

Software Delivery in the Age of AI

Sii's perspective on scaling AI across the software development lifecycle

Executive summary

AI is already an integral part of software delivery. Not everywhere, not evenly, and not always with clearly visible results — but the direction is clear. AI is moving from a tool that supports individual developers to an integral part of the software delivery system.

This report is for leaders who want to move from AI experimentation to measurable improvements in software delivery. It shows where the market stands today and identifies signals that matter for AI across the software development lifecycle (SDLC).

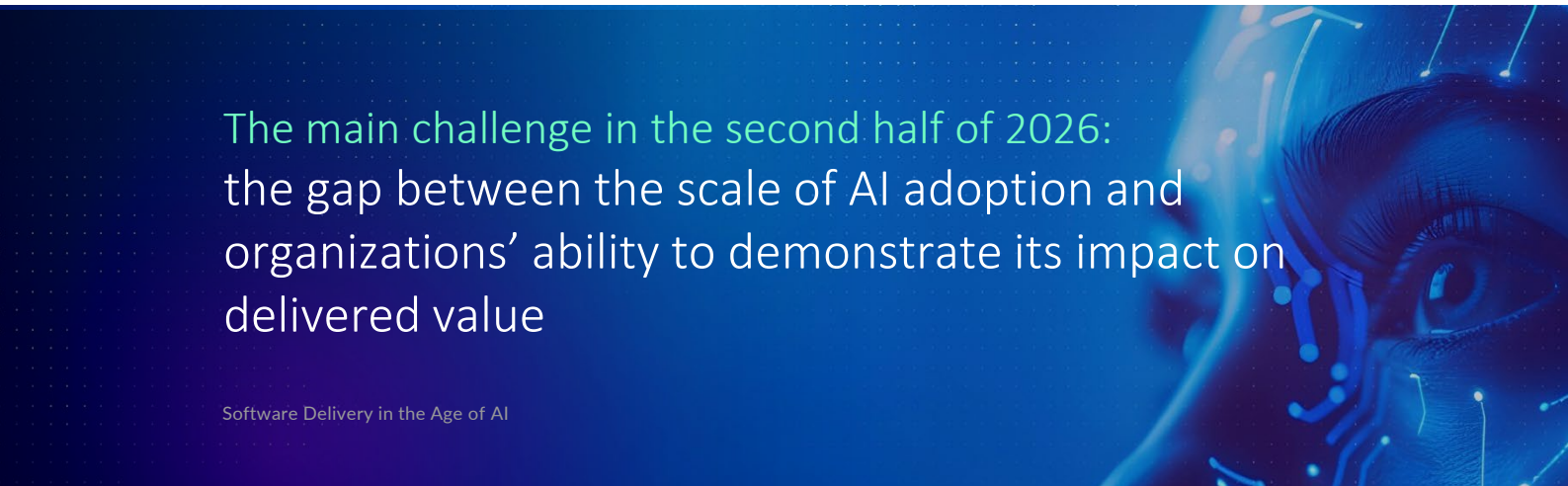
One of the report's key sections explains how Sii approaches scaling AI in practice: from the AI-native SDLC maturity model, through governance and security, to methods for controlling agent workflows. The report also presents results from Sii's own experiments in this area.

Readers mainly looking for concrete next steps can go directly to the "6–12 Month Action Plan for Leaders" section.

Most organizations started with developer productivity: code generation, code analysis, debugging, documentation, and unit testing. These use cases matter. They are visible and easy to pilot. But focusing only on local acceleration can create a false sense of progress.

Software delivery goes beyond writing code. If AI speeds up coding while review or testing still operate in the old way, the business sees little change.

This creates a measurement problem and leaves one question unanswered: is AI actually improving flow across the entire software delivery system?



The key is understanding whether AI is actually changing the end-to-end SDLC or simply increasing the number of AI tools in use. Many organizations already have access to number of AI tools. The next challenge is designing how teams work and deliver software with them.

That means looking for practical signals showing where AI improves flow, where it creates new bottlenecks, and where stronger controls are needed before scaling.

What to pay particular attention to:

1. AI assistants are becoming a standard part of engineering work.
2. Testing and quality assurance are key near-term areas for AI adoption.
3. AI agents are moving from demonstrations to supporting real processes.
4. Scaling AI across the SDLC requires governance: shared rules for security and AI use.
5. Whether AI adoption remains a collection of local experiments or becomes a repeatable way of working will depend on how developer platforms are built and evolved.

AI's real impact in the SDLC increasingly comes from improving the software delivery system, not simply from speeding up coding.

Core thesis

AI creates real value in the SDLC when task-level productivity gains translate into improvements across the entire software delivery lifecycle

1. Market landscape

In many development teams, AI has moved beyond experimentation and become part of everyday work. More importantly, its use is increasingly extending beyond coding. More roles in the software delivery process are being supported by AI.

From creating test cases and structuring requirements, through working with system documentation, to incident analysis. This is a clear signal that AI adoption is moving from individual developer tasks into broader SDLC workflows.

> *More than 50% of Gartner survey respondents use AI to write and understand code, debug, generate documentation, and create test cases*

Gartner, How to Maximize the Impact of Agentic AI in the SDLC, January 19, 2026.

> *According to Gartner, 66% of respondents using AI across at least 10 use cases reported improved speed to market.*

Gartner, How to Maximize the Impact of Agentic AI in the SDLC, January 19, 2026.

Tool adoption is a fact. Success is less certain.

Many companies have added AI tools to software delivery without redesigning how the process itself works. Teams may generate code faster, create tests, and summarize incidents, but time-consuming handoffs between teams or team members, approval waits, review queues, and delivery constraints often remain. The result is fragmented adoption rather than a redesigned software delivery system.

Within the same organization, one team may use AI intensively, another may avoid it, and a third may experiment outside officially approved tools. Policies are unclear, and metrics often track tool usage rather than software delivery outcomes. This makes it difficult to determine whether AI is actually improving flow, quality, or risk control.

