ChatGPT

Vibe Coding and AI-Assisted Development

“Vibe coding” refers to using conversational AI to generate software from natural-language prompts. As one cloud provider puts it, this approach promises that **“millions of non-coders”** could build and launch applications in seconds【1†L11-L19】. In practice, vibe coding involves iterative prompting and refinement: a user describes a goal, the AI writes code, the user tests it, gives feedback, and repeats【1†L49-L58】【41†L159-L169】. This can dramatically speed up prototyping and lower the barrier to entry. In the extreme “pure” form, developers *“forget that the code even exists”*, accepting AI output without scrutiny【41†L159-L169】. In the more responsible model, developers act as *co-pilots*, critically reviewing and testing AI-generated code【41†L159-L169】【41†L197-L206】.

Vibe coding is increasingly common: industry surveys report **90%** of developers at medium and large firms use AI assistants, and 30–40% of organizations actively promote AI-generated code to cut costs and accelerate development【29†L115-L123】. Dozens of new tools (Cursor, Replit, etc.) and major cloud services are built around this paradigm. There is evidence that even **novice programmers** can leverage AI to produce working code. For example, a study of introductory Java students found that those using ChatGPT wrote code with **fewer style violations and significantly lower cyclomatic complexity** than a control group【12†L85-L93】. In other words, AI helped beginners write *cleaner* code according to static analysis rules. Likewise, non-technical founders report using AI to build MVPs and simple apps in hours or days – though they often emphasize that they still **needed to learn and verify** what was happening under the hood【24†L35-L44】【29†L75-L84】.

In short, AI code generation *can* empower people with little coding background to create *functional prototypes* or small applications. A gap in user skill is partially bridged by the AI’s ability to produce boilerplate code. However, **production-grade software** – i.e. code that is reliable, performant, secure, maintainable and compliant in a real-world, at-scale environment – imposes many additional requirements. These requirements often involve design and operational considerations that AI tools do not inherently address. Below we review the evidence around both sides of this question.

Evidence of Non-Programmers Building Apps with AI

Several studies and reports suggest that AI tools can indeed help novices build useful software more quickly than they could alone. In education, controlled experiments found that students with no prior Java experience who used ChatGPT to implement assignments produced code that *adhered more closely to coding standards and was less complex* than code from students who did not use AI【12†L85-L93】. Qualitative surveys also indicate that beginners feel ChatGPT helps them understand logic and syntax: they learn by inspecting the generated code and asking follow-up questions.

In industry and open-source, anecdotal cases have appeared of non-technical or early-stage developers launching apps with heavy AI assistance. For instance, one marketing startup founder (a “rookie coder”) announced he had built an entire SaaS app by feeding prompts to an AI (Cursor) and never manually writing a line of code【29†L45-L54】. He described *“plugging prompts into an AI tool and let[ting] the system do the*

work"【29†L45-L54】. While this particular app was quickly exploited (discussed below), the fact that a complete app could be scaffolded without a skilled engineer is notable. Startups and citizen developers are indeed trying this approach: industry reports note that entrepreneurs and IT "citizen developers" are beginning to adopt AI builders for rapid prototyping and low-code automation (e.g. Lead dev tools on Replit, Firebase AI Studio, Glide with AI, etc.).

However, **most such builds are essentially prototypes or proofs-of-concept**. They typically target small scale – for example, an app with basic CRUD features, login screens, or simple data processing – and they use managed platforms that handle much of the infrastructure. In practice, experienced developers *using* AI have slowed down in some rigorous tests: one RCT with seasoned open-source devs found that those with access to advanced AI tools (Claude, GPT-4, etc.) actually took **19% longer** on average to complete issues than those without AI assistance【20†L93-L101】. (Dev expectations were the opposite – they *thought* AI would speed them up by ~20%, but they were slower.) This suggests that when real understanding and context are required – as in complex codebases – AI can introduce overhead. Novices may not face the same cognitive overhead, but they also lack the expertise to catch subtle bugs or design flaws that arise.

