The Three Phases of Agentic AI Adoption in Software Engineering

> A framework for understanding whether agents are being driven interactively by humans, delegated bounded work inside the old process, or integrated into the development lifecycle itself.

This is a living white paper. It is maintained as agent architectures, research, protocols, and operating patterns evolve.

Executive summary

Most engineering teams will soon be able to say they use agents. That alone will not mean they have transformed software delivery.

A team can use agents heavily and still move at the speed of the old process. Agents can write code in minutes, then wait days for ticket clarification, human review, security signoff, release windows, or production readiness. The visible activity goes up. The system-level flow barely moves.

That is the central warning of this paper: **the stable default is not agent-native engineering. The stable default is real agent value constrained by a human-shaped development lifecycle.**

This paper describes three practical phases of agentic AI adoption:

- **Phase 1 - Driven.** Humans drive session-bound agents interactively from an IDE, CLI, editor, shell, or chat loop.
- **Phase 2 - Delegated.** Humans delegate bounded work to agents, but the agents still deliver into the old tickets, pull request queues, review rituals, and release cadence.
- **Phase 3 - Native.** Agents become governed participants in how work is assigned, contextualized, evaluated, verified, released, remembered, observed, and audited.

The hard transition is **not** from non-agentic AI to agents. Many teams are already using agents in Phase 1. The hard transition is from **human-driven or human-delegated agents inside the old process** to **agents integrated into the software development lifecycle**.

That transition is difficult because real integration touches everything around code: product intent, acceptance criteria, internal context, documentation, version control, automated tests, agent evals, release safety, security policy, identity,

least-privilege tool access, review design, memory, decision logs, telemetry, work-in-progress limits, measurement, and human escalation.

If those pieces do not change, agents can still create useful local leverage, but they may produce more output without producing a reliably better system.

Self-improving agent teams are an important future capability, but they should not be treated as a fourth adoption phase. They are a compounding layer that becomes possible only after an organization has built a safe, observable, reversible, agent-native development lifecycle.

The model in one table

Phase	Short label	Full name	Control model	Lifecycle posture
1	**Driven**	Interactive Agentic Development	Human-driven, session-bound	Agents help inside the developer's local loop
2	**Delegated**	Delegated Agentic Development	Human-delegated, task-bound	Agents work inside existing tickets/PRs
3	**Native**	Agent-Native Development Lifecycle	System-orchestrated within human policy	Agents are integrated into lifecycle controls

The distinction between phases is not the model used, the vendor selected, or how impressive the demo looks. The distinction is how much of the development lifecycle has been redesigned around the presence of capable AI workers.

Phase 1 can be adopted by individuals. Phase 2 can be piloted by teams. Phase 3 requires operating-model work.

Phase 3 is not yet a broadly validated enterprise norm. It is a target operating model that early agent-native teams are exploring. This paper argues for the lifecycle controls required to make that model credible, not for unbounded autonomy.

The adoption cliff

A linear adoption plan says: start with IDE agents, delegate bigger tasks, then gradually let agents do more. That sounds reasonable, but it hides the hardest part.

Transition	What changes	Hard part	Difficulty
1 -> 2	Agents move from interactive sessions to delegated work	Integration, permissions, review habits	**Moderate**
2 -> 3	Agents move from delegated work to the lifecycle	Org design, governance, context, safety, incentives	**Cliff**

Why 1 -> 2 is moderate

Moving from Driven to Delegated is mostly a tooling and integration change. The team can keep its existing process. Agents can take tasks from the tracker, open pull requests, run tests, post summaries, and wait for human review.

There is real work here: access control, logging, secrets handling, acceptable-use rules, and task selection all matter. But the shape of the organization can stay familiar. The agent moves from a human-driven session into a delegated work slot inside the old system.

That is why Phase 2 often feels achievable. It can be sold as productivity tooling, piloted in one repository, and wrapped in existing controls.

Why 2 -> 3 is the cliff

Moving from Delegated to Native is different. The organization has to redesign how software work flows.

