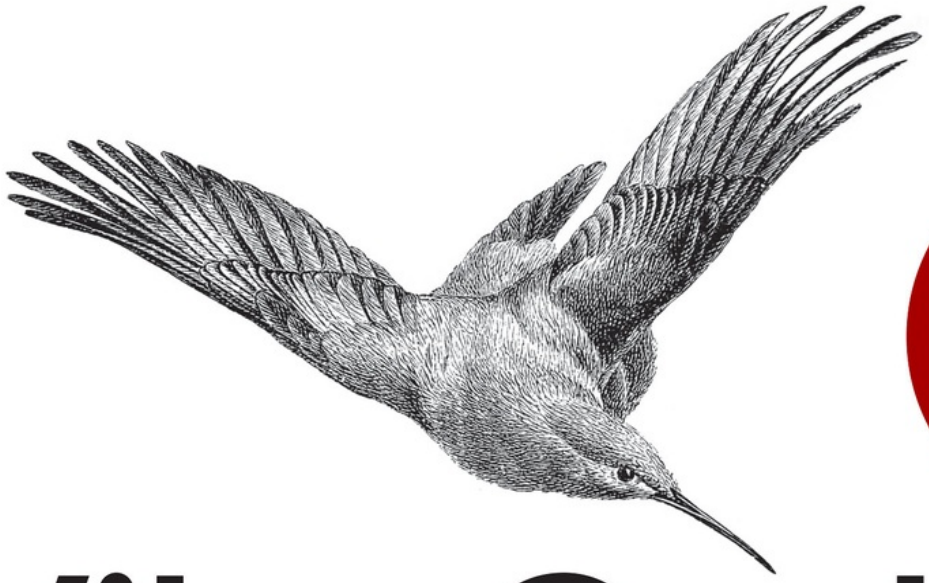
O'REILLY®



Early Release

RAW & UNEDITED

Vibe Coding

The Future of Programming

Leverage Your Experience in the Age of AI

Addy Osmani

Vibe Coding: The Future of Programming

Leveraging Your Experience in the Age of AI-Assisted Coding

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Addy Osmani

Vibe Coding: The Future of Programming

by Addy Osmani

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan

Development Editor: Sarah Grey

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

August 2025: First Edition

Revision History for the Early Release

- 2025-04-17: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9798341634756 for release details.

979-8-341-63470-1

Brief Table of Contents (*Not Yet Final*)

Chapter 1. The 70% Problem: AI-Assisted Workflows That Actually Work

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

AI-based coding tools are astonishingly good at certain tasks.[1] They excel at producing boilerplate, writing routine functions, and getting projects *most of the way* to completion. In fact, many developers find that an AI assistant can implement an initial solution that covers roughly 70% of the requirements.

A tweet from Peter Yang perfectly captures what I've been observing in the field:

> *Honest reflections from coding with AI so far as a non-engineer:*
>
> *It can get you 70% of the way there, but that last 30% is frustrating. It keeps taking one step forward and two steps backward with new bugs, issues, etc.*
>
> *If I knew how the code worked I could probably fix it myself. But since I don't, I question if I'm actually learning that much.*

Non-engineers using AI for coding find themselves hitting a frustrating wall. They can get 70% of the way there surprisingly quickly, but that final 30% becomes an exercise in diminishing returns.

This “70% problem” reveals something crucial about the current state of AI-assisted development. The initial progress feels magical - you can describe what you want, and AI tools like v0 or Bolt will generate a working prototype that looks impressive. But then reality sets in.

The 70% is often the straightforward, patterned part of the work – the kind of code that follows well-trod paths or common frameworks. As one *Hacker News* commenter observed, AI is superb at handling the **“**accidental complexity” of software (the repetitive, mechanical stuff) while the “essential complexity” – understanding and managing the inherent complexity of a problem – remains on human shoulders. In Fred Brooks’ classic terms, AI tackles the incidental, but not the intrinsic, difficulties of development.

Where do these tools struggle? Experienced engineers consistently report a “last mile” gap. AI can generate a plausible solution, but the final 30% – covering edge cases, refining the architecture, and ensuring maintainability – “needs serious human expertise”.

For example, an AI might give you a function that technically works for the basic scenario, but it won’t automatically account for unusual inputs, race conditions, performance constraints, or future requirements unless explicitly told. AI can get you most of the way there, but that final crucial 30% (edge cases, keeping things maintainable, and solid architecture) needs serious human expertise.

Moreover, AI has a known tendency to generate convincing but incorrect output. It may introduce subtle bugs or “hallucinate” nonexistent functions and libraries. Steve Yegge wryly likens today’s LLMs to “wildly productive junior developers” – incredibly fast and enthusiastic, but “potentially whacked out on mind-altering drugs,” prone to concocting crazy or unworkable approaches.

In Yegge’s words, an LLM can spew out code that looks polished at first glance, yet if a less experienced developer naively says “Looks good to me!” and runs with it, hilarity (or disaster) ensues in the following weeks. The AI doesn’t truly *understand* the problem; it stitches together patterns that *usually* make sense. Only a human can discern whether a seemingly fine solution hides long-term landmines. Simon Willison echoed this after seeing an AI propose a bewitchingly clever design that *only a senior engineer with deep understanding of the problem* could recognize as flawed. The lesson: AI’s confidence far

exceeds its reliability.

Crucially, current AIs do not create fundamentally new abstractions or strategies beyond their training data. They won't invent a novel algorithm or an innovative architecture for you – they remix what's known. They also won't take responsibility for decisions. As one engineer noted, "AIs don't have 'better ideas' than what their training data contains. They don't take responsibility for their work."

All of this means that the creative and analytical thinking – deciding *what* to build, *how* to structure it, and *why* – firmly remains a human domain. In summary, AI is a force multiplier for developers, handling the repetitive 70% and giving us a "turbo boost" in productivity. But it is *not* a silver bullet that can replace human judgment. The remaining 30% of software engineering – the hard parts – still requires skills that only trained, thoughtful developers can bring. Those are the durable skills to focus on, and Chapter 6 is dedicated to them. As one discussion put it: "AI is a powerful tool, but it's not a magic bullet... human judgment and good software engineering practices are still essential."

I've observed two distinct patterns in how teams are leveraging AI for development. Let's call them the "bootstrappers" and the "iterators." Both are helping engineers (and even non-technical users) reduce the gap from idea to execution (or MVP).

First, there are the *bootstrappers*, who are generally taking a new project from zero to MVP. Tools like Bolt, v0, and screenshot-to-code AI are revolutionizing how these teams bootstrap new projects. These teams typically:

- Start with a design or rough concept
- Use AI to generate a complete initial codebase
- Get a working prototype in hours or days instead of weeks
- Focus on rapid validation and iteration

The results can be impressive. I recently watched a solo developer use Bolt to turn a Figma design into a working web app in next to no time. It wasn't production-ready, but it was good enough to get very initial user feedback.

