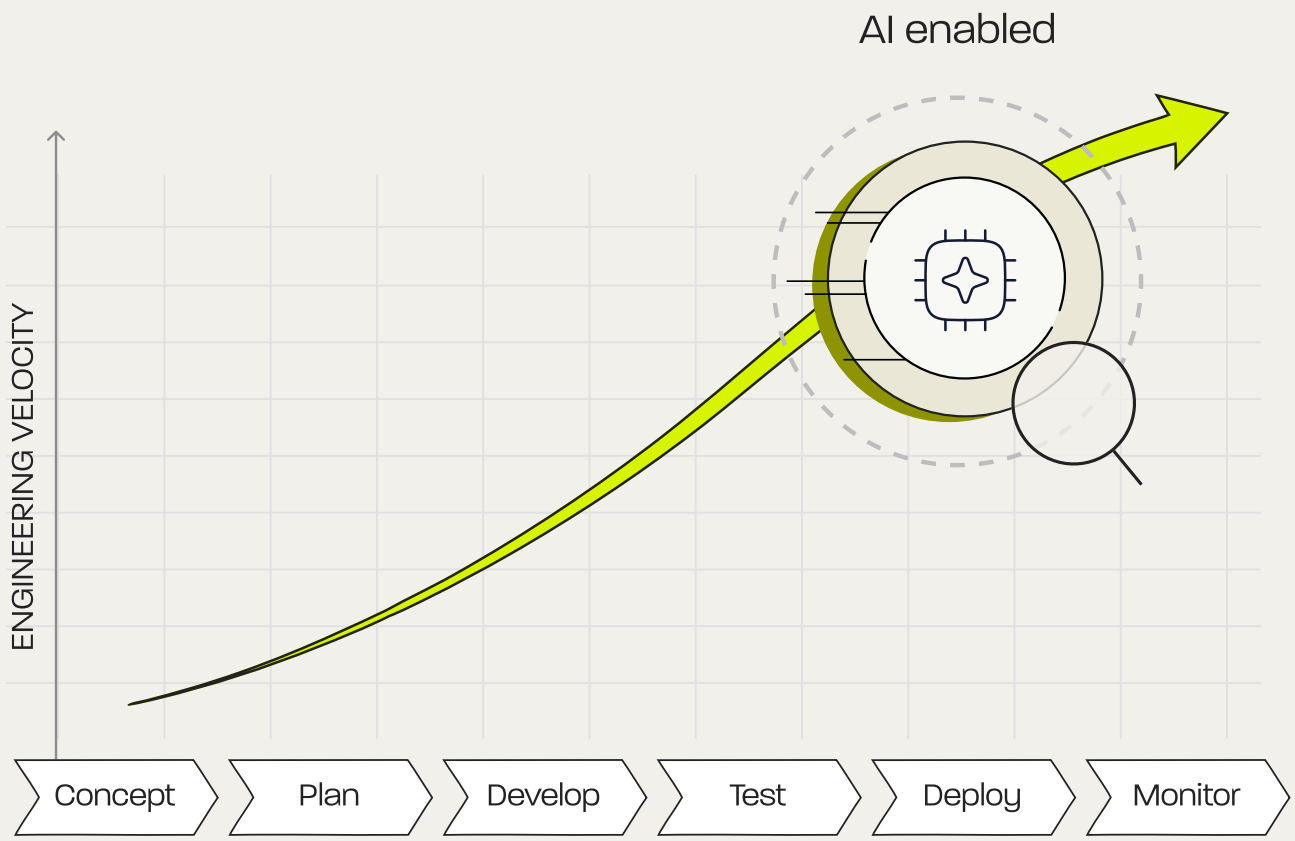
SOFTWARE ENGINEERING EXECUTIVES

AI ROI Playbook

How to measure, model, and realize AI value across the SLDC and production



Resolve.ai

Contents

Chapter 1: The AI ROI Gap in Software Engineering

AI has moved rapidly from experimentation to operational priority for software engineering organizations.

Engineering leaders are deploying coding assistants, copilots, and emerging AI agents that promise to automate portions of the software development lifecycle. Early results have been encouraging. Many teams report that AI-assisted tools help engineers complete routine development tasks faster, generate tests, and navigate complex codebases more easily. These improvements are real and often meaningful.

Yet as AI shifts from something teams experiment with to something organizations invest in at scale, a different question begins to dominate executive conversations: Is this investment producing measurable return?

Across many organizations, the answer remains uncertain.

Research from Gartner reflects this tension. Only 35 percent of software engineering leaders report significant ROI from AI investments in the software development lifecycle.[1] Adoption is rising quickly, but the discipline for measuring value has not kept pace.

This gap is not unusual in fast-moving technology cycles. New capabilities often appear before organizations develop consistent ways to evaluate their economic impact. What makes the AI moment unusual is the speed of adoption, driven by both top-down and bottom-up initiatives.

[1] Gartner, "6 Trends Driving Technology Adoption ROI in Software Engineering," 2026.

AI adoption has accelerated faster than ROI discipline

The adoption curve for AI has been steep, and that pace changes the burden of proof.

McKinsey's 2025 global survey found that 88 percent of organizations are now using AI in at least one business function, up significantly from prior years.[2] Bain reports that AI has become a top-three strategic priority for 74 percent of companies.[3]

When adoption grows this quickly, AI stops being treated as an experiment. It becomes a budget line item that must withstand scrutiny from finance, procurement, and, often, the board. This is where many organizations encounter friction.

AI does not behave like a conventional enterprise software purchase. Costs fluctuate with usage. Capabilities evolve quickly as models improve. Value often appears indirectly through workflow improvements rather than through a clean revenue line.

