PoC-Gym: Towards More Reliable LLM-Assisted Proof-of-Concept Exploit Generation

Derin Gezgin, Amartya Das, Shinhae Kim, Zhengdong Huang, Nevena Stojkovic, Claire Wang

Second Workshop on Large Language Models For Generative Software Engineering

What Is a Proof-of-Concept (PoC) Exploit?

A proof-of-concept exploit is a small, self-contained program that proves a reported vulnerability is *real* by running attacker-controlled input through the *actual* vulnerable code and triggering the unintended or dangerous behavior.

A vulnerability report alone

- flags a suspicious source→sink data flow
- says the code *might* be exploitable
- gives *no* evidence it can be triggered

A proof-of-concept

- supplies concrete input that *actually* triggers the flaw
- produces an observable effect at runtime
- emits a clear success signal (e.g. `[VULN]`)

A PoC turns a *claim* of exploitability into *executable evidence*.

A Concrete Example: “Zip Slip” Path Traversal

The bug

- A library unpacks ZIP entries *without* checking entry names.

The exploit idea

1. Craft an entry named with `../` so it escapes the target folder.
2. Feed it to the *real* vulnerable extractor.
3. Confirm a file landed *outside* the folder ⇒ print `[VULN]`.

```
public class PoCTest {
  public static void main(String[] a) throws
      Exception {
    // (1) Attacker input: name escapes the target
        dir
    ZipOutputStream zip = new ZipOutputStream(
        new FileOutputStream("evil.zip"));
    zip.putNextEntry(new ZipEntry(
        "../poc-exploited.txt"));
    zip.write("owned".getBytes());
    zip.close();
    // (2) Drive the *real* vulnerable library
    VulnerableZipExtractor.unzip("evil.zip", "
        output/");
    // (3) File escaped "output/" ? then the bug
        is real
    if (new File("poc-exploited.txt").exists())
      System.out.println("[VULN]" + " wrote outside
          dir");
  }
}
```

Main Claim

LLM coding agents can generate proof-of-concept exploits that *look* successful, but **lightweight validation overestimates their validity.**

Reliable PoC generation needs **dynamic validation against the vulnerable code path**.

We built PoC-Gym

LLM-assisted Java PoC pipeline

- CVE/CWE guidance
- static trace context
- iterative runtime feedback
- post-hoc trace validation

We tested

338

total runs

20 real-world Java CVEs

3 agent/model settings

We found

Runtime validation overestimates exploitability

116 → 65

116 pass runtime checks; only 65 reach the ground-truth vulnerable location

Why Generating Proof-of-Concept Exploits Matter

Tens of thousands of CVEs disclosed annually

SAST flags suspicious source–sink flows — a static alert is **not** proof of exploitability

A PoC turns a vulnerability report into **executable evidence**:

Reproduce the bug

Guide the patch

Create a regression test

Gap

Public CVEs often ship **without a usable PoC**; writing one by hand needs security expertise plus project-specific build knowledge.

Opportunity

LLM coding agents can inspect repositories, synthesize Java code, compile it, and iterate from execution feedback.

The Reliability Problem

Looks convincing

- Runs without errors
- Prints the expected output
- Creates the expected artifacts

But may not exploit the bug

- Hardcodes the side effect
- Simulates the vulnerable logic
- Reaches a benign API path
- Follows an unrelated trace

Does the PoC merely *satisfy a validation check*,
or does it *actually execute the vulnerable code*?

Two Levels of Validity

Level 1 · Runtime-valid

$c = 0 \ \wedge \ \texttt{[VULN]} \in s \ \wedge \ \textsc{SinkHit}(\tau)$

- Compiles and terminates
- Prints the success marker
- Reaches the selected static-trace sink (when trace guidance is used)