The second camp, the *iterators*, use tools like Cursor, Cline, Copilot, and

WindSurf for their daily development workflow. This is less flashy but potentially more transformative. These developers are:

- Using AI for code completion and suggestions
- Leveraging AI for complex refactoring tasks
- Generating tests and documentation
- Using AI as a "pair programmer" for problem-solving

But here's the catch: while both approaches can dramatically accelerate development, they come with hidden costs that aren't immediately obvious.

When you watch a senior engineer work with AI tools like Cursor or Copilot, it looks like magic. They can scaffold entire features in minutes, complete with tests and documentation. But watch carefully, and you'll notice something crucial: They're not just accepting what the AI suggests. They're constantly refactoring the generated code into smaller, focused modules. They're adding comprehensive error handling and edge-case handling the AI missed, strengthening its type definitions and interfaces, and questioning its architectural decisions. In other words, they're applying years of hard-won engineering wisdom to shape and constrain the AI's output. The AI is accelerating their implementation, but their expertise is what keeps the code maintainable.

Common Failure Patterns

Junior engineers often miss these crucial steps. They accept the AI's output more readily, leading to what I call "house of cards code" – it looks complete but collapses under real-world pressure.

Two Steps Back

What typically happens next follows a predictable pattern I call the "two steps back" pattern (shown in Figure 1-1):

- You try to fix a small bug.
- The AI suggests a change that seems reasonable.

- This fix breaks something else.
- You ask AI to fix the new issue.
- This creates two more problems.
- Rinse and repeat.

Two Steps Back

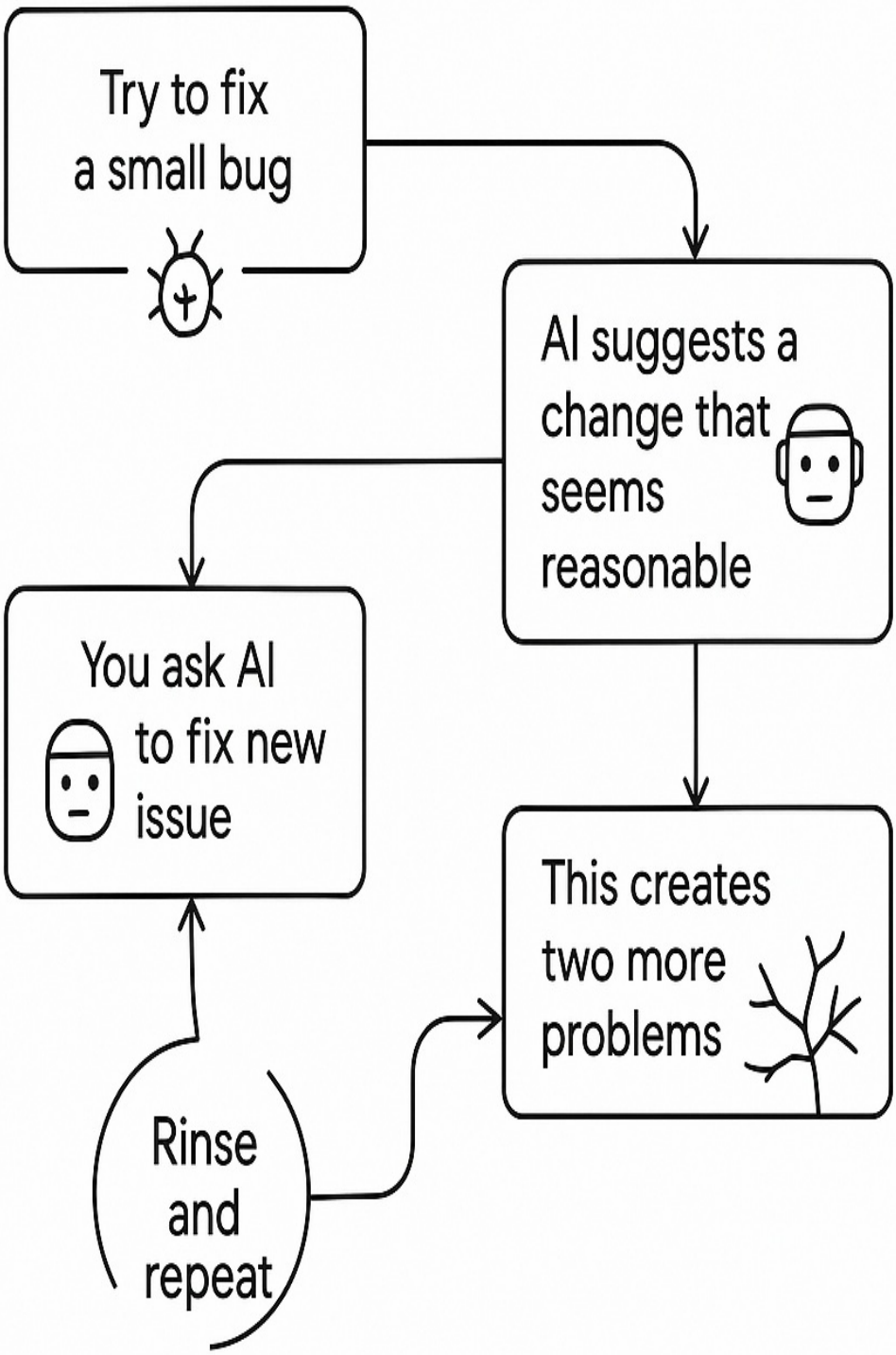


Figure 1-1. The “two steps back” antipattern

This cycle is particularly painful for non-engineers because they lack the mental models to understand what’s actually going wrong. When an experienced developer encounters a bug, they can reason about potential causes and solutions based on years of pattern recognition. Without this background, you’re essentially playing whack-a-mole with code you don’t fully understand. This is the “knowledge paradox” I mentioned back in this book’s preface**:** Senior engineers and developers use AI to accelerate what they already know how to do, while juniors try to use it to learn *what* to do.

This cycle is particularly painful for nonengineers using AI in a “bootstrapper” pattern, because they lack the mental models needed to address these issues building their MVP. However, even experienced ”iterators” can fall into this whack-a-mole trap if they overly rely on AI suggestions without deep validation.

There’s a deeper issue here: The very thing that makes AI coding tools accessible to nonengineers–their ability to handle complexity on your behalf–can actually impede learning. When code just “appears” without you understanding the underlying principles, you don’t develop debugging skills. You miss learning fundamental patterns. You can’t reason about architectural decisions, and so you struggle to maintain and evolve the code. This creates a dependency where you need to keep going back to the AI model to fix issues, rather than developing the expertise to handle them yourself.

This dependency risk may deepen with the rise of more autonomous ‘agentic’ AI systems. These agents don’t just suggest code snippets: they can plan, execute, and iterate on entire development tasks with minimal human input. As I write this in early 2025, tools like Cline, Devin AI, and Claude Code are already demonstrating the ability to autonomously debug, test, and even deploy code. While this promises efficiency, it also introduces new challenges. Without a solid understanding of the underlying processes, users may find themselves increasingly reliant on these agents, unable to intervene effectively when things go awry.