Industry analysis reflects this tension between expectations and maturity. Gartner's Hype Cycle for AI in Software Engineering notes that many AI capabilities are still evolving and often automate narrow development tasks rather than transforming the full software lifecycle.[4] As a result, organizations may see localized productivity improvements while still struggling to demonstrate broader operational or financial impact. Gartner makes a similar observation from a financial management perspective: AI systems require AI-specific financial management practices because unmanaged cost growth and unclear value attribution are common failure modes.[5]

In other words, the industry is not short on AI activity. It is short on repeatable methods for measuring AI ROI.

[2] McKinsey & Company, "The State of AI in 2025: Agents, Innovation, and Transformation," November 2025.

[3] Bain & Company, "Executive Survey: AI Moves from Pilots to Production," Q3 2025.

[4] Gartner, Hype Cycle for AI in Software Engineering, 2025.

[5] Gartner, "AI's Elusive ROI: How to Track, Manage, and Demonstrate the ROI of AI Investments," 2026.

> 88% of organizations are now using AI in at least one business function. Only 35% report significant ROI from those investments.
>
> — McKinsey & Gartner, 2025–2026

The Productivity Lens

Part of the difficulty comes from how engineering productivity is typically measured.

Most ROI discussions in software engineering focus on the part of the lifecycle that is easiest to observe: the development environment. Build-side productivity improvements are measurable. Teams can estimate time saved on routine development tasks, compare pull request cycle times, and gather developer feedback about perceived productivity.

These gains are valuable. In many organizations, they justify the adoption of coding assistants and development tooling on their own. But they measure only one portion of the engineering system. Software engineering work does not end when code is shipped.

The Operational Blind Spot

Once software reaches production, engineering work shifts toward maintaining and understanding complex systems.

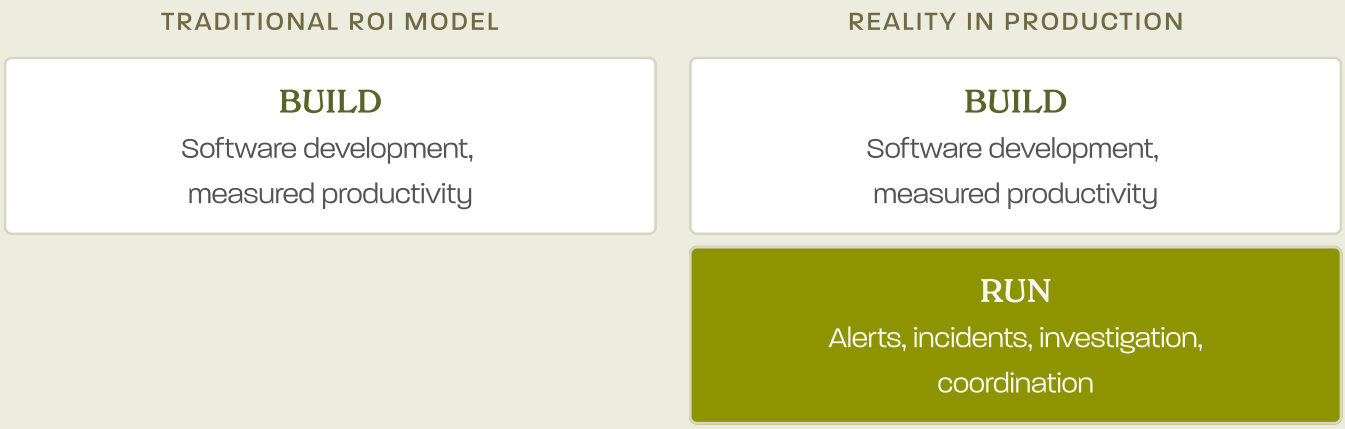
Engineers investigate alerts, diagnose incidents, analyze telemetry across services, and coordinate responses across teams when reliability is at risk. These activities are essential to maintaining stable production environments, but they rarely appear clearly in AI ROI models.

Industry research reinforces how much engineering effort goes into work outside of writing new code. An IDC report summarized by InfoWorld found that developers spend more time on operational and background tasks than on coding itself.[6]

[6] InfoWorld, "Developers Spend Most of Their Time Not Coding -- IDC Report," 2025.

Organizations measure improvements in how quickly software is created while leaving the operational cost of understanding and stabilizing that software largely outside the model. When those operational workflows remain unchanged, improvements in development productivity alone often fail to deliver the ROI leadership teams expect. In some environments, faster development can even increase operational load by introducing more changes into systems that must be monitored and supported.

Where Engineering Time Actually Goes



Operational work is not a footnote. It is half the system.

The speed of creation versus the speed of understanding

Software engineering involves two fundamentally different kinds of work. The first is creating software. The second is understanding complex systems once they are running. When systems fail or behave unexpectedly, engineers are rarely blocked by their ability to write code. They are blocked by their ability to assemble evidence, form a reliable hypothesis about what is happening, and coordinate decisions across multiple teams and tools.

This gap is becoming more pronounced as AI accelerates software creation. Code is now being produced and shipped at volumes that exceed what engineers can realistically maintain deep familiarity with. As more AI-generated code enters

production systems, the amount of implementation detail any individual engineer can hold in their head inevitably shrinks. The result is a growing gap between how quickly systems are created and how quickly teams can reliably understand them when something breaks.

This work is investigative by nature. Engineers must correlate logs, metrics, traces, infrastructure signals, and recent changes across distributed systems. The time between an alert appearing and a team reaching a confident decision can expand quickly when that evidence is fragmented across tools and teams. As this window grows, incident response teams grow as well. More engineers are pulled in. Outages last longer, compromising SLOs. In practice, this evidence latency becomes one of the most expensive hidden costs of operating modern software systems. Yet it rarely appears in traditional ROI calculations.

> **The Emerging Operational Risk**
>
> AI is accelerating the speed at which software is created. But the speed at which engineers can understand production systems has not increased at the same rate.
>
> As more AI-generated code enters production, the gap between creation and comprehension grows.
>
> The result is a rising operational burden when systems fail.