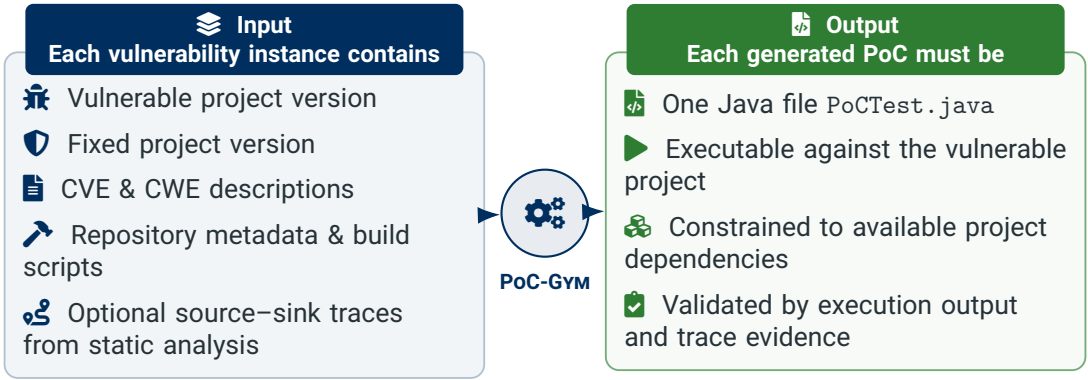
Level 2 · Post-hoc valid

$L^{\text{vuln}} \in \tau$

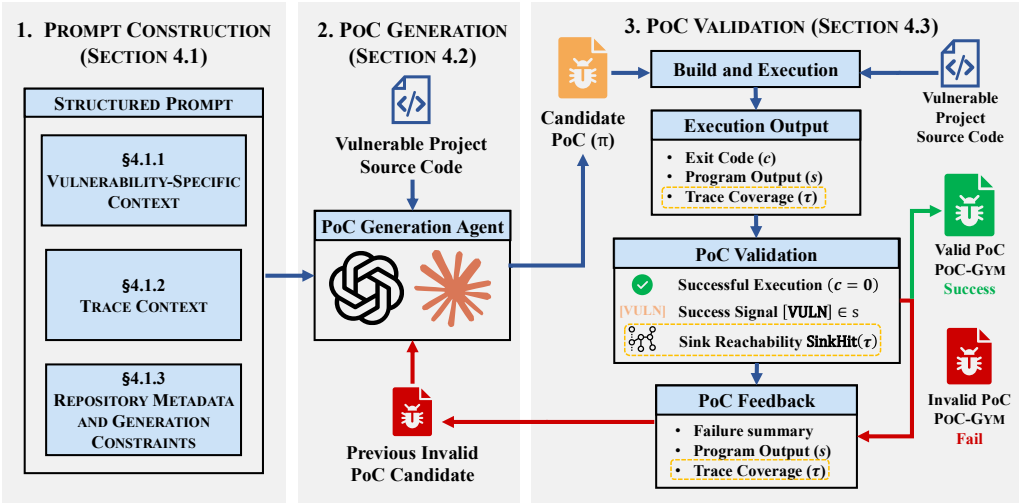
- A runtime-valid candidate *also* reaches the benchmark ground-truth vulnerable location
- Checked with dynamic AspectJ traces
- Grounded in CWE-Bench-Java annotations
- Stronger evidence of true exploitability

Increasing strictness. Post-hoc valid is the deeper check — every post-hoc-valid PoC is runtime-valid, but not the reverse.

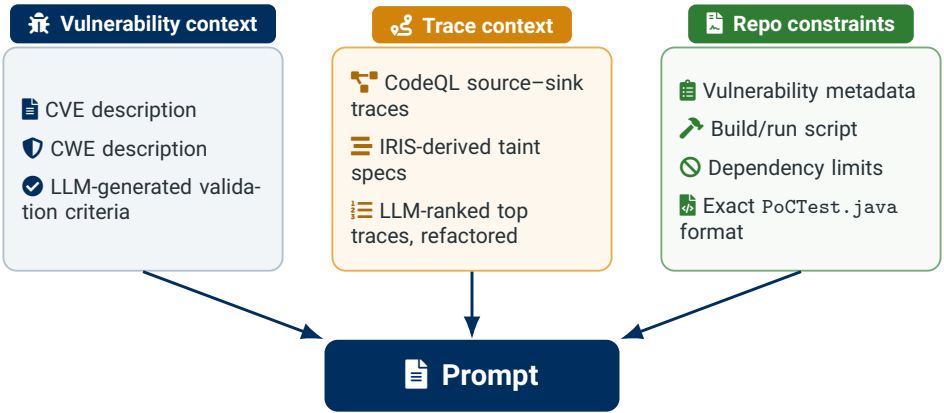
Evaluation Pipeline



System Overview

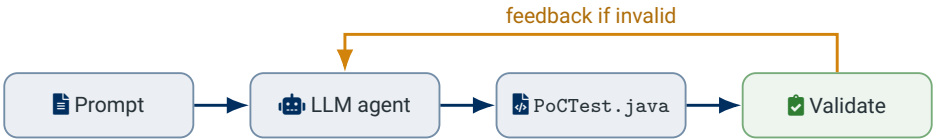


Prompt Construction



three contexts combined into one prompt

Generation and Validation Loop



Generation

- Fresh coding-agent session per run
- Agent sees the vulnerable project source tree
- Output constrained to a single Java main program
- Up to **10** iterations

Validation

1. Compile & execute the PoC
2. Check exit status and `[VULN]`
3. