 1 and the boundary between "what" and "how".



OWNS THE HOW
Dev Lead / Architect

Then: wrote the hardest code and reviewed every pull request - the first thing to become a bottleneck when implementation stops being the constraint. **Now**: owns architectural decisions, the project context and the agent configuration itself. Signs off Gate 2.



OWNS THE RESULT
Engineer

Then: wrote code line by line, ticket by ticket. **Now**: frames the task, runs the agent sessions and is accountable for what lands - the agent generates, the engineer answers for it. Signs off Gate 3, participates in Gate 4.



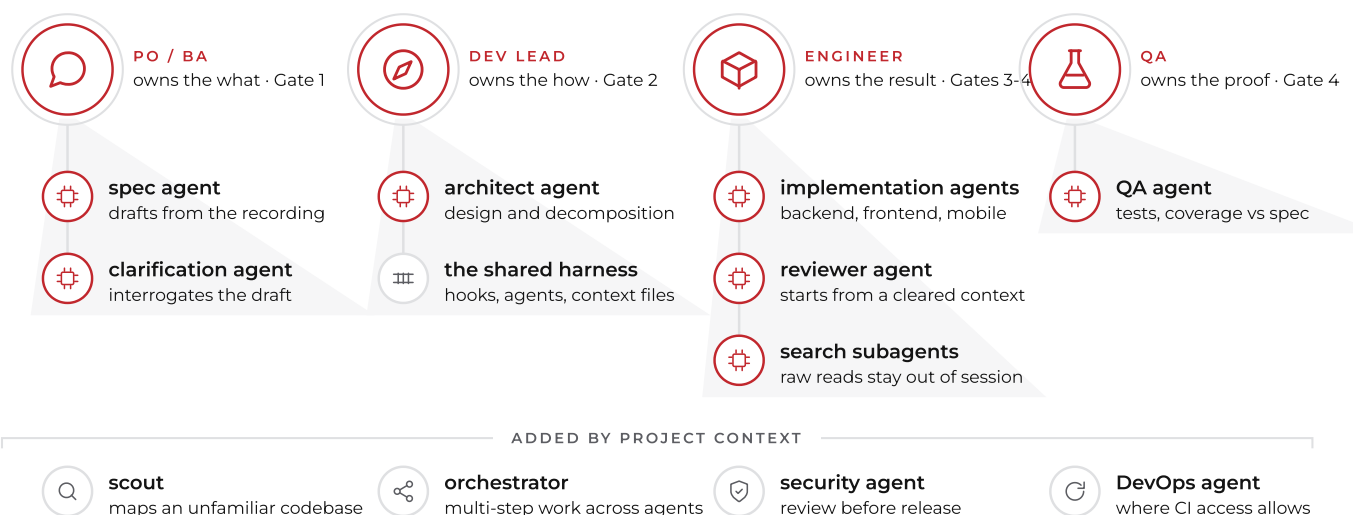
OWNS THE PROOF
QA

Then: ran regression checklists by hand. **Now**: designs the verification pipeline, reviews AI-written tests for the anti-patterns they are prone to - asserting on implementation detail, mocking the thing under test - and spends the freed time on exploratory sessions on a real environment. Works in the repository, so write access to Git is a precondition.

PEOPLE OWN THE GATES · AGENTS AMPLIFY WHATEVER THEY BRING



Deep expertise scales into more good decisions per week.
Thin or wrong judgement scales into confident mistakes, just as fast.
So the person, not the prompt, is the control point at every gate.



The agents are an amplifier, not a substitute - which is why the person, not the prompt, is the control point at every gate. A single general-purpose agent driving the whole cycle is the anti-pattern: nothing reproducible, nothing auditable. On a small team one person runs several of these - what is fixed is that each stage gets its own agent and context, not its own headcount.

Three rules every agent follows

The third is why review is a different session, not a second look by the same one.



🔍 Research before implementation

The agent reads the related files and finds the existing pattern before it writes anything. Implementation does not start until the context is loaded, or the agent invents a second way of doing what the codebase already does.

🔗 A commit per completed step

Work-in-progress states are committed too: the checkpoint matters more than the tidiness. One step can then be rolled back instead of a day, and every step is an auditable point.

👁️ The writer never reviews itself

Of the defects caught before merge, 1 in 5 is found only by this session: requirement violations, unhandled edge cases, drift from the project's own patterns.

Cost control: what a session costs, and the three levers

Everything in the context window is re-sent to the model on every turn, so spend grows with session length - and an overloaded context also degrades the output. The same discipline therefore pays twice, in quality and in cost. Three levers, in order of leverage.

⚙️ Match the model to the task

Mid tier for implementation by default, top tier for architecture and cross-cutting refactors, small tier for mechanical work - index regeneration, classification, log parsing. Routing happens inside the session too: the orchestrator delegates by task class, and the rules live in the project memory file.

🗑️ Keep the context lean

Standing instructions stay short, because every line is paid for again in every future session. One task per session, a clean context between unrelated tasks, and searches isolated in a subagent that returns a summary rather than the pile it read.

📈 Watch the spend

Each session runs under a turn and spend cap, so a loop stops on its own. A weekly baseline shows the trends: a session that swallowed half the repository, a routine task at three times its usual cost. Those are process bugs surfacing as cost.

