Mind Rocket Services

The New Agentic Organisation

Vision · Journey · Roadmap

February 2026

Part 1

The Vision

The Vision — Contents

Executive Summary 3
Why Now? The Case for Change 4
Strategic Principles 6
The Vision: What We Build 8
Conclusion: If Not Now, When? 18
Appendices 19

Executive Summary

Software development is becoming agentic. Modern software organisations are adopting agentic software development lifecycle (ADLC) practices. We now orchestrate agents that build the code on our behalf. This document shows how to do this successfully.

Why This Matters: The software development lifecycle is the tightest constraint for most technology organisations; the ability to create software limits business growth. Agents remove this constraint. When code generation becomes unlimited, we unlock novel ways to improve how we work.

This document is structured around three principles and three shifts in perspective. Principles are decision heuristics: when we face a trade-off, they tell us how to choose. The shifts in perspective are the changed mental models that shape how we build in an agentic era. The shifts describe what must change; the principles help us decide how to realise that change. Together they make up the vision of what good looks like.

Three Principles: Elevate the Next Constraint (focus on the binding bottleneck, then the next; the journey doesn't end). Everything as Code (if it's important it's code; the frontier is codifying the business problem so agents can write solutions at run-time). Humans Above The Loop (shift from reviewing every line to governing the verification systems that assure trust at scale).

Three Shifts in Perspective: 1) Vibe coding and rigorous engineering quality are not opposites to choose between; they are distinct modes that both have their place in the workflow. 2) Ways of working should be redesigned around how agents operate: context not instructions, specifications not conversations, code not documents. 3) We are not trying to reach a neat finished end state; we are learning to operate confidently within ongoing change.

Expected Outcomes: Business: value delivery compresses from quarters to weeks; prototypes reach stakeholders in hours, not months; technology becomes a growth enabler rather than a constraint. Technology: development cycles compress from two-week to two-hour sprints; trust is assured through engineered verification; the running product reflects today's understanding of the problem, not assumptions from six months ago. Developer experience: humans shift from repetitive execution to creative problem-solving; agents handle designing, coding and routine tasks; developers spend their time on architecture, intent and the work that requires judgement.

Implementation Approach: Make no mistake, we're at the start of the next industrial revolution. The pace of change might appear to be incomprehensibly fast, but the gain in productivity hasn't yet followed the step-change in opportunity, because it's a very large step. We'll deliver this extraordinary transformation in smaller comprehensible steps matched to our ability to execute.

Timeline: We're learning by doing, starting now. Early phases focus on establishing the four-phase workflow and agentic toolchain. Mid-term phases mature our engineering practices and guard rails. Long-term phases optimise for swarm behaviours and full agent orchestration.

Investment: Resource commitment reflects the strategic importance of this transformation. Operating at the forefront of development practices means investing in dead ends, paying the innovation tax, learning by doing. The payoff will be demonstrating what's possible when development constraints disappear.

Vision-Journey-Roadmap

This Vision sets out the case for change and what the target state will look like. The companion Journey describes how organisations should transform. Finally, the Roadmap describes how to plan and steer the implementation.

Why Now? The Case for Change

Current State Assessment

Where are we? Three key problems define the current state.

Humans are in the loop for everything, so the loop is slow. Every task (thinking, designing, coding, testing, reviewing) now comprises agents queueing behind people. This is why the path from problem to working software still takes months or quarters. Prototypes still feel like side-hustles with no clear path into production. Speed and rigour still feel like opposites and teams are forced to choose. They shouldn't have to. The answer isn't to remove humans; it's to move them to a higher level of abstraction where doing is reframed as shaping intent, practising craft, governing risk.

Constraints migrate; if we don't keep up we'll measure the wrong things. Today's bottleneck is human bandwidth. Agents will relieve that but the constraint won't disappear; it will migrate to review, then deployment, then intent clarity, then something we haven't anticipated. We compound this by measuring what's easy to count (velocity, throughput) rather than what matters (Are we building the right thing? Will the team sustain this pace? Where is the constraint moving next?).

Knowledge is scattered and stale. Artefacts live in Notion, PowerPoint and Excel, snapshots that drift the moment they're created. Only code repositories stay true; they reflect what's deployed and used by customers. When agents need context to do their work, scattered and stale knowledge becomes a direct constraint on output quality.

Market and Technology Drivers

What's changed? Agentic AI has passed a tipping point and everyone has noticed. Easy tools enable us to vibe ideas into designs and code; we can chat with an agent and get functioning code back. We are rapidly getting to the point of giving an agent a task and getting functioning code back. That is the essence of an ADLC - Agentic Development Lifecycle.

The autonomy horizon defines how long agents can do work reliably. Current frontier models achieve 80% success on fifteen-minute tasks, falling to 50% on one-hour tasks. This capability doubles every seven months, potentially every four months recently. By 2027, agents may handle week-long tasks autonomously. The boundary between what agents can do alone versus what needs human oversight moves predictably upward. Human involvement becomes about specifying the work in a way that agents can execute and produce reliable results.

This creates unprecedented opportunities. When code generation ceases to be a constraint, when it becomes effectively unlimited, we can find opportunities to use software in more novel ways to improve how we work. The question becomes: what should we build?, not can we build it?

Competitive Advantage Opportunity

Why now? We're in an Engel's Pause: a period where transformative technology exists but its economic impact has not yet materialised. During the Industrial Revolution, steam engines existed long before factories reorganised to exploit them. The "productivity paradox" of the 1980s and 90s saw massive IT investment with little measurable output gain until businesses redesigned their processes around the technology.

Agentic AI is in the same position now. The capability is here, but the ways of working, the processes, the organisational structures and the measurement frameworks have not caught up. Organisations that close the gap first, that reorganise around the technology rather than bolt it onto existing practice, will gain significant competitive advantage. Early adopters can demonstrate to the market what's possible when development constraints disappear. They prove that agentic development works at scale, with real products, serving real customers.

The Constraint Migration Challenge: AI adoption is not just about tool rollout; it's about systematic constraint elimination. Accelerating code generation doesn't eliminate bottlenecks; it shifts them downstream. As agents produce code faster, pressure builds on code review, testing and deployment. Early adopters who proactively eliminate constraints see individual productivity gains translate to organisational outcomes. Teams that focus only on AI tool adoption are surprised when metrics don't improve.

The business value is clear: remove the constraints, and the entire business accelerates. More features. Faster iterations. Higher quality. Lower costs. Technology becomes a growth enabler rather than a constraint.

Strategic Principles

Three principles guide every decision in this transformation. The shifts in perspective describe what changes; the principles resolve trade-offs and help us decide how to implement those changes.

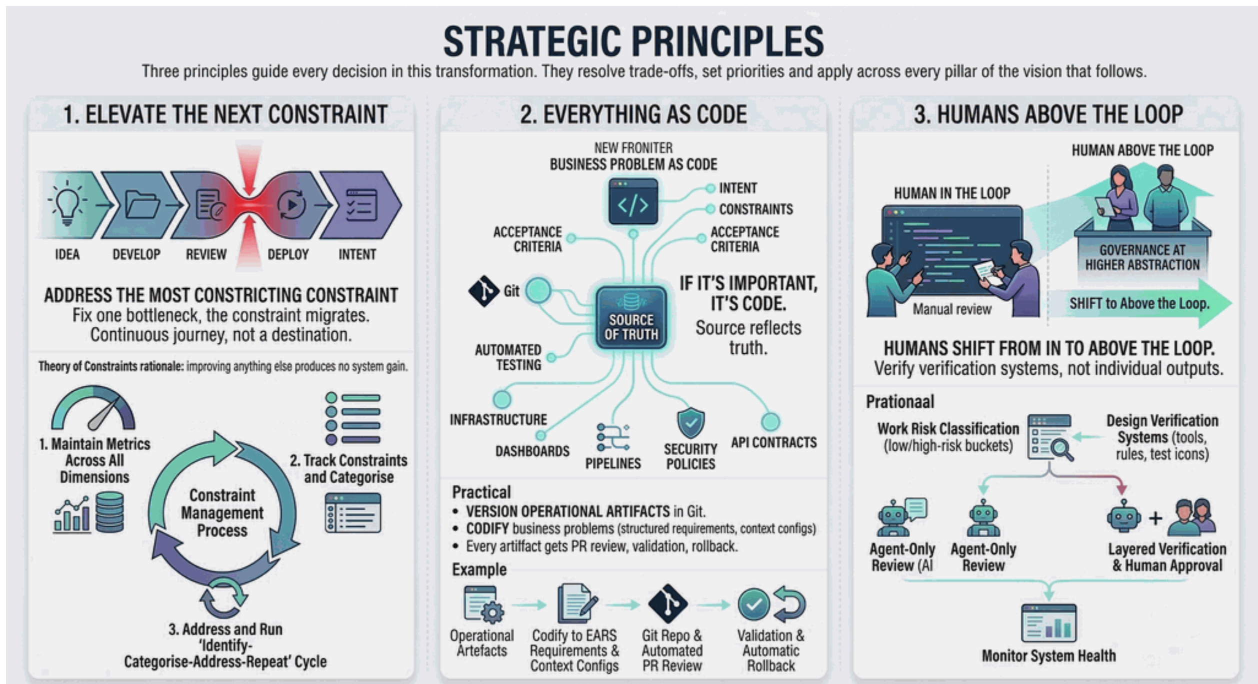

Figure 1: Strategic Principles

Elevate the Next Constraint

Principle: The only next action that yields real benefit is one that addresses the most constricting constraint. But constraints migrate: fix one and the bottleneck moves. New DevX is not a destination; stay on the journey.

Rationale: Borrowing from the Theory of Constraints: improving anything that isn't the bottleneck produces no system-level gain. The trap is assuming this means only fixing things. Three capabilities gate progress, and each can be the binding constraint at any given moment. **Prioritisation**: do we know which constraint is most constricting right now? **Discovery & Learning**: are we generating the insight needed to inform that judgement? **Execution**: can we act on what we've learnt, fast enough, at sufficient quality? And because constraints migrate as you address them (from code generation to review, from review to deployment, from deployment to intent), this is a continuous discipline, not a one-off fix.

Practical Implications: Maintain metrics across all three dimensions so you can see which capability is currently limiting throughput. Track constraints across categories (Tooling, Codebase & Architecture, Documentation, Workflow, Governance, Training, Culture) and run a tight cycle: Identify → Categorise → Address → Repeat.

Example: Execution metrics look healthy (cycle time is under two days) but Discovery & Learning metrics show only one hypothesis tested per fortnight. The binding constraint isn't delivery speed; it's learning velocity. We shift focus to faster experimentation loops. Learning velocity improves; the next constraint surfaces: PR review queues have ballooned. We introduce agent-to-agent review for low-risk changes. The constraint moves again. That's the point: the journey doesn't end.

Everything as Code

Principle: If it's important, it's becomes versioned, testable and machine-readable; it becomes code. Once it's code we can apply our existing tools to share unambigously, test and validate, measure and

analyse. The frontier is expressing the business problem itself as code, because once the problem is code, agents can write solutions at run-time.

Rationale: Code is the only reliable source of truth. It reflects what actually runs; it benefits from peer review, automated testing, version control and rollback. Artefacts in Notion or PowerPoint inevitably drift. We've already codified infrastructure, dashboards, pipelines, security policies and API contracts. The next leap is codifying the business problem (intent, constraints, acceptance criteria) in machine-readable form. This matters because when agents can read a precise, current problem definition, they can generate, test and deploy a solution at run-time. The running product reflects today's understanding of the problem, not assumptions embodied in code written six months ago. Features become irrelevant, understanding workflows becomes paramount. The problem becomes the specification becomes the product, and stays fresh.

Practical Implications: Continue turning operational artefacts into versioned code. Prioritise codifying business problems: structured requirements, context configs and acceptance criteria that agents can consume directly. Every artefact gets PR review, automated validation and rollback. Quality becomes automatic, not manual.

Example: A product owner writes a new pricing rule as a structured EARS requirement with Gherkin acceptance criteria. On commit, agents generate the implementation, run the acceptance tests and open a PR, all before a developer touches the code.

Humans Above The Loop

Principle: Humans shift from in the loop to above the loop. We don't hand off responsibility; we govern at a higher level of abstraction.

Rationale: When humans write code, trust comes from reviewing every line and reasoning about every change. That doesn't scale when agents generate code at volume. The shift isn't to delegate responsibility to automated systems; it's to operate at a higher level. First, verify the verification systems themselves: are the tests comprehensive, are the acceptance criteria correct, do the guard rails catch what they should? Then, verify the output of those systems: are test results consistent, are coverage trends healthy, are edge cases being caught? The same principle applies beyond testing. Risk-based oversight means humans are involved based on risk, not by default. Low-risk work proceeds autonomously; high-risk work gets layered verification with human final approval. The human role moves from doing to governing: designing the tests, defining the acceptance criteria, classifying risk, monitoring outcomes. Humans stay fully accountable; what changes is the level at which we verify.

Practical Implications: Design verification systems (tests, guard rails, acceptance criteria) rather than reviewing every output. Classify work by risk: ADLC phase, code area, impact radius. Low-risk work proceeds with agent-only review; high-risk work gets layered verification with human approval. Monitor the health of verification systems as a first-class concern. Test the tests.

Example: A team used to review every PR manually. After shifting above the loop, they design comprehensive test suites, define acceptance criteria per feature and set guard rails for security and compliance. Agents review agents for routine changes. Humans review aggregated dashboards: pass rates, defect escape rates, coverage trends. When the dashboard flags a drop in acceptance-test pass rates, they investigate the verification system, not individual PRs. They find the acceptance criteria for a new module were underspecified, fix them, and quality recovers. The human intervention was at the level of the system, not the code.

The Vision: What We Build

Good does not look like headcount reduction to prove an opex point: fewer developers doing the same amount of work but taking the same time over it. Good looks like completing what you set out to work on more quickly because agents handle designing, coding and build tasks because humans have crafted great context that make these tasks repeatable at high quality. Good looks like closing the inner loop from problem to solved; prototypes becoming production code in days, not months, value delivery in weeks, not quarters.

This vision rests on three shifts in perspective: vibe coding and rigorous engineering quality should both have a place in the workflow, ways of working should be redesigned around how agents operate and we should optimise for confidence amid ongoing change rather than pretend the journey has a final fixed endpoint.

Shift 1: Sequence Speed and Rigour

Today, teams still face a false choice: move fast with vibe coding or slow down for engineering quality. That is the wrong framing. The four-phase workflow replaces the trade-off with sequence: prototype to clarify intent, specify to capture what you've learnt, build to engineer it properly, deploy to ship it. Each phase has a different standard of rigour matched to its purpose. Prototypes need speed and iteration; vibe coding excels here. Production code needs reliability, compliance and debuggability; engineering practices deliver these. Agents can do both. Once that sequence is explicit, prototype work no longer sits outside production; it has a deliberate path into it.

This shift has two axes. The first is the target workflow itself: Prototype, Specify, Build, Deploy. The second is the maturity path organisations follow to become capable of running that workflow well; that operating journey is covered a later Journey technical paper.

The new DevX workflow transforms how we build software through four distinct phases: Prototype → Specify → Build → Deploy. Each phase has a clear purpose and transitions cleanly to the next.

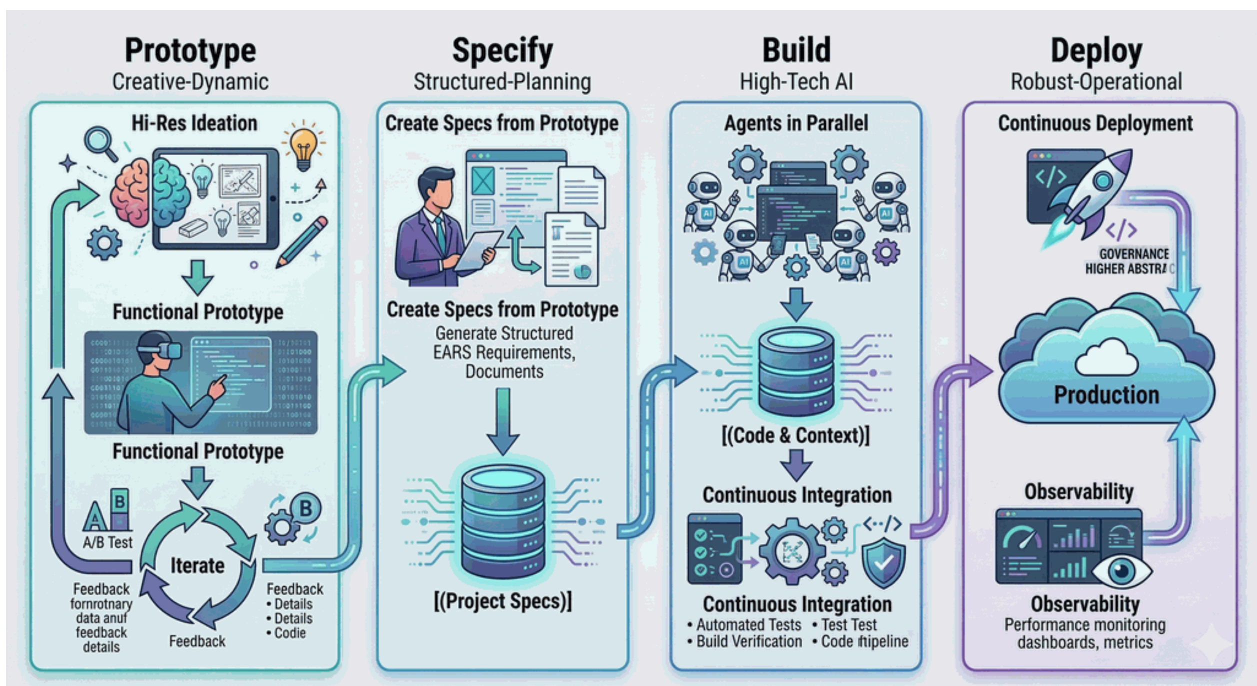

Figure 2: New DevX Workflow

Prototyping stays exploratory; vibe coding is the right approach there, not spec-driven formalism. Security changes and sensitive data handling require human approval regardless of phase. Infrastructure

changes often merit caution. The automation boundary is a design choice: match oversight to risk, and accept that some work stays human-heavy by design.

What This Means in Practice

This shift closes the loop from idea to production without forcing teams to choose between speed and quality. Prototypes stop being disposable side-hustles and become the fastest way to clarify intent. Specification turns what was learnt into reusable context, and build and deploy turn that context into reliable delivery. The result is faster movement with less rework, not faster chaos.

Shift 2: Work the Way Agents Work

The hardest shift isn't adopting new tools. It's resisting the instinct to force agents into human workflows and instead redesigning how we work around how agents work. Agents don't attend stand-ups, read between the lines, or ask clarifying questions mid-task. They need context not instructions, specifications not conversations, code not documents. Shoehorning agents into human workflows produces mediocre results. Redesigning workflows around agent capabilities unlocks their potential.

This shift covers three areas: the toolchain that supports agents, the context discipline that makes them effective and the ways of working that let humans and agents collaborate.

The Agentic Toolchain

The agentic toolchain connects local development environments with cloud-based agents, models and registries. This architecture enables developers to work with AI assistance whilst agents handle parallel build tasks in the cloud.

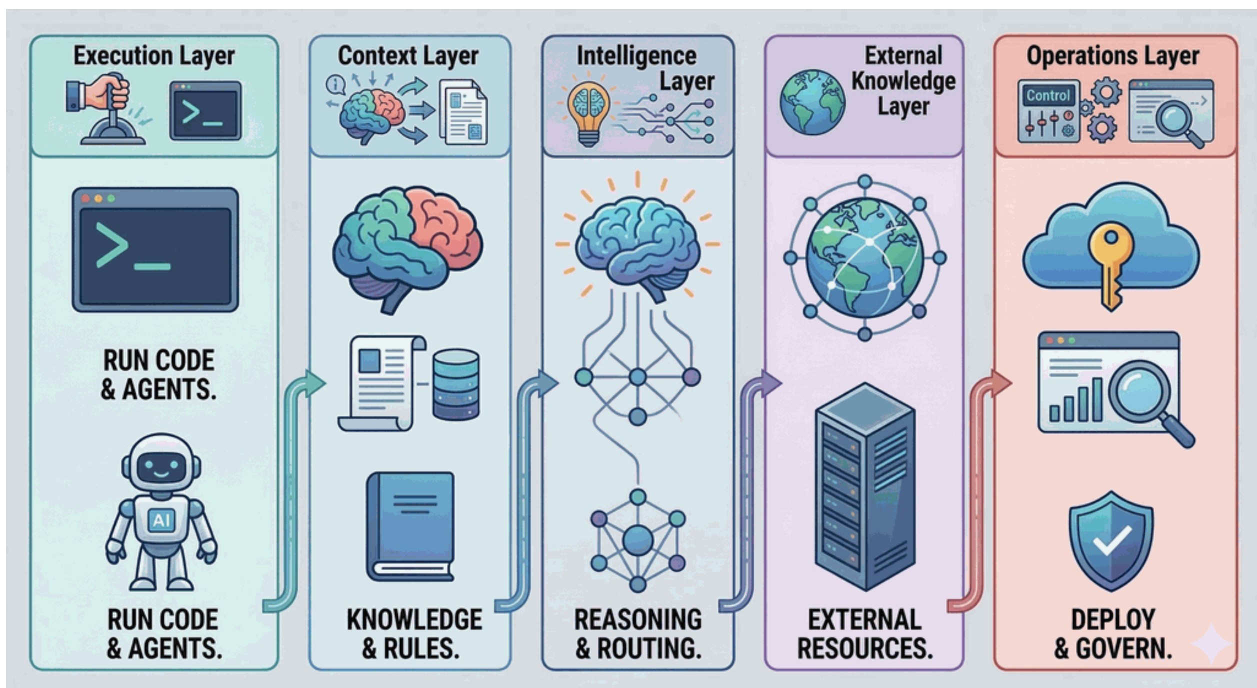

Figure 3: Agentic Toolchain

Execution Layer: Where work runs: the IDE or CLI where developers orchestrate; local or cloud containers for isolated environments; cloud agents and models that perform tasks; and repos (e.g. GitHub) that hold code and trigger pipelines. Developers and agents meet here.

Context Layer: What agents and developers know: model memories, task specifications, agent rules, project context and organisation standards. This layer holds the institutional knowledge and constraints that guide behaviour: specs, conventions and approved patterns. Context engineering is the discipline of curating and surfacing this context.

Intelligence Layer: How work is coordinated: workflows that define steps and sequencing; a context-sharing framework so agents and tools see the right information; intelligent model routing to select the right LLM for each task; and the MCP Proxy as the gateway for context and tools.

External Knowledge Layer: Third-party context and tools: context libraries and MCP servers that plug into Jira, Notion, docs, search, browser and other APIs. The MCP Proxy connects this layer to the intelligence and execution layers.

Operations Layer: Where delivery and governance live: CI pipelines, CD and cloud deployment, observability and compliance. Agents push to repos and trigger these pipelines; deployments become algorithmic and auditable.

Integration: The layers connect at defined interfaces: Execution consumes Context and Intelligence; Intelligence uses External Knowledge; Operations runs on outputs from Execution. Registries (specs, containers, plugins, models) version artefacts across the stack. Everything is code; everything flows through well-defined interfaces.

New Ways of Working

The shift to agentic development requires rethinking how teams operate. Roles evolve, workflows change and new collaboration patterns emerge.

Human-Agent Division of Labour

Humans focus on strategic work. Product managers define problems worth solving. Designers perfect user experiences. Engineers architect systems and perfect context (specs, rules, standards). QA engineers evolve from manual testers to guard rail designers and risk classifiers; routine testing moves to agents. The creative, high-judgement work stays human. The repetitive, well-defined work moves to agents.

Rethinking Agile Practices

The agile paradigm breaks. Traditional ceremonies designed for 2-week human coordination don't work for 2-hour agent cycles. Sprint planning shifts from estimating implementation to prioritising problems. Stand-ups disappear into continuous async updates. Reviews focus on outcomes, not code quality. Retrospectives move from sprint execution to platform improvements. Coordination ceremonies become decision forums. Synchronous time becomes precious, reserved for ambiguous decisions and strategic pivots. Optimising flow is paramount.

Risk-Based Oversight

Agent In The Loop replaces Human In The Loop. The default shifts from humans reviewing all agent work to agents reviewing other agents' work, with humans involved based on risk. Risk varies across three dimensions: ADLC phase (Prototype allows more autonomy; Deploy demands stricter oversight), code area (high-risk systems as defined in EU AI Act require closer review) and impact radius (security changes, sensitive data handling, critical systems trigger agent escalation to humans). Low-risk work proceeds autonomously. High-risk work gets layered agentic verification but a human has final approval.

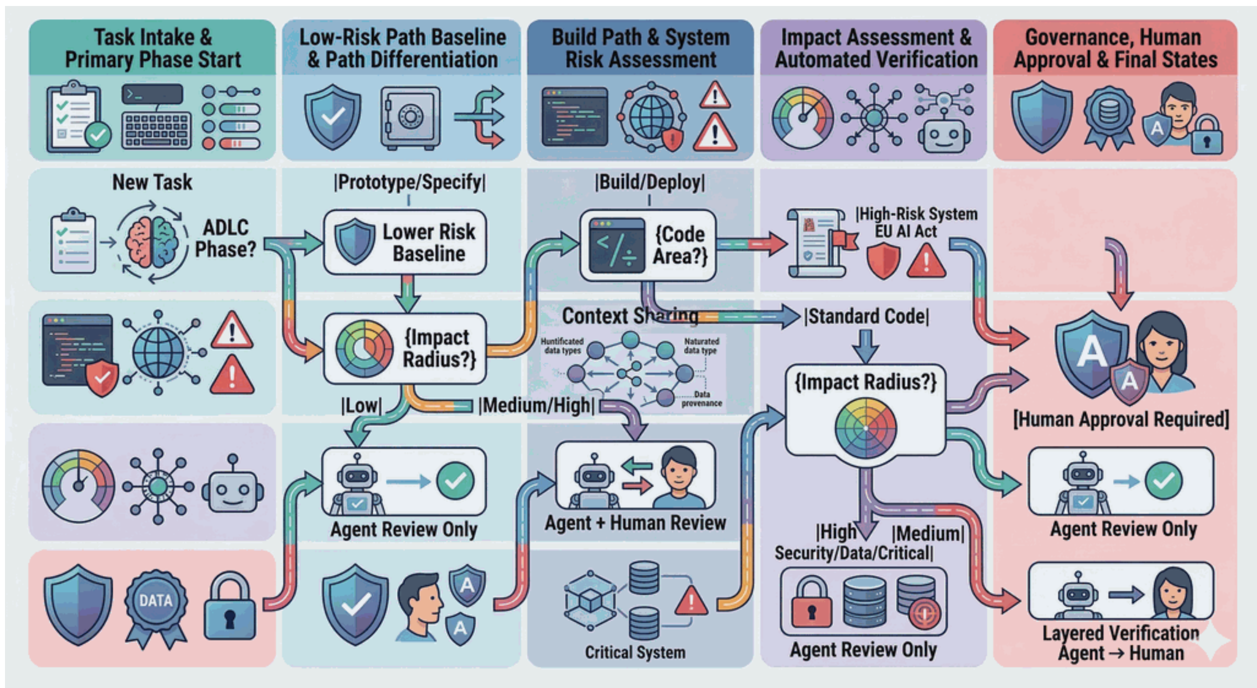

Figure 4: Risk-Based Oversight Model

Context Engineering as Primary Discipline

Curated context becomes the primary work product. Specifications are one form of that context; so are rules, standards and task intent. Writing and shaping intent through context matters even more when writing for autonomous agents, so teams must invest time perfecting it. Context reviews replace some code reviews. The ability to shape intent through context becomes a core skill.

Trust Through Measurement

Trust becomes the key metric. When agents write code, trust shifts from "who wrote it" to "does it pass comprehensive tests". Trust becomes measurable: test coverage, pass rates, defect escape rates, security scan results, rollback frequency. Teams write tests first, then let agents implement. Automated verification runs continuously. Code merges when tests pass, regardless of author. Guard rails make trust operational across three timeframes: dev-time (run multiple agents on same task, select best output), build-time (security policies, spending limits, specs enforcement, testing requirements) and run-time (validate each non-deterministic output for content safety, format, hallucination detection). Test to trust. Then trust the code.

