

AI Assisted Software Development

A practical operating model for rapid safe and high quality delivery

Brad Micallef • White paper • 7 October 2026

Executive summary

AI assisted software development has moved from demonstration to working practice. The question is no longer whether an artificial intelligence system can produce useful code. It can. The useful question is how an organization should direct that capability so it produces complete, secure, maintainable solutions at a speed that conventional development made impractical.

The first paper in this series argued that powerful technology does not remove corporate responsibility. This paper applies that principle to everyday software delivery. AI is a better hammer: it can increase the force, speed, and range of a skilled team, but it does not decide what should be built, whether the design is sound, or whether the result is safe for the people who must use it. Those remain human responsibilities. [1]

The operating model proposed here rests on seven conclusions.

- A better tool raises the value of judgment. AI reduces the cost of producing code, which makes architecture, requirements, risk decisions, and verification more important rather than less.
- Company knowledge should become executable guidance. Reusable skills can encode development standards, security rules, user experience practices, approved patterns, testing expectations, and delivery checklists so they are applied repeatedly instead of rediscovered in every prompt. [2]
- The developer role moves up the value chain. The strongest contributors will combine technical depth with the work of an architect, business analyst, test designer, and accountable reviewer.
- Code is evidence, not the product. Teams should review code where risk and maintainability require it, while giving less weight to personal stylistic preferences that are already governed by formatters, analyzers, tests, and shared conventions.
- Small iterations improve both speed and control. Short cycles expose mistaken assumptions earlier, keep changes reviewable, and allow evidence from tests and users to redirect the work before cost accumulates. [6]
- End users belong inside the development loop. AI makes it practical to turn a discussion into a working slice quickly, observe real use, and revise the solution while the problem is still fresh.
- Safety must be built into the path of least resistance. Automated checks, bounded permissions, protected secrets, human approval, traceable changes, and production controls should surround AI generated work from the beginning. [3–5]

The objective is not to maximize the amount of software produced. It is to deliver more valuable capability, with stronger evidence of correctness, in less elapsed time. AI code development has arrived. Organizations now need an operating model equal to the tool.

1 Purpose and relationship to the first paper

Responsibility continues into application development

Corporate Responsibility and the Safe Development of AI addressed responsibility at the level of advanced AI development and deployment. Its central claim was that competition, incomplete regulation, and technical uncertainty do not make human choices inevitable or unaccountable. [1] This paper begins from the same premise but moves closer to ordinary business practice: using AI to create software.

The risks here are usually more familiar than loss of control by a frontier model. They include a misunderstood requirement, weak authorization, exposed credentials, inaccessible interfaces, fragile dependencies, incorrect calculations, poor error handling, and a release that works in a demonstration but fails in real operations. The fact that AI can produce code quickly can amplify these failures if an organization confuses output with completion.

The same capability can also improve the work. AI can create test fixtures, compare an implementation with a standard, explain unfamiliar code, generate accessibility checks, trace a failure across layers, and produce working alternatives for users to evaluate. The decisive issue is the system around the tool: what context it receives, what it is allowed to change, what evidence it must produce, and who approves the result.

Author perspective

I approach this subject as a technology and operations practitioner. My work has required innovative systems to become dependable operating capability, not merely convincing prototypes. That distinction shapes the argument. A software team succeeds when users can perform important work safely and effectively, the organization can support the solution, and the system can continue to evolve.

I offer the practices below as an operating model grounded in work I am doing now. I use this approach personally and professionally to build applications in Blazor and, separately, applications in Xcode. These are real examples of reusable skills helping produce successful software, not hypothetical demonstrations. The companies associated with my professional background are context for my perspective; this paper does not speak on their behalf.

2 The better hammer

A tool changes the work before it changes the responsibility

A better hammer does not eliminate carpentry. It changes what a carpenter can accomplish in a day and which parts of the work consume attention. The person still decides what is being built, measures the material, selects the joint, protects the finished surface, and checks that the structure carries the expected load. Greater striking power makes poor aim more consequential, not less.