Key takeaway: AI tools can dramatically accelerate development of simple programs and assist beginners in writing syntactically correct code. Empirical studies show novices writing *better-structured* code with AI's help【12†L85-L93】. Industry anecdotes corroborate that non-programmers can get functional prototypes or minimal viable products running using generative AI. However, **building robust, production-ready software is a much higher bar**. It generally still requires expertise to architect the system, handle edge cases, and integrate testing and security measures.

Risks of Limited Expertise in Vibe-Coded Software

Even as AI lowers the coding bar, production software demands concern for **architecture, scalability, maintainability, testing, security, privacy, and compliance**. There is strong evidence that lacking these skills leads to serious risks in AI-generated code:

- **Architecture and Scalability:** AI tends to focus on *coding tasks*, not on high-level design. A novice prompting for a feature may get a working function or endpoint, but they may not consider whether it fits a scalable service architecture. For example, AI might generate a synchronous workflow for a task that in production should be asynchronous or distributed. Novices may not realize the need for message queues, caching, horizontal scaling, or database sharding, resulting in designs that break under load. We find no formal study quantifying this gap, but industry warnings are clear: vibe coding *"creates a development culture where pre-production security (and, by extension, architecture review) is often bypassed entirely,"* and it *"empowers non-technical developers to rapidly create applications… completely outside the purview of IT and security teams"*【3†L129-L138】. In effect, entire systems can be spun up that have no coherent architectural plan. Without expert input, this risks single points of failure, unmonitored performance bottlenecks, and brittle integration patterns.

- **Maintainability and Technical Debt:** AI-generated code can be cryptic or suboptimal. One study of Copilot's code in GitHub projects found that many snippets contained *"38 different CWE categories"* of weaknesses【9†L72-L80】. Even if the code "works", it may lack clear documentation or conventional structure. Novices often do not know best practices like using consistent naming, writing comments, or factoring code into modules. This can quickly create technical debt: code that only the AI

understands, or that depends on magic prompts. As Contrast Security notes, heavy reliance on AI can *"cause an erosion of foundational developer skills, creating a dangerous 'comprehension gap' where teams can no longer effectively debug or respond to incidents in production"*【41†L188-L193】. Without programmers who understand the code, bugs will be hard to fix and the codebase hard to evolve.

- **Performance and Reliability:** AI generators typically emphasize correctness over efficiency. They may produce solutions that work on small inputs but degrade badly under heavy use. For example, an AI might write a simple loop or database query without considering time complexity, or it might default to in-memory data structures that won't scale. Novices may not notice until performance starts to suffer. Similarly, reliability issues (race conditions, lack of retries, unhandled exceptions) are common. Without knowledge of concurrency, fault-tolerance, and logging, generated code may silently drop errors or crash unpredictably. We found no specific peer-reviewed study on AI code performance, but industry warnings frequently point out that AI tools *"are optimized to provide a working answer, fast. They are not inherently optimized to ask critical security [or robustness] questions, resulting in code that is insecure by default"*【30†L77-L86】. By extension, performance optimizations (like caching or load handling) may also be absent.

- **Testing and Observability:** Vibe coding moves quickly to a working prototype, often without building in test suites or monitoring. As one post-mortem noted, an AI-generated app might launch with *"not a single catastrophic bug, but the absence of the review step that would have caught an ordinary, well-understood configuration mistake"*【33†L211-L219】. Indeed, in one real incident a startup's entire user base was broken because the AI mis-translated code that should generate unique IDs: it hard-coded the same UUID for every new user, causing a catastrophic outage that lasted days【24†L35-L44】. This flaw worked through development and testing (AI tools often return code that runs but is semantically wrong) and was only caught when real users signed up. In that case, *no automated test covered the user signup flow* (if it had, it would have revealed the duplicate ID error)【24†L100-L109】. This illustrates the risk: novices may ship without writing unit tests, integration tests, or end-to-end tests. Observability is also lacking: there may be no logs or metrics to tell *why* the system fails under load. Contrast Security warns that vibe coding *"produces insecure code at a velocity that outpaces traditional security processes, placing unaudited code directly into production"*【41†L159-L169】【41†L219-L227】. By analogy, the same applies to quality assurance processes in general.