Moreover, as agentic AI systems become more integrated into development workflows, the potential for cascading errors increases. An autonomous agent might make a series of decisions that, while individually sound, collectively lead

the project in an unintended direction. Without the expertise to audit and correct these decisions, users risk building on unstable foundations.

In essence, while agentic AI offers powerful tools for software development, it also amplifies the importance of foundational knowledge. To harness these tools effectively and responsibly, users must cultivate a deep understanding of software principles, ensuring they remain in control of the development process rather than becoming passive observers.

The Demo-Quality Trap

It's becoming a pattern: Teams use AI to rapidly build impressive demos. The happy path works beautifully. Investors and social networks are wowed. But when real users start clicking around? That's when things fall apart.

I've seen this firsthand: error messages that make no sense to normal users, edge cases that crash the application, confusing UI states that never got cleaned up, accessibility completely overlooked, performance issues on slower devices. These aren't just low-priority bugs–they're the difference between software people tolerate and software people love.

Creating truly self-serve software–the kind where users never need to contact support–requires a different mindset, one that's all about the lost art of polish. You need to be obsessing over error messages, testing on slow connections and with real, nontechnical users, making features discoverable, and handling every edge case gracefully. This kind of attention to detail (perhaps) can't be AI-generated. It comes from empathy, experience, and caring deeply about craft.

What Actually Works: Practical Workflow Patterns

Before we dive into coding in Part II of this book, we need to talk about modern development practices and how AI-assisted coding fits within a team workflow. Software development is more than writing code, after all – it's a whole workflow that includes planning, collaboration, testing, deployment, and maintenance. And vibe coding isn't a standalone novelty – it can be woven into agile methodologies and DevOps practices, augmenting the team's productivity

while preserving quality and reliability.

In this section, we'll explore how team members can collectively use vibe coding tools without stepping on each other's toes, how to balance AI suggestions with human insight, and how continuous integration/continuous delivery (CI/CD) pipelines can incorporate AI or adapt to AI-generated code. I'll also touch on important considerations like version-control strategies.

After observing dozens of teams, here are three patterns I've seen work consistently in both solo and team workflows:

- *AI as first drafter*, where the AI model generates the initial code and developers then refine, refactor, and test it.
- *AI as pair programmer*, where developer and AI are in constant conversation, with tight feedback loops, frequent code review, and minimal context provided
- *AI as validator*, where developers still write the initial code and then use AI to validate, test, and improve it

In this section, I'll walk you through each pattern in turn, discussing workflows and tips for success.

AI as First Drafter

It's important to make sure everyone on the team is on the same page before you ask your AI model to begin drafting any code. Communication is key so that developers don't ask their AI assistants to do redundant tasks or generate conflicting implementations.

In daily stand-ups (a staple of agile workflows), it's worth discussing not just what you're working on, but also if you plan to use AI for certain tasks. For example, two developers might be working on different features that both involve a utility function for date formatting – if both ask the AI to create a `formatDate` helper, you might end up with two similar functions. Coordinating upfront ("I'll generate a date utility we can both use") can prevent duplication.

Teams that successfully integrate AI tools often start by agreeing on coding

standards and prompting practices. For example, the team might decide on a consistent style (linting rules, project conventions) and even feed those guidelines into their AI tools (some assistants allow providing style preferences or example code to steer outputs). As Codacy's blog notes, by familiarizing the AI with the team's coding standards, you get generated code that is more uniform and easier for everyone to work with. On a practical level, this could mean having a section in your project README for "AI Usage Tips," where you note things like "We use functional components only" or "Prefer using Fetch API over Axios," which developers can keep in mind when prompting AI.

Another practice is to use your tools' *collaboration features* if available. Some AI-assisted IDEs allow users to share their AI sessions, or at least the prompts they use. If Developer A got a great result with a prompt for a complex component, sharing that prompt with Developer B (perhaps via the issue tracker or a team chat) can save time and ensure consistency.

As for using version control, the fundamentals remain – with a twist. Using Git (or another version control system) is non-negotiable in modern development, and that doesn't change with vibe coding. In fact, version control becomes even more crucial when AI is generating code rapidly. Commits act as the safety net to catch AI missteps; if an AI-generated change breaks something, you can revert to a previous commit.

One strategy is to commit more frequently when using AI assistance. Each time the AI produces a significant chunk of code (like generating a feature or doing some major refactoring) that you accept, consider making a commit with a clear message. Frequent commits ensure that if you need to bisect issues or undo a portion of AI-introduced code, the history is granular enough.

Also, try to isolate different AI-introduced changes. If you let the AI make many changes across different areas and commit them all together, it's harder to disentangle if something goes wrong. For example, if you use an agent to optimize performance and it also tweaks some UI texts, commit those separately. (Your two commit messages might be "Optimize list rendering performance [AI-assisted]" and "Update UI copy for workout completion message [AI-assisted]"). Descriptive commit messages are important; some teams even tag commits that had heavy AI involvement, just for traceability. It's not about blame, but about understanding the origin of code – a commit tagged with "[AI]" might signal to a

reviewer that the code could use an extra-thorough review for edge cases.

Essentially, the team should treat AI usage as a normal part of the development conversation: share experiences, successful techniques, and warnings about what not to do (like “Copilot suggests using an outdated library for X, be careful with that”).

Review and refinement are crucial to this pattern. Developers should manually review and refactor the code for modularity, add comprehensive error handling, write thorough tests, and document key decisions as they refine the code. The next chapter goes into detail about these processes.

AI as Pair Programmer

Traditional pair programming involves two humans collaborating at one workstation. With the advent of AI, a hybrid approach has emerged: one human developer working alongside an AI assistant. This setup can be particularly effective, offering a blend of human intuition and machine efficiency.

In a human-AI pairing, the developer interacts with the AI to generate code suggestions, while also reviewing and refining the output. This dynamic allows the human to leverage the AI’s speed in handling repetitive tasks, such as writing boilerplate code or generating test cases, while maintaining oversight to ensure code quality and relevance.

For instance, when integrating a new library, a developer might prompt the AI to draft the initial integration code. The developer then reviews the AI’s suggestions, cross-referencing with official documentation to verify accuracy. This process not only accelerates development but also facilitates knowledge acquisition, as the developer engages deeply with both the AI’s output and the library’s intricacies.

Let’s compare this to traditional human-human pair programming:

- *Human-AI pairing* offers rapid code generation and can handle mundane tasks efficiently. It’s particularly beneficial for solo developers or when team resources are limited.
- *Human-human pairing* excels in complex problem-solving scenarios, where nuanced understanding and collaborative brainstorming are

essential. It fosters shared ownership and collective code comprehension.

Both approaches have their merits, and your choice between them can be guided by the project's complexity, resource availability, and the specific goals of the development process.