AI plays the same role in software development. It can generate a data access layer, a user interface, a migration, a test suite, or a deployment definition in minutes. That speed is valuable. But the generated artifacts contain assumptions. Some are visible in the code; others are hidden in defaults, missing cases, package choices, security boundaries, and interpretations of the request. The team must make those assumptions explicit and test them.

The misleading comparison is AI versus a developer typing the same code. The useful comparison is one delivery system versus another. In the conventional system, scarce engineering time is spent translating familiar patterns into syntax, waiting for handoffs, and discovering requirement errors late. In an AI assisted system, more of that time can move to understanding the problem, shaping the architecture, designing tests, evaluating user behavior, and tightening the feedback loop.

Speed is a design constraint

Speed should not mean skipping work. It should mean arranging the work so that evidence arrives sooner. A fast team narrows a requirement, builds the smallest coherent slice, runs automated checks, places it in front of a user, and learns. It does not generate an entire application before asking whether the first workflow is correct.

The better hammer analogy also explains why organizations should not measure adoption by lines of code, prompts submitted, or the percentage of code generated. Those measures reward activity. Better measures describe outcomes: elapsed time from a validated need to usable capability, escaped defects, recovery time, security findings, user task success, adoption, and the amount of rework caused by misunderstood requirements.

3 Turn company standards into reusable skills

From tribal knowledge to repeatable behavior

Most organizations already have development standards. They may exist in architecture diagrams, security policies, coding conventions, design systems, review habits, and the judgment of experienced people. Their weakness is often not substance but availability. A developer or AI agent must know that the guidance exists, find the current version, understand when it applies, and translate it into the immediate task.

A reusable skill packages that knowledge for a recognizable workflow. It can contain instructions, references, approved examples, templates, and scripts. The agent sees enough metadata to recognize when the skill applies, then loads the detailed guidance when needed. This allows a company to codify a process without placing every rule into every request. Official OpenAI documentation describes skills as modular instructions and supporting files that codify processes and conventions; it also warns that skills are privileged instructions that should be reviewed and controlled. [2]

The concept is broader than any one vendor. Whether implemented as an agent skill, repository guidance, a project template, a policy-as-code check, or a controlled internal service, the purpose is the same: make the approved way easier to follow, observable in execution, and testable in outcome.

A practical skill library

Skill	What it encodes	Evidence it should require
Solution architecture	Approved boundaries, dependency direction, naming, data ownership, integration patterns, and exceptions process.	Architecture decision record, dependency checks, build results, and a concise explanation of deviations.
Secure development	Threat modeling prompts, authentication and authorization rules, secret handling, input validation, logging, dependency policy, and release gates.	Threat notes, security scan results, dependency inventory, negative tests, and approval for sensitive changes.
User experience	Design system components, accessibility targets, responsive behavior, error states, content style, and platform conventions.	Screenshots or previews, accessibility checks, keyboard and assistive-technology paths, and user acceptance notes.
Testing	Test pyramid, naming, fixtures, mocking boundaries, critical user journeys, performance thresholds, and regression expectations.	Passing unit, integration, UI, accessibility, and performance checks appropriate to the change.
Data and privacy	Classification, retention, encryption, least privilege, audit needs, migration practices, and prohibited data exposure.	Schema review, migration tests, access-control tests, data-flow notes, and rollback evidence.

Skill	What it encodes	Evidence it should require
Delivery	Branch and review rules, environments, feature flags, observability, release approval, rollback, and incident handoff.	Traceable change record, pipeline results, deployment plan, monitoring signals, and rollback procedure.

Skills are maintained products

A skill can reproduce a bad practice as efficiently as a good one. It therefore needs an owner, version, review date, test cases, and a change process. Security and architecture skills should be reviewed by the people accountable for those domains. UI skills should be tested against representative screens and accessibility needs. A testing skill should be evaluated on whether it catches seeded failures, not on whether it produces a large number of tests.

Skills should be narrow enough to activate predictably. A single instruction file called company standards will become vague, contradictory, and difficult to validate. A better library separates concerns while defining precedence. Project-specific guidance can refine an enterprise baseline, but an application should not silently weaken mandatory security or compliance requirements.

