Agentic-Z3: Multi-Agent Framework for LLM-Guided SMT Solving

Lemeng Qi **Gaocheng Yang**

Abstract

Large Language Models (LLMs) excel at natural language understanding but struggle with formal reasoning tasks requiring logical precision. We present **Agentic-Z3**, a multi-agent framework that combines LLMs with Z3 SMT solver for autonomous constraint solving. Our system employs three specialized agents: an *Architect* for hierarchical planning, a *Worker* for type-aware code generation with test-time reinforcement learning (TTRL), and a *Coach* for diagnosis and skill crystallization. We address key failure modes through type-aware probing, soft reset mechanisms for stagnation, and evolutionary memory via a skill library. Evaluated on the TestEval path coverage benchmark (15 hard tasks), Agentic-Z3 achieves 76.8% exact path match and 98.2% execution correctness, outperforming zero-shot LLM prompting (+5.4%), AutoExe (+8.9%), and single-shot Z3 translation (+64.3%), while maintaining near-perfect test validity.

1 Introduction

Large Language Models have demonstrated remarkable capabilities in code generation and natural language reasoning (1; 6). However, formal reasoning tasks requiring *logical precision*—such as satisfiability modulo theories (SMT) constraint solving—remain challenging. LLMs trained on natural language and dynamically-typed Python are prone to type errors, timeouts, and lack the formal guarantees that constraint solvers provide.

Motivation. Consider the task of generating test inputs to cover a specific execution path in a Python function. This requires: (1) understanding the program semantics, (2) translating path conditions into formal constraints, (3) finding satisfying assignments, and (4) handling errors when translation or solving fails. Pure LLM prompting can reason about semantics but lacks formal grounding (3). Symbolic execution tools like AutoExe (2) provide formal reasoning but struggle with complex program constructs. Single-shot "text-to-Z3" translation is brittle and fails to recover from errors.

Key Challenges. We identify three recurring failure modes in LLM-SMT integration:

1. *Type error sensitivity*: Z3 is mathematically strict (Int vs Real, String vs Char), while LLMs generate dynamically-typed code
2. *Stagnation on timeouts*: Without intervention, agents repeatedly generate similar failing strategies
3. *Lack of reusability*: Each problem is solved from scratch without learning from successes

Our Approach. We present Agentic-Z3, a multi-agent framework that decomposes SMT solving into specialized roles: *planning* (Architect), *coding with probing* (Worker), and *diagnosis with learning* (Coach). The key innovations are:

- **Type-Aware Probing**: Pre-flight type checking via deterministic probe scripts that catch errors before expensive code generation

Preprint. Under review.

- **TTRL with Soft Reset**: Test-time reinforcement learning that detects stagnation (consecutive timeouts/duplicates) and triggers conversation reset while preserving failure context
- **Skill Crystallization**: Successful solutions are parameterized into reusable templates stored in a vector database for future retrieval
- **Hybrid SMT + LLM Fallback**: Low-confidence detection automatically switches to pure LLM reasoning when constraint translation is weak

Contributions. (1) A modular multi-agent architecture for LLM-guided SMT solving with clear separation of concerns. (2) Type-aware probing mechanism that eliminates type errors without LLM overhead. (3) TTRL loop with soft reset that prevents death spirals via stuck detection and temperature boosting. (4) Evolutionary memory system that learns reusable patterns across problems. (5) Comprehensive evaluation on TestEval benchmark demonstrating state-of-the-art path coverage (76.8% exact match) with detailed analysis of when formal methods help vs hurt.

2 Related Work

LLMs for Test Generation. Recent work has explored using LLMs for automated test generation. The TestEval benchmark (3) provides a systematic evaluation framework for targeted coverage tasks (line, branch, path). Direct prompting approaches achieve high execution rates but lack formal guarantees. Our work builds on this foundation but adds formal constraint solving.

Symbolic Execution + LLMs. AutoExe (2) and related work (4; 5) combine symbolic execution with LLMs to generate inputs satisfying path conditions. AutoExe uses LLM-powered slicing and constraint translation but lacks iterative refinement. Our TTRL mechanism provides error-driven repair that these approaches lack.

Formal Verification with LLMs. LLM-Sym (7) uses self-refine loops for verification tasks. We adopt a similar error-feedback principle but focus on SMT solving specifically and add multi-agent decomposition. The neuro-symbolic reasoning literature (7) provides foundations for combining neural and symbolic methods.

Test-Time Reinforcement Learning. AlphaProof (1) demonstrates test-time RL for mathematical reasoning, using search and refinement at inference time. We adapt the core insight—iterative exploration with failure-driven adaptation—to the SMT domain. Our soft reset mechanism is inspired by BFS-Prover's context clearing strategy but preserves compressed failure summaries.

