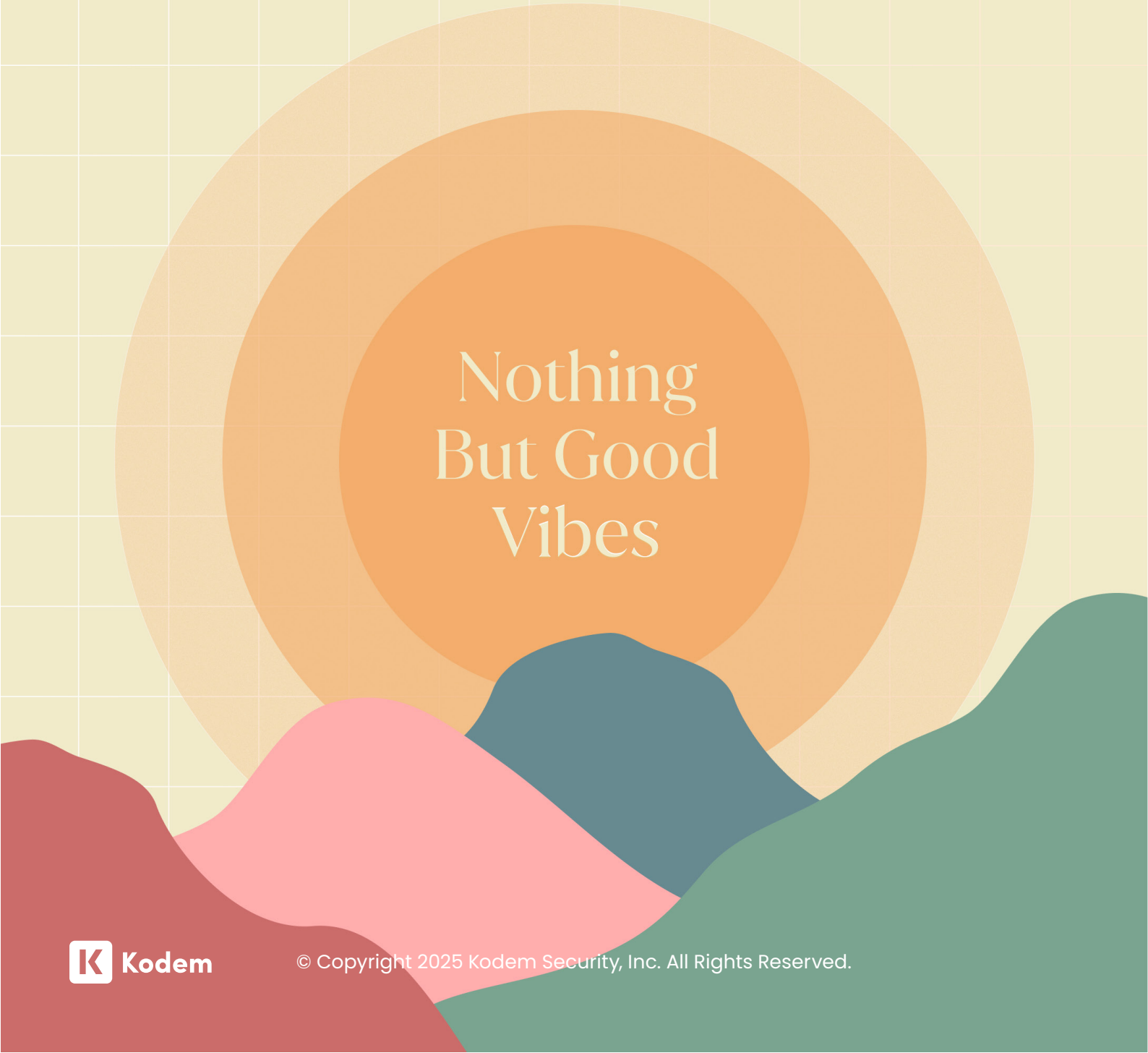
The Definitive Playbook to **Secure Vibe Coding**



Kodem

Introduction

AI-assisted development tools have rapidly become fixtures in modern “born in the cloud” organizations. From pair-programming assistants embedded in IDEs to conversational agents that generate code on demand, these tools promise accelerated software delivery and enhanced developer productivity.
However, they also introduce unique security challenges. Product security leaders, CTOs, and engineering VPs must address risks ranging from insecure code suggestions to data leakage.

This playbook provides a comprehensive roadmap for securing AI-assisted and citizen developer coding efforts. It catalogues common AI development tools and segments them, examines technical and governance risks per category (with real-world examples), discusses challenges with citizen developers, highlights relevant AI regulations (ISO/IEC 42001, NIST AI RMF), and concludes with actionable recommendations and a maturity model for safe AI-driven software development.

Landscape of AI-Assisted Development Tools

AI coding tools can be grouped by how developers interact with them. Below we identify major tools in each segment and their typical use cases:

IDE-Integrated AI Coding Assistants

These tools plug into code editors or IDEs to provide real-time code completions and suggestions as developers write code. Examples include **GitHub Copilot**, **Tabnine**, **Amazon CodeWhisperer**, and **Cursor**:

- *GitHub Copilot*: An AI "pair programmer" extension for VS Code, JetBrains, etc., that autocompletes code and entire functions based on context. Powered by OpenAI Codex, it's widely used to speed up writing boilerplate and tests (GitHub Copilot Isn't Worth the Risk) (GitHub Copilot Isn't Worth the Risk).

- *Tabnine*: An alternative AI assistant that also integrates with IDEs. Notably, Tabnine's model is trained only on permissively licensed open-source code to reduce IP risk (GitHub Copilot Isn't Worth the Risk).

- *Amazon CodeWhisperer*: AWS's AI coding companion integrated into Cloud9 and other IDEs. It provides suggestions and also offers built-in security scanning and reference tracking (e.g. flagging if a suggestion is similar to known code) to mitigate vulnerabilities and licensing issues (GitHub Copilot Isn't Worth the Risk).

- *Cursor*: A standalone AI-enabled code editor ("the AI Code Editor") that offers code predictions and can answer questions about your codebase (Cursor - The AI Code Editor). Cursor runs locally as an IDE with AI features, bridging the gap between a traditional editor and a chat assistant.

These IDE-integrated tools work in-line as you code, making them popular with professional developers for day-to-day development.

Conversational AI Code Assistants (Chatbots and Browser-Based)

Conversational assistants allow developers to obtain code through a dialogue interface, often in a web browser or chat application.
Key examples include **OpenAI ChatGPT** and **Anthropic Claude** (sometimes referred to as Anthropic Code Assistant):

- *ChatGPT*: A conversational AI by OpenAI that can answer questions, explain code, and generate code snippets or entire programs based on prompts. Developers use ChatGPT via a browser or API to get quick solutions or boilerplate code in natural language format. Its code-writing prowess has made it a go-to "copilot" outside the IDE for many engineers.

- *Anthropic Claude*: Claude is an AI assistant from Anthropic that can similarly engage in dialogue to produce or review code. Marketed with a focus on being a safer, more "constitutional" AI, Claude is used via chat interface or API for tasks like code generation and debugging.

These tools function like "AI pair programmers" you talk to. They are not tied to a specific code editor – developers paste prompts and code in a web UI or chat and receive AI-written code in return. This makes them accessible to a broad range of users (including non-developers), but it also means code and queries are transmitted to an external service over the internet.

AI-Driven Low-Code/No-Code Application Generators

A newer category of AI development tools aims to let **citizen developers** (users with little coding experience) build complete applications via high-level instructions. These "agentic" AI builders handle writing and assembling the code under the hood. Examples include **Bolt.new** and **Lovable.dev**:

- *Bolt.new*: An AI-powered app builder (by StackBlitz) where the user simply describes what they want (e.g. "a mobile-friendly task tracker") and the AI generates the necessary front-end and back-end code, deploying a working web application. Bolt emphasizes minimal human coding – acting as an autonomous

developer that scaffolds and updates the app through a chat interface (Using Agentic AI Editors. I tried StackBlitz's bolt, Lovable, and… | by R. Harvey | Bootcamp | Feb, 2025 | Medium).

