CN

Definitive edition

Founder-Led Engineering Ops

A minimum operating system for building serious software
spec-driven, agent-orchestrated, and governed by quality

Founder. Builder. Operator.

An atemporal operating manual with a current stack snapshot — April 2026

Carlos Navarro Cabrera
0xnavarro.com

- Spec-driven development
- Frontier-model implementation
- Structured audits and QA
- Final human authority

© 2026 Carlos Navarro Cabrera

Executive Summary

Artificial intelligence has radically lowered the cost of producing code, but it has not solved software engineering. What is often marketed today as speed — prompting, vibe coding, or instant feature generation — mostly produces artifacts. Serious software still requires specification, architecture, traceability, verification, and judgment. The difference is that an increasing share of the work is now executed by agents rather than people.

This document describes the minimum system I currently use to build software in a founder-led environment. It is spec-driven, agent-orchestrated, and governed by quality. It starts from a simple conviction: serious software is not improvised; it is specified, executed, and verified.

In practice, that means working from formal specifications, separating design from execution, dividing development into phases and branches, assigning distinct roles to different agents, auditing with multiple model families, and preserving final human authority over intent, security, risk, and release. The goal is not to maximize raw output; it is to maximize useful throughput with control.

As of April 2026, the current stack snapshot uses Kiro IDE and Kiro CLI as the main operating environment; Claude Opus 4.6 as the primary implementation and iterative review model; GPT-5.3-Codex with extremely high reasoning as the final auditor; Playwright MCP for browser QA; and a set of project-specific MCPs and skills for database work, observability, security, and contextual automation.

1. Why software engineering needs a new operating model

During 2025 the market popularized vibe coding: the idea that informal intent was enough for AI to produce useful software. That approach may be acceptable for prototypes, demos, or quick experiments, but it breaks down when real complexity appears: dependencies, architecture, edge cases, security, QA, scope changes, traceability, or maintenance.

The underlying problem is that many people have confused model output with engineering. Generating code is not the same as building software. A model may write functions, screens, migrations, or tests, but that does not mean the resulting system is aligned with a business problem, a coherent architecture, or an acceptable quality standard.

The answer is not to stop using AI. The answer is to restore discipline. If software engineering previously relied on specifications, reviews, QA, ownership, and governed releases, those principles now need to be recovered and adapted to an agentic context. Instead of employees with different functions, we can now have agents with separated contexts, responsibilities, and missions. What does not disappear is the need for governance.

That is why this model does not romanticize improvisation. It is designed to create an execution system where AI accelerates work without destroying process. The promise is not more lines of code. The promise is reliable systems built with speed, evidence, and control.

> *“Serious software is not improvised; it is specified, executed, and verified.”*

2. Core Thesis

My core thesis is that the agentic era does not eliminate software engineering; it makes it more important. Once the cost of producing code falls, the bottleneck shifts toward specification quality, design quality, execution correctness, verification robustness, and final judgment.

Three principles follow. First: serious software is not improvised; it is specified, executed, and verified. Second: agents replace work, not architecture, judgment, or business intent. Third: models also have biases, so a robust system requires more than one model family to reduce blind spots.

This changes the central question. The question is no longer “which prompt produces more code?” It is “which system turns business intent into software that is correct, verifiable, and ready to ship?” That is the difference between apparent productivity and real construction.

FOUNDER-LED ENGINEERING OPS

3. From business problem to working branch

The system begins before code. It begins with the problem definition, the user, the operational friction, the market opportunity, and the quality bar required. If the original intent is vague, the rest of the process becomes contaminated.

The first layer is specification. A professional specification is not defined by nice section titles alone. It must reduce ambiguity to the point that downstream design and execution do not require constant structural corrections. It must capture functional and non-functional requirements, actors, user stories, use cases, main and alternate flows, constraints, acceptance criteria, edge cases, and external dependencies.

The second layer is design. This is where the architecture is defined, system boundaries are fixed, decisions are documented, ADRs are added when needed, and traceability between requirements and components is established. Architecture and business decisions should act as the source of truth. Code comes later and must reflect them rather than replace them.

