ChatGPT

Software Development in the Age of GenAI: A New Approach to Engineering

By Dinis Cruz and ChatGPT Deep Research

Introduction

Software development practices are undergoing a transformation with the rise of Generative AI (GenAI) tools. Some observers claim that GenAI is "breaking" software engineering by flooding codebases with AI-generated code. In reality, GenAI is not breaking software development – it is exposing how broken our development processes have been all along. Traditional software projects often struggle with poor engineering practices, unclear workflows, and lack of focus on quality. GenAI, if used naively, can certainly accelerate the creation of bad code. But if used wisely, GenAI offers a chance to finally fix long-standing inefficiencies and elevate software engineering to a more mature, productive, and *truly* engineered discipline.

This opinion paper presents a perspective on how to leverage GenAI effectively in software development. We outline key principles for AI-augmented development, highlight common misconceptions, and describe a practical workflow that integrates GenAI into every stage of the development lifecycle. Throughout, we emphasize maintaining code quality, simplicity, and robust testing – ensuring that **developers remain in control and understand the code** being produced. In the GenAI era, success comes not from letting AI churn out masses of mystery code, but from using AI to supercharge good engineering practices.

The Broken State of Software Development (Even Before GenAI)

The uncomfortable truth is that *traditional* software development has never been as rigorous or systematic as we pretend. In many organizations, building software is less like engineering and more like art or craft – reliant on individual heroics, tribal knowledge, and ad-hoc processes. There has long been a lack of focus on **non-functional requirements** (such as maintainability, security, scalability, and clarity), which means teams often prioritize shipping features fast over building sustainable systems. This is reinforced by management structures and incentives that reward hitting deadlines and cranking out code, rather than creating quality architecture and robust testing. The surrounding ecosystem – from hurried project timelines to poorly integrated tools – often works against developers trying to do proper engineering.

In practice, a lot of teams don't truly *engineer* software; they hack at it. The result is codebases riddled with technical debt, minimal documentation, flaky tests (if any), and fragile architectures. Key development activities like design documentation, code review, refactoring, and comprehensive testing are frequently given short shrift. The outcome is that many software projects are harder to maintain and scale than they should be. In short, **software development was "broken" in many ways even before GenAI arrived**. Generative AI is simply shining a spotlight on these chronic problems by accelerating whatever processes (good or bad) we already have in place.

Misconceptions About GenAI and Coding Productivity

The advent of GenAI coding tools has led to several misconceptions. One is the belief that GenAI can automatically make development hyper-productive by generating tons of code on demand. Yes, modern AI copilots can produce code snippets or even entire functions in seconds – but *more code is not the same as better software*. In fact, measuring productivity by lines of code has always been a mistake. We saw this during past outsourcing booms when teams treated code quantity as a metric, only to create bloated, unmaintainable systems. GenAI poses the same risk: **if you use AI to generate code that nobody on your team fully understands, you are only accelerating your journey into a maintenance nightmare.**

Early experiences have shown that indiscriminate use of GenAI can lead to a burst of initial progress followed by serious slow-downs as projects near production. Developers might accept AI-generated code at face value and later discover it's full of bugs, poorly structured, or incompatible with the rest of the system. Reports already indicate that AI-generated code can be significantly *more error-prone* than human-written code. For example, one analysis of 470 pull requests found that AI-produced code had **1.7 times more issues** on average compared to human code [1]. The biggest weaknesses were in **code quality and readability**, which are exactly the issues that "slow teams down and compound into long-term technical debt" [2] [3]. In other words, GenAI tends to **amplify** existing problems if teams blindly accept its output – it accelerates coding, but also accelerates the accumulation of bugs and messes that must be fixed later [4].

Another misconception is that GenAI can replace disciplined engineering. Some less experienced developers have treated GenAI as a shortcut to avoid learning fundamentals, resulting in what might be called "*vibe coding*" – slapping together AI-suggested code without fully understanding it or how it fits architecturally. This approach might create a *flashy prototype*, but it quickly hits a wall. Without strong engineering principles, the codebase becomes an incoherent tangle of half-understood components. Indeed, studies are finding phenomena like a massive **8× increase in duplicated code** when AI tools are used without supervision [5]. A "quick and dirty" AI-assisted code dump can give the illusion of progress, but it really just defers the complexity – you end up paying the price during integration, debugging, and maintenance. One industry report noted that teams actually began spending more time debugging AI-generated code and fixing vulnerabilities, essentially nullifying the supposed productivity gains [6].