Against this backdrop, it is useful to distinguish levels of AI adoption across the software development lifecycle:

At the first level, AI supports individual developer tasks but operates within clear boundaries: approved tools, data policies, and maintained instructions. Humans remain fully accountable for the outcome.

The next stage begins when AI use stops being an individual habit and becomes a shared, repeatable team practice across selected SDLC phases.

The team uses shared prompts, instructions, and project context, and measures the impact of AI adoption.

At the highest level, agents perform multi-step work, while rules are enforced technically through monitoring, decision gates, and quality controls.



Agentic task execution

AI agents handle bounded tasks with access to tools: updating dependencies, creating pull requests, triaging security vulnerabilities, maintaining tests, and preparing documentation. Most organizations are somewhere between task support and workflow support. Some are piloting agentic execution.

Few, however, are ready for broad autonomy. Sii sees the same pattern in client conversations. The question is more often "Which AI tool should we buy?" than "How do we use AI without losing control over delivery quality, security, and cost?"

Key takeaway

The market has moved beyond basic curiosity about AI.
The harder task now is to design adoption that works.

2. The productivity trap

The simplest narrative around AI is faster coding. But that is incomplete.

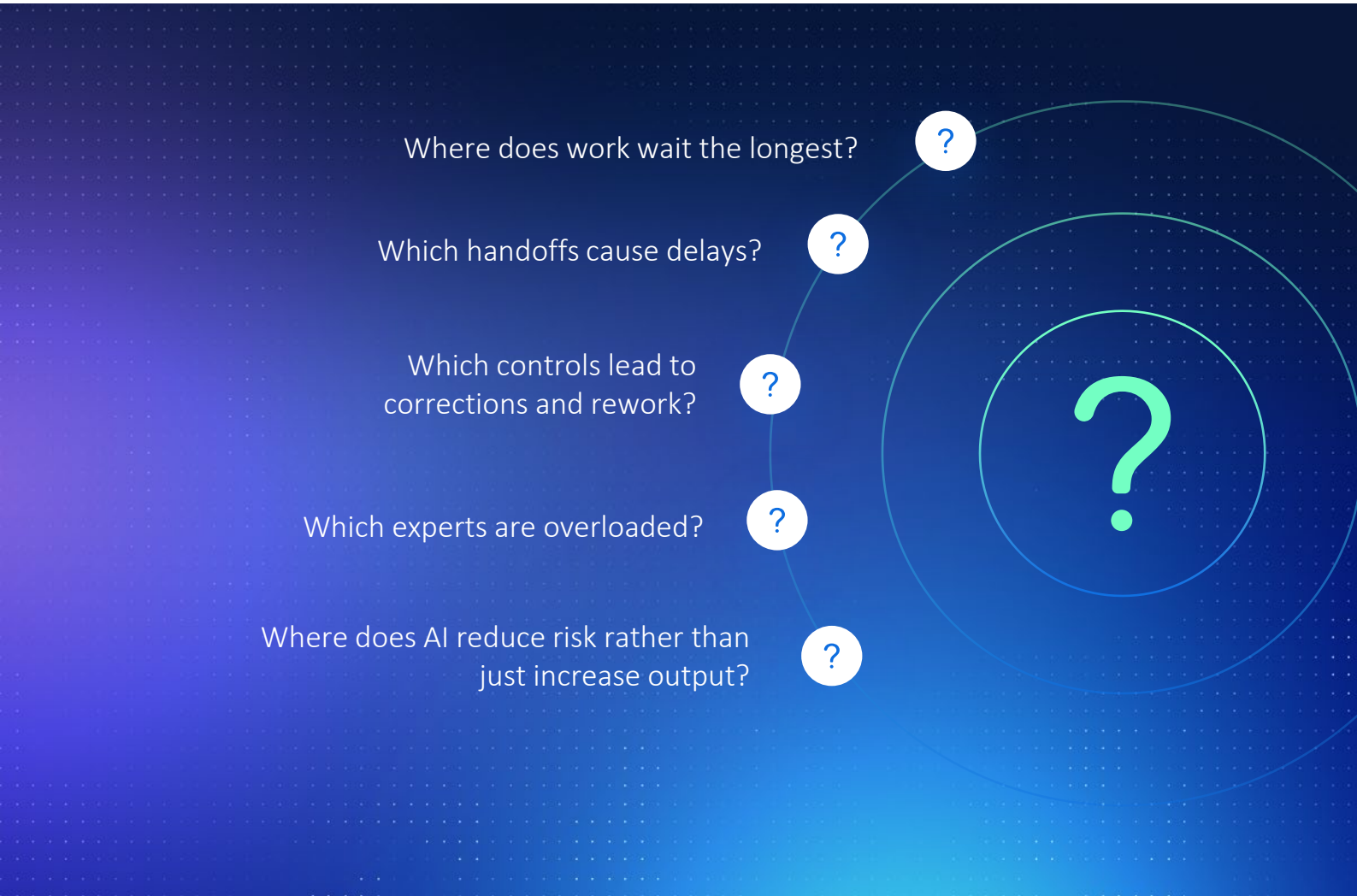
The fact that a developer can finish a task faster with AI does not mean the organization delivers software faster. Work still has to pass through code review, testing, security checks, delivery, and production readiness.

Speeding up one stage does not remove constraints elsewhere in the process. Faster coding may simply increase review queues, a higher volume of changes can overload QA, and late security checks can generate more rework just before release.

Similarly, manual approvals can still slow down releases, even if earlier stages have been successfully automated.

This is how many AI initiatives lose credibility: they report tool adoption, prompt counts, or time saved in isolated tasks. Leaders then ask why lead time, defect rates, or deployment predictability have not improved.

The answer is usually simple: AI was applied where it was easiest, not where the system had bottlenecks.