The safe pattern is to treat external or unreviewed skills as untrusted. Do not allow a skill to grant itself broader permissions, retrieve secrets, approve its own release, or bypass independent controls. High-impact actions should remain behind explicit approval and enforced policy. [2]

4 The same skill template in Blazor and Xcode applications

Two successful applications of one delivery method

The examples in this section come from applications I am developing now. In my personal and professional work, I use reusable AI skills to build Blazor applications and, in a separate development environment, Apple-platform applications in Xcode. The languages, frameworks, build systems, interface conventions, and deployment paths are different. The method that makes the work successful is substantially the same.

I do not try to force the two platforms to share every implementation detail. I reuse the delivery template: define the user outcome, supply the relevant business and technical context, load the company or project standards, build a thin vertical slice, require tests and other evidence, put the result in front of a user, revise quickly, and release deliberately. Platform-specific skills then translate that template into the architecture and conventions appropriate to Blazor or Xcode.

This distinction matters. The reusable asset is not a preferred programming language or a universal code structure. It is a disciplined way to package standards and direct AI through the complete development cycle. A reader can apply the same template to Java, Python, TypeScript, C++, embedded software, cloud services, desktop applications, mobile applications, or another architecture used by the reader's company. The local skills should express that organization's standards, risks, user experience practices, and architectural choices.

In both of my examples, the work begins with a thin vertical slice: one valuable user task that crosses the interface, application logic, data boundary, and relevant security controls. The agent receives the domain language and acceptance scenarios, then loads the platform skill that defines the architecture and test approach. I review the result as working behavior supported by evidence, not as a pile of generated files.

Concern	Blazor application	Apple application in Xcode
Interface	Razor components use the company web design system, semantic HTML, responsive layouts, keyboard support, and WCAG 2.2 criteria.	SwiftUI uses platform-native navigation and controls, the Human Interface Guidelines, Dynamic Type, VoiceOver labels, and supported device layouts.
Architecture	Business rules remain in a testable application core. Infrastructure implements explicit interfaces and is wired through dependency injection.	Domain and application logic remain independent of SwiftUI views. Protocols define services, and dependencies are injected for previews and tests.
Security	Server-side authorization protects data and operations. Client code contains no secrets. Authentication, antiforgery, validation, and logging follow the selected hosting model.	Credentials use approved platform storage. Network trust, entitlements, privacy declarations, data protection, and server authorization are explicit.
Testing	Fast domain tests, component tests, API integration tests, browser journeys, accessibility checks, and targeted performance tests.	Swift Testing for unit and integration logic, XCTest and XCUIAutomation for critical user journeys, accessibility audits, and performance tests.
Delivery	The pipeline builds, formats, analyzes, tests, scans dependencies, deploys to a review environment, and promotes with approval and rollback.	The pipeline builds and tests supported destinations, performs static checks, signs controlled artifacts, distributes a test build, and promotes through the approved release path.

How I use the template for Blazor

For Blazor applications, I use skills that direct the agent toward explicit boundaries among the UI, application core, infrastructure, and tests. The skill gives the agent the structure I expect before it begins generating a feature, so the implementation does not have to be repeatedly reorganized after the code appears. Microsoft's architecture guidance describes dependency inversion and a testable application core as useful patterns for nontrivial ASP.NET Core applications. [7] The skill also recognizes when a smaller application is better served by a simpler arrangement.

The Blazor skills make the hosting and trust model explicit. Microsoft's security guidance warns that client-side code can be inspected and tampered with and that secrets do not belong in client-side code. Authorization enforced only in the browser is not a security boundary. [8] My working pattern therefore requires the agent to identify which decisions must be enforced on the server and to produce negative tests showing that unauthorized calls fail even when the interface is bypassed.

For quality, the skills require the appropriate combination of unit tests for business rules, component tests for important states, API integration tests, and end-to-end browser tests for the journeys that matter most. Microsoft's Blazor testing guidance distinguishes component unit tests from browser-based end-to-end tests and notes that a dependency on browser DOM behavior often requires the latter. [9] The objective is not the greatest number of tests. It is the least expensive body of evidence that can expose a meaningful failure.