Operating with Agent Failure Modes

Agents fail in new ways. Agentic systems introduce failure modes that differ from traditional bugs: premature task abandonment, context flooding, repeating failed actions, poor navigation and incorrect verification. These are metacognitive; agents make poor strategic decisions about how to work, not just what to write. The end state includes mitigations: task decomposition so agents work in reliable chunks, context reset when usage exceeds thresholds, verification checklists in specifications and reviewer agents that validate completeness before submission. Good looks like systems that understand these failure modes and are able to mitigate them.

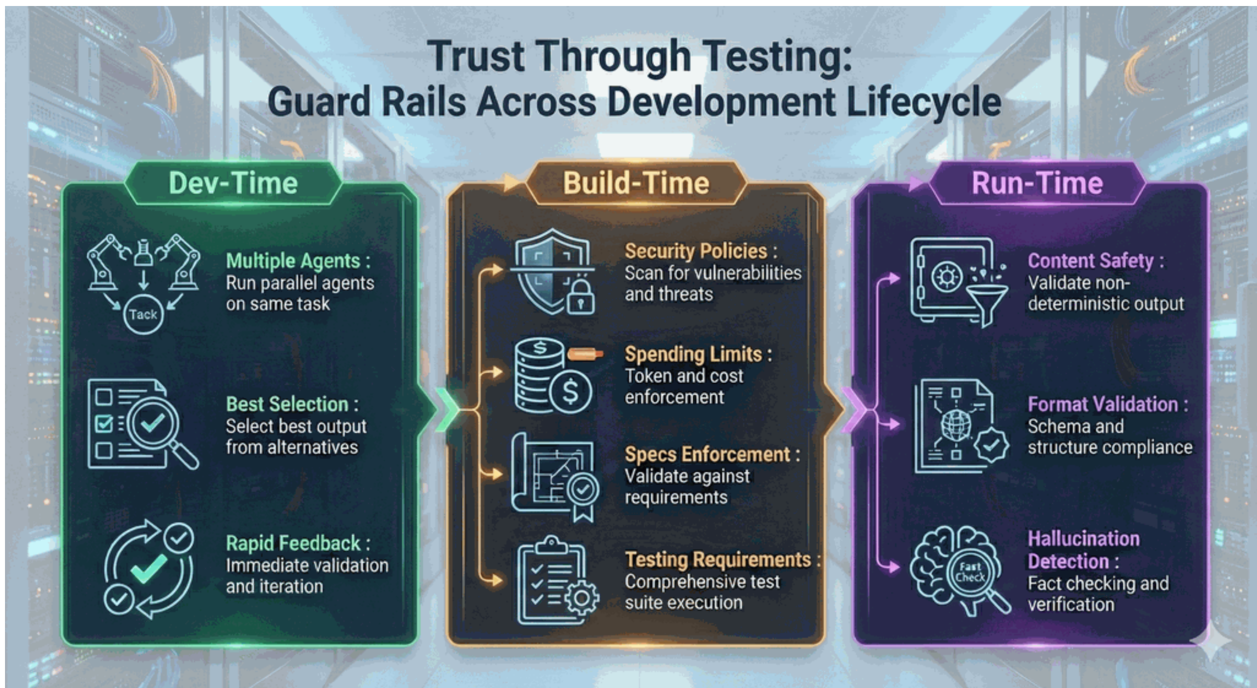

Figure 5: Guard Rails Timeline

Three timeframes: **Dev-time** (before commits): run multiple agents on the same task, select best output via testing and static analysis, immediate feedback, catch obvious errors early. **Build-time** (CI/CD): security scanning, spending and token limits, specification compliance, test suite execution, code quality and coverage, dependency scanning. **Run-time** (production): content safety and format validation for non-deterministic output, hallucination detection, performance monitoring, automated rollback on degradation. Guard rails are not about preventing agents from working; they are about letting agents work safely at speed so humans can focus on intent.

Agent-to-agent verification: Multi-agent systems with cross-verification catch deeper inconsistencies (pattern mirrors peer review). Typical flow: primary agent generates solution; review agent checks quality, style, best practices; security agent scans; test agent validates against specs and edge cases; integration agent verifies compatibility; human reviews aggregated feedback and approves intent.

Task Decomposition Strategy

Work decomposes differently. Start with architectural decomposition: break repositories into smallest possible components that can be worked on independently, decompose UIs into components that compose independently. Map tasks and their context to these components. Then size tasks to fit autonomy horizons: one-hour tasks execute reliably. Build safeguards into parts that resist decomposition: build pipelines, UI-to-backend queries, database calls. These integration points become bottlenecks if not protected. The decomposition hierarchy flows: architecture → components → tasks → subtasks within autonomy limits.

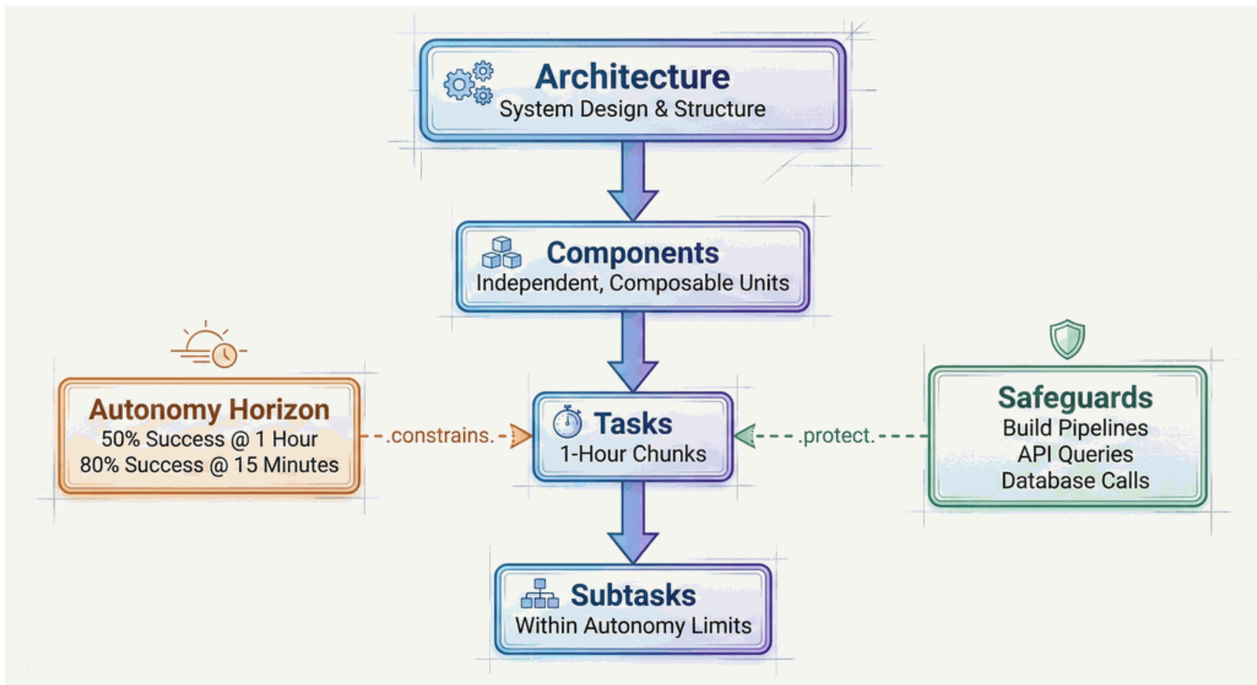

Figure 6: Task Decomposition Hierarchy

Accelerated Feedback

Feedback loops tighten. Prototypes appear in hours. Intent clarifies in hours. Implementations complete in days. Deployments happen in minutes. The entire cycle compresses. Learning accelerates. Teams can quickly see the impact of their work upon users. Product management becomes hypothesis-based testing.

Context Engineering and Agent Ops

Agent effectiveness depends on context quality and operational excellence. Context engineering manages the hierarchy that shapes agent behaviour: third-party best practices flow into organisational standards, which combine with project context and developer preferences to form task-specific context. Each layer adds constraints and capabilities. The art is delivering the right context at the right time. Too much context overwhelms the agent. Too little produces poor results regardless of model capability.

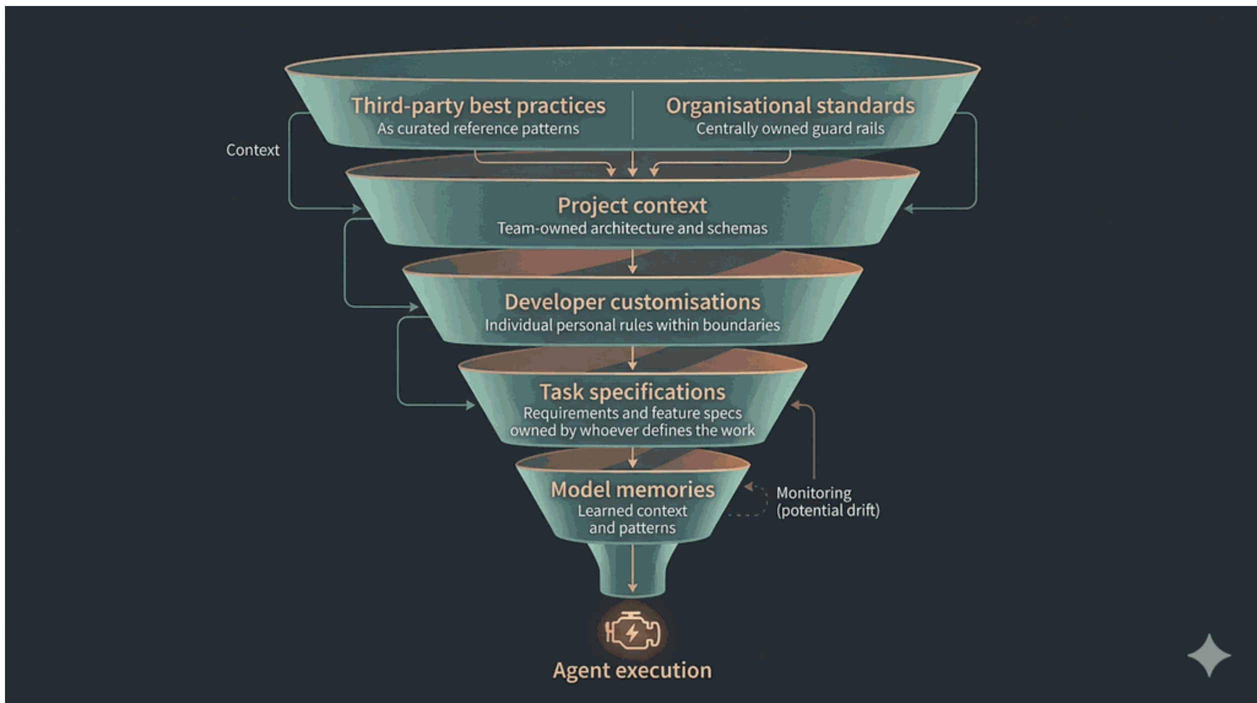

Figure 7: Context Hierarchy

The six-layer context hierarchy: (1) **Third-party best practices**: framework conventions, language idioms, industry standards; sourced from documentation and community knowledge. (2) **Organisational standards**: guard rails, approved tools, code style guides, compliance (e.g. GDPR, SOC2); maintained centrally, versioned in Git. (3) **Project context**: architecture docs, database schemas, API contracts, existing codebase, integration patterns; lives in the project repository. (4) **Developer customisations**: personal preferences, preferred testing frameworks, custom MCP servers; stored in developer profiles. (5) **Task specifications**: immediate requirements, acceptance criteria, related issues, user stories; defined per task or ticket. (6) **Model memories**: learned patterns from previous interactions, project-specific conventions, error patterns and solutions; continuously updated by the model.

Manage each layer deliberately. Curate documentation sources and keep them current. Codify guard rails and compliance as versioned configs. Treat the codebase itself as living context. Let developers extend with personal profiles. Invest precision in task specifications, the most volatile layer, with the highest marginal return on agent output quality. Audit model memories for drift.

Agent Ops completes the picture: define agents as code (persona, temperature, tool access, context hierarchy), version everything in Git and track agent performance (success rates, drift, context collapse) so you know which layer is degrading.

Example: A team notices agent output quality dropping on a backend service. Their AgentOps dashboard shows first-attempt success rate fell from 92% to 74% over two weeks. They trace it to layer 3: a major schema migration updated the database but the architecture docs still described the old model. Agents were generating code against stale project context. They update the docs, success rate recovers within a day. The fix wasn't retuning the agent or changing the prompt; it was refreshing the right layer of context.

What This Means in Practice

This shift turns AI adoption from a tool rollout into an operating model redesign. Teams stop relying on heroics, tacit knowledge and line-by-line rescue, and start building environments in which agents can succeed repeatedly. Context becomes infrastructure, guard rails become code and human judgement moves up the stack. The result is not fewer humans, but better leverage from the humans you already have.

Shift 3: Navigate, Don't Arrive

The destination is uncertain. The technology evolves faster than we can plan. The endpoint we imagine today may not be the endpoint we want to reach. The shift is to stop treating transformation as a programme with a tidy finish line and start treating it as a capability for adapting well under continuous change. Metrics still matter, but only as instruments for navigation. Three questions matter, mirroring the three capabilities from "Elevate the Constraint": Are we focusing on the right things (prioritisation)? Are we learning fast enough (discovery)? Are we converting insight into outcomes (execution)?

Lag indicators tell you what happened. Lead indicators tell you what's coming. Measure both or get surprised. Most organisations obsess over productivity metrics: cycle time, throughput, quality. These are lag indicators: they tell you whether you were productive. But they can't tell you whether you'll stay productive. For that, you need lead indicators. A team can hit every productivity target this quarter and still be six months from a talent exodus. The lag metrics won't warn you; the lead metrics will.

Figure 8: Success Metrics

Productivity

Are we productive now? These metrics tell you what happened, after the fact.

Focus: Are we working on the right things?

- **Feature utilisation rate**: Percentage of shipped features actually used by customers (signals we're building what matters)
- **Value delivery ratio**: Proportion of capacity spent on new value vs support and maintenance (signals we're not drowning in toil)

Speed: Are we shortening the learning loop?

- **Cycle time**: Time from work started to work deployed (signals constraints are being eliminated)
- **Time to merge PRs**: How long PRs wait for review and approval (signals review isn't a bottleneck)
- **Merges per developer per week**: Throughput normalised by team size (signals sustainable acceleration)

Predictability: Are we building what we think we're building, and deploying when we said we would?

- **Agentic task success rate**: Tasks completed successfully end-to-end without human rescue (signals effective decomposition and intent clarity)
- **Sprint capacity accuracy**: Actual vs planned capacity delivered (signals reliable estimation)
- **Velocity stability**: Consistency of delivery across sprints (signals predictable output)

Quality: Are we building it right?

- **Comprehensive test coverage**: Automated verification across functional correctness, intent alignment, policy compliance and data safety (signals testing extends beyond code to behaviour and safety)
- **Refactoring rate**: Percentage of changes that improve existing code vs add new features (signals we optimise freely when generation is cheap)
- **Bug resolution ratio**: Bugs resolved vs bugs created (signals quality debt isn't accumulating)

Durable Productivity

Will we stay productive? These metrics predict what's coming, before it arrives.

Discovery: If building code is cheap, and the key skill is deciding what to build, how good are we at making that decision?

- **Hypothesis velocity**: Hypotheses tested per week (signals we're learning fast and making evidence-based decisions)
- **Exploration breadth**: Alternative approaches prototyped before committing (signals we're avoiding premature convergence)
- **Learning efficiency**: Insights gained per experiment (signals we're designing sharp experiments)
- **Feature utilisation rate**: Percentage of shipped features actually used by users (signals better intent alignment)

Trust: When code is unlimited, do people trust it? Trust predicts whether productivity gains will stick.

- **Autonomy horizon**: Median duration of agent sessions that complete without human intervention (signals agents handling longer/more complex work)
- **Rework rate**: Percentage of tasks requiring multiple agent commits before completion (signals first-time success improving)
- **Context integrity**: Correlation between context file changes (rules, agent definitions, standards) and preceding agent commits (signals agents working well within boundaries)

Innovation: Are we innovating, and is that innovation effective?

- **Constraint cycle time**: Time from constraint identified to constraint eliminated (signals we're getting faster at removing bottlenecks)
- **Innovation effectiveness**: Percentage of improvement initiatives that actually eliminate their targeted constraint (signals innovation is hitting constraints not missing)
- **Knowledge effectiveness**: Rate at which access to knowledge artefacts decays over time (signals knowledge is being refreshed without becoming unstable or neglected)
    - Fast decay: new knowledge is superseding old; signals active innovation
    - Flat access: stable, useful knowledge; signals mature domain
    - Rapid decay with no replacement: signals neglect

Warning Signs

Watch for the gap between lag and lead. Productivity metrics can look healthy while durability metrics quietly deteriorate.

Productivity warnings (lag looks good but something's off):

- Individual productivity increasing but organisational throughput flat. Bottlenecks are migrating, not disappearing.
- PR backlogs growing despite faster code generation. Review has become the constraint.

- Cycle time improving but feature utilisation dropping. We're shipping faster but building the wrong things.

Durable productivity warnings (lead indicators deteriorating):

- Same constraints keep reappearing. We're patching, not eliminating.
- Context files churning weekly. Rules and agent definitions aren't assuring engineering-grade code.
- Exploration spikes per quarter have decreased. Learning for tomorrow is being sacrificed for doing today.

These patterns mean you're burning through lead indicators to hit lag targets. That's a short-term spike, not a sustainable trend. Rebalance before the lag metrics collapse.

What This Means in Practice

The promise of AI-augmented development is not doing the same work with fewer people. It is removing code generation as the bottleneck so teams can focus on solving the right problems, learning faster and shipping with more confidence. Lag metrics show whether delivery is accelerating now; lead metrics show whether that acceleration will last. Measure both, or you risk mistaking a short-term spike for durable productivity.

Conclusion: If Not Now, When?

We opened with "why now?", an Engel's Pause where transformative capability exists but its economic impact hasn't materialised. Three principles set the frame: elevate the constraint, express everything as code, move humans above the loop. Three shifts in perspective described how to respond: treat vibe coding and rigorous engineering quality as complementary phases of the same flow, redesign ways of working around agents and get comfortable operating in continuous change rather than chasing a finished end state. Now let's test the logic with a counterfactual: what happens if we wait?

The Cost of Waiting

The autonomy horizon doubles every four to seven months. Waiting a year doesn't mean starting a year behind. It means starting against a capability frontier that has moved two to four doublings ahead, with competitors who've been learning by doing the entire time. The Engel's Pause closes. Early movers have already reorganised their workflows, built their context discipline and trained their teams. Late adopters don't get to skip that learning; they do it under competitive pressure instead of ahead of it.

Late adopters can still be brilliant at continuously eliminating the next constraint, but they're optimising the wrong flow. Organisations that delay will keep perfecting human-in-the-loop processes: faster code review by people, better sprint planning for people, tighter hand-offs between people. The model itself is the constraint. Every improvement that assumes humans are in the loop for everything calcifies that assumption further, making the eventual shift to humans-above-the-loop harder. The gap widens, not because the technology isn't ready, but because the organisation has invested in the wrong architecture of work.

When This Vision Under-delivers

Honesty matters: this doesn't work everywhere. It under-delivers where trust is low and teams won't let agents operate with appropriate autonomy. It stalls where context discipline is weak; agents are only as good as the context they receive, which is often a reflection of the person delivering it. This vision also disappoints where organisations expect tool rollout alone to move metrics, without thinking about adoption, constraints, engineering impact and return on investment. The transformation is organisational, not just technological.

If Not Now, When?

The capability is here. The competitive window is open. The question isn't whether agentic development will reshape software engineering; that's already settled. The question is whether we learn by doing now, while the Engel's Pause gives us room to experiment, or later, when the pause has closed and the price of learning has gone up.

Appendices

Glossary of Terms

Agent: An AI system that can autonomously perform tasks, make decisions and interact with tools and APIs to achieve specified goals.

ADLC: Agentic Software Development Lifecycle: the practice of using AI agents throughout the software development process.

BDD: Behaviour-Driven Development: testing approach that validates features against user requirements and expected behaviours.

Context engineering: Designing, curating and versioning the hierarchy of context (specs, rules, standards, task intent) that shapes agent behaviour.

Context registry: Registry of versioned context, including specifications and agent rules, that define system behaviour.

DevOps: Development and Operations practices combined: the Build phase in our four-phase workflow where agents create production code.

DevX: Developer Experience: the overall experience of developers using tools, processes and practices to build software.

EARS: Easy Approach to Requirements Syntax: a structured format for writing clear, testable requirements.

GitOps: Operations practices where Git serves as the single source of truth for declarative infrastructure and applications.

Hive Pattern: Multiple agents with specialised roles (planner, coder, reviewer, tester) working on different parts of a system simultaneously, coordinated by an orchestrator through shared context. Suited to work that decomposes into subtasks with dependencies. Improves quality through specialisation; incurs coordination overhead.

IDE: Integrated Development Environment: tools like Cursor and Windsurf that developers use to write code.

MCP: Model Context Protocol: a standard for providing AI models with access to organisational context (Jira, Notion, APIs, etc.).

Orchestrator: An agent that coordinates the work of other agents, managing dependencies and resolving conflicts.

Prototype: High-fidelity mockup with functional interactions: the first phase of our workflow where ideas become tangible.

Specification: One kind of context artefact: a formal description of what a system should do; the source of truth that drives agent behaviour and validates output.

Swarm Pattern: Multiple agents exploring different approaches to the same problem in parallel, with an orchestrator selecting or synthesising the best result. Suited to problems with multiple valid solutions where the optimal approach is unclear. Increases success probability; consumes more compute.

TDD: Test-Driven Development: practice of writing tests before code to ensure code matches specifications and context.

Vibe Coding: Fast, iterative, exploratory coding to quickly bring ideas to life: appropriate for prototypes, not production.

Reference Materials

Internal Documentation:

- The New Dev eXperience Workflow (source document for this vision)
- Implementation Plan for the New DevX Workflow
- DevX Day Setup Guide

Tools and Platforms:

- Cursor: AI-powered IDE
- Windsurf: AI-assisted development environment
- Claude: Anthropic's AI model for code generation
- Figma: Design and prototyping tool with MCP integration
- Devin: Autonomous AI software engineer
- Tessl: Specification registry platform

Infrastructure:

- GitHub: Code repository and version control
- Jira: Project management and ticket tracking
- Notion: Documentation and knowledge base
- Flux: GitOps deployment automation
- Grafana: Monitoring and observability

Further Reading

On context engineering and context for AI:

- Model Context Protocol documentation
- "Specifications as the New Source Code" (OpenAI presentation by Sean Grove): specs as one form of context
- OpenAI Model Spec (example of executable specifications)

On Agentic Development:

- Anthropic's Claude documentation on agentic workflows

On DevX and Developer Productivity:

- "The SPACE of Developer Productivity" (Microsoft Research)
- "Accelerate: The Science of Lean Software and DevOps" (Forsgren, Humble, Kim)
- State of DevOps Reports (DORA metrics and research)

On Software Engineering Practices:

- "Test Driven Development: By Example" (Kent Beck)
- "Behaviour-Driven Development" (Dan North)
- "GitOps: Operations by Pull Request" (Weaveworks)

On Specifications and Requirements (specifications as context artefacts):

- EARS notation guide (Easy Approach to Requirements Syntax)
- "Writing Effective Use Cases" (Alistair Cockburn)
- "Specification by Example" (Gojko Adzic)

On AI and LLMs:

- "Building LLM Applications for Production" (Chip Huyen)
- "Prompt Engineering Guide" (DAIR.AI)

Part 2

The Journey

The Journey — Contents

Introduction: The Transformation Journey	23
Implementing Shift 1: Sequence Speed and Rigour	25
Implementing Shift 2: Work the Way Agents Work	28
Implementing Shift 3: Navigate, Don't Arrive	34
Implementing Technology Change	35
Implementing Business Change	43
What Stays Human	48
Conclusion: The Redesigned System	50

Introduction: The Transformation Journey

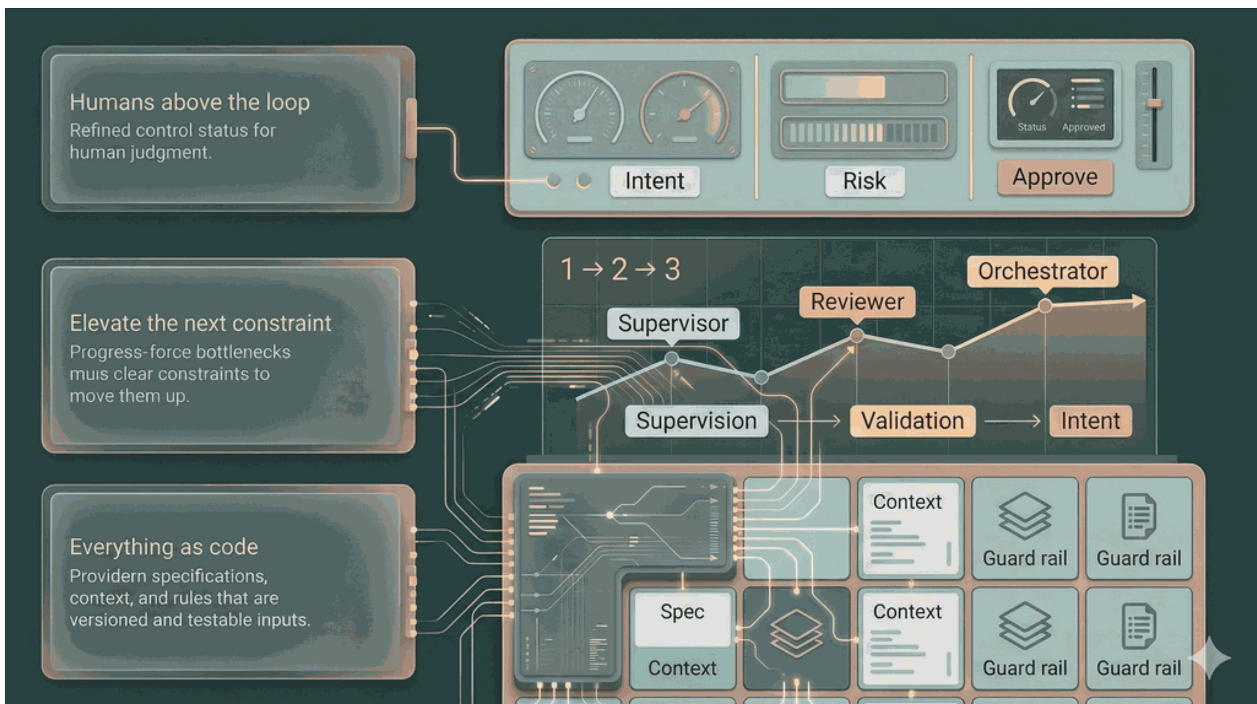

Figure 9: The Big Picture

Where Are We?

The promise of agentic software development is compelling. Faster delivery. Functional prototypes in hours. More ideas tested. More code written. A path to making technology less of a growth constraint on the business.

The reality for many teams is more nuanced. They've rolled out copilots, chat tools, maybe even autonomous agents, but the expected step-change in outcomes has not followed. More code may be appearing, yet throughput, learning speed and trust have not improved in proportion. This is the same pattern described in the Vision document through Engel's Pause: the capability is real, but the surrounding workflows, context discipline and governance have not yet been redesigned to exploit it.

That gap usually shows up through three current-state problems. **Humans are in the loop for everything, so the loop is slow.** Agents can help with thinking, designing, coding, testing and reviewing, but in most teams they still queue behind people. **Constraints migrate, so static improvement plans fail.** Accelerating code generation does not remove bottlenecks; it moves them. **Knowledge is scattered and stale.** Important context still lives across slide decks, wiki pages, tickets and tribal memory. Those artefacts drift.

But you can write lots of code really quickly. The question becomes: if you can write a limitless amount of code, what should you do with it?

Where Do We Want to Go?