WORKED EXAMPLE - THE BIGGEST LEVER, PRICED

Navigation instead of ingestion

One architectural question, priced two ways.

ANSWERING ONE ARCHITECTURAL QUESTION

Grep, open the files, grep again



~10,000 tokens

One query against the graph



~400 tokens returned in under 0.3 seconds

ABOUT 25 TIMES LESS CONTEXT FOR THE SAME QUESTION

60 to 125x fewer tokens than a wide repository sweep

🔍 The problem

Before an agent changes anything it has to work out how the pieces relate. With only text search it does that by reading: grep a name, open what matched, grep the next name it found there, open those too. Everything it opens stays in the **context window** and is re-sent to the model on every turn, so one question - "how does payment capture reach the rights service?" - costs on the order of 10,000 tokens before any work starts. A sweep across an unfamiliar repository runs to 25,000-50,000.

🔗 What we do instead

Index the relationships once. A **semantic graph** is extracted over the repository's docs and code: **nodes** are documents, concepts and symbols, **edges** are the references and dependencies between them. One workspace indexed this way came to roughly 700 nodes and 1,000 edges. Extraction runs on a local open-weights model at no API cost, or on a cloud model for a fraction of a cent per document, and refreshes as the repository changes.

📈 What changes

The agent stops reading to find out and starts asking. A relationship query returns that part of the graph - the two endpoints and what sits between them - in a fraction of a second, at about 400 tokens. The same move works for "what depends on this module" and "where is this rule defined": the answer is a subgraph, not a pile of files.

What is standardised, and what each team decides

The standard is a process, so it is written as requirements that can be audited. The choices below it are deliberate: tooling moves faster than any document, and a project that cannot use our default must still be able to comply.

MANDATORY ON EVERY PROJECT

- 01 **An SDD framework is chosen and recorded** in the project context, and the four phases are run in order. No phase is skipped, and each artifact is a file.
- 02 **The spec carries the five sections** - requirements, verifiable criteria, worked examples, non-functional constraints, out-of-scope. A spec without criteria is not a spec.
- 03 **Four gates, each with a named owner** and a recorded trace - a review comment, an approval, a decision record. The purpose of the check is fixed; the artifact that carries it is not.
- 04 **Context at three levels, none empty**, with the project level owned by the Dev Lead and kept modular.
- 05 **The policy baseline is configured in the repository**: secret protection, action safety, static analysis and tests on change, context control, agent action log.
- 06 **One agent per stage**, prompts versioned in the repository, with a research pass before implementation, a commit per step, and review by an agent that did not write the code.
- 07 **The delivery baseline is in place** before agents are let in: review before merge, CI on every change, static analysis, tests landing with the code, a reproducible deploy to non-production, infrastructure under review where it matters, observability, secrets in a managed store, pinned dependencies.

CHOSEN PER PROJECT

SDD framework

Speckit as the default - closest to a standard SDLC, its commands map onto the four phases. **Specsmd** where a team wants the process restructured around bolts rather than sprints. **BMAD** where a prescribed Analyst / PM / Architect / Dev / QA role model fits. **Superpowers** for a harness-native full cycle, **GSD** for small and exploratory work. Anything else that keeps the four phases and the gates intact.

Agent harness

Any tool clearing the capability bar: native tool calling through the provider API, tiered permissions per tool set at project level, context management at the window limit, an auto-loaded project memory file, resumable session state. We scored six against twelve parameters: **Claude Code** is our default and **OpenCode** clears the same bar; **Codex CLI** and **Cursor** pass the core and become candidates once their hooks and subagent support is confirmed; an IDE completion tool such as **GitHub Copilot** is not a harness - it misses the permission and session requirements the rest of this document rests on.

Vendor, model, region

Chosen against the project's contractual and regulatory constraints - enterprise terms directly, or hosting pinned to a region. More than one is in production across the portfolio, and the process does not change when the choice does.

Enforcement vehicle

Harness hooks, CI jobs or both. The behaviour is mandated, the mechanism is not: client pipeline access varies.

Engineering choices

Stack, branching model, test strategy, architectural style. Where your own pipelines and review tooling apply, we run inside them.

ITEM 07, SPELLED OUT · IN PLACE BEFORE THE FIRST SPEC

The pipeline is part of the harness

An agent produces changes faster than a team can inspect them by hand, so the path from commit to a running environment has to be automated before agents are let in - otherwise their mistakes surface too late. This is a pre-flight check on every project, not a later improvement.

Version control and review

Every change enters the main line through a reviewed pull request on a protected branch; the policy hooks, linter and test rule live in the repository, not on a laptop. Secrets in a managed store, dependencies pinned.

CI and CD

Build, static analysis and the agreed test level on every pull request, with red blocking the merge. A reproducible deploy to a non-production environment - staging or a per-slice feature environment - so the end-to-end suite runs against deployed code. Database migrations versioned and applied by the pipeline.

Infrastructure as code

Where infrastructure is a significant part of the project, it is declared in the repository and reviewed like code - Terraform, Pulumi or equivalent - with image scanning before deploy and a documented, verified rollback. An agent then changes environments through the same gates as it changes code.

Agents changed the cost of producing code, not the cost of being wrong about what to build. **The approach puts the expensive decisions in front of a named human role at four gates, and enforces the rest in code** - which is what makes the speed usable rather than merely impressive.