How I use the template for Xcode

For applications developed in Xcode, I use a separate set of skills that applies the same delivery method through Apple-native practices. The skills begin with platform behavior rather than visual imitation. They direct the agent to prefer standard SwiftUI controls and navigation, use Apple's Human Interface Guidelines, support accessibility and localization, isolate domain logic from views, and inject service dependencies behind protocols. Apple describes the Human Interface Guidelines as the source of principles and best practices for experiences across its platforms. [10]

The Xcode testing skill follows the same evidence-first principle as the Blazor skill while using the tools of the Apple ecosystem. Apple recommends many fast, isolated unit tests, fewer integration tests, and a smaller set of UI tests for critical user tasks, supplemented by performance tests where regressions matter. [11] Swift Testing covers unit and integration logic, while XCTest and XCUIAutomation exercise the interface. Accessibility is not merely a review item: Apple's UI automation uses accessibility information to identify and operate interface elements, so accessible design also makes important user flows more testable. [11–12]

The success of these two implementations is not that they produce identical code. It is that they give AI the same kind of disciplined context, require comparable evidence, and keep the human developer and end user responsible for the outcome. That is the portable pattern a company should preserve when it builds skills for its own technology stack.

A feature from request to release

1. Frame the outcome. The business analyst and user describe the job to be done, constraints, unacceptable outcomes, and observable acceptance scenarios.
2. Select the slice. The architect identifies the smallest end-to-end behavior that can test the riskiest assumption.
3. Load the standards. The agent uses the relevant architecture, security, UI, testing, and delivery skills, including platform-specific guidance.
4. Generate and explain. The agent produces the implementation, tests, assumptions, and any requested deviation from the standards.
5. Run the evidence. Automated builds, analyzers, tests, dependency checks, and security checks execute in a constrained environment.
6. Review the behavior. The developer examines architecture, trust boundaries, failure modes, and high-risk code; the user exercises the feature in a review build.
7. Revise quickly. Findings become the next small change while the context is current. The agent updates implementation and regression tests together.
8. Release deliberately. A human approves promotion, monitoring confirms expected behavior, and rollback remains available.

5 The developer role moves up the value chain

Technical knowledge becomes more valuable

When the cost of producing syntax falls, the value of knowing what the syntax should accomplish rises. A developer still needs to understand data structures, transactions, identity, concurrency, networks, performance, platform APIs, and failure modes. Without that knowledge, it is difficult to distinguish a plausible implementation from a dependable one.

The role changes from being primarily the author of every line to being the person who creates the conditions for a correct system. That work resembles several classical roles at once.

- As architect, the developer chooses boundaries, assigns responsibilities, manages dependencies, and preserves options for change.
- As business analyst, the developer converts an expressed request into domain concepts, rules, exceptions, and acceptance scenarios.

- As test designer, the developer identifies how the solution could be wrong and creates evidence that important failures are detected.
- As security practitioner, the developer identifies trust boundaries, limits authority, protects data, and insists on safe defaults.
- As delivery owner, the developer ensures the system can be deployed, observed, supported, recovered, and improved.

These responsibilities are not new. AI makes it harder to hide their absence behind the visible effort of coding. A feature that once took weeks to type can now appear in hours, leaving the team face to face with the real questions: Is this the right feature? Does it preserve the domain? Can it fail safely? Can a user understand it? Can we prove it works?

Accountability cannot be delegated to the agent

An AI system can propose an architecture and critique its own implementation, but self-critique is not independent assurance. The accountable team decides which risks require a second reviewer, a security specialist, a domain expert, or direct user validation. The more consequential the change, the stronger the evidence and separation of duties should be.

This is consistent with the secure software development view that security practices belong throughout the life cycle rather than as a final inspection. NIST's Secure Software Development Framework provides a common vocabulary for preparing the organization, protecting software, producing well-secured software, and responding to vulnerabilities. [3] AI generated code should enter that life cycle, not a special lane around it.

6 Move the emotional reward from code to outcomes

Personal style is not the same as quality

Software development has always contained craft. Pride in clear code, a well-shaped model, or an elegant solution can be healthy. The problem arises when the emotional reward of authorship becomes a reason to preserve manual effort, reject a correct solution because it is not written in one person's preferred style, or spend scarce review time on choices already governed by a shared standard.