Best Practices for AI Pair Programming

To maximize the benefits of AI-assisted development, consider the following practices:

Initiate new AI sessions for distinct tasks

This helps maintain context clarity and ensures the AI's suggestions are relevant to the specific task at hand.

Keep prompts focused and concise

Providing clear and specific instructions enhances the quality of the AI's output.

Review and commit changes frequently

Regularly integrating and testing AI-generated code helps catch issues early and maintains project momentum.

Maintain tight feedback loops

Continuously assess the AI's contributions, providing corrections or refinements as needed to guide its learning and improve future suggestions.

AI as Validator

Beyond code generation, AI can serve as a valuable validator, assisting in code review and quality assurance. AI tools can analyze code for potential bugs, security vulnerabilities, and adherence to best practices. For example, platforms like DeepCode and Snyk's AI-powered code checker can identify issues such as missing input sanitization or insecure configurations, providing actionable insights directly within the development environment. Platforms such as Qodo

and TestGPT can automatically generate test cases, ensuring broader coverage and reducing manual effort. And many AI tools can assist in monitoring application performance, detecting anomalies that might indicate underlying issues.

By integrating AI validators into the development workflow, teams can enhance code quality, reduce the likelihood of defects, and ensure compliance with security standards. This proactive approach to validation complements human oversight, leading to more robust and reliable software. These tools enhance the efficiency and effectiveness of the QA process by handling repetitive and time-consuming tasks, allowing human testers to focus on more complex and nuanced aspects of quality assurance.

Incorporating AI into the development process, whether as a pair programmer or validator, offers opportunities to enhance productivity and code quality. By thoughtfully integrating these tools, developers can harness the strengths of both human and artificial intelligence.

To maximize the benefits of both AI and human capabilities in QA, I recommend a few best practices:

- Use AI for initial assessments and preliminary scans to identify obvious issues.
- Prioritize human review for critical areas, such as complex functionalities, user experience, and areas where AI may have limitations.
- Foster an environment of continuous collaboration, where AI tools and human testers work in tandem, with ongoing feedback loops to improve both AI performance and human decision-making.

The Golden Rules of Vibe Coding

Be specific and clear about what you want

> Clearly articulate your requirements, tasks, and outcomes when interacting with AI. Precise prompts yield precise results.

Always validate AI output against your intent

AI-generated code must always be checked against your original goal. Verify functionality, logic, and relevance before accepting.

Treat AI as a junior developer (with supervision)

Consider AI outputs as drafts that require your careful oversight. Provide feedback, refine, and ensure quality and correctness.

Use AI to expand your capabilities, not replace your thinking

Leverage AI to automate routine or complex tasks, but always remain actively engaged in problem-solving and decision-making.

Coordinate upfront among the team before generating code

Align with your team on AI usage standards, code expectations, and practices before starting AI-driven development.

Treat AI usage as a normal part of the development conversation

Regularly discuss AI experiences, techniques, successes, and pitfalls with your team. Normalize AI as another tool for collective improvement.

Isolate AI changes in Git by doing separate commits

Clearly identify and separate AI-generated changes within version control to simplify reviews, rollbacks, and tracking.

Ensure that all code, whether human or AI-written, undergoes code review

Maintain consistent standards by subjecting all contributions to the same rigorous review processes, enhancing code quality and team understanding.

Don’t merge code you don’t understand

Never integrate AI-generated code unless you thoroughly comprehend its functionality and implications. Understanding is critical to maintainability and security.

Prioritize documentation, comments, and ADRs

Clearly document the rationale, functionality, and context for AI-generated code. Good documentation ensures long-term clarity and reduces future technical debt.

Share and reuse effective prompts

Document prompts that lead to high-quality AI outputs. Maintain a repository of proven prompts to streamline future interactions and enhance consistency.

Regularly reflect and iterate

Periodically review and refine your AI development workflow. Use insights from past experiences to continuously enhance your team's approach.

By adhering to these golden rules, your team can harness AI effectively, enhancing productivity while maintaining clarity, quality, and control.

---

1 This chapter is based on an essay originally published on my Substack newsletter, *Elevate with Addy Osmani*, "The 70% Problem: Hard Truths about AI-Assisted Coding," December 4, 2024.

Chapter 2. Beyond the 70%: Maximizing Human Contribution

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

You've seen how AI coding assistants like Cursor, Cline, Copilot and WindSurf have transformed how software is built, shouldering much of the grunt work and boilerplate–about 70%.[1] But what about that last "30%" of the job that separates a toy solution from a production-ready system? This gap includes the hard parts: understanding complex requirements, architecting maintainable systems, handling edge cases, and ensuring code correctness. In other words, while AI can generate *code*, it often struggles with *engineering*.

Tim O'Reilly, reflecting on decades of technology shifts, reminds us that each leap in automation has changed *how* we program but not *why* we need skilled programmers. We're not facing the end of programming, but rather "the end of programming as we know it today," meaning developers' roles are evolving, not evaporating.

The challenge for today's engineers is to embrace AI for what it does best (the first 70%) while doubling down on the durable skills and insights needed for the remaining 30%. This article dives into expert insights to identify which human skills remain crucial. We'll explore what senior and mid-level developers should continue to leverage and what junior developers must invest in to thrive

alongside AI.

This chapter's goal, then, is to offer you pragmatic guidance for maximizing the value of that irreplaceable 30%, with actionable takeaways for engineers at every level.

Senior Engineers and Developers: Leverage Your Experience with AI

If you're a senior engineer, you should see the advent of AI coding tools as an opportunity to amplify your impact – if you leverage your experience in the right ways. Senior developers typically possess deep domain knowledge, intuition for what could go wrong, and the ability to make high-level technical decisions.

These strengths are part of the 30% that AI can't handle alone. This section looks at how seasoned developers can maximize their value.

Be the Architect and the Editor-in-Chief

Let AI handle the first draft of code, while you focus on architecting the solution and then refining the AI's output. In many organizations, Steve Yegge writes, we may see a shift where teams need "only senior associates" who "(a) describe the tasks to be done (i.e. create the prompts), and (b) review the resulting work for accuracy and correctness." Embrace that model. As a senior dev, you can translate complex requirements into effective prompts or specifications for an AI assistant, then use your critical eye to vet every line produced. You are effectively pair-programming with the AI – it's the fast typer, but you're the brain.

Maintain high standards during review: ensure the code meets your organization's quality, security, and performance benchmarks. By acting as architect and editor, you prevent the "high review burden" from overwhelming you. (A cautionary note: if junior staff simply throw raw AI output over the wall to you, push back – instill a process where they must verify AI-generated work first, so you're not the sole safety net.)

Use AI as a Force Multiplier for Big Initiatives

Senior engineers often drive large projects or tackle hairy refactors that juniors can't approach alone. AI can supercharge these efforts by handling a lot of mechanical changes or exploring alternatives under your guidance. Steve Yegge introduced the term *chat-oriented programming* (CHOP) for this style of working – "coding via iterative prompt refinement," with the AI as a collaborator. Leverage CHOP to be more ambitious in what you take on.