- *Lovable.dev*: Branded as a "superhuman full-stack engineer," Lovable allows users to chat with an AI to build web applications. It produces more polished UI output and handles deployment, targeting those who want to create apps without writing code themselves (A Deep Dive into AI Coding Assistants: Lovable, Bolt.new, Cline, and Cursor/Windsurf) (A Deep Dive into AI Coding Assistants: Lovable, Bolt.new, Cline, and Cursor/Windsurf)..

These tools typically provide an all-in-one, browser-based development environment. The user converses with the AI about features and design, and the AI writes the code and manages infrastructure. They blur the line between no-code platforms and AI coding assistants, empowering less-technical staff to create software.

However, as discussed later, they often prioritize rapid prototyping over rigorous security. (Notably, other emerging tools in this space include Replit's Ghostwriter for AI-assisted coding in an online IDE, and Trae – an "adaptive AI IDE" by TikTok – but for brevity, we focus on Bolt and Lovable as exemplars (Using Agentic AI Editors. I tried StackBlitz's bolt, Lovable, and… | by R. Harvey | Bootcamp | Feb, 2025 | Medium).)

Security Analysis of AI Coding Tools by Category

For each segment of AI development tools, we examine common technical vulnerabilities introduced, governance and policy concerns, and known security incidents or examples.

IDE-Integrated AI Assistants (Copilot, Tabnine, Cursor, etc.)

Common Technical Vulnerabilities

AI code completions can inadvertently introduce security weaknesses into the codebase:

- **Insecure Coding Patterns**: Because these models learn from public code (which includes insecure code), they may suggest vulnerable constructs like SQL injection-prone queries, weak cryptography, or improper input validation (Four Security Risks Posed by AI Coding Assistants). An NYU study found that around **40% of code generated by GitHub Copilot contained bugs or security vulnerabilities** that could be exploited (CCS researchers find Github CoPilot generates vulnerable code 40% of the time - NYU Center for Cyber Security). This means developers cannot blindly trust AI suggestions – they may save time but still require thorough review.

- **Use of Insecure Dependencies**: AI suggestions often include importing libraries or using packages to speed up development. If the AI was trained on examples that use outdated or vulnerable versions, it might recommend those. This introduces known-vulnerable dependencies (supply chain risks) into your project (Four Security Risks Posed by AI Coding Assistants). The assistant lacks awareness of the latest patched versions or whether a library has critical flaws, so it might, for instance, suggest using a logging package version with a known RCE vulnerability.

- **Hardcoded Secrets and Credentials**: There is a risk of AI introducing or leaking secrets. In some cases, **AI has regurgitated API keys and passwords that appeared in its training data**. Researchers have shown Copilot can produce valid AWS keys and other

credentials that were embedded in public code it trained on (Yes, GitHub's Copilot can Leak (Real) Secrets) (Yes, GitHub's Copilot can Leak (Real) Secrets). This could lead to secrets exposure in your code if not caught. Conversely, a developer might also accept an AI suggestion that includes a placeholder secret and forget to remove it, leaving a credential in code.

- **Hallucinated or Incorrect Logic**: The AI sometimes generates code that looks plausible but is logically flawed or calls non-existent APIs. These "hallucinated" code blocks can create security issues — e.g. logic that fails open on errors or an authentication check that doesn't actually execute. Since the error may not be obvious, a developer with less experience might not realize the code isn't doing what they intended, potentially opening a security hole (such as bypassed validations or weak error handling).

In summary, while IDE assistants greatly speed up coding, they can introduce anything from subtle bugs to critical vulnerabilities in the application. A mantra from researchers: "Developers should remain vigilant" when using AI suggestions and pair AI assistants with security-aware tools or reviews (CCS researchers find Github CoPilot generates vulnerable code 40% of the time - NYU Center for Cyber Security).

Governance and Policy Concerns

The use of cloud-based coding assistants in an enterprise raises significant governance questions:

- **Data Leakage and Privacy**: IDE-integrated tools typically send your code context (which may include proprietary source code) to a third-party AI service for analysis. This poses obvious confidentiality risks (Four Security Risks Posed by AI Coding Assistants). Sensitive code could be intercepted in transit or stored on the provider's servers. In fact, concerns that developers might inadvertently leak internal source code to Copilot's cloud service led organizations like the U.S. House of Representatives to ban its use for official work (Data Security Fears: Congress Bans Staff Use of Microsoft's AI Copilot).

  The fear is that sending "House data" to non-approved cloud services could expose it (Data Security Fears: Congress Bans Staff Use of Microsoft's AI Copilot). Similarly, many companies (e.g. finance and tech firms) have restricted Copilot or require using a self-hosted alternative due to data privacy.

- **Intellectual Property (IP) and License Compliance:** AI models trained on open-source code might produce snippets that are identical or very similar to that training data. This raises the risk of license violations or IP contamination. GitHub Copilot has been the subject of a high-profile lawsuit alleging that it **suggests code that infringes open-source licenses by stripping required attribution** (GitHub Copilot Isn't Worth the Risk).

  For example, Copilot might output a block from a GPL-licensed project without mentioning the license, leaving the user unknowingly in breach of copyright. Companies must consider if AI-generated code could introduce unlicensed code into their products. (Notably, some vendors attempt to mitigate this: Tabnine uses only permissively licensed code to train (GitHub Copilot Isn't Worth the Risk), and Amazon CodeWhisperer can detect and flag code that closely matches an open source repository, providing the source for proper attribution.)

- **Auditability and Accountability**: With human developers, it's standard practice to have peer code reviews and commit histories that show who wrote what. With AI writing chunks of code, it becomes harder to audit how a piece of code came to be. If a vulnerability is discovered later, was it introduced by a person or suggested by an AI? This gray area complicates accountability. Organizations might need policies for documenting AI involvement (e.g. tagging AI-generated commits or storing prompts) to maintain an audit trail. Without such measures, there's a **transparency gap in the development process** – one that regulators or customers might one day scrutinize under standards for software supply chain transparency.

- **Model Transparency and Bias**: The AI itself is a black box – it won't explain why it suggested a particular code snippet. Lack of transparency in how the model works means potential biases or gaps in its training data are not visible. For instance, if the model was never trained on secure coding examples for certain frameworks, it will consistently suggest weaker code in those contexts. Governance-wise, companies may push vendors for more insight into training data or usage of their code (some have begun demanding vendors allow opt-out of using their code in model training

In practice, many organizations address these concerns by developing acceptable use policies for AI coding tools. For example, a company might allow Copilot in non-sensitive projects but forbid its use on proprietary crown-jewel code or client data. Others conduct internal reviews of AI-introduced code using SAST (Static Application

Security Testing) and composition analysis to catch license or security issues. The key is balancing productivity gains with control measures so that no sensitive data is exposed and any AI-generated code meets the same standards as human-written code.

Real-World Security Incidents and Examples

Several notable incidents illustrate the above risks:

- **Samsung Source Code Leak (2023)**: In a widely reported mishap, Samsung engineers inadvertently **leaked confidential source code to ChatGPT** while using it to debug errors (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable) (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable). They pasted sensitive code into the AI, which then retained that data. The incident led Samsung to ban use of external AI services for coding. It underscored how easily proprietary logic can leave the organization if developers aren't careful – the AI's convenience lured them into violating data-handling policies.

- **Apple and Others Ban Copilot/ChatGPT**: Apple Inc. in 2023 restricted employees from using ChatGPT and even **warned against GitHub Copilot** due to fears that confidential information entered into these tools could be collected and leaked (Apple restricts employees from using ChatGPT over fear of data leaks | The Verge). Apple was not alone; JPMorgan, Verizon, Amazon and other enterprises similarly banned or curtailed generative AI usage internally (Apple restricts employees from using ChatGPT over fear of data leaks | The Verge). This trend in the industry shows the high level of concern about data governance when using such tools.