The necessary shift is not to stop looking at code. Code review remains essential where implementation detail carries risk: authorization, concurrency, money, safety, privacy, migrations, error handling, external interfaces, and difficult-to-test behavior. The shift is to stop treating every stylistic difference as evidence of quality or every generated line as something a person must admire before it can be useful.

Formatters, compilers, analyzers, architecture tests, security scans, and agreed conventions should settle routine questions. Human attention should concentrate on intent, boundaries, assumptions, failure modes, operability, and the parts of the implementation that evidence cannot yet cover. A team should be able to explain why a change is safe and valuable without requiring one reviewer to rewrite it in a personal dialect.

The new reward

The reward moves from solving the problem by writing all the code to delivering a complete solution that people can use. That solution may contain a richer feature set than the old economics allowed: better onboarding, clearer error recovery, stronger accessibility, meaningful audit history, automated import and export, role-based workflows, diagnostics, tests, and documentation. These are often the features omitted when the budget is consumed by basic implementation.

AI changes that trade. It does not make extra features free; each one still creates complexity and support obligations. It does make it possible to explore and implement valuable completeness earlier. The architect's task is to use that capacity selectively, adding features that reduce user effort or operational risk rather than decorative scope that obscures the core job.

7 Rapid iteration is a quality mechanism

Small changes make uncertainty visible

Long delivery cycles allow assumptions to harden. A requirement is discussed, documented, interpreted, implemented, integrated, and finally shown to a user after the cost of changing it has grown. AI reduces the time required to create a working slice, which makes a different sequence practical: discuss, demonstrate, observe, and revise.

DORA research has long associated small batches and fast feedback with effective software delivery. Its guidance emphasizes that small batches help teams obtain customer feedback and reduce the risk accumulated in each change. [6] AI strengthens the economics of this pattern because the cost of producing or revising a slice is lower. The discipline of keeping the slice small remains essential.

A rapid iteration should answer a question. Can the user complete the task? Does the domain model represent the exception? Is the security boundary enforceable? Does the workflow remain understandable on a phone? Can the service meet its response target? An iteration that merely adds more generated code without reducing uncertainty is not learning; it is inventory.

Quality gates should become faster too

If generation takes minutes but validation takes days, the organization has moved the bottleneck without improving delivery. The answer is not to weaken validation. It is to automate repeatable evidence and reserve human review for judgment.

Gate	Automated evidence	Human judgment
Intent	Acceptance scenarios are present and traceable to tests.	The user and analyst confirm that the scenarios describe the real need.
Architecture	Dependency and layering rules pass; prohibited references are absent.	The architect confirms boundaries, tradeoffs, and justified exceptions.
Security	Secret, dependency, static, and dynamic checks run; negative authorization tests pass.	The reviewer evaluates threat assumptions, residual risk, and required approvals.
Experience	Accessibility checks, viewport tests, snapshots, and critical journeys pass.	Representative users confirm clarity, usefulness, and recovery from mistakes.
Operations	Deployment, health, telemetry, and rollback checks pass in a representative environment.	The owner confirms readiness, support expectations, and release timing.

8 Put the end user inside the development loop

Near real time participation changes the conversation

Traditional requirements processes ask users to predict what they will need from an abstract description. That is difficult. People are far better at reacting to a concrete workflow: the fields that are missing, the sequence that feels backward, the language that does not match the work, and the exception that occurs every Friday afternoon.

AI makes it possible for a developer and user to work much closer to the same clock. A session can begin with a problem, produce a constrained prototype or working slice, and end with observed behavior and revised acceptance scenarios. The user is no longer merely a source of requirements at the beginning and a recipient of testing at the end. The user becomes part of continuous discovery and validation.

That proximity improves both product quality and trust. Users can see which requests are easy, which create security or data consequences, and which require a broader design choice. Developers can observe the actual work instead of optimizing a simplified description. Misunderstandings become ordinary findings rather than late-stage failures.

