

Security Perspective on Agentic Software Development



Key considerations in applying agentic workflows through the software development lifecycle

Authors :

- Dr Vesselin Tzvetkov Security Architect and Senior Cybersecurity Advisor
- Felix Gröbert Principal Engineer - Product Security

Table of Contents

01. Introduction	4
Overview of traditional development lifecycle	4
How does the developer's journey change and what does it mean for security?	5
Over the waterline: Front loading security in agentic software development IDE	5
Under the waterline: Extensive security checks in CI/CD	6
02. Foundational Principles of Agentic Software Engineering	7
Task Selection for Autonomous Agents	7
Probabilistic alignment	7
Granularity and risk containment	7
Data volume and quality	7
Considerations for Agentic Security Engineering	8
Securing Agentic Software Development	8
Secure Agent Harness	8
Agentic governance	9
Secure Platform and infrastructure	9
Maturity levels	10
03. Agent Framework for secure SDLC	12
Plan Phase	14
Code, Build and Test Phase	15
Deploy & Operate Phase and Collaborate	17
04. Future outlook and conclusion	18
Constant security fundamental aspects	19
Google references	19

01. Introduction

AI-assisted engineering has in a very short time transitioned from basic, autocomplete assistants to agentic systems capable of reasoning, planning, and executing complex, multi-step tasks in an autonomous manner. As software development shifts to incorporate agents, we need to rethink both securing AI use, and also how AI can help secure code, networks, and systems.

Using security agents in the development lifecycle offers significant security benefits, but also poses challenges. Agentic software development is key for handling the massive influx of AI-discovered flaws, collapsed time-to-exploitation of vulnerabilities, triage and review fatigue in security teams — and the commensurate, drastically increased speed required for continuous patching.

This paper provides an overview of agent advantages and outlines how to begin the journey, while providing an overview of the challenges.

Overview of traditional development lifecycle

Before we dive into what a development lifecycle augmented by security agents looks like, we first need to establish a typical development lifecycle.