Having AI assistance lowers the bar for when a project is worth investing time in at all since what might have taken days can now be done in hours. Senior devs can thus attempt those "wouldn't it be nice if…?" projects that always seemed slightly out of reach.

The key is to remain the guiding mind: you decide which tools or approaches to pursue, and you integrate the pieces into a cohesive whole. Your experience allows you to sift the AI's suggestions – accepting those that fit, rejecting those that don't.

Mentor and Set Standards

Another crucial role for senior engineers is to coach less experienced team members on effective use of AI and on the timeless best practices. You likely have hard-won knowledge of pitfalls that juniors may not see, like memory leaks, off-by-one errors, and concurrency hazards..

With juniors now potentially generating code via AI, it's important to teach them how to self-review and test that code. Set an example by demonstrating how to thoroughly test AI contributions, and encourage a culture of questioning and verifying machine output. Some organizations (including even law firms) have instituted rules that if someone uses an AI to generate code or writing, they must *disclose it and verify the results themselves* – not just assume a senior colleague will catch mistakes.

As a senior engineer, champion such norms on your team: AI is welcome, but diligence is required. By mentoring juniors in this way, you offload some of the oversight burden and help them grow into that 30% skillset more quickly.

Continue to Cultivate Domain Mastery and Foresight

Your broad experience and context are more important than ever. Senior

developers often have historical knowledge of why things in the company are built a certain way or how an industry operates. This domain mastery lets you catch AI's missteps that a newcomer wouldn't.

Continue investing in understanding the problem domain deeply. That might mean staying up to date with the business's needs, user feedback, or new regulations that affect the software. AI won't automatically incorporate these considerations unless you tell it to. When you combine your domain insight with AI's speed, you get the best outcomes.

Also, use your foresight to steer AI. For instance, if you know that a quick fix will create maintenance pain down the line, you can instruct the AI to implement a more sustainable solution. Trust the instincts you've honed over the years – if a code snippet looks "off" or too good to be true, dig in. Nine times out of ten, your intuition has spotted something that the AI didn't account for. Being able to foresee the second- and third-order effects of code is a hallmark of senior engineers; don't let the convenience of AI blunt that habit. Instead, apply it to whatever the AI produces.

Hone Your Soft Skills and Leadership

With AI shouldering some coding, senior developers can spend more energy on the human side of engineering. This includes communicating with stakeholders, leading design meetings, and making judgment calls that align technology with business strategy. Tim O'Reilly and others suggest that as rote coding becomes easier, the value shifts to deciding *what* to build and *how to orchestrate* complex systems.

Senior engineers are often the ones orchestrating and seeing the big picture. Step up to that role. Volunteer to write that architecture roadmap, to evaluate which tools (AI or otherwise) to adopt, or to define your org's AI coding guidelines. These are tasks AI can't do – they require experience, human discretion, and often, cross-team consensus-building. By amplifying your leadership presence, you ensure that you're not just a code generator (replaceable by another tool), but an indispensable technical leader guiding the team.

In short, continue doing what seasoned developers do best: seeing the forest for the trees.

AI will help you chop a lot more trees, but someone still needs to decide *which* trees to cut and *how* to build a stable house from the lumber. Your judgment, strategic thinking, and mentorship are now even more critical. A senior developer who harnesses AI effectively can be dramatically more productive than one who doesn't – but the ones who truly excel will be those who apply their human strengths to amplify the AI's output, not just let it run wild.

As one Redditor observed, "AI is a programming force multiplier" that "greatly increases the productivity of senior programmers." The multiplier effect is real, but it's your expertise that's being multiplied. Keep that expertise sharp and at the center of the development process.

Midlevel Engineers: Adapt and Specialize

If you're a midlevel engineer, you face perhaps the most significant pressure to evolve. Many of the tasks that traditionally occupied your time – implementing features, writing tests, debugging straightforward issues – are becoming increasingly automatable.

This doesn't mean obsolescence; it means elevation. The focus shifts from writing code to more specialized knowledge, which the following sections explore.

Learn to Manage Systems Integration and Boundaries

As systems become more complex, understanding and managing the boundaries between components becomes crucial. This includes API design, event schemas, and data models – all requiring careful consideration of business requirements and future flexibility. Deepen your computer-science fundamentals, including gaining an advanced understanding of disciplines like:

- Data structures and algorithms
- Distributed-systems principles
- Database internals and query optimization
- Network protocols and security

This knowledge helps you understand the implications of AI-generated code and make better architectural decisions.

Learn to handle edge cases and ambiguity, too. Real-world software is rife with oddball scenarios and changing requirements. AI tends to solve the general case by default. It's up to the developer to ask "What if…?" and probe for weaknesses.

The durable skills here are critical thinking and foresight – enumerating edge cases, anticipating failures, and addressing them in code or design. This might mean thinking of null input, network outages, unusual user actions, or integration with other systems.

Build Your Domain Expertise

Understanding the business context or the user's environment will reveal edge cases that a generic AI simply doesn't know about. Experienced engineers habitually consider these scenarios. Practice systematically testing boundaries and questioning assumptions. Specialize in complex domains where human understanding remains crucial.

Generic domains include:

- Financial systems with regulatory requirements
- Healthcare systems with privacy concerns
- Real-time systems with strict performance requirements
- Machine learning infrastructure

Software-engineering-specific domains include frontend and backend engineering, mobile development, DevOps, and security engineering, to name a few. Domain expertise provides context that current AI tools lack and helps you make better decisions about where and how to apply them.

Master Performance Optimization and DevOps

While LLMs can suggest basic optimizations, identifying and resolving system-wide performance issues requires a deep understanding of the entire stack, from

database query patterns to frontend rendering strategies. Understanding how systems run in production becomes more valuable as code generation becomes more automated. Focus on fields like:

- Monitoring and observability
- Performance profiling and optimization
- Security practices and compliance
- Cost management and optimization

Focus on Code Review and Quality Assurance

With AI writing lots of code, the ability to rigorously review and test that code becomes even more critical. “Everyone will need to get a lot more serious about testing and reviewing code,” Steve Yegge emphasizes. Treat AI-generated code as you would a human junior developer’s output – you are the code reviewer responsible for catching bugs, security flaws, or sloppy implementations. This means strengthening your skills in unit testing, integration testing, and debugging.

Writing good tests is a durable skill that forces you to understand the spec and verify correctness. It’s wise to assume nothing works until proven otherwise. As Builder.io CEO Steve Sewell notes, AI often yields “functional but horribly optimized code” until you guide it through iterative improvement.