The third layer is the implementation plan. The plan translates design and requirements into phases, tasks, subtasks, and completion criteria. In my current workflow, each meaningful phase is executed in its own branch. A branch is not considered complete because it “works”; it is complete because it has passed implementation, audit, QA, final review, and merge readiness criteria.

This sequence — requirements, design, implementation plan, phase branches — is what makes auditing, correction, and scale possible. Without it, the system collapses back into improvisation assisted by AI.

FOUNDER-LED ENGINEERING OPS

© 2026 Carlos Navarro Cabrera — 10 April 2026

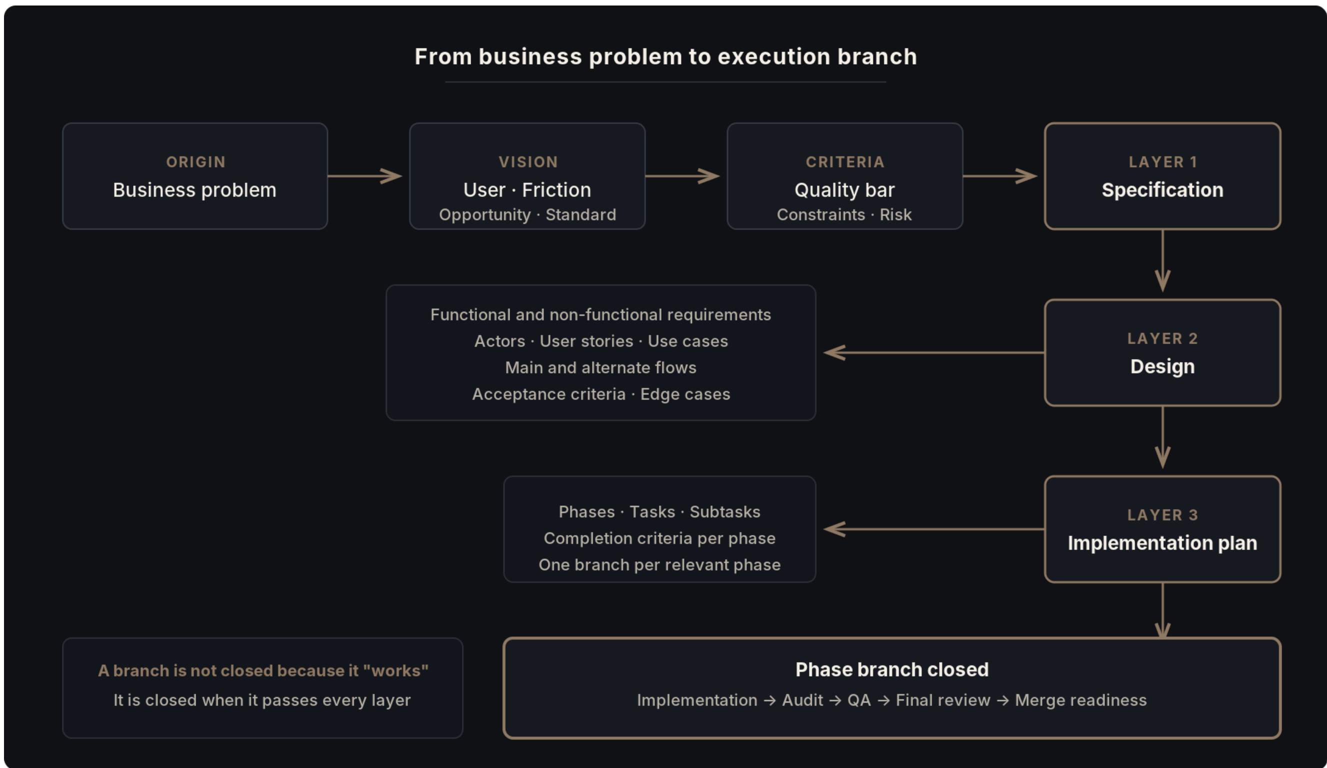

Diagram 1. From business problem to execution branch in a spec-driven workflow

4. Roles in the current system

The current minimum system has two role layers: one human orchestration role and a set of specialized agent roles. The human orchestrator decides which business is being built, which problem is worth solving, which architecture is acceptable, which risks are tolerable, and when a version is genuinely ready. That role remains central because final responsibility, business vision, and acceptance judgment remain human.