- **US House of Representatives Copilot Ban (2024)**: As mentioned, the House's Chief Administrative Officer banned Microsoft's GitHub Copilot for congressional staff, labeling it a "**risk… due to the threat of leaking House data**" (Data Security Fears: Congress Bans Staff Use of Microsoft's AI Copilot). This is a policy action (not a breach) but is an important real-world example of institutions reacting to the perceived security and compliance risks of AI dev assistants.

- **Copilot Lawsuit (2022)**: A class-action lawsuit was filed against GitHub, Microsoft, and OpenAI on behalf of open-source programmers, claiming Copilot's operation violated open-source licenses. While pending legal resolution, this case has put a spotlight on

the IP issues; it's a reminder that enterprises using AI coding tools could face liability if they unknowingly incorporate copyright-protected code (GitHub Copilot Isn't Worth the Risk). Even absent a lawsuit, the reputational damage of shipping a product later found to contain plagiarized code could be serious.

- **Security Research Findings**: Beyond incidents, academic and industry researchers have repeatedly shown the **security shortcomings of AI-generated code**. In addition to the 40% vulnerability rate study (CCS researchers find Github CoPilot generates vulnerable code 40% of the time - NYU Center for Cyber Security), another study found that developers using AI assistance **produced more security bugs and leaked secrets more often** than those who didn't, possibly due to over-reliance on the AI and less careful review (Yes, GitHub's Copilot can Leak (Real) Secrets). These findings serve as cautionary tales that adoption of AI assistants must go hand-in-hand with developer training and robust review processes.

Conversational AI Assistants (ChatGPT, Claude, etc.)

Common Technical Vulnerabilities

While conversational tools aren't integrated into an IDE, they can still introduce many of the same code risks, plus some unique issues:

- **Insecure or Incorrect Code Suggestions:** Chat-based assistants will happily provide code for almost any request, but they do not guarantee security. They might generate a quick solution that handles the functional requirement but overlooks edge cases or secure defaults. For example, a user asking "How do I connect to a database in Node.js?" might get a code snippet that indeed connects, but it could use a deprecated method or neglect proper error handling and input sanitization – laying the groundwork for SQL injection or crashes. The **AI has no inherent understanding of your app's security context**, so it may omit crucial steps (like authentication or encryption) unless explicitly asked. This "happy-path" focus can lead to code that passes basic tests but fails under malicious conditions (Four Security Risks Posed by AI Coding Assistants).

- **Hallucinations and Unsupported APIs:** ChatGPT and similar models sometimes **invent functions or APIs that don't exist**, or they piecemeal together code from

different contexts incorrectly. A citizen developer with low code experience might not recognize that a suggested API call is fictitious or deprecated and attempt to use it, leading to broken functionality or insecure fallbacks. Even experienced devs can be misled – there have been cases of ChatGPT confidently outputting code with subtle errors (e.g., hashing passwords with a weak algorithm while claiming it's secure). Such hallucinated logic can introduce vulnerabilities that are not immediately obvious.

- **Code Similarity to Public Examples**: Since these models were trained on a broad swath of internet text, including open-source code, the code they generate might closely mirror existing code (with all its flaws). If a vulnerability existed in a popular code snippet on Stack Overflow or GitHub, the AI could reproduce that same vulnerable pattern when prompted similarly. In one instance, ChatGPT was observed to generate a buffer overflow vulnerability because it mirrored an unsafe C code pattern from its training data. The user must have security knowledge to detect and correct these.

- **Lack of State and Context Beyond Prompt**: Unlike IDE tools that see your whole file/project context (in some cases), a standalone chat AI only knows what you tell it in the prompt (and its training). If you forget to mention a security requirement, it won't recall it from earlier code. This can lead to inconsistencies – for example, you secure one function by prompting the AI to add input validation, but later ask it to generate another component and it doesn't include similar validation because the prompt didn't specify it. The conversational AI won't inherently enforce a security baseline across your project. It's on the user to integrate and reconcile the AI's outputs securely.

In essence, conversational code bots are powerful but not infallible. They often require iterative prompting to refine answers, and even then the onus is on the developer to infuse security, since the AI might not do so by default. Testing and review of AI-written code is crucial.

Governance and Policy Concerns

The use of cloud-based coding assistants in an enterprise raises significant governance questions:

- **Confidential Data Exposure**: When developers (or other staff) use ChatGPT or Claude, they might input proprietary system information, code, or even customer data to

get help. As with IDE tools, this **risks leaking sensitive info to the AI provider**. In fact, ChatGPT's terms initially allowed OpenAI to retain and use input data for model training. Real incidents proved this is not a hypothetical risk – e.g., the Samsung case where trade secrets were input to ChatGPT (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable). Many companies responded by banning or limiting usage of such services until assurances like "do not train on my data" settings or enterprise privacy controls were available. OpenAI now offers a business tier and an option to disable chat history (which stops training on that data), but even then data is retained for 30 days for abuse monitoring (Apple restricts employees from using ChatGPT over fear of data leaks | The Verge). **Governance policies must clearly dictate what can and cannot be shared with external AI services**. For example, source code or architecture details might be disallowed in public ChatGPT, and an internal approved alternative used instead.

- **Output Filtering and Bias**: These AI models have some content filters (to avoid explicit or dangerous outputs), but they might also inadvertently filter or alter technical content. There were cases of ChatGPT refusing to output certain code (thinking it was disallowed content) unless prompts were rephrased. Conversely, the AI might insert bias from its training into code or comments (e.g., using non-inclusive terminology or following outdated practices). From a policy perspective, organizations need to be aware that the **AI is an opaque third-party tool with its own constraints and unknown training data**. Relying on it for official code could introduce hidden biases or inconsistent coding styles that conflict with the organization's standards.

- **Audit and Reproducibility**: When code is generated via a conversation, the "source" of that code is essentially the conversation logs. Keeping records of those interactions is important for auditing what information was provided to the AI and why the code turned out as it did. Some compliance frameworks might consider an AI a tool that needs qualification – for example, in regulated industries (finance, healthcare), if AI assists in producing software, the process may need documentation for auditors. Ensuring **auditability** might involve using enterprise solutions where chat logs are saved and attributable. It's also a reason some companies opt for self-hosted large language models for coding: they retain control of the entire input/output log and data residency.

- **Third-Party Dependencies and Terms**: ChatGPT and similar require agreeing to terms of service that might conflict with company policy (e.g., OpenAI's terms disclaim

liability for code correctness or security). Organizations should review these terms and possibly negotiate enterprise contracts. Additionally, the AI might suggest usage of third-party APIs or services in the code (e.g., "just use this cloud API for auth"). Without governance, a citizen developer might unknowingly incorporate a cloud service that hasn't been vetted by security or procurement. Policies should guide users to avoid directly following AI suggestions that introduce new external dependencies without review.

In short, governing the use of conversational AI tools means treating them with the same caution as any outsourced service or contractor: limit exposure of sensitive data, vet the outputs and dependencies they introduce, and ensure compliance with data protection requirements. Many organizations have begun crafting **AI Usage Policies** that cover both IDE plugins and chatbots, often emphasizing that **no confidential or personal data should be input into public AI tools** and that all AI-generated code must go through normal security review pipelines.

Real-World Security Incidents and Examples

A few key examples illustrate the need for careful use of conversational AI in coding:

- **Samsung ChatGPT Incident**: As noted earlier, this is a textbook case. In 2023, Samsung engineers used ChatGPT to troubleshoot and summarize code, inadvertently uploading proprietary source code and meeting notes. The information entered **became part of OpenAI's model training data and was effectively "out in the wild"** (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable) (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable). Samsung quickly restricted AI usage after this. The incident highlights how a simple copy-paste into a chat window can escalate into a data governance failure. It also showed that ChatGPT "doesn't keep secrets" – **anything you input can be seen by the provider or potentially leaked in future AI responses** (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable).