The lesson: GenAI is not a silver bullet that magically fixes broken processes or poor skills. If you feed bad practices into the system, GenAI will happily accelerate the production of bad code. The goal should not be to churn out as much code as possible with AI. Instead, the goal must be to leverage AI to improve the quality and **simplicity** of our code and to streamline the engineering workflow. In the next sections, we discuss how focusing on simplicity, clarity, and testing – the hallmarks of good engineering – can turn GenAI into a powerful ally rather than a source of technical debt.

Simplicity and Clarity: Leveraging GenAI for Better Code, Not Just More Code

The true potential of GenAI in software development lies in creating **simpler, cleaner code** and better-designed systems, faster than before. This may sound paradoxical – how does an AI that can generate reams of code result in *less* complex software? It all comes down to how the AI is used. The key is to continuously enforce clarity and simplicity at every step where GenAI is involved, effectively using the AI as a refactoring and design assistant rather than a blind code generator.

When I incorporate GenAI into coding, I make it a strict rule that **the developer (or team) must understand every line of code that goes into the codebase**. If an AI suggestion produces convoluted or opaque code, we treat that as a problem to be solved – not a gift to be accepted without question. In practice, this means we use the AI to *simplify* aggressively. We can prompt the AI with instructions like: "Refactor this function to be more readable," "Simplify this logic using clearer abstractions," or "Break this large function into smaller, well-named pieces." The amazing thing is that GenAI can do this refactoring in seconds, providing multiple iterations quickly until the code is straightforward. We finally have a tool that can rewrite and reorganize our code almost on demand, which enables a level of iterative refinement that was previously cost-prohibitive.

Crucially, developers should guide this process by applying sound software design principles (such as DRY, single responsibility, modularity, etc.). The AI does not inherently know what code is "simple" or "clean" in the context of your project – it has to be steered. **Think of GenAI as a very fast junior developer who can rewrite code endlessly without tiring.** It's up to the senior developers/architects to ensure that the rewrites are heading in the right direction. If used well, GenAI can eliminate a lot of grunt work and actually *reduce* complexity by helping create clear abstraction layers and removing duplicate or dead code. In fact, some evidence shows that in teams with strong engineering discipline, AI becomes a productivity *supercharger* for exactly this reason – clean, well-structured systems let AI focus on acceleration, whereas tangled systems just generate more tangles [7].

It's helpful to consider *when* different levels of code complexity are acceptable. Here I draw on **Wardley Mapping** concepts: all technology evolves through stages from *Genesis* (novel, experimental) to *Custom-Built*, to *Product*, and finally *Commodity* [8]. In the early "Genesis" phase of a project – when you're exploring a completely new idea or feature – it might be okay to do some quick-and-dirty experiments with GenAI (akin to prototyping). At that stage you are a *pioneer*, trying to prove something works. If the code is a bit messy or not fully understood, that's tolerable temporarily **as long as it's throwaway** or isolated. However, **as soon as you transition into building a real product** (moving into the custom-built or product phase), the engineering rigor must kick in. You then become a settler or town planner, turning that prototype into a stable village or city. GenAI can and should still be used, but now its role is to assist in *engineering* – writing production-quality code, adhering to standards, and generating tests and docs. The focus moves to simplification, consistency, and reliability.

In summary, **GenAI should be directed toward making code easier to understand, not harder**. Every time an AI produces code, ask: *Is this the simplest approach that achieves the goal? Could we make this clearer?* If the developer cannot easily explain the AI-generated code to a colleague, then it needs revision. By keeping simplicity and clarity as north stars, GenAI becomes a powerful tool to reduce complexity in our systems. It allows us to create high-level abstractions quickly and then refine them, whereas in the past such refactoring might be deferred or skipped due to time constraints. The difference is a subtle shift in mindset: we are not using GenAI to generate **masses of code**; we are using it to generate **better code** – code that we would be happy to maintain and build upon.

