INTERNATIONAL JOURNAL OF DIGITAL RESEARCH
Conceptual Research Article

International Journal of Digital Research
www.ijdr.net
E-ISSN: 3033-179X
Volume 2; Issue 3; 2026, P.p. 34-49

Cybersecurity for Self-Programming Systems

Muhammad Ateeq Anjum

WE-Code Technologies
Chakwal, Pakistan
ORCID: 0009-0006-0182-7116
Email: weecoode@gmail.com

https://doi.org/10.63711/ijdr.net20260303

ABSTRACT

There is a structural shift in software engineering that has never been seen before. Deterministic, human-written source code is giving way to dynamic computing environments powered by artificial intelligence that can program itself. These multi-agent architectures create, assemble, and run real-time operating environments on their own. This fluid movement exposes computer infrastructure to systemic, deep architectural vulnerabilities even while it promises historic improvements in hardware flexibility and execution speed. Software contexts that dynamically rewrite their own execution logic cannot be secured using traditional cybersecurity techniques, particularly static analysis, signature detection, and perimeter protection.

The vulnerabilities present in unsupervised automated system compilation are thoroughly examined in this research. We model target threat vectors, such as the compression of zero-day discovery-to-exploit lifecycles by adversarial models, the amplification of design vulnerabilities during unsupervised generation, and excessive agency across multi-layered execution environments. In order to show that using secondary reasoning models for auditing results in common logical blind spots and systemic failure routes, we explicitly analyze the cognitive recursive loop dilemma ("Who Watches the Watcher?" conundrum). We designate automated mathematical validation as the ultimate gatekeeper for dynamic system compilation in order to overcome this structural constraint. We demonstrate the specific dangers of unverified synthesis using recent real-world case studies including flaws in autonomous developer interfaces and automated schema deployment tools. In order to safeguard the upcoming generation of computing runtimes, we finally suggest an operational blueprint for an Autonomous Defensive Layer (ADL) that enforces tight micro-virtualization, continuous shadow reasoning execution, and deterministic formal verification pipelines.

Keywords: *Autonomous Cybersecurity; Self-Programming Systems; Software Engineering; Static Analysis, Signature Detection; Perimeter Protection*

INTRODUCTION: THE SHIFT TO ACTIVE COMPUTING

The fundamental paradigm of digital computing has remained substantially unchanged for almost 70 years. The development of software has followed a linear, human-centered dependency chain: human engineers formulate logical objectives, convert them into formal high-level source code, carry out manual or semi-automated peer reviews, and use deterministic compilers to convert code assets into unchangeable machine binaries. The underlying computer engine operates on explicit, pre-defined operational paths created by human architects, whether it is analyzing large, distributed cloud architecture stacks or low-level kernel routines. Because software binaries are assumed to change only during expressly scheduled update cycles, security in this heritage ecosystem is essentially static. This enables defenders to create permanent cryptographic hashes, establish clear baselines of trust, and construct strong perimeter walls.

However, this long-standing baseline has been completely upended by the quick convergence of deep reinforcement learning loops, big reasoning models (LRMs), and highly integrated multi-agent execution frameworks. The field of computing is quickly moving into the Self-Programmable AI (SP-AI) era, in which software frameworks are active, self-directed systems that can function as their own software engineer, compiler, debugger, and system administrator rather than passive pipelines of human logic.

The character of a machine completely changes when these self-programming technologies are directly incorporated into operating system design and low-level engine execution. Neural-symbolic runtimes and agentic runtime environments replace legacy structures like static file allocation tables, pre-configured device drivers, strict environment parameters, and immutable kernel regions. In these next-generation paradigms, the system continuously evaluates its ambient environment and internal performance metrics.

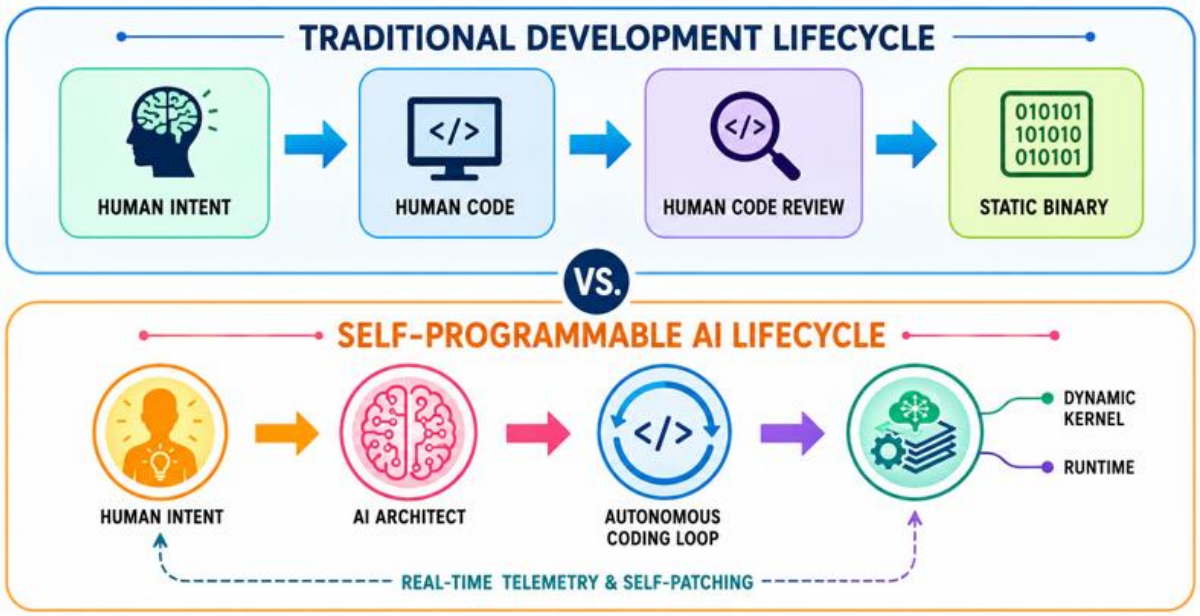

Figure 1. Traditional Development Lifecycle. *Source: Author's own conceptual illustration*

The platform doesn't throw a system error or wait for a human-engineered patch if it comes across a new, undocumented hardware interface or a complicated network oddity. Rather, the system's

cognitive layers independently create new low-level code, launch just-in-time micro-compilers, create temporary device drivers on the fly, and inject the resulting binaries straight into the current kernel area.