The goal is not "fewer people doing the same work". The goal is closing the loop from problem to solved more quickly because agents handle designing, coding and routine execution whilst humans shape intent, govern risk and improve the system of work itself.

The Vision document defines three principles and three shifts in perspective. This Journey explains how to move towards them in practice.

- When our problem is that humans are in the loop, in the way, we apply the principle of **Humans Above The Loop** to design our first transformation: ways of working that move people from reviewing every line to designing verification, defining intent and intervening according to risk.
- When our problem is that our static improvement plans are failing, we apply the principle of **Elevate the Next Constraint** to design our second transformation: navigate the journey with metrics, warning signs and explicit constraint-management loops rather than assuming a fixed end state.
- When our problem is that knowledge is scattered and stale, we apply the principle of **Everything as Code** to design the third transformation: turning specifications, rules, context and operational artefacts into versioned inputs that agents and humans can trust.

The destination is an agentic development lifecycle where speed and rigour are sequenced rather than forced into a false choice, ways of working are designed around how agents actually operate, and teams navigate a moving frontier rather than optimise for yesterday's bottleneck.

Vision-Journey-Roadmap

The preceding Vision document sets out the case for change and what the target state will look like. This Journey describes how organisations should transform. Finally, the companion Roadmap describes how to plan and steer the implementation.

Implementing Shift 1: Sequence Speed and Rigour

Implementing Shift 1 means replacing the false choice between fast prototype work and rigorous engineering with a deliberate sequence that makes both modes useful. In practice, that means helping teams move from blurred, improvised hand-offs towards a workflow where Prototype, Specify, Build and Deploy are distinct, dependable and connected by better context and verification. The maturity model in this section is about how organisations typically grow into that operating rhythm.

This shift has two axes. The first is the target workflow as introduced in the previous Vision technical paper. The second is the maturity path organisations follow to make that workflow real in practice; that is the focus of this section.

New DevX Workflow

The new DevX workflow transforms how we build software through four distinct phases: Prototype → Specify → Build → Deploy. Each phase has a clear purpose and transitions cleanly to the next.

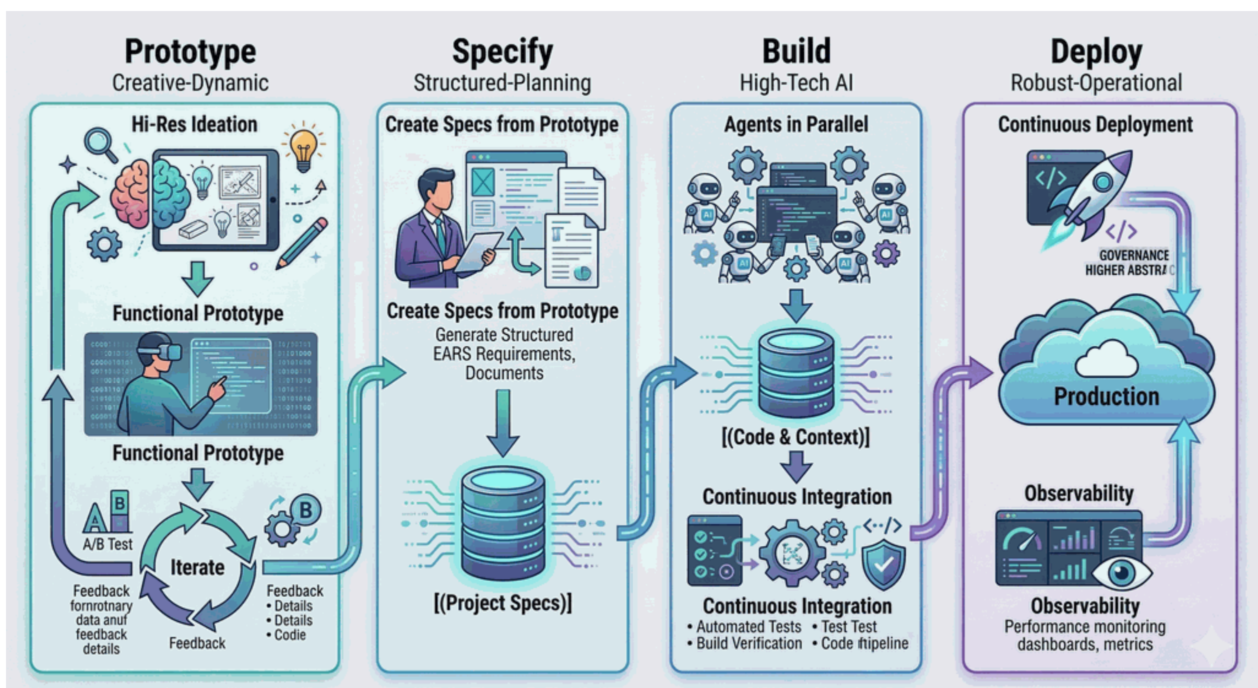

Figure 10: New DevX Workflow

Phase 1: Prototype: Vibe code high-fidelity ideas. Build functional prototypes using agent-augmented tools. Iterate based on stakeholder feedback until the approach feels right. The prototype perfects design intent at the expense of jankiness but don't worry, it won't reach production.

Phase 2: Specify: Break prototypes down into clear context. Prototype code will be messy and convoluted and a little bit broken because that's expected from rapid iteration. It can't be refactored into engineering quality. Instead, extract clear context that captures the perfected design: contracts, schemas, behaviours, constraints, performance targets. Layer institutional knowledge in the form of context. This is the source of truth that drives development, orchestrates agents and validates output.

Phase 3: Build: Agents generate production code from code + context. Run parallel agents using two distinct patterns: swarm (multiple approaches in parallel when the optimal solution is unclear; higher success probability, more compute) and hive (specialised roles (planner, coder, reviewer, tester) coordinating through shared context; better quality for complex features, coordination overhead). Choose the pattern to fit the problem.

Updates appear where the devs are, not where they're forced to be. Test everything continuously: code against context and specs, integration points, model performance, infrastructure compliance, solutions against engineering principles. Specialised Agents test and fix security, observability, UI issues as part of each commit or PR. Devs remain in-the-loop and responsible.

Everything becomes code and context, from application logic to agents and crucially, the problems themselves. When the problem can be expressed as code, the software solution can be written on demand.

Phase 4: Deploy: Automation, but not agents, executes deployment. Pipelines trigger agent tasks: update orchestration charts, rebuild containers, deploy models, refresh monitoring dashboards, migrate schemas, call external APIs. Deployment becomes algorithmic, reliable, fast, non-blocking.

Transitions: Each phase produces artefacts that feed the next. Prototypes become context. Context drives builds. Builds trigger deployments. Deployments are pipelined and non-blocking. The workflow is linear but iterative. Short inner feedback loops mean reverting is less common.

Prototyping stays exploratory; vibe coding is the right approach there, not spec-driven formalism. Security changes and sensitive data handling require human approval regardless of phase. Infrastructure changes often merit caution. The automation boundary is a design choice: match oversight to risk, and accept that some work stays human-heavy by design.

Adoption Path: Three Maturity Stages

Three maturity stages illustrate how teams typically move towards that workflow in practice. They describe the shift from experimenting with AI assistance inside existing habits to building an engineering system where agents can operate with meaningful autonomy.

Figure 11: Three Maturity Stages

Stage 1: DevX as Supervisor. Individuals adopt AI within existing workflows. Humans remain embedded in the loop; they prompt, review, edit and validate every output. This is the beginning of the journey, where the four-phase workflow is still weak, context is individual and therefore inconsistent, and productivity gains are constrained by supervision overhead.

Stage 2: DevX as Reviewer. Teams redesign their workflow so agents can succeed more often. Developers curate and share context, guard rails become codified, and agents verify other agents' work.

Human attention shifts upward from writing and line-by-line review towards intent, context quality and system health. The constraint moves from supervision to validation.

Stage 3: DevX as Orchestrator. Organisations operate a risk-based system where humans govern above the loop. Developers provide intent and context to an agentic system that plans, executes, reviews and tests through coordinated agents. The constraint moves again, from code validation towards problem clarity, intent quality and organisational ability to continuously improve. This is the last stage but not the end of the journey.

The Transformation Path

The stages are a progression in operating model, not a maturity scoreboard that every task must climb. Some work should stay early in the journey by design. Prototyping is the clearest example: the point is speed, exploration and better conversations, so Stage 1 is often the right place to stay. Other kinds of work, such as refactoring, bug fixes and well-specified feature delivery, can move further as context improves and verification becomes more trustworthy.

The limiting factor is not ambition; it is risk. Infrastructure, data-sensitive changes and security-critical work may benefit from better context, stronger guard rails and more orchestration, but they rarely justify removing humans from the approval path entirely. So the practical question is not "How do we get everything to Stage 3?" but "Which kinds of work should mature how far, and under what controls?" That is the thread running through the rest of this document.

This document now turns to the operating model changes that make that workflow real: how people work with agents, how context is shared, how guard rails are designed, and how progress is measured without mistaking movement for transformation.

The Timeline: Treat the stages as directional, not contractual. Most organisations will spend months building Stage 1 habits, longer establishing Stage 2 infrastructure and practices, and the longest learning how far Stage 3 autonomy should really go. Each stage delivers value independently; the point is learning by doing, not racing to declare arrival.

- **Stage 1** typically takes 3-6 months of practice before teams are ready to rely on stronger shared context and reduced supervision.
- **Stage 2** often takes 6-12 months because this is where the infrastructure, guard rails and operating model changes start to compound.
- **Stage 3** tends to be the longest horizon, often 12-18 months or more, because it depends on mature context, trust and organisational readiness.

Implementing Shift 2: Work the Way Agents Work

Implementing Shift 2 means redesigning the operating model around how agents actually succeed. That is less about buying tools and more about changing how teams share context, design guard rails, review work and run delivery. The sections that follow describe that shift in practice, from early supervised use through to shared context, agent-to-agent verification and risk-based orchestration.

In a human-led operating model, the classic constraint triangle is time, cost and quality: you can optimise for two at the expense of the third. In an agentic operating model, those constraints are restated: the operating triangle becomes **execution time** (how long the agentic system takes to complete a unit of work), **token cost** (how much inference the work consumes), and **autonomy horizon** (how long the system can continue producing work of acceptable quality before human intervention is required).

Shifting the way we look at quality is important: it is embodied in the autonomy horizon, because the horizon ends when the work can no longer be trusted to remain good enough without human review, correction or escalation. Human-in-the-loop steps therefore act as constriction points that shape delivery flow. The practical question becomes where those constriction points sit and whether they are moving. The three stages that follow can be read through that lens: Stage 1 is short autonomy horizon and high human friction; Stage 2 extends the horizon and makes token cost visible; Stage 3 optimises all three through orchestration, governance and continuous evaluation.

Stage 1: DevX as Supervisor

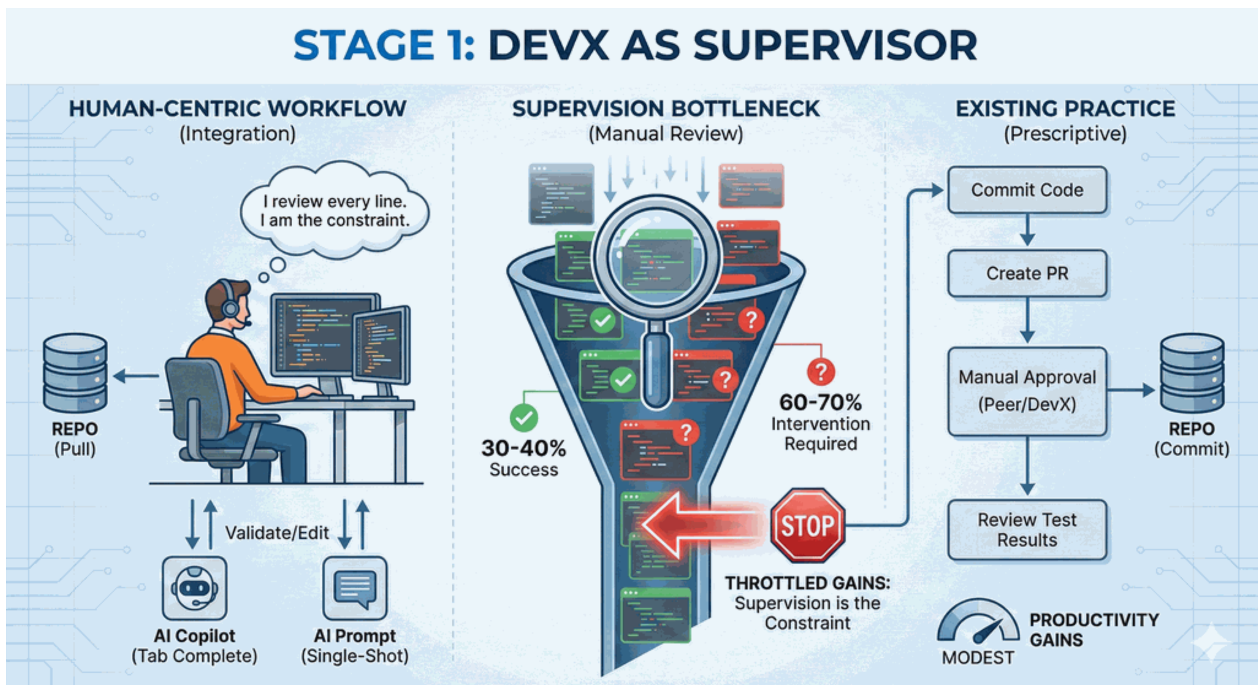

Figure 12: Stage 1: DevX as Supervisor

Characteristics and Practices

Starting at the beginning, stage 1 represents the current reality for organisations starting to adopt AI-assisted development. The developer remains an individual contributor who integrates AI into their existing workflow. The overall pattern is still highly supervised and prescriptive: humans remain responsible for orchestrating the tools and validating the output.

Key Characteristics:

- **Supervision-Heavy**: Every agent contribution is carefully reviewed and tested before acceptance

- **Single-Shot Interactions**: Developers use AI for tab completion and prompt-based code generation, receiving immediate responses they must validate
- **Existing Workflow Integration**: AI tools fit into current development practices rather than transforming them
- **Manual Code Review**: All code, whether human or agent-generated, goes through the same review process
- **Modest Productivity Gains**: Increased code creation volume is throttled by supervision overhead

Current Reality: Frontier models autonomously complete 30-40% of complex tasks without human intervention, meaning 60-70% of agentic tasks are still unreliable. This level of reliability necessitates high supervision. Human intervention ranges from checking and correcting draft output to catching logical errors, triggering rework, or abandoning the attempt entirely. The constraint is not only model capability; it is also the fragility of the systems and workflows wrapped around the model.

In practice, Stage 1 still looks like a human-led workflow with AI inserted into it. The developer prompts, reviews, edits, tests and submits; the agent helps within the loop, but does not meaningfully reshape the loop itself.

Bottleneck: Manual review of every agent output when success rates are 30-50%. Supervision becomes the constraint on productivity gains.

Early Practices

At this stage the specific product matters less than the habits around it. Teams get value by starting with low-risk work, keeping requests scoped, validating outputs immediately and learning from failure rather than treating each interaction as a one-off. Teams should expect to pay an innovation tax here: some effort goes into experimentation, context shaping and dead ends, because those are the costs of learning a new way of working.

- Start where the value is measurably clear, so the innovation is pulled into the business not pushed upon it.
- Keep tasks small enough that humans can still review outcomes confidently.
- Treat prompting, validation and context-setting as learnable skills.
- Capture failure patterns so the team improves collectively, not just individually.

Key Takeaway: Stage 1 is about learning to work with AI, not replacing your workflow. Master the fundamentals (effective prompting, validation practices, understanding failure modes) before attempting workflow transformation. The supervision overhead you experience now is the price of building the expertise you'll need later.

Stage 2: DevX as Reviewer

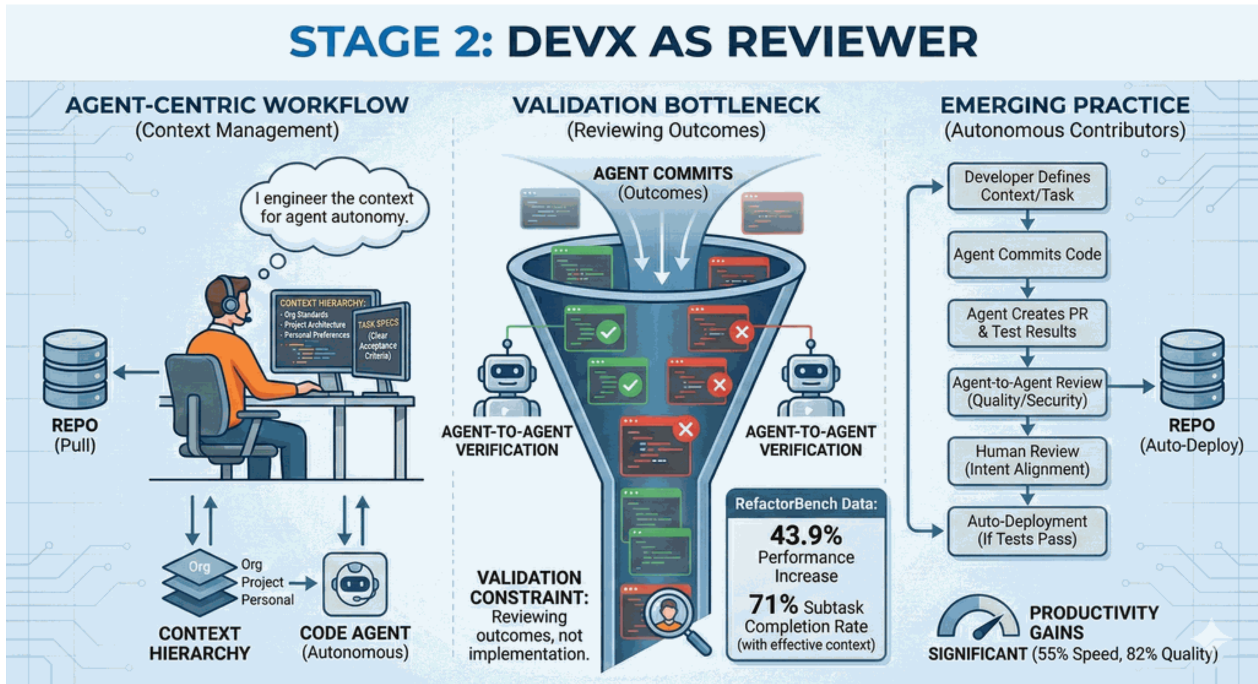

Figure 13: Stage 2: DevX as Reviewer

Characteristics and Practices

The shift in Stage 2 is from supervising every line of code to engineering the environment in which agents succeed. Stage 2 represents emerging best practice (2025-2026), where developers transform their workflow to help agents deliver code autonomously. The bottleneck moves from supervision to validation.

Key Characteristics:

- **Context Management Focus**: Developers invest time building persistent context that agents can access across interactions
- **Agent-to-Agent Verification**: Agents test other agents' work; humans review outcomes rather than implementations
- **Workflow Transformation**: Development practices adapt to agent capabilities rather than forcing agents into human workflows
- **Guard Rails Enforcement**: Organisational standards, security policies, and quality gates operate automatically
- **Meaningful Productivity Gains**: When context management is done well, teams spend less time rescuing agents from avoidable misunderstandings and more time moving work forward with confidence.

Why this matters: Research on state-aware code editing and retrieval-augmented code generation points in the same direction: agents perform better when they have relevant context and a clearer representation of what has already happened. The point is to give them the right information at the right time.

In practice, Stage 2 shifts the developer from directly producing every change towards defining intent within shared context and reviewing outcomes rather than hand-crafting every implementation step. The exact workflow mechanics vary, but the direction is consistent: less line-by-line human supervision and more reliance on shared context and automated verification.

Bottleneck: Validation of code becomes the constraint. Developers spend less time writing and more time verifying that agent output meets requirements, follows standards, and integrates correctly.

Context Engineering

Context engineering is the art of delivering the right context at the right time. Too much context overwhelms the agent. Too little produces poor results regardless of model capability. The Stage 2 shift is that context stops being improvised and individual, and becomes layered, shared and governable.

In practice, that usually means moving from one-off prompts towards a context hierarchy that combines external best practice, organisational standards, project knowledge, task intent and controlled access to live systems.

Guard Rails and Validation

The Stage 2 shift is that more of what used to live in reviewer judgement becomes explicit and testable through guard rails and agent-to-agent verification. Human review should narrow towards intent, risk and architectural judgement rather than re-checking every routine decision.

Signals of Progress:

- **First-Attempt Success Rate**: the large majority of agent PRs pass all automated checks without rework
- **Policy Violation Rate**: policy violations become rare rather than routine
- **Security Scan Pass Rate**: security scans are a blocking gate and pass reliably
- **Test Coverage**: coverage is maintained or improves through agent-generated tests
- **Human Review Time**: review time falls materially from Stage 1 levels as routine checking moves to agents

Key Takeaway: Stage 2 is about building the infrastructure that makes agent autonomy possible. The task is not simply to use better tools, but to engineer an environment where agents can succeed independently. The context hierarchy, guard rails, and validation workflows built here become the foundation for Stage 3's orchestration capabilities.

Stage 3: DevX as Orchestrator

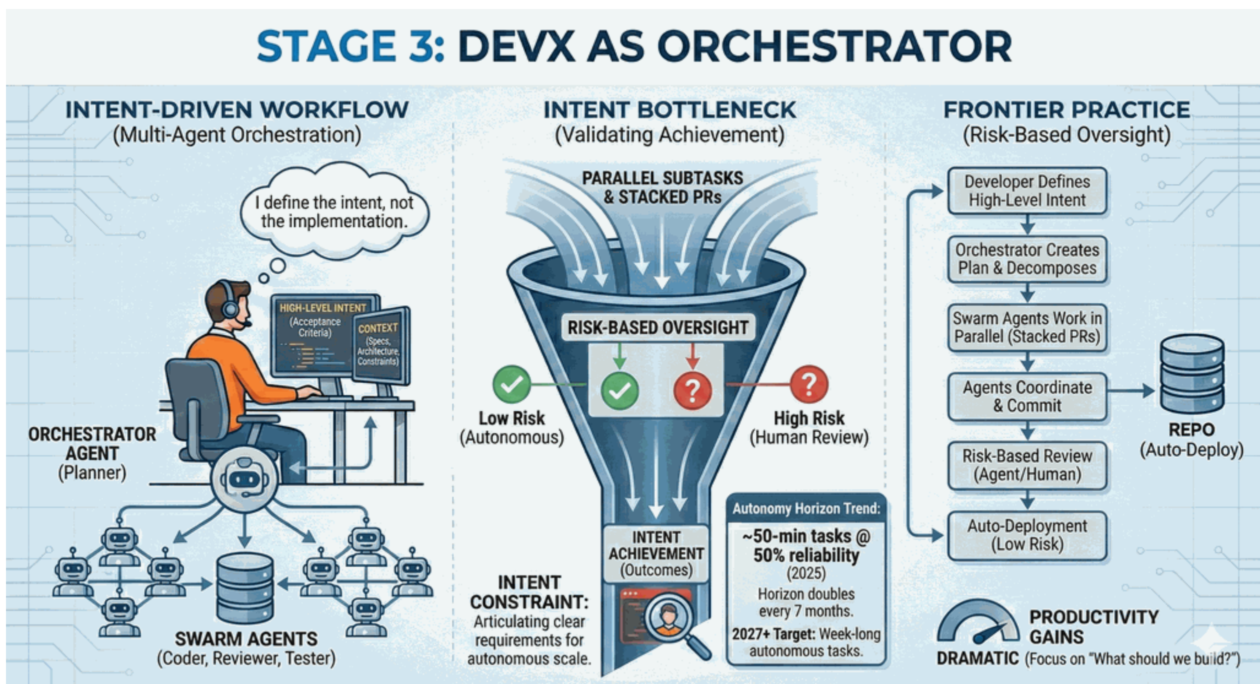

Figure 14: Stage 3: DevX as Orchestrator

Characteristics and Practices

Stage 2 infrastructure is operational. Stage 3 shifts the frontier again: the focus moves from validating code to validating intent. Stage 3 represents a forward-looking vision (2027+) in which developers orchestrate multiple agents working in parallel on complex tasks. The bottleneck moves from validation to intent specification.

Key Characteristics:

- **Intent-Driven Development**: Developers describe what they want to achieve, not how to implement it
- **Multi-Agent Orchestration**: Swarm and hive patterns coordinate multiple agents on complex, decomposed tasks
- **Risk-Based Oversight**: Human involvement scales with risk level; low-risk work proceeds fully autonomously
- **Agent-of-Agents Architecture**: Orchestrator agents coordinate specialist agents (planner, coder, reviewer, tester)
- **Dramatic Productivity Leap**: Focus shifts from "can we build it" to "what should we build"

What This Means: As of March 2025, agents complete ~50-minute tasks with 50% reliability and ~15-minute tasks at 80% reliability. The autonomy horizon doubles every 7 months. By 2027, agents may handle week-long tasks autonomously. This isn't speculation: it's trend extrapolation from today.

In practice, Stage 3 shifts the developer again: away from validating implementations and towards supplying intent, constraints and risk boundaries to an orchestrated system. The detailed workflow, coordination and oversight mechanisms belong in the implementation section; the defining change here is that the human becomes the governor of the system rather than its manual coordinator.

Bottleneck: Intent specification becomes the constraint. Developers must articulate requirements clearly enough for autonomous execution. Ambiguous intent produces unpredictable results at scale.

Swarm and Hive Patterns

Two orchestration patterns emerge for coordinating multiple agents. Choose based on task characteristics and coordination needs.

Swarm Pattern (Parallel Exploration):

Swarm agents work independently on the same problem, exploring different approaches simultaneously. An orchestrator selects the best solution or synthesises elements from multiple attempts. This pattern is useful when the search space is unclear and exploration matters more than coordination efficiency.

When to Use:

- Problem has multiple valid solutions
- Optimal approach is unclear upfront
- Exploration benefits from diversity
- Time matters more than compute cost

Hive Pattern (Coordinated Specialisation):

Hive agents have specialised roles and coordinate through shared context. A planner decomposes work, coders implement, reviewers check quality, testers validate. This pattern is useful when the problem decomposes cleanly and the value comes from coordination rather than parallel exploration.

When to Use:

- Problem decomposes into distinct subtasks
- Subtasks have dependencies
- Specialisation improves quality

- Coordination overhead is justified

The important point for the journey is not the exact mechanics but the change in operating model: developers stop acting as the only coordinator and instead choose orchestration patterns that fit the work. The enabling infrastructure for shared state, task routing, progress tracking and conflict handling sits in `Implementing Technology Change`.

Risk-Based Workflow

Not all work deserves the same oversight. Risk-based workflows scale human involvement with actual risk, enabling agents to move more freely on low-risk tasks whilst maintaining stronger controls on high-risk work.

The important shift in Stage 3 is that oversight becomes conditional rather than uniform. Low-risk work can move with much less human friction, whilst higher-risk work still pulls people into the loop.

Key Principle: Trust but verify. Low-risk work proceeds autonomously, but comprehensive monitoring detects anomalies. If agent success rates drop or policy violations increase, the system automatically escalates oversight level until issues resolve.

Key Takeaway: Stage 3 is about trusting the system that has been built. It draws on the investment in context infrastructure, guard rails, and validation workflows to achieve meaningful agent autonomy. The developer's job transforms from writing code to defining problems worth solving. This is not the end of software development; it is the evolution to a higher level of abstraction where humans focus on what machines cannot: creativity, judgement, and understanding what customers actually need.

Implementing Shift 3: Navigate, Don't Arrive

Implementing Shift 3 means treating transformation as something to navigate, not something to finish. The technology frontier keeps moving, the constraint keeps migrating, and apparent progress can be misleading, so teams need a way to tell whether rollout is actually changing outcomes.

Measuring Progress by Following the Constraint

In a human-led operating model, progress is often narrated through fixed-plan commitments, plan adherence and whether work was delivered when expected. In an agentic operating model, those measures become less useful. The more revealing questions are about flow: where work is queueing, where human review or approval is throttling throughput, which part of the system is acting as the constriction point, and whether the next constraint is being elevated or merely relocated.