Around that role, several agents execute specialized work. The implementation agent owns the branch and modifies the code. The technical audit agent reviews implementation iteratively, detects findings, evaluates coherence with design, and looks for regressions or specification mismatches. The QA agent validates observable behavior through real flows rather than theory. The final auditor reviews the branch again from another angle and with another model family, focusing on merge readiness and residual risk.

At times I add other agents or subagents for specific tasks, but these form the minimum stack. The point is not to maximize agent count. The point is to make mission, context, and ownership explicit. Splitting sessions by role prevents the same model from self-justifying within a single context and improves verification quality.

In Kiro AI I also work with subagents launched by the primary agent itself. This is especially useful in audit. The auditor does not act as a single monolithic voice, but as a coordination layer that decides which subagents to open based on the branch diff, the risk profile, the touched area, and the nature of the change. Some focus on architecture and traceability to the design; others focus on security, regression, or spec mismatch. The main audit agent then consolidates those outputs into deduplicated, prioritized findings.

© 2026 Carlos Navarro Cabrera — 10 April 2026

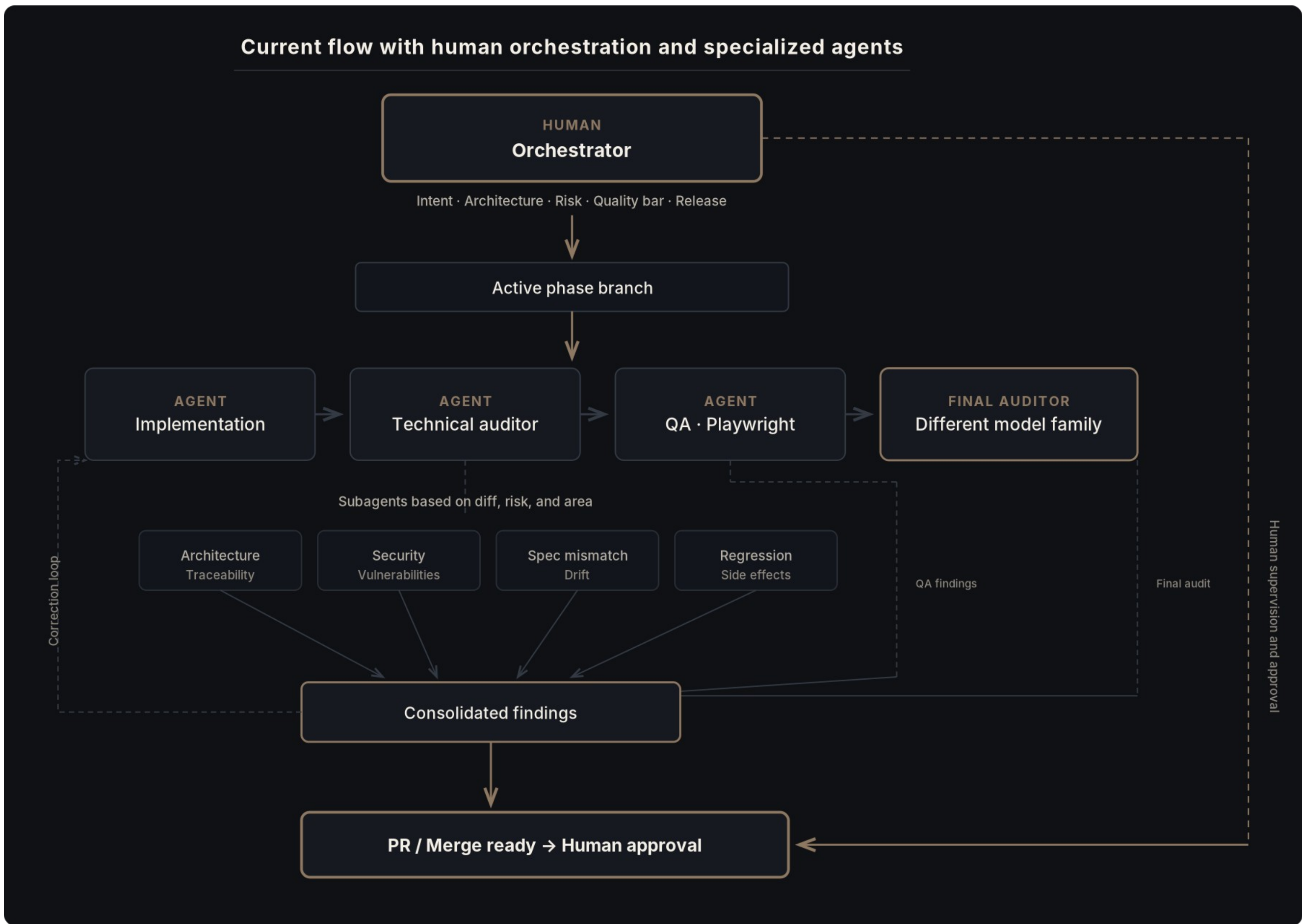

Diagram 2. Current flow with human orchestration, specialized agents, and audit subagents

SNAPSHOT

Minimum role map

Role	Primary mission	Why it is separated
Human orchestrator	Defines intent, acceptable architecture, risk, and release criteria.	Preserves business vision, accountability, and final judgment.
Implementation agent	Executes the phase and changes branch code.	Concentrates execution ownership and avoids diffuse editing.
Technical auditor	Generates iterative findings and looks for inconsistencies.	Introduces another mission and another angle within the workflow.
QA agent	Validates real behavior through browser flows and evidence.	Checks observable experience, not just code structure.
Final auditor	Evaluates merge readiness and residual risk.	Uses another model family to reduce bias and blind spots.

FOUNDER-LED ENGINEERING OPS

© 2026 Carlos Navarro Cabrera — 10 April 2026

5. Why the system uses frontier models

At the moment, the apparent savings from cheaper models are often smaller than the real costs they generate. A weaker model does not just make worse mistakes; it also drags those mistakes into later stages, multiplies audits, increases rework, and raises the risk that defective software reaches production.