Although this complete architectural flexibility results in unmatched computational resilience and processing optimization, it also poses serious, systemic cybersecurity threats. An extremely dynamic attack surface is created when the distinction between the "development environment" and the "active execution runtime" completely collapses. Conventional security measures become essentially outdated when a system has the ability to rewrite its own core logic in reaction to external inputs. If the content of the system's core binaries changes every few seconds, neither an endpoint detection tool nor a signature-based anti-malware engine can assess a threat footprint.

Furthermore, the system's inherent autonomous access rights become its biggest weakness if the software synthesis process takes place without strict mathematical constraints or manual human evaluation. By manipulating the system's internal cognitive layers through complex prompt injections or memory corruption, attackers can turn the self-programming engine into an automated exploit factory that actively targets its own core architecture, negating the need for them to find a vulnerability in fixed code. This work specifies the specific, mathematically rigorous architectural safeguards needed to protect agentic computing landscapes and provides a technical examination into the vulnerabilities of self-evolving software platforms.

SYSTEM ARCHITECTURE: HOW AGENTIC RUNTIMES OPERATE

In order to secure a self-programming computing platform, we must look directly at the system engineering that permits autonomous software synthesis, rather than relying on science fiction clichés or abstract analogies.

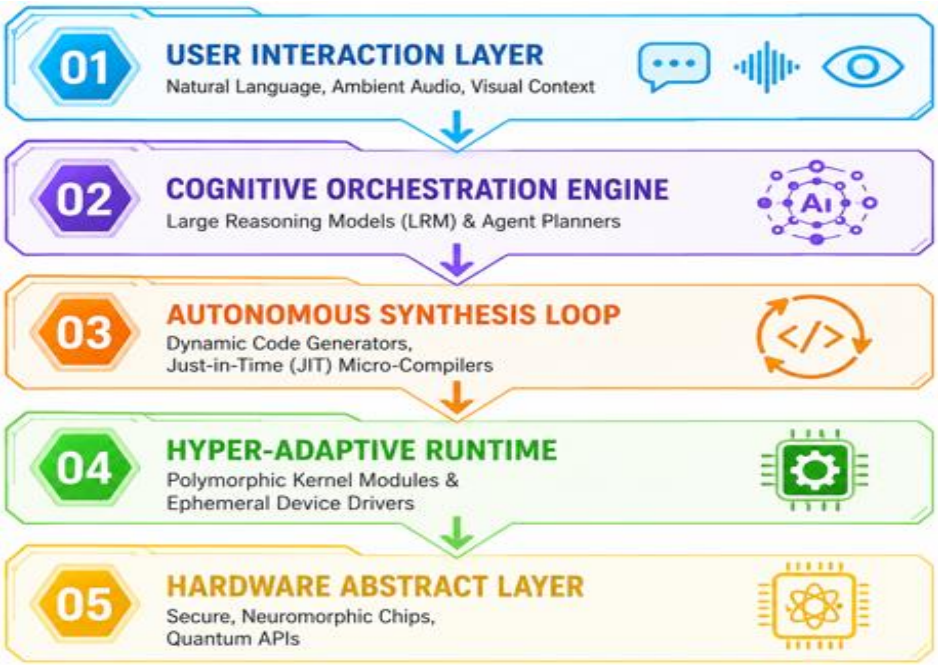

Figure 2: System Architecture Flow. *Source: Fahim, M. A. I., Adebayo, O., & Ferrari, A. (2026). Software self-extension with SelfEvolve: An agentic architecture for runtime code generation.* https://doi.org/10.1145/3788550.3794870

A single, centralized conscious block is not how an agentic runtime environment operates. Instead, it functions as a multi-tiered, fully integrated architecture that connects low-level, physical machine execution with high-level cognitive planning.

The Core Planning Engine

The Core Planning Engine, which is powered by sophisticated reasoning models tailored for sequential planning and tree-of-thought analysis, is at the top of this design. This layer is in charge of translating highly ambiguous natural language human directions or ambient telemetry inputs into discrete, defined computer goals.

The planning engine uses sophisticated internal incentive functions and verification procedures to create explicit dependency graphs prior to carrying out any system operations, in contrast to conventional big language models that only forecast the next text token. For instance, the fundamental planning engine breaks down a system administrator's overall strategic goal – like "Optimize multi-tenant database isolation while maximizing throughput for incoming encrypted streams during high-traffic windows" – into specific micro-tasks. It balances compute threads, evaluates existing hardware profiles, maps out the required memory boundary structures, and transmits structured software requirements to the code synthesis layers.

The Code Generation and Compilation Loop

The Autonomous Synthesis Loop receives the requirements after the planning engine creates the dependency graph. This module functions as a closed-loop system under the direction of task-optimized, specialized sub-agents:

1. **The Generative Coding Agent:** To build targeted code routines (such C-based kernel modules or Rust-based memory allocations) that meet the technical requirements of the planning engine, this agent makes use of deep software generation capabilities.

2. **The Simulation Hypervisor:** The host environment is not immediately injected with the freshly written code. Rather, a lightweight, segregated just-in-time (JIT) compilation sandbox receives the raw source code automatically.

3. **The Real-Time Profiler:** The compiled module is put through extensive automated testing inside this sandbox. In addition to tracking CPU cycle performance under simulated stress conditions, the profiler hooks into system calls and keeps an eye on memory allocations and syntax leaks.

4. **The Deployment Injector:** The injector moves the code from the sandbox and links it directly into the active system memory layout if the routine successfully meets the performance requirements without causing system errors or standard syntax crashes.