The practical task is therefore to follow work through the system. How long does work wait for review? Where do agents hand back uncertain output? Where does rework accumulate? Which checks, approvals or coordination steps are building queues faster than they clear them? Progress is not just more output. It is shorter queues, smoother movement through governed paths, less avoidable rework, and clearer evidence that the current bottleneck has shifted.

The stage model already showed the main migration pattern: the constraint moves from supervision to validation to intent specification. That migration matters because each stage creates a different flow problem. Early on, queues form around human checking and correction. Later they form around validation, integration and policy checks. Later still they form around discovery, intent quality and risk classification. The operating challenge is not to defend one delivery cadence, but to keep identifying and elevating the next live constraint.

Traditional software measures still matter, but they should be read through this flow lens. Lead time, cycle time, rework, queue length, approval latency and deployment latency all say more than schedule conformance about whether the system is actually moving. In agentic delivery, they sit alongside execution time, token cost and autonomy horizon, because the goal is not merely to generate more activity but to let good work move further with less friction and with controls proportionate to risk.

Productivity and durability still need to be measured together. Productivity asks whether useful work is moving now; durability asks whether the operating model is becoming more dependable as autonomy rises. If queues are shrinking but comprehension, trust or context integrity are collapsing, the gain will not last. If controls are strong but work barely moves, the system has become safe but stagnant.

Measurement, however, is only half the story. Once the metrics reveal where the new constraints sit, organisations still have to change the surrounding system: technology, roles, incentives, operating norms, team design and learning patterns. That is why the journey ends not with dashboards but with implementation.

Implementing Technology Change

The three shifts describe what has to become true. This section turns to the practical question: how do we actually move towards them? Implementing technology change means turning the operating patterns described earlier into a dependable technical system. Shared context has to become accessible and governable. Important artefacts have to become versioned and testable. Verification has to move from informal reviewer judgement into codified checks and feedback loops. Orchestration has to sit on top of infrastructure that can coordinate work, track state and surface failures. The subsections below describe those capability areas.

Context Infrastructure

Agentic development depends on reliable context infrastructure: versioned specifications, organisational standards, project context, task-specific intent and integration layers that let agents reach external knowledge without losing control. This is where the context-engineering idea from the Vision becomes operational.

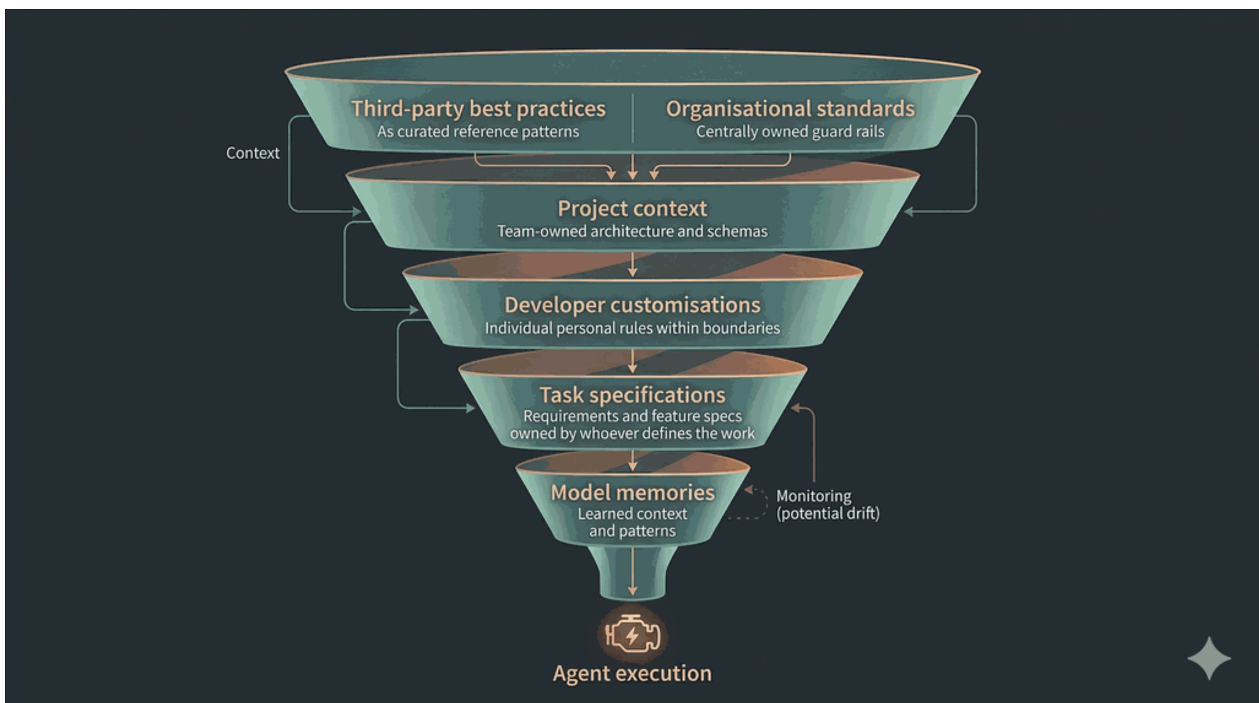

Figure 15: Context Hierarchy

The hierarchy is the same one introduced in the Vision: third-party best practices feed into organisational standards, which combine with project context, developer customisations, task specifications and model memories to shape execution. The layers matter because they separate what should be broadly shared from what should stay local, volatile or personal.

The implementation questions are: who owns each layer, how it matures across the stages, how it fails, and what it takes to keep it trustworthy?

Ownership by layer:

- **Third-party best practices**: curated from external documentation and reference material
- **Organisational standards**: owned centrally by engineering, platform, security or compliance
- **Project context**: owned by the team maintaining the service or codebase
- **Developer customisations**: owned by the individual developer within agreed boundaries
- **Task specifications**: owned by whoever defines the work and acceptance criteria
- **Model memories**: monitored rather than blindly trusted, because they can drift

How the hierarchy matures:

- **Stage 1**: context is mostly ad hoc, copied into prompts and held individually
- **Stage 2**: context becomes shared, layered and explicitly curated
- **Stage 3**: context becomes infrastructure for orchestration, so multiple agents can work from the same governed picture of the task

Common failure modes:

- project context is stale even though the code changed
- layers conflict with each other and the agent follows the wrong one
- too much context is supplied and the signal gets diluted
- task specifications are weak, so the most important layer is also the least precise
- model memories accumulate drift and outdated assumptions
- ownership is unclear, so no one maintains the layer that is failing

Operational signals:

- first-attempt success improves because agents already have the needed context
- developers repeat less setup and prompting from task to task
- review gets faster because intent, standards and project constraints are already present
- failures can be traced to a specific layer instead of feeling random
- teams can update one layer and see quality recover without changing the model

This is what makes context engineering valuable for Stage 3 as well as Stage 2. Multi-agent orchestration only works when agents are drawing from the same governed context rather than improvising from partial prompts and stale summaries.

Agent Comprehension:

As agent output scales, comprehension is not only a human problem. Agents themselves have to work across an ever-expanding body of context: project history, organisational rules, task intent, prior decisions, live state and their own intermediate traces. The challenge is not only making standards explicit; it is helping the agent hold the right context at the right time without being overwhelmed by volume, contradiction or stale material. As the context surface grows, the risk shifts from simple omission to context collapse: the agent may miss the governing signal, follow the wrong layer, or lose the thread of the task altogether. Better context therefore is not only about completeness; it is about making an expanding context base navigable, prioritised and usable by the agentic system.

Everything as Code

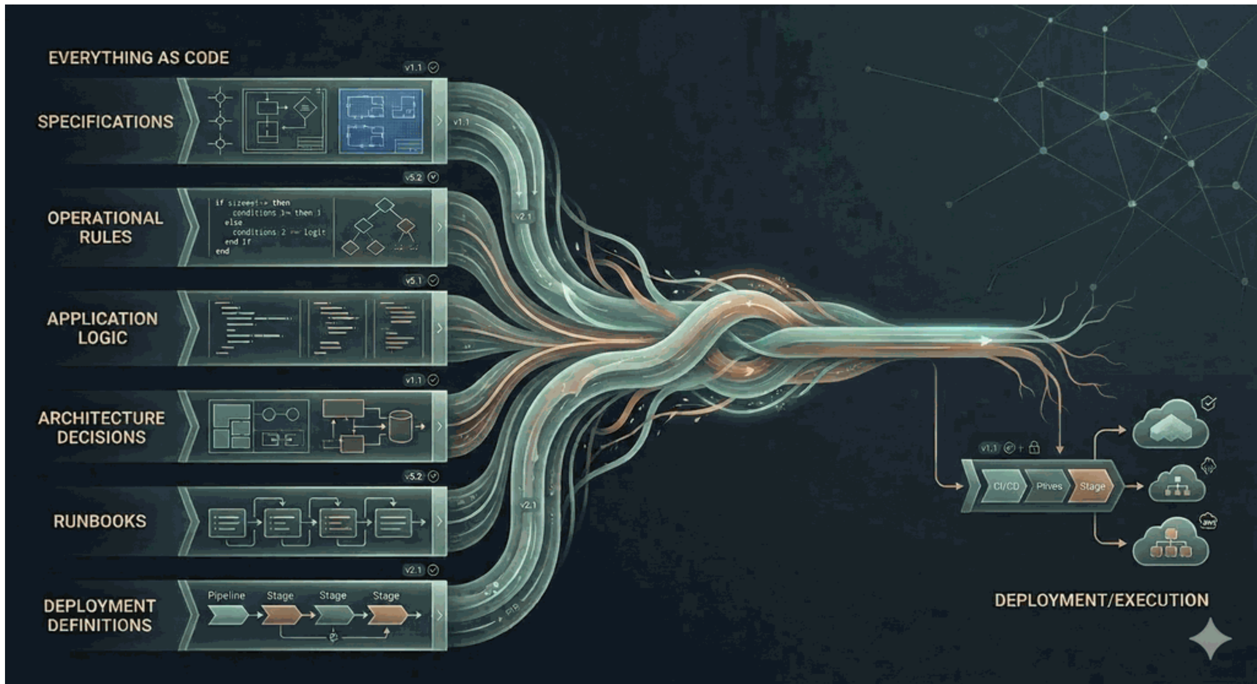

Figure 16: Everything as Code

Implementing this capability means steadily moving important artefacts into versioned, testable forms. Specifications, rules, architectural decisions, runbooks, deployment definitions and other operational knowledge should increasingly behave like code: reviewed, validated, versioned and kept current. As stated in the Vision, when agents can read a precise, current problem definition, they can generate, test and deploy a solution at run-time. The problem definition becomes a living specification that continues to shape the product as it changes.

That is not a speculative end-state. It already works in narrower domains. OpenAPI and schema-first development turn API specifications into generated clients, routers and validation. Policy engines turn codified rules into run-time decisions. Decision models turn business rules into executable logic. Spec-driven development tools now use living specifications to drive planning, task breakdown and implementation for coding agents. The frontier is joining these patterns together so that more of the product becomes a governed, executable expression of the current problem definition rather than a frozen implementation of last quarter's assumptions.

The operational question is therefore twofold: what should be pulled into this discipline, and what kind of environment can reliably consume it?

- **Already code**: application logic, schemas, migrations, pipelines and deployment definitions
- **Should become code**: specifications, architecture decisions, runbooks, alerts, dashboards and the most important operational rules
- **Harder to codify but still worth structuring**: roadmap logic, design rationale, decision records and tribal knowledge

This only works if the surrounding environment can actually consume those artefacts reliably. In practice, that usually means:

- explicit schemas and contracts
- strong typing or structured interfaces
- versioned configuration
- separable policy/rule layers
- testable boundaries

- predictable build/deploy pipelines

Automation-resistant environments tend to have:

- implicit behaviour spread across the codebase
- business rules buried in controllers/services
- weak separation between policy, workflow, and implementation
- lots of tribal knowledge
- fragile or manual deployment/testing paths

Once artefacts are codified and the environment can consume them, they still have to be maintained with the same discipline as code:

- changes are versioned and reviewable
- critical artefacts are validated or tested
- documentation and specifications are updated with code changes
- stale artefacts are pruned as tech debt, not just accumulated
- high-value artefacts are auditable
- maintaining these artefacts is part of delivery, not side work

AgentOps

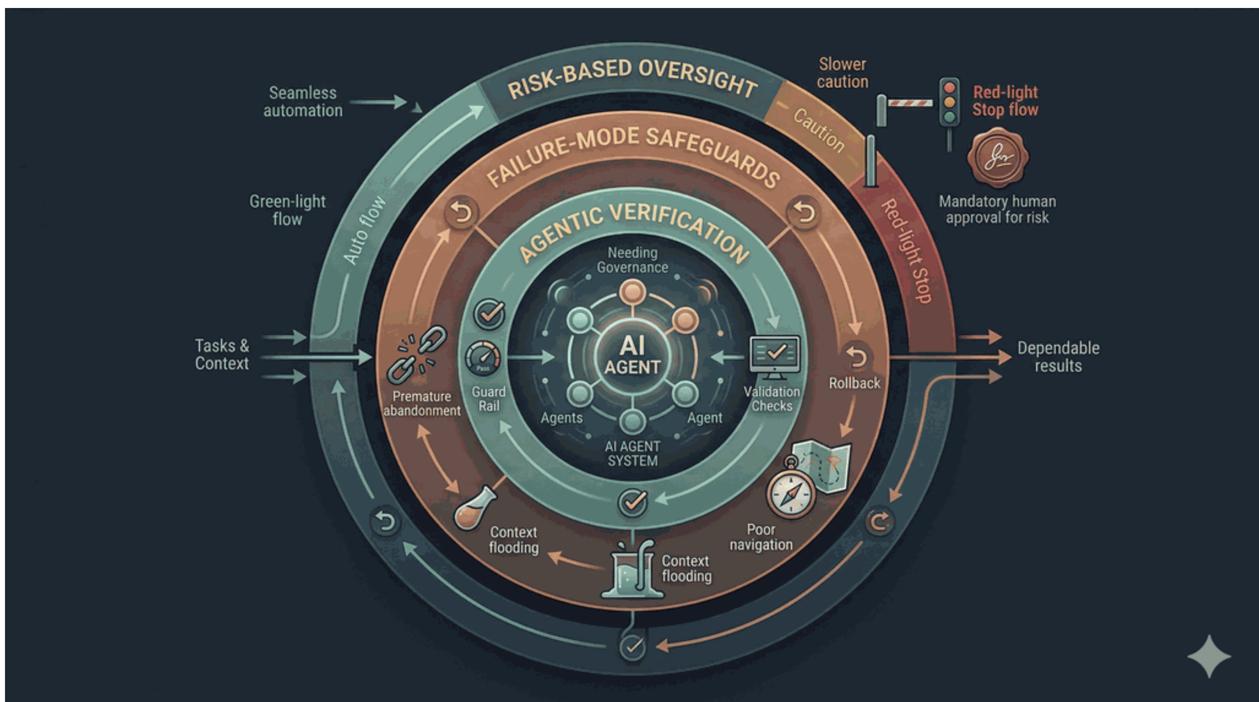

Figure 17: AgentOps

AgentOps (agent operations) is the operational discipline for managing AI agents across their lifecycle. As autonomy grows, teams need it to make agent behaviour dependable, not just observable. AgentOps also owns and operates context infrastructure (the layered, versioned context described earlier) so that context, guard rails and agent behaviour are managed as one system.

Within that broader discipline, three capabilities matter most here: **agentic verification**, so guard rails and validation become technical capabilities rather than aspirations; **failure-mode safeguards**, so teams design for how agents actually fail rather than only checking outputs; and **risk-based oversight**, so the level of human involvement changes with the nature of the work rather than staying fixed.

Agentic Verification means codifying standards, running checks at the right points, surfacing policy failures, and monitoring whether automated verification is actually improving trust. The goal is not

necessarily maximum automation but dependable verification that lets human attention move up the stack.

For that to work in practice, AgentOps also needs an operating discipline around the agents themselves: versioned agent definitions, monitoring for output quality and drift, active management of context quality, and clear escalation paths when behaviour degrades. In practice this usually includes running agent evals operationally: replaying representative tasks, comparing versions, tracking regressions, and checking whether changes improve speed, cost and reliability rather than merely sounding more capable. Core practices usually include:

- agent definitions as code
- performance monitoring
- incident response for degraded behaviour
- agent versioning
- context quality management

The most useful signals are usually operational rather than theatrical: first-attempt success rate, drift over time, context collapse, policy violation trends, and the ability to compare or roll back agent changes when behaviour worsens. Staffing models, platform choices and rollout patterns should be treated as implementation details and managed in the implementation companion document.

Failure-mode safeguards matter because agents fail differently than humans. Traditional technical failures are often visible and caught quickly; the harder problems are metacognitive. Agents can stop too early, lose the thread of the task, repeat failed actions, navigate badly, or verify work poorly whilst sounding confident. The implementation challenge is therefore not only to test outputs, but to build systems that make agent process more dependable.

The failure modes to design for most explicitly are:

- **Premature task abandonment**: the agent stops before the work is actually complete
- **Context flooding**: the agent gets lost in error handling and loses the primary goal
- **Repeating failed actions**: the agent retries the same ineffective approach without integrating feedback
- **Poor navigation and incorrect verification**: the agent edits the wrong places, misses dependencies, or claims success too early

Risk-based oversight is how those controls get applied in practice. The concept from Stage 3 becomes real only when the organisation can classify work consistently and attach the right controls to each class. In practice that usually means combining several dimensions, such as delivery phase, code area and impact radius, to decide whether work can flow through automated checks alone or needs escalating human approval.

The exact model varies, but the pattern is durable:

- Lower-risk work can rely more heavily on agent review and automated verification
- Medium-risk work typically combines automated checks with targeted human review
- High-risk work keeps mandatory human approval and stronger compliance or security controls

The key is not to eliminate humans uniformly, but to apply scarce human attention where it matters most.

Key principle: Test the agent's process, not just the code output. Success means the agent worked intelligently, not just got lucky.

Agentic Service Management (AGSM)

AgentOps addresses the engineering discipline: making agents buildable, testable and deployable. But every major technology shift extends two traditions. Cloud computing created DevOps for the engineering pipeline and extended ITIL service management for operational governance. Agents require the same dual extension. Agentic Service Management (AGSM) is the service management counterpart

to AgentOps: the practices that make agents governable, trustworthy and compliant, governed by the Humans Above The Loop principle.

The framing that ops departments become "an HR function for AI agents" is insufficient; there's more to it than onboarding, monitoring, retiring agents. AGSM must build the structure as well: the governance model, the trust infrastructure, the supply chain assurance and the knowledge systems that make autonomous agents operationally defensible.

Agent lifecycle management extends configuration management to agents. Agent registries serve the role that CMDBs serve for infrastructure: a centralised system of record tracking agent identity, ownership, capabilities, version, status (development, test, production, retired) and policy bindings. Agent definitions become declarative governance artefacts — an intent file and a lockfile — mirroring infrastructure-as-code patterns. When an agent degrades, the registry tells you what changed. When a new agent is proposed, the registry enforces governance before deployment. When an agent is retired, the registry preserves the historical record.

Operationalising trust extends service level management and change management to agents. What are the quality, cost and first-attempt success commitments for an agent in production? How are those commitments monitored? When an agent's behaviour changes — through a context update, a model swap, a policy revision or an eval regression — that change needs a governance process: risk assessment, decision record, rollback path. Not necessarily a Change Advisory Board, but a process that is lightweight and defensible. The volume of agent-driven change makes per-change approval unworkable. Instead, AGSM develops the framework within which changes flow: risk classification rules, context governance policies, eval thresholds, rollback triggers and escalation criteria. The framework is the governed artefact, not the individual change. Governance operates on the system, not on each transaction; AGSM then measures whether the framework is working and revises it when the metrics show it is not catching what it should.

Managing the comprehension gap extends knowledge management to agents. The comprehension gap is the risk that agent throughput outpaces the organisation's ability to understand and defend what changed. The response is not to slow agents down but to build the knowledge systems that keep human understanding close enough. Each agent session produces two outputs: the work product and the lessons learnt. Those lessons must flow back into the context hierarchy, weighted by confidence, subject to freshness decay, and traceable to the session that produced them. Without this discipline, every session starts from scratch, the same failure class recurs, and the organisation pays the same learning cost repeatedly. The maths is simple: pay the innovation tax once or pay the learning cost manyfold. On the human side, reviewers need better tooling, not more hours: summaries, diff explanations, risk annotations and intent traceability help reviewers focus on judgement rather than reconstruction. The goal is not that humans inspect everything but that humans can inspect anything, quickly enough to maintain defensible oversight.

Agentic supply chain extends supply chain assurance to agent-produced artefacts. Software moves across organisational boundaries; trust must be portable. Agents introduce new security vectors (prompt injection through upstream context, tool-use manipulation, privilege escalation through chained actions, and data exfiltration through seemingly legitimate API calls) and new compliance requirements (demonstrating that agents operated within approved constraints, that context was governed and current, that verification matched the risk). The operational response is defence in depth: least-privilege tool access, strict input and output validation at every boundary, runtime guardrails that block prohibited actions, and continuous monitoring for anomalous behaviour. The evidence chain from intent through context through agent action through verification through approval is the agentic equivalent of an audit trail.

Key principle: The question is not whether to govern agents but how to make that governance lightweight enough to sustain and rigorous enough to defend.

Workflow Patterns

This is where the workflow patterns introduced in Shift 2 become concrete. As organisations move from supervised use towards orchestration, delivery flows change with them: pull requests move from human-authored and human-reviewed towards agent-authored, agent-reviewed and risk-governed flows; testing shifts from validating human code to validating agent output against specifications and guard rails; deployment shifts from scheduled and manual towards increasingly automated and risk-based release paths.

Typical workflow patterns look like this:

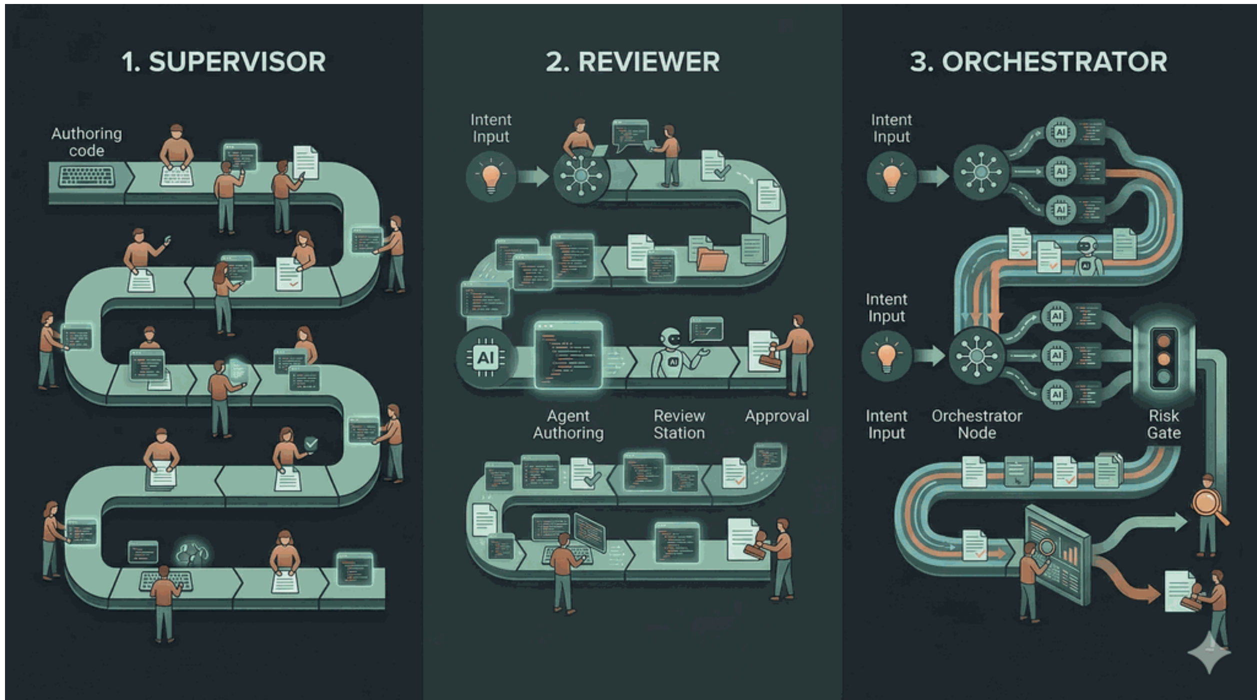

Figure 18: New Workflow Patterns

Stage 1 Supervisor pattern:

1. Developer creates or updates code with AI assistance
2. Developer reviews and edits the agent output directly
3. Developer runs tests and checks the result
4. Developer creates the pull request
5. Human review and normal CI/CD gates still apply

Stage 2 Reviewer pattern:

1. Developer defines the task with clearer acceptance criteria
2. Agent works from layered context rather than a one-off prompt
3. Agent produces code and supporting test output
4. Other automated checks or agents review for quality, policy and security
5. Developer reviews intent alignment and approves the change

Stage 3 Orchestrator pattern:

1. Developer provides intent, constraints and risk boundaries
2. An orchestrator decomposes work and coordinates specialist agents
3. Agents build, test and review in parallel where appropriate
4. The oversight path varies by risk level
5. Human attention focuses on approval of higher-risk work and validation of intent achievement

The exact workflow design will vary, but the underlying capability requirements are stable:

- Shared context and specifications that agents can reliably act on
- Verification systems that narrow the need for routine human review
- Review and deployment paths that reflect change risk rather than treating all work the same
- Operational visibility into what agents did, why they did it, and whether the result should be trusted

Orchestration Infrastructure

The orchestration patterns introduced in Stage 3 depend on supporting infrastructure. The market currently looks more like a set of vendor-specific orchestration runtimes than a true interoperable layer, so the only option is to build our own control plane.

The supporting architecture will vary, but the capability requirements remain stable:

A **shared internal model** of specifications, decisions, tasks and progress. In practice, that often starts with centrally defined conventions: common folder taxonomies, canonical markdown and YAML artefacts, agreed filenames, frontmatter fields and schema-validated metadata. Over time, those contracts usually harden into shared libraries and services so different runtimes are working from the same governed structure rather than inventing their own.

A way to **route, sequence and reconcile work** across agents and runtimes. In practice this usually means combining one or more of the four marketplace buckets: agent runtimes to coordinate specialist workers, durable workflow engines to manage long-running state and retries, event backbones to route messages and work, and emerging interoperability protocols to connect across vendor boundaries. Some of that may be implemented through vendor tooling, but the organisation needs to own the rules of routing and reconciliation.

Consistent visibility into status, failures, handoffs and audit trails. Standard observability foundations such as logs, analytics and OpenTelemetry help here, but agentic systems need an additional semantic layer: visibility into task trajectories, tool use, context and memory reads, agent-to-agent handoffs, guard rail interventions and human approvals. Without that, teams can see that something happened, but not why the agentic process succeeded or failed.

Verification, policy and escalation hooks that apply regardless of vendor. Guard rails cannot live only inside whichever orchestration product is fashionable this quarter. They need to be expressed in declarative forms that can be tested at dev-time, build-time and run-time, so policy checks, approval points, risk classification and escalation paths apply consistently across runtimes and survive tool changes.

Mechanisms for **converging on a validated outcome** without losing control. Parallel exploration only creates value if the system can compare alternatives, resolve disagreements, surface uncertainty and land on an outcome that is reviewable and trustworthy. The point is not just to let many agents run; it is to bring their work back together under governed decision-making.

This is the technical layer that makes multi-agent orchestration usable in practice rather than just conceptually appealing.

Implementing Business Change

The journey is not only technical. If agentic development changes who defines work, who reviews it, how quality is established and where judgement sits, then the organisation itself has to adapt. Measurement can tell you that constraints have moved; it cannot by itself redesign the organisation around them. Business change is the organisational work that makes change durable: learning has to compound, discovery has to improve, and trust has to grow. The people working above the loop also need room to grow, because the system still depends on human judgement above the loop.