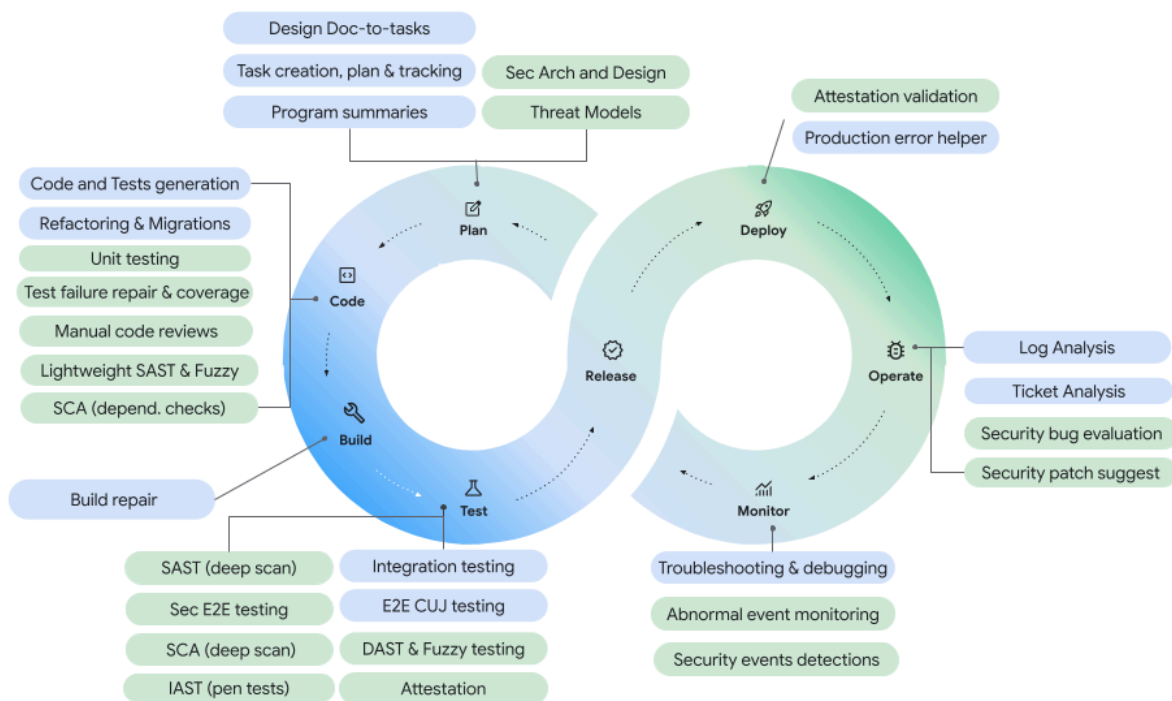


Diagram 1: Overview Security SDLC

Typical development lifecycle phases, like planning, coding, building, testing, releasing, deploying, operating, and monitoring, include multiple security tasks that are executed in one-off scans. Traditionally, one-off security scans take significant time, meaning developers might receive feedback and the ability to fix issues only an hour after submitting their code.

The diagram presents a generalized view of a development lifecycle and specific elements of the lifecycle may differ depending on your organization.

How does the developer's journey change and what does it mean for security?

Generally, manual security work in the Integrated Development Environment (IDE) will increasingly be replaced by agents; for example, addressing the growing burden on code and security reviewers, triage, threat modeling, and fixing caused by a higher volume of AI discovered flaws. Security engineers will spend significant time building, tuning, and securing agents (such as a secure [agent harness](#)) rather than manually reviewing and patching code (see [Approach for Secure AI Agents: An Introduction](#)).

Software developers now are in a similar place to [circuit engineers 40 years ago](#). At the end of the 1980s, the engineers stopped drawing, reviewing and signing individual transistors for digital chips to focus on reviewing the standard cells and gate arrays with abstracted transistor logics. Later in the 1990s, they stopped designing at the gate level altogether, and began using specific language to specify behavior. Today's CPUs have billions of transistors that would be impossible to design manually.

The traditional software development lifecycle, as previously described on Diagram 1, consists of two primary operational blocks. First is agentic development in the integrated development environment (IDE) — what the developer interacts with directly on their screen is "over the waterline." The second is the agentic development pipeline, encompassing the automated background processes such as validation, testing and deployment that occur after code is committed, "under the waterline."

Over the waterline: Front loading security in agentic software development IDE

As agentic software development accelerates the process and allows non-experts to generate code, it can make missing secure by default enforcement happen much faster, and with bigger impacts. The security tasks as shown in diagram 1 stay the same but become condensed, accelerated and front loaded in agentic development IDE. This significantly changes how security requirements are provided and fast and small lifecycles leveraging specialized agents, instructional workflows, and prompts are now embedded in the development IDE.

Running front-loaded fast security scans pre-submit, as opposed to traditional one-off security scans, allows fast security validation and fixes in pre-submit phase, before the changes are committed. The developers benefit from fast feedback with far less waiting time and overall much more secure code in the early stage. We build a secure [agent harness](#) and integrate security early into the agentic development workflows providing agentic plugins (skill, rules, helper function, and MCP tools integrations) as detailed later in section 3. An additional benefit is that technically scanning the small code change requires much less context for the agent and therefore is very cost effective.

Under the waterline: Extensive security checks in CI/CD

Significant in the development but less visible is CI/CD, what happens when the code is submitted in a repository e.g. post-submit phase, see Diagram 1, e.g. security reviewing, scanning, testing, validation etc.

Post-submit scans are performed in CI/CD and are extensive, covering much more context than front-loaded scans to provide comprehensive security insights. They are executed less frequently on code changes and require more time than front-loaded scans, but are necessary to gain deep security insights. Security agents will also be used for extensive post-submit scans and will handle the technical orchestration and control of embedded traditional CI/CD tools.

Rather than replacing tools that are well-suited for specific tasks, as detailed later in Section 3, agents act as a specialized, virtual agile security team leveraging these tools. Security agents run specialized, bounded personas, such as security code reviewer, tester, and threat modeler. Each agent has clear instructions, context, and tools. A main task of the security team is to create, tune, maintain, and secure the agent harness, rather than performing code validation on developer code directly. The use of agents in CI/CD collapses the lifecycle, as previously discussed, because execution is extremely fast; for example, one agent can detect a vulnerability and immediately trigger another agent to generate a potential patch, followed by a security evaluation within a single short cycle.

02. Foundational Principles of Agentic Software Engineering

Task Selection for Autonomous Agents

Autonomous agents operate on probabilistic frameworks rather than deterministic code. To optimize efficiency and mitigate risk, security task selection should be based on three core dimensions:

Probabilistic alignment

- **Optimal use cases:** Complex, exploratory tasks that tolerate non-deterministic outcomes (e.g., threat modeling, fuzz testing, incident investigation).
- **Anti-pattern:** Tasks requiring absolute or binary precision (e.g., SQL queries, regex-matching on credit card numbers), which should remain with traditional algorithmic tools. Leverage agents and LLMs to define the deterministic task, like define SQL query and trigger the execution on database, but not the perform query itself on data.
- **Governance and HITL:** While human-in-the-loop (HITL) guardrails are necessary for critical operations, consider that high-volume validation requests trigger decision fatigue and context-switching overhead. If a strict, non-negotiable security workflow exists, it should be hard-coded rather than delegated to agent discretion.

Granularity and risk containment

- **Atomic scope:** Agents should be restricted to small, isolated tasks with unambiguous success criteria and a constrained blast radius (e.g., generating a specific security unit test).
- **Anti-pattern:** Delegating subjective, high-stakes decisions based on vague criteria, such as authorizing production release readiness, is a critical anti-pattern.

Data volume and quality

- **Synthesis:** Agents excel at digesting vast, unstructured datasets to identify systemic patterns (e.g., parsing high-volume vulnerability telemetry to isolate root causes). The ideal operational model leverages agents for resource-intensive data aggregation, surfacing structured insights for final engineering sign-off and strategic remediation.
- **Anti-Pattern:** Expecting the agent to conclude tasks on wrong and missing data (e.g. garbage data) leading to wrong results causing a lot of effort for the human operator to fix.

Considerations for Agentic Security Engineering

The shift toward agentic workflows introduces new requirements for security teams:

- **Policy-as-Instruction:** Security requirements must evolve from static documents into declarative prompts and instructional workflows. This allows security patterns to be naturally embedded directly into the agentic IDE.
- **Governance of Agentic "Shadow Code":** Because agents can generate entire applications via natural language (agentic software development), there is a risk of rapid, uncontrolled deployments. Security must enforce "secure-by-default" configurations at various levels to prevent vulnerabilities from reaching production at an accelerated rate.
- **Reviewer and Triage Fatigue:** The sheer volume of AI-generated code is increasing the burden on human reviewers. This overloads the [Human-in-the-Loop](#) that is not more realistic to this speed. To counter this, security teams must equip themselves with Audit Agents to automate security reviews as much as possible. The code audit agents are highly specialized agents maintained by the security team. The Security engineers review the Agent that does the reviews, but not the code that the agent reviews.
- **Probabilistic Governance:** Security leaders must distinguish between tasks suitable for agents (probabilistic tasks like threat modeling or investigation) and those requiring traditional, deterministic tools. High-stakes decisions, such as final release approvals based on vague criteria, should remain human-centric to avoid catastrophic failures.
- **The "Blast Radius" of Malicious Prompts:** Traditional copy-paste attacks are evolving into prompt injection or "faulty memorized patterns" within models. Security engineering must focus on limiting the "blast radius" of agent actions to ensure a compromised agent cannot perform rogue actions across the entire codebase.
- **Unknown elements of AI/ML:** AI/ML systems represent a new attack surface with a new class of vulnerabilities and security requirements that the industry is still defining in real time, requiring new security practices that go beyond traditional AppSec.

Securing Agentic Software Development

Before leveraging agents for development and security tasks they need to be secured by default. We structure this in three main groups:

Secure Agent Harness

[Agent harness](#) is a software environment that surrounds the model and needs to be secure by default. Here are the main points to focus:

- **Agentic security:** LLMs add new attack vectors, as pointed out in Google's Secure AI Framework (SAIF), such as prompt injection. Follow [Google's Approach for Secure AI Agents: An Introduction](#), [Agent Gateway overview](#) and [Implementing Secure AI Framework Controls in Google Cloud](#)
- **Secure Agentic IDE:** Provide necessary hardening of the Vibe IDE, including used agents, [allow/deny](#) lists of URLs, separate [browser profile](#), validated extensions, plugins, API etc.
- **External tools and Model Context Protocol (MCP):** You should keep special attention on secure the external tools and external API, see [AI security and safe](#),
 [Foundations of Secure MCP: Architecture and Threat Model](#)
- **Context engineering with dedicated security agents:** Agents need the right security context in order to produce secure code: Just saying "do secure code is not enough". These artifacts need to be clearly formulated and embedded in a static and dynamic context of your security agents as explained later in chapter 3.
 - a. **Clear Security Instructions:** Agents need clear and precise security coding instructions (prompts) for secure coding, testing, and reviewing provided in [skills](#), [rules](#) and [plugins](#).
 - b. **Embed Secure Context and Architecture:** Provide artifacts of secure architectures, context, blueprints, and security patterns in the agents' context. This is critical to keeping the code base consistent with your enterprise standard and avoiding "spaghetti code".
 - c. **Prevent bias:** Keep rules, and context for each of your agents separate if they are naturally in separate roles to avoid influencing, like for example secure review and code developer. The same if valid between test, prod and pre prod.

Agentic governance

- **AI governance:** Agents must be governed, which includes managing agent identities, registries, and gateways following [Govern your agents](#) and well-recognized best practices, such as [ISO/IEC 42001:2023 - AI management systems](#) and the [AI Risk Management Framework | NIST](#).
- **Review and Governance:** Security reviews are critical for providing feedback on how to improve the agents. Avoiding over-reliance ("Citizen developer" risk) and preventing code quality degradation are key governance challenges.

Secure Platform and infrastructure

Secure your platform runtime, development environments, and integration with code where the AI Agents are developed and deployed, see the [Google Cloud security best practices center](#).

Maturity levels

Enterprises utilize established governance and software development flows consisting of approval gates, compliance checkpoints, and security reviews. While this structure remains, agents will execute the coordination and orchestration under human and audit oversight.

	Level 1: Task-Specific Agents (Single Task Agent)
Scope	Dedicated execution of single-purpose security actions within the existing workflow
Examples	Writing unit tests, conducting initial security code reviews on Pull Requests, or assisting with information-seeking tasks to reduce the ~5 hours developers spend searching for data each week
Human Role	Direct oversight and manual validation of every agent output
Enterprise Impact	Primarily acts as a "productivity booster" for individual contributors, mitigating the effects of frequent interruptions

	Level 2: Multi-Agent Group (Agent Squads)
Scope	Modular grouping of agents to perform end-to-end "mini-workflows"
Examples	A "Testing Constellation" where one agent writes security tests while another executes them and a third presents a summarized report for human approval
Human Role	Reviewing process outcomes and managing the "handoff" between different agent groups
Enterprise Impact	Begins to address "tool sprawl" by providing a unified agentic interface over the hundreds of fragmented tools typical in enterprise environments

	Level 3: Fully Orchestrated Governance Flow (Agent Swarms)
Scope	Execution of complex, multi-stage flows including automated approvals and cross-tool integrations
Examples	Fully automated vulnerability remediation flows-from detection and exploitability analysis to patch generation and deployment-integrated with enterprise ticketing systems
Human Role	Strategic oversight, performing high-level cross-checks, and acting as the final "governance gate" for high-stakes decisions
Enterprise Impact	Directly counters the "workflow multiplier" risk by enforcing secure-by-default logic across the entire organizational pipeline, effectively removing the manual bottlenecks and waiting states that currently stall enterprise delivery

03. Agent Framework for secure SDLC

The transition toward agentic development enables a significant contraction of the software development lifecycle, allowing for the rapid execution and aggregation of complex operations within local environments - front loaded security checks (pre-submit checks) and following CI/CD deeper security scans (post-submit check). The two cycles complement and deliver fast and comprehensive security analysis.