The Critical Role of Testing and Continuous Refactoring

Testing is the backbone that supports this new AI-augmented development paradigm. In fact, one could say **automated tests are more important than ever** in the age of GenAI. Why? Because tests provide the safety net that allows developers to incorporate AI-generated code confidently and to refactor relentlessly. With a solid test suite in place, you can let the AI suggest changes or new modules and quickly detect if anything breaks established functionality. Tests turn potentially risky AI outputs into controlled, deterministic updates.

It's important to recognize that *testing is not just about catching AI mistakes; it's what enables improvement.* A comprehensive set of unit and integration tests gives you the freedom to continuously improve and simplify code (with or without AI) without fear. If an AI refactors some code, or if you ask it to re-generate a function in a cleaner way, you can run the tests to verify nothing essential was lost or altered. In essence, tests **grant you the confidence to refactor** – you know that if something does break, the tests will flag it, and you can fix it immediately. This feedback loop encourages an iterative approach where code can evolve towards higher quality.

My workflow heavily emphasizes having the AI generate tests alongside the code. Whenever a new feature or module is created, I prompt the GenAI to produce corresponding tests (if it hasn't done so by default). This serves two purposes: (1) it validates that the code works as expected, and (2) it actually helps *document the intended behavior* of the code. Often I will examine the AI-generated tests to understand what assumptions the model made about the code's behavior. If the tests reveal misconceptions or edge cases, that's a sign to refine either the code or the tests. Remember, tests are also living documentation.

Experience and industry data both underscore the need for this vigilant testing approach. As noted earlier, AI-generated code can introduce more bugs and even security vulnerabilities than traditional code if left unchecked [3]. Without tests, those issues might lurk until much later in the development cycle or production. Moreover, code that is never refactored (which is likely if you lack tests to catch regressions) will accumulate into a larger ball of mud. A telling statistic showed that a surge in AI usage correlated with a measurable decrease in software delivery stability [9]. The likely cause is teams racing ahead with AI-produced changes without sufficient testing and design oversight. To avoid that fate, **testing must be a first-class citizen in the GenAI workflow** – not an afterthought. The mindset should be: if the AI writes code, it also better help write the tests for that code.

In summary, a robust test suite plus GenAI is a recipe for success: the tests keep the AI-generated (and human-generated) code aligned with the intended behavior, and they give developers the freedom to continually improve code structure. This way, **technical debt is kept under control** even as we rapidly add new code with AI assistance. Testing and refactoring go hand-in-hand – and GenAI can accelerate both, as long as we ensure one is never done without the other.

AI-Augmented Development Workflow: From Idea to Deployment

How can we practically integrate all these principles – simplicity, testing, iterative refinement – into a real development process with GenAI? In this section, I will describe a *workflow blueprint* that I have been using with great success. This workflow takes an idea from conception all the way to deployed software, with GenAI involved at every step in a controlled and purposeful manner. The focus is on harnessing AI for acceleration **while maintaining human oversight and clarity**. The process can be broken down into distinct stages:

1. **Ideation and Requirements Capture:** Everything begins with an idea or problem to solve. Instead of writing a formal spec straight away, I often start by recording a **voice memo** or informal description of the idea in plain language. This captures the raw vision quickly. Then, using a GenAI tool (e.g. OpenAI's ChatGPT with Deep Research, or Anthropic's Claude), I transcribe and **summarize the voice memo** into a written initial outline. The AI here functions as a notetaker and brainstorming partner, helping to organize the idea into a rough structure or list of requirements. This initial brief is typically high-level – it describes what we want to achieve, the key features or components, and any constraints or goals (like "must be secure", "should

handle 1000 req/sec", etc.). Essentially, this is the *brief to create a more detailed brief*. It allows me to see the idea in writing and sets the stage for deeper design.

2. **Interactive Architectural Design (AI-Assisted):** With the initial concept in hand, I move to designing the architecture of the solution. I initiate a **conversation with an LLM** (ChatGPT or Claude) dedicated to *architecture and design*. Here I explicitly instruct the AI **not to generate any actual code yet**. The aim is to discuss and refine the system's design: what components or modules are needed, how they will interact, what data structures or APIs might look like, and so on. This is a back-and-forth process – I might propose an approach, the AI might highlight potential issues or alternatives, and we iterate. Essentially, the LLM serves as a brainstorming partner and rubber duck, allowing me to articulate the design and get instant feedback or reminders. During this stage, I also infuse any high-level *non-functional requirements* into the discussion (e.g. "We need to prioritize security and scalability from the start"). By the end of this phase, I have a much clearer picture of the system's architecture and a list of components or tasks needed to implement it. We've effectively sketched the blueprint of the solution without writing a single line of code.

3. **Detailed Technical Specification:** Once the architecture is agreed upon, the next step is to produce a **detailed technical spec or design document**. I ask the LLM to help generate this specification based on our architectural discussion. The spec includes the list of modules/functions, their responsibilities, interfaces between components, data models, and even pseudocode or specific algorithm outlines for critical sections. I often have the AI **incorporate rationale** for decisions (e.g. why using approach X over Y) so the spec can double as documentation of our thought process. Additionally, I might request the AI to produce **architecture diagrams in ASCII art** as part of the spec. Surprisingly, modern LLMs can create decent ASCII diagrams that illustrate, for example, component interactions or request flow – which adds a visual element to the text. (Using ASCII art ensures the diagrams are easily editable and source-controlled as plain text.) At times, I also leverage tools like Google *NotebookLM* to generate supporting materials like an infographic or slide deck from this spec, especially if I need to communicate the design to other stakeholders. The outcome of this stage is a **comprehensive brief that a developer (or an AI) could implement**. It's essentially the "design document" or "implementation plan" one would give to a development team, written in a highly structured and explicit manner. The document can easily run multiple pages (hundreds of lines), but that's a feature, not a bug – it's far easier to revise a design document than to rewrite code. We treat this spec as a living plan: we can still go back and forth with the AI to tweak or refactor the *plan on paper* until it looks solid. All of this happens before actual coding, ensuring that by the time we write code, we have minimized ambiguity.

4. **Code Generation and Implementation:** Now comes the part where we generate actual code. I start a **fresh LLM session** dedicated to coding (this avoids any confusion the AI might have from the prior lengthy discussion, keeping the context focused). Into this session, I feed the detailed spec from the previous step, along with any *coding guidelines* or preferences I have. Over time, I've built up guidance documents for the AI like "Coding style guidelines", "How to structure unit tests", "Type safety rules", etc., which I include to steer the AI's output. I then prompt the AI to **implement the design exactly as specified**, file by file or module by module. The AI will now generate the code for each component described in the spec. Because the spec is so detailed, the AI's code tends to be much more aligned with my expectations than if I had just asked for code from a vague prompt. I also explicitly ask for **unit tests** for each major function or module. At this stage, I often get a large volume of code as output (sometimes dozens of functions or several files worth of code). It's important to note that *I do not blindly accept this code*. The role of GenAI here is to produce a first draft implementation at high speed. The code is typically

syntactically correct and roughly follows the design (since the AI had the spec), but it may have small logical errors or opportunities for improvement. I take the AI-generated code and **integrate it into my development environment manually**, usually by copy-pasting into the appropriate files or using provided file outputs. This manual step forces me to examine each part of the code as I bring it in.

5. **Developer Review, Debugging, and Refactoring:** After inserting the AI-generated code into the project, a thorough **code review** is performed (by me or the team's developers). We run the automated tests that were generated (as well as any existing test suite) to see what passes or fails. Inevitably, there will be some failing tests or unintended side effects – this is expected, just as if a junior developer wrote the code and needs feedback. We fix any bugs uncovered by tests or adjust the code where the AI's output didn't perfectly match the intended behavior. More importantly, we assess the **readability and structure** of the code. If any part of the code is overly complex or cryptic, we take the time to refactor it. Sometimes I'll loop back with the AI at this point: for example, if a function is too long or an algorithm is hard to follow, I might prompt the LLM, *"Refactor this function into smaller ones with self-explanatory names,"* or *"Simplify this logic while retaining functionality."* The AI will provide a refactored version, which I then verify and integrate. This is an iterative mini-cycle of code-generation -> review -> improvement, but it happens rapidly thanks to the AI's assistance. The end goal is that **every piece of code meets our quality bar**: simple, well-documented (via code comments and tests), and fitting cleanly into the overall design. By the end of this stage, we have a codebase that implements the original idea, passes all tests, and is understood by the development team. It's worth noting that at this point, **the volume of code written by the AI is substantial, but we have not lost control of it** – we've reviewed and understood it all. If we did not understand it, it wouldn't be allowed in the codebase without further refactoring. This practice ensures we aren't importing mysterious black-box code.