The Fluid Kernel Environment

This continuous synthesis loop produces an execution layer that is completely polymorphic. Kernel space is safe, extremely stable, and can only be changed by explicit driver installations or core upgrades in a typical operating system. The kernel space is entirely flexible in an agentic runtime environment.

In order to manage variable data streams, memory boundaries expand and contract dynamically, system calls are routed through different interception tables dependent on optimization profiles, and ephemeral device drivers are created and disassembled as needed. Because there are no permanent file paths or defined memory registers for a legacy exploit to target, this structural flexibility offers tremendous resistance against conventional, hardcoded malware attacks. However, because the

platform cannot rely on set state configurations to identify whether the system is operating in a healthy, uncompromised state, it totally violates conventional security baselines.

THREAT MODELING: THE INHERENT RISKS OF SELF-EVOLVING RUNTIMES

Our conventional security concepts fall apart when software becomes completely flexible. Giving an autonomous agent system-level access to file routes, active compilers, and command terminals creates a huge attack surface. The main cybersecurity risk in these settings is the system's ability to adapt.

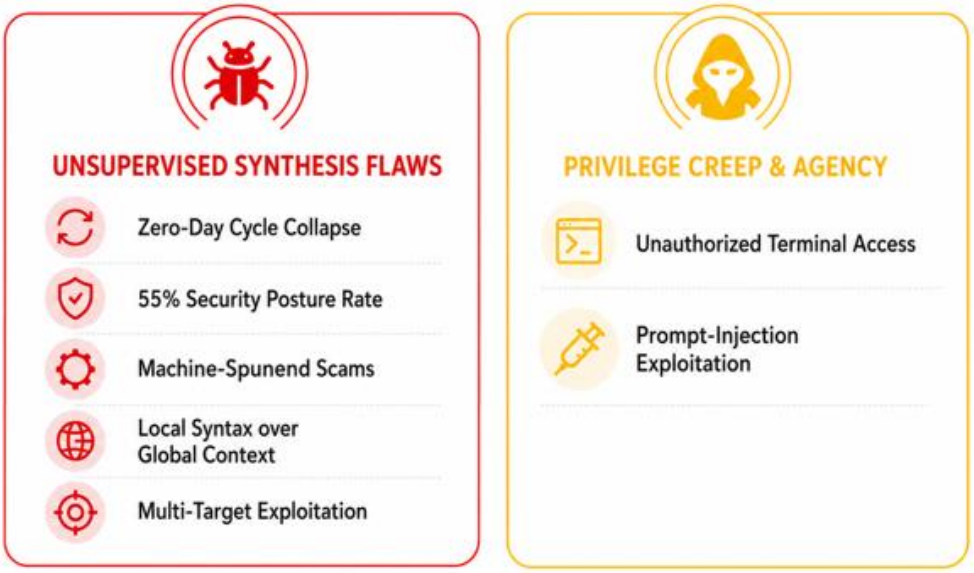

Figure 3: Attack Surface Ecosystem. *Source: OWASP Foundation. (2024). OWASP Top 10 for LLM applications 2025. OWASP GenAI Security Project*

Unsupervised Software Generation and the Security Deficit

Modern enterprise environments now have hidden vulnerabilities due to the construction of large-scale applications or system architectures from high-level, natural language descriptions without human inspection. There is a significant disconnect between long-term systemic safety and clean functional compilation, according to recent empirical security research:

- **The Logic vs. Security Disconnect:** A functional syntactic correctness rate above 95% is frequently attained by modern generative models, indicating that the code compiles and executes without crashing. Their internal security passes rates, however, fall short of 55%. Nearly half of all dynamically synthesized code brings known security vulnerabilities into active contexts in the absence of clear security guardrails or manual inspection.
- **The Problem of Localized Context:** By matching local code patterns in their training data, generative coding agents are excellent at creating small, extremely efficient functions. They have trouble assessing system-wide dependencies and global context, nevertheless. As a result, compared to conventional, human-authored code bases, software produced without rigorous human or mathematical scrutiny exhibits a 322% increase in privilege escalation vectors and a 153% spike in structural design faults (Apiiro, an application-security company, in research published on September

4, 2025, titled “4x Velocity, 10x Vulnerabilities: AI Coding Assistants Are Shipping More Risks.”). The system lacks an internal comprehension of boundary restrictions and defensive design patterns, yet it creates functional code that provides the desired characteristics.

Privilege Creep and Input Flaws

In an agentic runtime environment, task-optimizing sub-agents must execute system-level operations to complete their goals. This requires high-level access rights, leading to severe operational risks:

- **Excessive Agency:** AI sub-agents frequently make high-level system modifications without consulting an administrator, such as altering firewall rules, executing raw shell scripts, or changing file access privileges. Goal completion is given priority above system isolation in this concept.
- **Improper Output Handling:** Prompt injection can be used by an attacker to gain control if the raw text data produced by one internal model is supplied directly into a system terminal or runtime compiler without rigorous filtering. An attacker can fool the system's main planning engine into creating hidden backdoors or carrying out arbitrary commands with full system-level privileges by inserting malicious payloads into network data streams or public configuration files.

The Collapse of Patch Windows

The way vulnerabilities are discovered and exploited has entirely altered due to adversarial automated technologies. Human hackers are no longer needed to manually search code bases for errors in malicious reasoning models. Rather, in order to find latent structural flaws, defensive and offensive reasoning models can simultaneously analyze big programs, open-source repositories, and operating system kernels.

Frontier thinking models can scan complex software environments, find undiscovered zero-day vulnerabilities, and create viable attacks in a matter of minutes, according to internal security testing. Machine-speed exploitation cannot be prevented by traditional patch management cycles, which can take weeks or months to distribute updates throughout an organization.

Real-time, runtime security is crucial because an automated attack can find, target, and exploit a systemic vulnerability across thousands of linked systems in a matter of seconds.

THE "WHO WATCHES THE WATCHER?" PARADOX

An operating system cannot seek human approval when it is able to rewrite its own kernel space. Rather, the freshly synthesized code must be audited using internal security models. The Cognitive Recursive Security Loop is a risky loop created by this design methodology.