Durable Change as Joint Optimisation

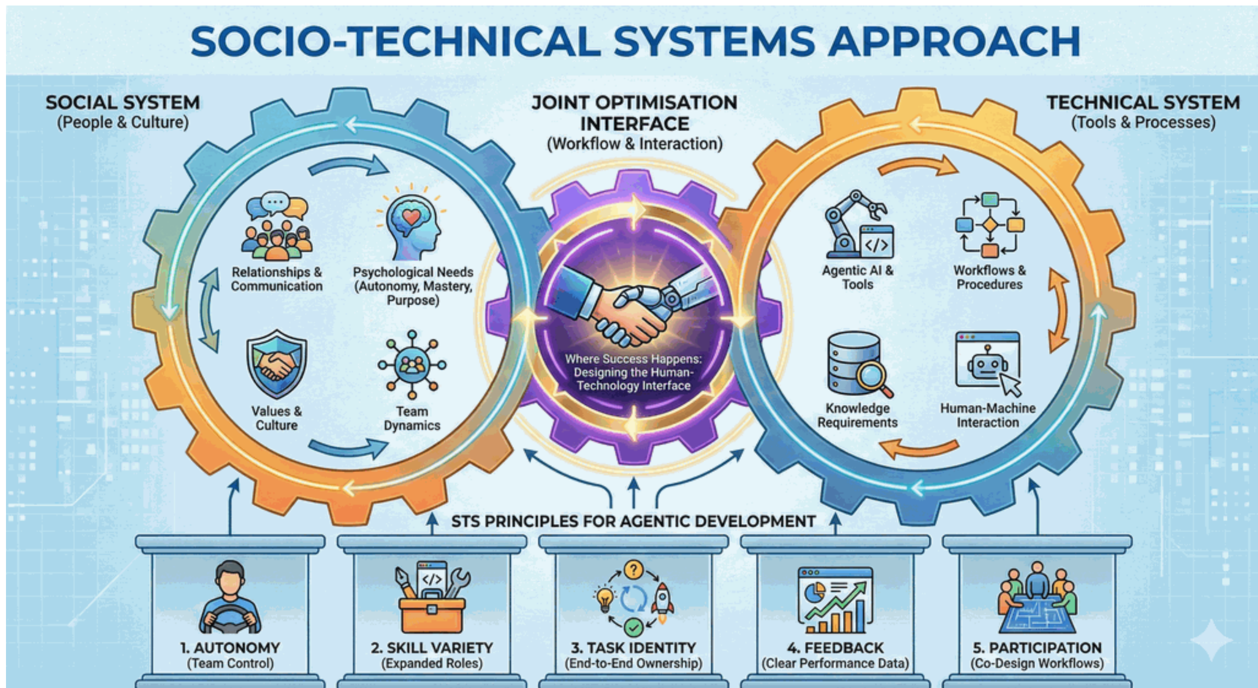

Figure 19: Socio-Technical Systems Approach

Socio-technical systems thinking provides the framing for this Journey. Durable change means jointly optimising the social system and the technical system as the operating model changes. If the technical system gets faster while the social system loses autonomy, mastery, purpose or belonging, the apparent productivity gain will not last. In that sense, durability is not only about whether output stays high; it is also about whether the new way of working improves the conditions that let people trust and thrive in the system.

That changes the practical question. The organisation does not only need better tools; it needs better interaction patterns, incentives, team habits and decision rights. Roles evolve, but they should not collapse into passive supervision. Product managers prototype directly. Developers spend more time on specifications, context and verification. QA work shifts towards guard rails and risk controls. Teams become more asynchronous, but still need clear ownership and meaningful moments of shared judgement.

This framing also sharpens the skills question. Emerging evidence suggests that using agents without sustained mental engagement can weaken skill formation. Some atrophy is healthy: if a capability is genuinely low value, the organisation should be willing to delegate it to agents. But erosion in high-value, high-risk work is different. Where the operating model still depends on human judgement, debugging, review and validation, those capabilities need to deepen rather than decay. Weakening them is a durability failure. That does not mean excluding agents from risky work; it means choosing interaction

patterns that match the risk profile of the work. The business challenge, then, is to strengthen the skills the operating model still depends on while allowing lower-value capabilities to atrophy safely.

Change Paths and Adoption

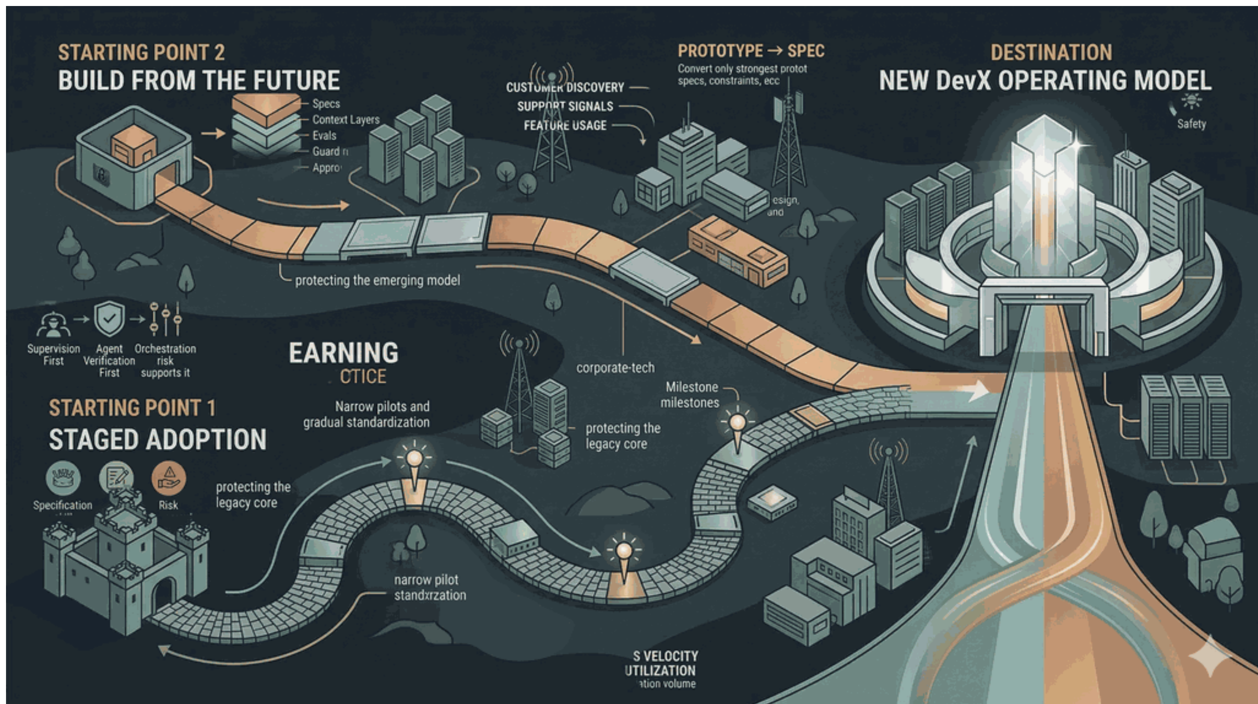

Figure 20: Change Paths

Even with the right operating model, organisations do not all absorb change at the same rate. Their starting point depends on factors such as leadership engagement, learning climate, available resources, prior change experience, and the degree to which teams can turn local experiments into shared operating practice. A sensible New DevX Journey therefore begins by assessing not only transformation ambition, but also the organisation's current capacity to learn and reconfigure.

In practice, two broad patterns recur. Where change capacity is limited, or where the new model is tightly entangled with the existing core, the safer route is staged adoption: narrow pilots, visible wins, and gradual standardisation. Where the target model can be protected from the legacy system and leadership is willing to back it, organisations can sometimes build from the future first and then work backwards to integrate the new governance, workflows and capabilities needed to sustain it. A word of caution: most organisations do not leap directly to the future state. They usually move through a capacity-constrained sequence of experimentation, local adoption and gradual codification, sometimes combining that staged path with build-from-the-future initiatives at the edges.

The timing still matters. Referring back to the maturity stages, DevX as Supervisor usually creates the first visible wins through assistance on repetitive work. DevX as Reviewer is where shared context, guard rails and verification start to compound. DevX as Orchestrator takes longest because it depends on mature trust, governance and organisational readiness. Treat timelines as directional rather than contractual; the goal is to match the path to the organisation's capacity, not to force every organisation down the same route at the same speed.

In practice this means:

- assessing the organisation's change and learning capacity before designing the adoption path
- choosing whether to stage the change in the core, build from the future, or combine the two
- proving value and reliability before widening adoption

- protecting whichever part of the system most needs stability: the legacy core in staged change, or the emerging future-state model when building from the future
- quantifying the innovation tax of change explicitly: time spent learning new toolchains, workflows and controls is time not spent on short-term delivery

Whatever path an organisation takes, the New DevX Journey requires business change in three key areas: learning, discovery and trust.

Learning

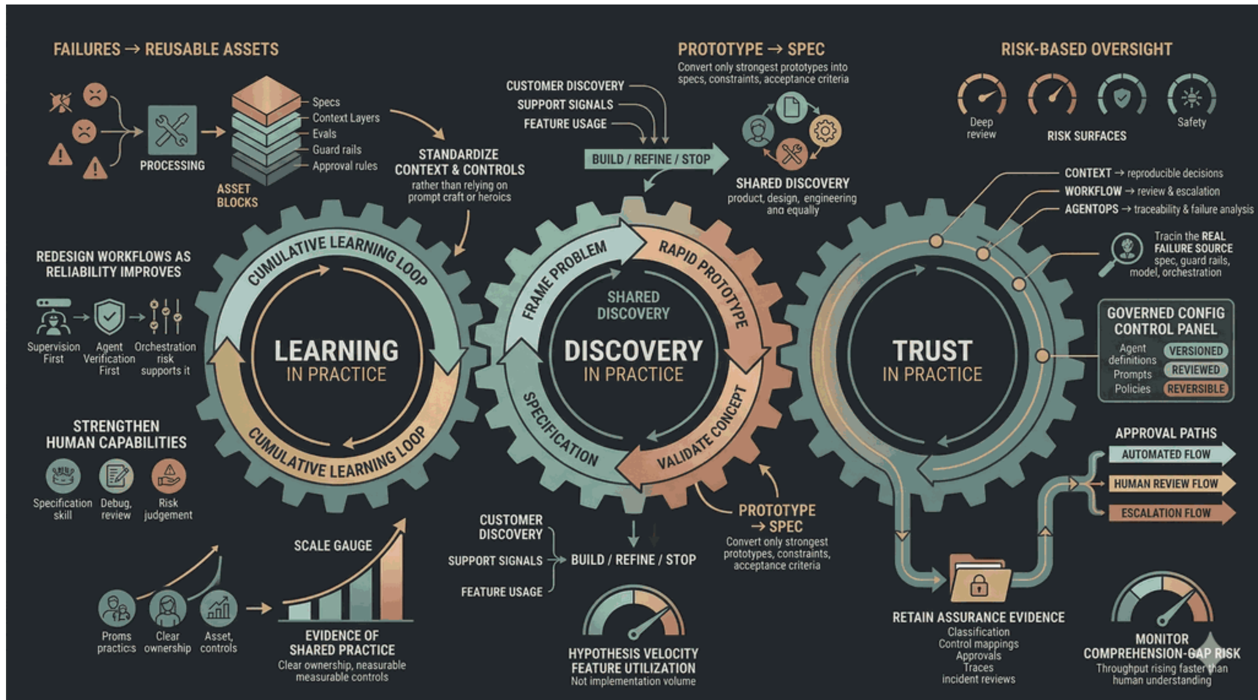

Figure 21: Learning, Discovery and Trust in Practice

Progress along this journey depends on learning how to implement agentic ways of working safely. Organisations need to discover what makes autonomous work reliable, governable and worth trusting, then turn those lessons into shared context, controls, workflows and team habits. They also need to build the capabilities that let those new patterns spread without eroding the human judgement the operating model still requires.

The learning capability matches the maturity of the developer's capability. In **DevX as Supervisor**, people learn prompting, validation, failure recognition and when to stay cognitively engaged rather than delegating too quickly. In **DevX as Reviewer**, teams learn context engineering, specification writing, guard rail design and how to review work against intent rather than only against code style. In **DevX as Orchestrator**, organisations learn intent articulation, orchestration, risk classification and system-level judgement about where autonomy should and should not increase.

In practice this means:

- turning repeated agent failures into reusable assets: better specifications, stronger context layers, evals, guard rails and approval rules
- learning which context, controls and ownership patterns actually improve outcomes, then standardising those inputs rather than relying on prompt craft or individual heroics
- redesigning workflows as reliability improves: human supervision first, agent-to-agent verification next, and wider orchestration only where risk and evidence support it, whether that learning happens in the core or in a build-from-the-future environment

- strengthening the human capabilities the operating model still depends on: software engineering expertise, wider domain knowledge, specification, debugging, review and risk judgement
- scaling only when local experiments or build-from-the-future patterns have become shared operating practice with clear ownership, measurable controls and evidence that they work

The point is to make implementation learning cumulative. Teams should not just discover that an agent failed, or that a workflow felt faster. They should learn what changed in the operating model so the next attempt becomes more reliable, and then decide what should spread back into the core. Capability building is therefore not only a training problem. It is also a workflow design problem. The everyday interaction patterns with agents should bolster the skills the organisation still values and allow the ones it values less to atrophy.

Discovery

As code generation gets cheaper, discovery becomes a larger source of advantage. The constraint moves upstream into defining problems worth solving, sharpening intent, and deciding what evidence is strong enough to justify commitment. Discovery therefore becomes part of the operating model rather than a pre-delivery activity carried out by product alone.

In practice this means:

- running a repeatable loop of problem framing, rapid prototyping, concept validation and specification
- letting product managers, designers and engineers share discovery work rather than handing it across functionally
- converting only the strongest prototypes into explicit specifications, constraints and acceptance criteria for delivery
- using customer discovery, support signals and feature usage to decide what gets built, refined or stopped
- rewarding hypothesis velocity and feature utilisation rather than implementation volume

The practical output of discovery is not a prototype by itself. It is a clearer statement of intent that downstream agentic delivery can actually consume. The prototype perfects intent; the specification makes it buildable. That handoff matters because weak discovery creates weak specifications, and weak specifications become unreliable autonomy later in the journey.

Trust

Trust is often described as confidence in the tools, but in practice it is confidence in the whole operating system: the context, the controls, the oversight model and the people asked to exercise judgement within it. That is why durability in trust cannot be measured only by autonomy horizon or context integrity. It also depends on whether people remain cognitively engaged enough to validate, debug and challenge the work that agents produce.

Trust also has to survive external scrutiny. Compliance is, in effect, trust made provable: the ability to show an external body how a system has been risk-classified, which controls follow from that classification, and why those controls are sufficient. Under frameworks such as the EU AI Act, that may mean demonstrating both the risk class of the software and the adequacy of the human oversight, verification and governance wrapped around it.

As agent output volume rises, human trust weakens in a different way. Even when agents are working within strong controls, reviewers may no longer have enough time or cognitive bandwidth to understand what changed, why it changed, and whether it complies with project, organisational, industry and legal standards. That creates a **comprehension gap**: the organisation can be producing more artefacts than it can meaningfully inspect. For a regulated organisation, how can it defend its compliance position? The answer is Trust as a measurable metric.

Trust in agentic development cannot stop at the edge of one firm. Software is a supply chain: code, models, packages, services and infrastructure move across organisational boundaries. In that environment, trust has to be portable. It must be expressed as evidence that suppliers, customers, auditors and regulators can independently verify, not only as confidence inside the delivery team.

Trust becomes a practical business capability when the organisation builds a risk-and-controls model that can satisfy both internal confidence and external scrutiny.

In practice this means:

- applying oversight by risk surface, not uniformly: not every file, workflow or service needs the same scrutiny, so critical interfaces, regulated flows, high-impact libraries and safety-relevant changes should face deeper review than low-risk areas
- connecting the shifts: context infrastructure makes decisions reproducible, workflow patterns define where review and escalation happen, AgentOps owns traceability and control operation, and AGSM owns governance, trust evidence and cross-system failure analysis
- investigating the real source of failure: a production issue may trace back not only to an implementation bug, but to a defect in specification, guard rails, model choice, orchestration or operating assumptions
- controlling the agentic system itself: agent definitions, prompts, policies, tool permissions and context sources should be treated as governed configuration: versioned, reviewed and reversible
- designing approval paths: organisations need clear rules for what can flow through automated checks, what needs targeted human review, and what requires escalation or sign-off
- retaining assurance evidence: classification decisions, control mappings, approval records, action traces, validation results, incident reviews and exception logs should be kept as proof that controls exist and operate
- monitoring comprehension-gap risk: teams should watch for signals that agentic throughput is rising faster than human understanding, such as very large PRs, bursts of agentic commits in critical human-machine-interface libraries, shallow review times, repeated reversions or reviewer uncertainty

The point is to turn trust from a cultural claim into an operating capability. Internal trust and external compliance then rest on the same foundation: a clear understanding of risk, controls proportionate to that risk, and evidence that those controls are actually working.

What Stays Human

From Ceremonies to Flow

The operating model changes described above have a direct consequence for how teams coordinate. The agile paradigm was designed for two-week human coordination cycles. Sprint planning, daily stand-ups, sprint reviews and retrospectives assume that work moves in batch, that coordination is synchronous, and that the same people who plan the work also execute it. That last assumption is the one that breaks first. When agents execute the development work, the question is not "how do we run the same ceremonies faster" but "what do humans need to decide, and what do agents need to know?"

The replacement is not a faster version of the same activities. It is a separation into two distinct layers: an **intent layer** where humans define what problems matter, what outcomes to pursue and what constraints to elevate; and an **execution layer** where agents sequence, prioritise and coordinate their own development work within the boundaries that intent sets. Each layer needs its own framework. Forcing both through the same ceremony conflates strategic decisions with implementation logistics.

Sprint planning splits into intent planning and execution orchestration. Humans stop estimating how long work takes and start defining which problems matter most, ranked by constraint signal and business value. Separately, agents need a framework for sequencing their own work: dependency ordering, parallelisation decisions, resource allocation across concurrent tasks. That framework is expressed as context (specifications, priorities, constraints, orchestration rules), not as a ceremony.

Daily stand-ups become decision forums. Agent traces, dashboards and operational signals provide continuous visibility into execution status; no human needs to report that. Synchronous time is reserved for the decisions that agents cannot make: resolving ambiguity in intent, choosing between competing priorities, adjusting risk tolerance, unblocking work that requires judgement the context does not encode.

Sprint reviews become outcome reviews. The cadence shifts from "what did we ship this sprint" to "what measurable outcomes changed since last review." The review forum focuses on whether shipped work achieved its intent, whether feature utilisation and flow metrics confirm the hypothesis, and where the constraint has migrated.

Retrospectives become constraint reviews. The question shifts from "how did we execute?" to "is our intent-to-execution pipeline working?" Are agents receiving clear enough context to self-organise? Is the intent layer producing decisions fast enough? Are the boundaries between human judgement and agent execution in the right place?

Estimation becomes risk classification. When code generation is cheap, the useful question is not "how long will this take" but "what level of risk and oversight does this work require, and how clear is the intent?" Risk classification then drives workflow routing: low-risk, high-clarity work flows to agents with minimal human involvement; high-risk, ambiguous work triggers layered verification.

Above the Loop

Working the way agents work does not remove the need for human judgement; it changes where that judgement is applied. As execution becomes increasingly automated, this enables the most valuable human contribution to move upward into deciding what should be done, what should not be done, and what level of risk is acceptable.

This is what being above the loop means. Humans stop coordinating every implementation step and instead govern the system that performs the work. The role shifts from operator to steward: clarifying intent, shaping constraints, reviewing outcomes, making great decisions and retaining accountability for the consequences.

Several categories of judgement remain fundamentally human.

Ethics

Being ethical is not negotiable. Yet making ethical decisions means balancing trade-offs between the needs of individuals, cohorts, customers, society and being a successful business. Ethical governance cannot be formalised as a static constraint optimisation problem, which is why agents are bad at being ethical. Moral agency and liability cannot be delegated. Within the loop are agents analysing the trade-offs; above the loop humans retain responsibility for deciding what is ethically responsible, and for redress if it goes wrong.

Compliance

Compliance is sometimes reduced to its technical essentials: collation of signals from work done, then presented to regulatory bodies for approval. Following this complex process is within the sweet spot for agents. Agents can help implement controls, monitor adherence, generate evidence and share with regulators. However, compliance is also a business function. Deciding how regulations apply, interpreting ambiguous requirements, determining whether controls are proportionate, and defending those decisions to auditors, regulators and customers - all of which agents are bad at. Within the loop are agents developing compliance telemetry; above the loop are humans exercising risk management, negotiating with regulators, and when compliance is done well, creating business advantage.

Product Judgement

Product Management is an optimsation problem: bring the next feature to market that has the biggest impact on KPIs for lowest investment. Agents can analyse usage data, extrapolation improvements, and of course generate the features. But creating meaningful value is rarely extrapolation, it is rejecting norms to find product-market fit by deeply understanding human behaviour. It is interpreting weak signals to discover unexpressed needs. Within the loop are agents apply Product Management; above the loop are humans applying Product Judgement. As the marginal cost of feature implementation approaches zero, the strategic value shifts entirely to this high-judgment boundary.

Organisational Prioritisation

We have seen how the constraint moves from the volume of software produced to the quality of intent. The finite resource becomes human attention. Organisations must still decide which problem to tackle next and how many of the previous solutions to maintain. Within the loop, agents are building software to solve problems according set priorities. Above the loop are humans reconciling organisation context, stakeholder interests and long-term objectives to set the priorities.

Risk Acceptance

Reframing as risk acceptance unifies these categories. Agents understand risk through classification, controls and evidence, but they do not undersatnd risk appetite. Humans have great appetitle for taking risks, but often make poor choices through bias. By placing agents within the loop and humans above the loop, we can design a system for risk acceptance that blends the strengths of both approaches. Humans apply governance frameworks to agents, agents measure telemetry and develop insights, humans evaluate, accept or remediate.

Conclusion: The Redesigned System

The promise of agentic software development is real, but the lesson of this Journey is that faster code generation is not, by itself, the transformation. The real change is redesigning the system around it. If agents can write a limitless amount of code, the question is not how to produce even more. The question is how to decide what is worth building, how to verify it, and how to keep learning where the next constraint will appear.

That is the “so what?” of New DevX. Speed and rigour have to be sequenced rather than forced into opposition. Ways of working have to be redesigned around how agents actually succeed. Context, rules and specifications have to become governed inputs rather than scattered background knowledge. And business change has to concentrate on three capabilities: learning, discovery and trust.

The practical takeaway is straightforward. Do not treat AI adoption as a tooling rollout. Treat it as a redesign of workflow, context and governance. Move humans above the loop rather than keeping them inside every step. Expect constraints to migrate and build the habit of following them. Turn important knowledge into versioned, testable artefacts. Match the change path to the organisation’s actual capacity. Strengthen the human judgement the operating model still depends on.

The organisations that benefit most from this shift will not be the ones that simply generate the most code. They will be the ones that learn fastest, specify most clearly, and build the most trustworthy system for combining human judgement with agentic execution.

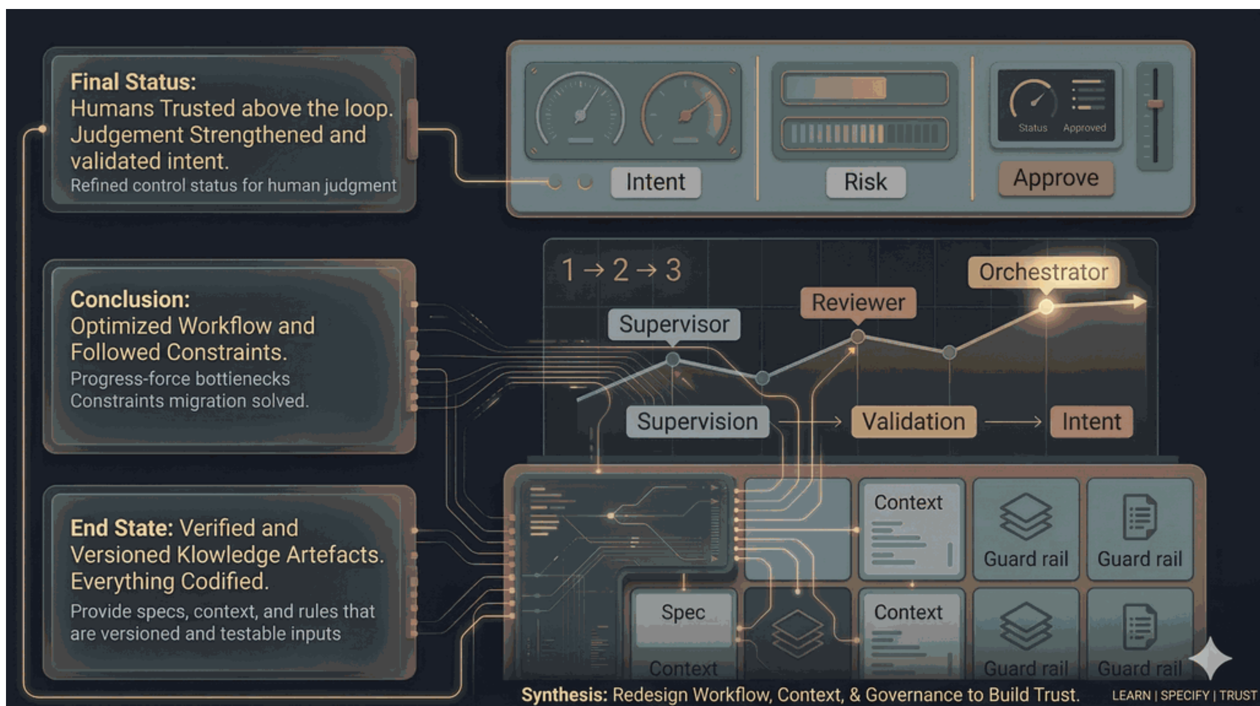

Figure 22: Journey Conclusion

Part 3

The Roadmap

The Roadmap — Contents

Introduction 53

Choose Your Own Adventure 54

The Roadmap 57

Building the New Capabilities 70

Conclusion: The Next Constraint 75

Appendices 76

About the Author 88

Introduction

We've made the case for the New DevX in the Vision and described the transformation in the Journey. The harder question is what to do with that understanding: where to start, how fast to move, what to measure, and how to respond when the constraint shifts. This Roadmap exists to answer those questions.

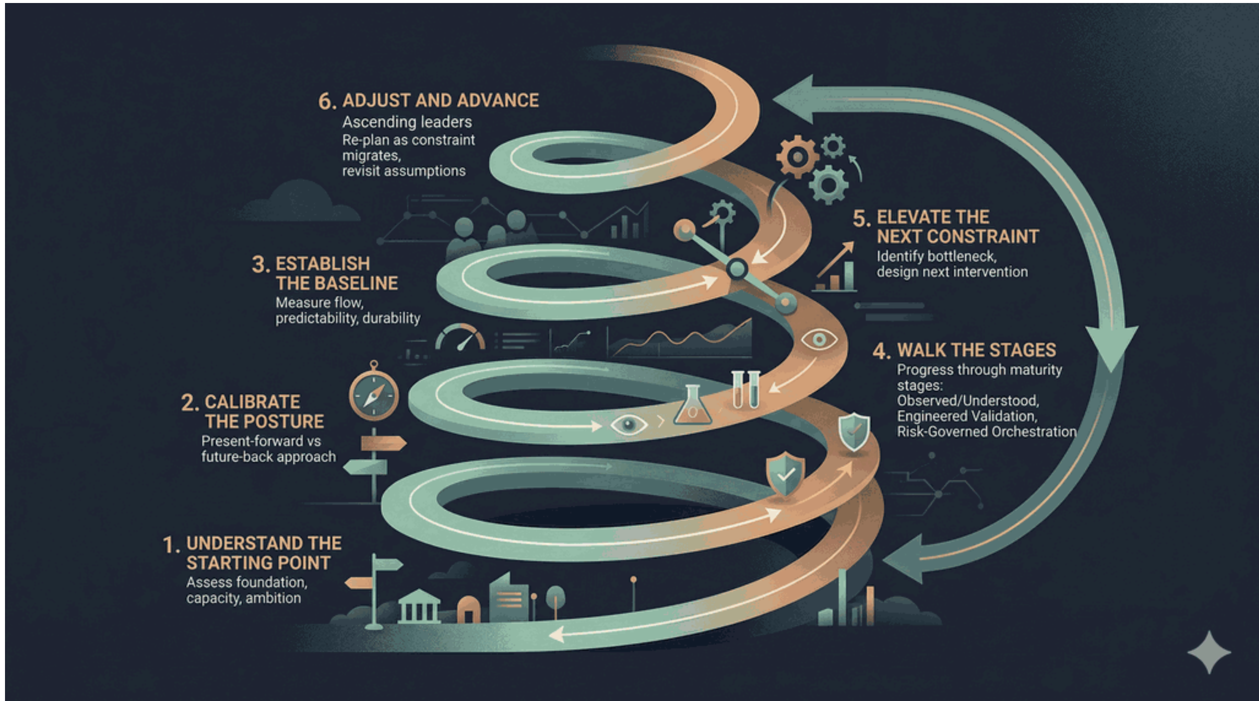

Figure 23: The Roadmap at a Glance

This Roadmap is intended to be a practical sequence of decisions and actions. It helps engineering leaders calibrate the right implementation posture, establish a credible baseline, interpret checkpoints and benchmarks in context, and decide what to do next when progress slows or risks rise. Used well, this document helps a leadership team make progress with greater clarity and confidence. It provides a way to steer deliberately, using evidence, checkpoints and explicit next-step decisions.