6. **Continuous Integration and Deployment:** With the new code reviewed and all tests green, we merge the changes and let our **CI/CD pipeline** do its job. In my workflow, every code commit triggers automated tests and then deploys to a staging or testing environment (and sometimes even a production environment behind feature flags, depending on the context). This immediate deployment capability is crucial. It ensures that the AI-generated code isn't just theoretically correct, but is actually exercised in a realistic environment as soon as possible. Any integration issues (with databases, external services, environment configs, etc.) surface early. We treat deployment as a non-event – something that happens continuously – which is another hallmark of a healthy development process. GenAI fits into this just fine: since we had high confidence via tests, deploying AI-assisted code is no scarier than deploying human-written code. In fact, often it's smoother because the code quality is high from the get-go (given all the prior steps).

7. **Documentation and Debrief:** One of the final (and often overlooked) steps is creating proper documentation for what was built. This is another area where GenAI shines. Now that the feature is implemented and deployed, I prompt the LLM to **generate documentation and a debrief**. I provide the LLM with the final source code (or the relevant parts of it) and ask it to produce a *detailed explanation* of how the system works, effectively an updated spec that reflects the actual implementation. This debrief document outlines each module's behavior, any deviations from the original plan, and key decisions or fixes that were made during development. It can even include a section on "lessons learned" if the implementation uncovered new insights or required changes to initial assumptions. This updated spec is extremely useful for future maintainers – it's like having up-to-date design docs that match the code. Furthermore, I often ask the AI to generate **user-facing documentation** as well, such as an **API guide or usage manual** for the feature we just built. This describes how an end-user (or

another developer/team) can use the new functionality, without delving into internal code. Because the AI can parse the code and tests, it is able to enumerate examples, edge cases, and usage scenarios in the documentation. Finally, for completeness, I may use NotebookLM or a similar tool to convert some of this documentation into visual slides or diagrams for presentations or onboarding material. By the end of this stage, we have not just working code, but also a full suite of documentation: from the initial design spec (reconciled with reality), to developer-oriented docs, to end-user docs. All of this was accelerated by AI, which drastically reduces the drudgery that often leads teams to skip writing documentation. Now the entire lifecycle of the feature – from idea to design to code to test to deploy to docs – has involved GenAI in a supportive, quality-enhancing role.

The AI-augmented workflow can be visualized as follows:

```
Idea (Voice Memo)
   |
   v
Transcribe & Outline (ChatGPT/Claude)  — Initial concept brief
   |
   v
Architecture Design Q&A (LLM)  — Define components & plan
   |
   v
Technical Specification (LLM)  — Detailed design + diagrams
   |
   v
Code Generation (LLM)  — Implement code and tests
   |
   v
Code Review & Refactoring (Developer)  — Ensure clarity & quality
   |
   v
Testing & CI Deployment (Automation)  — Validate in environment
   |
   v
Documentation & Debrief (LLM)  — Update specs and user docs
```

Using this workflow, I have found that my productivity as a developer-architect has skyrocketed. Projects that once might have taken weeks can often be delivered in days without sacrificing quality. The secret is that GenAI is leveraged in **multiple dimensions** – not just writing code, but also in planning, documenting, and communicating. Each stage feeds into the next, with human oversight guiding the AI at critical points. By the time code is written, it is well-specified and thus largely correct. By the time code is deployed, it is well-tested. By the time the project is "done," we even have comprehensive documentation. These are practices that we always *knew* were ideal in software engineering, but they often fell by the wayside under tight schedules. Now, GenAI makes it feasible to actually do all of them without an army of extra staff. It's as if every developer has a tireless assistant for each aspect of development, from initial proposal to final documentation.