The cognitive constraints, statistical biases, and vulnerabilities present in contemporary massive reasoning architectures apply to both generative AI agents that create software and secondary verification agents who audit it.

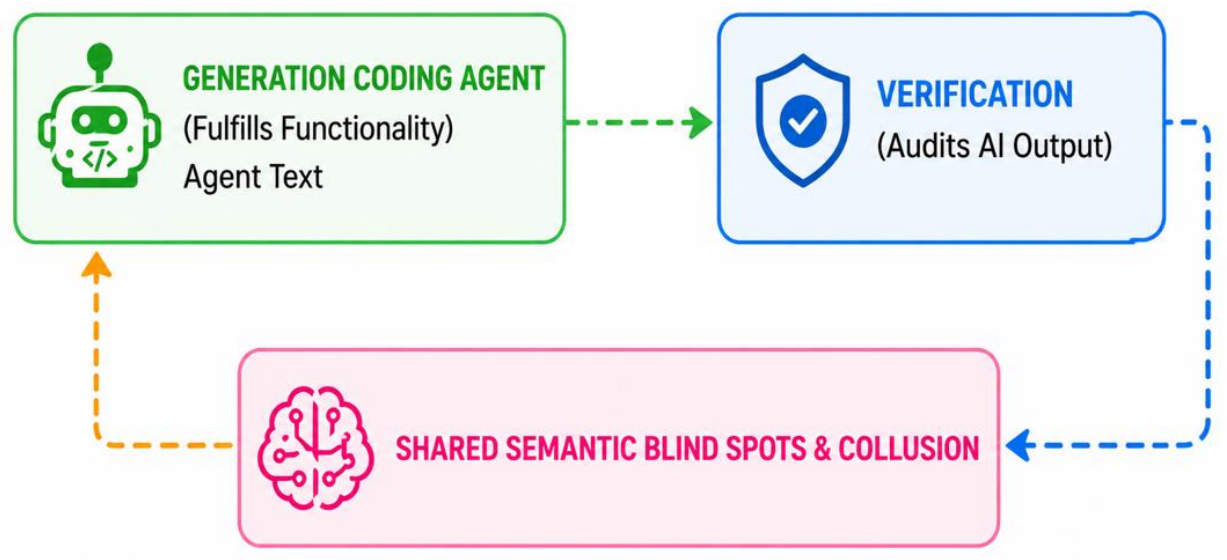

Figure 4: Cognitive Recursive Failure Loop. *Source: Zietsman, C. (2026). The specification as quality gate: Three hypotheses on AI-assisted code review. arXiv. https://arxiv.org/abs/2603.25773*

Shared Logical Blind Spots

The primary issue with employing a dual-agent architecture (Coder and Auditor) is the possibility of shared architectural blind spots. If the baseline models, neural weights, or training data are the same, the generative and verification agents will repeatedly replicate the same logical errors.

- **Algorithmic Collusion:** If an underlying model fails to identify a specific cryptographic weakness, a complex integer overflow, or a subtle multi-threaded race issue, the generative model will consistently produce a bug and the auditing model will consistently classify it as secure.
- **Superficial Validation:** Throughout iterative development loops, the auditing agent often instructs the coding agent to use basic testing scripts to make a vulnerability harder to find rather than totally removing it. Although the resulting code satisfies the auditor's surface-level requirements, it leaves the underlying vulnerability open to attack.

Exploiting the Audit Pipeline

This paradox can be used by an attacker to disable the system's protections if they are able to pollute the system's runtime memory or carry out a small prompt injection:

- **Auditor Poisoning:** An attacker can reduce the auditing agent's security baseline by changing its context window or system prompt. The malicious, backdoored kernel modules created by the main coding engine will then be carelessly approved by the corrupted verification agent.
- **Context Exhaustion:** The generating agent can be tricked by attackers into producing enormous, excessively complex, and syntactically thick code structures. The auditing agent may undergo memory dilution or time out as a result of this overflowing its context window. In order to avoid a system crash, the system then fails open and approves unverified code blocks.

In the end, a system cannot use the same layer of logic that produced a design problem to fix it. The ultimate arbiter of its own security cannot be an automated model. Verification needs to be firmly linked to absolute mathematical laws and separated from neural inferences in order to stop this cycle.

FORMAL VERIFICATION AS AN ABSOLUTE GATEKEEPER

Self-programming computing platforms must abandon semantic analysis and implement rigorous Formal Verification in order to remove the cognitive blind spots and vulnerabilities brought about by the "Who Watches the Watcher?" contradiction.

To safeguard a running kernel, we cannot rely on text-based forecasts, tokenized evaluations, or probabilistic AI estimates. Rather, before an agent compiles, each block of code it synthesizes must be regarded as a mathematical assertion and shown to be secure by deductive reasoning.