Multi-Agent Systems. Recent surveys (8; 9) highlight the effectiveness of multi-agent architectures for complex reasoning. We specialize this paradigm for SMT solving with domain-specific agents (planning, coding, diagnosis) rather than generic reasoning agents.

Skill Learning and Memory. Template-based approaches like LEGO-Prover extract reusable patterns from successful proofs. Our skill crystallization mechanism extends this to SMT solving, using vector similarity (ChromaDB) for retrieval rather than exact matching.

Positioning. Unlike pure prompting (no formal grounding) or pure symbolic execution (expensive, brittle), we achieve a hybrid approach: structured decomposition, iterative refinement, and evolutionary memory. Our evaluation shows when this complexity is justified (formal precision tasks) versus when simpler methods suffice.

3 Methodology

3.1 System Architecture

Agentic-Z3 implements a state machine orchestrated by an *Engine* that coordinates three specialized agents (Figure **??**). The solving pipeline proceeds through distinct phases:

Planning Phase. The Architect agent analyzes the problem and produces a structured *blueprint* containing: (1) variables with precise Z3 types (Int, Real, Bool, String, BitVec), (2) constraint groups with logical dependencies, and (3) solving strategy hints. This hierarchical decomposition enables meaningful diagnosis when failures occur—unsat cores can be mapped back to high-level constraint groups.

TTRL Loop. The Worker agent performs type-aware probing, generates Z3 code with tracked constraints, and executes it. On UNKNOWN/ERROR, the TTRLCache detects stagnation and may trigger soft reset. The loop exits on SAT/UNSAT or after max retries.

Result Handling. SAT results trigger skill crystallization (Coach extracts reusable templates). UNSAT results with high-severity diagnosis trigger re-planning. The system balances exploration (soft reset) with exploitation (skill reuse).

3.2 Type-Aware Probing

LLMs trained on Python generate dynamically-typed code, but Z3 requires strict type declarations. A common failure is mixing Int and Real in arithmetic or comparing String with Char. Traditional approaches crash on first type error.

Our Solution. Before generating full constraint code, the Worker builds a minimal *probe script* deterministically (no LLM call):

1: **for** each variable v in blueprint **do**
2: Declare v with blueprint type (e.g., `x = Int('x')`)
3: Add trivial type-correct constraint (e.g., `x >= 0`)
4: **end for**
5: Execute probe with short timeout (1000ms)

If the probe succeeds, types are verified and cached across retries. If it fails, the error is captured and the system retries with diagnosis. This eliminates an entire class of failures early and cheaply.

Key Insight. Probe results are cached per blueprint (cached across retries), reducing overhead. The deterministic generation avoids LLM variability and ensures consistency.

3.3 TTRL with Soft Reset

Traditional retry loops suffer from *death spirals*: the LLM generates similar failing code repeatedly. Inspired by AlphaProof's test-time RL (1), we implement a threshold-based reset mechanism.

Stuck Detection. The TTRLCache tracks: (1) consecutive UNKNOWN results (solver timeouts), (2) consecutive ERROR results with identical messages, (3) code hash duplicates. Soft reset triggers when:

- Consecutive UNKNOWNs $\geq$ threshold - 1 (default: 2)
- OR duplicate code hash detected
- OR many identical errors (≥ 4) suggesting unfixable bug

Soft Reset Process. Unlike hard reset (full restart), soft reset carefully preserves context:

1. Capture compressed failure summary from TTRLCache
2. Clear LLM conversation history (removes bias)
3. Re-inject: (a) original problem, (b) "what NOT to do" summary
4. Boost temperature ($0.2 \rightarrow 0.7$) to force exploration diversity

Why This Works. The reset breaks local optima (LLM forgets the failed path) while preserving lessons (knows what failed). Temperature boosting ensures different code generation. This is complementary to error-driven repair: errors get traceback feedback without reset, while timeouts/duplicates trigger reset.

3.4 Skill Crystallization

When Z3 returns SAT, the solution is not just cached—it's *generalized* into a reusable template. The Coach agent performs:

Parameterization. Replace numeric literals with placeholders: `x <= 100` becomes `x <= {{UPPER_BOUND}}`. This is heuristic-based (skips small numbers < 10) but fast.

LLM Enrichment. Optionally, the Coach prompts the LLM for rich metadata: template name, description, applicable patterns. This adds semantic search capability but incurs cost (disabled in benchmarks).

Storage and Retrieval. Templates are stored in ChromaDB with vector embeddings. On new problems, the top-k similar skills are retrieved and injected into the Worker's prompt. This enables curriculum learning: warmup problems build foundational skills used for harder problems.

Trade-off. Skill library adds overhead (vector DB I/O, LLM crystallization call) but provides long-term benefit across many problems. For benchmarks where each problem is solved once, we disable it.