A Phase 3 system asks hard questions:

- **Where does work come from?** If agents can discover, decompose, and dispatch work, the ticket tracker may no longer be the primary work router.
- **What proves the work is worth doing?** If agents can produce implementation faster than humans can refine intent, the lifecycle needs clearer product goals, acceptance criteria, and user feedback loops.
- **Where does context live?** If agents need current architecture, docs, logs, metrics, policies, and design intent, those assets have to become maintained infrastructure, not scattered tribal knowledge.
- **What replaces broad manual review?** If agents can generate code faster than humans can inspect it, the quality gate has to shift earlier and become more automated, contextual, policy-driven, and evidence-based.
- **How are agents evaluated?** The organization needs regression tests for prompts, tools, workflows, and policies, not just tests for the product code agents modify.
- **How does security change?** Agents with filesystem, terminal, browser, package, cloud, or MCP access expand the attack surface. Prompt injection, tool misuse, identity and privilege abuse, command execution, memory poisoning, context over-sharing, credential exposure, and supply-chain risks become lifecycle concerns.
- **How is work-in-progress controlled?** Agents can create more simultaneous work than humans can safely absorb. Queue depth, review capacity, and deployment risk become first-class controls.

- **How do humans know what happened?** Status meetings do not scale to machine-speed workers. The system needs decision logs, traces, tool-call telemetry, audit trails, and clear escalation paths.
- **How is success measured?** Lines of code, accepted suggestions, or number of agent pull requests can inflate activity without measuring value. The organization needs outcome, flow, quality, reliability, security, cost, and developer-experience metrics.

None of those are simply tool settings. They touch product management, platform engineering, DevSecOps, management systems, architecture, compliance, and culture.

That is why the Phase 2 plateau is stable. The agents may improve, but the lifecycle around them remains human-shaped.

A running example: dependency upgrade

The difference between phases becomes clearer when the task is the same.

Imagine a routine dependency upgrade with some risk: a library has a security update, but the change may affect tests, configuration, and deployment behavior.

Phase 1: Driven

An engineer opens an IDE or CLI agent and asks it to inspect the dependency, summarize the changelog, update the package, run tests, and suggest fixes. The agent can do meaningful work, but the engineer is steering each step: which files to inspect, which commands to run, which diff to keep, when to stop, and whether the result is safe.

The agent is real. The autonomy is local and session-bound.

Phase 2: Delegated

A human creates or selects a ticket: "upgrade this dependency." An agent takes the task, works in a branch or sandbox, updates files, runs tests, opens a pull request, and posts a summary.

The work is delegated, but the lifecycle is unchanged. The pull request waits in the normal review queue. Security may review it through the normal cadence. Release timing follows the normal release process. The agent made the work faster, but the work still flows through the old system.

Phase 3: Native

The lifecycle itself knows how to handle this class of work. A policy identifies the dependency update as eligible for agent handling. The agent has governed access to the relevant repository, dependency metadata, changelog, test history,

service ownership, and release policy. It creates a small change, runs task-specific evals and quality gates, records its reasoning and tool calls, checks WIP limits, escalates if a risk threshold is crossed, and lands or stages the change according to pre-approved policy.

Humans do not disappear. They define the policy, own the risk, review exceptions, and audit the trail. The difference is that the lifecycle no longer depends on humans manually routing every routine step.

This is the move from agent usage to agent-native work.

Three distinctions that get conflated

The word "agent" hides several different questions. Untangling them makes the adoption path clearer.

Dimension	Phase 1	Phase 2	Phase 3
Control model	Human-driven	Human-delegated	System-orchestrated within human policy
Runtime location	IDE, editor, CLI, or local shell	Local or remote task environment	Durable platform, cloud, or cluster worker
Lifecycle integration	Developer inner loop	Existing tickets, PRs, review queues	Native routing, gates, memory, logs
Operational identity	Usually session-bound	Task-bound or job-bound	Durable, permissioned, observable identity

A local IDE or CLI agent is still an agent. It may reason through a task, inspect files, call tools, edit code, run tests, and iterate. What makes it Phase 1 is not that it lacks agency. What makes it Phase 1 is that the human is still driving the loop and the agent is not yet a durable participant in the team's development system.

Likewise, moving an agent from a laptop to the cloud or a Kubernetes cluster does not automatically make it more autonomous. Runtime location changes durability, scalability, isolation, observability, and governance options. Autonomy comes from the control model: what the agent is allowed to initiate, decide, modify, and ship without a human steering each step.

The practical graduation is therefore:

session-bound tool -> delegated worker -> governed lifecycle participant.

Throughout this paper, Driven, Delegated, and Native describe adoption phases. Commanded and governed describe control models. A Phase 1 or Phase 2 agent can still be a real agent, but it usually remains commanded by the human-

driven process around it. Phase 3 becomes credible when agents operate under durable identity, policy, activation, memory, observability, and escalation rather than moment-to-moment prompting.

Phase 1: Driven - Interactive Agentic Development

In Phase 1, the team is already using agents. The agent may inspect files, reason through a task, call tools, edit code, run tests, summarize logs, and iterate. What makes this Phase 1 is the control model: the human drives the loop from an IDE, editor, CLI, local shell, or chat session.

Where the value comes from: reduced blank-page friction, faster syntax recall, quicker exploration of unfamiliar APIs, lightweight tutoring, and faster first drafts. Controlled studies show that AI assistance can speed bounded tasks, while broader field studies show that the impact depends heavily on task type, codebase maturity, developer experience, and verification burden.

Lifecycle posture: the software delivery process is mostly unchanged. The agent sits in the developer's inner loop. Planning, work assignment, review, release, incident response, and governance still work the same way.

Common failure mode: mistaking individual acceleration for organizational transformation. A team can have high agent usage and still have the same cycle time, review queues, release constraints, production risk, and user feedback delays.

Signs you are in Phase 1:

- Agents are primarily launched from an IDE, editor, CLI, local shell, or chat session.
- A human steers the agent moment to moment and decides when the task is complete.
- Agent output is always mediated by a human before it becomes work product.
- Agents have little or no durable team identity across tasks.
- The team does not assign ownership of work to AI systems.
- Adoption is mostly measured through usage, sentiment, or perceived individual productivity.

Phase 2: Delegated - Delegated Agentic Development

Agents complete bounded tasks independently, but they work inside the existing human development process.

Where the value comes from: delegation, parallelism, and machine-time execution. Agents can drain maintenance work, explore bugs, write tests, update docs, prepare migrations, or prototype small features while humans focus on

judgment-heavy work.

Lifecycle posture: the lifecycle is still human-shaped. Work is routed through tickets, human-readable summaries, manual review queues, sprint plans, release calendars, and existing approval paths. The agent has become a delegated worker, but the work system has not changed.

The hidden ceiling: the team's total performance is still capped by the slowest human queue. Agent work can finish in minutes and then wait days for prioritization, review, security signoff, release windows, or product decisions.

Common failure mode: generating more work than the organization can safely evaluate. Individual authors may move faster, but reviewers absorb the verification tax. The team sees more pull requests, more diffs, more summaries, and more decisions, without a proportional improvement in delivered value.

Signs you are in Phase 2:

- Agents open pull requests, but humans route and merge them.
- The ticket tracker remains the authoritative source of work assignment.
- Agents report status into human tools rather than participating in structured coordination.
- Release cadence still follows calendar rituals or manual approval gates.
- Agent metrics emphasize output volume more than lifecycle outcomes.
- Review queues, WIP, or production-readiness work expand as agent output increases.

Phase 3: Native - Agent-Native Development Lifecycle

Agent-native engineering begins when the development lifecycle is redesigned around persistent AI workers.

The question changes from "How do we add agents to our process?" to "What process would we design if agents were first-class participants in the system?"

Where the value comes from: reduced coordination delay. Routine work no longer waits for a human to copy state between tools, schedule it into a sprint, manually route it to another worker, or inspect every line after the fact. Quality gates move closer to the authoring moment. Context is available to both humans and agents. Work can move in small, reversible increments.

Lifecycle posture: agents are integrated into the way work is assigned, contextualized, executed, evaluated, verified, released, remembered, observed, and audited.

Structural changes typical of Phase 3:

- **Structured work routing.** Agents can receive, decompose, and hand off work through explicit protocols rather than only through human-readable tickets.