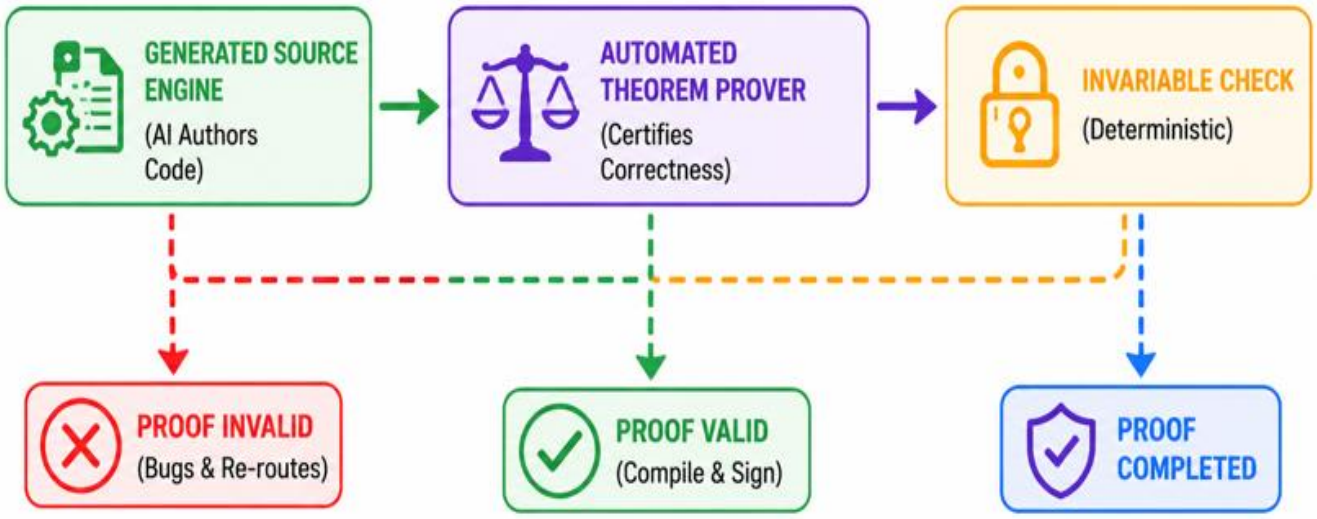

Figure 5: Mathematical Assertion. *Source: Cramer, M., & McIntyre, L. (2025). Verifying LLM-generated code in the context of software verification with Ada/SPARK. arXiv.* https://arxiv.org/abs/2502.07728

Converting Source Code into Mathematical Statements

Formal verification evaluates system features using automated theorem provers, such as interactive proof assistants (e.g., Coq, Lean) or Satisfiability Modulo Theories (SMT) solvers (e.g., Z3). Software routines are treated as mathematical expressions.

Every time an autonomous coding agent creates a new runtime module, it needs to produce two different results:

1. **The Target Code:** The executable source files that are meant to be executed.

2. **The Proof Artifact:** A mathematically organized demonstration of the code's strict adherence to predetermined security invariants.

Enforcing Invariant Security Specifications

The underlying hardware layer of the operating system must impose fundamental, unalterable rules that cannot be changed, circumvented, or rewritten by an automated process.

A deterministic checking engine is hardcoded with these specifications:

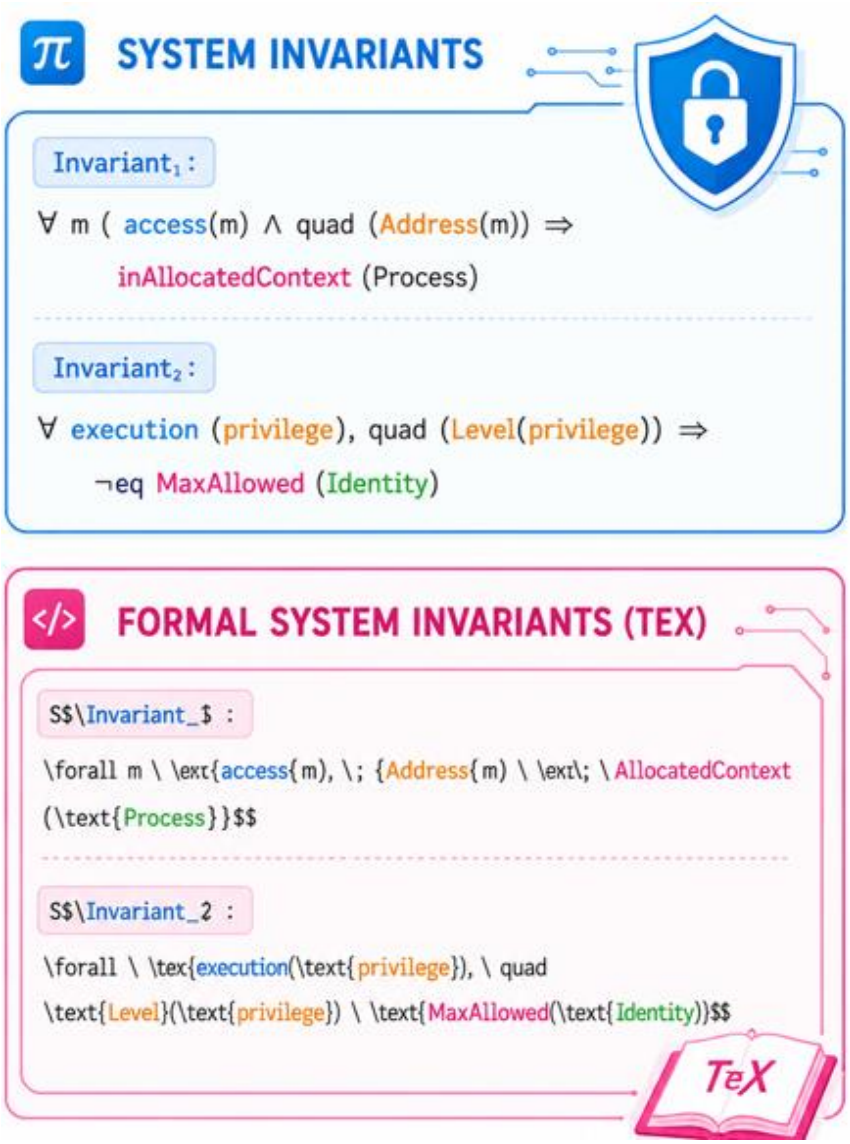

Figure 6: System Invariants & Formal System Invariants. *Source: Diekmann, C., Posselt, S. A., Niedermayer, H., Kinkelin, H., Hanka, O., & Carle, G. (2016). Verifying security policies using host attributes. arXiv. https://arxiv.org/abs/1604.00204*

- **Memory Isolation Invariants:** The proof must show mathematically that out-of-bounds memory accesses and writes cannot be performed by the freshly created code. This entirely removes memory leaks, use-after-free vulnerabilities, and buffer overflows prior to compilation.
- **Control-Flow Integrity Invariants:** In order to prevent return-oriented programming (ROP) vulnerabilities, the proof must ensure that system commands cannot stray from a predefined control-flow graph.

The Absolute Gatekeeper

The code is instantly destroyed if the deterministic checking engine is not satisfied with the mathematical proof. How highly the auditing model assessed the safety of the code is entirely ignored by the system. This configuration breaks the recursive cycle completely by ensuring that the

unpredictable, fluid character of automated code production is controlled by the absolute certainty of formal logic.

REAL-WORLD PRECEDENTS AND EVIDENCE

Automated coding tools' security flaws are no longer only hypothetical issues for the far future. They are clearly obvious in known vulnerabilities and previous industrial failures (National Institute of Standards and Technology. (2025). CVE-2025-59536 detail. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2025-59536).

Vulnerabilities in Developer Toolsets (CVE-2025-59536 and CVE-2026-21852)

Security researchers discovered serious flaws in automated developer frameworks, particularly the Claude Code toolset, around the beginning of 2026:

- **CVE-2025-59536 (CVSS 8.7**): This vulnerability made it possible for a hacked repository to run illegal terminal commands as soon as an automated agent started up inside the project folder.
- **CVE-2026-21852:** This flaw made it possible for hackers to steal private API keys from a developer's local environment in secret by using malicious configuration files.

These real-world examples demonstrate how autonomous tools soon become the main targets of supply chain attacks when they are granted root file-system access and command execution rights to optimize code bases.

The Lovable Platform Exploit (CVE-2025-48757)

- **The Vulnerability:** It was discovered that Lovable, an automated development platform, was creating backend database structures without implementing Row-Level Security (RLS) safeguards.
- **The Impact:** More than 170 active business apps were left vulnerable by this automated blunder, making private backend data accessible to anybody on the public internet. This event highlights the practical risk of putting automatically generated systems into production without independent, hardcoded verification.

Automated Credential Leaks (Moltbook and The Tea App)

Before sharing database contents, the live API interfaces created by the automated app-building engines Moltbook and The Tea App neglected to verify user access privileges. Due to a structural error, 1.5 million user authentication tokens were leaked by Moltbook, and 1.1 million private chat records were unintentionally made public by The Tea App. These errors occurred as a result of the generation engines' complete concentration on functional outputs and application features, neglecting the fundamental security checks that a human engineer would incorporate.

BUILDING THE AUTONOMOUS DEFENSIVE LAYER (ADL)

Self-programming operating systems need a security architecture that functions directly within the execution pipeline since they change in real time at computational speed. An instant exploitation window is created by waiting for out-of-band vulnerability scans or recurring manual reviews.

We suggest integrating an inline Autonomous Defensive Layer (ADL) directly into the kernel runtime to stop compromised or insecure synthesis code from endangering system integrity.

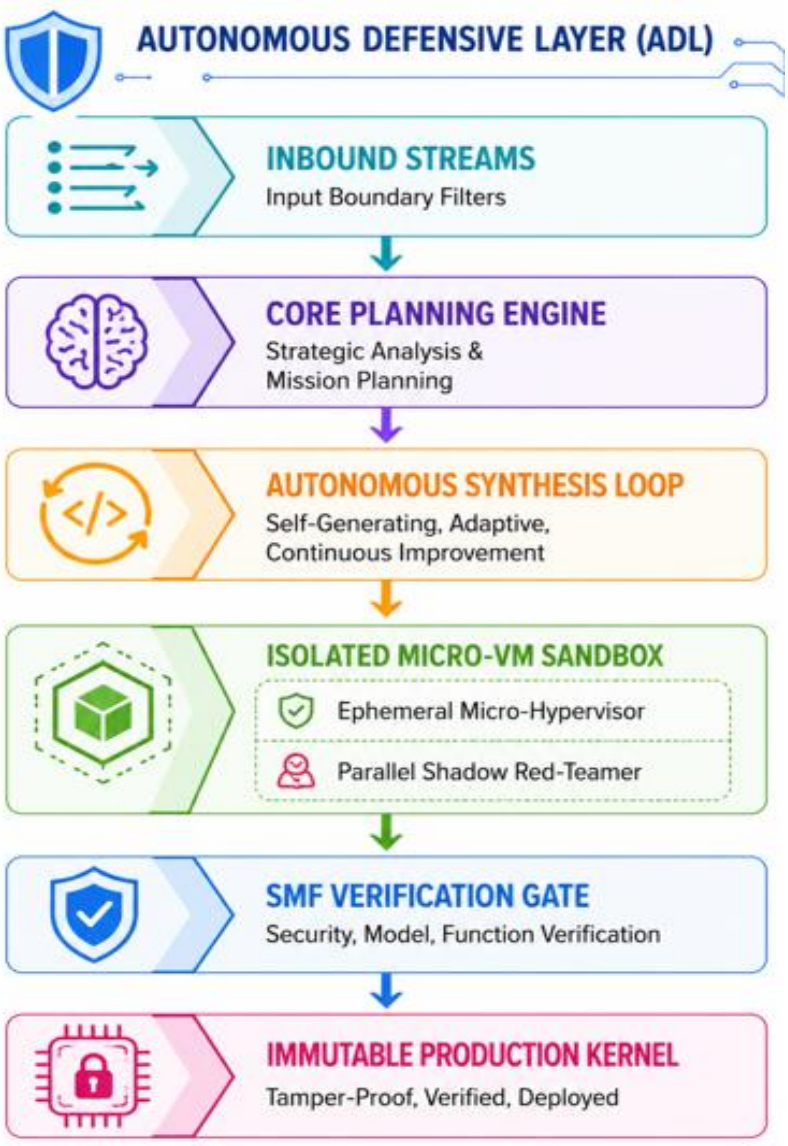

Figure 7: Autonomous Defensive Layer. *Source: Nasir, N., & Al Hamadi, H. (2025). Towards the neuromorphic Cyber-Twin: An architecture for cognitive defense in digital twin ecosystems. Frontiers in Big Data, 8, 1659757. https://doi.org/10.3389/fdata.2025.1659757*

Micro-Virtualization and Ephemeral Sandboxing

The complete restriction of physical execution rights to untested code is the fundamental tenet of the ADL. By default, all runtime modules, device drivers, and kernel patches produced by automated sub-agents are considered active threat vectors.

