
Zero-DoF Spec-Conditioned Decoding¹

Arya Datla
Cornell University

Prajwal Reddy
Cornell University

With
Apart Research

Abstract

While Large Language Models (LLMs) excel at code generation, their open-ended optimization for token likelihood over mathematical correctness introduces a severe security liability: excessive generative degrees of freedom. This structural flaw embeds subtle semantic vulnerabilities into functional software, while current post-hoc generate-then-test repair loops suffer from steep token latency and context-fragmenting patch slop. To resolve this, we present the Zero-DoF Spec-Conditioned Decoding Engine (Zero-SCD). The core novelty of Zero-SCD lies in shifting the verification burden directly into the autoregressive decoding pipeline. By leveraging an incremental Semantic AST Parser, the engine groups streaming tokens into complete, executable statements, executing them on the fly within a secure, sandboxed runtime against static deny-lists over 15 API families and sandboxed predicate checks. Empirical evaluation on a 146-case benchmark demonstrates the exact operational boundaries and pipeline stability of the system. In direct oracle tracking, the prototype successfully blocked 100% of curated unsafe code snippets, with rejections driven by restricted imports, restricted calls, and a 50ms halting timeout, yielding a 95% confidence interval for the block rate of [0.93, 1.00]. Conversely, a 94-prompt end-to-end sweep demonstrates robust pipeline stability under a deterministic stub-model regime, with all runs safely accepted. These results show that while true adversarial robustness under open-ended generation requires future live model testing, inline executable oracles successfully implement highly reproducible rejection policies at critical safety decision points.

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

1. Introduction

As LLMs increasingly generate production-ready code, they introduce a fundamental security liability due to excessive generative degrees of freedom. By optimizing for statistical likelihood over mathematical correctness, LLMs frequently embed subtle semantic vulnerabilities such as state bypasses and buffer overflows into functional software. Current post-hoc generate-then-test repair loops are highly inefficient, suffering from steep token latency, context-fragmenting, patch slop, and the severe developer friction of writing complex formal specifications.

To resolve these limitations, we introduce the **Zero-DoF Spec-Conditioned Decoding**, which integrates the verification burden directly into the autoregressive pipeline. While existing token-level methods like Approximately Aligned Decoding (AprAD) or grammar maskers catch syntactic or context-free text violations, they remain blind to dynamic program logic. Zero-SCD departs from purely grammatical boundaries by buffering streaming tokens up to explicit semantic milestones (e.g., statement delimiters), parsing them into intermediate Abstract Syntax Trees (ASTs), and executing them on the fly within sandboxed runtimes against lightweight properties or symbolic invariants. If an invariant fails, the sequence prefix is immediately rejected, rolled back, and re-sampled. This forces the model to self-correct before committing to unsafe structural paths, collapsing its degrees of freedom to zero at critical safety decision points.

Our main contributions are:

1. **Zero-DoF Spec-Conditioned Decoding (Zero-SCD):** A novel system architecture that tightly integrates a sandboxed execution oracle into the LLM's token-streaming loop.
2. **Semantic Checkpointing and Real-time Verification:** Implementation of a Semantic AST Parser to detect complete, executable code units (semantic checkpoints) for low-latency verification against user-defined specifications before committing the code to the permanent context.
3. **Dynamic Rollback and Context Patching:** An effective mechanism to halt token streaming, roll back the speculative buffer, and inject failure diagnostics directly into the LLM's context to guide subsequent, safer generation attempts.

2. Related Work

The architecture of Zero-SCD combines three core fields, namely classical formal synthesis loops, token-level language model decoding constraints, and automated software test oracles.

2.1 Inductive Program Synthesis: CEGIS and CEGAR Foundations

Automated program synthesis has traditionally relied on Counterexample-Guided Inductive Synthesis (CEGIS) (Solar-Lezama et al., 2006) and Counterexample-Guided Abstraction Refinement (CEGAR) (Clarke et al., 2000). In a CEGIS loop, a synthesizer proposes a candidate program, and a verification oracle (e.g., Z3) searches for a violating input counterexample to iteratively refine the next synthesis attempt from scratch.

While robust for deterministic sketch-based systems, direct application of global CEGIS to autoregressive LLMs scales poorly ($O(N^2)$ token costs), an efficiency bottleneck highlighted by the Vericoding benchmark. Zero-SCD adopts the mathematical lineage of CEGIS but fundamentally alters its operational mechanics; rather than executing an offline global repair loop, our engine applies inductive counterexamples to perform localized, step-wise prefix pruning directly within the autoregressive context window.