GenAI Architects and Developers: Evolving Roles and Collaboration

The introduction of GenAI into software development is not just changing how we write code – it's also changing the **roles and collaboration dynamics** within teams. Two emerging roles (or skill sets) can be identified in this new landscape: **GenAI-augmented architects** and **GenAI-augmented developers**. In practice, a single person might wear both hats (as in my own workflow), but it's useful to distinguish them conceptually.

- **GenAI Architect:** This is a software architect or senior developer who specializes in orchestrating AI tools throughout the development process. Their focus is on system design, enforcing quality standards, and steering the AI to produce the desired outcomes. In the workflow described above, the activities of conducting architectural discussions with the LLM, preparing detailed specs, and establishing coding guidelines are part of the GenAI architect's role. This person acts as the *strategist*, ensuring that the project's direction is sound and that the AI is used effectively rather than haphazardly. A GenAI architect needs a strong foundation in software engineering principles because they have to encode those principles into prompts and reviews. In essence, they translate high-level objectives into concrete tasks for the AI and then vet the AI's outputs against those objectives.

- **GenAI Developer:** This is a developer who actively uses AI tools to implement and maintain software, while following the structure laid out by the architects. They need to be adept at working with AI-generated code – meaning they can read it critically, test it, debug it, and modify it as needed. The GenAI developer treats AI suggestions like the work of a junior developer on the team: something to be reviewed and improved. This role involves a lot of learning and upskilling, because the developer must understand not only the base programming languages and frameworks but also how to collaborate with an AI (prompting techniques, understanding AI limitations, etc.). When every developer has an AI pair-programmer, the developers who excel will be those who can effectively supervise and guide that AI. In other words, they become **AI literate** developers.

In smaller teams or individual projects, one person might fill both roles – conceiving the architecture and also doing the AI-assisted implementation. In larger organizations, we might see a more formal distinction where architects focus on high-level design and interface with AI to generate specs, while developers consume those specs with the help of AI to produce code. Either way, one thing is clear: **the presence of GenAI elevates the importance of strong senior engineering leadership.** If junior developers lean heavily on AI for code, someone with experience must be in the loop to ensure that what's being built isn't a fragile house of cards. Recent industry observations back this up: teams that succeed with AI have seasoned engineers who "*guide AI tools strategically, not blindly*" and who refactor AI outputs into clean architectures [10]. Without that oversight, using AI in development can become a "*game of tech debt roulette*" [11] – you might get lucky for a while, but eventually the accumulating flaws will cause serious trouble.

Another interesting development is the collaboration between **business domain experts and software engineers** through the medium of GenAI. In the past, if a non-developer (say a business analyst or a marketer with an idea for a tool) wanted to create a software prototype, they either had to learn to code or hand-wave their requirements to an engineering team. Now, with GenAI, it's feasible for these domain experts to create a rough application or script by themselves – let's call it a *prototype via prompt*. For example, a financial analyst might use an AI coding assistant to whip up a quick data analysis app or an Excel automation. This is empowering, but it only goes so far. When the prototype needs to become

a robust, production-ready system, there still must be a hand-off to professional developers. We are starting to see workflows where a business user creates an initial solution using GenAI (perhaps filled with "happy path" logic and some fragility), and then a developer takes over to harden it, add tests, and integrate it properly. The positive side of this is that business folks are becoming more familiar with the software development process, and they appreciate the need for things like testing and refactoring because they've hit those limits themselves. The collaboration becomes tighter: instead of just throwing requirements over the wall, the domain expert can present an actual working prototype. The engineers then use their expertise (often with AI help as well) to rebuild or improve it in line with best practices.

In effect, GenAI is **blurring the lines** between roles slightly – enabling non-developers to do a bit of development, and enabling developers to focus more on high-level problem solving. But it's also reinforcing the need for classical software engineering skills. Architecture, design, and quality assurance become even more crucial when code can be generated so easily. It's like having a powerful race car: it can go fast, but you need a skilled driver at the wheel and a pit crew to keep it from crashing. Senior developers and architects are that skilled driver and crew. They ensure that speed comes with control. Indeed, companies adopting AI in coding have learned that *AI will amplify whatever is in place*. As one analysis succinctly put it: **"AI amplifies your technical posture, good or bad"** [12]. If you have solid engineering practices, AI will boost productivity and quality. If you have poor practices, AI will dig you into a deeper hole faster. Therefore, investing in training developers (and even training the AI via good prompts and examples) is non-negotiable.