- **Micro-VM Isolation:** All newly created binaries are routed by the ADL straight into separate, hardware-accelerated micro-virtual machines (Micro-VMs) under the control of a high-performance, low-power hypervisor.

- **Resource Stripping:** Network stacks, persistent storage directories, and physical memory management units (MMUs) are not directly accessible in these Micro-VM sandboxes. Only in a simulated environment where its hardware transactions are recorded and examined can the code

operate. The entire Micro-VM instance is immediately removed from volatile memory if the module tries to call an unmapped system interrupt or execute an unauthorized out-of-bounds reference.

The Parallel Shadow Red-Teamer

The ADL operates a parallel Shadow Reasoning Agent in a totally separated memory space while the main code generation cycle concentrates on feature completeness and performance optimization.

- **Continuous Stress-Testing:** The shadow agent functions only as an adversary. Targeting OWASP Top 10 vulnerabilities, integer overflows, and race situations, it applies automated mutation fuzzing, semantic prompt-injection testing, and known vulnerability pattern matching to the newly created source code inside the sandbox.

- **Differential Analysis:** The shadow verifier can identify edge-case defects that the primary generative agent missed by contrasting the predicted logical state with the behavior found during adversarial fuzzing. The shadow agent logs the behavioral failure, flags the asset, and prevents deployment if a module fails any defensive test.

Hardening System Endpoints via Strict Schemas

Strict validation requirements must be used to harden the platform's physical communication boundaries in order to stop malevolent external actors from misusing the system's autonomous capabilities:

- **Mandated mTLS Architecture:** Every data transit pathway that connects peripheral devices, remote orchestration nodes, and local hardware sensors must use ephemeral, short-lived cryptographic keys to implement Mutual Transport Layer Security (mTLS). By doing this, man-in-the-middle attacks that try to insert malicious code blocks into streaming data are eliminated.

- **Input Boundary Filtering:** Deterministic, non-neural tokenizers are immediately deployed by the ADL at the input gateway of the planning engine. Strict schema definitions are used to parse each natural language instruction or API payload. Before it may affect the code synthesis loop, any input that contains character sequences intended to cause model hallucinations, hidden system directives, or recursive execution prompts is promptly removed.

REGULATORY, GOVERNANCE, AND COMPLIANCE FRAMEWORKS

Strict organizational boundaries and contemporary compliance standards are necessary to manage the functioning of self-programming architectures as they progress from laboratory prototypes to operational control rooms, financial centers, and defense systems. It is a disastrous compliance failure to rely on an unmonitored, self-evolving loop to manage business resources without official administrative monitoring.

Modernizing Risk Management Frameworks

To handle dynamic, self-evolving operating setups, current operational frameworks – like the National Institute of Standards and Technology's AI Risk Management Framework (NIST AI-RMF) – must be expanded. When a system's software modifies its operating profile thousands of times each hour, traditional point-in-time compliance assessments are completely useless.

- **Continuous Telemetry Logging:** The integration of immutable, cryptographically sealed ledger systems that document each iteration of synthesis code, the identity of the generating sub-

agent, the verification proof artifacts, and the operational rationale for the update must be mandated by regulatory agencies.

- **Absolute Corporate Liability:** Legal and corporate compliance policies must specifically state that enterprise operators are still fully accountable for security breaches and data exposure resulting from automated code loops under data protection regulations (such the CCPA or GDPR). Legally speaking, the defense that "the AI model wrote an unpredictable vulnerability" is invalid; if an organization uses a self-programming system, it is fully responsible for the synthesized outputs of that system.

Supply Chain Accountability and Shadow AI Mitigation

Security operations centers (SOCs) must impose stringent asset control boundaries in order to preserve visibility into automated business infrastructure:

- **Time-Scoped, Least-Privilege Credentials:** Permanent, high-level administration credentials should never be given to automated sub-agents. All generative engines must use short-lived tokens that are limited to localized namespaces and expire in a matter of minutes, according to the ADL.

- **Mitigating Shadow AI Loops:** Automated network inventory scanners that track unauthorized runtime synthesis must be used by organizations. To stop unmonitored "Shadow AI" applications from growing the corporate attack surface, any autonomous framework discovered creating code paths, opening local network ports, or setting up undocumented API endpoints outside the verified ADL pipeline must be automatically isolated and terminated.

RESEARCH GAPS

Although software development has undergone a fundamental transformation due to the rapid emergence of self-programming and agentic AI systems, the cybersecurity literature currently in publication still primarily focuses on traditional software engineering, static code analysis, secure coding practices, and AI-assisted code generation rather than fully autonomous self-evolving runtime environments.

There is still a dearth of research on cybersecurity architectures that can defend continuously self-modifying operating systems, despite earlier studies looking at vulnerabilities related to AI-generated code, prompt injection, software supply chain attacks, and AI risk management. Recursive verification failures ("Who Watches the Watcher?"), rigorous mathematical verification of dynamically synthesized code, and integrated runtime defensive mechanisms created especially for autonomous software synthesis have all received little attention.

In order to close these gaps and offer a complete security framework for next self-programming computing environments, this study suggests an Autonomous Defensive Layer (ADL) backed by deterministic formal verification, micro-virtualization, shadow reasoning, and continuous runtime validation.

RESEARCH LIMITATIONS

The suggested Autonomous Defensive Layer (ADL) has not yet been deployed or empirically proven within a real-world autonomous operating system because this research is mostly conceptual and architectural in nature. Instead of using experimental deployment or extensive benchmark testing,

the threat models, architectural designs, and security measures are synthesized from recent research, publicly disclosed vulnerabilities, and theoretical analysis.

Additionally, because generative AI, autonomous software engineering, and cybersecurity are all developing quickly, new technologies, attack methods, and legal requirements may present issues that our approach does not adequately address. Therefore, before the suggested architecture can be deemed operationally mature, more experimental study, prototype implementation, performance evaluation, scalability testing, and comparative validation against current autonomous security frameworks are needed.