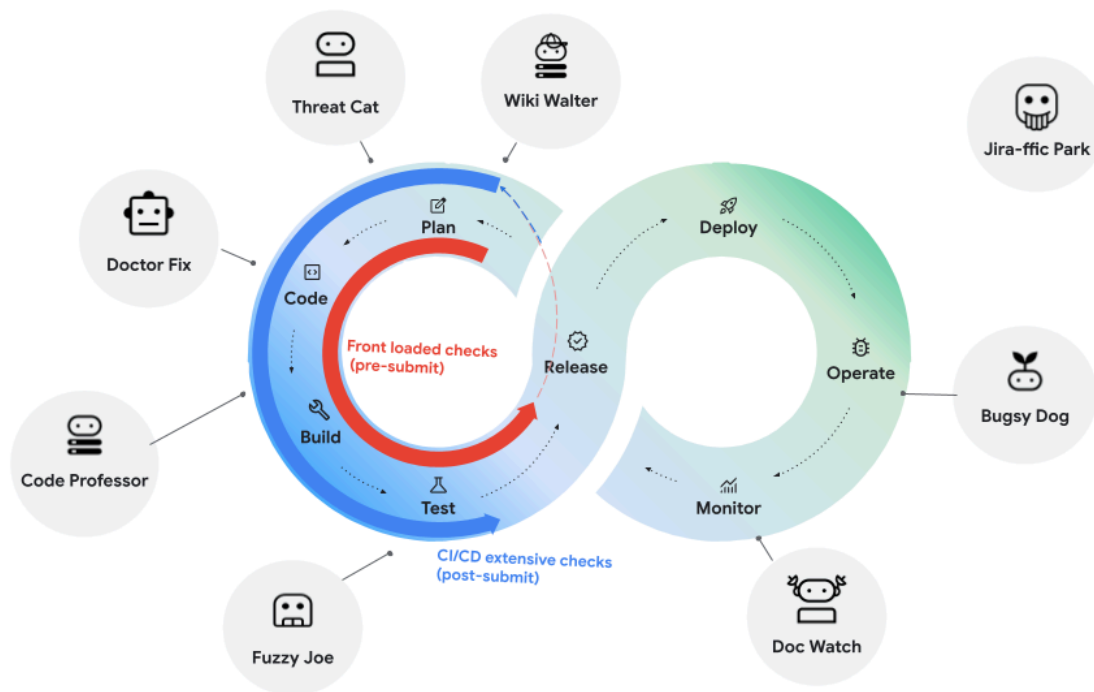


Diagram 3: Agentic Security Personas

The new framework operates on the principle that "code is cheap, but assurance is costly". This framework requires agentic threat modeling, triage and analysis, Proof of Concept (PoC) for vulnerability exploitation, secure agentic code generation, and industrialized "VulnOps" pipelines to handle the staggering volume of AI-discovered vulnerabilities. Components include agentic testing (fuzzing and formal verification) and system-level deterministic platform constraints.

Let us create figurative agents characters, e.g. a group of agents that help deliver value in the different development phases: Wiki Tom, Threat Cat, Doctor Fix, Code Professor, Fuzzy Joe, Buggy Dog and Doc Watch. Each of these agentic personas need to be secured as already mentioned and described as follows.

Agentic personas mimic specific security skills and operate with specialized tools that can be tailored to an organization's unique requirements. Drawing from our security experience, these agents deliver significant value by streamlining workflows, though the actual complexity and security gains vary based on how well-suited a given task is for autonomous execution. The tables below detail the specific personas mapped across each phase of the development lifecycle, the role of the developer (human),

the agents, and security values as tasks. They are grouped over the phases in the development: plan, code, test, release, deploy and operate.