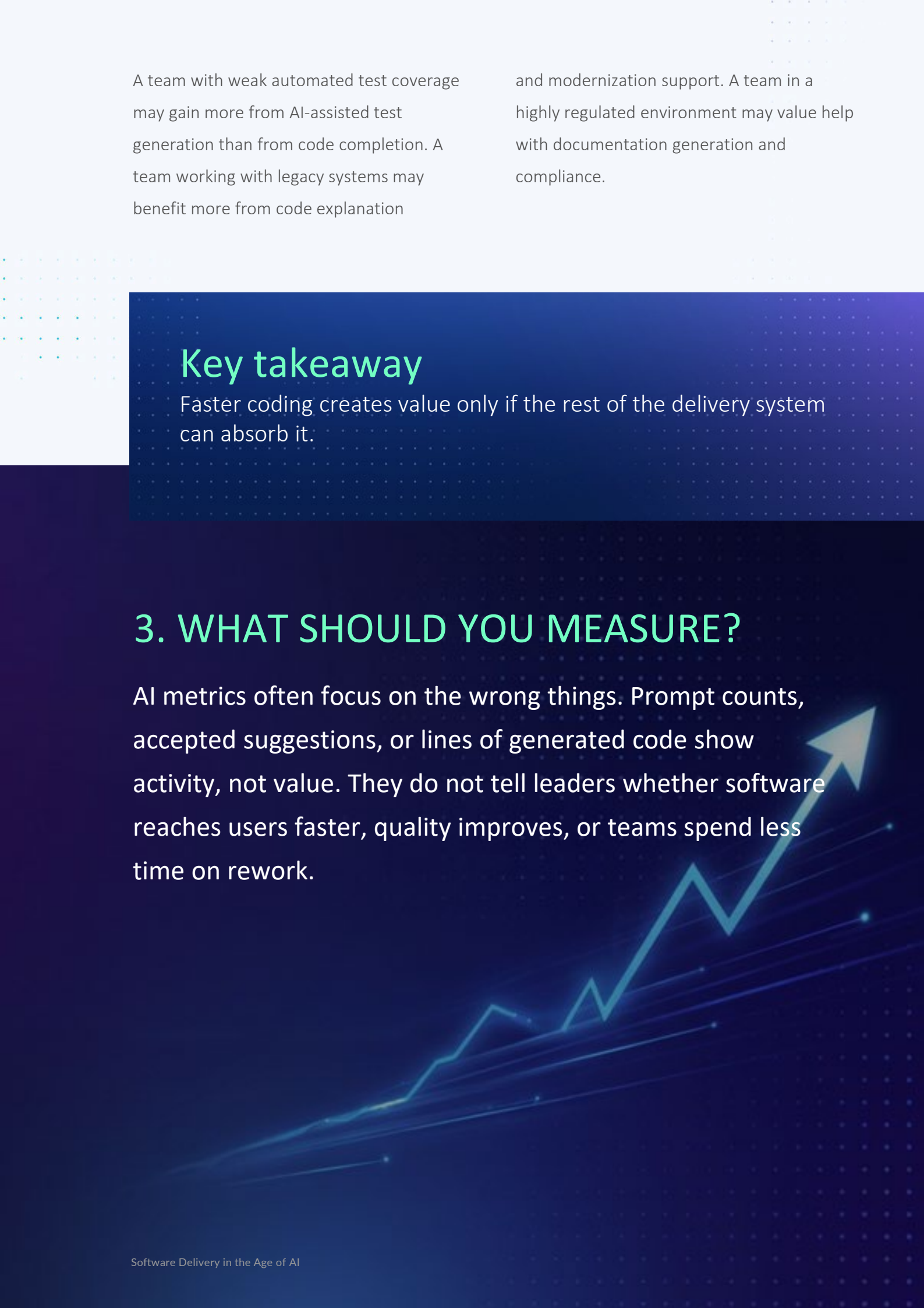
A team with weak automated test coverage may gain more from AI-assisted test generation than from code completion. A team working with legacy systems may benefit more from code explanation and modernization support. A team in a highly regulated environment may value help with documentation generation and compliance.

Key takeaway

Faster coding creates value only if the rest of the delivery system can absorb it.

3. WHAT SHOULD YOU MEASURE?

AI metrics often focus on the wrong things. Prompt counts, accepted suggestions, or lines of generated code show activity, not value. They do not tell leaders whether software reaches users faster, quality improves, or teams spend less time on rework.



A practical measurement model should cover four areas.

Quality and risk: Has the escaped defect rate changed? Has rework decreased? Has test stability improved?

Adoption: Who uses AI? Which tools? In which teams? For which SDLC tasks?

Flow: Has task lead time decreased? Has code review time fallen? Has the delivery cycle improved? Do tasks spend less time waiting in queues?

Developer experience and business outcomes: Do developers report less toil? Has onboarding improved? Do customers receive changes faster?

Agentic workflows require additional controls:

- How often do AI-generated changes pass review on the first attempt?
- How often does a human need to correct AI output?
- Is the project context used by AI up to date?
- Which agentic workflows have clear acceptance criteria?
- Which actions are logged and auditable?

Key takeaway

The goal is not to create bureaucracy.
It is to stop confusing activity with progress.

4. From assistants to agents

AI assistants help people perform tasks. AI agents attempt to complete a bounded scope of work independently.

The difference matters. An assistant can explain a method, suggest a test, or draft documentation. A human remains in full control at every step.

An agent can analyze a ticket, inspect files, make changes, run tests, and prepare a pull request. It can work in the background while the developer focuses on something else.

That shifts AI from helper to active participant in the workflow.

As of mid-2026, common agentic scenarios include:

- preparing pull requests;
- generating and maintaining tests;
- updating dependencies;
- remediating security vulnerabilities;
- code review summaries;
- documentation updates;
- incident summaries;
- refactoring proposals.