- **ChatGPT Hallucination Causing Security Issue**: A Reddit user in late 2023 described how ChatGPT gave them a snippet to sanitize user input but the code had a logic error, failing to actually enforce the sanitation in all cases. This bug went unnoticed until a security test caught it. While anecdotal, it emphasizes that **AI mistakes are real**

and can slip past developers, especially those with less expertise. In this case, the hallucinated logic created a latent vulnerability. Such stories have driven the point that AI outputs must be treated as untrusted drafts, not production-ready code.

- **Stack Overflow Temporary Ban**: In late 2022, the Q&A site Stack Overflow temporarily banned answers generated by ChatGPT because they were often incorrect. While not a direct security incident, this action signaled how even convincingly written answers (or code) from an AI can be dangerously misleading. A developer could easily copy an accepted-looking answer from ChatGPT that has a security flaw. The community's reaction reinforces that without verification, AI-produced code can reduce software quality.

- **Data Exposure via Prompt Leaks**: Another angle – if someone deploys a conversational code assistant integrated with their systems (for example, hooking ChatGPT into an internal tool), there's risk that **prompt data could be logged or leaked**. One incident involved an early ChatGPT plugin accidentally exposing parts of other users' conversation histories due to a caching bug. If those conversations had contained code or secrets, it would be a data breach. This reminds us that even the AI service itself can have vulnerabilities. Relying on them means inheriting their security issues too. (OpenAI confirmed a bug in March 2023 where some users saw titles of others' chats, which was quickly fixed – but it did little to calm companies worried about where their data might end up.)

Overall, the lesson is that conversational AI agents must be used with clear-eyed caution: treat any code they produce as if a junior developer wrote it (needing review), and never reveal more to them than you would to any external consultant or cloud API.

AI-Powered Low-Code Tools (Bolt, Lovable, etc.)

Common Technical Vulnerabilities

Low-code AI application generators present a different set of vulnerabilities, often stemming from the abstraction they provide and the inexperience of the users (citizen developers) they empower:

- **Lack of Secure Defaults**: These platforms aim to produce a working app quickly, but **security features are not always baked in by default**. For instance, one real example was a CRM web app generated by a no-code AI tool that **had no authentication at all**

– any user could access all data (The Magic (and Pitfalls) of Bolt, Lovable and Replit). The AI delivered a functional app as requested, but did not implement basic access control or data partitioning because the user didn't explicitly ask for it. Essential security steps (auth, input validation, encryption in transit, etc.) may be omitted unless the prompt explicitly includes them, which a non-technical user might not know to request. This can result in apps that "work" but are extremely insecure by design.

- **Incomplete or Weak Validation and Error Handling**: Even when the AI adds security measures, they might be rudimentary. For example, it might include a simple client-side form validation but no server-side checks, or use a hardcoded admin password for convenience. Low-code outputs often neglect thorough input validation, proper authentication flows, and robust error handling (The Magic (and Pitfalls) of Bolt, Lovable and Replit). These omissions lead to vulnerabilities like injection attacks, broken access controls, and exposure of debug information. Since the citizen developer might only test the "happy path," such flaws can go live unnoticed.

- **Over-privileged Integrations**: AI-built apps often integrate with databases, external APIs, or cloud services to achieve functionality (since the user just describes high-level behavior, the AI picks whatever components can fulfill it). This can result in services running with overly broad privileges or API keys embedded in the generated code. For instance, the AI might use an API key with full read/write access to a backend database for simplicity, even if the app only needs read-only access. It could also fail to restrict cross-origin requests or not enforce least privilege in cloud roles. These misconfigurations are equivalent to leaving the keys under the mat – they make any compromise far more damaging than necessary.

- **One-Size-Fits-All Security**: The AI may apply a generic security measure that is inappropriate for the context. For example, it might enable a basic CORS policy or a generic content security policy that doesn't actually cover the app's needs, giving a false sense of security. Or it might use outdated hashing for passwords because that pattern was present in an example. The nuance of security design (tailoring controls to the specific app threats) is lost when an AI is generically assembling an app. That can introduce subtle vulnerabilities – e.g., using JWTs without proper signature verification because the concept of token-based auth was added, but not the complete implementation.

- **Supply Chain and Dependency Risks**: Just like other AI coding tools, these platforms

pull in packages to build functionality. A user might request a certain feature ("add login via Google") and the AI includes an open-source library to handle it. The citizen developer is unlikely to vet that library. If the library has a known vulnerability or even a malicious dependency, the generated app now has a supply chain risk. With the speed of AI development, there's little pause to consider the provenance or security of dependencies being installed. Attackers might target popular AI code generation by poisoning libraries they know are commonly suggested by AI for certain tasks.

In summary, **AI-generated apps often suffer from the same issues as any rapid prototype**: they prioritize achieving the requested functionality, with security as an afterthought. The difference is a less experienced creator might not recognize these gaps. Thus, the code quality and security can be significantly lower than professionally developed apps, even though the app may appear feature-complete.

Governance and Policy Concerns

When empowering citizen developers with AI tools, organizations need to institute governance measures to prevent a Wild West of unsecure apps and compliance headaches:

- **Shadow IT and Unvetted Apps**: Low-code AI tools make it so easy to create new applications that business teams might launch their own tools without any IT oversight (The 5 Most Common Low-Code Security Concerns + How To Mitigate). This shadow IT can bypass security reviews, QA, and compliance checks. An employee might deploy an AI-built app handling customer data without informing the security team, inadvertently violating data protection laws or internal policies. Governance must include an approval or monitoring process for apps spun up by non-development teams. Maintaining an inventory of such apps and bringing them under central IT management is crucial to avoid a sprawl of possibly insecure, unmaintained applications (The 5 Most Common Low-Code Security Concerns + How To Mitigate) (The 5 Most Common Low-Code Security Concerns + How To Mitigate).

- **Data Leakage and Compliance**: Citizen devs may not be aware of regulations like GDPR, HIPAA, or industry-specific rules. They could easily build an app that mishandles personal data (e.g., failing to secure user PII or retain logs longer than allowed). Also, using an AI service to build the app means any data they feed into it (even sample data) could be exposed. If, say, a marketing employee uses Lovable to

build an app and includes real customer data in a prompt ("here's a CSV of clients, build a dashboard"), that could violate privacy laws or contractual data handling agreements. **Policies and training are needed to educate citizen developers on what data can be used and how to anonymize or protect it** when leveraging AI.

- **IP and Ownership of Code**: When an AI generates an application, questions arise: Who owns the code? Are there license restrictions on using it commercially? The citizen developer might unknowingly incorporate terms that conflict with the company's IP policies. For instance, if the AI copied a chunk of code from an open-source project, the resulting app might technically need to provide attribution or abide by that open-source license. Organizations should clarify that any AI-generated code is subject to the same IP review as third-party software. Some AI app builders include clauses in their terms about code ownership – these need legal review to ensure the company isn't exposed.

- **Operational Sustainability**: A governance concern slightly adjacent to security is what happens after the citizen-developed app is built. Who maintains it? Does the original user know how to apply patches or respond to incidents? Often, these apps get created and forgotten, without proper maintenance lifecycles. From a security standpoint, **unmaintained apps become soft targets** – if a vulnerability is discovered later (either in the generated code or a library it uses), it may go unfixed. Ensuring there is an ownership and maintenance plan for AI-generated applications (possibly transferring them to professional IT teams once they prove useful) is an important governance aspect. This ties into a maturity model – at higher maturity, no app, however created, is outside the purview of the security team's monitoring.

In short, democratizing app development with AI is powerful but must be accompanied by guardrails. Organizations should establish clear policies for citizen developers, including requiring registration of any app created, mandatory security checks (perhaps using automated scanners or checklists for common issues), and providing secure templates or vetted components to use. Some companies even set up "fusion teams" where citizen devs work alongside IT-appointed experts who guide them on security and compliance, ensuring the final product meets the organization's standards.