Cultivate a testing mindset: verify every critical logic path, use static analysis or linters, and don’t shy away from rewriting AI-given code if it doesn’t meet your quality bar. Even if you’re following the “AI as validator” pattern discussed in the previous chapter, quality assurance is not an area to simply outsource to AI – it’s where human diligence shines. When software doesn’t work as expected, you need real problem-solving chops to diagnose and fix it. AI can assist with debugging (for example, by suggesting possible causes), but it lacks true understanding of the specific context in which your application runs. Human testers possess domain-specific knowledge and an understanding of user expectations that AI currently lacks. This insight is vital when assessing the relevance and impact of potential issues. Diagnosing complex bugs often requires creative problem-solving and the ability to consider a broad range of

factors—skills that are inherently human. And evaluating the ethical implications of software behavior, such as fairness and accessibility, requires human sensitivity and judgment.

Being able to reason through a complex bug – reproducing it, isolating the cause, understanding the underlying systems (OS, databases, libraries) – is a timeless engineering skill. This often requires a strong grasp of fundamentals (how memory and state work, concurrency, etc.) that junior developers must learn through practice. Use AI as a helper (it might explain error messages or suggest fixes), but *don't rely on it thoughtlessly*. The skill to methodically troubleshoot and apply first principles when debugging sets great developers apart. It's also a feedback loop: debugging AI-written code will teach you to prompt the AI better next time or avoid certain patterns.

Learn Systems Thinking

Software projects are not just isolated coding tasks; they exist within a larger context of user needs, timelines, legacy code, and team processes. AI has no innate sense of the big picture, like your project's history or the rationale behind certain decisions (unless you explicitly feed all that into the prompt, which is often impractical). Humans need to carry that context.

The durable skill here is systems thinking – understanding how a change in one part of the system might impact another, how the software serves the business objectives, and how all the moving pieces connect.[2] This holistic perspective lets you use AI outputs appropriately. For example, if an AI suggests a clever shortcut that contradicts a regulatory requirement or company convention, you'll catch it because you know the context. Make it a point to learn the background of your projects and read design docs, so you can develop your judgment about what fits and what doesn't.

Be Adaptable–and Never Stop Learning

Finally, a meta-skill: the ability to learn new tools and adapt to change. The field of AI-assisted development is evolving rapidly. Engineers who keep an open mind and learn how to effectively use new AI features will remain ahead of the curve – Tim O'Reilly suggests that developers who are "eager to learn new

skills" will see the biggest productivity boosts from AI. Invest in learning the *fundamentals* deeply and staying curious about new techniques. This combination enables you to harness AI as a tool without becoming dependent on it.

It's a balancing act: use AI to accelerate your growth, but also occasionally practice without it to ensure you're not skipping core learning (some developers do an "AI detox" periodically to keep their raw coding skills sharp. In short, be the engineer who learns constantly – that's a career-proof skill in any era.

Get Good at Cross-Functional Communication

The ability to translate between business requirements and technical solutions becomes more valuable as implementation time decreases. Engineers who can effectively communicate with product managers, designers, and other stakeholders will become increasingly valuable. Good areas of focus here include:

- Requirements gathering and analysis
- Technical writing and documentation
- Project planning and estimation
- Team leadership and mentoring

Learn System Design and Architecture

Instead of spending days implementing a new feature, mid-level engineers might spend that time designing robust systems that gracefully handle scale and failure modes. This requires deep understanding of distributed systems principles, database internals, and cloud infrastructure – areas where LLMs currently provide limited value.

Practice designing systems that solve real-world problems at scale. These skills remain valuable regardless of how code is generated, as they require understanding business requirements and engineering tradeoffs.

Designing a coherent system requires understanding trade-offs, constraints, and the "big picture" beyond writing a few functions. AI can generate code but won't

automatically choose the best architecture for a complex problem.

The overall design – how components interact, how data flows, how to ensure scalability and security – is part of that 30% that demands human insight. This includes:

- Load balancing and caching strategies
- Data partitioning and replication
- Failure modes and recovery procedures
- Cost optimization and resource management

Senior developers have long honed this skill, and midlevel and junior devs should actively cultivate it. Think in terms of patterns and principles (like separation of concerns and modularity) – these guide an AI-generated solution toward maintainability. Remember, *solid architecture doesn't emerge by accident*; it needs an experienced human hand on the wheel.

Use AI!

Remember that AI should be an integral part of your workflow–it's not something to resist. Practical ways to incorporate AI into your daily work include:

- Scaffolding initial code structures
- Quick prototypes and proof-of-concepts
- Pair-programming for faster debugging and problem-solving
- Suggesting optimizations and alternative approaches
- Handling repetitive code patterns while you focus on architecture and design decisions

Venture into UI and UX Design

There's a growing narrative that midlevel software engineers should "just quit" – that pure engineering skills will become obsolete as AI handles the

implementation details. While the conclusion is overstated, the discourse about the importance of skills beyond engineering (like design) deserves examination. In a representative exchange on X in December 2024, @nullpointered wrote:

> *If you are a software engineer who's three years into your career: quit now. there is not a single job in CS anymore. it's over. this field won't exist in 1.5 years.*

To which @garrytan replied in a quote tweet:

> *Learn X design and product design and you will become stronger than you could ever imagine.*

Successful software creation has always required more than just coding ability. What's changing is not the death of engineering, but rather the lowering of pure implementation barriers. This shift actually makes engineering judgment and design thinking more crucial, not less.

Consider what makes applications like Figma, Notion, or VS Code successful. It's not just technical excellence – it's the deep understanding of user needs, workflows, and pain points. This understanding comes from:

- User experience design thinking
- Deep domain knowledge
- Understanding of human psychology and behavior
- System design that considers performance, reliability, and scalability
- Business model alignment

The best engineers have always been more than just coders. They've been problem solvers who understand both technical constraints and human needs. As AI tools reduce the friction of implementation, this holistic understanding becomes even more valuable.

However, this doesn't mean every engineer needs to become a UX designer. Instead, it means developing stronger product thinking abilities and building better collaboration skills with designers and product managers. It means thinking more about users, understanding their psychology and behavior patterns and learning to make technical decisions that support user experience goals.

You're at the point of achieving technical elegance: now balance it out with close attention to practical user needs.

Tan went on to tweet:

> *UX, design, actual dedication to the craft will take center stage in this next moment*
>
> *Actually make something people want. Software and coding won't be the gating factor. It is the ability to be a polymath and smart/effective in many domains together that creates great software.*

The future belongs to engineers who can bridge the gap between human needs and technical solutions – whether that's through developing better design sensibilities themselves or through more effective collaboration with dedicated designers.

Junior Developers: Thrive Alongside AI

If you're a junior or less-experienced developer, you might feel a mix of excitement and anxiety about AI. AI assistants can write code that you might not know how to write yourself, potentially accelerating your learning. Yet there are headlines about the "death of the junior developer," suggesting entry-level coding jobs are at risk. Contrary to popular speculation, while AI is significantly changing the early-career experience, junior developers *are not obsolete*.

You need to be proactive in developing skills that ensure you're contributing value beyond what an AI can churn out. The traditional path of learning through implementing basic CRUD applications and simple features will evolve as these tasks become increasingly automated.