Plan Phase

Name	 Wiki Tom: Security Architecture Documentation, Policy and Blueprints	 Threat Cat: Threat Modeling Agents	 Threat Cat: Threat Intelligence Agent Predictive OSINT
Description	Agents performing document analysis and defining the structural layout, summaries, and reusable security artifacts.	Systematically identifying potential threats and vulnerabilities in the design phase and vulnerability reports via Threat Modeling. Crucial for prioritizing security controls early.	Gathers and correlates external Open-Source Intelligence (OSINT) and internal vulnerability data to proactively identify evolving attack patterns.
Agent's Role	Analyzes code and documentation; generates blueprints, policies, and architectural diagrams based on natural language or code requirements from architects.	Ingests design documents or vulnerability reports and autonomously generates threat models, identifying edge cases humans often miss, by leveraging internal and external databases of known threats.	Monitors dark web activity and public threat feeds (e.g., Google Threat Intelligence's dark web intelligence capability) and maps newly discovered exploit chains directly to the organization's existing asset inventory and active projects.
Examples	Ingests a high-level requirement ("GCP microservices with mTLS") and outputs a compliant architecture diagram and low-level design. Analyzes code and comments to generate up-to-date security documentation.	Feeds a data flow diagram (DFD) to an LLM which outputs a STRIDE analysis matrix with specific mitigation suggestions. Analysis of external vulnerability reports and constructs threat models as input for bug analysis (Bugsy Dog).	Identifies a newly released zero-day exploit chain on a niche library and flags the three specific production repositories using it.
Human Role	Architects shift from writing boilerplate to reviewing agent-generated designs for business logic alignment.	Developers refine the agent's findings rather than starting from a blank page. Humans are irreplaceable here for understanding the interplay of system components, out-of-the-box thinking, real-world company structures, operational realities, tooling landscapes, and the presence of "Shadow IT."	Security analysts use the agent's insights to refine threat scenarios for Threat Cat and update overarching organizational risk profiles.
Security Value	HIGH - Creating secure baselines with all parameters properly set is a massive win, removing manual documentation deep-dives. Corner cases, undocumented features, and greenfield areas will still require human checks and intensive support.	MEDIUM - Alleviates a traditional manual bottleneck by creating baseline versions with well-known threats. However, deep security expertise remains essential to scope the integrations properly, prevent "check-the-box" exercises, and clean up the foundational data warehouses the agent operates on.	HIGH - Provides early warning of new threats, successfully shifting threat modeling workflows from reactive to predictive.

Code, Build and Test Phase

Name	 Doctor Fix: Code & Security, PoC and Patch Generation	 Doctor Fix: Code Hardening Agent and Refactoring	 Code Professor: Source Code Analysis & PR Review Agent	 Code Professor: Supply Chain Agent and Dependency Guard	 Fuzzy Joe: Automated Fuzzing, Deep Scan & Application Testing (SAST & DAST)
Description	Analyzing code to determine the most effective place to remediate a bug or vulnerability and suggesting a precise patch. Generation of Proof of Concepts (PoC)	Automatically refactors or rewrites code from legacy languages to memory-safe alternatives or applies standardized hardening patterns at scale.	Continuous review of code changes and vulnerabilities to catch logic errors, security flaws, and evaluate exploitability.	Focuses on third-party dependency security by analyzing open-source software (OSS) components for vulnerabilities, licensing conflicts, and malicious injections.	Testing applications via Fuzzing (stressing with semi-random inputs) and DAST/Continuous Penetration Testing (simulating active attacks). Leveraging Code professor to generate Proof of Concepts (PoC)
Agent's Role	Analyzes a vulnerability and the surrounding codebase to generate a tailored remediation patch. Proof of Concepts (PoC) based on Fuzzy Joe Agents or Buggy Dog.	Identifies large sections of code that can be automatically migrated to a secure language (e.g., C to Rust) or automatically inserts mandatory security constructs into existing pipelines.	PR Review: Provides the "first pass" review, summarizing changes, explaining downstream impacts, and flagging anti-patterns. Vulnerability Analysis: Analyzes the code paths to see if the vulnerable function is actually used (reachability/exploitability reasoning).	Manages the dynamic Software Bill of Materials (SBOM), assesses the exploitability of all transitive dependencies, and flags or quarantines untrustworthy components. Triggering and updating depending.	Fuzzing: Generates malicious stress inputs, monitors application state/logs, and mutates input to influence application behavior. DAST: Acts as an "Automated Red Teaming" agent, continuously attacking the app with evolving strategies based on threat model scenarios. SAST: Deep code scan for code vulnerabilities in intensive analysis
Examples	Injection vulnerability in an API endpoint and suggests a specific fix utilizing input validation or parameterized queries.	Automatically refactors a legacy parsing utility into a modern language, stripping out potential buffer overflow vectors.	Evaluates a Pull Request to check if cryptographic libraries are being instantiated correctly. Analyzes code causing memory corruption to deduce if it can lead to arbitrary command execution via a public API.	Detects a typosquatting package attempt in a package manifest and quarantines it before the build server pulls it down.	Generates massive amounts of unstructured data causing an application to crash in an indefinite state. Executes automated attack paths against active API endpoints to