- **Security Vulnerabilities:** A large body of empirical research shows that AI-generated code often contains serious security flaws. For example, Pearce et al. (2021) found that about **40%** of 1,689 programs generated by GitHub Copilot were vulnerable to attack【6†L35-L40】. More recent work confirms similar risks: a 2024 study of actual Copilot-generated snippets in GitHub projects found **32.8% of Python and 24.5% of JavaScript** snippets had security weaknesses (spanning dozens of CWE categories, including critical ones like command injection)【9†L72-L80】. An ETH Zurich/ LogicStar benchmark (BaxBench) demonstrated that even the *best* LLMs fail to generate deployment-ready code: **62%** of AI solutions were either incorrect or contained vulnerabilities, and roughly half of the "correct" solutions were still insecure【5†L28-L36】. In practice, this means that code coming from AI often replicates common mistakes (SQL injection, improper auth, buffer overflows, etc.) because the models were trained on public code that contains such patterns. Industry reports echo these findings: e.g. a Veracode study found **45%** of over 100 AI-generated code samples failed security tests and introduced critical web app risks【29†L136-L144】.

Security flaws from AI-generated code can be subtle. Unit42 (Palo Alto) summarizes documented incidents where vibe-coded software lacked basic protections: *"A sales lead application was successfully breached because the vibe coding agent neglected to incorporate key security controls such as authentication and rate limiting"*【30†L60-L69】. Other case examples include a prompt-injection exploit causing arbitrary code execution, an authentication bypass due to exposing internal IDs, and even an AI agent that inadvertently deleted a production database despite instructions not to touch it【30†L60-L69】. Crucially, these are *real incidents* or experiments, not hypothetical. Many vulnerabilities show up only at runtime; traditional static scanners often miss them because AI-generated business logic can be complex and non-idiomatic 【41†L174-L182】【41†L221-L229】.

- **Privacy and Data Handling:** Besides code quality, vibe-coded apps risk mismanaging sensitive data. AI tools do not automatically enforce privacy best practices. They may generate code that, for example, leaks personally identifiable information (PII) into logs, mishandles encryption keys, or allows data to be exposed via APIs. OWASP's GenAI project warns that embedding LLMs risks inadvertently exposing sensitive data through outputs, and that data sent to an AI (or learned by it) may be insecurely stored or processed【38†L12-L20】. Moreover, novices often do not design proper data flows: they might forget to sanitize inputs, fail to implement proper consent or data retention policies, or leave test credentials in production. SecurePrivacy (a privacy engineering blog) highlights that AI-driven tools typically skip essential "compliance plumbing": audit logs, consent mechanisms, access controls, and data residency rules rarely emerge spontaneously from a prompt【33†L279-L288】【33†L308-L317】. For example, an AI-generated signup form might default to a single "I agree" checkbox (non-compliant) rather than collecting granular consents, or it might configure a database with *"overly permissive rules"* that expose an entire dataset because no one explicitly asked for fine-grained controls【33†L211-L219】【33†L308-L317】. The result can be serious: IBM's 2025 data breach report attributes an extra **$670,000** average cost to breaches caused by "shadow AI" apps that escaped IT governance【33†L239-L247】. In short, **legal and privacy compliance will not be magically solved by AI**. Organizations remain data controllers under GDPR/CCPA regardless of who (human or AI) wrote the code, and novices may be unaware of these obligations【33†L254-L263】.

- **Authentication, Secrets, and Supply Chain:** AI tools often generate example or placeholder authentication logic rather than robust solutions. One concrete failure mode: an AI might insert a hard-coded API key or use a weak JWT secret. Contrast and Unit42 note that AI can easily "recommend vulnerable third-party libraries or introduce code with restrictive licenses" because they optimize for running code, not legal safety【41†L188-L193】. A particularly pernicious risk is *dependency hallucination*: LLMs sometimes fabricate plausible-sounding libraries or functions that don't exist, leading developers to try to use them and breaking the build. Unit42 calls this the "phantom supply chain risk"【31†L1-L4】. In real cases, we have seen prompts accidentally include special characters or random code (often because of copy/paste issues) that crash the system in production, requiring debugging time novices lack.