That is why my operating principle is to use frontier models whenever budget allows. This is not about luxury. It is about total cost of correction. In the current market, paying for stronger reasoning and better judgment is usually cheaper than paying for rework.

As of April 2026, Opus 4.6 is the primary model for implementation and iterative review. It is fast, consistent, and deep enough to follow complex specifications, execute phased plans, and maintain operational continuity. GPT-5.3-Codex, with extremely high reasoning, acts as the final auditor. I use it because it repeatedly catches issues the other model misses. That divergence is not a flaw in the workflow; it is the reason to work with two distinct model families.

The principle is not just to use the best model available. It is to combine at least two model families in order to reduce training bias, context bias, and recurring blind spots. System quality does not come from a single magical model; it comes from governed interaction between several.

CURRENT STACK SNAPSHOT — APRIL 2026

Current stack snapshot

Layer	Tool	Why it is used
IDE	Kiro IDE	Spec-driven development; main phase execution.
CLI	Kiro CLI	Audit, QA, isolated sessions, and operational control.
Modelo principal	Claude Opus 4.6	Implementation and iterative review.
Auditor final	GPT-5.3-Codex	Extremely high reasoning; final pre-merge review.
QA browser	Playwright MCP	Real flows, functional evidence, and regressions.
Skills	Project-specific	Project-specific skills; not generic context-free templates.
MCPs	DB / observability / security	Instrumental tools according to domain and environment.

© 2026 Carlos Navarro Cabrera — 10 April 2026

6. Tooling rationale

Kiro IDE is part of the system because its structure aligns with spec-driven development. It allows requirements, design, and tasks to exist as first-class artifacts instead of loose notes or disordered prompts. In my practice, that alignment matters more than any isolated feature.

Kiro CLI plays a different role. I use it for more isolated and operational sessions: audits, QA, targeted reviews, and tasks where context separation matters. It also gives me a cleaner way to manage sessions, setup, and cost. The IDE and the CLI are not used because they are fashionable; they are used because they support different roles within the system.

In that instrumental layer, Kiro AI also lets me launch subagents from the main agent. This matters in audit because the system can dynamically decide which specialized lenses it needs based on the change set: architecture, security, contracts, regression, traceability to the spec, or coherence with the design. The visible session is still one, but the actual review may come from a topology of subagents that gets consolidated into a single actionable output.