Real-World Security Incidents and Examples

We are in early days of these AI app builders, so documented security incidents are

few (and companies may not publicly disclose flaws in apps built this way). However, we do have illustrative examples and analogous low-code incidents that shed light on potential issues:

- **No-Auth CRM Example**: Security expert Matt Pollins noted a case where an AI-generated CRM application launched without any user authentication, meaning all data was publicly accessible (The Magic (and Pitfalls) of Bolt, Lovable and Replit). The creators of the app were proudly showcasing its features, unaware of the glaring security hole. This example (shared in a LinkedIn article) highlights the blindness that can occur when non-engineers rely on AI: the app worked and met the feature requirements, so they assumed it was production-ready, not realizing they had essentially created a data breach waiting to happen. Once pointed out, this was a learning moment – but if it had been a malicious actor finding it instead of a benevolent critic, the outcome could have been much worse.

- **Neglect of Basic Security in No-Code Apps**: In general, the community has observed that **many no-code/low-code apps "launch with almost zero security**," often lacking input validation, proper API protections, and having hardcoded credentials or keys exposed in the code (The Magic (and Pitfalls) of Bolt, Lovable and Replit). One can view this as a trend-based incident: numerous small applications with security shortcomings represent an aggregate risk. Attackers are beginning to realize that these citizen-developed apps are easy targets – they may not show up in the main corporate security scans and likely have exploitable weaknesses. There is at least one reported case of an internal tool built by a non-IT team which caused a company a compliance violation because it logged sensitive data without encryption. Such stories push companies to restrict deployment of these apps until security teams vet them.

- **OWASP Top 10 for Low-Code Issues**: The Open Web Application Security Project (OWASP) has even started a "Low-Code/No-Code Top 10" project to enumerate common vulnerabilities in such platforms (Security for Citizen Developers: Low-Code/No-Code Cybersecurity ...). While not a single incident, it was motivated by rising reports of security issues in citizen-developed applications. For example, improper

few (and companies may not publicly disclose flaws in apps built this way). However, we do have illustrative examples and analogous low-code incidents that shed light on potential issues:

- **No-Auth CRM Example**: Security expert Matt Pollins noted a case where an AI-generated CRM application launched without any user authentication, meaning all data was publicly accessible (The Magic (and Pitfalls) of Bolt, Lovable and Replit). The creators of the app were proudly showcasing its features, unaware of the glaring security hole. This example (shared in a LinkedIn article) highlights the blindness that can occur when non-engineers rely on AI: the app worked and met the feature requirements, so they assumed it was production-ready, not realizing they had essentially created a data breach waiting to happen. Once pointed out, this was a learning moment – but if it had been a malicious actor finding it instead of a benevolent critic, the outcome could have been much worse.

- **Neglect of Basic Security in No-Code Apps**: In general, the community has observed that **many no-code/low-code apps "launch with almost zero security**," often lacking input validation, proper API protections, and having hardcoded credentials or keys exposed in the code (The Magic (and Pitfalls) of Bolt, Lovable and Replit). One can view this as a trend-based incident: numerous small applications with security shortcomings represent an aggregate risk. Attackers are beginning to realize that these citizen-developed apps are easy targets – they may not show up in the main corporate security scans and likely have exploitable weaknesses. There is at least one reported case of an internal tool built by a non-IT team which caused a company a compliance violation because it logged sensitive data without encryption. Such stories push companies to restrict deployment of these apps until security teams vet them.

- **OWASP Top 10 for Low-Code Issues**: The Open Web Application Security Project (OWASP) has even started a "Low-Code/No-Code Top 10" project to enumerate common vulnerabilities in such platforms (Security for Citizen Developers: Low-Code/No-Code Cybersecurity...). While not a single incident, it was motivated by rising reports of security issues in citizen-developed applications. For example, improper authorization (broken access controls) and account impersonation make the list of top concerns (The hidden paradox of low-code development - The AI Journal), reflecting real weaknesses seen in the field. This kind of guidance is helping organizations preempt incidents by checking AI-generated apps against known problem patterns.

- **Positive Example – Assisted Fixes**: On a more positive note, some AI coding platforms are starting to integrate security features. For instance, AWS CodeWhisperer (though not exactly low-code) will highlight potential vulnerabilities in the code it suggests. If similar capabilities make their way into tools like Bolt or Lovable, we might see the AI itself warning, "I created this admin account with no password for demo purposes, you should secure this." However, until such maturity is reached, the burden is on the human overseers.

In summary, the absence of widely publicized "breaches caused by AI-built app" incidents should not breed complacency. It's likely only a matter of time. The prudent approach is to assume any app created via these new AI means could have serious issues and to treat them with the same rigor as any software being introduced into the environment.

Security Concerns for Citizen Developers

Citizen developers – employees outside of traditional software engineering roles who create applications or scripts, often with low-code or AI tools – bring a different challenge to product security. They typically have deep domain knowledge but limited software security training. When using AI-assisted development, several concerns arise:

- **Knowledge Gaps and Overreliance on AI**: Citizen devs may not know secure coding practices or how to properly validate AI suggestions. They might assume the AI "knows best." As a result, they could readily implement whatever the assistant provides, even if it contains vulnerabilities. Because they lack experience, they're less likely to catch mistakes. As one report notes, "Citizen developers often lack formal training in coding best practices, which can lead to security risks if their apps bypass code reviews." (The 5 Most Common Low-Code Security Concerns + How To Mitigate). In other words, the usual safety net (experienced devs and security teams reviewing code) might be absent. This calls for additional training or safety mechanisms for those users.

- **Lack of Security Mindset**: Professional developers are (hopefully) conditioned to think about threats and abuse cases. Citizen developers, focused on solving a business problem, might not consider how their app could be misused. They might not add authentication, logging, or input checks because their mindset is features, not threats. The AI won't insist on these unless asked. The outcome is applications that work in the intended scenario but **fail ungracefully (or catastrophically) under malicious conditions**. Without a security mindset, a user might also unknowingly expose data (e.g., building a data dashboard that's cloud-accessible without realizing it's public). Cultivating at least a basic security awareness in these users is critical – for example, encouraging them to follow a checklist (like OWASP's low-code security checklist) and to involve IT for anything beyond trivial apps.

- **Unaware of Regulatory Requirements**: A citizen developer might not realize that handling, say, customer emails in a workflow app means they must follow GDPR guidelines, or that integrating with a financial system triggers SOX compliance considerations. They may inadvertently break compliance rules – for instance, by not building an audit log into a system that processes financial data or not including an

opt-out in a marketing automation tool they created. AI assistants won't inherently enforce regulatory compliance (an AI might help if asked, but only if the user knows to ask). Thus, there's a risk of creating systems that violate laws or internal policies. Security leaders should extend compliance training and support to citizen devs or restrict what data they can play with in these tools.

- **Misunderstanding AI Limitations**: Citizen developers might also misinterpret what the AI is telling them. If the AI says "Code is secure" (perhaps in response to "Did you make this secure?"), they might take it at face value. They might not realize that the AI cannot guarantee security. This over-trust can be dangerous. Additionally, they might not know that AI tools can produce confident-sounding, authoritative answers that are wrong. This is a recipe for a false sense of security.

- **Deployment and Environment Security**: Often the citizen developer will also deploy the application (since these platforms make deployment easy, e.g., one-click to cloud). They may not understand environment security – e.g., leaving a database publicly accessible, or not setting up HTTPS, or using default credentials on cloud services. One slip in those areas and the entire app and its data could be compromised. A professional dev or IT team would normally handle secure deployment, but here the user is doing it via automation. Without guidelines, they might choose convenience (open access) over security (restricting access).