Three principles from the Vision resolve the trade-offs that arise along the way. **Elevate the Next Constraint** tells you what to work on: the binding bottleneck, then the next, then the next. **Everything as Code** tells you how to make improvements stick: codify what matters so it is versioned, testable and governable. **Humans Above The Loop** tells you where people belong: governing verification systems and shaping intent, not reviewing every line. These principles are not aspirational slogans; they appear throughout this document as the reasoning behind specific decisions.

How to Use This Roadmap

The document follows a deliberate sequence. It starts by helping you understand where the organisation sits today and how aggressively it should move. It then walks through the three stages of the journey, with benchmarks and checkpoints for each. It describes the new capabilities that the operating model depends on. And it provides an operating loop for deciding what to do next when the constraint shifts.

Each section builds on the previous one, but the process is iterative: constraints migrate, and the right response is to revisit earlier assumptions rather than defend a fixed plan. The goal throughout is not to optimise activity volume but to optimise movement through the system, predictability of that movement, and evidence that the next constraint is being elevated rather than merely relocated.

Choose Your Own Adventure

The Journey makes the case that this transformation is not only technical. Changing who defines work, who reviews it, how quality is established and where judgement sits means the organisation itself has to adapt. Measurement can reveal that constraints have moved; it cannot by itself redesign the organisation around them. Durable change requires jointly optimising the technical system and the social system: if agent-speed delivery gets faster while people lose autonomy, mastery, purpose or clarity of role, the productivity gain will not last.

That joint optimisation shapes how aggressively the organisation should move. Not every team absorbs change at the same rate. Starting points differ by leadership engagement, learning climate, available resources, prior change experience, and the degree to which local experiments can become shared operating practice. A roadmap that ignores those differences either moves too fast for teams that need to build capacity, or too slowly for teams that are ready to push further.

Assessing Readiness

The three principles from the Vision point to three readiness factors. Assessing these factors helps predict where friction will show up first and how aggressively you can act when it does. Bear in mind that some friction is good; doing hard work mindfully to make it easier is better than doing work because it's already easy.

- When foundations are low, Elevate the Next Constraint says fix the engineering baseline before widening agent use. No amount of AI tooling compensates for fragile pipelines and missing tests.
- When context is scattered and practices are improvised, Everything as Code says codify standards, context and guard rails before scaling. Agents need governed inputs; ad hoc knowledge does not scale.
- When supervision is the constraint, Humans Above The Loop says redesign verification rather than adding more reviewers. The response to slow human review is not faster human review; it is engineered verification that lets human attention move up the stack.

Readiness Factor	High	Medium	Low
Engineering Foundations	Git workflows are mature, CI/CD is dependable, tests are meaningful, documentation is current, codebase supports parallel work	Version control and CI/CD exist, but documentation is scattered, manual steps remain, quality controls are uneven	Deployment is still manual or fragile, documentation is sparse or stale, no dependable engineering baseline
Capacity for Change	Leadership sponsors change, teams learn quickly, enough slack to codify new practices rather than improvising forever	Appetite exists, but teams need clear evidence and visible wins before they trust a new operating model	Change fatigue starts high, local experiments rarely become shared practice, the organisation struggles to standardise
Transformation Ambition	Strong appetite to move quickly, protected spaces for experimentation, leadership willing to tolerate unevenness	Ambition exists, but balanced against mixed portfolios, operational dependencies, and controlled progress	The organisation should advance deliberately; the surrounding environment demands steadier adoption and clearer proof before widening

No Single Path

These readiness factors do not produce a crisply defined execution path; life isn't that simple. They calibrate where on a spectrum of possible postures you should operate. Furthermore, because the technology frontier keeps moving in ways no one fully anticipates, this transformation is something to navigate, not something to finish. The Roadmap does not optimise toward a fixed destination. It advances the current system by identifying and fixing the next most important constraint, and your position on the spectrum should be recalibrated as the environment changes.

At one end of the spectrum sits a **present-forward** posture: build infrastructure stage by stage, advancing only when evidence confirms readiness, fixing constraints as they reveal themselves. This posture suits organisations that change deliberately; it trades speed for predictability because the infrastructure is in place before the pressure arrives.

At the other sits a **future-back** posture: define the target operating model, begin operating as close to it as possible, and fill in the missing infrastructure in order of which pressing constraint has the loudest voice. This posture demands an organisation that can change quickly under pressure, because the constraints will arrive before the infrastructure is ready and the response has to be fast, focused and not fragile.

The two postures manage risk differently. Present-forward manages risk through **sequencing**: infrastructure is in place before the constraint arrives. Future-back manages risk through **scoping**: start with a minimum viable business (MVB), the smallest business unit, product area or workflow where the organisation is willing to operate at the target model before the supporting infrastructure is proven. If the team cannot improvise fast enough, if constraints arrive that overwhelm the response, only the MVB is affected. The blast radius is contained by design, and the learning stays useful even if the first attempt is rough.

The intent at every point on the spectrum between these two postures is the same: find the binding constraint and remove it. The postures differ in whether the organisation anticipates the constraint or discovers it by operating closer to the end state.

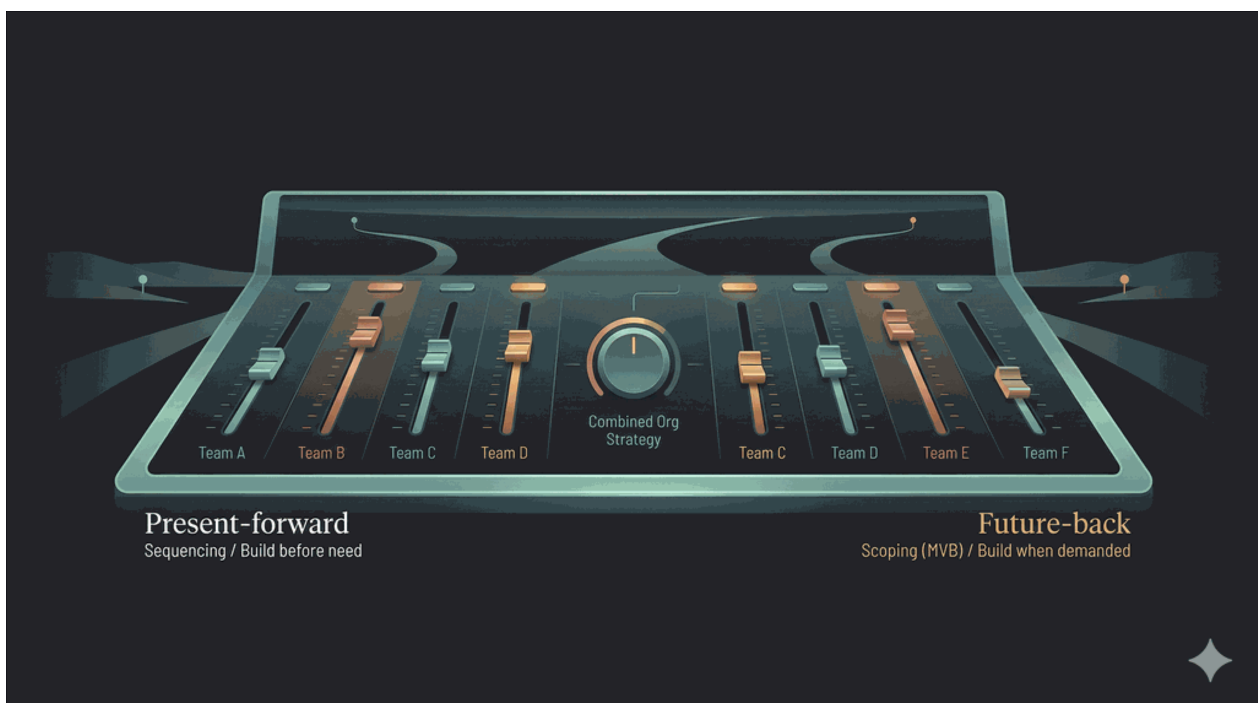

Figure 24: Calibrating the Implementation Posture

Readiness Factor	Present-forward	Future-back
Engineering foundations	Medium to high	Medium to high, especially in greenfield areas
Capacity for change	Low to medium	Medium to high
Transformation ambition	Low to medium	Medium to high
Best fit	Legacy estates, regulated environments, large organisations	Startups, greenfield teams, protected product areas
Intervention style	Build infrastructure ahead of the constraint	Build infrastructure when the constraint demands it
Trade-off	Slower to start, steadier, easier to govern	Faster to learn, rougher, easier to overheat
Pressure profile	Infrastructure absorbs pressure before it arrives	Must change quickly under pressure
Risk management	Sequencing: build before need	Scoping: start with a minimum viable business

Organisation Ambidexterity

Whilst the spectrum is not a binary choice, a single team can't meaningfully operate multiple postures at once. You can't be "build before need" and "improvise under pressure" on the same Monday. Organisations succeed by separating them structurally: different units with different cultures, operating rhythms, incentives and leadership, unified by shared strategic intent, executing in different ways.

The practical implication is that a team settles at a point on the spectrum and operates there with reasonable consistency. A team can be moderately present-forward, or aggressively future-back, or anywhere in between; it is a true spectrum, not two buckets. But the team cannot oscillate between postures day to day, because each posture requires its own habits around planning, improvisation and risk tolerance. The organisation's change strategy places different teams at different points: a regulated enterprise might operate its core platform team present-forward while a protected greenfield product team operates future-back.

Recalibrating

A team's position on the spectrum is stable but not permanent. Reassess quarterly. The posture should shift when:

- recurring constraints suggest the team needs more infrastructure than it has built
- team scope or system complexity outgrows local improvisation
- volatility stays high and the team cannot stabilise under the current approach
- regulators, customers, or leadership now need auditable checkpoints
- confidence grows and the team can afford to move faster
- the technology frontier shifts and the current posture no longer fits the opportunity

In practice, many organisations become more present-forward for the core and more future-back at the edges as they mature. The spectrum is not a one-time choice; it is a posture that recalibrates as constraints migrate and the environment comes into sharper focus.

The Roadmap

The constraint does not disappear as the organisation matures; it migrates. Understanding where the bottleneck sits at each stage is the key to choosing the right intervention.

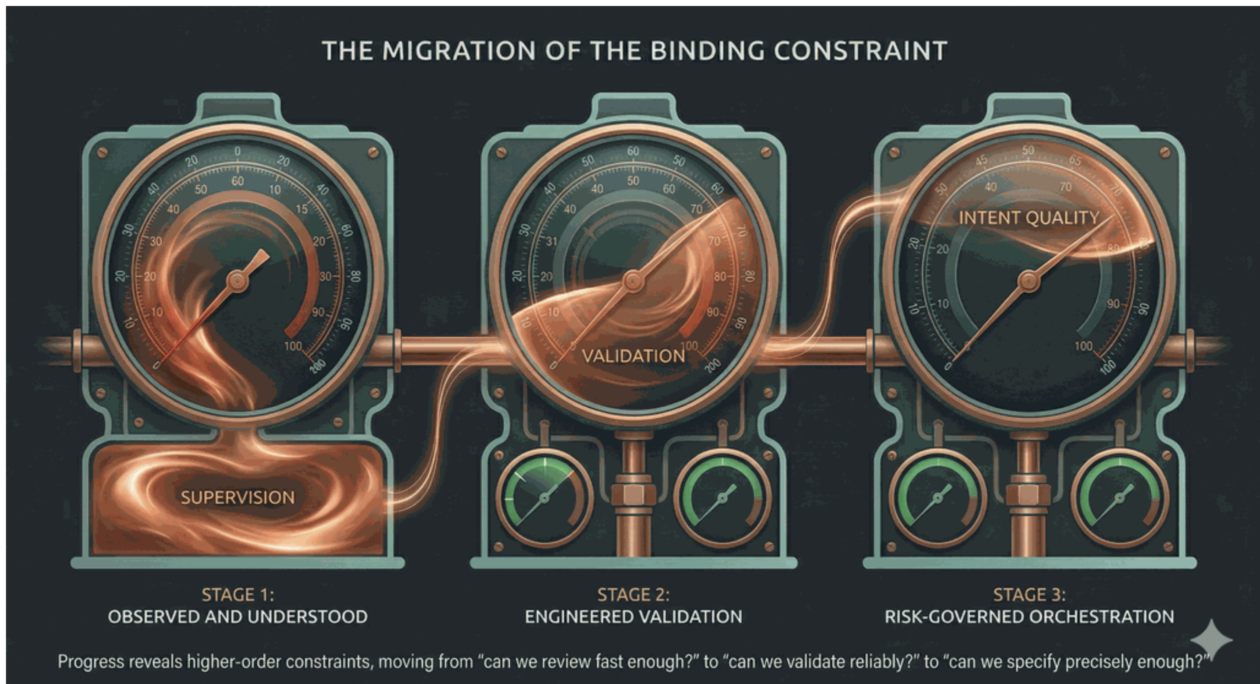

Figure 25: Constraint Migration Across Stages

Every organisation moves through these three stages; readiness changes the path, not the destination. Each stage represents a level of organisational maturity across all three shifts simultaneously. Treat every benchmark value below as a planning scaffold for local calibration.

Stage	Sequence Speed and Rigour	Work the Way Agents Work	Navigate, Don't Arrive	Checkpoint question
Stage 1: Observed and Understood	AI tools are in use; speed comes from individual productivity; rigour still follows existing practices	Context is ad hoc and individual; agents fit into human workflows; supervision is universal	Baselines are being established; constraints are felt but not yet tracked systematically	Can we see where work waits, and do we know what to fix next?
Stage 2: Engineered Validation	Prototype and build phases are emerging as distinct; speed for exploration, rigour for delivery	Shared context, guard rails, and agent verification reduce routine drag; human review narrows to intent	Flow metrics are tracked; constraint migration is visible; improvement is evidence-based	Has validation replaced supervision as the main bottleneck?
Stage 3: Risk-Governed Orchestration	Workflow is mature; prototype, specify, build, deploy is the operating rhythm	Risk-based oversight; multi-agent orchestration on suitable work; humans above the loop	Durability metrics are active; cross-team consistency; continuous navigation	Is intent quality now the main live constraint for the work that should be autonomous?

Benchmarks are organised by five pillars. The first four are **lag indicators** (they tell you what happened); the fifth predicts what is coming:

- **Focus**: are we working on the right things?
- **Speed**: are we delivering efficiently?
- **Predictability**: how consistently are we delivering?
- **Quality**: are we working in the right way?
- **Durability** (lead): will these gains last?

Durability metrics sit beside the lag metrics, not replace them. Productivity metrics can look healthy while durability metrics quietly deteriorate; measure both, or you risk mistaking a short-term spike for a sustainable trend. Where strong cross-industry benchmarks exist, the numbers below borrow from that research; where they do not, they are conservative starting heuristics to replace after local baselining.

Stage 1: Observed and Understood

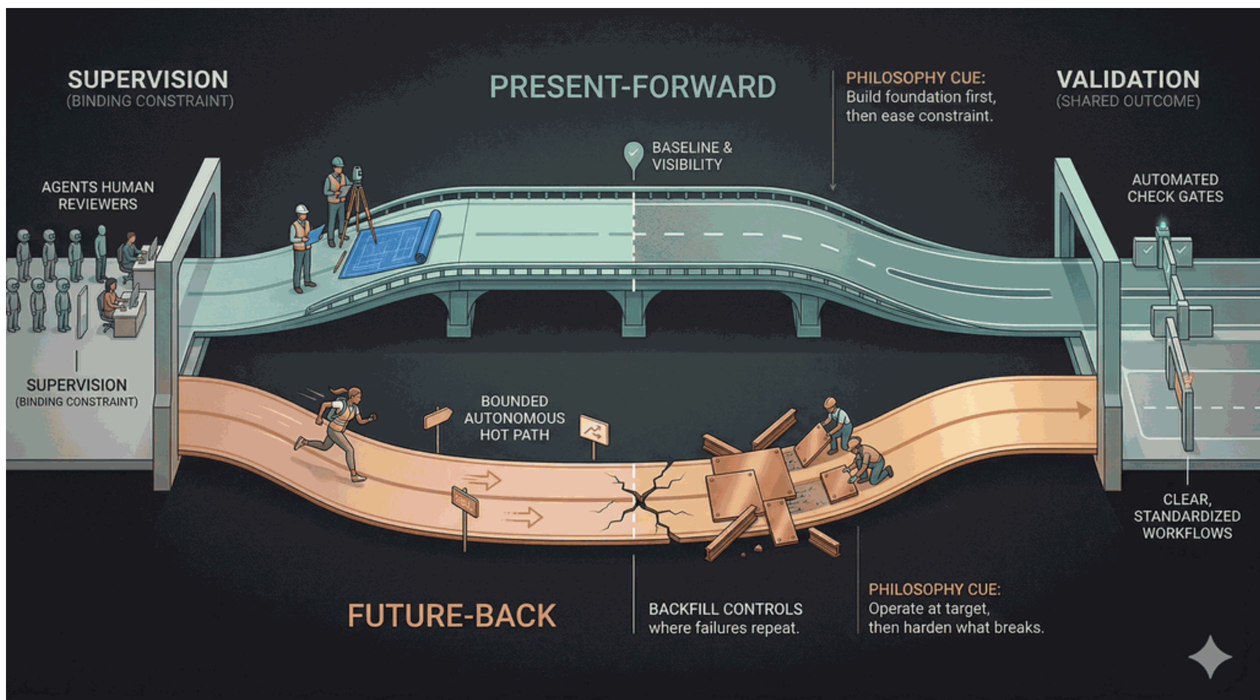

Figure 26: Stage 1 — Observed and Understood

The first stage addresses the most visible constraint in early adoption: every agent output queues behind a human reviewer. Elevate the Next Constraint says this supervision overhead is the binding bottleneck and no other improvement will yield system-level gain until it is addressed. The goal is not to remove supervision immediately; it is to make the constraint visible, establish baselines, and create the conditions for supervision to narrow safely.

Primary goal: establish real usage, visible baselines, and a credible next investment.

Present-Forward Approach

Typical planning range: 3-6 months. Build the foundations before widening agent use.

Focus areas:

- tool rollout and adoption in a pilot area
- basic prompting, validation, and failure-mode learning
- baseline measurement of PR flow, ticket flow, and volatility
- low-risk experimentation with explicit quality checks

Use these signals first: code cycle time or lead time for change; delivery time or cycle time; team volatility; adoption and quality companion signals.

Present-forward benchmarks — Baseline and visibility

Pillar	Metric	Target	Why
Speed	Code cycle time stage breakdown	`>=90%` of pilot PRs instrumented for 4 weeks	Reveals where work actually waits so you can act on evidence, not intuition
Speed	Lead time for change p50/p90	`>=90%` measurement coverage for 4 weeks	Captures both typical and tail performance before you try to improve either
Speed	Flow efficiency	Baseline established; `<20%` treated as wait-heavy	Shows how much of the cycle is active work versus queuing
Predictability	Team volatility	`8-12` weeks baseline; no unexplained `>20%` spike	Unstable teams mask every other metric; baseline stability first

You are ready to narrow routine supervision when the baseline is visible and stable enough to tell good changes from noise. Until then, reducing review is guesswork.

Present-forward benchmarks — Controlled reduction of routine supervision

Pillar	Metric	Target	Why
Speed	Review-stage wait	`>=30%` down from baseline or `p75 <=24h`	Present-forward starts by lowering the review bottleneck on a safe lane
Speed	Review effort	Median review time `<=14h`	Reviewer effort should fall on repeated low-risk work
Quality	Rework rate	`<5%`	Confirms narrower supervision is not creating hidden rework
Quality	Quality incident trend	No increase vs baseline; severe incidents `=0`	The safety net: quality must not degrade as you loosen review

Future-back approach

Start from the target operating model and build infrastructure when the live constraint exposes a gap. The stages still apply, but the sequence is shaped by what the end state demands.

Common early triggers:

Trigger signal	Likely constraint	Typical response
Agents produce inconsistent output almost immediately	weak project context	commit basic architecture notes, coding standards, and task templates to Git

Trigger signal	Likely constraint	Typical response
Quality is unpredictable and failures are repetitive	missing guard rails	add linting, test gates, and lightweight security checks

Future-back benchmarks — Bounded autonomous hot path

Pillar	Metric	Target	Why
Speed	Hot-path lead time	Median <1 day	Future-back proves the model on a narrow, high-frequency, low-risk workflow
Speed	Review-stage wait	Median pickup <=4h	Review delay on the hot path should trend down after initial tuning
Quality	Rework/reopen rate	<5%	Bounds rework so you know autonomy is not creating hidden cost
Quality	Rollback frequency	<5%	Rollbacks stay contained as autonomy is widened

You are ready to backfill controls when failure patterns are understood well enough to codify. Until then, you are still learning what breaks.

Future-back benchmarks — Backfill controls where failures repeat

Pillar	Metric	Target	Why
Quality	Failure-mode recurrence	Same failure class repeats <=1 time/month	Repeated failure classes should be explicitly named and tracked
Durability	Context integrity incidents	<2 incidents/month on the hot path	Context gaps tied to repeat failures should trend down
Durability	Guard-rail coverage	>=80% of repeated failure classes covered	Codifies repeated failure classes into reusable controls

Stage 1 Checkpoints

- ☐ AI use is real in a pilot area: adoption is frequent, usage is intensifying, and teams are building trust in tool output.
- ☐ Baselines exist for code cycle time / lead time for change, delivery time / cycle time, and team volatility.
- ☐ Teams can name the top three recurring failure modes and the top three flow bottlenecks.
- ☐ Low-risk work shows early improvement without a quality regression.
- ☐ The organisation knows whether the next big investment belongs in context, workflow, guard rails, or training.

Planning checkpoint: move to Stage 2 when the organisation can see where work waits, has documented the dominant failure modes, and has evidence that better context or guard rails would unlock the next gain.

Stage 2: Engineered Validation

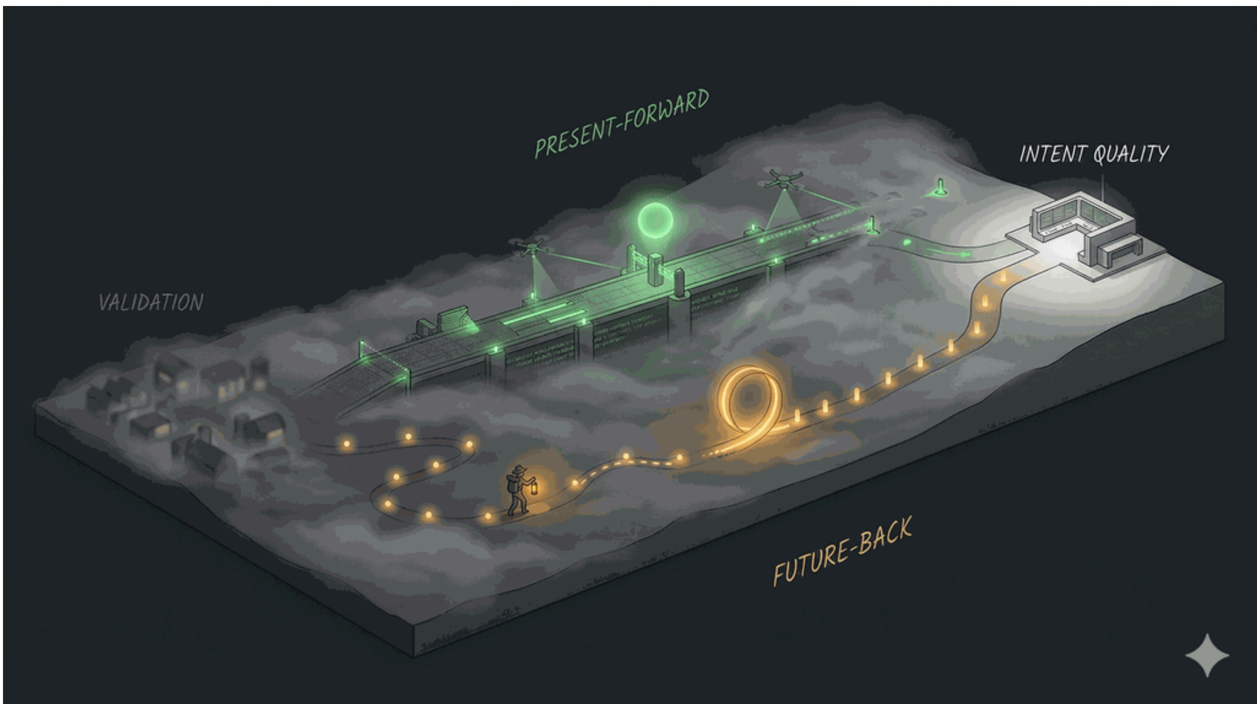

Figure 27: Stage 2 — Engineered Validation

Once supervision narrows, the constraint migrates to validation: are the outputs correct, compliant, and consistent? Everything as Code is the principle that resolves this: codify standards, guard rails, and context so that validation becomes automated and repeatable rather than dependent on individual reviewer judgement. The goal is to engineer the environment so agents succeed more often and human review narrows towards intent, risk and architecture.

Primary goal: make agent success more repeatable by engineering the environment around the work.

Present-Forward Approach

Typical planning range: 6-12 months. Build shared context and verification infrastructure before widening autonomy.

Focus areas:

- shared context, versioned standards, and specification discipline
- agent-to-agent review and automated verification
- guard rails that turn reviewer judgement into reliable controls
- operational visibility into where validation still creates queues

Use these signals first: code cycle time stage breakdowns; flow efficiency; team volatility; context integrity, autonomy horizon, and rework rate.

Present-forward benchmarks — Shared context and engineered checks

Pillar	Metric	Target	Why
Speed	Review-stage wait	`30–50%` below baseline	Present-forward engineers the

Pillar	Metric	Target	Why
			environment so review delay falls structurally
Speed	Flow efficiency	Sustained at 25–35% on pilot lanes	Shows waiting states are shrinking, not just active work speeding up
Predictability	Team volatility	No team in high volatility for 2 consecutive reviews	Local improvements should not create new instability
Durability	Context integrity	>=90% of critical artefacts reviewed every 30 days	Context drift is no longer a weekly operational fire

You are ready to narrow human review to intent and risk when shared context is strong enough for humans to review intent, not detail. Until then, humans are still compensating for weak context.