- **AI-accessible internal context.** Code, docs, architecture notes, policies, logs, metrics, and design decisions are available through governed access paths.
- **Intent and acceptance criteria.** Agents work against explicit goals, constraints, tests, and definitions of done, not vague prompts.
- **Agent and workflow evaluation.** Prompts, skills, routing rules, tool policies, and agent behaviors have regression tests and acceptance checks.
- **Automated quality gates.** Tests, type checks, linters, static analysis, security scans, policy checks, and contract checks become load-bearing parts of the workflow.
- **Small batches and WIP control.** Changes are constrained to reviewable, testable, reversible units, and the system limits how much agent-produced work can accumulate in downstream queues.
- **Decision logs and traces.** The system records what agents did, why they did it, what evidence they used, what tools they called, what failed, what was retried, and what requires human attention.
- **Governed memory.** Persistent context is scoped, reviewable, compacted, corrected, expired, isolated, and audited.
- **Human escalation.** Agents know when to stop and ask for judgment rather than continuing with false confidence.
- **Lifecycle measurement.** The team measures cycle time, review time, change failure rate, rework, deployment recovery, product outcomes, developer experience, security findings, cost, and trust.

Common failure mode: allowing agent autonomy to expand faster than observability, rollback, security, evaluation, and human ownership. Agent-native does not mean unconstrained automation. It means the lifecycle has been redesigned so autonomy is bounded, visible, testable, reversible, and accountable.

Signs you are in Phase 3:

- Routine low-risk changes within pre-approved classes can land when explicit gates pass, with audit trails, rollback, and escalation paths.
- The ticket tracker, if it exists, is no longer the only work router.
- Agents use governed internal context rather than only prompt text and local repository state.
- Humans read decision logs, traces, and audit trails to understand work that happened asynchronously.
- The team can explain not only what agents produced, but how the lifecycle controlled, evaluated, verified, measured, and contained it.

What real SDLC integration requires

Phase 3 is not achieved by moving an agent from an IDE to the cloud, or by buying a more autonomous coding worker. Those may be useful steps, but they are not the same as lifecycle integration. Phase 3 requires redesigning the development lifecycle around durable, governed agent participants.

Instead of treating this as a fourteen-item checklist, think of it as four load-bearing pillars.

Pillar 1: Intent and context

Agents are only as useful as the work they are pointed at and the context they can safely use.

This pillar includes:

- **A clear AI stance.** Teams need to know which tools are allowed, which data may be shared, which tasks are appropriate, when human approval is required, and who owns the outcome.
- **Intent, acceptance criteria, and user feedback.** Agents need crisp product intent, non-goals, constraints, definitions of done, and user or operational feedback loops. Otherwise they can move quickly while optimizing the wrong thing.
- **AI-accessible internal context.** Code, docs, architecture diagrams, runbooks, API contracts, logs, metrics, style guides, and product intent need governed access paths.
- **Healthy documentation and data.** Documentation quality becomes infrastructure. So do decision records, service catalogs, dependency maps, runbooks, policy files, and operational signals.

Pillar 2: Control and safety

Agents need boundaries before they need more autonomy.

This pillar includes:

- **Version control, rollback, and AI artifact management.** Teams should version prompts, agent instructions, tool manifests, skills, routing rules, eval suites, policy files, and model configuration alongside product changes.
- **Evaluation and acceptance testing.** Product tests are necessary but not sufficient. Agent-native teams need acceptance tests for common agent tasks, regression suites for prompts and skills, policy-conformance tests, tool-use tests, adversarial tests, memory-safety checks, and evaluations that measure whether agents stop and escalate when they should.
- **Automated quality gates.** Tests, type checks, linters, static analysis, security scans, policy checks, contract tests, preview environments, and production safety checks become load-bearing parts of the workflow.
- **Security, identity, and tool governance.** Every agent and tool action should have traceable identity, scoped permissions, short-lived credentials, isolated execution where possible, validated inputs and outputs, protected secrets, and human approval for high-risk or irreversible actions.

Pillar 3: Flow and platform

Agents can create more work than the system can absorb. The lifecycle needs to control flow, not just generate output.