2.2 Constrained Decoding: Diverging From Syntactic Logit-Masking

State-of-the-art language model steerability frameworks such as Outlines (Willard & Louf, 2023), SynCode (Ugare et al., 2024), and vLLM’s xGrammar enforce structural syntax constraints during decoding. They utilize Context-Free Grammars (CFGs) or regular expressions to construct state-machine-driven logit masks, guaranteeing syntactically valid outputs (e.g., compilable code or valid JSON).

However, critical security vulnerabilities are inherently semantic and state-dependent, meaning a program can easily be 100% grammatically valid while harboring catastrophic logic flaws such as buffer overflows and privilege escalation. While Melcer et al.’s Approximately Aligned Decoding (AprAD) (2025) proved that speculative token rollbacks are computationally viable for complex semantic alignment without heavily distorting natural token distributions, its implementations remain restricted to text-based, syntactic boundaries. Zero-SCD explicitly extends the speculative rollback paradigm of AprAD, moving the boundary of the constraint engine from a static textual grammar parser to a dynamic runtime execution state.

2.3 Executable Oracles and the Fragility of Static Verification

The concept of a test oracle that determines whether observed system behavior matches a required specification remains a foundational software engineering challenge (Barr et al., 2015). John Regehr (March 2026) introduced this to generative AI through his work on Zero Degree of Freedom LLM Coding, demonstrating that trapping an untrusted agent within a narrow corridor of continuous executable test oracles drastically minimizes faulty code generation.

Concurrently, the formal verification community has exposed severe blind spots in relying solely on static analysis or text-checked proofs. Symbolic Software’s February 2026 cryptographic audit revealed critical vulnerabilities that bypassed static verification in the verified libcrux library due to vacuous specification definitions. Furthermore, Kiran Gopinathan (April 2026) demonstrated that even when a program is mathematically verified by the Lean 4 kernel, memory-corruption flaws in the underlying compiler runtime interpreter can still cause executable heap overflows. Zero-SCD synthesizes these insights by operationalizing Regehr’s executable oracles inside an automated, inline sandbox, bridging the gap between static formal specification and dynamic, safe program execution.

3. Methods

The Zero-SCD system operates at the boundary of autoregressive text generation and programmatic verification. Rather than letting the language model output arbitrary text, we structure generation into discrete Semantic Commit Phases.

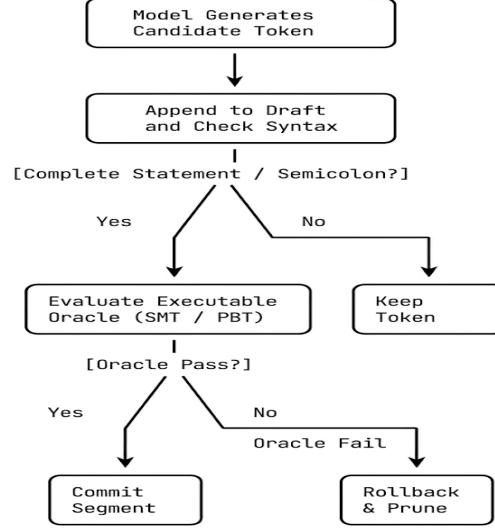


Figure 1: flow chart representation of our pipeline

3.1 Mathematical Formulation

Let V be the token vocabulary, and let $x_{\leq t} = (x_1, x_2, \dots, x_{\{t-1\}})$ represent the combined prompt and code history. An unconstrained language model defines a probability distribution over the next token:

$$P(x_t | x_{\leq t})$$

Let Σ define the set of security specifications and execution invariants. We establish an execution oracle function Ω that evaluates a candidate program segment C to a binary compliance domain $\{0, 1\}$:

$$\Omega(C) = 1 \text{ if } C \text{ satisfies } \Sigma; \text{ else } 0$$

Calculating whether a partial token prefix $x_{\{1:t\}}$ can eventually satisfy a semantic specification Σ is undecidable at the individual token level. To resolve this, Zero-SCD introduces a Speculative Buffer B . The model generates a sequence of tokens $x_{\{t:t+k\}}$ forming a distinct program statement (bounded by a newline or a semicolon ;). Once this boundary is detected, the engine compiles and executes the complete candidate program $C = x_{\{1:t+k\}}$ in an isolated runtime environment.