Why Traditional ROI Models Struggle with AI Systems

The financial models typically used to evaluate technology investments were not designed for systems that behave like AI.

Traditional enterprise software behaves predictably once deployed. Organizations purchase a system, deploy it across users, and expect relatively stable costs and benefits over time. AI systems behave differently. Their capabilities evolve rapidly as models improve. Their costs fluctuate based on usage patterns. Their outputs are

probabilistic rather than deterministic. And because both cost and value shift as organizations learn how AI interacts with real workflows, a single ROI calculation performed during procurement rarely captures these dynamics.

The Role of Pilots in Proving AI Value

Another challenge appears when organizations try to evaluate AI value too broadly, too early.

Not every engineering workflow is equally well suited to a short proof period. Some, such as alert investigation and triage, occur often enough to generate meaningful evidence quickly. Others, such as major incident response or multi-team remediation, are less frequent, more variable, and harder to evaluate within a narrow time window.

That does not make early pilots misleading. It means they should be scoped to the workflows where value is most clearly and repeatedly observed. In practice, the strongest pilots do not attempt to prove everything at once. They establish credibility in high-volume investigative workflows first, then expand into more complex operational use cases as adoption, integration depth, and organizational confidence grow.

A More Complete View of AI ROI

Taken together, these dynamics explain why many organizations struggle to demonstrate AI return even when engineers find the tools useful. Developer productivity metrics capture only part of the engineering lifecycle, while traditional financial models struggle to account for the evolving economics of AI systems.

The next chapter introduces a framework for closing that gap, showing how engineering leaders can measure AI impact across the full lifecycle of software systems and translate those improvements into financial outcomes that leadership teams can defend

Chapter 2: The Full-Stack AI ROI Framework for Software Engineering

Engineering leaders need a framework that measures the full lifecycle of software systems. Software organizations do not create value in only one place. Engineers build systems, but they also operate them. Yet most AI ROI discussions measure only one side of that equation.

A more complete view begins by recognizing that engineering organizations operate across two distinct domains.

Build-Side ROI: Improving Software Delivery

Build-side value refers to improvements that occur during the development phase of the software lifecycle.

AI systems can assist engineers in writing new code, modifying existing functionality, generating tests, and reviewing pull requests. These capabilities can reduce the time required to complete common development tasks and help engineers work more efficiently with complex codebases.

Typical metrics used to measure build-side improvements include development cycle time, pull request review duration, test generation coverage, onboarding speed to unfamiliar systems, and feature delivery velocity.

For many organizations, these improvements represent the first visible form of AI value in engineering. They are relatively easy to pilot and measure, and often the first place leadership sees evidence that AI is helping engineers work faster. But build-side improvements represent only part of the economic picture.

Run-Side ROI: Improving Production Operations

Run-side value focuses on the operational side of the engineering lifecycle.

Once software enters production, engineering work shifts toward maintaining system reliability and responding to unexpected behavior. Engineers investigate alerts, analyze telemetry, diagnose incidents, and coordinate actions that stabilize systems under pressure.

AI systems capable of reasoning across operational signals can accelerate this investigative process. Instead of manually assembling context across logs, metrics, traces, deployment histories, and infrastructure signals, engineers can rely on AI systems to help correlate evidence and surface the most relevant information.

Key metrics that capture run-side improvements include time required to reach a confident understanding of a system issue, mean time to resolution during incidents, number of engineers involved in incident response, escalation frequency, and change failure rates. In many organizations, this is where the largest untapped efficiency opportunity exists.

Why Both Domains Matter

When engineering leaders evaluate AI across both build-side and run-side domains, they gain a clearer view of where meaningful economic value actually appears.

The important insight is that AI influences the economics of engineering organizations in multiple places across the lifecycle of software systems. A team can become faster at producing code while remaining slow at understanding production systems. In that scenario, cost is not removed from the system. It is redistributed within it. A full-stack framework prevents that blind spot.

The important insight is that AI influences the economics of engineering organizations in multiple places across the lifecycle of software systems. A team can become faster

at producing code while remaining slow at understanding production systems. In that scenario, cost is not removed from the system. It is redistributed within it. A full-stack framework prevents that blind spot.

The important insight is that AI influences the economics of engineering organizations in multiple places across the lifecycle of software systems. A team can become faster at producing code while remaining slow at understanding production systems. In that scenario, cost is not removed from the system. It is redistributed within it. A full-stack framework prevents that blind spot.

> Build-side AI improves how quickly teams create software. Run-side AI improves how quickly teams understand and stabilize software in production.
>
> Most organizations measure only one. The strongest ROI case requires both.

The AI ROI Framework for Software Engineering

AI value in engineering appears across the full lifecycle of software systems. Measuring ROI requires connecting improvements in engineering workflows to measurable operational and financial outcomes. The framework in this playbook follows a simple sequence.

The AI ROI Framework for Software Engineering

SOFTWARE DELIVERY

BUILD SIDE
Code generation · Test generation · PR review · Developer productivity

PRODUCTION OPERATIONS

RUN-SIDE
Alert triage · Incident investigation · Root cause analysis · Telemetry correlation

WHERE ENGINEERING TIME IS MEASURED

OPERATIONAL BASELINE
Incidents per month · Investigations per day · Engineers per incident · Investigation duration

ENGINEERING HOURS RETURNED

FINANCIAL IMPACT
Engineering hours returned · FTE capacity · Salary-equivalent value · Payback period

BUILD INTERNALLY

DELIVERY DECISION
Build internally · Adopt a platform

Build-side value (software delivery): AI improves the speed at which teams create software, including code generation and modification, test generation and coverage, pull request review cycles, developer onboarding and codebase navigation, and feature delivery velocity.

Operational baseline (where ROI is measured): Production investigations and incidents become the measurable unit of engineering effort, using inputs such as incidents per month, alerts or investigations per day, investigation duration, and engineers involved per investigation.