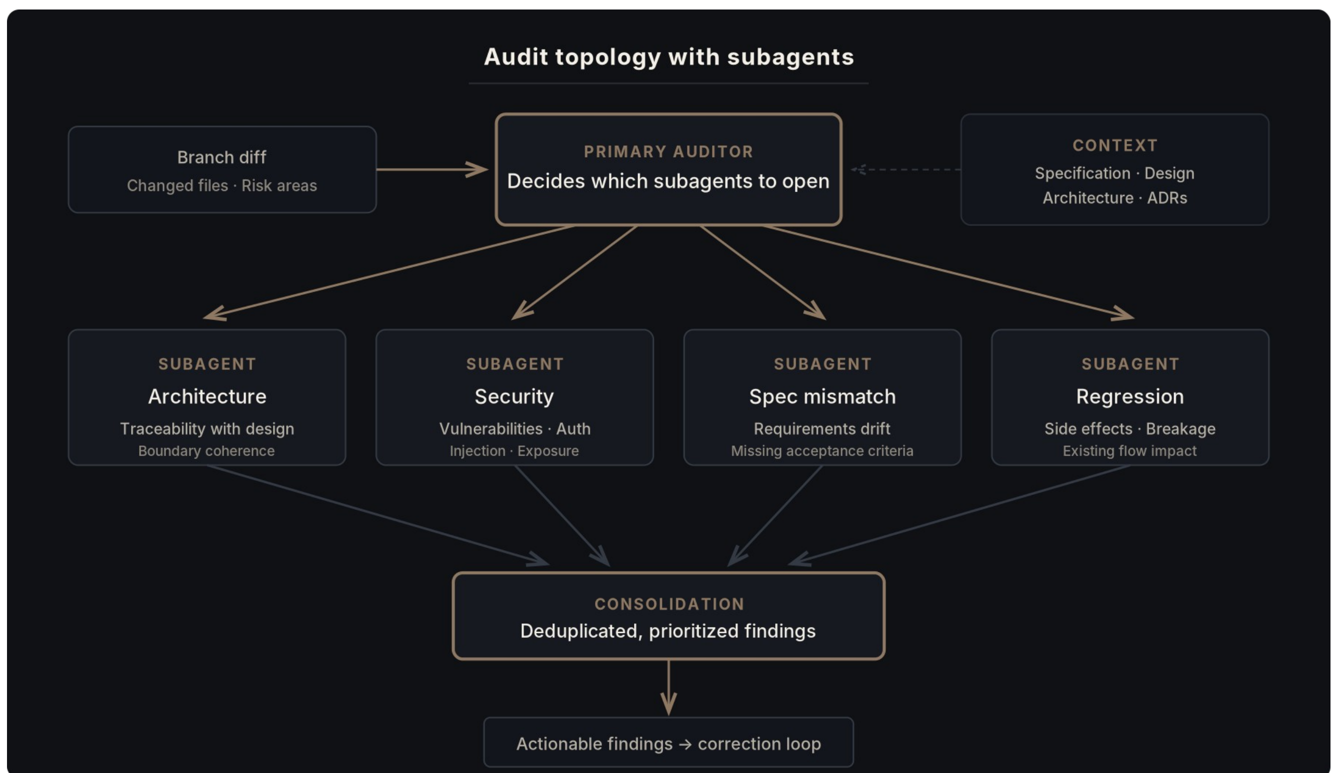

Diagram 3. Audit topology with subagents for architecture, security, spec mismatch, and regression

Playwright MCP is the critical component for browser QA. The goal is not to simulate quality but to interact with the application the way a user would, traverse flows, observe real failures, and produce evidence. The system can also incorporate other MCPs for databases, observability, security, dependencies, logs, or infrastructure. They are not accessories; they are the instrumental layer that connects agentic execution to the real environment.

Skills also matter, but not as generic interchangeable prompts. In my workflow, the valuable skills are the ones adapted to the project, the domain, and the quality bar I want to enforce. Reuse is only useful if it preserves context and maintains standards.