This pillar includes:

- **Small batches and WIP limits.** The lifecycle should force work into small, testable, reversible increments and limit queue depth so agent output does not overwhelm review, security, or release capacity.
- **Review redesign.** Some review moves earlier into the authoring loop: automated feedback before a PR exists, agent-generated test plans, policy checks, risk classification, and acceptance checks. Human review remains for architecture, product judgment, security risk, user impact, unclear requirements, and trust-building.
- **Platform engineering.** Agents need consistent ways to build, test, run, inspect, deploy, retrieve context, request credentials, call tools, report results, and escalate to humans.
- **Model, vendor, and runtime operations.** Teams need model version tracking, rollout plans, fallback behavior, cost and rate-limit monitoring, latency budgets, context-window limits, outage handling, vendor data controls, and regression testing when a model or tool changes.

Pillar 4: Measurement and learning

Humans cannot govern what they cannot see, and they cannot improve what they do not measure.

This pillar includes:

- **Agent and tool observability.** The team needs telemetry for prompts, retrieved context, tool calls, tool arguments, permission decisions, model selections, memory reads and writes, test results, retries, escalations, cost, latency, and final changes.
- **Measurement beyond output.** Counting accepted suggestions, lines of code, or agent-authored pull requests can reward noise. Integration should be measured through system outcomes: lead time, review turnaround, deployment frequency, change failure rate, recovery time, rework, escaped defects, security findings, WIP, cost, developer experience, user impact, and trust.
- **Feedback into the operating model.** The point of measurement is not to prove that agents are good. It is to discover where agents help, where they move cost, where they create risk, and where the lifecycle needs redesign.

The Phase 2 plateau

The Phase 2 plateau is the paper's central warning.

A plateaued organization can look progressive. Agent usage is high. Agents are active. Pull requests are flowing. The board deck has examples. Developers report that some tasks feel faster.

But the system-level picture may be different:

- Cycle time barely moves because work still waits in old queues.
- Review load increases because humans must inspect more generated work.

- Substantive throughput lags visible activity because many changes are small, shallow, or maintenance-oriented.
- Trust drops because verification burden rises.
- Security teams slow adoption because tool access, identity, and data exposure are unclear.
- Production readiness consumes the time saved during prototyping.
- WIP expands because agents create more concurrent work than the lifecycle can safely absorb.
- Model, tool, or vendor changes cause behavior shifts the team cannot easily explain.

This plateau is not irrational. Existing processes encode audit history, compliance needs, management visibility, security controls, product intent, and trust. Replacing them with agent-native flows requires leadership, platform investment, and careful migration.

The mistake is not choosing to stay in Phase 2. The mistake is pretending that more agent usage will automatically turn Phase 2 into Phase 3.

Crossing the cliff

A Phase 2 team should not start by asking, "How do we make agents more autonomous?"

It should ask, "Which part of our development lifecycle are we willing to redesign around agents first, and what runtime would make that workflow safe, observable, and repeatable?"

There are three honest postures.

1. Plateau deliberately

Keep the current process, capture local value, and revisit later. This is reasonable when trust, regulation, architecture, platform maturity, observability, security posture, or leadership support is not ready.

The key is honesty. Phase 2 is useful, but it is not agent-native.

2. Build a narrow bridge

Pick one workflow where the risk is bounded and the feedback loop is clear:

- documentation updates tied to code changes;
- dependency upgrades with strong tests;
- bug-class drainage in a well-covered subsystem;
- internal tooling improvements;

- test generation and maintenance;
- low-risk refactors with explicit policy gates;
- safe operational checks with read-only access.

For that workflow, build the smallest agent-native loop that can work:

- explicit product intent and acceptance criteria;
- dispatch rules;
- governed context access;
- task-specific evals;
- automated quality gates;
- decision logging;
- tool-call telemetry;
- WIP limits;
- clear escalation criteria;
- rollback paths;
- security boundaries;
- before-and-after metrics.

The goal is not to prove that agents can do everything. The goal is to prove that one slice of the lifecycle can move from human-routed to agent-native without losing safety, quality, explainability, or accountability.

3. Adopt or build a substrate

Some organizations will adopt an agent-native platform. Others will build one. Either path is a platform decision, not a simple tooling decision.