In summary, limited software-engineering knowledge combined with AI coding produces blind spots. Errors in *"architecture, data flow, or security"* that a trained engineer would catch simply slip by. Contrast Security bluntly states that *"Undisciplined vibe coding leads to systemic insecurity,"* noting research confirming **40–62% of AI-generated code contains flaws**【41†L174-L182】. Moreover, the lack of human review means these flaws often end up *in production* where they may remain hidden until exploited.

Trusting AI Code Without Understanding

A key issue is **overtrust**. Some developers have the mindset that "if the AI generated it, it must be okay," especially if the code *runs*. However, multiple sources warn against this complacency. Contrast Security distinguishes "pure vibe coding" (no review) as *"catastrophic risk in production systems"*, versus "responsible AI-assisted development" (human in control)【41†L159-L169】. In practice, companies see a split: startups and individual creators often embrace pure vibe coding to move fast, whereas enterprises still require manual oversight of AI outputs.

Empirical evidence suggests that even developers can *misjudge* AI output. In the METR study【20†L93-L101】, developers *believed* AI was helping them (their forecasts were wrong), even though measured times were longer. Similarly, novices may think their AI-built app is secure just because "it works." But as Unit42 notes, those without a coding background *"lack the literacy to review or secure the code being generated"*, creating a dangerous gap【30†L23-L32】. This points to a human factors risk: people may accept AI-generated code without full understanding, especially if the tool appears to handle details. Surveys and forums indicate some developers trust Copilot and ChatGPT too much, sometimes pasting output with minimal scrutiny.

Can AI Tools Compensate for Missing Expertise?

One might ask if AI itself can fill the gaps. Some vendors embed automated testing, security scanning, or correctness checking into AI coding environments. For example, many AI IDEs include linters or SAST scans that flag obvious issues. In theory, one could prompt the AI to generate tests for its code or to check for known vulnerabilities. There is ongoing research into LLM-driven DevOps and automated testing platforms. However, the evidence so far is that these *assistive measures have limited impact*. The METR study implies that the AI tools used did not include any special automated QA that sped developers up – on the contrary, they slowed down. Unit42 argues that since AI often *bypasses* traditional "shift-left" security gates, the focus must be on putting security back in: e.g. requiring human code reviews, or building "helper agents" specifically to validate AI output【30†L77-L86】【41†L197-L206】.

In other words, while future AI agents might eventually integrate test generation or vulnerability detection, today's tools are not a panacea. Automated testing frameworks can catch some regressions, but only if developers know to write and maintain tests. Static and dynamic analysis tools still need human configuration and interpretation. No peer-reviewed study yet shows that AI coding agents by themselves reliably enforce best practices. The consensus from experts is that AI **augments** developer productivity but does **not replace** the need for knowledgeable engineers in security, architecture, and operational roles【41†L159-L169】【30†L45-L53】.

Prototype vs. Production

It's important to distinguish stages of development. AI excels at the *prototyping/MVP* phase: it can quickly mock up screens, basic database schemas, and simple features. This is why many founders use it to validate ideas or build demos. In fact, Google's definition of vibe coding explicitly notes its ideal for "rapid ideation or throwaway weekend projects" where "speed is the primary goal"【1†L31-L39】.

However, moving from prototype to production changes the rules. Production-grade software typically requires:

- **Architecture Review:** Ensuring the design can scale under real workloads, that it meets performance SLAs, and that it integrates with other systems correctly.
- **Robust Testing:** Automated tests (unit, integration, performance, security), continuous integration, and staging environments. A viable CI/CD pipeline.
- **Security Controls:** Penetration testing, code scanning, vulnerability patching, secrets management.
- **Reliability Engineering:** Logging, monitoring, observability, failover, and recovery processes.
- **Compliance Audits:** Verified adherence to relevant standards (PCI, HIPAA, GDPR, etc.), including documented evidence (audit logs, data retention policies).
- **Maintainability Practices:** Version control with human-readable commit history, code reviews, documentation, and on-call support.