This does not mean the SDLC becomes fully autonomous. The limitations remain clear: incomplete context, weak domain understanding, security and licensing risks, architectural mismatch, and difficulty auditing machine-made decisions.

The maturity path in the Sii AI-native SDLC standard:

The levels and transition criteria are part of the Sii AI-native SDLC standard. They are used to decide when to increase autonomy within the assessed scope.

1

L1 Assisted:

Task support using approved tools, data rules, maintained instructions, and human accountability.

2

L2 Integrated:

A shared, repeatable team practice across selected SDLC phases, maintained assets, and measurement against a baseline.

3

L3 AI-native:

Multi-step agentic work, technical enforcement of the standard, monitoring, human gates, and durable evidence of quality and outcomes.

The levels are cumulative. L3 AI-native requires, among other things, at least two consecutive measurement periods with no increase in defect or rework rates versus baseline, and no production incident attributed to an AI-generated change that passed review. Evidence applies only to the assessed scope.

Service maturity and task autonomy are separate decisions. A well-performing service may intentionally remain at L1 Integrated or L2 AI-native, while one person using an advanced agent does not make a team AI-native.

Key takeaway

Agents work best when the task has clear boundaries, complete context is available, and human verification is built into the system.

5. Model vs. harness

Top AI models are becoming less differentiated in overall quality. In practice, this means model choice alone is no longer the main source of competitive advantage. Leading models now perform at broadly similar levels on software engineering tasks; differences increasingly concern behavior and details rather than overall capability class.

An enterprise layer that directs the model and embeds it in project context is therefore becoming more important: it supplies domain knowledge, architectural rules, working standards, constraints, controls, budgets, and task-routing rules. We refer to this layer as the “harness” or “meta-harness”.

> *“Advantage does not come from the AI model itself, but from how it is integrated with process, tools, and knowledge”*
>
> *Finding from Sii’s 11-day experiment: 70 tickets, 3 production projects, full telemetry.*

As models mature, the harness matters more

The distinction between model and harness is highly practical. Without a harness, working with an agent resembles a conversation: a human asks, the agent responds, and the human evaluates the result. Everything happens step by step, with no guarantee that the same process will be repeated consistently.

With a harness, the task moves through a defined process: planning, implementation, testing, and review, with explicit control points between stages. The model provides the reasoning, while the harness keeps its actions within defined boundaries and prevents uncontrolled changes to the codebase.

A harness handles six things: observing the environment, supplying context, controlling actions, executing them, maintaining state, and verifying results.

An effective harness rests on three pillars.



Contracts and gates

Agent output has a predefined format (e.g., JSON) rather than free-form text. If information is missing or the result is ambiguous, the process stops — the agent does not guess. A human approves only two points: the plan and the code merge. The remaining steps are automated, but not unsupervised.



Routing and economics

Each role — planning, implementation, review — is handled by a model selected for the task and budget. Each task has a hard cost ceiling; exceeding it stops the run instead of opening an unlimited bill.



Auditability

Every run leaves a complete record: the plan, decisions made, cost, and action log. Oversight is therefore not merely written into policy; it is a natural consequence of how the process is built.

The two layers — model and harness — must work together: the agent performs the task, while the harness sets boundaries and verifies the result.

The number of human gates should primarily reflect risk level, permission scope, and degree of environment isolation. In an isolated harness where the agent operates within an approved plan, fewer human decisions may be needed. In a development environment or with broader permissions, additional control points are required.

Key takeaway

The harness does not replace the model — it governs how the model operates. The harness largely determines how effective the agent can be.

Harness tailored to the organization

Research suggests that tool effectiveness depends less on the AI model alone than on the harness built around it and its fit to the project domain (Guo et al., 2026, arXiv:2606.20683). Sii's internal experiment supports this conclusion: we extended Claude Code — Anthropic's harness — with our own devkits, data-source integrations, subagents, and internal tool integrations. As a result, 86% of tickets completed the full process without human intervention beyond the defined gates, compared with 33% using the baseline Claude Code configuration.



Delivery rate: 86%

60 of 70 tickets completed the full process along the standard path. The remainder were stopped at a specific stage with a clearly identified reason — none ended without an explanation.



Cost: Average USD 6.11 per delivered ticket

Median cost was USD 2.96, and the most expensive ticket cost USD 31.21. Higher cost, however, was driven not by greater difficulty but by broader task scope. Splitting large tickets before work begins is therefore one of the simplest and cheapest ways to optimize the process.



Autonomy ratio: 9.7×

536 agent calls for 55 human decisions. Based on our observations, in an "agent + prompt" model the ratio is close to 1:1, because almost every agent step requires a human response. The harness automates that loop and reduces the number of points where intervention is needed.



Zero silent failures

Every stop had a clearly identified technical cause. The process is fail-closed: without positive verification, the run stops, and the agent does not approve its own work.

Works best for brownfield projects

An existing product, a steady flow of tickets, and many people working in parallel. What matters most is quality, predictability, and control over each ticket — not the speed of the first code change.

Not always worth it: solo greenfield

One person, a rapid prototype, and a simple goal: get a working application running as quickly as possible. In this scenario, gates, additional artifacts, and limits may add more overhead than value.

A harness does not solve poorly prepared tickets or tasks that require exploration — those still belong to people. What changes is the developer's role: less time writing code, more time precisely defining tasks and evaluating the agent's work. Accountability shifts to where humans still have an advantage: setting intent and making decisions. The experiment shows that a controlled agentic process can operate on production projects, with human intervention limited to defined gates and costs measured at ticket level. However, it does not compare delivery time or total cost with a non-agentic process and should not be treated as a benchmark for other project types.

Key takeaway

A harness extends the model's capabilities by providing control, repeatability, and access to domain context — especially in larger, more complex projects.