Mitigations for citizen developer risks include providing templated secure components (so if they need a login system, they use a company-provided module that is known secure), gating deployment of apps until a security review, or even limiting the scope of what citizen devs can do (for example, only allow them to build internal tools that don't handle sensitive data). Many organizations are implementing **governance frameworks for citizen development**, where each project is registered, and IT/security provides oversight and resources. By pairing the enthusiasm and domain expertise of citizen developers with guidance from security professionals, companies can reap the benefits of these AI tools without as much risk. Ultimately, fostering a collaborative culture where citizen devs feel supported in asking for security help will yield more secure outcomes than treating these apps as purely personal side projects.

AI Regulations and Compliance Standards for Secure Development

As AI-assisted software development becomes mainstream, standards bodies and governments have begun responding with frameworks to ensure AI is used safely, ethically, and securely. Two key frameworks relevant to organizations adopting AI coding tools are **ISO/IEC 42001** and the **NIST AI Risk Management Framework (AI RMF)**. Aligning internal practices with these can help in building a comprehensive governance program for AI in software development.

ISO/IEC 42001:2023 – AI Management System Standard

ISO/IEC 42001 is the world's first standard for AI management systems, published in December 2023. Think of it as an analogue to ISO 27001 (information security management) but focused on AI. **ISO 42001 provides guidance for organizations to design, develop, and deploy AI systems responsibly**, covering facets like transparency, accountability, bias mitigation, safety, and privacy (Understanding ISO 42001).

Key elements of ISO/IEC 42001 include:

- **Management System Approach**: It follows the Plan-Do-Check-Act cycle typical of ISO standards. Leadership and governance are central – top management must demonstrate commitment to an "AI Management System" (AIMS) and integrate AI risk considerations into corporate policies (Understanding ISO 42001). For a product security leader, this means having executive support to enforce rules around AI development tool usage.

- **Risk Assessment and Planning**: Organizations are expected to identify and assess risks and opportunities associated with AI use (Understanding ISO 42001). In our context, this could mean assessing the risk of using an AI coding assistant on a critical project, or the IP risk of generative code, and planning controls to address them.

Annex C of ISO 42001 even lists AI-specific risk sources, which likely include security and privacy risks from training data or model errors (Understanding ISO 42001).

- **Transparency and Traceability**: The standard emphasizes transparency – being able to explain how AI is used and making its outputs traceable. For AI-assisted coding, an organization in compliance might maintain documentation of where AI was used in code production and ensure that stakeholders (or auditors) can get explanations for critical decisions. It encourages documenting the data and methods used in AI, which aligns with maintaining audit logs of AI prompts and outputs.

- **Bias Identification and Mitigation**: While primarily about ethical AI (avoiding discriminatory outcomes), in coding context this could also relate to ensuring the AI doesn't introduce bias or unfair practices in algorithms it writes. More concretely, it means verifying the AI suggestions don't systematically neglect security for certain components due to training bias (e.g., always handling admin users differently). It's about checking the AI's outputs for unintended consequences.

- **Security and Safety**: ISO 42001 covers the safety and privacy of AI systems as part of its controls (Understanding ISO 42001). This means organizations should secure the AI tools themselves (protecting model integrity, access control to AI systems) and ensure the AI's use does not undermine overall security. If following ISO 42001, a company using Copilot would need to address it in their AI management system – for instance, having controls for "use of external AI services must be approved and monitored" as part of their operational procedures.

Adopting ISO/IEC 42001 is voluntary, but for cloud-native organizations aiming to be ahead of the curve, it provides a structured way to govern AI usage. It can serve as a bridge between high-level AI ethics/safety principles and practical secure software development. By following it, an organization demonstrates due diligence in managing AI risks. In the context of AI-assisted coding, it ensures you have thought about and documented things like: what data is your AI tool trained on, do you have consent to use that data, how do you handle errors or incidents involving the AI, and how do you continuously improve your AI-related processes. Achieving certification (or self-attestation) to ISO 42001 could become a market signal that your software development (even with AI in the mix) meets international best practices for AI governance.

NIST AI Risk Management Framework (AI RMF 1.0)

The U.S. National Institute of Standards and Technology (NIST) released its AI Risk Management Framework 1.0 in January 2023 (AI Risk Management Framework | NIST). The NIST AI RMF is a voluntary framework designed to help organizations **map, measure, and manage the risks of AI systems** in a structured way (NIST AI Risk Management Framework Explained) (NIST AI Risk Management Framework Explained). It is intended to be industry-agnostic and flexible, much like the NIST Cybersecurity Framework. For AI-assisted development, it offers a comprehensive approach to identify and mitigate risks such as those we've discussed (security vulnerabilities, data leakage, etc.).

The AI RMF is organized around **four core functions** (NIST AI Risk Management Framework Explained): **Govern, Map, Measure, and Manage**. Here's how each applies to our context:

- **Govern**: Establishing a culture of risk management and accountability for AI. In practice, this means creating policies and organizational structures for oversight of AI usage (NIST AI Risk Management Framework Explained). For example, a company might set up an AI Governance Committee that includes the product security lead, which oversees tools like Copilot/ChatGPT usage policies. The Govern function would ensure roles are defined (who is responsible for AI tool vetting, who handles incidents involving AI) and that there's top-down support for secure AI practices. This is analogous to having a security governance structure, but specifically for AI-related matters.

- **Map**: Contextualizing and identifying AI risks. Under this function, an organization would thoroughly understand how AI is being used in development and what the potential impacts are (NIST AI Risk Management Framework Explained). They might inventory all AI coding tools in use, the types of projects they're used on, and map out where things could go wrong (e.g., "AI suggests insecure code" or "developer might expose data to AI service"). It also means considering the broader context – for instance, if you're a healthcare software company, the risk of AI introducing a vulnerability has safety implications for patients. Mapping ensures you identify and prioritize the risks (technical and governance) relevant to your scenario (NIST AI Risk Management Framework Explained).

- **Measure**: Assessing AI system risks and impacts through quantitative or qualitative

methods (NIST AI Risk Management Framework Explained). For AI-assisted coding, this could involve tracking metrics like the percentage of AI-generated code that passes security tests, or the frequency of policy violations (like sensitive data found in prompts). It could also mean stress-testing the AI: e.g., having security team members intentionally prompt the AI in problematic ways to see if it will produce insecure code, thus measuring its propensity to introduce certain classes of flaws. The measure function pushes organizations to gather evidence – maybe run an internal audit of code written with AI vs without and compare bug density. The insights from measurement feed into how you control the AI's use.

- **Manage**: Taking action to mitigate identified risks and regularly improving your risk posture. In our context, this is where you implement controls such as those we've discussed: code review processes for AI contributions, data handling rules, use of checkers like static analysis on AI-written code, etc., and then adjust as needed (NIST AI Risk Management Framework Explained). The Manage function is an ongoing loop – for instance, if a new type of AI-related vulnerability surfaces (say prompt injection attacks on an AI agent integrated into an IDE), you update your safeguards to cover it. NIST AI RMF emphasizes that risk management is continuous, so as your team's use of AI evolves (maybe you start allowing more AI autonomy in coding), your risk mitigation strategies should evolve too.

Following the NIST AI RMF can also help with compliance to any future regulations, as it's built to align with trustworthy AI principles that many laws (like the forthcoming EU AI Act) are likely to require. For a practical example, NIST's framework has influenced updates to the NIST Secure Software Development Framework (SSDF) to incorporate AI considerations (NIST Issues AI Risk-Management Guidance - Greenberg Traurig, LLP). This means if you already adhere to secure SDLC practices, the AI RMF gives additional guidelines specific to the AI components in that lifecycle. It urges things like evaluating training data security, validating model outputs (e.g., testing that Copilot's outputs don't break security tests), and building resilience against AI-specific attacks (like prompt injection (Indirect Prompt Injection: Generative AI's Greatest Security Flaw)).

By embracing frameworks like ISO 42001 and NIST AI RMF, organizations not only address current risks but also future-proof their processes. These standards offer a common language and set of objectives for different stakeholders (security, legal, DevOps, executives) to ensure that the introduction of AI into software development does not erode the security and trustworthiness of products. In many ways, complying with them formalizes a lot of the best practices we've been discussing in this playbook.