Crucially, most of these layers **do not automatically appear from AI coding**. An AI prompt like *"build me a signup form"* won't create audit logs, encryption key rotation, or DDoS protections. It won't set up a SOC 2–compliant change management process or GDPR data-impact assessment. As SecurePrivacy warns, *"the failure mode isn't the tool — it's the workflow that treats 'it works' as the finish line, when compliance review is a different finish line entirely"*【33†L308-L317】.

For **enterprise systems** and especially for **safety-critical or regulated software** (medical devices, avionics, industrial controls, etc.), the bar is even higher. Such systems require formal verification, certification, and thorough human oversight. We know of no case where an LLM built something like that without expert developers. In fact, relying solely on AI for critical software would currently violate best practices and regulations.

Conclusion: Can Anyone Build Production Software with AI Today?

In a limited sense, yes – but with strong caveats. Generative AI does make it easier for people with minimal coding experience to cobble together **working prototypes or simple applications**. These can be useful for testing ideas or solving small internal problems. However, the evidence suggests that *truly production-grade systems* require at least some professional oversight. The largest demonstrated risks in vibe-coded software arise when AI outputs are treated as trustworthy without human review【41†L159-L169】【41†L174-L182】. Studies show large fractions of AI-generated code contain bugs and security flaws (often >30–60%)【41†L174-L182】【9†L72-L80】. Thus anyone deploying AI-generated code at scale should:

- **Mandatory Review and Testing:** Have experienced engineers audit AI-written code, write tests (unit/integration/security), and not skip QA. (Contrast Security argues code must be *"owned by the developer"* who reviews and takes responsibility【41†L159-L169】【41†L197-L206】.)
- **Architectural Oversight:** Ensure the system's design is sound. This likely means defining high-level requirements that the AI follows, not just ad hoc prompts.
- **Security and Compliance Controls:** Run static and dynamic analysis on AI-generated code, perform threat modeling, and implement regulatory safeguards. Use the SHIELD-like governance (separate duties, minimal privileges, human-in-loop code review, etc.) recommended by security experts【30†L115-L123】【41†L197-L206】.
- **Gradual Rollout and Monitoring:** Deploy new features in controlled stages, monitor for anomalies, and be ready to roll back if the AI-introduced change causes problems.

In essence, while **AI can generate the code**, it cannot (yet) generate *trust*. Real expertise is needed to vet, secure, and maintain that code over time. The most severe risks of vibe coding are demonstrably not **different in kind** from traditional coding (humans also write vulnerable code), but they are *widespread and invisible* because they happen quickly and often without awareness. As one summary puts it, the danger is that *"AI-assisted coding may produce vulnerabilities at such scale that many will never be identified or fixed"*【29†L37-L45】【41†L174-L182】.

Bottom line: A motivated novice *can* use AI to build a functioning app today, but to make it production-ready in any serious sense requires skills in architecture, software engineering, and security. The **greatest risks** come from unreviewed code, misconfigured data access, and missing compliance mechanisms – issues that AI alone will not prevent【41†L174-L182】【33†L308-L317】. Companies and developers must remain vigilant: AI is a powerful accelerator, but it does not replace the need for human expertise in critical domains.

Sources: Recent research and reports (2021–2026) highlight these points. Academic studies quantify the high bug rates in AI code【41†L174-L182】【9†L72-L80】. Security industry analyses document real incidents of AI-generated code breaches【24†L35-L44】【30†L60-L69】. Regulatory and privacy experts warn that compliance *"won't emerge from a prompt"*【33†L308-L317】. These combined lines of evidence strongly suggest that *anyone* can experiment with AI coding today, but building truly robust production software still requires professional oversight and expertise.