Financial translation: Workflow improvements translate into returned engineering capacity, expressed as engineering hours returned, full-time equivalent capacity, salary-equivalent value, and payback period and ROI.

Delivery decision: Once ROI becomes visible, organizations determine how the capability will be delivered: build internally and own the system lifecycle, or adopt a platform and leverage shared infrastructure and tooling.

This sequence connects engineering reality to financial decision-making. It begins with the workflows engineers already perform and ends with the economic choices leadership teams must make.

The next chapter shows how to translate operational improvements into financial terms that executives and finance teams can evaluate.

Chapter 3: Translating Engineering Metrics into Financial ROI

Understanding where AI creates value in software engineering is only the first step. The next challenge is translating improvements in engineering workflows into financial terms that executives, finance teams, and procurement organizations can evaluate.

Engineering organizations naturally measure performance using operational metrics: mean time to resolution, deployment frequency, change failure rate, and the number of alerts or incidents handled each week. These metrics provide insight into how systems behave and how efficiently teams operate. But they rarely translate directly into business outcomes without a connecting layer.

Starting with Operational Baselines

The most defensible ROI models begin with operational baselines that reflect how engineering work is actually performed.

Production systems generate a steady stream of operational signals. Alerts fire. Incidents occur. Systems behave unexpectedly. Each of these events requires investigation before engineers can determine what action is safe to take. In most enterprises, a backlog of uninvestigated alerts accumulates as system noise and, over time, becomes incidents.

That investigative work consumes a measurable amount of engineering time. Engineers gather logs and metrics, correlate telemetry across services, review recent deployments, and coordinate with teams responsible for adjacent systems. Because these activities recur throughout the lifecycle of a system, they provide a reliable baseline for measuring engineering effort.

This is why many organizations model AI value using operational inputs such as incidents per month, alerts or investigations per day, average investigation duration, and engineers involved per investigation. When AI systems reduce the time required to investigate or diagnose system behavior, those improvements translate directly into engineering capacity returned.

Converting Engineering Effort into Economic Value

Once operational baselines are established, the next step is translating engineering effort into financial terms.

The most common approach is to calculate the value of engineering capacity returned using fully loaded hourly cost. If an engineering organization returns several thousand hours of investigative work per year through faster triage or root cause analysis, those hours represent capacity that can be redirected toward other engineering priorities. This capacity is expressed as engineering hours returned, full-time equivalent value of

engineering time.

Finance leaders may not track incident response metrics, but they understand labor cost and capacity. Translating operational improvements into those terms allows engineering leaders to frame AI investments in language that aligns with budget decision-making.

Hard Cost First, Opportunity Cost Second

This model is intentionally conservative.

In most software organizations, the engineers pulled into investigative work are not a separate operations team. They are application engineers, platform engineers, and site reliability engineers who are also responsible for delivering new product features, improving infrastructure, and advancing reliability initiatives. Returning their time is not about reducing headcount. It is about restoring capacity for the work the organization actually needs them to do.

When those teams spend significant time investigating incidents and alerts, the business absorbs costs that rarely appear in ROI models: product work slows, platform improvements are delayed, reliability investments compete with incident response, and engineers spend time firefighting instead of building.

AI does not change the number of engineers an organization needs. It changes what those engineers can accomplish. The financial model in this playbook captures the hard cost of engineering time returned. Faster product delivery, greater reliability, and compounding improvements to platform quality are additional benefits. If the investment produces positive economics based solely on engineering capacity returned, everything else becomes upside.

> **This Is Not a Workforce Reduction Exercise**
>
> The goal of AI in engineering is not to replace engineers.
>
> It is restoring engineering capacity currently consumed by operational toil: investigations, incident response, and system triage.
>
> When that capacity returns, teams can invest it in shipping product, improving reliability, and strengthening the platform.

This approach gives engineering leaders a defensible financial model that can withstand scrutiny while remaining grounded in the real work engineers perform. The next chapter examines the ownership decision that follows: whether to build this capability internally or adopt a platform that delivers it.

Chapter 4: Build vs. Buy: The Ownership Decision Behind AI Systems

Once the ROI case is credible, the next question usually appears quickly. Should we build this ourselves or buy it?

At first glance, this appears to be a technical comparison. Teams evaluate models, agent frameworks, orchestration tools, and integration capabilities. Engineering teams naturally believe they can assemble something internally. After all, building software is what they do.

But this framing misses the real decision.

Build versus buy is an ownership decision. The question is not whether an engineering team can assemble a working system. It is whether the organization is prepared to own the ongoing work required to build it into something reliable, operate it under real production conditions, and continuously improve it as the environment changes. A useful way to understand that commitment is through three ongoing loops: Build, Operate, and Improve.

Loop 1: Build the Product

The first loop is building the product. In production engineering environments, that means integrating with observability platforms, incident management systems, cloud infrastructure, CI/CD tooling, and internal knowledge bases. It also means building the orchestration layer that allows the system to retrieve context, reason across signals, and produce outputs that engineers trust during real incidents.

This work is substantial, but it is also the most visible part of the effort. Teams can see progress quickly, and the system starts to resemble a real product. That visibility is one reason internal builds seem attractive early on. But building a prototype is not the same as operating a production system.

Loop 2: Operate It as a Real System

Once deployed, the system becomes another production service that must be monitored and maintained. Operating it means managing inference latency and uptime, token consumption and model cost, retrieval quality and context discipline, system observability, and resilience when dependencies fail or degrade.

Teams quickly discover that access to telemetry is not the same as extracting signal from it. Logs can overwhelm context windows. Metrics can be noisy without system awareness. Traces are difficult to interpret without preserving execution context. Workflows that look effective in a demonstration environment may behave differently under the variability of real production incidents. At this stage, the organization has effectively created a new production service that requires ongoing operational ownership.