© 2026 Carlos Navarro Cabrera — 10 April 2026

7. Quality governance

Quality is not delegated to hope. It is governed. In this system, the formal unit of control is the finding: a structured, traceable, actionable observation produced by audit, QA, or final review. A finding needs severity, evidence, a reference to the affected area, a recommendation, and a state. If it cannot be turned into a formal finding, it is probably not yet clear enough.

The central rule is that fixed does not mean closed. A finding is only closed when another layer verifies the result. This preserves separation between execution and validation and prevents the system from self-approving.

The other major rule is that architecture and prior decisions are the source of truth. If the code contradicts the specification or the design, the code is wrong even if it compiles. AI produces output quickly; that is precisely why a strong governance layer is required to filter, compare, and reject.

In this system QA does not arrive as late-stage cosmetics. It arrives as a mandatory functional layer. And final audit does not exist to repeat the same review one more time; it exists to introduce another perspective before merge. That closure is what creates operational confidence.

FOUNDER-LED ENGINEERING OPS

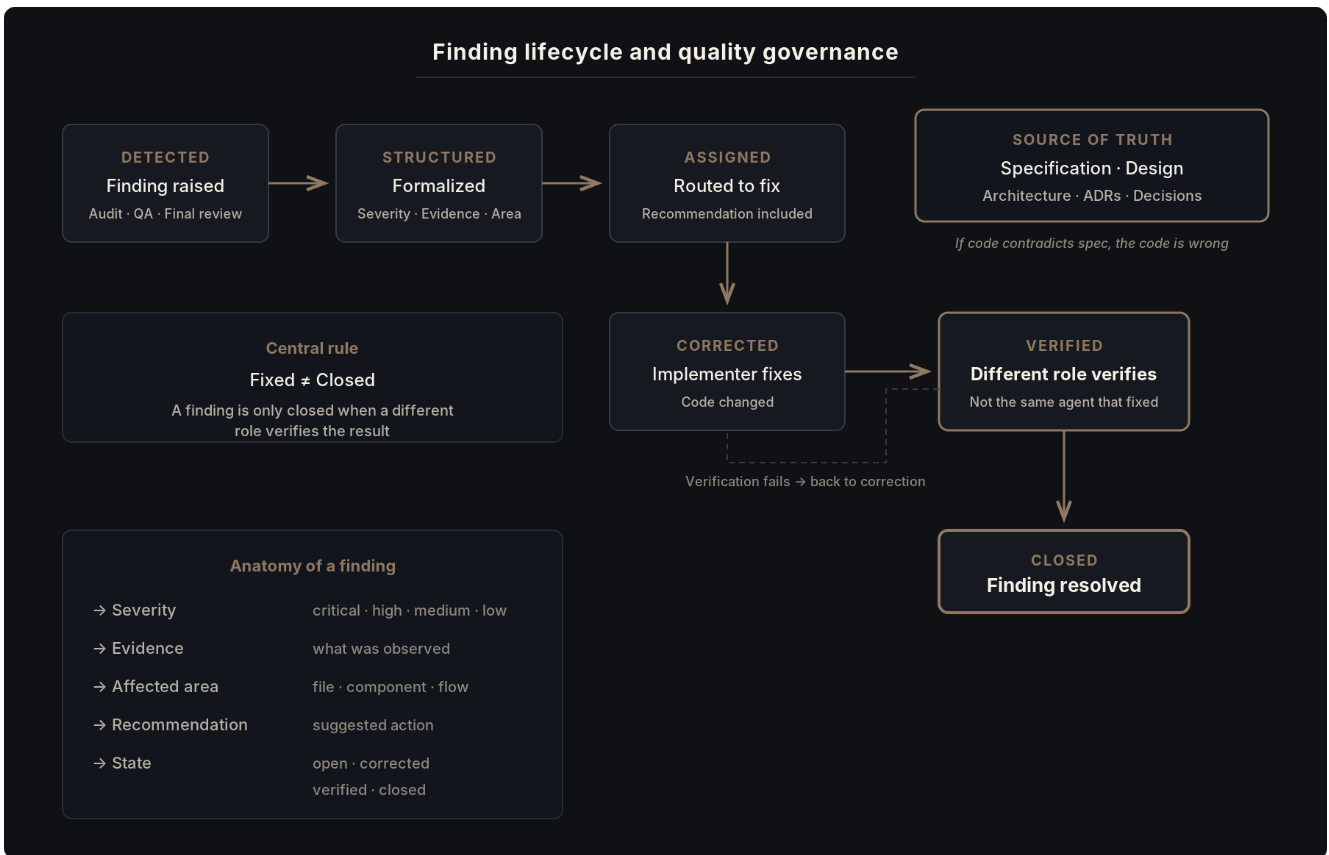

Diagram 4. Finding lifecycle and quality governance

8. Snapshot of the current workflow

The current workflow behaves like a governed loop. The human orchestrator defines intent, constraints, and judgment criteria. Specifications, design, and the implementation plan follow