A credible substrate should provide context access, work routing, tool permissions, identity, memory, evals, logs, quality gates, security controls, WIP limits, rollback, human escalation, observability, and measurement. It may run in the cloud, in a Kubernetes cluster, in remote sandboxes, or in local environments, but runtime location is only one part of the design. If it only provides a more autonomous coding agent, it is not enough.

The market is young. Claims should be validated through pilots, not accepted from demos.

What comes after agent-native: the compounding layer

Self-improving agent teams are still worth discussing. They are just not a fourth adoption phase.

Once a team has an agent-native lifecycle, the same machinery that improves the product can begin improving the team system itself. Agents may propose or implement bounded changes to:

- reusable skills;
- schedules;
- routing rules;
- memory structure;
- quality checks;
- documentation conventions;
- automation scripts;
- alert thresholds;
- coordination protocols;
- eval suites;
- tool policies;
- observability dashboards.

This is the compounding layer. It is powerful because improvements to the work system improve future work. But it is also risky because the system is now changing parts of its own operating environment.

A safe compounding layer requires:

- human-owned policy boundaries;
- clear agent and tool identities;
- scoped permissions;
- audit trails;
- reversible changes;
- measurement of impact;
- explicit approval for high-risk changes;
- regression tests for the changed operating layer;
- protection against agents weakening their own controls.

The right test is simple: can the team answer, "What did the agents change about how the team works this week, why did they change it, what evidence supported the change, what policy allowed it, and how do we revert it?"

If the answer is yes, compounding may be safe enough to explore. If the answer is no, the team is not compounding; it is creating unmanaged automation risk.

Evidence and boundaries

This framework is a planning lens, not a settled scientific taxonomy. It synthesizes several evidence streams.

- **AI assistance is widely adopted, but trust remains mixed.** Developer surveys show high AI usage alongside concerns about accuracy, security, privacy, and verification.
- **Many organizations still use GenAI primarily as a personal assistant.** Case-study research on software organizations suggests that process-level adoption is still less common than individual productivity use.
- **Productivity effects are context-dependent.** Controlled experiments show gains on bounded tasks, while field studies in mature codebases show that AI can add verification cost or even slow experienced developers.
- **The surrounding system determines whether local speed becomes organizational performance.** DORA's AI research emphasizes that AI amplifies existing strengths and weaknesses. Platform quality, user focus, internal context, small batches, version control, and clear policy all matter.
- **Agentic software engineering is emerging, but governance remains unresolved.** Recent agentic-SDLC and AI teammate research shows movement from code completion to delegated execution, while also highlighting evaluation, trust, technical debt, security, and attention management as open problems.
- **Security is not optional.** NIST, NSA/Five Eyes, and OWASP guidance all point toward the same practical posture: incremental rollout, explicit accountability, least privilege, monitoring, human oversight, validation, and risk containment.
- **Memory is both capability and attack surface.** Persistent memory can improve long-running work, but it also creates risks around stale context, sensitive data, prompt persistence, memory poisoning, and cross-session influence.
- **Self-improvement is plausible but early.** Research systems such as Reflexion, Self-Refine, and Voyager support the idea that agents can improve through feedback, memory, iterative refinement, and skill libraries. They do not prove that enterprise software teams can safely run unconstrained self-improving systems.

Claims this paper intentionally does not make

- It does not claim every organization should become agent-native now.
- It does not claim tickets, sprints, or planning tools disappear everywhere.
- It does not claim agents should merge all code without human review.
- It does not claim productivity automatically improves when AI is added.
- It does not claim self-improving systems should be unconstrained.
- It does not claim the three phases are universal laws.
- It does not claim Phase 3 is a mature industry standard.

The narrower claim is stronger: **agentic AI creates local acceleration first. Capturing that acceleration at the organizational level requires integrating agents into the development lifecycle.**

Self-assessment

Answer yes or no. The goal is not to win a score; it is to expose where your development lifecycle actually sits.