Four pillars for scaling SDLC agents

Skills: The developer role is changing — less time may be spent writing code itself, and more on precise task definition, result verification, and auditing the agent's work.

Governance: The rules for how an agent operates must be embedded in the process: who approves the plan, when the agent may start work, what requires human approval, and what is recorded for audit.

Security: The agent should have access only to what it genuinely needs — specific repositories, tools, and environments. Sensitive data can stay inside the organization, while selected tasks can be routed to local models.

Cost: The budget should act as a circuit breaker: a per-task limit, cost control at each stage, and automatic stop when the threshold is exceeded. Cost should be known before the run starts, not only at the end.

In practice, the same questions return whenever agents are scaled:

- Who approves the plan before the agent starts working?
- What happens if the agent exceeds the budget?
- Can you reconstruct why a given change reached production?
- Can sensitive data stay local instead of being sent to the cloud?
- Is the cost of a task known before it starts?

Model choice is tactical. The conditions for scaling are strategic.

Key takeaway

As models mature, more of the advantage shifts to the system around the model — the harness, context, controls, and the way work is organized.

6. What works across the SDLC

AI can support the entire SDLC, but its value, maturity, and risk vary by stage.



Discovery and requirements

At this stage, AI primarily helps organize information: summarizing stakeholder materials, grouping feedback, and identifying ambiguity. It can also draft user stories or acceptance criteria, but product decisions still require human judgment.



Architecture and design

In architectural work, AI is useful for analyzing documentation, drafting ADRs, mapping dependencies, and assessing the impact of changes. The final decision still belongs to the architect because design choices affect cost, security, and maintainability.



Coding

The most visible impact of AI today is in day-to-day coding: code generation, legacy code explanation, refactoring, debugging, and unit test creation. The benefit quickly disappears if seemingly correct code moves forward without rigorous review.



Testing and quality

Testing offers a broad field for AI use — from generating test cases and test data, through automation drafts, to failure analysis and test maintenance. It is also practical because outcomes are relatively easy to verify and measure.

Security



In security, AI can accelerate dependency analysis, vulnerability triage, threat modeling, and preparation of secure-coding guidance. These use cases require particularly careful verification, because a wrong recommendation can increase risk rather than reduce it.

DevOps and operations



In DevOps, AI is most useful where large amounts of information need to be processed quickly: preparing release notes, summarizing incidents, supporting postmortems, or analyzing observability data. The greatest value today comes from reducing repetitive analysis and reaching relevant information faster, not from full autonomy.

The best starting point depends on the organization's needs. A product company with weak requirements discovery may begin by supporting user-need discovery. Financial institutions may start with compliance support. A large engineering organization may begin with testing, documentation, or code review.

Key takeaway

Choose use cases based on delivery constraints, not current trends.

7. Testing as a practical entry point

Testing deserves separate attention. It is repetitive, has clear verification criteria, and produces measurable outcomes. It also directly affects delivery flow — when testing is slow or unstable, faster coding does little to help.

AI can help QA teams with:

- generating test cases;
- creating test data;
- preparing automation scripts;
- analyzing failed tests;
- maintaining regression suites;
- identifying gaps in code coverage;
- supporting self-healing test automation.

Public Sii materials based on internal experiments and testing-related solutions point in the same direction. We focus on automated test creation and AI-assisted test quality — not speed alone. A generated test must effectively verify system behavior. A fast test that checks the wrong behavior adds noise, not value.

- Sii Testing Lab Study: Exploring the “AI Boom” in Test Automation
- From automation to autonomy: The evolution towards Agentic Testing, illustrated by mabl
- AI at work for a Test Developer – what Codex, Claude Design, Claude Opus 4.7, Codex Security, and Claude Mythos really change
- Leveraging AI and Playwright for Efficient UI & API Testing



AI does not eliminate testers; it changes their work. Testers spend less time on repetitive tasks and focus more on risk, code coverage, validation logic, and review.

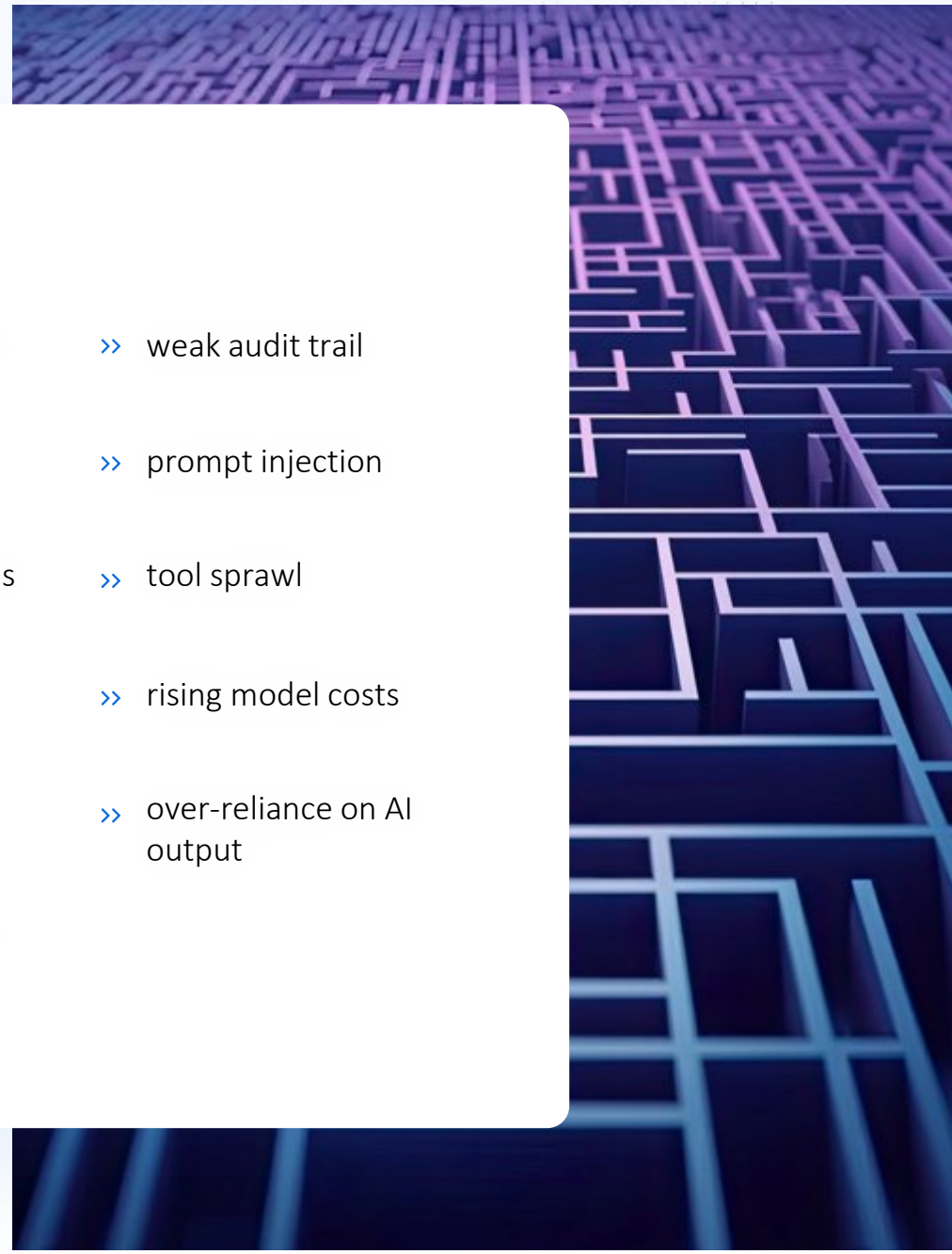
Testing can be a good first area for scaling where it is a material flow constraint. Its advantages include repetitive tasks, relatively clear verification criteria, and measurable outcomes.