from that. Each phase is executed in its own branch. The implementation agent builds; the technical auditor reviews; the implementer corrects; QA validates real flows; and a final auditor, using another model family, issues the last verdict before PR or merge.

This is not a system designed to maximize symbolic autonomy. It is a system designed to maximize practical reliability. Given the current state of technology, I prefer a workflow with multiple verification layers and final human authority over a narrative of total autonomy without guarantees.

9. Toward an agentic orchestrator

The natural next step is to reduce some of the manual coordination burden. That does not mean eliminating the human role; it means automating the lower operational plane. An agentic orchestrator could receive business vision, target architecture, quality policies, and risk criteria; compile them into specifications and plans; route work across specialized agents; consolidate findings; and manage gates before PR or release.

That system would be valuable precisely because it would preserve the current structure: business vision, architecture as source of truth, separated roles, different model families, real QA, and final approval. The change would be in who moves the workflow between stages, not in eliminating the stages themselves.

In other words, the most interesting future automation is not an AI that merely writes more code. It is an orchestration layer that turns business intent into governed software execution. That is the path from a founder-operated microfactory to a partially automated engineering operating system.

© 2026 Carlos Navarro Cabrera — 10 April 2026

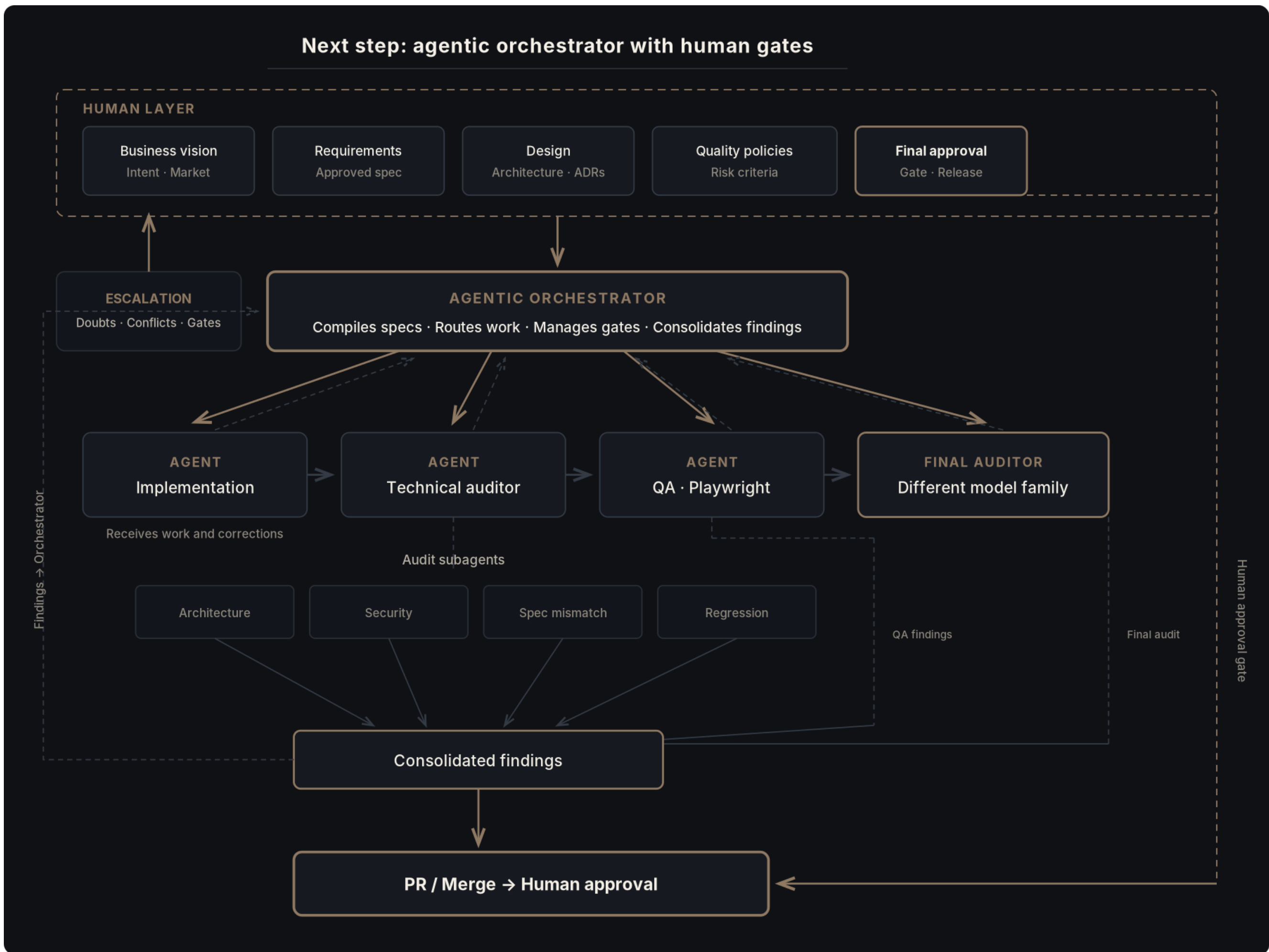

Diagram 5. Next step with an agentic orchestrator and human gates over the same agent stack

10. Operational conclusion

This system does not exist to produce more code. It exists to produce better software. Its purpose is to turn frontier models and agentic execution into an operational discipline: specify before implementing, separate roles, verify through multiple layers, and preserve human authority over architecture, quality, security, and release.

The result is not speed alone. It is speed with traceability, auditability, QA, evidence, and control. That difference is what makes it possible to build real companies on top of AI-generated software rather than fragile prototypes.

FOUNDER-LED ENGINEERING OPS

11. Execution thesis

AI has not eliminated software engineering; it has made formal engineering more urgent. The market’s mistake has been to confuse output with construction. Generating code is not building software. Building software requires professional specification, architecture as source of truth, phased execution, structured audit, real QA, and clear governance.