Participation needs structure

Near real time does not mean every comment becomes an immediate production change. The team still needs a product owner, a clear decision path, representative users, protected data, and an agreed boundary between a review environment and production. Feedback should be recorded as an observation, rule, defect, or preference so the response is deliberate.

User testing should include people who encounter the difficult conditions: new users, infrequent users, administrators, people using assistive technologies, and people responsible for recovery when the system fails. For web applications, WCAG 2.2 provides a current accessibility baseline. [13] For Apple applications, platform accessibility testing should include supported devices and relevant assistive technologies. [12]

9 Safety and speed require one delivery system

Secure by design becomes the default path

Safety cannot depend on remembering a long checklist after the agent finishes. The architecture, skills, permissions, and pipeline should make the secure path the easy path. CISA's secure-by-design guidance places responsibility on software producers to own customer security outcomes and provide secure defaults rather than transferring the burden to users. [5]

For AI assisted development, this means the agent receives only the access required for the task; production credentials are unavailable; generated dependencies are evaluated; changes are traceable; protected branches and required checks cannot be bypassed by the agent; and release authority remains separate from code generation. The environment should assume that prompts, external content, packages, and skills can be malicious or simply wrong.

NIST's SSDF companion profile for generative AI emphasizes that AI systems inherit ordinary software risks while adding concerns around model artifacts, data, natural-language interaction, and iterative development. [4] Even when an organization is using an AI coding agent rather than building a foundation model, the general lesson is useful: AI belongs inside the security model, with its inputs, artifacts, permissions, and dependencies treated as part of the attack surface.

A minimum control set

- Approved repositories, models, tools, and skill versions, with an owner for each and a process for review and retirement.
- Least-privilege workspaces with no production secrets, bounded network access where practical, and explicit approval for high-impact actions.
- A written task definition with acceptance scenarios, security context, data classification, and known constraints before broad generation begins.
- Automated formatting, compilation, tests, architecture checks, secret scanning, dependency analysis, static analysis, and artifact integrity checks.
- Independent review proportional to risk, including specialists for authentication, privacy, safety-critical logic, or regulated data.
- Review environments and feature flags that allow users to test realistic behavior without exposing production systems or data.
- Traceability from request to generated changes, evidence, approval, deployment, monitoring, and rollback.
- A learning loop that turns escaped defects, incidents, and review findings into better skills, tests, and platform defaults.

These controls are not arguments for bureaucracy. They are the reusable fixture that allows a team to move quickly without renegotiating basic safety on every feature. The organization gains speed by deciding once where possible, automating what is repeatable, and escalating only the exceptions that require judgment.

10 An adoption path for established teams

Begin with one product and one measurable workflow

A company does not need to redesign its entire engineering organization before learning. It should choose a meaningful but bounded workflow, a team with technical and business access, and a baseline against which improvement can be measured. The first objective is not maximum autonomy. It is a trustworthy loop from need to deployed evidence.

Phase	Primary work	Exit evidence
1 Establish the baseline	Select one product slice. Record current lead time, defects, review effort, user wait time, and security or quality gates.	A defined outcome, representative users, acceptance scenarios, risk classification, and baseline measures.
2 Encode the paved road	Create the first architecture, security, UI, testing, and delivery skills from standards already used by the team.	Reviewed skill versions, reference examples, automated checks, and seeded tests that demonstrate the skills catch known problems.
3 Deliver in small slices	Use the skills to build thin vertical features in constrained workspaces and show them to users frequently.	Passing evidence, user observations, documented deviations, and a release or a justified decision not to release.
4 Improve the system	Analyze rework, escaped defects, slow gates, false positives, and recurring agent mistakes.	Updated skills and platform defaults, better tests, shorter feedback time, and no deterioration in safety or reliability.
5 Expand deliberately	Apply the model to another team or platform after the first workflow is stable and understood.	Comparable outcomes across teams, clear ownership, training, governance, and a controlled exception process.

Measure complete delivery

The measurement system should resist vanity metrics. Useful measures include lead time from a validated need to a usable release, deployment frequency where frequent deployment is appropriate, change failure and recovery, escaped defects, security findings by stage, accessibility defects, user task success, rework caused by requirement misunderstanding, and time spent waiting for review or environments.