Loop 3: Improve the System Continuously

This is the most underestimated loop.

Behavior evolves constantly. Changes to prompts, workflows, tools, or models can

improve one class of behavior while degrading another. Systems that perform well on simple cases may struggle with more complex investigations. Without structured improvement disciplines, teams make changes blindly.

This is why evaluation becomes a core mechanism for reliability. Organizations need evaluation frameworks, regression testing, governance and change controls, and systematic iteration across incident types. Synthetic benchmarks overstate readiness. Real reliability requires population-level measurement across many incident types, services, and teams.

The challenge also grows with scale. A single engineer using these tools may achieve good results through experimentation. A multi-team organization relying on the same system during high-severity incidents requires much stronger reliability guarantees. More agents, more tools, or more prompts do not automatically improve outcomes. Without disciplined improvement practices, the system can become harder to trust over time rather than easier.

The Three Loops of AI System Ownership

Any production AI system must sustain three continuous loops:

Build — Creating the capability and integrating it into real workflows.

Operate — Running the system reliably under production conditions.

Improve — Continuously evaluating and refining the system as models and environments evolve.

Organizations choosing to build internally must fund all three.

What DIY Ownership Really Means

These three loops clarify what an internal build truly involves. A DIY path does not mean building the initial capability. It means sustaining the entire system lifecycle, including the orchestration layer, telemetry retrieval infrastructure, evaluation and regression frameworks, governance and reliability controls, and ongoing upgrades as

models and tools evolve.

This is not building a feature. It is funding a long-term operating model.

Some organizations will decide that ownership is strategically worthwhile. If the capability represents core intellectual property and the organization is prepared to fund the operating burden, building internally can be justified. But the threshold is not whether the team can build a prototype. It is whether the organization is prepared to sustain Build, Operate, and Improve as permanent responsibilities.

The Financial Side of the Decision

An internal build requires funding for the full lifecycle of the capability: product development, infrastructure and model usage costs, operational reliability work, and continuous improvement.

A platform model distributes part of that burden across many customers. The vendor invests in shared infrastructure, evaluation frameworks, orchestration tooling, and reliability practices that individual organizations would otherwise need to build themselves. The internal team focuses primarily on integration, governance, and adoption within their environment.

From a financial perspective, the decision is less about whether a team can build a reasonable prototype and more about where the ongoing cost structure should sit. The next chapter examines how these dynamics play out across industries and where AI ROI typically first emerges.

Chapter 5: Industry Examples: Where AI ROI Appears Across Different Environments

The framework for measuring AI ROI in software engineering is broadly consistent across industries. Engineering organizations still build software, operate production systems, and allocate limited engineering time across competing priorities.

What changes across industries is where operational complexity concentrates and how quickly failures translate into business impact. The question is not whether one industry is more complex than another. The question is where engineering time is currently being consumed and what happens when AI reduces the time required to understand production systems under pressure.

Financial Services

In financial services environments, incidents often become expensive before they become obvious.

A spike in card authorization declines or a delay in settlement processing may initially appear to be a localized service issue. In reality, the failure domain may span internal policy changes, upstream network behavior, third-party payment processors, and customer-facing applications.

The technical problem is only part of the burden. Engineering teams must also establish why they believe a specific failure path is correct, preserve an evidence trail, and support risk, audit, and compliance stakeholders who require defensible explanations after the fact.

AI creates value here by helping teams correlate evidence across fragmented systems and ownership boundaries more quickly, moving faster toward a decision-
AI creates value here by helping teams correlate evidence across fragmented

systems and ownership boundaries more quickly, moving faster toward a decision-ready picture of what is happening and what action is safe to take. The resulting value typically appears through faster isolation of the true failure domain, fewer escalations across engineering and risk teams, stronger incident timelines for post-incident review, and reduced effort spent establishing confidence in the diagnosis.

Enterprise SaaS

In enterprise SaaS environments, the expensive part of incident response is often not remediation itself but determining what to trust.

A routine deployment may trigger latency regressions across several services simultaneously, creating an alert pattern that appears broader than the actual fault domain. Multiple teams begin investigating. Some signals represent primary causes. Others are downstream symptoms. Recent code changes, configuration updates, infrastructure shifts, and feature flags all complicate the picture.

This is where engineering time becomes trapped. The challenge is not simply that systems are distributed. It is that narrowing the search space is expensive. AI creates value by helping teams move from "something is wrong somewhere" to a clearer hypothesis about the most likely failure path and the safest next action. That shift produces lower mean time to resolution, smaller incident response teams, less time coordinating across service owners, and more confident decisions under pressure.

Digital Commerce and Large-Scale Retail Platforms

In digital commerce environments, operational failures translate into business impact very quickly.

A checkout slowdown during a major promotion or a payment error during peak traffic can quickly become a revenue event. Engineering teams responding in these moments must reason across multiple layers simultaneously: frontend and client

behavior, backend services and APIs, payment gateways and third-party integrations, and inventory and fulfillment systems. The challenge is not only identifying what is broken. It is isolating the revenue-impacting failure path quickly enough to minimize business impact.

AI creates value by accelerating cross-system triage and helping engineers determine where the real failure is occurring, resulting in faster isolation of checkout or payment failures, shorter revenue-impacting incidents, fewer escalations between product and infrastructure teams, and quicker stabilization during peak traffic windows.

Different Industries, Different Cost Signatures

These examples highlight three distinct operational cost signatures.

In financial services, the dominant cost is evidence cost: significant engineering effort required to assemble defensible explanations under regulatory pressure. In enterprise SaaS, the dominant cost is coordination: narrowing distributed signals into a reliable failure path across multiple service owners. In digital commerce, the dominant cost is revenue exposure: technical ambiguity that can quickly translate into lost business during high-traffic events.