Key takeaway

AI-assisted testing works best when testers remain accountable for quality logic.

8. Governance and human oversight

AI-assisted software delivery cannot scale on trust alone. Risk increases with autonomy — a code suggestion is one thing, an agent changing files, calling tools, or creating pull requests is another. It can affect delivered artifacts and, indirectly, production systems.



Key risks

- hidden dependencies
- incorrect code
- inconsistent standards
- insecure code
- data leakage
- license or IP violation
- weak audit trail
- prompt injection
- tool sprawl
- rising model costs
- over-reliance on AI output

A useful governance model answers concrete questions:

- Which tools are approved?
- What data may be used?
- Which use cases are allowed?
- Which tasks require human approval?
- Which outputs require automated checks?
- How are AI-assisted changes recorded?
- Who is accountable for risk?



Human oversight should match the nature of the work.

Low-risk, repetitive work can operate with human-on-the-loop oversight. Medium-risk work requires checkpoints. High-risk work requires human approval (human-in-the-loop).

Critical systems require strict rules, standards, and controls.

This is particularly important in regulated environments, systems of record, financial logic, authorization logic, sensitive-data processing, and core intellectual property.

Key takeaway

The more autonomy AI receives, the more transparent the control mechanisms must be.

9. Governance for the Sii AI-native SDLC

A policy has value when it translates into a decision, a tool, or a person. Every material risk has a control and an owner assigned — starting in the pilot phase.

Three groups of mechanisms turn principles into decisions.

1

Data and access

An approved model access point, limited context scope, and no sharing of secrets. The agent operates under an identity with least privilege.

2

Approval and output controls

Any action requiring approval has a named human approver, a defined approval scope, and a recorded decision before execution. Tests, scanners, and independent acceptance criteria verify the output.

3

Stop and replay

Limits on attempts, time, and cost stop the run. The execution record includes model and configuration versions, sources, tool calls, control results, and decisions — with no secrets in the logs.

Accountability should be assigned to named roles: the Delivery Manager owns the service and rollout, the Solution Architect owns the technical side, and domain experts, QA, and security provide criteria and assessment. The agent does not assume accountability for an accepted outcome.