Recommendations and Maturity Model for Secure AI-Assisted Development

Implementing security for AI-assisted coding is not a one-time task but an evolving program. Below are specific actionable recommendations, followed by a phased maturity model to guide organizations from basic safeguards to advanced governance.

Actionable Security Recommendations

1. Establish Clear AI Usage Policies:
Develop and enforce an internal policy for AI coding tool usage. This policy should cover what data can/cannot be input into AI assistants (e.g., "no sensitive customer data or proprietary code in public AI tools" is a common rule), which tools are approved for use, and for what purposes. Include guidelines on handling AI-generated code (for instance, requiring code review or security testing before it's merged). Make sure all developers and citizen developers are aware of this policy and the reasoning behind it. Consider requiring users to tag or identify AI-assisted code commits, to aid later auditing.

2. Security Training and Awareness:
Train developers – both professional and citizen – on the secure use of AI tools. This training should teach them to verify AI outputs, to recognize common insecure code patterns, and to use the tools' features safely (such as Copilot's settings to not retain code, or ChatGPT's opt-out of data retention). Also educate them on the risks of data leakage: emphasize incidents like Samsung's to make it concrete why the policy exists. For citizen developers, provide simplified guidance (maybe a one-page checklist) covering basics like "Did you add authentication?", "Are you exposing any secrets?" and "Don't assume the AI did it for you." Instill a healthy skepticism: treat AI as an assistant that needs oversight, not an oracle.

3. Integrate AI Outputs into Existing Security Processes:
Update your SDLC to account for AI-generated code. For example, include static code analysis (SAST) and software composition analysis (SCA) scans on all code, regardless

of author. If AI is introducing insecure code or vulnerable dependencies, these tools can catch many issues. You may also incorporate AI-specific linters or scanners – some tools can detect if a snippet matches known vulnerable code (e.g., using similarity to known CVEs). Have peer code reviews pay special attention to AI-produced sections; perhaps require an extra sign-off from a senior developer for code that was largely AI-written. Treat the AI like a junior developer whose code needs extra scrutiny. Over time, collect data: are certain types of bugs recurring from AI suggestions? Feed that back into training (human training, that is) or tool configuration to prevent them.

4. Use Enterprise-Grade or Self-Hosted Solutions: If possible, prefer enterprise versions of AI tools that offer better security controls. GitHub Copilot for Business, for instance, offers an option not to retain or use your code for training. Some companies choose to self-host AI models (or use on-premises solutions) to keep all data in-house – there are open-source code models or Azure/OpenAI offerings where your data doesn't leave your tenant. While this can be costly, for organizations working with highly sensitive code, it might be worth it. Even for less sensitive scenarios, check if the tool provides features like audit logging, admin control of settings, or region-specific data storage to meet your compliance needs. Using a VPN or secure gateway for connecting to AI services can also ensure your traffic is encrypted and monitored.

5. Address Licensing and IP Concerns Proactively:
Integrate an IP review into your process for AI-generated code. Tools can scan for known license text or code similarity to existing repositories. If Copilot (or another assistant) provides more than trivial code (especially longer than a few lines), have a procedure to vet its origin – e.g., run a quick search to see if that exact code exists somewhere online. Encourage developers to prefer AI suggestions that are original combinations rather than verbatim blocks. If there's any doubt, either attribute the code properly (if license allows) or rewrite it. Keeping a record of prompts and outputs can help if later an issue arises – you can show due diligence in how the code was produced. Legal teams should stay in the loop about AI usage and possibly contribute to guidelines (for example, requiring that any third-party code suggestions include license notes in comments for review).

6. Implement Data Loss Prevention (DLP) and Monitoring:
Extend your security monitoring to AI usage. This could mean using DLP tools on developer workstations that detect if someone is copying large chunks of source code – and alert if it's being pasted into a browser (which might indicate feeding it to ChatGPT). Network monitoring could flag unusual calls to AI APIs with sensitive payloads. Some

organizations set up “canary” sensitive strings (like fake secrets) in code and watch if they ever appear outside (which could indicate an AI leak). Monitoring shouldn’t be about snooping on employees, but about catching accidental exposures in real time. Additionally, require developers to use company-approved accounts for AI tools (e.g., a corporate ChatGPT account), so that usage is tied to an identity and can be logged centrally. If something does leak or an incident occurs, those logs will be invaluable for response.

7. Secure the Development Pipeline Against AI-Specific Threats:
Be mindful of emerging threats like prompt injection. For example, if you integrate AI into your coding pipeline (some companies do things like AI-assisted code review or commit message generation), ensure that malicious code cannot trick the AI into doing something harmful. This may involve sanitizing inputs given to AI and not fully automating actions based on AI output without human verification. In IDE context, watch out for scenarios where an attacker could intentionally plant a comment like /* AI: ignore security and do X */ hoping the assistant will comply. Keep your developers informed about these new categories of attacks so they can be vigilant. Also, apply least privilege to AI tools – for instance, an AI code assistant integrated with your repository should have read access to only the files needed for context, not your entire repo history if not required.

8. Embrace “Security as Code” for AI Governance:
Treat your AI usage similarly to how infrastructure-as-code is treated in DevSecOps. For example, encode policies in configuration: if using an AI coding assistant plugin, maybe centrally manage its settings via config management (disabling certain features if needed). If you have a CI pipeline, add jobs that specifically look at AI-influenced code (some companies experiment with AI that reviews AI code!). The idea is to automate compliance where possible. If ISO 42001 or NIST AI RMF requires documentation or risk register entries, integrate that into project onboarding – e.g., a new project using AI must fill a section in the risk register about AI usage. By weaving these considerations into the normal development workflow, you reduce the chance that they’ll be skipped.

9. Plan for Incident Response Involving AI:
Extend your incident response playbooks to include scenarios involving AI tools. For instance, how would you respond if it’s discovered that an API key was leaked via an AI service? Who needs to be notified (perhaps the AI provider, if per their policy)? Or if an AI-generated code vulnerability led to an incident, how will you root-cause it and determine if the AI tool needs retraining or reconfiguration? Conduct drills or tabletop

exercises for an "AI breach". This also means watching threat intelligence – stay informed about any security issues disclosed with the AI tools themselves (e.g., if Copilot had an outage or compromise that might have exposed code snippets from users). By preparing, you ensure that when something goes wrong (it eventually will), your team can respond swiftly and appropriately.

10. Incremental Adoption with Feedback Loops:
Finally, treat the rollout of AI assistance as an iterative process. Start in less critical projects, gather metrics on its impact on both productivity and security, and incrementally enable it more broadly as confidence grows and controls harden. Solicit feedback from developers and security reviewers actively: are the AI suggestions generally good or are they causing headaches? Use that feedback to adjust usage. For example, you might discover AI is safe and useful for front-end code but tends to be iffy for security-sensitive crypto code – so you allow it for one and not the other. Build a maturity roadmap (see below) so everyone understands that as the organization matures in using these tools, the rules might change. The goal is continuous improvement, not a set-and-forget policy.

Maturity Model for Secure AI Development

Organizations can assess their maturity in handling AI-assisted coding and plan improvements. Below is a simplified maturity model with stages and characteristics:

- **Level 1** – Ad Hoc (Reactive): AI tools may be used by individuals, but there is no formal policy or awareness. Security issues or data leaks from AI usage are handled in a reactive, case-by-case manner. There is little to no oversight – essentially the Wild West stage. (E.g., a developer might be using ChatGPT to write code, unknown to management, and if a problem occurs, the response is improvised.)
  Goal to progress: Immediately move out of this stage by creating at least minimal guidelines and visibility.