Present-forward benchmarks — Narrow human review to intent and risk

Pillar	Metric	Target	Why
Focus	Human review scope mix	>=50% of review comments on intent/risk/architecture	Human review should concentrate on intent, architecture, and exceptions
Durability	Autonomy horizon	>=3 consecutive low-risk workflow steps unaided	Reliable multi-step work becomes normal on suitable tasks
Quality	Rework from misunderstood intent	<3%	Rework from unclear intent should trend down

Future-back approach

Common mid-stage triggers:

Trigger signal	Likely constraint	Typical response
Review queues become the new bottleneck	workflow and verification	introduce agent review, tighten PR scope, improve specification templates
Waiting time dominates total flow	approvals and release workflow	use flow efficiency and code-cycle stages to automate or redesign waiting states
Teams diverge wildly in performance	inconsistent local practice	standardise context, rules, and operating conventions before scaling further

Future-back benchmarks — Fast intent loops on active lanes

Pillar	Metric	Target	Why
Quality	First-attempt success	>=70% on comparable low-risk tasks	Future-back proves intent quality on active lanes before standardising
Speed	Intent-clarification cycle time	Median <1 business day	Clarification loops should shorten on active lanes
Quality	Rework by cause	<5% attributable to ambiguous intent	Rework from ambiguous intent should fall across repeated patterns

You are ready to converge local practices when local intent quality is reliable enough to standardise across teams. Stay here and keep tightening intent loops. The risk at this point is premature standardisation of practices that have not yet proved themselves locally.

Future-back benchmarks — Converge local practices into shared model

Pillar	Metric	Target	Why
Predictability	Cross-team volatility	Inter-team spread <=20% for 2 review cycles	Cross-team volatility should narrow across participating teams
Durability	Guard-rail adoption consistency	>=80% of comparable work uses standard guard rails	Guard rails should be applied consistently across comparable work
Durability	Context layer ownership coverage	100% of shared context layers have named owner and cadence	Ownership should be explicit across the shared context layers

Stage 2 Checkpoints

- [ ] Agent contribution is visible and growing: % AI authored and AI assisted PRs is tracked, and the ratio is shifting toward autonomous work on suitable tasks.
- [ ] Review-stage delay in code cycle time materially falls from the Stage 1 baseline.
- [ ] Flow efficiency improves because waiting states, manual approvals, or back-and-forth rescue work have been reduced.
- [ ] Key teams move toward medium or low volatility, and review queues stop dominating the system.
- [ ] Human review time narrows towards intent, architecture, and exceptions rather than routine correctness checking.
- [ ] Autonomy horizon extends to multi-step tasks, and context integrity stabilises.

Planning checkpoint: move to Stage 3 when validation, not supervision, is the primary bottleneck; low-risk review queues are shrinking; and the organisation can trust shared context and agent review enough to widen the autonomy boundary safely.

Stage 3: Risk-Governed Orchestration

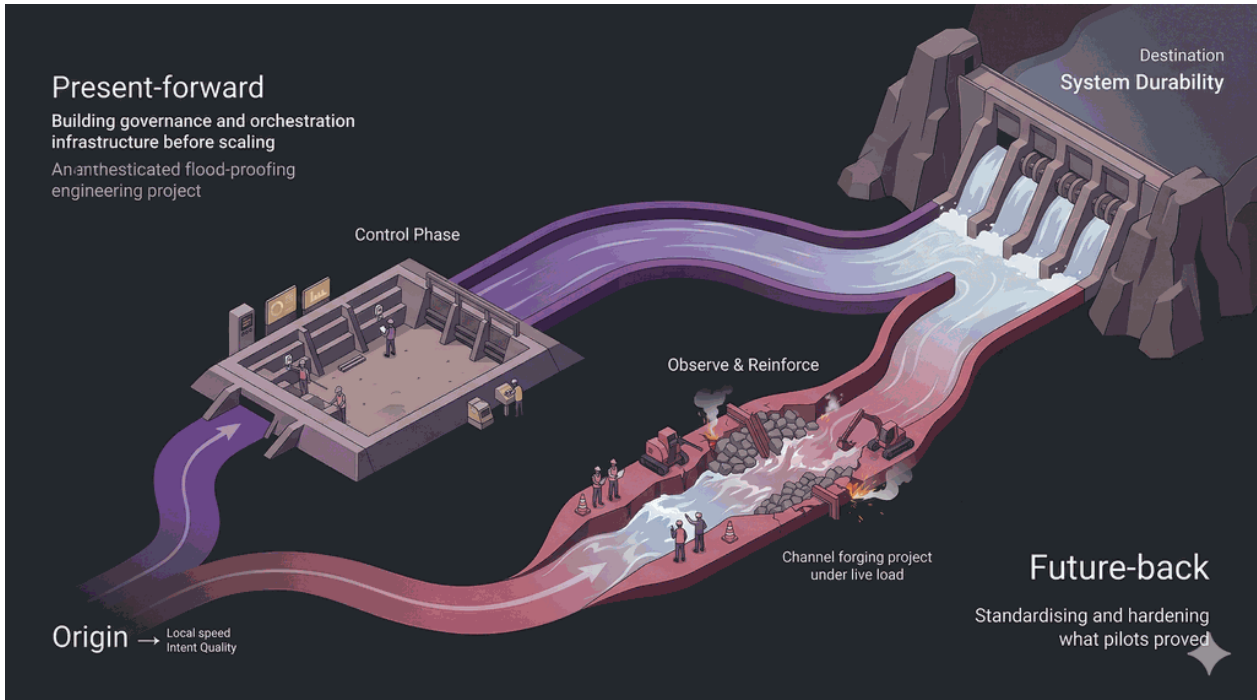

Figure 28: Stage 3 — Risk-Governed Orchestration

When validation is dependable, the constraint moves upstream to intent quality: are we specifying the right work clearly enough for autonomous execution? Humans Above The Loop is the principle at work: the human role shifts from checking outputs to governing the verification systems and shaping the intent that drives them. The goal is to let suitable work move through orchestrated, risk-governed paths whilst humans concentrate on the decisions that require judgement.

Primary goal: let suitable work move through orchestrated, risk-governed paths with humans above the loop rather than inside every step.

Present-Forward Approach

Typical planning range: 12-18 months+. Build risk classification and orchestration infrastructure before widening autonomy.

Focus areas:

- orchestration patterns and dependency handling
- risk classification and oversight paths
- agent evals and operational comparison of agent changes
- discovery, intent quality, and system-level learning

Use these signals first: lead time for change for low-risk work; cross-team volatility; autonomy horizon; hypothesis velocity; innovation effectiveness.

Present-forward benchmarks — Risk-routed orchestration

Pillar	Metric	Target	Why
Speed	Low-risk lead time for change	Median <=1 calendar day	Present-forward builds infrastructure so low-risk work moves fast by default

Pillar	Metric	Target	Why
Speed	Review share of code cycle	<30% of total elapsed time	Review should no longer be the dominant share on low-risk lanes
Quality	Risk classification accuracy	>=90%	Risk classes should reliably predict the correct oversight path
Durability	Autonomy horizon	>=1 working day or >=5 coordinated steps unaided	Autonomy widens without routine human interruption

You are ready to sustain and harden when low-risk flow is fast and governed enough to scale. Until then, widening adoption will amplify whatever is still fragile.

Present-forward benchmarks — Sustain and harden

Pillar	Metric	Target	Why
Predictability	Cross-team volatility	Within +/-15% of pilot baseline across teams	Volatility should not re-widen as adoption spreads
Quality	Rework	<5%	Rework stays low while throughput gains are sustained
Durability	Context-drift incidents	<=1 critical incident/ quarter	Context drift remains controlled as the system hardens
Durability	Hypothesis velocity	>=3 decision-grade experiments/month	Learning velocity remains visible in planning and review

Future-back approach

Common late-stage triggers:

Trigger signal	Likely constraint	Typical response
Multi-agent conflicts or orchestration drift appear	tooling plus coordination	add shared state, stronger decomposition rules, agent evals, and clearer oversight paths

Operating rules for future-back at this stage: instrument metrics from the beginning, even if you are not yet gating on them. Timebox firefighting; if the same constraint returns twice, codify the fix. Treat lightweight design decisions and specifications as mandatory, not optional. Protect high-risk work even if the rest of the path is experimental.

Future-back benchmarks — Standardise after rapid exploration

Pillar	Metric	Target	Why
Predictability	Cross-team volatility	Pilot-to-scale spread narrows by >=25%	Performance should converge beyond the original pilot

Pillar	Metric	Target	Why
Speed	Low-risk lead time distribution across teams	>=75% of teams at <=1 day median	Low-risk flow should become comparably fast across teams
Quality	Policy exception rate	<10% of low-risk items require exception	Exception rates should drop as shared standards spread

You are ready to scale orchestration when pilot performance is durable enough to become the operating model. Stay here and keep standardising. The risk at this point is scaling practices that only work under pilot conditions.

Future-back benchmarks — Durable orchestration at scale

Pillar	Metric	Target	Why
Quality	Rework on low-risk lanes	<3%	Throughput stays high without rework creeping up
Quality	Risk misclassification incidents	<2%	Misclassification stays low as orchestration scales
Quality	Auditability/compliance pass rate	>=95%	Governance outcomes remain stable under higher throughput
Durability	Hypothesis velocity	No drop >10% while throughput rises	Learning should not collapse into delivery-only optimisation

Stage 3 Checkpoints

- ☐ Low-risk PR flow becomes same-day by default, and lead time for change continues to fall for the kinds of work that should be automated.
- ☐ Cross-team volatility narrows, showing the operating model is becoming more consistent across teams rather than only in one flagship group.
- ☐ Risk-based oversight is trusted and accepted; low-risk work flows faster without increasing high-risk exposure.
- ☐ Engineering impact is balanced across focus, speed, predictability, and quality; gains in one area are not at the expense of another.
- ☐ Hypothesis velocity rises, and intent quality becomes a more visible constraint than review throughput.
- ☐ Orchestration reliability, not raw agent activity, becomes the main operational concern.

Planning checkpoint: stay in Stage 3 while the main live constraint is moving upstream into intent, discovery, and learning rather than collapsing back into review, release, or basic governance.

Use posture-specific dashboards in mixed estates. Do not average present-forward and future-back lanes into one maturity score, or you will hide the live constraint.

Dependencies Between Stages

Do not treat the stages as maturity theatre. They depend on each other.

- **Stage 1 to Stage 2** usually requires dependable Git workflows, CI/CD, baseline measurements, and enough team comfort with agent failure modes to codify better controls.
- **Stage 2 to Stage 3** usually requires shared context that agents can trust, guard rails that have proved they work, and a real risk-classification model rather than vague confidence.

The critical path is usually context plus workflow, not model capability. Faster models do not remove the need to fix the system around them.

The Operating Loop

The Operating loop is Elevate the Next Constraint turned into a repeating practice. The metrics reveal where the system is struggling; the Operating Loop converts that signal into a structured intervention. It's the same loop irrespective of maturity stage or implementation path.

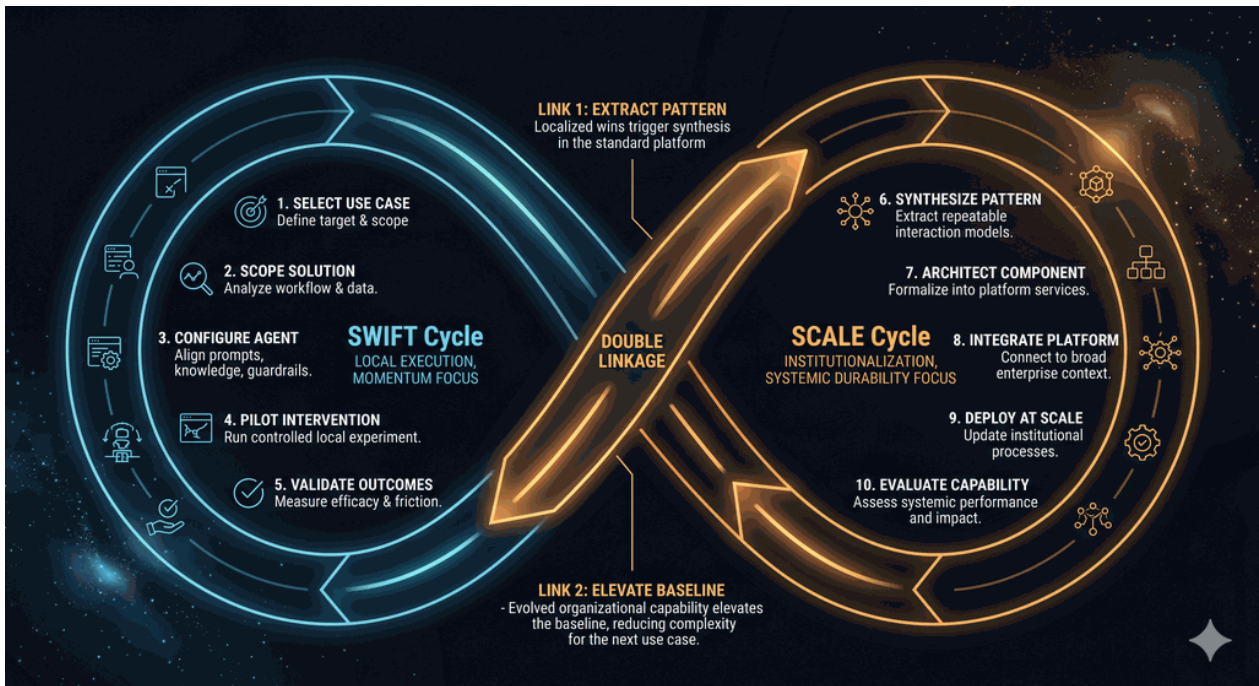

Figure 29: The Operating Loop

Loop A: The SWIFT Cycle (Localised Execution)

This loop focuses localised capability: zoom in on a use case, eleveate then fix the binding constraints, until it's time to move on.

1. Select Use Case: Isolate a specific, bounded workflow or component. Avoid scope creep; do not attempt to optimise the entire system simultaneously.
2. Watch Metrics: Establish baseline telemetry. You cannot improve what you cannot accurately measure; ensure quantitative data exists before planning any intervention.
3. Isolate Constraint: Identify the next single most significant bottleneck currently limiting the Watch Metrics within the selected Use Case.
4. Fix Constraint: Deploy a targeted engineering or process intervention to remove the binding constraint. Or elevate it; improve its metric until it's no longer the binding constraint.
5. Terminate Loop: When the cost of further intervention exceeds the engineering value gained it's time to move on to the next highest-priority Use Case.

Loop B: The SCALE Cycle (Organisation Durabiliyt)

This loop focuses on systemic capability. This is how the organisation gains velocity without paying the innovation tax.

6. Synthesise Collateral: Extract the successful SWIFT intervention into a generalised, reusable pattern, architectural standard, or platform tool. Make it easier to use the tool than not.
7. Communicate Learnings: Actively disseminate the synthesised collateral across the organisation. Choose from the full range of business change tools, from documentation wikis to internal marketplaces to scores in end-of-year appraisals.
8. Assess Reuse: Monitor organisational telemetry to verify that other teams are actually adopting the collateral and achieving similar productivity gains.
9. Let Go of Legacy: Formally deprecate and sunset the outdated processes, tools, or cognitive load that the new capability replaces. Actively manage process debt.
10. Evolve Capability: The baseline for the next SWIFT cycle is now permanently elevated.

Real Signals

Each row maps an observable signal to the most likely binding constraint and a typical next move. When the Operating Loop surfaces a documentation or context constraint, Everything as Code tells you the response is codification, not more wiki pages. When it surfaces a review bottleneck, Humans Above The Loop tells you the response is better verification systems, not faster human review. The point is not to follow this table mechanically but to use it as a starting frame for conversation.

Signal	Metric to check	Most likely live constraint	Likely next move
Long wait from PR opened to first review	code cycle time stage breakdown	review capacity, PR size, or weak agent review	tighten PR scope, strengthen agent review, route low-risk work differently
Long wait from approved to merged or deployed	code cycle time stage breakdown, lead time for change	merge queue, release process, or deployment friction	automate release steps, simplify approvals, improve deployment paths
Total cycle time flat but active work is small	flow efficiency	hidden waiting, handoffs, or approvals	redesign statuses, reduce manual gates, collapse unnecessary queues
One team erratic, others stable	team volatility	local team workflow, staffing, or context discipline	intervene locally, not organisation-wide
Multiple teams erratic	cross-team volatility	uneven operating model across the organisation	standardise context, rules, and stage expectations
Flow is better but learning is weak	companion metrics such as hypothesis velocity	discovery constraint	invest in faster experimentation and better intent framing
Autonomy increases but reversions and uncertainty rise	rework rate, context integrity, review quality	trust/comprehension constraint	slow down, improve context, tighten risk boundaries

Constraint Categories

Keep the constraint taxonomy simple enough to act on. When the Operating Loop surfaces a bottleneck, classify it into one of these categories so the response targets the right part of the system.

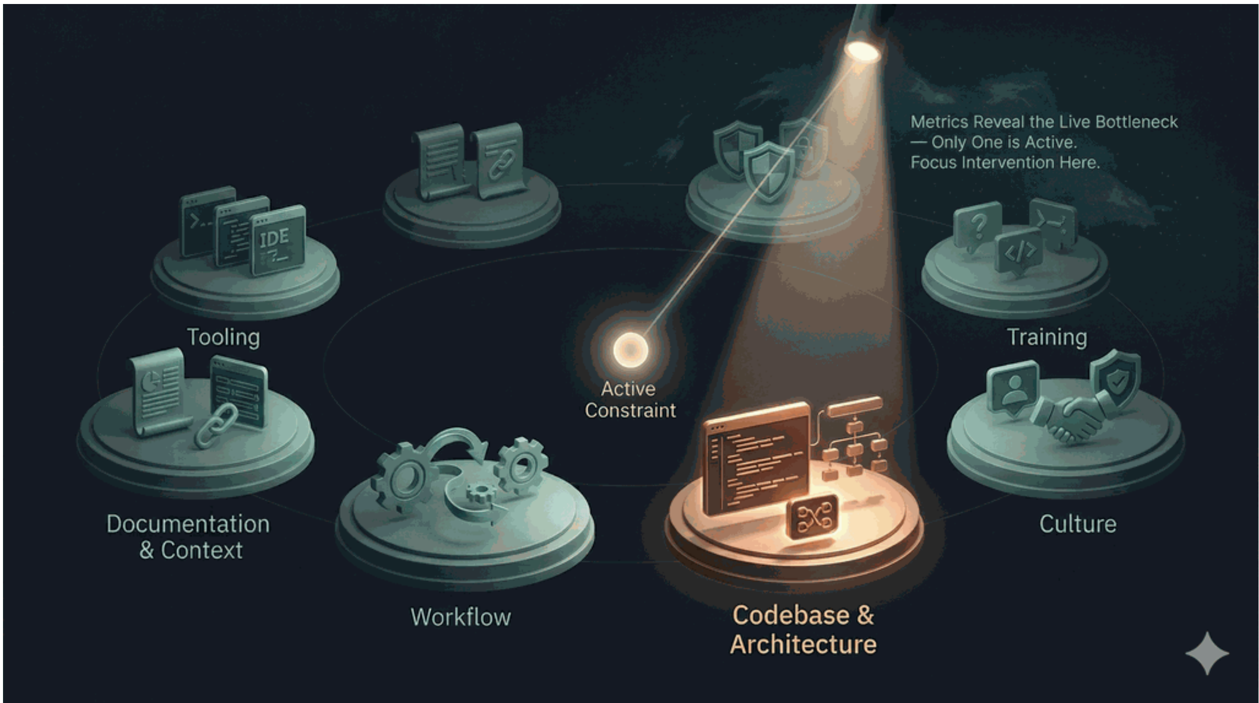

Figure 30: Constraint Categories

Do not try to improve all seven at once. Use the metrics to identify which category is actually binding now.

Building the New Capabilities

The stages describe how the operating model matures; this section describes the specific capabilities that the operating model depends on. Every major technology shift creates a new operational discipline and extends two traditions in parallel. Cloud computing introduced DevOps for the engineering pipeline and extended ITIL service management for operational governance. Containers introduced platform engineering. Each shift did not speed up the previous function; it introduced fundamentally new concerns that required new disciplines.

AI agents introduce the next set: non-deterministic behaviour, autonomous multi-step reasoning, tool use with real-world consequences, and knowledge that decays between sessions. These extend both traditions. **AgentOps** extends the engineering tradition: the practices that make agents buildable, testable and deployable, governed by Everything as Code. **Agentic Service Management (AGSM)** extends the service management tradition: the practices that make agents governable, trustworthy and compliant, governed by Humans Above The Loop. AgentOps engineers the conditions under which agents can act well. AGSM engineers the conditions under which humans can defend the decisions those agents made.

Which discipline to invest in depends on where the constraint currently sits. If agents are failing because context is poor, evals are missing, or failure modes are untracked, the constraint is in AgentOps. If agents are succeeding but the organisation cannot defend what changed, cannot govern agent changes, or cannot scale trust beyond individual reviewers, the constraint is in AGSM. The subsections below describe what to build at each stage, what to measure, and what to do when it is not working.

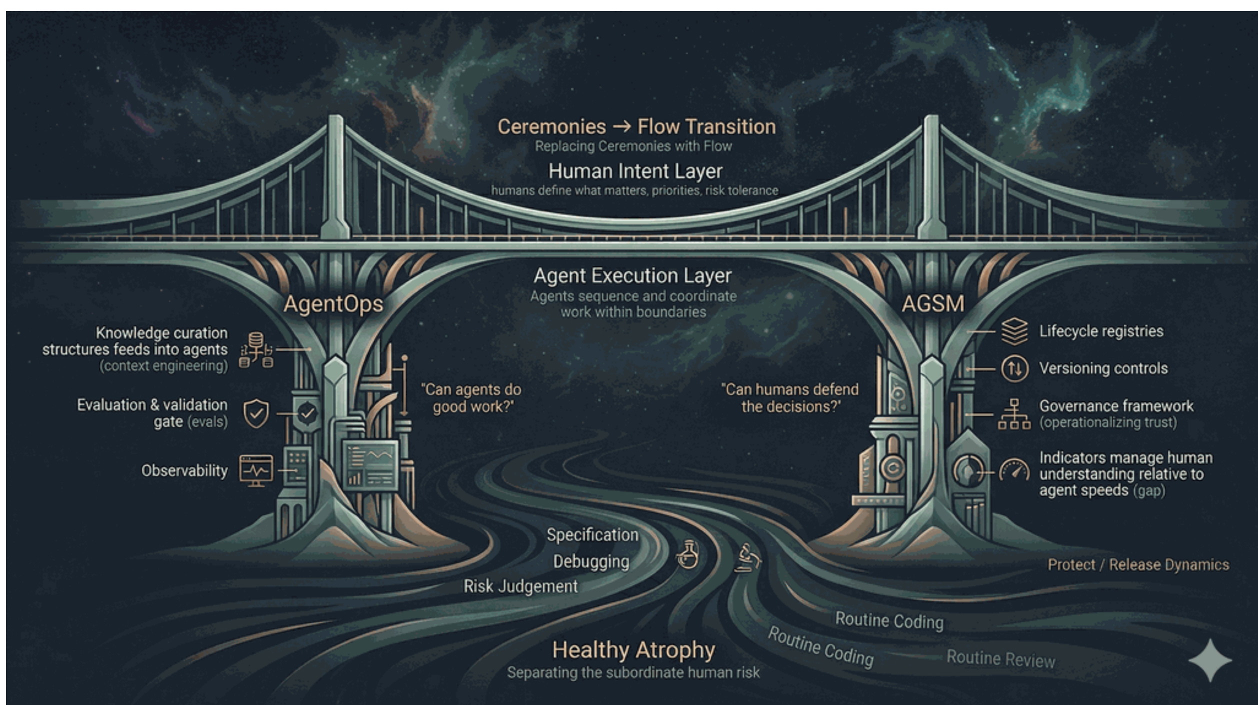

Figure 31: Building the New Capabilities

Replacing Ceremonies with Flow

Traditional agile ceremonies assumed planning and execution lived in the same heads. When agents execute the development work, that assumption breaks. The replacement separates an **intent layer** (humans define what matters, in what order, with what risk tolerance) from an **execution layer** (agents sequence and coordinate work within those boundaries).

What to change at each stage:

- **Stage 1**: Lighten ceremonies. Use stand-ups to learn about agent failure modes and calibrate how much context agents need to self-organise.
- **Stage 2**: Make the separation explicit. Humans operate at the intent layer: prioritisation, risk decisions, outcome evaluation. Agents operate at the execution layer, orchestrated by context rather than ceremony. Reserve synchronous time for intent alignment and boundary decisions.
- **Stage 3**: Decision forums, constraint reviews and strategic pivots replace the full ceremony stack. The intent layer runs on a Operating Loop cadence, not a sprint-driven one.

Benchmarks

Step	Constraint	Metric	Target value	Readiness signal
Lighten ceremonies (Stage 1)	Ceremonies consume time without matching agent rhythm	Time in ceremony	>=25% reduction from baseline	Teams report ceremonies feel useful, not performative
Separate intent and execution (Stage 2)	Planning conflates strategy with implementation logistics	Intent-to-execution latency	Median <1 business day	Humans define intent; agents self-organise execution from context
Operating Loop cadence (Stage 3)	Fixed cadence does not match flow speed	Hypothesis velocity	>=3 decision-grade experiments/ month	Constraint reviews replace retrospectives; decisions are timely
		Cycle time	No increase vs Stage 2 baseline	Speed holds or improves as ceremonies thin

AgentOps

AgentOps makes agents buildable, testable and deployable. It has three pillars: context engineering (curating and governing the knowledge that shapes agent behaviour), evals and validation gates (comparing agent versions and ensuring no agent grades its own work), and observability and failure modes (tracing reasoning, detecting metacognitive failures such as premature abandonment, context flooding, repeating failed actions and poor verification).

What to build at each stage:

- **Stage 1**: Log agent interactions and capture failure patterns. Start building shared project context in the repository. The goal is observability and learning, not automation.
- **Stage 2**: Version agent definitions. Run evals routinely: replay representative tasks, compare versions, track regressions. Assign a named owner and freshness cadence to each context layer. Enforce validation gates so no agent grades its own work. First-attempt success rate becomes a key signal.
- **Stage 3**: Run agent-to-agent evals and orchestration health monitoring as standard practice. Context becomes infrastructure for orchestration: multiple agents work from the same governed picture. Rollback and version comparison are routine when behaviour degrades.

Benchmarks

Step	Constraint	Metric	Target value	Readiness signal
Failure-mode logging (Stage 1)	Failures feel random, no shared understanding	Failure-mode catalogue	Top 5 failure classes named and tracked	Team can explain how agents fail, not just that they fail
Context layer ownership (Stage 2)	Context drift causes avoidable rework	Context integrity	>=90% of critical artefacts reviewed every 30 days	Context gaps trace to a specific layer, not random noise
Eval coverage (Stage 2)	No way to compare agent versions	Eval pass rate	>=80% of task types covered by evals	New agent versions can be compared before deployment
Failure-mode codification (Stage 2)	Same failures recur across sessions	Failure-mode recurrence	Same class repeats <=1 time/ month	Repeated failures convert to reusable controls
First-attempt success (Stage 2-3)	Rework from poor context or weak specifications	First-attempt success	>=70% on comparable low-risk tasks	First-attempt success rises across comparable tasks
Orchestration health monitoring (Stage 3)	Degraded behaviour not caught quickly	Recovery time	Recovery from degradation in <4 hours	Regressions trace to a specific context layer or agent version

Agentic Service Management (AGSM)

AGSM makes agents governable, trustworthy and compliant. Where AgentOps asks "can agents do good work?", AGSM asks "can humans defend the decisions those agents made?" It has four pillars: agent lifecycle management (versioned definitions, registries tracking identity, ownership, version and status), operationalising trust (governance operates on the framework – risk classification rules, eval thresholds, rollback triggers – not on each individual change), managing the comprehension gap (the risk that agent throughput outpaces the organisation's ability to understand and defend what changed), and agentic supply chain (trust must be portable across organisational boundaries; the response is defence in depth).

What to build at each stage:

- **Stage 1**: Start the agent registry and capture governance decisions. Trust is personal: reviewers understand what they review because volume is low enough. The goal is awareness and habit formation.
- **Stage 2**: Register and version agent definitions. Route agent changes through lightweight governance. Flow knowledge from previous sessions into new ones. Acknowledge and actively manage the comprehension gap through reviewer tooling (summaries, risk annotations, intent traceability).
- **Stage 3**: AGSM becomes an operating capability. Agent registries, risk classification, control mappings, approval records, action traces and validation results form the evidence base. Compliance evidence is auditable by design. Oversight is applied by risk surface, not uniformly.