Most organizations experience some mixture of these patterns, but one usually dominates. Understanding which one helps leaders identify where AI is most likely to create measurable value first, making the ROI conversation much easier to ground in operational reality.

Operational Cost Signatures

Different industries experience operational toil in different forms:

Financial services → Evidence cost

Enterprise SaaS → Coordination cost

Digital commerce → Revenue exposure

Understanding which dominates in your environment helps identify where AI can create measurable value first.

Chapter 6: Hypothetical Case Study: Modeling AI ROI

The goal of a case study is not to predict a universal outcome. It is to demonstrate how an engineering leader can move from operational reality to a defensible economic model.

Consider a hypothetical company with approximately 250 engineers operating a distributed platform that supports customer-facing applications. It ships continuously, runs a broad production surface area, and experiences roughly 150 incidents per month. Together, these incidents create a recurring operational burden. Engineers are pulled into triage, evidence gathering, telemetry review, and coordination across services before they can determine what is happening and what action is safe to take.

The case for AI does not begin with abstract productivity claims. It begins with the observation that a measurable share of engineering time is already being consumed by understanding production systems.

The Baseline: How Much Engineering Time Is Being Consumed Today?

Assume the following baseline:

- Average responders per incident: **6 engineers**
- Average time to resolution: **90 minutes**

Using those inputs:

150 incidents x 6 engineers x 1.5 hours = **1,350 engineering hours per month**

1,350 x 12 = **16,200 engineering hours per year**

That number matters because it turns operational effort into something leadership can reason about. This is not invisible toil. It is a recurring investment of engineering capacity required simply to keep the system stable.

The Incident ROI Formula

Annual Engineering Hours Consumed
Incidents/month x Engineers/incident x Duration (hrs) x 12

Engineering Capacity Available to Return
Apply improvement assumption (20%, 40%, or 60%)

Annual Productivity Value
Hours returned x fully loaded engineering cost

The Intervention: What Changes If AI Accelerates Investigation?

Introduce AI into the model conservatively. Assume the system does not eliminate incidents and does not replace human decision-making. Instead, assume it improves the parts of the workflow where time is most often lost: correlating telemetry across systems, surfacing relevant evidence faster, narrowing the likely failure domain, and summarizing the investigation's current state.

If those capabilities reduce investigation time by 40 percent, average resolution time falls from 90 minutes to 54 minutes.

Monthly engineering hours saved: 150 x 6 x 0.6 hours = **540 hours saved per month**

Annual engineering capacity returned: 540 x 12 = **6,480 hours returned per year**

AI does not need to automate the entire workflow to create material value. It only needs to reduce the time required to understand and respond.

Translating Capacity into Financial Value

Apply a conservative engineering cost estimate of $120 per hour:

6,480 hours x $120 = **$777,600 in annual productivity value**

This figure reflects only the direct engineering labor cost associated with incident response. It does not account for the broader opportunity cost created when application, platform, and site reliability engineers spend time investigating production issues instead of building new features, improving infrastructure, or advancing reliability initiatives. In many organizations, those are the same teams. The financial model captures only the hard cost. The strategic upside of returning that capacity to higher-value work is additional.

What This Case Study Demonstrates

A defensible AI ROI model in software engineering follows a consistent sequence: start with a real operational burden, quantify the current engineering effort required to manage it, model a realistic workflow improvement, translate that improvement into engineering capacity returned, and express that capacity in financial terms.

This approach makes the business case understandable to finance and procurement teams while keeping the argument grounded in the actual work engineers perform.

Chapter 7: Build vs. Buy Financial Model

Once engineering leaders see the value side of AI beyond software development, the next question becomes unavoidable.

If the operational gains are real, what is the most rational way to deliver the capability?

At this point, many organizations drift back into a technical debate. Teams compare models, frameworks, and agent tooling. They ask whether their internal platform team could put something similar together. Recent advances in AI tooling have made it possible to assemble early prototypes faster than ever, enabling even junior engineers to produce working demos quickly. They evaluate vendor solutions and try to determine which appears most capable.

But the real decision is not technical. It is economic.

The question is not whether an engineering team can build something that works. The question is whether the organization wants to own the full cost structure of building, operating, and continuously improving a production-grade system, or whether it prefers to adopt a platform that absorbs part of that burden across a broader customer base.

A financial model helps make that choice concrete.

Start with the Value Side of the Equation

Before comparing build and buy, leaders must establish the value the system is expected to create. As shown in the case study in the previous chapter, the process starts with the operational baseline. But the build-versus-buy decision requires a second layer of questions: not just what the value is, but what it actually takes to deliver that value reliably in production.

On the value side, the questions are fairly straightforward:

- How many incidents or investigations occur each month? Is that number trending upward or declining?
- How much engineering time do they consume today?
- How much of that effort could realistically be reduced with assistance?

These questions anchor the model. Without them, the build-versus-buy discussion turns into opinion.

What the Cost Side Actually Involves

This is where most organizations underestimate the commitment.

A frontier model can serve as one layer in the stack. But production incident investigation requires a broader operational system around it. That system must integrate into the workflows engineers already use, coordinate evidence across logs, metrics, traces, deployments, code, and internal knowledge, support shared multiuser investigations, ground its outputs in operational evidence engineers can trust, and improve over time as the environment changes.

Building that system is not a project. It is an internal product.

And the environment it operates in is not stable. Models change. Prompting patterns evolve. Toolchains shift. New failure modes emerge as coverage expands. What works on a narrow set of alerts in a pilot rarely generalizes cleanly to the long tail of complex, ambiguous production incidents where signal is sparse, and noise is high.

The real cost model has three loops that never close.

1. Build covers the initial system. This includes integrations across telemetry, infrastructure, code, deployments, and knowledge sources, orchestration logic for multistep investigations, security, permissions, and auditability, and a workflow experience that fits how incidents are actually run.