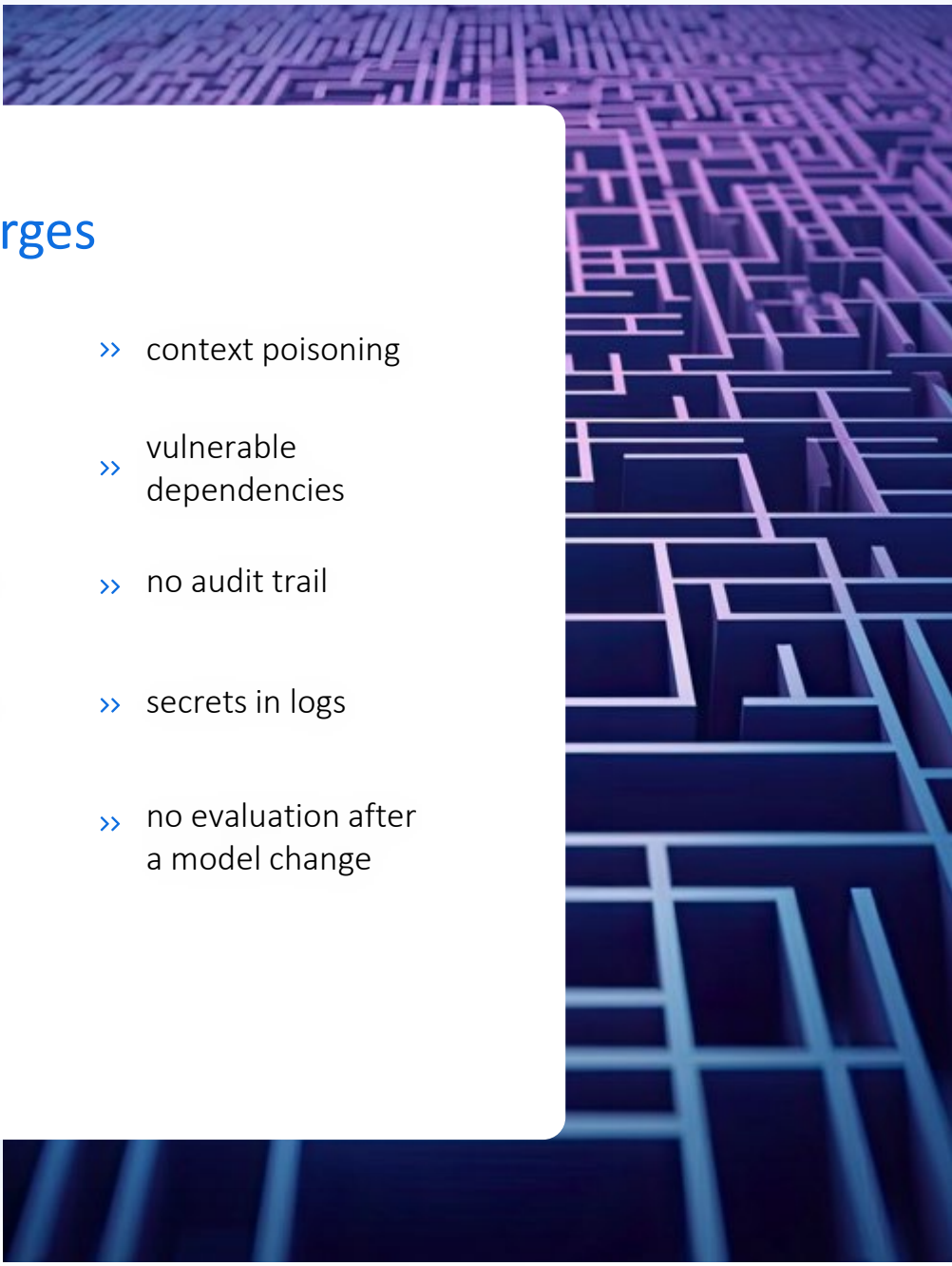
Key takeaway

Security does not come from an instruction in the prompt, but from permissions and the approval process.

10. Security when working with agents

An agent that reads files, calls tools, and creates pull requests expands the attack surface of the delivery system. Filtering model output is not a sufficient control.

One important practical risk is prompt injection: a document, ticket, or repository file may contain an instruction that tries to redirect the agent's behavior.



Where risk emerges

- prompt injection
- secret leakage
- excessive permissions
- unknown integrations
- production access
- data leaving the organization
- context poisoning
- vulnerable dependencies
- no audit trail
- secrets in logs
- no evaluation after a model change

Four mechanisms that limit the impact of an agent error.

Permissions: the agent runs under a least-privilege identity; allowed commands, repositories, and integrations are explicitly defined.

Isolation: a segregated environment with no access to production or secrets. Git, CI, and permissions are controlled by the harness.

Integrations: APIs and MCP require separate permission controls. A connection alone does not grant authority to act.

Data: an approved model access point, limited context scope, no secrets. Routing can direct selected tasks to local models.

Practices that keep these mechanisms effective:

- Content retrieved by the agent is data with an assigned trust level.
- The audit trail covers execution and decisions, never secrets.
- After changing the model, instructions, or integration, rerun the evaluation suite.
- If there is no approval to process data, work on client data stops.
- A return to supervised operation must be possible at any time.

Agent security depends primarily on permissions, isolation, configuration, and enforcement mechanisms. Prompts can support correct behaviour, but should not be the primary security control.

Key takeaway

Filtering model output is not enough — permissions, isolation, and approvals matter.

11. Context, controls, and specification

As agent autonomy increases, trusted context, deterministic checks, and clear acceptance criteria become more important.

Three practices become increasingly important.

1

Context for AI agents

Agents need architectural decisions, coding standards, domain rules, security policies, and repository-specific instructions. Without context, they may generate code that works in isolation but does not fit the system.

2

Checks and guardrails

AI-generated artifacts should pass deterministic controls: tests, static analysis, security scanning, dependency checks, policy validation, and audit controls.

3

Specifications (specification-driven development)

As AI takes on more execution work, requirements and acceptance criteria need to become more precise. If an organization cannot describe and verify the work, an agent cannot execute it reliably.

Documentation becomes infrastructure. Architectural decisions, standards, and acceptance criteria become assets used by both people and tools.

Before increasing autonomy, improve context, controls, and specifications.

Key takeaway

The quality of AI output depends on the quality of the context in which AI operates.

12. Internal developer platform as the foundation for scaling

Bottom-up AI experimentation is a good first step, but it is not enough to scale across an organization. Without shared patterns, teams solve the same problems independently: access to tools, best practices, context, review rules, security controls, data policies, and cost control. This creates fragmentation that a developer platform can reduce.

An AI-enabled internal developer platform can provide:

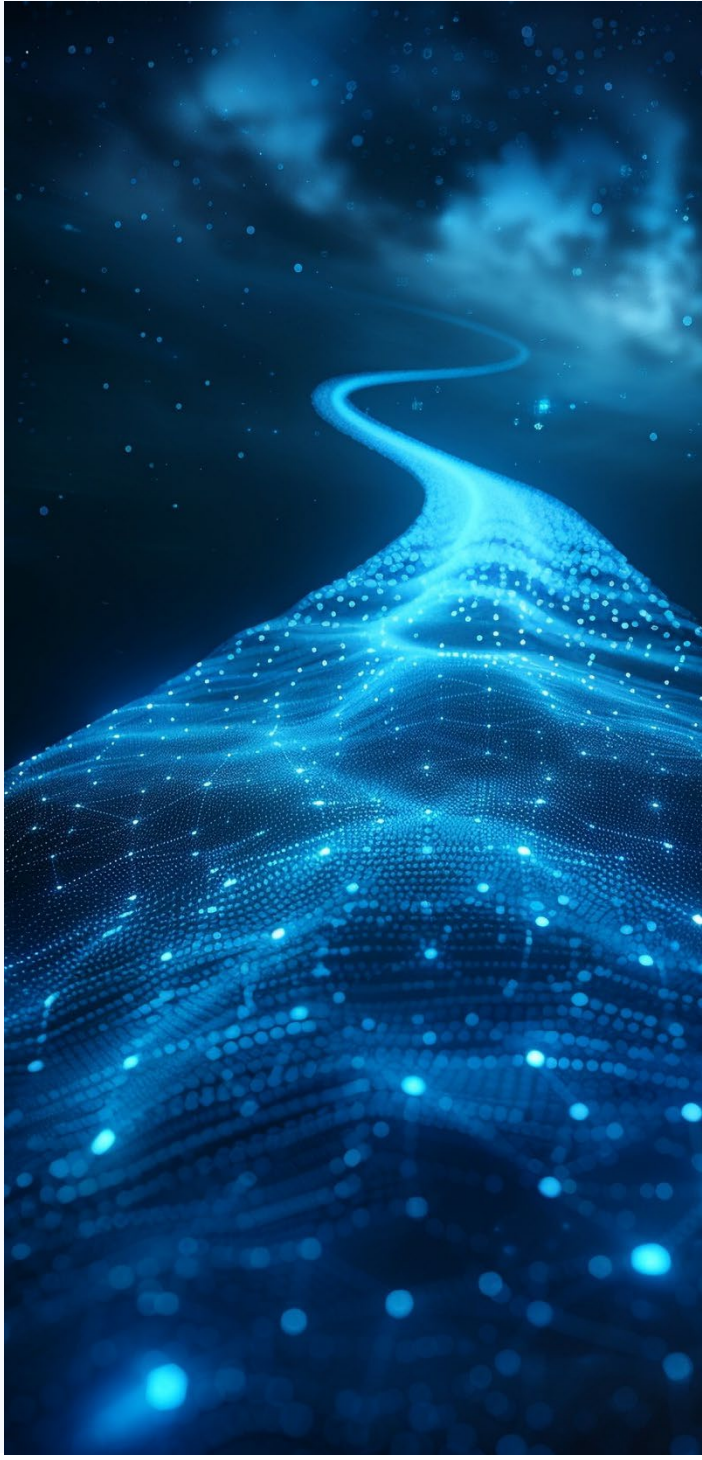
- approved tools and models;
- reusable workflows;
- repository guidance;
- coding and testing standards;
- secure defaults;
- CI/CD controls;
- policy as code;
- audit logs;
- cost monitoring;
- developer portal integration.



The goal is simple: make the safe path the easiest path.

This matters even more for agents. They need controlled access to repositories, issue trackers, pipelines, and documentation. Without a platform layer, agentic workflows become difficult to monitor and govern.

Platform teams can package AI use cases as reusable workflows, e.g., vulnerability triage, test generation, dependency updates, release notes, or documentation updates.



Key takeaway

AI adoption scales when teams receive reusable workflows — not just tool licenses.

13. A 6–12 month action plan for leaders

Leaders do not need to wait for full autonomy.
They need a concise, practical agenda.

0–3 months: Map and focus

- Map where AI is already being used. Include approved tools and unauthorized uses (shadow AI).
- Identify the real bottlenecks. Check code review times, testing delays, security queues, release approvals, and rework.
- Select a small number of use cases that are repeatable, verifiable, and measurable. Good candidates include test generation, code explanation, documentation, requirements refinement, vulnerability triage, and release notes.
- Establish baseline metrics before scaling.

3–6 months: Pilot and govern

- Run controlled pilots. Avoid the highest-risk workflows at the start.
- Define approved tools, data rules, review rules, and requirements checks.
- Create reusable workflows. Do not let every team invent its own approach.
- Measure outcomes, not just token and tool usage.

6–12 months: Scale proven solutions

- Scale proven use cases across the organization.
- Connect AI workflows to repositories, CI/CD, and quality gates.
- Launch bounded agentic workflows where risk is low and outcomes are easy to verify.
- Improve context management, automated checks, and specifications.

Questions to ask now

- Where is AI already being used in our SDLC?
- Which uses are approved, and which are shadow AI?
- Which bottlenecks constrain delivery?
- Which AI use cases reduce waiting time or rework?
- Which outputs require human approval?
- Are AI-generated artifacts verified?
- Is AI adoption supported by a shared developer platform?
- Do we measure AI's impact on delivery?

Key takeaway

The next stage of AI in the SDLC should be practical, controlled, and measured.



Summary:

From experimentation to AI-enabled delivery

The companies that benefit most will be those that connect AI to the full value-delivery flow, quality, security, and business outcomes — not those that simply adopt the latest AI tools first.

AI can reduce repetitive work. It can accelerate analysis, testing, documentation, and defect fixing. But it can also introduce new risks: weak outputs, hidden vulnerabilities, over-reliance on AI, difficulty verifying results, and gaps in regulatory or requirements compliance.

At the same time, agent economics illustrate the scale of the potential. In Sii's experiment, the cost per delivered ticket was USD 6.11 (median: USD 2.96). AI still requires controls and human involvement, but the ability to perform an increasing share of engineering work at such low cost changes the economics of software development.

Sii's approach is practical: start with real SDLC constraints, select concrete use cases, measure outcomes, keep humans accountable, and scale through reusable delivery patterns.



Ready for the next step?

Let's talk about AI in the SDLC

Explore our approach, our services, and how our teams work.



Marcin Mosiołek
AI Competency Center Lead
mmosiolek@sii.pl



Łukasz Biegański
AI Strategist
lbieganski@sii.pl