Agents replace operational work, not judgment. Models accelerate execution, but they do not replace intent, business design, security standards, risk judgment, or final accountability. That

is why the correct system is not a chain of prompts; it is a minimal engineering organization under governance.

12. Business vision and the founder technical manifesto

In the new paradigm, the winners will be companies and founders who can turn intent into systems with more clarity, more speed, and better control than everyone else. Software itself will become cheaper as a production capability, but competitive advantage will continue to come from deciding what to build, for whom, with which architecture, at what quality bar, and with what market timing.

That changes the role of the technical founder. The founder is no longer only the person who manually writes the most code. The founder becomes the one who frames the right problem, sets the standard, designs the architecture, decides acceptable risk, and governs a factory of execution amplified by agents. The edge is not just in programming; it is in translating business vision into reliable systems faster than others.

Within three to five years, more of this workflow may be automated. But business intent, market interpretation, security judgment, and release accountability will remain deeply human. AI may commoditize software production; it does not commoditize responsibility or the ability to interpret reality.

That is why this document is not a defense of hype. It is a defense of discipline. In an era where code generation becomes cheaper every year, the difference will belong to those who use it to build companies, not just demos.

13. Final principles

1. Serious software is not improvised; it is specified, executed, and verified.

2. Generating code is not the same as building software.

3. Agents replace work, not architecture, judgment, or business intent.

4. Architecture and decisions are the source of truth; code must reflect them.

5. For serious software, frontier models should be the default standard.

6. One model family is not enough; multiple families reduce bias and blind spots.

7. Human governance remains non-negotiable in quality, security, and release.

8. Useful speed is not raw output; it is verifiable throughput with control.

9. The technical founder of the future is, above all, an architect, operator, and systems decision-maker.

© 2026 Carlos Navarro Cabrera — 10 April 2026