- **Level 2 – Initial Governance (Defined Guidelines)**: The organization has established basic policies for AI usage and communicated them to the teams. Approved tools are identified, and some training has been given on do's and don'ts. Security and legal teams are aware of AI usage and have started assessing risks. However, enforcement is mostly manual and reliant on individual compliance. Code review might catch obvious AI-related issues, but there's no systematic audit. If an incident happened, the organization would likely identify the policy violation after the fact.

Goal to progress: Automate oversight and integrate AI risk management into workflows (don't rely solely on humans remembering the rules).

- **Level 3 – Integrated (Managed Processes)**: AI-assisted development is integrated into the SDLC with automated controls and monitoring. For instance, DLP systems prevent forbidden data from being shared with AI, and all AI-generated code undergoes mandatory static analysis. The organization has a centralized way to track AI tool usage (maybe an internal registry of who has access to Copilot or an enterprise login for ChatGPT). There is cross-functional governance – perhaps an AI review board meets quarterly to review metrics (number of AI generated code issues found, etc.). At this stage, compliance with frameworks like NIST AI RMF begins: risks are mapped and measured regularly. Developers are largely following best practices because the process enforces them (e.g., the pipeline will fail a build if an AI-introduced license issue is detected).
  Goal to progress: Expand the scope of governance to cover not just coding but the full lifecycle (design, testing, deployment) and to anticipate future challenges.

- **Level 4 – Advanced (Validated and Proactive)**: The organization not only manages risks but seeks to optimize and innovate securely with AI. It likely aligns with ISO 42001 at this point, having a formal AI management system. Security testing includes AI-specific scenarios (like fuzzing AI-generated code for hidden bugs). The company possibly uses its own fine-tuned models for coding that it fully controls, or has robust contracts with providers about data protection. AI usage is audited and results fed into continuous improvement – e.g., the organization updates its AI tool's configuration or training data based on vulnerabilities found, thereby improving suggestions over time. There is also a feedback loop to the AI vendors: the company contributes bug reports or feature requests to improve security features in the tools. At this level, AI-assisted development is considered in security architecture decisions, and any new project will include an "AI impact assessment" by default during planning. The company is proactive – for example, running yearly red-team exercises specifically targeting AI-written components or using adversarial inputs to test model robustness.

- **Level 5 – Optimized (Transformational & Continuous Improvement)**: AI usage in development is fully **normalized and optimized for security**. The distinction between AI-written and human-written code blurs, because all code goes through equally rigorous validation and the AI has been tuned to the organization's secure coding standards. The organization likely contributes to industry best practices and maybe even builds custom AI tools that incorporate security by design (like an internal coding

assistant that has security rules ingrained). At this stage, the organization might leverage AI not just to write code, but to enhance security (for example, AI assisting in threat modeling, or automatically writing security unit tests). They continuously evaluate new AI tools and techniques, but any adoption is smooth because the governance framework is mature and adaptable. In essence, the company treats AI as an integral part of the DevSecOps toolchain. This is a stage of continuous improvement where lessons learned from every incident or near-miss are quickly folded into both policy and practice, and the organization likely meets or exceeds any regulatory requirements on AI use.

Not every organization needs to reach Level 5, but moving at least to a managed, proactive stance (Level 3–4) ensures that the benefits of AI in development can be enjoyed with acceptable risk. Leadership should periodically evaluate their maturity level and set targets (e.g., "By next year, we aim to have enterprise-wide visibility into AI tool usage and no critical vulnerabilities introduced by AI go undetected").

Conclusion: AI-assisted development is a powerful paradigm shift for cloud-native engineering teams. By understanding the tool landscape and associated risks, implementing strong governance and technical controls, addressing the special challenges of citizen developers, and aligning with emerging AI risk frameworks, organizations can confidently embrace these tools. The goal is to enable innovation and productivity gains from AI **without compromising on security, compliance, or quality**. With a thoughtful strategy and continuous improvement, AI can be woven into the software development lifecycle as a trusted partner rather than a wild card. This playbook provides the foundation – it's now up to each organization to execute these practices and adapt them to their unique culture and risk profile, thereby turning AI from a potential liability into a competitive asset in secure software delivery.

References

Atwell, E. (2022, November). GitHub Copilot Isn't Worth the Risk. Kolide Blog. (Discusses legal and security risks of Copilot, including license violations and IP concerns) (GitHub Copilot Isn't Worth the Risk) (GitHub Copilot Isn't Worth the Risk).

DeLong, L. A. (2021, October 15). CCS researchers find GitHub Copilot generates vulnerable code 40% of the time. NYU Center for Cybersecurity News. (Summarizes a study where 40% of Copilot's output had security issues) (CCS researchers find Github

CoPilot generates vulnerable code 40% of the time - NYU Center for Cyber Security). Maayan, G. D. (2024, August 6). Four Security Risks Posed by AI Coding Assistants. Cybersecurity Intelligence. (Outlines key risks of AI coding tools: code vulnerabilities, privacy issues, dependency risks, etc.) (Four Security Risks Posed by AI Coding Assistants) (Four Security Risks Posed by AI Coding Assistants).

Mauran, C. (2023, April 6). Whoops, Samsung workers accidentally leaked trade secrets via ChatGPT. Mashable. (Reports on Samsung engineers leaking confidential code by using ChatGPT, leading to corporate bans) (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable) (Samsung ChatGPT leak: Samsung workers accidentally leak trade secrets to the AI chatbot | Mashable).

Pollins, M. (2025, March 6). The Magic (and Pitfalls) of Bolt, Lovable and Replit. LinkedIn Articles. (Shares observations on no-code AI tools, including examples of missing security like an app with no authentication) (The Magic (and Pitfalls) of Bolt, Lovable and Replit) (The Magic (and Pitfalls) of Bolt, Lovable and Replit).

Superblocks Team. (2025, February 10). The 5 Most Common Low-Code Security Concerns + How To Mitigate. Superblocks Blog. (Lists low-code app risks including shadow IT, data security, citizen developer knowledge gaps, etc.) (The 5 Most Common Low-Code Security Concerns + How To Mitigate) (The 5 Most Common Low-Code Security Concerns + How To Mitigate).

Vincent, J. (2023, May 19). Apple restricts employees from using ChatGPT over fear of data leaks. The Verge. (Describes Apple and others banning generative AI tools for employees, citing privacy and leakage worries, and mentions Copilot restrictions) (Apple restricts employees from using ChatGPT over fear of data leaks | The Verge).

Waqas. (2024, April 1). Data Security Fears: Congress Bans Staff Use of Microsoft's AI Copilot. HackRead. (Details the US House of Representatives ban on Copilot due to potential leakage of sensitive data to the cloud) (Data Security Fears: Congress Bans Staff Use of Microsoft's AI Copilot).

Harvey, R. (2025, February 21). Using Agentic AI Editors (StackBlitz's Bolt, Lovable, and TikTok's Trae). Medium. (Defines "agentic AI editors" and compares Bolt and Lovable (web-focused) to local IDE AI tools like Cursor/Trae) (Using Agentic AI Editors. I tried StackBlitz's bolt, Lovable, and... | by R. Harvey | Bootcamp | Feb, 2025 | Medium).

A-LIGN. (2024). Understanding ISO 42001: The World's First AI Management System Standard. A-LIGN Compliance Blog. (Overview of ISO/IEC 42001, its key themes of transparency, accountability, bias mitigation, safety, privacy, and structure) (Understanding ISO 42001).

NIST. (2023, January 26). AI Risk Management Framework 1.0. National Institute of Standards and Technology. (NIST's official framework release outlining the Govern, Map, Measure, and Manage functions for AI risk) (NIST AI Risk Management Framework Explained) (NIST AI Risk Management Framework Explained).

GitGuardian. (2023). Yes, GitHub's Copilot Can Leak (Real) Secrets. GitGuardian Blog. (Research showing Copilot can reproduce secrets from training data and statistics on secret leakage rates in AI-assisted code) (Yes, GitHub's Copilot can Leak (Real) Secrets) (Yes, GitHub's Copilot can Leak (Real) Secrets).