3.5 Hybrid SMT + LLM Approach

Pure SMT translation fails when path conditions involve internal state or complex loops. Pure LLM lacks formal grounding. Our hybrid approach detects translation quality and adapts:

Low-Confidence Detection. We classify a result as low-confidence if: (1) SAT but zero constraints were added (translation failed), (2) model doesn't cover any input parameters, or (3) status is non-SAT.

Fallback Strategy. On low-confidence, automatically switch to zero-shot LLM prompting using TestEval templates. This preserves execution quality when SMT is unreliable. The system logs which path it took (direct Z3, engine, fallback) for analysis.

4 Experimental Setup

Dataset. We use TestEval (3), a benchmark of 210 LeetCode problems with targeted coverage tasks. We select 15 hard-difficulty tasks (difficulty=3) with multiple sampled execution paths per task, totaling 56 paths. Tasks are selected reproducibly via random seed 42.

Baselines. We compare against three approaches:

- **Zero-shot**: Direct LLM prompting using TestEval templates (no formal reasoning)
- **AutoExe LLM**: LLM-based symbolic execution (mimics AutoExe (2) reasoning)
- **Vanilla Z3**: Single-shot text-to-Z3 translation with no iterative refinement
- **Agentic-Z3 (ours)**: Full multi-agent pipeline with TTRL and skill library

Evaluation Metrics.

- *Syntax*: % of generated tests that compile as valid Python
- *Exec*: % of tests that execute successfully and call the target function
- *Exact*: % of tests whose executed path exactly matches the target path
- *Similarity*: Average LCS-based path similarity (0–1 scale)

Tests are executed on instrumented code that logs line execution. Exact match requires the logged path to perfectly match the reference path.

Implementation. All approaches use GPT-4 (gpt-4-turbo). Agentic-Z3 uses: Z3 timeout=5000ms, max_retries=3, stateless history mode (prevents context overflow in benchmarks). Blueprint caching is enabled to reduce Architect calls for multi-path tasks. Skill library and crystallization are disabled for fair comparison (each problem solved once).

5 Results

5.1 Main Results

Table 1 shows our main findings. Agentic-Z3 achieves the highest exact path coverage (76.8%), outperforming all baselines.

Table 1: Path Coverage Benchmark Results (56 paths, 15 hard tasks)

Approach	Syntax	Exec	Exact	Similarity
Agentic-Z3 (ours)	100.0%	98.2%	**76.8%**	0.9071
Zero-shot	100.0%	100.0%	71.4%	0.9036
AutoExe LLM	87.5%	85.7%	67.9%	0.7786
Vanilla Z3	91.1%	37.5%	12.5%	0.1571

Error Analysis. Agentic-Z3 has only 1 execution error (1.8%): a ValueError on complex edge parameter structure. Zero-shot has 0 errors (perfect execution). AutoExe has 8 errors (14.3%), mostly missing test functions. Vanilla Z3 has 35 errors (62.5%), dominated by `function_not_called` (28.6%) where the generated test uses `pass` instead of calling the target function.

5.2 Key Findings

Finding 1: Formal grounding improves exact path targeting (+5.4%). Despite zero-shot's perfect execution, Agentic-Z3 achieves 76.8% vs 71.4% exact match. When constraint translation is faithful, Z3 finds precise satisfying assignments. Error-driven repair (injecting tracebacks into prompts) helps the LLM fix bugs iteratively.

Finding 2: Single-shot translation catastrophically fails. Vanilla Z3's naive approach shows: 91.1% syntax $\rightarrow$ 37.5% exec $\rightarrow$ 12.5% exact. The 16/35 `function_not_called` errors reveal that without iterative refinement, translation failures cascade. Many generated scripts output `pass` fallbacks. This validates our iterative approach.

Finding 3: Near-perfect execution with 6.1× path coverage gain over Vanilla Z3. Agentic-Z3's test hardening (safe defaults, syntax validation, guaranteed function calls) ensures 98.2% execution. The 76.8% vs 12.5% exact match (6.1× improvement) demonstrates the value of multi-agent iteration, diagnosis, and repair.

Finding 4: Cost-quality tradeoff. Zero-shot uses $\sim$1 LLM call/path. Agentic-Z3 uses $\sim$3–5 calls/path (Architect planning + Worker retries + optional Coach). We achieve +5.4% exact match at 3–5× API cost. For applications requiring formal precision, this tradeoff is justified.

5.3 Ablation Analysis

Blueprint Caching. Reusing Architect plans across paths of the same task (same function signature, different constraints) reduces LLM calls by $\sim$3×. This optimization makes multi-path tasks viable.

Direct Z3 Fast Path. Our runner first attempts direct constraint translation (regex-based, no LLM). This succeeds for $\sim$60% of paths with simple conditions. Only when translation is weak or results are low-confidence do we escalate to the full engine.