Consider a typical junior task: implementing a new API endpoint following existing patterns. Previously, this might have taken a day of coding and testing. With AI assistance, the implementation time might drop to an hour, but the crucial skills become:

- Understanding the existing system architecture well enough to specify the requirement correctly

- Reviewing the generated code for security implications and edge cases
- Ensuring the implementation maintains consistency with existing patterns
- Writing comprehensive tests that verify business logic

These skills can’t be learned purely through tutorial following or AI prompting – they require hands-on experience with production systems and mentorship from senior engineers.

This evolution presents both challenges and opportunities for early-career developers. The bar for entry-level positions may rise, requiring stronger fundamental knowledge to effectively review and validate AI-generated code. However, this shift also means junior engineers can potentially tackle more interesting problems earlier in their careers.

Here’s how to invest in yourself to handle that 30% gap effectively.

Learn the Fundamentals: Don’T Skip the “Why”

It’s tempting to lean on AI for answers to every question (“How do I do X in Python?”) and never truly absorb the underlying concepts. Resist that urge. Use AI as a tutor, not just an answer vending machine. For example, when AI gives you a piece of code, ask *why* it chose that approach, or have it explain the code line by line.

Make sure you understand concepts like data structures, algorithms, memory management, and concurrency without always deferring to AI. The reason is simple: when the AI’s output is wrong or incomplete, you need your own mental model to recognize and fix it. If you’re not actively engaging with why the AI is generating certain code, you might actually learn less”, hindering your growth. So take time to read documentation, write small programs from scratch, and solidify your core knowledge. These fundamentals are durable; they’ll serve you even as the tools around you change.

Practice Problem Solving and Debugging Without the AI Safety Net

To build real confidence, sometimes you have to fly solo. Many developers advocate doing an “AI-free day” or otherwise limiting AI assistance periodically . This ensures you can still solve problems with just your own skills, which is important for avoiding skill atrophy. You’ll find it forces you to truly think through a problem’s logic, which in turn makes you better at using AI (since you can direct it more intelligently).

Additionally, whenever you encounter a bug or error in AI-generated code, jump in and debug it *yourself* before asking the AI to fix it. You’ll learn much more by stepping through a debugger or adding print statements to see what’s going wrong.

Consider AI suggestions as hints, not final answers. Over time, tackling those last tricky bits of a task will build your skill in the very areas AI struggles – exactly what makes you valuable.

Focus on Testing and Verification

As a junior dev, one of the best habits you can develop is writing tests for your code. This is doubly true if you use AI to generate code.

When you get a chunk of code from an LLM, don’t assume it’s correct – challenge it. Write unit tests (or use manual tests) to see if it truly handles the requirements and edge cases. This accomplishes two things: it catches issues in the AI’s output, and it trains you to think about expected behavior before trusting an implementation.

You might even use the AI to help write tests, but *you* define what to test. Steve Yegge’s advice about taking testing and code review seriously applies at all levels. If you cultivate a reputation for carefully verifying your work (AI-assisted or not), senior colleagues will trust you more and you’ll avoid the scenario where they feel you’re just “dumping” questionable code on them.

In practical terms, start treating testing as an integral part of development, not an afterthought. Learn how to use testing frameworks, how to do exploratory manual testing, and how to systematically reproduce bugs. These skills not only make you better at the 30% work, they also accelerate your understanding of how the code really works.

Remember: if you catch a bug that the AI introduced, *you* just did something the AI couldn't – that's added value.

Build an Eye for Maintainability

Junior devs often focus on "getting it to work." But in the AI era, getting a basic working version is easy – the AI can do that. The harder part (and what you should focus on) is making code that's readable, maintainable, and clean.

Start developing an eye for good code structure and style. Compare the AI's output with best practices you know of; if the AI code is messy or overly complex, take the initiative to refactor it. For instance, if an LLM gives you a 50-line function that does too many things, you can split it into smaller functions. If variable names are unclear, rename them.

Essentially, pretend you're reviewing a peer's code and improve the AI's code as if a peer wrote it. This will help you internalize good design principles. Over time, you'll start prompting the AI in ways that yield cleaner code to begin with (because you'll specify the style you want). Software maintainers (often working months or years later) will thank you, and you'll prove that you're thinking beyond just "make it run" – you're thinking like an engineer. Keeping things maintainable is exactly in that human-driven 30%, so make it your concern from the start of your career.

Develop Your Prompting and Tooling Skills (Wisely)

There's no denying that "prompt engineering" – the skill of interacting with AI tools effectively – is useful. As a junior dev, you should absolutely learn how to phrase questions to AI, how to give it proper context, and how to iterate on prompts to improve the output (Chapter 2 of this book is a good place to start). These are new skills that can set you apart (many experienced devs are still figuring this out too!). However, remember that prompting well is often a proxy for understanding the problem well. If you find you can't get the AI to do what you want, it might be because *you* need to clarify your own understanding first. Use that as a signal.

One strategy is to outline a solution in plain English yourself before asking the AI to implement it. Also, experiment with different AI tools (Copilot, Claude,

etc.) to see their strengths and weaknesses. The more fluent you are with these assistants, the more productive you can be – but never treat their output as infallible. Think of AI like a super-charged Stack Overflow: an aid, not an authority.

You might even build small personal projects using AI to push your limits ("Can I build a simple web app with AI's help?"). Doing so will teach you how to integrate AI into a development workflow, which is a great skill to bring into a team. Just balance it with periods of working without the net, as mentioned earlier.

Seek Feedback and Mentorship

Lastly, one durable skill that will accelerate your growth is the ability to seek out feedback and learn from others. An AI won't get offended if you ignore its advice, but your human teammates and mentors are invaluable for your development-especially when it comes to soft skills, leadership, communication, and navigating office politics.

Don't hesitate to ask a senior developer why they prefer one solution over another, especially if it differs from what an AI suggested. Discuss design decisions and trade-offs with more experienced colleagues–these conversations reveal how seasoned engineers think, and that's gold for you. In code reviews, be extra receptive to comments about your AI-written code. If a reviewer points out that "this function isn't thread-safe" or "this approach will have scaling issues," take the time to understand the root issue. These are exactly the kinds of things an AI might miss, and you want to learn to catch them. Over time, you'll build a mental checklist of considerations.

Additionally, find opportunities to pair program (even if remotely). Perhaps you can "pair" with a senior who uses AI in their workflow–you'll observe how they prompt the AI and how they correct it. But even more important, you'll see how they communicate, lead discussions, and handle delicate team dynamics. Being open to feedback and actively asking for guidance will help you mature from doing tasks that an AI could do to doing the high-value tasks that only humans can do. In a sense, you're trying to acquire the wisdom that usually comes with experience, as efficiently as you can. That makes you more than just another coder in the room–it makes you the kind of engineer teams are eager to keep and

promote.