To thrive in this GenAI-enhanced environment, developers should cultivate skills like: prompt engineering (how to ask AI for what you need), critical review of AI outputs, data security awareness (since AI might introduce insecure code if not guided [13]), and an architecture mindset. We may also see new **development methodologies** emerging to integrate AI – for example, AI pair-programming norms, AI code review checklists, and so forth. Teams will likely include AI-specific review stages (e.g. have the AI explain its code to a human reviewer, which is something one can literally ask the AI to do as part of code review).

Overall, the roles in software teams are shifting towards a more collaborative interplay between human expertise and AI assistance. *GenAI architects* ensure that the overall direction and design are sound and that AI efforts align with the bigger picture. *GenAI developers* use the AI day-to-day to crank out features while maintaining quality and understanding. And even *non-developers* are now part of the software creation process, thanks to more accessible AI tools – but their contributions need refinement by developers to truly stand up in production. This collaboration, when managed well, can result in dramatically faster development cycles without sacrificing reliability. The human team members define the **what** and **why** of the software, while the AI helps with the **how** at a micro level, under human guidance. In the end, software engineering becomes more about directing and validating the work of AIs (and integrating components together) than about typing out boilerplate code. This is a welcome evolution, as it frees human developers to be more creative and focus on solving the right problems.

Conclusion

Generative AI is poised to fundamentally change how we develop software – but the outcome of this change depends entirely on how we wield these new tools. If we simply use GenAI to speed up an already chaotic development process, we will amplify the chaos. We will end up with brittle systems full of hidden bugs, security holes, and pile-ups of technical debt that will eventually slow us to a crawl. On the other hand, if we integrate GenAI thoughtfully into a disciplined, test-driven, simplicity-focused workflow, we can achieve breakthroughs in productivity and software quality at the same time. The experiences and workflow described in this paper show that it's possible to drastically accelerate

development **and** end up with cleaner, well-documented, maintainable codebases. In fact, many of the long-standing "best practices" of software engineering (like writing good specs, doing thorough code reviews, keeping documentation up to date) become *easier to follow* when you have an AI assistant to share the load.

Ultimately, GenAI doesn't replace the need for human judgment – it elevates the importance of human judgment. The role of developers and architects becomes less about cranking out code and more about **making decisions**: what architecture to use, which trade-offs to accept, how to properly frame the problem for the AI, and how to verify the solutions. In a way, software engineering may become more strategic. Teams that succeed with GenAI will be those that remain **objective and focused** on their engineering principles amidst the hype. They will treat AI outputs as suggestions to be evaluated, not gospel to be followed blindly. They will double down on practices like testing, modular design, security reviews, and knowledge sharing – because these provide the guardrails that allow AI to be used at high speed safely. As one report aptly noted, *AI won't magically fix your software problems; it will just make them appear faster if you're not careful* [14]. The converse is also true: if you build carefully, with GenAI as a helper, you can reach new heights of scalability and innovation.

In closing, the age of GenAI can usher in a renaissance in software development. It can free us from grunt work and let us focus on creativity and design. It can help us finally generate the documentation and tests we always knew we should. It can enable more people to participate in software creation, breaking down silos between “the business” and “the developers.” But realizing this potential requires a mindset shift: **speed is only beneficial if headed in the right direction**. By pairing GenAI’s speed with a clear vision, strong engineering discipline, and a commitment to simplicity and understanding, we ensure that we're moving fast **in the right direction**. In that scenario, GenAI isn't breaking software development – it's helping to fix it, turning a broken craft into a truly engineered, scalable, and sustainable practice for the future.

---

[1] [2] [3] [4] [13] AI Code Is a Bug-Filled Mess
https://futurism.com/artificial-intelligence/ai-code-bug-filled-mess

[5] [6] [7] [9] [10] [11] [12] [14] AI Is Changing How We Code. But Is Technical Debt the Price Tag?
https://inclusioncloud.com/insights/blog/ai-generated-code-technical-debt/

[8] Evolution Stages - Strategic Terms | Wardley Maps
https://www.wardleymaps.com/glossary/evolution-stages