Benchmarks

Step	Constraint	Metric	Target value	Readiness signal
Agent registry (Stage 1-2)	Unknown agent inventory	Registry completeness	100% of production agents registered	All agents have identity, ownership, version, status
Governance framework (Stage 2)	Per-change approval bottleneck	Risk classification accuracy	>=90%	Risk labels reliably predict the correct oversight path
Comprehension-gap management (Stage 2-3)	Reviewer overload as throughput rises	Review depth indicators	No worsening trend vs baseline	Review quality holds as throughput rises
Knowledge flywheel (Stage 2-3)	Same learning cost paid repeatedly	Failure-mode recurrence	Same class repeats <=1 time/ month	Lessons from previous sessions flow into new ones
Auditable compliance (Stage 3)	Cannot defend compliance position at scale	Auditability/ compliance pass rate	>=95%	Evidence chain is auditable without reconstruction
Comprehension-gap monitoring (Stage 3)	Throughput outpacing understanding	Comprehension-gap indicators	Very large PRs, shallow reviews, reversions: no worsening	Comprehension-gap signals are tracked and stable

Healthy Atrophy and Dangerous Erosion

The practical question is which human capabilities to protect and which to allow to atrophy as agents take on more work. Choose interaction patterns that match the risk profile: closer engagement and deliberate practice on high-risk work, broader delegation on low-risk work.

What to protect at each stage:

- **Stage 1**: All skills are still exercised. Focus on building new skills: effective prompting, validation, failure recognition.
- **Stage 2**: Routine coding and review skills start to atrophy. Specification writing, context engineering, debugging, and guard rail design must deepen.
- **Stage 3**: Intent articulation, risk classification, system-level judgement and orchestration governance become critical. Invest deliberately in those.

Measure: review quality (should hold or improve even as review volume falls), specification precision (should improve over time), ability to debug complex agent failures, whether atrophy is happening in consciously chosen areas.

Dependencies Between Capabilities

The four capabilities are not independent investments. Context engineering is the foundation: evals, trust, and comprehension-gap management all depend on governed context. Evals enable trust: you cannot operationalise trust without a way to compare agent versions and detect regressions. Trust enables wider autonomy, which is when the comprehension gap becomes a real constraint. Ceremony replacement depends on all three: the intent/execution separation only works when context is strong enough for agents to self-organise.

The practical implication: invest in context engineering first, evals second, governance third. Attempting AGSM without AgentOps foundations produces governance theatre.

Capability Warning Signs

When the Operating Loop surfaces a constraint in one of these capabilities, use this table to find the most likely cause and a first response.

Warning sign	Metric to check	Likely issue	Response
Context investment rises but first-attempt success does not	context integrity, first-attempt success	wrong layers being improved, or task specifications still weak	trace failures to a specific layer; fix the weakest
Evals pass but production quality drops	eval coverage, rework rate	eval tasks do not represent real work	refresh eval suite from recent production failures
Agent cost keeps rising	cost per task, context size, model mix	context bloat or wrong model routing	prune context, route simple work to cheaper models, set budgets
Comprehension gap widens	review depth, reversions, uncertainty signals	throughput outpacing reviewer tooling	narrow autonomy scope temporarily; invest in traceability and summaries
Governance slows delivery without improving outcomes	lead time for change, risk classification accuracy	controls are too uniform or misclassified	apply risk-based routing; thin controls on low-risk lanes
Same failure class recurs despite codified fix	failure-mode recurrence, context integrity	fix is not reaching the agent context at the right time	check context loading order and layer freshness
Developer resistance grows despite good metrics	survey signals, participation, review feedback	change saturation or skill-erosion anxiety	slow the rollout; invest in skill-building for Stage 2-3 capabilities
Ceremonies thin but cycle time worsens	cycle time, intent-to-execution latency	intent quality dropped when planning ceremony removed	restore intent-setting forum; separate intent from execution more clearly

Conclusion: The Next Constraint

This Roadmap turns the Vision and Journey into planning decisions and operational actions. Three principles resolve the trade-offs: Elevate the Next Constraint tells you what to do next; Everything as Code tells you how to make it stick; Humans Above The Loop tells you where people belong in the system.

The practical path is: calibrate the posture to the organisation's appetite and capacity; walk the stages, using the Operating Loop to find and fix the binding constraint at each one; build the new capabilities (AgentOps for the engineering discipline, AGSM for the governance and trust infrastructure, a post-ceremony operating cadence) as the system matures; and keep recalibrating as constraints migrate and the environment shifts.

The organisations that move best through this journey will not be the ones that generate the most plans. They will be the ones that learn fastest where work is waiting, fix the next bottleneck before the previous fix wears off, and build the engineering, governance and human capabilities that make the operating model durable as autonomy rises.

So what do you next? What's your next most pressing constraint?

Appendices

Appendix A: Metrics Reference

This section defines the metric concepts used throughout the Roadmap, organised by the five pillars introduced in the stage overview. The first four pillars are **lag indicators** (they tell you what happened); the fifth predicts what is coming. Productivity metrics can look healthy while durability metrics quietly deteriorate; measure both.

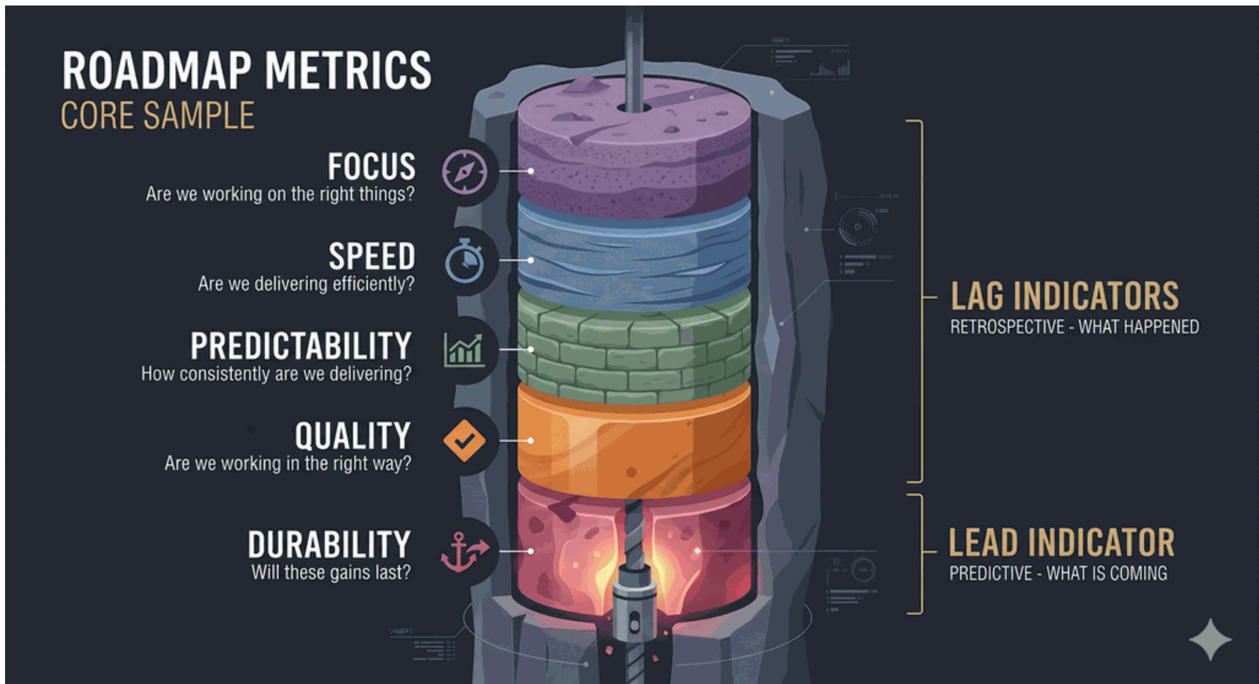

Figure 32: Practical Metric Stack

Focus Metrics

Are we working on the right things?

Metric	Use it for	Notes
Human review scope mix	Review quality	Whether review comments concentrate on intent, architecture, and exceptions rather than routine correctness.
Feature utilisation	Value after delivery	Whether shipped capability is adopted; product analytics, usage studies, or qualitative feedback. Qualitative-heavy.
Intent-to-execution latency	Decision responsiveness	Median elapsed time from a human prioritisation decision to agent action; ceremony replacement signal.
Time in ceremony	Overhead tracking	Hours per person per sprint spent in synchronous ceremonies; tracks whether ceremony lightening is real.

Speed Metrics

Are we delivering efficiently?

Metric	Use it for	Notes
Code cycle time	PR lifecycle flow	Best when you need to see where PRs wait: first review, approval, merge, deployment.
Lead time for change	DORA-compatible speed reporting	Same underlying measure as code cycle time; use this label for DORA or executive reporting.
Delivery time / cycle time	Ticket flow	Useful when the bottleneck sits in the wider ticket workflow, not just in PRs.
Flow efficiency	Active vs waiting time	Essential for proving whether a "speed" problem is really a waiting problem.

Predictability Metrics

How consistently are we delivering?

Metric	Use it for	Notes
Team volatility	Team-level predictability	Use to see if one team's completion pattern is unstable.
Cross-team volatility	Organisation-level consistency	Use to see whether the operating model is uneven across teams.

Quality Metrics

Are we working in the right way?

Metric	Use it for	Notes
Rework rate	Churn after "done"	Reopens, defect bounce-backs, or follow-on fixes; links flow to quality and sustainability.
Risk classification accuracy	Governance fit	Whether risk tiers match reality; retrospective comparison of incidents and classifications.
Eval pass rate	Agent version confidence	Fraction of eval tasks that pass for a given agent version; tracks whether agents are improving or regressing.
Eval coverage	Verification breadth	Fraction of task types covered by evals; blind spots in eval suites let regressions through.
Cost per task	Economic sustainability	Agent infrastructure cost divided by completed tasks; watch for rising cost without rising quality.

Metric	Use it for	Notes
Comprehension-gap indicators	Understanding vs throughput	Composite of very-large-PR rate, shallow-review-time rate, reversion rate, and reviewer uncertainty signals; tracks whether throughput is outpacing comprehension.

Durability Metrics (Lead Indicators)

Will these gains last?

Metric	Use it for	Notes
Autonomy horizon	Durability of speed gains	How far teams can decide without escalation; surveys, governance audits, or decision-log review.
Hypothesis velocity	Learning cadence	How fast hypotheses move from idea to evidence; often experiment logs, bet reviews, or manual counts per period.
Context integrity	Shared understanding	Whether documentation, runbooks, and agent context stay accurate; freshness checks, onboarding friction, drift signals. Qualitative-heavy.
Innovation effectiveness	Outcomes from exploration	Whether innovation effort produces useful results; portfolio reviews and outcome narratives beyond raw output. Qualitative-heavy.
Developer trust and comprehension signals	Belief and clarity	Trust in tooling, clarity of the system, and comprehension of changes; surveys, interviews, or lightweight pulse prompts. Qualitative-heavy.
Agent registry completeness	Governance readiness	Whether all production agents have identity, ownership, version and status recorded; prerequisite for auditable AGSM.
Recovery time from degraded behaviour	Operational resilience	How quickly the team detects and reverses a behaviour regression; combines observability and rollback capability.

Planning Questions

Planning question	Best-fit pillar	Best-fit metrics	Why these matter
Where is work waiting?	Speed	Code cycle time, lead time for change,	They expose queueing, review friction,

Planning question	Best-fit pillar	Best-fit metrics	Why these matter
		delivery time, cycle time, flow efficiency	approval delay, merge delay, deployment delay, and inactive time.
Are we becoming more predictable?	Predictability	Team volatility, cross-team volatility	They show whether one team is unstable or the operating model is uneven across the organisation.
Are we speaking DORA to executive audiences?	Speed	Lead time for change	It is the same underlying PR lifecycle measure as code cycle time, but with DORA framing.
Are we just producing more activity?	Quality	Commit count, PR creation rate; only when paired with flow or quality measures	Raw activity alone is easy to game and says little about actual progress.
Will today's gains last?	Durability	Hypothesis velocity, autonomy horizon, context integrity, innovation effectiveness	They predict whether improvements are sustainable or whether you are borrowing from the future.

Appendix B: Building in Plandek

If your organisation uses Plandek, the generic metric concepts used throughout this Roadmap map directly to Plandek's metric library. This section provides that mapping so the rest of the document remains tool-agnostic.

Metric Mapping

Pillar	Roadmap concept	Plandek metric	Notes
Focus	Human review scope mix	--	not yet implemented
Focus	Feature utilisation	--	not yet implemented
Focus	Intent-to-execution latency	--	not yet implemented; Section 4 Ceremonies
Focus	Time in ceremony	--	not yet implemented; Section 4 Ceremonies
Speed	Code cycle time	`CODE CYCLE TIME`	Includes stage breakdowns: opened → first review → approved → merged → deployed
Speed	Lead time for change	`LEAD TIME FOR CHANGE`	Same underlying measure as code cycle time with DORA-compatible labelling
Speed	Delivery time	`DELIVERY TIME`	Ticket-level: creation to done
Speed	Cycle time	`CYCLE TIME`	Ticket-level: work started to done
Speed	Flow efficiency	`FLOW EFFICIENCY`	Ratio of active time to total elapsed time
Speed	Review effort	--	not yet implemented
Speed	Intent-clarification cycle time	--	not yet implemented
Predictability	Team volatility	`KANBAN VOLATILITY`	Variability in a single team's completion patterns
Predictability	Cross-team volatility	`AVERAGE TEAM VOLATILITY`	Aggregated volatility across multiple teams
Quality	Rework rate	--	not yet implemented
Quality	Quality incident trend	--	not yet implemented
Quality	Rollback frequency	--	not yet implemented
Quality	First-attempt success	--	not yet implemented
Quality	Rework by cause	--	not yet implemented
Quality	Risk classification accuracy	--	not yet implemented
Quality	Risk misclassification incidents	--	not yet implemented
Quality	Policy exception rate	--	not yet implemented
Quality	Auditability/compliance pass rate	--	not yet implemented

Pillar	Roadmap concept	Plandek metric	Notes
Quality	Eval pass rate	--	not yet implemented; Section 4 AgentOps
Quality	Eval coverage	--	not yet implemented; Section 4 AgentOps
Quality	Cost per task	--	not yet implemented; Section 4 warning signs
Quality	Comprehension-gap indicators	--	not yet implemented; composite of review depth, reversions, uncertainty; Section 4 AGSM
Durability	Autonomy horizon	--	not yet implemented
Durability	Hypothesis velocity	--	not yet implemented
Durability	Context integrity	--	not yet implemented
Durability	Context integrity incidents	--	not yet implemented
Durability	Context-drift incidents	--	not yet implemented
Durability	Guard-rail coverage	--	not yet implemented
Durability	Guard-rail adoption consistency	--	not yet implemented
Durability	Innovation effectiveness	--	not yet implemented
Durability	Developer trust and comprehension signals	--	not yet implemented
Durability	Agent registry completeness	--	not yet implemented; Section 4 AGSM
Durability	Recovery time from degraded behaviour	--	not yet implemented; Section 4 AgentOps
Durability	Failure-mode recurrence	--	not yet implemented
–	Commit count	`COMMITS COUNT`	Activity signal only; not assigned to a pillar
–	PR creation rate	`CREATED PULL REQUESTS`	Activity signal only; not assigned to a pillar
–	PR efficiency ratio	`PR EFFICIENCY QUOTIENT`	Composite dashboard summary
–	Throughput ratio	`THROUGHPUT QUOTIENT`	Composite dashboard summary
–	Merge rate	`MERGE RATE`	Composite dashboard summary
–	Sprint completion	`SPRINT COMPLETION OVERALL`, `SPRINT COMPLETION TARGET`, `SPRINT COMPLETION ADDED`	Family of sprint-based metrics; transition aid only

Plandek Usage Notes

- **Stage breakdowns**: code cycle time stage breakdowns (opened → first review → approved → merged → deployed) are available natively in Plandek dashboards.
- **Volatility thresholds**: Plandek provides low / medium / high volatility classifications that map to the Roadmap's predictability language.
- **DORA reporting**: use lead time for change as the DORA-compatible label when reporting to executive audiences.

Appendix C: Anti-Patterns and Bad Metrics

OKR guidance distinguishes **outcomes** (the change you want) from **outputs** (what you shipped) and from **activities** (what you did). A recurring implementation failure is writing **key results as tasks or busy work**; metrics that can be "met" without improving customer or organisational value. Related traps include **vanity metrics** (numbers that look good but do not support decisions), **scorecard sprawl**, and **single-number optimisation** that distorts behaviour once it becomes the target.

Treating activity as success

Anti-pattern: Incentivising motion instead of progress through the value stream. This is the engineering form of **output- or activity-based key results**: counting events (commits, PRs opened, tickets touched) rather than whether waiting time fell, quality held, or value reached users.

In this work: Commit count and PR creation rate are **activity signals**, not DevEx or flow success metrics. They are easy to inflate and weakly coupled to outcomes; they can rise while cycle time, rework, or comprehension worsen. If you show them at all, pair them with **Speed**, **Predictability**, **Quality** or **Durability** measures from this document so nobody optimises the wrong curve.

Lag-only steering

Anti-pattern: Running the programme from **lagging indicators** alone; measures you see after the fact (e.g. quarterly lead time) with no **leading indicators** tied to inputs you control this week (where work waits, batching, review latency, WIP). Good OKR practice uses leading measures as levers and lagging measures as proof they worked; lag-only reporting encourages late excuses instead of early correction.

In this work: Lead time for change and end-to-end cycle time are lag summaries relative to a useful horizon. **Stage breakdowns** on code cycle time and **flow efficiency** are leading: they show where to act before the lag number moves. Prefer that decomposition when you are migrating **active constraints**; a single duration line on a dashboard rarely tells you which policy or handoff to change.

Ceremony and agile dogma as the scoreboard

Anti-pattern: Letting **process compliance** stand in for outcomes; metrics that reward the plan ritual rather than flow of value. **Sprint completion** and **story-point velocity** are familiar examples: they optimise "did we finish what we forecast?" not whether work moved smoothly, whether predictability improved across teams, or whether the system stays legible for people and agents.

In this work: Sprint-style metrics may be a **transition aid** where the organisation still thinks in sprints; they are **not** the success model for a roadmap centred on **flow** and **constraint migration**. If they dominate executive reporting, you risk under-investing in PR-stage waits, cross-team volatility, and companion signals (context integrity, rework) that determine whether speed lasts.

Composite indices as the main story

Anti-pattern: Letting one rolled-up score (efficiency quotients, blended "productivity" indices) replace explanation. That invites **Goodhart-style** distortion: the index becomes the target, and local gaming hollows out the behaviour you actually wanted. OKR literature also warns against **too many** metrics; a single composite "fixes" sprawl by hiding trade-offs instead of resolving them.

In this work: PR efficiency and **throughput**-style ratios can sit on a dashboard as **summaries**; they are **too composite** to be the primary diagnostic here. When speed and quality trade off, decompose into the flow and predictability metrics in this reference rather than debating one opaque number.

Steering principle: Keep the default narrative anchored on the Practical Metric Stack. For each figure on a scorecard, ask whether it measures an **outcome** you care about or **activity** you can game, and whether every **lag** view you show has at least one **leading** view of where work waits.

Warning Signs

Warning sign	What it usually means	Check next	Typical response
Individual productivity rises but organisational throughput stays flat	the bottleneck moved downstream	code cycle time, delivery time, flow efficiency	identify the new queue; do not celebrate local speed in isolation
PR backlog grows despite faster code generation	review or approval is now the constraint	code cycle time stage breakdown	strengthen agent review, reduce PR size, redesign approval paths
Delivery time improves but active work share stays low	waiting is hidden inside the flow	flow efficiency	remove manual handoffs, reduce waiting states, simplify release flow
One team becomes highly erratic	local practice is unstable	team volatility	intervene locally with context, workflow, or staffing changes
Many teams become erratic at once	the operating model is inconsistent	cross-team volatility	standardise the operating model before scaling further
Human review time does not fall in Stage 2	people are still checking routine detail	review-stage code cycle time, reviewer feedback	improve specifications, agent review, and guard rails; retrain review focus
Context files churn but quality does not improve	the system is patching symptoms	context integrity, rework rate	simplify the context hierarchy; fix ownership and freshness
Experimentation falls while delivery appears healthy	short-term output is crowding out learning	hypothesis velocity, feature utilisation	protect discovery capacity; do not spend all gains on throughput
Agent throughput rises faster than human understanding	trust and compliance risk are growing	review depth, reversions, uncertainty signals	narrow autonomy scope temporarily; improve explainability and traceability

Appendix D: Further Reading

On replacing ceremonies with flow

- Reinertsen, D.G. (2009). The Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas Publishing. Argues that invisible queues are the root cause of poor development performance; provides 175 principles for reducing batch size, limiting WIP and accelerating feedback. The theoretical foundation for why time-boxed coordination ceremonies become a constraint when cycle times compress.
- Anderson, D.J. (2010). Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press. Operationalises flow principles into a practical method: replace sprint boundaries with WIP limits and pull-based scheduling; decouple delivery cadence from governance cadence through seven coordination rhythms (replenishment, delivery planning, service delivery review, risk review, strategy review, operations review, stand-up).
- DORA (2025). State of AI-assisted Software Development. Google Cloud. AI-assisted teams produce 98% more pull requests, but PRs are 154% larger and take 91% longer to review; organisational throughput stays flat. Concludes that AI amplifies existing strengths and weaknesses rather than transforming them. The strongest current empirical evidence that faster code generation moves the constraint to review and coordination.
- Intent Flow (2026). Intent Flow: A Post-Sprint Workflow for Teams That Learn as They Go. https://intentflow.org/. Proposes a learning-loop workflow (intent, work, finding, refined intent) that replaces backlogs, burndowns and assigned work with cycles that end when understanding is sufficient rather than on calendar dates.

On AgentOps as operational discipline

- Fuller, B. (2026). "AgentOps: It's Not Agent Orchestration. It's Context Orchestration." https://www.bodenfuller.com/writing/context-orchestration. Argues from practitioner experience that the high-leverage problem in multi-agent systems is not routing or message passing but what is in the context window when agents start working. Frames AgentOps as context orchestration with a six-step operational cycle (research, plan, pre-mortem, build, validate, post-mortem) and a knowledge flywheel that compounds learnings across sessions.
- 12-Factor AgentOps (2026). https://12factoragentops.com/. Inspired by the Twelve-Factor App, organises AgentOps into four categories: Foundation (context management, version control, scoped tasks), Workflow (research before build, external validation, locking progress forward), Knowledge (extract learnings, compound knowledge, measure fitness not activity), Scale (worker isolation, hierarchical supervision, harvest failures as data).
- Solsta (2026). "BehaviorSpec: A Declarative Contract for Governing AI Agent Behavior." https://www.solsta.io/post/behaviorspec. Proposes the agent equivalent of infrastructure as code: a declarative governance model with an intent file (purpose, scope, tool permissions, constraints) and a lockfile (binding approved intent to immutable artefact identities at deployment), mirroring Terraform's configuration and state separation.
- Uplatz (2026). "The 'Ops' Evolution: A Comparative Analysis of MLOps, LLMOps, and AgentOps for Enterprise AI." https://uplatz.com/blog/the-ops-evolution. Maps the evolution from MLOps (predictive accuracy) through LLMOps (hallucination, non-determinism) to AgentOps (behavioural integrity of autonomous systems). Useful for positioning AgentOps within the broader ops lineage.

On extending ITIL for agentic governance

- Fabrix.ai (2026). "Changing Role of ITIL and SACM Practices in the Agentic Era." https://www.fabrix.ai/webinars/changing-role-of-itil-and-sacm-practices-in-the-agentic-era-with-fabrixs-agentops/. Positions IT departments as "an HR function for AI agents" — onboarding, monitoring and retiring autonomous systems. Extends SACM (Service Asset Configuration Management) to agents; frames the shift from rule-based automation to proactive AI monitoring.

- ITIL (2026). "AI Governance in Organizations." https://www.itil.com/Itil-News-and-Announcements/itil-ai-governance-organizations. ITIL 4′s governance framework for AI across four perspectives: decision authority and risk management, ethical principles, data governance and performance management, regulatory compliance and operational standards. Acknowledges that traditional IT governance is insufficient for AI's unique challenges.
- Prefactor (2026). "Building and Maintaining an Enterprise Agent Registry." https://prefactor.tech/learn/building-enterprise-agent-registry. Practical guidance on agent registries as the CMDB equivalent for agents: centralised systems of record tracking identity, ownership, capabilities, version, status and policy bindings. Covers automated discovery, governance-gated registration, and lifecycle tracking from ideation through retirement.
- Collibra (2026). "Collibra AI Agent Registry: Governing Autonomous AI Agents." https://www.collibra.com/blog/collibra-ai-agent-registry-governing-autonomous-ai-agents. Centralised registration, lifecycle tracking, policy alignment and contextual lineage connecting agents to business use cases and datasets.
- Joshi Management Consultancy (2026). "Scaling IT Operations AI Agents: A Governance Playbook for 2026." https://joshimc.com/php/blog_post.php?slug=scaling-it-operations-ai-agents-a-governance-playbook-for-2026. Reports organisations deploying an average of 28 AI agents face governance barriers including legacy ITSM integration (46%), data quality issues (42%) and change management processes (39%). Argues for structured integration layers, centralised observability and audit infrastructure.

On skill atrophy and cognitive engagement

- Shen, J. and Tamkin, A. (2026). "How AI Impacts Skill Formation." Anthropic. arXiv:2601.20245. A randomised controlled trial with 52 junior Python developers learning a new library. AI-assisted developers scored 17% lower on comprehension (Cohen's d = 0.738, p = 0.01), with the largest gaps on debugging questions. Identifies six distinct AI interaction patterns: full delegation destroyed learning (scores as low as 24%); conceptual questioning preserved it (scores up to 86%). The first controlled evidence that cognitive offloading in professional coding contexts impairs the skills needed to supervise AI.
- Parasuraman, R. and Manzey, D.H. (2010). "Complacency and Bias in Human Use of Automation: An Attentional Integration." Human Factors, 52(3), 381-410. The foundational model of automation complacency and automation bias. Shows both effects occur in naive and expert users, cannot be overcome by training or instructions alone, and result from attention being redirected away from automated tasks under multiple-task load. Provides the theoretical framework for why delegation without engagement weakens skill.
- Maakaron, M. et al. (2025). "The Vicious Circles of Skill Erosion: A Case Study of Cognitive Automation." Aalto University. Demonstrates that skill erosion operates through reinforcing loops of automation reliance, complacency and reduced mindfulness, and that the erosion often goes unrecognised by both workers and managers.

On organisational ambidexterity

- O'Reilly, C.A. and Tushman, M.L. (2004). "The Ambidextrous Organization." Harvard Business Review, 82(4), 74-81. The foundational argument for structural separation: exploitation and exploration succeed in parallel only when housed in distinct units with different cultures and operating rhythms, linked by shared strategic intent.
- O'Reilly, C.A. and Tushman, M.L. (2013). "Organizational Ambidexterity: Past, Present, and Future." Academy of Management Perspectives, 27(4), 324-338. A decade-later review of the evidence; reinforces the structural model and examines when integration works and when it does not.
- Birkinshaw, J. and Gibson, C.B. (2004). "Building Ambidexterity into an Organization." MIT Sloan Management Review, 45(4), 47-55. The case for contextual ambidexterity: individuals switching

between modes within the same unit, enabled by a culture of trust, discipline, stretch and support. Evidence is weaker than the structural model; conditions are demanding.
- Moore, G.A. (2015). Zone to Win: Organizing to Compete in an Age of Disruption. Diversion Books. Distinguishes performance zone (core, exploit) from transformation zone (future, explore); argues they need different management systems and cannot share the same operating model.

About the Author

Avi Sinharay is a technologist and senior leader with 25 years of experience across large-scale multinationals and high-growth start-ups. He holds a Master's degree in Electronics and Information Sciences from the University of Cambridge.

Avi's career spans technology strategy, large programme delivery, and operating model design — from leading 150-person engineering departments to growing scale-ups. He has held senior technology roles and has advised boards and executive teams across media, telecoms, health tech, and AI.

He writes about technology leadership and agentic software development at sinharay.tech.

M:nd Rocket SERV:CES LTD_