Communicate and Collaborate

Building software is a team sport. AI doesn't attend meetings (thank goodness) – humans still must talk to other humans to clarify requirements, discuss trade-offs, and coordinate work. Strong communication skills are as valuable as ever. Practice asking good questions and describing problems clearly (both to colleagues and to AI).

Interestingly, prompting an AI is itself a form of communication; it requires you to precisely express what you want. This overlaps with a core engineering skill: *requirements analysis*.[3] If you can formulate a clear prompt or spec, it means you've thought through the problem.

Additionally, sharing knowledge, writing documentation, and reviewing others' code are collaborative skills that AI cannot replace. In the future, as developers work "with" AI, the human-to-human collaboration in a team – making sure the right problems are being solved – stays vital. One emerging trend is that developers may focus more on high-level design discussions (often with AI as a participant) and on coordinating tasks, essentially taking on more of a conductor role. Communication and leadership skills will serve you well in that conductor's seat.

Shift Your Mindset: From Consuming to Creating

It's worth noting a mindset shift for juniors in the AI era: you need to move from just *consuming solutions* to *creating understanding*. In the past, you might have struggled through documentation to eventually write a feature; now an AI can hand you a solution on a platter. If you simply consume it (copy-paste and move on), you haven't grown much.

Instead, use each AI-given solution as a learning case. Dissect it, experiment with it, and consider how you might have arrived at it yourself. By treating AI outputs not as answers to end all questions but as interactive learning material, you ensure that you – the human – are continuously leveling up. This way, rather than replacing your growth, AIit accelerates it.

Many experts believe that while AI might reduce the need for large teams of junior “coder-grinders,” it also *raises the bar* for what it means to be a junior developer. The role is shifting to someone who can work effectively with AI and quickly climb the value chain. If you adopt the habits above, you’ll distinguish yourself as a junior developer who doesn’t just bring what an AI could bring (any company can get that via a subscription), but who brings insight, reliability, and continuous improvement – traits of a future senior developer.

Future-Proof Your Career with Durable Engineering Skills

In summary, to thrive in an AI-enhanced development world, engineers at all levels should double down on the enduring skills and practices that AI cannot (yet) replicate. These capabilities will remain crucial no matter how advanced our tools become. In particular, focus on:

- Strengthening your system design and architecture expertise
- Practicing systems thinking and maintaining a contextual understanding of the big picture
- Honing your skills in critical thinking, problem-solving, and foresight
- Building expertise in specialized domains
- Reviewing code, testing, debugging, and quality assurance
- Improving your communication and collaboration skills
- Adapting to change
- Continuously learning, keeping your fundamentals strong while gaining new skills and updating your knowledge
- Using AI

These skills form the human advantage in software engineering. They are durable because they don’t expire with the next framework or tooling change; if anything, AI’s rise makes them more pronounced. Simon Willison has argued that AI assistance actually makes strong programming skills *more* valuable, not

less, because those with expertise can leverage the tools to far greater effect.

A powerful machine in unskilled hands can be dangerous or wasted, but in capable hands it's transformative. In the AI era, an experienced engineer is like a seasoned pilot with a new advanced co-pilot: the journey can go faster and farther, but the pilot must still navigate the storms and ensure a safe landing.

Software engineering has always been a field of continuous change – from assembly language to high-level programming, from on-prem servers to the cloud, and now from manual coding to AI-assisted development. Each leap has automated some aspect of programming, yet each time developers have adapted and found even more to do. As Tim O'Reilly notes, past innovations "almost always resulted in more work, more growth, more opportunities" for developers. The rise of AI is no different. Rather than making developers irrelevant, it is reshaping the skillset needed to succeed. The mundane 70% of coding is getting easier; the challenging 30% becomes an even larger part of our value.

To maximize that human 30%, focus on the timeless engineering skills: understanding problems deeply, designing clean solutions, scrutinizing code for quality, and considering the users and context. Experienced programmers are gaining more from AI because they know how to guide it and what to do when it falters. Those who combine these skills with AI tools will outperform those who have only one or the other. In fact, the consensus emerging among experts is that AI is a tool for the skilled: that "LLMs are power tools meant for power users." This means the onus is on each of us to become that "power user" – to cultivate the expertise that lets us wield these new tools effectively.

Ultimately, the craft of software engineering is more than writing code that works. It's about writing code that *works well* – in a real-world environment, over time, and under evolving requirements. Today's AI models can assist with writing code, but cannot yet ensure the code works well in all those dimensions. That's the developer's job.

By doubling down on the skills outlined above, senior developers can continue to lead and innovate, midlevel developers can deepen their expertise, and junior developers can accelerate their journey to mastery. AI will handle more and more of the routine, but your creativity, intuition, and thoughtful engineering will turn that raw output into something truly valuable. AI is a powerful tool, but it's all about how we use it. Good engineering practices, human judgment, and a

willingness to learn will remain essential.

In practical terms, whether you are pair-programming with an "eager junior" AI that writes your functions, or reviewing a diff full of AI-generated code, never forget to apply your uniquely human lens. Ask: Does this solve the *right* problem? Will others be able to understand and maintain this? What are the risks and edge cases? Those questions are your responsibility. The future of programming will indeed involve less typing every semicolon by hand and more directing and curating – but it will still require developers at the helm who have the wisdom to do it right.

In the end, great software engineering has always been about problem-solving, not just code-slinging. AI doesn't change that: it simply challenges us to elevate our problem-solving to the next level. Embrace that challenge, and you'll thrive in this new chapter of our industry.

---

1 This chapter is based on two essays I first published on my Substack newsletter, *Elevate with Addy Osmani*: "Beyond the 70%: Maximizing the Human 30% of AI-Assisted Coding," first published March 13, 2025, and "Future-Proofing Your Software Engineering Career," first published December 23, 2024.

2 To learn more about systems thinking, check out *Thinking in Systems: A Primer*, 2nd edition, by Donella H. Meadows (Rizzoli, 2008), and *The Fifth Discipline: The Art and Practice of the Learning Organization* by Peter M. Senge (Crown, 2010).

3 For more on this topic, see *Fundamentals of Software Architecture*, 2nd edition, by Mark Richards and Neal Ford (O'Reilly, 2025) and *Head First Software Architecture*, by Mark Richards, Neal Ford, and Raju Gandhi (O'Reilly, 2024).

About the Author

Addy Osmani is a senior engineering leader at Google Chrome, where he works on developer experience, performance, and AI-powered software development tools. He has over 20 years of industry experience building web technologies and has authored multiple books on software engineering best practices.

He has worked extensively with AI-driven developer tools, testing and evaluating emerging platforms like GitHub Copilot, OpenAI Codex, v0.dev, Cursor, and Cline. His writing on AI-assisted software development has influenced thousands of developers, and his leadership at Google Chrome has helped shape the future of web performance and AI-augmented developer workflows.

This book distills his deep expertise in software engineering and his hands-on experience with AI-powered coding assistants, offering developers practical strategies to integrate AI into their daily workflow and adapt to the rapidly changing landscape of software development.