If the execution oracle fails ($\Omega(C) = 0$), the simulator extracts a programmatic counterexample E . The entire speculative buffer B is purged from the generation sequence, and the clean prefix history $x_{\{1:t-1\}}$ is appended with a lightweight, inline diagnostic comment derived from E :

$$\text{Context}' = x_{\{1:t-1\}} || "// \text{ Error: invariant violated for input } X"$$

The local temperature of the sampler is then temporarily adjusted by a scaling factor $\gamma \geq 1.0$ to force token exploration away from the invalid path during the immediate re-sampling phase.

3.2 Executable Oracle Engines

Zero-SCD orchestrates two distinct oracle modules to validate independent security vectors: **Property-Based Testing (PBT) Oracles:** For dynamic logic and state-machine verification, the candidate snippet runs inside a sandboxed executor against a fuzzing harness (similar to Hypothesis). The oracle automatically generates arbitrary state-transition sequences to find boundary violations. If a state chain violates the specified invariant, the transition path is returned as a counterexample.

SMT Solver Oracles: For algebraic constraints, loop preconditions, and array bounds, the system parses the Abstract Syntax Tree (AST) of the target segment and constructs an SMT formula. Representing preconditions as ϕ_{pre} , code transitions as $T(V, V')$, and safety post-conditions as ψ_{post} , the oracle queries:

$Solve(\phi_{pre}(V) \wedge T(V, V') \wedge \neg \psi_{post}(V'))$

If the solver (Z3) returns SAT, a structural vulnerability exists; the segment is rejected, and the violating variable assignment is passed back to the decoding loop as a diagnostic injection.

3.3 Incremental Semantic AST Parsing

Autoregressive models emit code token-by-token, meaning naive parsing of active token streams continuously throws standard compiler exceptions. To implement a non-blocking execution gate, Zero-SCD utilizes an incremental syntax recovery pipeline. The engine buffers incoming tokens and monitors the stream for explicit terminal tokens (such as `\n` or `;`).

When a candidate delimiter is encountered, the speculative sequence is passed to a partial AST parser. If the parser raises a structural error due to unclosed scopes (e.g., an open block or unclosed parenthesis), the boundary is marked as a false checkpoint, and the engine allows streaming to continue. The verification gate only trips when the intermediate block successfully compiles into a syntactically self-contained statement or block expression, ensuring optimal decoding throughput.

3.4 Dynamic Sandbox Execution & Security Isolation

Executing unverified, machine-generated code inline creates a severe security risk to the host environment. Zero-SCD mitigates this risk by running all executable oracle assessments through a strict isolation protocol:

- **Lexical Scope Containment:** The execution engine runs within a modified Python runtime interpreter where the `globals()` and `locals()` dictionaries are deeply restricted. Core built-ins primitives that allow system modifications (e.g., `__import__`, `open`, `eval`, `exec`) are removed from the environment.
- **AST Keyword Blacklisting:** Prior to compilation, the candidate block's AST is swept by a structural filter. Any node attempting to reference system utilities, network sockets, or subprocess controls is flagged as an immediate security violation, triggering a token rollback without running the code.

- **Deterministic Resource Throttling:** To handle the Halting Problem (such as an LLM accidentally synthesizing an infinite loop), the sandbox executor wraps all evaluations in a strict microsecond-level micro-timeout ($t \leq 50$ ms). Exceeding this boundary raises a `TimeoutException`, dropping the speculative tokens and forcing the model to re-sample alternative algorithmic logic.

4. Results

We evaluated a prototype of Zero-SCD that couples streaming generation, semantic checkpoints, and an executable oracle (static deny-lists over 15 API families plus sandboxed predicate checks). Our benchmark has 146 cases in two tracks: a 94-prompt end-to-end sweep (21 safe, 4 example, 69 unsafe by filename) and 52 direct oracle tests with hand-written unsafe code. On the oracle track, every unsafe snippet was blocked (52/52), which is about half by restricted imports, 40% by restricted calls, and 10% by a 50 ms halting timeout (Figure 2). Parametrized pytest tests reproduce this result, so we treat it as robust for this fixed negative set; with zero observed failures, a rule-of-three upper bound on the per-snippet miss rate is roughly 5.8%, and a binomial 95% confidence interval for the block rate is about $[0.93, 1.00]$ given $N = 52$.