Name	 Doctor Fix: Code & Security, PoC and Patch Generation	 Doctor Fix: Code Hardening Agent and Refactoring	 Code Professor: Source Code Analysis & PR Review Agent	 Code Professor: Supply Chain Agent and Dependency Guard	 Fuzzy Joe: Automated Fuzzing, Deep Scan & Application Testing (SAST & DAST)
					validate if a Threat Cat mitigation works.
Human Role	Senior developers and security engineers review the generated code changes and validate the patch before deployment.	Validates the functional correctness of the rewritten code and ensures secure coding patterns are adopted uniformly across the organization.	Reviewing the agent's conclusions, providing steering inputs for Doctor Fix, and shifting focus to higher-level architectural fit and business logic.	Approves or denies new dependencies based on the agent's risk assessment and manages the strategic remediation of critical supply chain vulnerabilities.	Initializing test parameters (cold start), validating core assumptions/results, providing overall testing strategy, and adjusting the scope.
Security Value	MEDIUM - Highly dependent on the quality, architecture, and maintenance maturity of the existing codebase.	HIGH - Proactively eliminates entire foundational classes of vulnerabilities (such as memory unsafety) across the enterprise codebase.	HIGH (PR Review) / MEDIUM (Vuln Analysis) - Significantly reduces cognitive load on senior devs though its added value over deterministic, modern AST/SAST tools is still being evaluated.	HIGH - Maintaining the integrity of the software supply chain is an essential of the development lifecycle. Agentic oversight combined with traditional dependency analysis tools to deliver highly effective security outcomes and solves the "dependency hell" of non-exploitable libraries,	HIGH (Fuzzing/SAST/DAST) / MEDIUM (Pen Test) - Outstanding at generating unstructured outputs and finding unexpected crashes. However, human creativity in pen-testing remains vastly superior for parsing high-risk business logic flaws.

Deploy & Operate Phase and Collaborate

Name	 Bugsy Dog: Issue and Bug Triage and Analysis	 Doc Watch: Runtime, Development and Agents Monitoring	 Doc Watch: Posture, Attack Surface and Compliance Agents	 Jira-ffric Park: Cooperation and Task Management Agent
Description	Determining if an active production or development defect is security-relevant, performing initial triaging, establishing root causes based on logs and preparing post mortems	Observing system health metrics, logs, and internal agent actions to find abnormal behaviors, crashes, or prompt injection attacks.	Provides continuous security posture management by translating declarative governance policies into machine-readable checks based on a single "Security Bar." Ensures compliance and privacy controls are maintained across the entire workflow.	Synchronization of tasks between teams, ticket boards, metrics, and reporting. While not purely security-focused, it heavily optimizes security collaboration.
Agent's Role	Analyzes the incoming defect text and crash logs to estimate its root cause, priority, and overall security severity. Triggering PoC and testing with Fuzzy Joe.	Correlates massive logs to find "needles in a haystack" representing abnormal executions. Monitors the inputs, outputs, and chains-of-thought of all other agents globally to enforce security policies and stop malicious manipulation.	Monitors assets against the security bar based status and on the output of the monitoring agents, identifies definition to the target posture, automatically suggests remediation, and maps internal rules to industry standards (NIST/CCM) and compliance requirements.	Assigns stories to the right teams, structures tickets according to specific team needs, flags duplicates, manages notifications, and proactively prepares work backlogs.
Examples	Processes a generic crash report and, by looking at relevant code parts, determines that it is a buffer wrap-around with high potential security impact, upgrading priority.	Detects unusual library usage with an exfiltration pattern and alerts the SOC Detects a hidden prompt injection in incoming code trying to trick Code Professor into reporting a false secure status.	Detects that encryption at rest is not activated in the build worker node that is a requirement in security policy and reports a risk with potential remediations.	Detects an incoming bug issue, identifies that it is a duplicate of an existing vulnerability report, links them, and schedules it into the correct team's upcoming sprint planning.
Human Role	Reviews, approves, and adjusts the severity rankings and technical assignments provided by the agent.	Defining core metrics, tuning detection criteria, validating incident outputs, and responding to SOC alerts.	Reviews and confirms policy-as-code translations and addresses high-severity misconfigurations flagged by the agent.	Reviewing the agent's work queues, granting final planning intent approval, and aligning tasks with business goals.
Security Value	HIGH - Rapidly accelerates triage and determines severity efficiently by instantly reviewing relevant code segments alongside log context.	LOW (Logs) / HIGH (Agent Guard) - Streamed log analysis with AI is still early-stage and costly, but Agent Actions Monitoring is absolutely essential and a mandatory requirement to guarantee the integrity of an agentic CI/CD loop expertise	HIGH - Directly enforces "secure-by-default" layouts at the operational layer, actively preventing policy drift and scaling corporate compliance efforts. Critical for managing enterprise compliance and privacy risks in an scalable way.	HIGH - Offloads significant administrative planning and coordination overhead from engineering teams, ensuring cross-functional security alignment via a structured, human-moderated model.