1. Are agents primarily driven from IDE, editor, CLI, local shell, or chat sessions?
2. Does a human steer the agent moment to moment and decide when each task is complete?
3. Does every agent-generated change pass through a human before it becomes work product?
4. Is agent state mostly session-bound rather than attached to a durable team identity?
5. Can agents complete bounded delegated tasks independently?
6. Do agents receive work through the same ticket tracker humans use?
7. Do agents primarily report through human-readable PR summaries, chat, or comments?
8. Does agent work still wait in normal human review, release, or approval queues?
9. Are product intent, acceptance criteria, and non-goals legible to agents?
10. Can agents access internal context through governed, least-privilege paths?
11. Are prompts, agent instructions, skills, tool manifests, eval suites, and policy files versioned?
12. Are agent-assisted changes forced into small, reversible batches?
13. Are WIP limits or queue-depth controls applied to agent-created work?
14. Do task-level agent evals or acceptance tests run before relying on agent output?
15. Do automated quality and security gates run before human review for routine changes?
16. Does each agent and tool action have a traceable identity and scoped permission set?
17. Can agents route work to other agents through structured coordination rather than human copy/paste?
18. Is there an authoritative decision log, trace surface, or tool-call audit trail for agent work?
19. Is persistent memory governed, reviewable, isolated, expirable, and correctable?
20. Can routine low-risk changes within pre-approved classes land when explicit gates pass, with rollback and escalation?
21. Are AI outcomes measured through lifecycle, quality, reliability, security, cost, and product metrics rather than output volume?
22. Can agents propose improvements to skills, schedules, memory, routing, evals, policies, or automation?
23. Are those self-improvements bounded, audited, tested, measured, and reversible?

Scoring:

- **Mostly yes to 1-4; mostly no afterward:** Phase 1. Agents are interactive and human-driven.
- **Yes to 5-8; mostly no to 9-21:** Phase 2. Agents are delegated workers inside the old process.

- **Yes to 9-21:** Phase 3. Agents are integrated into the development lifecycle.
- **Yes to 22-23, with strong evidence for 9-21:** The team may be ready to explore the compounding layer.

Mixed signals are common. A team that has agents but no governed context, decision logs, quality gates, evals, observability, WIP controls, or lifecycle metrics is probably in Phase 2. A team that allows self-improvement without auditability is not advanced; it is exposed.

Conclusion

The most important distinction in AI adoption is not which tool a team buys. It is whether AI is integrated into the development lifecycle.

In Phase 1, humans drive agents interactively inside their local development loop. In Phase 2, humans delegate bounded work to agents inside the old process. In Phase 3, the process itself changes so agents can participate safely and usefully in how software is planned, built, evaluated, checked, released, remembered, observed, and improved.

The strategic mistake is assuming that more agent usage naturally produces Phase 3. It does not. Without product intent, context, evals, gates, rollback, logs, security boundaries, identity, platform support, WIP control, observability, and measurement, the organization gets more output without a reliably better system.

For teams at Phase 2, the next step is not maximum autonomy. The next step is lifecycle integration: pick a narrow slice of work, make the intent and context available, define the gates, run the evals, log the decisions, observe the tool calls, measure the result, and keep the human role clear.

The question for engineering leaders is therefore not only, "Are we using AI?" Most teams are, or soon will be.

The better question is: **where, exactly, is AI wired into our development lifecycle, and what controls make that wiring safe enough to trust?**

Sources and further reading

The framework in this paper is an interpretation of current practice and research, not a direct restatement of any one source. The sources below are grouped by evidence type so readers can separate established guidance from emerging work.