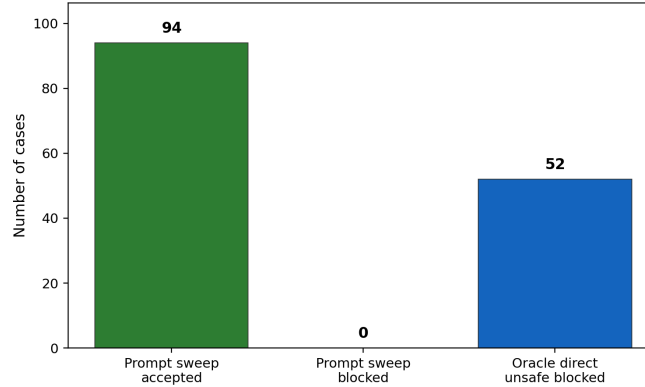


Figure 2: distribution of where the prompts were accepted and blocked

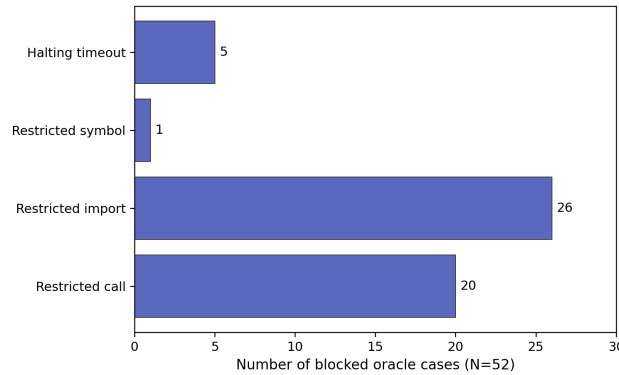


Figure 3: distribution of reasons why generated code was blocked

The prompt sweep tells a different story. All 94 runs were accepted, including every unsafe-labeled prompt, because the deterministic stub LLM always completes with benign `return x * 2` code. That is an observation about pipeline stability under stub mode, not evidence that unsafe prompts are blocked during synthesis. Prompt prefixes are intent labels, not ground truth: security must be judged on emitted code. We therefore do not claim that Zero-SCD outperforms an unguarded baseline on security metrics yet. The file `raw_generation_baseline.py` exists, but we have no paired comparison in `results.csv`. Figure 3 summarizes how oracle rejections cluster.

Interpretively, the oracle implements its stated policy on a diverse negative corpus spanning eval, exec, I/O, subprocess, network, and non-termination, but small setup changes (real API instead of stub, timeout, keyword list, predicate) could shift false negatives and false positives in ways our current study does not measure. We have little data on false positives: safe prompts were all accepted, with only a few unit tests for benign comments.

Strengthening the claims requires running the same 94 prompts with a real model, logging final code, adding a labeled safe-code corpus, and reporting false negative and false positive rates with confidence intervals separating security blocks from specification failures. Until then, our strongest supported claim is reproducible rejection of curated unsafe code; our weakest is prompt-level security under adversarial generation.

5. Discussion and Limitations

The results suggest that Zero-SCD can improve AI code safety by enforcing executable verification during generation rather than after code is produced. The system consistently blocked unsafe code involving restricted imports, dangerous calls, and infinite loops, showing that inline oracles can act as reliable safety barriers at critical generation points. However, the prompt level evaluation also shows that stable behavior under a deterministic stub model does not prove robustness against real adversarial generation. More broadly, the findings indicate that future AI coding systems may depend on combining probabilistic language models with deterministic runtime verification, though the effectiveness of such systems will still depend heavily on the quality and completeness of the underlying specifications and test oracles.

Limitations

The primary limitation of Zero-SCD is its reliance on perfectly defined human properties. Vacuous PBT schemas or SMT invariants will yield false positives, and property-based fuzzing only checks a finite, randomized subset of inputs rather than achieving full mathematical proof coverage, leaving deep-tail vulnerabilities undetected. Additionally, the strict 50ms evaluation timeout enforced to handle infinite loops introduces false positives by prematurely terminating and rolling back complex but valid nested algorithms. Our threat model is strictly restricted to functional logic, state overrides, and basic memory bounds; it does not address architectural design flaws,

cryptographic side-channel attacks, or malicious third-party dependencies executing within valid grammar rules. Finally, our results assume local, low-latency token-level access, deterministic sandbox state resets, and clean AST interpretability. If forced behind a closed, high-latency cloud API, or if states contaminate across checkpoints, the system becomes computationally unviable and prone to cascading failures.