Hybrid Fallback. Low-confidence detection (SAT but no constraints added, or model doesn't cover inputs) triggers zero-shot fallback. This explains why Agentic-Z3's similarity (0.9071) is close to zero-shot (0.9036): hard paths use LLM reasoning when SMT fails.

Parameter-Aware Defaults. Generic defaults (empty lists, zero values) cause crashes on structured parameters (edges as 3-tuples, 2D grids). Our parameter-name-aware defaults (e.g., `[[0, 1]]` for edges) eliminate such failures.

6 Discussion and Limitations

When Agentic-Z3 Excels. The framework is most effective for problems where: (1) path conditions map cleanly to SMT constraints (arithmetic, comparisons, simple string ops), (2) formal precision matters (need provable satisfaction), and (3) iterative refinement can fix translation errors. The 76.8% exact match validates this for a non-trivial subset of hard tasks.

Translation Loss. Our current translation pipeline has limitations:

- Heuristic filtering drops loop headers and internal-variable conditions, which may delete meaningful constraints
- Regex-based converter weakens unsupported constructs to `True`, making the SMT problem easier but less faithful
- Direct Z3 mode may silently skip failed constraint translations

These explain why similarity scores (0.9071 vs 0.9036) are comparable—when translation is weak, we fall back to LLM reasoning.

API Cost. The multi-agent pipeline is token-heavy. Planning, diagnosis, and skill crystallization add overhead. For single-use tasks (benchmarks), this costs 3–5× more than zero-shot. For repeated use (curriculum learning), the skill library amortizes cost.

Time Overhead. Multiple LLM round-trips plus Z3 execution add latency. Retry loops compound this. For real-time applications, the wall-clock time may be prohibitive. However, the 98.2% execution rate suggests the approach is robust.

Execution Gap (98.2% vs 100%). The single ValueError on edge parameter structure is addressable via enhanced parameter-aware defaults. The tradeoff is between generic safety (broad coverage) and specific correctness (exact structure matching).

7 Conclusion and Future Work

We presented Agentic-Z3, a multi-agent framework that combines LLMs with Z3 SMT solver through hierarchical planning, type-aware probing, and test-time reinforcement learning. Our approach achieves state-of-the-art exact path coverage (76.8%) on hard tasks, demonstrating that formal methods can enhance LLM reasoning when translation quality is managed.

Key Insights. (1) Type-aware probing eliminates a major class of failures deterministically. (2) Soft reset with temperature boosting breaks stagnation loops. (3) Hybrid SMT + LLM fallback preserves quality when translation is weak. (4) Single-shot approaches fail catastrophically; iteration is essential.

Future Directions. (1) *AST-based translation*: Replace regex with AST visitors to reduce semantic loss. (2) *Translation coverage scoring*: Measure % of conditions captured, gate SMT vs LLM routing. (3) *Persistent caching*: Cache by (task, path, model) to avoid redundant API calls. (4) *Prompt slimming*: Send only relevant code slices, not full functions. (5) *Cost instrumentation*: Track tokens/time per path to optimize efficiency.

The framework is modular and generalizes beyond test generation to any LLM + SMT integration task (program verification, synthesis, planning). While costlier than pure prompting, it provides formal precision where guarantees matter.

References

References

[1] Thomas Hubert et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi:10.1038/s41586-025-09833-y

[2] Yihe Li, Ruijie Meng, and Gregory J. Duck. Large Language Model Powered Symbolic Execution. *Proc. ACM Program. Lang.*, 9:3148–3176, 2025.

[3] TestEval: Benchmarking Large Language Models for Test Case Generation. Dataset and benchmark framework, 2024.

[4] Yaoxuan Wu, Xiaojie Zhou, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. Generating and Understanding Tests via Path-Aware Symbolic Execution with LLMs. arXiv:2506.19287, 2024.

[5] Wenhan Wang, Kaibo Liu, Zeyu Sun, An Ran Chen, Ge Li, Gang Huang, and Lei Ma. Can Large Language Models Solve Path Constraints in Symbolic Execution? arXiv:2511.18288, 2024.

[6] Python Symbolic Execution with LLM-powered Code Generation. arXiv:2409.09271, 2024.

[7] Sumit Kumar Jha. Planning using Neuro Symbolic Reasoning. arXiv:2309.16436, 2023.

[8] Guibin Zhang et al. The Landscape of Agentic Reinforcement Learning for LLMs: A Survey. arXiv:2509.02547, 2024.

[9] Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. Scaling up Multi-Turn Off-Policy RL and Multi-Agent Tree Search for LLM Step-Provers. arXiv:2509.06493, 2024.

[10] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic Memory for LLM Agents. arXiv:2502.12110, 2024.

A Implementation Details

Code Availability. Full implementation available at the project repository. `https://github.com/qilem/Agentic_Z3`