2. Operate covers what happens once the system is live. This includes latency and reliability during real incidents, token efficiency and cost control at scale, monitoring and debugging of the system itself, and on-call readiness for platform failures.

3. Improve is the loop most organizations underestimate. Every prompt, tool, skill, and model change can regress behavior. Without a formal evaluation program, changes are made blindly. Without governance, teams diverge. Without systematic iteration, the system stalls at an early performance plateau and loses engineering trust.

> **The Build vs. Buy Misconception**
>
> The real question is not: Can we build something that works?
>
> The real question is: Do we want to own the full lifecycle of building, operating, and improving this system for years?

Why the Improvement Loop Is So Hard

The hardest part of improvement is evaluation.

Production failure modes are not obvious. Short tests and synthetic benchmarks can overstate readiness. Real reliability has to be measured across many incident types, services, and teams, not just on the cases that are easiest to test. A system that

looks strong on simple alerts can break on ambiguous incidents where signal is sparse and multiple failure hypotheses compete.

Telemetry adds another layer of difficulty. Access to observability data is not the same as reliably extracting signal from it. Metrics overwhelm models without stack context. Log retrieval breaks on cost and context limits without just-in-time slicing. Traces require trace-aware workflows to follow execution paths, especially when logs are missing.

This is why a general-purpose model with tool calling can handle narrow, engineer-initiated tasks but struggle to generalize across the variability of real production environments.

Modeling the Cost of Building Internally

A realistic internal model must account for three categories of cost.

Product development costs include the effort required to create the initial system. Typical inputs include:

- engineering time to build the core system
- integrations with observability and incident systems
- workflow and orchestration design
- interface development and internal enablement

Operating costs apply once the system is deployed. It becomes another production service that must be monitored and maintained. Operating costs often include:

- model usage and token consumption
- inference and infrastructure costs
- monitoring and reliability ownership
- telemetry retrieval and data access discipline
- uptime, latency, and resilience requirements

Continuous improvement costs are ongoing. These systems do not remain static once deployed. Improving and maintaining them requires:

- evaluation frameworks
- regression testing
- prompt and workflow updates
- model upgrades
- governance and change control
- maintenance as tools and APIs evolve

Taken together, these categories represent the full lifecycle cost of owning the capability internally.

Modeling the Cost of Adopting a Platform

The economics of a platform model look different.

Organizations adopting a platform still incur cost, but the structure changes. Instead of funding every layer of the system directly, they pay for access to a capability whose infrastructure, evaluation methods, telemetry tooling, and operational discipline are shared across many customers. That shared investment creates a multiplier effect on advancement, because improvements made once can benefit the broader customer base.

Typical cost inputs include:

- platform licensing or usage fees
- implementation and integration effort
- internal ownership for rollout and governance
- ongoing administrative or operational support

The difference is not the absence of cost. It is the distribution of cost.

A platform provider absorbs a meaningful portion of the burden of building, operating, and improving. The internal team focuses primarily on integration, governance, and adoption within its own environment.

Comparing the Outputs That Matter

Once both delivery paths are modeled, the comparison becomes much clearer.

A practical build-versus-buy financial model should produce four outputs:

- **Annual ROI:** The expected return relative to the cost of delivering the capability.
- **Payback period:** How long it takes for the value created by the system to exceed the cost of the investment.
- **Break-even threshold:** The minimum operational improvement required for the investment to make economic sense.
- **Sensitivity scenarios:** How the model behaves under different assumptions about adoption, usage, or operational improvement.

Sensitivity scenarios are particularly important because these systems rarely operate under perfectly stable assumptions. Adoption ramps unevenly. Usage grows over time. Model costs and infrastructure requirements evolve.

A model that works only under ideal assumptions is not much of a model.

What the Financial Model Is Really For

The decision criteria that usually settle the question are simpler than the technical comparison suggests.

· Is this capability core intellectual property that the organization will fund as an internal product for years?

· Can the organization staff a dedicated team to sustain Build, Operate, and Improve continuously?

· Does it have the evaluation discipline to measure reliability and prevent regressions as the incident volumes grow?

· Can it deliver access parity, governance, and auditability without creating a second system to maintain?

If the honest answer is that the organization wants production-grade outcomes without becoming a platform team, buying is usually the rational choice. If the capability is genuinely strategic and the organization is prepared to fund it like a product with eyes open, building it can be justified.

But the threshold is not whether the team can build a working prototype. Early wins come quickly. The threshold is whether the organization can sustain the system through the long tail, where incidents are harder, the environment is changing, and the gap between a demo and a production-grade system becomes visible.

The next chapter provides a practical worksheet that engineering leaders can use to build this model using operational metrics most organizations already track.

Chapter 8: AI ROI Worksheet for Engineering Leaders

Most engineering leaders do not need a perfect financial model on the first attempt. What they need is a model credible enough to support a real conversation with finance, procurement, and executive leadership.

This worksheet makes that process straightforward. Work through the four steps below using your organization's actual numbers. A completed example follows each step.

Step 1: Establish Your Operational Baseline

INPUT	YOUR ORGANIZATION	EXAMPLE
Monthly incident volume	_____	150 incidents
Average responders per incident	_____	6 engineers
Average time to resolution (hours)	_____	1.5 hours
Fully loaded engineering cost (per hour)	_____	$120

Monthly incidents x average responders x average resolution time = monthly engineering hours consumed

_____ x _____ x _____ = _____ hours per month

_____ hours per month x 12 = _____ annual hours consumed

Example: 150 x 6 x 1.5 = 1,350 hours per month. 1,350 x 12 = 16,200 annual hours consumed.

This is the operational baseline. It represents a recurring investment of engineering capacity required simply to keep production systems stable.

Step 2: Estimate Realistic Workflow Improvements

Estimate how much faster investigation workflows could move if AI helps correlate telemetry, surface relevant evidence, narrow the failure domain, and summarize the current state of an investigation. Assumptions should be modest and defensible rather than aggressive.