AI-specific measures can include the rate of generated changes that pass the first evidence run, the categories of recurring corrections, skill activations and exceptions, and the amount of human effort spent on high-value review rather than mechanical repair. The goal is not a high acceptance rate. A low acceptance rate may show that a skill, context package, or task definition needs improvement. The important question is whether the delivery system learns.

11 AI software development has arrived

The positive case

AI code development is no longer a forecast or a topic reserved for conference demonstrations. Working systems can inspect repositories, propose plans, generate implementations, run tools, execute tests, diagnose failures, and revise their work. Skills can give those systems reusable instructions and supporting resources for company workflows. [2] The capability is present, and its practical limits are now being discovered through use.

This should be received as an opportunity. Small organizations can build internal tools that previously could not compete for engineering capacity. Product teams can test more ideas before committing to one. Developers can spend more time with users and difficult system questions. Quality practices that were repeatedly deferred—automated tests, accessibility, diagnostics, documentation, migration tooling, and consistent error handling—can become part of the standard feature rather than optional polish.

The opportunity is not automatic. Producing more code faster can produce more confusion faster. The advantage belongs to organizations that know what good looks like and can express it as architecture, skills, tests, controls, and observable outcomes. They will not use AI to avoid engineering. They will use it to practice engineering at a higher level.

A closing statement

I support the rapid adoption of AI assisted software development. It gives capable teams a better tool and makes ambitious solutions economically possible. The responsible response is neither resistance nor uncritical acceleration. It is to build a delivery system that converts speed into learning, standards into repeatable behavior, and generated code into evidence-backed capability.

Developers remain central to that system. Their value will be measured less by how much syntax they personally produce and more by whether they understand the business, shape a sound architecture, expose risk, design decisive tests, involve users, and deliver a solution that works in the conditions for which it is responsible.

The cultural transition will be uncomfortable because software has long connected professional identity with authorship. The more useful identity is ownership of the outcome. We should be proud of clear code, but prouder of a complete solution that is secure, maintainable, accessible, supportable, and genuinely useful.

The better hammer is already in our hands. The task now is to become better builders.

References

Sources accessed or checked on 7 October 2026. Numbered citations identify support for factual statements and external guidance. Recommendations and applications to AI assisted delivery are the argument of this paper; inclusion of a source does not imply endorsement of the entire paper.

- [1] Brad Micallef. Corporate Responsibility and the Safe Development of AI. White paper, 10 September 2026. Companion paper in this series.
- [2] OpenAI. Skills. Official documentation describing reusable instructions, supporting files, discovery, versioning, and safety considerations. [Open source](#)
- [3] National Institute of Standards and Technology. Secure Software Development Framework SSDF Version 1.1. NIST SP 800-218, February 2022. [Open source](#)
- [4] National Institute of Standards and Technology. Secure Software Development Practices for Generative AI and Dual-Use Foundation Models. NIST SP 800-218A, July 2024. [Open source](#)
- [5] US Cybersecurity and Infrastructure Security Agency and international partners. Shifting the Balance of Cybersecurity Risk Principles and Approaches for Secure by Design Software. [Open source](#)
- [6] DORA. State of DevOps research and DORA Core guidance concerning small batches, fast feedback, and software delivery performance. [Open source](#)
- [7] Microsoft. Common web application architectures and architectural principles for modern ASP.NET Core applications. [Open source](#)
- [8] Microsoft. ASP.NET Core Blazor authentication and authorization and Blazor WebAssembly security guidance for .NET 10. [Open source](#)
- [9] Microsoft. Test Razor components in ASP.NET Core Blazor for .NET 10. [Open source](#)
- [10] Apple. Human Interface Guidelines. [Open source](#)
- [11] Apple. Testing in Xcode, including Swift Testing, XCTest, UI testing, and the test pyramid. [Open source](#)
- [12] Apple. Performing accessibility testing for your app. [Open source](#)
- [13] World Wide Web Consortium Web Accessibility Initiative. Web Content Accessibility Guidelines 2.2 and summary of new criteria. [Open source](#)