Established guidance and operating-model research

- DORA, **State of AI-assisted Software Development 2025**. Useful for the claim that AI's organizational impact depends on the surrounding system, not the tool alone.
- DORA, **Balancing AI tensions: Moving from AI adoption to effective SDLC use**. Useful for the verification tax, code review pressure, workflow gap, and the distinction between adoption and effective lifecycle use.
- DORA, **AI Capabilities Model**. Practical framing for the capabilities that amplify the benefits of AI adoption.
- DORA, **Capability catalog**. Useful for connecting AI adoption to broader delivery capabilities: documentation quality, continuous delivery, test automation, pervasive security, WIP limits, platform engineering, user focus, small batches, and version control.
- Google Cloud Blog, **Introducing the DORA AI Capabilities Model**. Useful for the seven AI capabilities: clear AI stance, healthy data ecosystems, AI-accessible internal data, version control, small batches, user focus, and quality internal platforms.
- DORA, **Version control**. Relevant to rollback, auditability, reproducibility, and versioning AI artifacts such as prompts and agent configuration files.
- DORA, **Platform engineering**. Relevant to the platform layer needed to turn local AI acceleration into organizational performance.
- DORA, **AI-accessible internal data**. Relevant to context engineering, internal data access, and trust.
- Nicole Forsgren et al., **The SPACE of Developer Productivity**. Useful for avoiding one-dimensional productivity metrics.
- Abi Noda et al., **DevEx: What Actually Drives Productivity**. Useful for measuring developer experience alongside system outcomes.

Security, governance, and risk management

- NIST, **AI Risk Management Framework**. Useful for grounding AI adoption in govern, map, measure, and manage practices.
- NIST, **Secure Software Development Practices for Generative AI and Dual-Use Foundation Models**. Useful for grounding AI-specific secure development practices in the broader SSDF.
- NSA and Five Eyes partners, **Careful Adoption of Agentic AI Services**. Useful for agentic AI risk categories, incremental deployment, accountability, monitoring, and human oversight.
- OWASP, **Top 10 for Large Language Model Applications**. Useful for prompt injection, insecure output handling, excessive agency, sensitive information disclosure, and related LLM application risks.
- OWASP, **AI Agent Security Cheat Sheet**. Useful for agent-specific controls around tools, prompt injection, memory, human-in-the-loop approval, output validation, observability, multi-agent communication, and data protection.
- OWASP, **MCP Top 10**. Useful for model-context and tool-layer risks: token exposure, scope creep, tool poisoning, command execution, insufficient auth, audit gaps, shadow servers, and context over-sharing.

- OWASP, **Agentic Security Initiative**. Useful for the emerging agentic application security taxonomy and mitigation guidance.
- Yue Liu et al., **Your AI, My Shell: Demystifying Prompt Injection Attacks on Agentic AI Coding Editors**. Useful for the security implications of high-privilege coding agents.
- Frontiers, **A systematic literature review on the impact of AI models on the security of code generation**. Useful as a review of code-generation security risks and mitigation strategies.

Empirical adoption and productivity evidence

- Stack Overflow, **2025 Developer Survey: AI**. Useful for adoption, sentiment, trust, agent accuracy, and security concerns.
- Kai-Kristian Kemell, Matti Saarikallio, Anh Nguyen-Duc, and Pekka Abrahamsson, **Still just personal assistants? A multiple case study of generative AI adoption in software organizations**. Useful for grounding the claim that organizational adoption is often still personal-assistant-oriented.
- Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer, **The Impact of AI on Developer Productivity: Evidence from GitHub Copilot**. Useful as evidence that AI assistance can speed bounded tasks.
- Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein, **Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity**. Useful as a counterweight showing that AI impact can be negative in mature, high-context codebases.

Emerging agentic-SDLC and self-improvement research

These sources are useful signals, but many are recent preprints or early research artifacts. They should inform the framework without being treated as settled enterprise evidence.

- Hao Li, Haoxiang Zhang, and Ahmed E. Hassan, **The Rise of AI Teammates in Software Engineering**. Useful evidence that autonomous coding agents are operating in real repositories, while trust and utility gaps remain.
- Happy Bhati, **Agentic AI in the Software Development Lifecycle**. Useful as a recent synthesis of agentic SDLC concepts and open problems.
- OWASP, **Agent Memory Guard**. Useful as an emerging practical reference for memory poisoning risks and mitigations.
- Noah Shinn et al., **Reflexion**. Useful background for feedback-driven agent improvement.
- Aman Madaan et al., **Self-Refine**. Useful background for iterative agent refinement.
- Guanzhi Wang et al., **Voyager**. Useful background for skill libraries and compounding agent capability.

This framework describes patterns visible in AI-assisted and agent-native engineering work as of 2026. Phase boundaries are heuristic; teams in transition will show mixed signals. The framework is offered as a planning lens, not

as a universal maturity model.