SCENARIO	INVESTIGATION TIME REDUCTION	DESCRIPTION
Conservative	20%	Assistance with evidence gathering; engineers drive most of the analysis
Moderate	40%	Meaningful acceleration of triage, correlation, and hypothesis formation
Aggressive	60%	Most investigative groundwork is handled automatically; engineers validate and decide

Your assumption: _____ % reduction in investigation time

A 40 percent reduction is a reasonable moderate assumption for planning purposes. Even a 20 percent improvement at scale can produce meaningful financial impact.

Step 3: Calculate Engineering Capacity Returned

Time saved per incident = average resolution time x improvement percentage

_____ hours x _____ % = _____ hours saved per incident

Monthly hours saved = monthly incidents x average responders x hours saved per incident

_____ x _____ x _____ = _____ hours saved per month

Annual hours returned = monthly hours saved x 12

_____ x 12 = _____ hours returned per year

Annual productivity value = annual hours returned x fully loaded engineering cost

_____ hours x $_____ = $_____

Example (40% improvement assumption):

· Time saved per incident: 1.5 x 40% = 0.6 hours saved per incident

· Monthly hours saved: 150 x 6 x 0.6 = 540 hours per month

· Annual hours returned: 540 x 12 = 6,480 hours per year

· Annual productivity value: 6,480 x $120 = $777,600

This calculation captures only direct engineering labor. It does not include the broader opportunity cost of engineering teams spending time on incident response instead of building product, platform, and reliability improvements. If the investment produces positive economics on this basis alone, the strategic upside is additional.

Step 4: Add the Investment Side and Calculate ROI

Internal build cost estimate:

COST CATEGORY	YOUR ESTIMATE	NOTES
Product development (engineering time)	$_____	Initial build, integrations, and orchestration design
First-year operating costs	$_____	Model usage, infrastructure, and monitoring
First-year improvement costs	$_____	Evaluation frameworks, prompt updates, and governance
Total first-year internal cost	$_____	

Platform cost estimate:

COST CATEGORY	YOUR ESTIMATE	NOTES
Platform licensing or usage fees	$_____	Annual cost
Implementation and integration effort	$_____	One-time or amortized
Internal operational ownership	$_____	Ongoing administration and governance
Total first-year platform cost	$_____	

ROI outputs:

OUTPUT	FORMULA	YOUR RESULT
Annual productivity value	From Step 3	$_____
Total first-year cost	From above	$_____
Net first-year value	Productivity value minus total cost	$_____
Annual ROI	Net value ÷ total cost x 100	_____ %
Payback period	Total cost ÷ monthly productivity value	_____ months

Break-Even Threshold

Break-even improvement % = total first-year cost ÷ (annual baseline hours x hourly cost)

$_____ ÷ (_____ baseline hours x $_____) = _____ %

If this percentage is below your Step 2 assumption, the investment is economically positive under your model.

Example output (platform scenario, $300,000 annual cost):

OUTPUT	RESULT
Annual productivity value	$777,600
Total first-year platform cost	$300,000
Net first-year value	$477,600
Annual ROI	159%
Payback period	~4.6 months
Break-even threshold	23% investigation time reduction

In this example, the investment produces positive economics if the investigation time is reduced by more than 23 percent, well below the model's moderate 40 percent assumption.

How to Use This Worksheet Going Forward

This is a starting point, not a one-time exercise. As your organization gains experience using AI in production investigation workflows, update the inputs and recalculate. Incident volumes shift. Resolution times improve. Adoption ramps unevenly across teams.

A model built on real operational data, even rough data, is significantly more credible in finance and procurement conversations than one built on vendor benchmarks or industry averages.

Chapter 9: AI ROI Will Be Won in Production

The first wave of AI adoption in software engineering has centered on the development environment. Coding assistants are visible, easy to trial, and relatively straightforward to measure. When a tool helps an engineer complete routine tasks faster, generate tests automatically, or navigate unfamiliar codebases more easily, the improvement is immediate and tangible.

But that is only one layer of the engineering system.

Software engineering does not end when code is shipped. In many organizations, the more expensive and less visible work begins after software reaches production. Systems degrade. Dependencies fail. Alerts fire. Engineers are pulled into investigations that span logs, metrics, traces, deployments, infrastructure changes, and knowledge scattered across teams. This work is necessary. It is also expensive, not only because incidents consume engineering labor directly, but because they

consume the attention of the same application, platform, and site reliability engineers responsible for building what comes next.

That is the core economic insight behind this playbook.

The long-term ROI of AI in software engineering will not be determined only by how much faster teams write code. It will be determined by how much operational toil AI can remove from the system, how quickly it can help teams understand what is happening in production, and how much engineering capacity it can free up for higher-value work.

When AI reduces investigation time, shortens the path from alert to confident decision, and helps teams stabilize systems faster, the result is not simply lower incident response cost. The result is reclaimed engineering time, one of the scarcest resources inside a software organization. That capacity can be redirected to shipping products faster, improving the customer experience, hardening the platform, reducing reliability risk, and paying down technical debt that would otherwise remain deferred.

A conservative ROI model can prove the value of AI using direct engineering labor alone. That is the right place to start. But the strategic significance is larger than the model itself. When engineers spend less time firefighting, the organization gains more time to build. That is where durable advantage begins to compound.

The organizations that realize the most value from AI will not necessarily be the ones with the most pilots, the most tooling, or the most ambitious automation claims. They will be the ones who apply AI where engineering time is currently being lost, measure the impact clearly, and build an operating model that turns those gains into repeatable advantage.

For software engineering leaders, the real shift is from treating AI as a point productivity tool to treating it as a lever on the operating economics of the engineering organization.

That is why AI ROI will ultimately be won in production.