04. Future outlook and conclusion

Even the future can not be predicted and there are some clear threads that we think will become the state of the art.

Hand made

“Hand made” code or review will become rare - these tasks will be taken by agents and the developer will focus on the abstraction layer and concepts. The change can be compared figuratively like when the engineers stopped “reviewing and signing” individual transistors for digital chips at the end of the 1980s and focused on reviewing the logic blocks in Logic Synthesis, considering that modern processors consist of billions of transistors as of today [IEEE [Electronic design automation \(EDA\)](#)].

Pre-baked solutions

The pre-baked solution will play an important role: blue prints and proven solution blocks will become massive tools for agents that use them as in Lego boxes. The exact integration will be made by the agents.

Secure Agent Harness

The developer will be working on agent engineering, which is a new emerging discipline that will become more and more important, also with facets of security of agents and their actions. We have not yet found all the protection options, so there is a lot to do in this space.

Abstraction, Architecture and Code Development

The abstraction view of the architect will be more critical for the development also the abstract thinking as an important skill for the future developer will drive them toward security architecture rather than low level code details. Still, there will always be niche areas where sufficient knowledge is collected to build Agent and there developer will still be needed to help gaining this initial know-how.

Skills development

The traditional path of starting as a junior developer writing small snippets of code and gradually progressing to complex systems and architecture is likely to change. We can expect the transition to architectural thinking to become much faster and more efficient. This shift should begin as early as the academic and pre-professional phases of a career. Determining how to cultivate deep architectural competency and abstract thinking without requiring decades of traditional development experience remains an open challenge, particularly in mitigating the risks associated with “citizen developers.”

Non-Technical Organizational Bottlenecks

Many enterprise hurdles are inherently non-technical-stemming from interpersonal dynamics, internal competition, unclear processes, responsibility gaps, perpetual re-organization and shifting business strategy. The degree to which agentic systems can resolve these high-level frictions remains a significant open question, though we anticipate a transformative positive impact of agents serving as a collaboration catalyst.

Open Source Software

Open source is also undergoing a transition due to the lower cost of code production, which will likely lead to more frequent forks and proactive refactoring for security reasons. However, open source is more than just code; it represents a philosophy of optimal architecture, API standardization, and business case specification. While the intrinsic value of raw code may shift, the importance of architecture as a reference point for security and stability is likely to remain constant.

Constant security fundamental aspects

Despite the upcoming paradigm shift, three fundamental aspects of security engineering will remain constant:

1. Outside-In Strategy: Security programs will continue to be shaped by two primary factors: the demand on security by business strategy and priorities, aka security appetite, and business risks that are influenced by shifting tactics of attackers and regulation.
2. Human-Centric Strategic Governance: AI can significantly enhance posture management and take tactical decisions in order to help us scale. Still humans will remain to define the strategic goals and be the ultimate arbiters of what constitutes "good" software at strategic level.
3. The Security Flywheel and [OODA](#) (Observe, Orient, Device, Act) Cycles: The technical loop of finding and fixing technical risks will persist at higher abstraction levels, even as the specific execution methods evolve.

Google references

- [CodeMender](#)
- [CodeMender overview](#)
- [OSS-Fuzz](#)
- [Agentic AI for Security Operations](#)
- [Changing the game: Using agentic AI to secure infrastructure code](#)
- [AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms - Google DeepMind](#)
- [Google Antigravity](#)
- [How Google Does It: Threat modeling, from basics to AI](#)