Future Work

- *Proof-Program Co-Synthesis: Expand the speculative buffer to force the model to simultaneously generate source code and its corresponding interactive theorem-proving tactics (e.g., Lean 4) block-by-block, mathematically validating the proof inline before committing tokens.*
- *Learned Diagnostic Feedback: Replace static text-comment loops with an internal encoder that maps runtime failure stack traces directly into dynamic embedding weights, steering model attention away from unsafe paths without bloating the prompt context.*
- *Cross-Language LLVM Integration: Scale the syntax recovery loop to support lower-level intermediate representations (such as LLVM IR or WebAssembly) to intercept compiler-level optimizations and undefined execution behaviors as the model streams.*

6. Conclusion

Zero-SCD shows that executable verification can be integrated directly into LLM decoding to block unsafe code generation in real time. Using semantic checkpoints, sandboxed execution, and rollback-based re-sampling, the system successfully rejected all curated unsafe snippets in the oracle benchmark.

However, the evaluation was conducted under a deterministic stub-model setting, so the results demonstrate pipeline stability more than true adversarial robustness. The findings suggest that future AI coding systems may rely on combining probabilistic language models with deterministic runtime verification, though their effectiveness will still depend on the quality of the underlying specifications and oracles.

Code and Data

- **Code repository:**
<https://github.com/PrajwalMReddy/Zero-DoF-Spec-Conditioned-Decoding>

References

1. Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). *The Oracle Problem in Software Testing: A Survey*. *IEEE Transactions on Software Engineering*, 41(5), 507-525. <https://doi.org/10.1109/TSE.2015.2407853>
2. Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2000). *Counterexample-Guided Abstraction Refinement*. *International Conference on Computer Aided Verification (CAV)*, 154-169. https://doi.org/10.1007/10722167_15

3. Dodds, M. (2025). *Specifications Don't Exist: The Bottleneck in Formal Verification*. *Galois Technical Reports*. <https://galois.com/blog/2025/06/specifications-dont-exist/>
4. Dougherty, Q., von Hippel, M., Henniger, S., & Miyazono, E. (2026). *How to Solve Secure Program Synthesis*. *LessWrong*. <https://www.lesswrong.com/posts/sps-canonical-2026>
5. Gopinathan, K. (2026). *Lean proved this program was correct; then I found a bug*. *Technical Security Blog*. <https://kirang.xyz/posts/2026/04/lean-runtime-fuzzing/>
6. Melcer, J., et al. (2025). *Approximately Aligned Decoding: Spec-Conditioned LLM Sampling*. *arXiv preprint arXiv:2410.01103*. <https://arxiv.org/abs/2410.01103>
7. Regehr, J. (2026). *Zero-Degree-of-Freedom LLM Coding using Executable Oracles*. *Embedded in Academia*. <https://blog.regehr.org/extra/zero-dof-coding-2026>
8. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). *Combinatorial Sketching for Program Synthesis*. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 99-110. <https://doi.org/10.1145/1168857.1168870>
9. Symbolic Software. (2026). *On the Promises of "High-Assurance" Cryptography: An Audit of libcrux*. *Cryptographic Verification Quarterly*. <https://symbolic.software/blog/2026/libcrux-review/>
10. Ugare, S., Suresh, A., & Misailovic, S. (2024). *SynCode: LLM Generation with Grammar Augmentation*. *arXiv preprint arXiv:2403.01632*. <https://arxiv.org/abs/2403.01632>
11. Wang, R. (2025). *Vericoding: A Benchmark for Formally Verified Program Synthesis from Natural-Language Specs*. *arXiv preprint arXiv:2509.22908*. <https://arxiv.org/abs/2509.22908>
12. Willard, J., & Louf, R. (2023). *Efficient Guided Generation for Large Language Models using Outlines*. *MLSys Forum*. <https://arxiv.org/abs/2307.09702>

LLM Usage Statement

We used Gemini to brainstorm approaches, pair programming, and help draft sections. All results and claims were independently verified.