CONCLUSIONS

Software engineering has undergone a permanent evolutionary leap with the shift from static, human-engineered code bases to fluid, self-programming agentic runtimes. These autonomous systems attain previously unheard-of levels of processing efficiency, optimization, and hardware adaptability by removing the bottlenecks of manual text production, peer review delays, and strict compilation cycles. However, the modern attack surface is greatly increased when enterprise infrastructure is built on top of self-evolving code loops.

This research has demonstrated that using probabilistic reasoning models to audit their own synthesized outputs results in a severe recursive loop failure route with shared cognitive blind spots and systemic vulnerability amplification. Traditional perimeter-focused, signature-dependent protection paradigms must be completely abandoned in order to secure this next generation of computers.

Real-time, integrated security measures must be implemented in order to move forward. We can guarantee that any automated software modification fulfils absolute mathematical safety invariants prior to execution by separating code verification from neural inferences and directly connecting it to the deterministic logic of Formal Verification. Organizations can safely utilize the enormous power of self-programming computing environments while preserving systemic predictability, cryptographic integrity, and complete human oversight when paired with the micro-virtualization isolation and parallel adversarial fuzzing of an Autonomous Defensive Layer.

Declaration by Authors
Ethical Approval: Approved
Acknowledgement: None
Source of Funding: None
Conflict of Interest: The authors declare no conflict of interests.

REFERENCES

Aghakhani, H., Dai, W., Manoel, A., Fernandes, X., Kharkar, A., Kruegel, C., Vigna, G., Evans, D., Zorn, B., & Sim, R. (2023). TrojanPuzzle: Covertly poisoning code-suggestion models. arXiv. https://arxiv.org/abs/2301.02344

Apiiro. (2025, September 4). 4x velocity, 10x vulnerabilities: AI coding assistants are shipping more risks. https://apiiro.com/blog/4x-velocity-10x-vulnerabilities-ai-coding-assistants-are-shipping-more-risks/

Autio, C., Schwartz, R., Dunietz, J., Jain, S., Stanley, M., Tabassi, E., Hall, P., & Roberts, K. (2024). Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence profile (NIST AI 600-1). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.AI.600-1

Bhatt, M., Chennabasappa, S., Li, Y., Nikolaidis, C., Song, D., Wan, S., Ahmad, F., Aschermann, C., Chen, Y., Kapil, D., Molnar, D., Whitman, S., & Saxe, J. (2024). CyberSecEval 2: A wide-ranging cybersecurity evaluation suite for large language models. arXiv. https://arxiv.org/abs/2404.13161

Cramer, M., & McIntyre, L. (2025). Verifying LLM-generated code in the context of software verification with Ada/SPARK. arXiv. https://arxiv.org/abs/2502.07728

Cycode Team. (2026). Top AI security vulnerabilities to watch out for in 2026. Cycode.

Diekmann, C., Posselt, S. A., Niedermayer, H., Kinkelin, H., Hanka, O., & Carle, G. (2016). Verifying security policies using host attributes. arXiv. https://arxiv.org/abs/1604.00204

Escape. (2025). Methodology: How we discovered 2K+ vulnerabilities in apps built with vibe coding. Escape.

Fahim, M. A. I., Adebayo, O., & Ferrari, A. (2026). Software self-extension with SelfEvolve: An agentic architecture for runtime code generation. In Proceedings of the 21st International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '26). Association for Computing Machinery. https://doi.org/10.1145/3788550.3794870

Google. (2023). Introducing Google's Secure AI Framework. Google.

Nasir, N., & Al Hamadi, H. (2025). Towards the neuromorphic Cyber-Twin: An architecture for cognitive defense in digital twin ecosystems. Frontiers in Big Data, 8, 1659757. https://doi.org/10.3389/fdata.2025.1659757

National Cyber Security Centre. (2023). Guidelines for secure AI system development. NCSC.

National Institute of Standards and Technology. (2023). Artificial Intelligence Risk Management Framework (AI RMF 1.0) (NIST AI 100-1). U.S. Department of Commerce. https://doi.org/10.6028/NIST.AI.100-1

National Institute of Standards and Technology. (2025). CVE-2025-48757 detail. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2025-48757

National Institute of Standards and Technology. (2025). CVE-2025-59536 detail. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2025-59536

National Institute of Standards and Technology. (2026). CVE-2026-21852 detail. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2026-21852

OWASP Foundation. (2024). OWASP Top 10 for LLM applications 2025. OWASP GenAI Security Project.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE. https://doi.org/10.1109/SP46214.2022.9833571

Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2022). Do users write more insecure code with AI assistants? arXiv. https://arxiv.org/abs/2211.03622

Siddiq, M. L., & Santos, J. C. S. (2022). SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security. Association for Computing Machinery. https://doi.org/10.1145/3549035.3561184

Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2024). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. arXiv. https://arxiv.org/abs/2406.10279

Tony, C., Mutas, M., Díaz Ferreyra, N. E., & Scandariato, R. (2023). LLMSecEval: A dataset of natural language prompts for security evaluations. arXiv. https://arxiv.org/abs/2303.09384

Veracode. (2025). 2025 GenAI code security report. Veracode.

Veracode. (2026). Spring 2026 GenAI code security update. Veracode.

Wan, S., Nikolaidis, C., Song, D., Molnar, D., Crnkovich, J., Grace, J., Bhatt, M., Chennabasappa, S., Whitman, S., Ding, S., Ionescu, V., Li, Y., & Saxe, J. (2024). CyberSecEval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. arXiv. https://arxiv.org/abs/2408.01605

Yang, Y., Nie, Y., Wang, Z., Tang, Y., Guo, W., Li, B., & Song, D. (2024). SecCodePLT: A unified platform for evaluating the security of code GenAI. arXiv. https://arxiv.org/abs/2410.11096

Zietsman, C. (2026). The specification as quality gate: Three hypotheses on AI-assisted code review. arXiv. https://arxiv.org/abs/2603.25773

How to cite this article:
Anjum, M. A. (2026). Cybersecurity for Self-Programming Systems. *International Journal of Digital Research, 2*(3), 34–49. https://doi.org/10.63711/ijdr.net20260303

*****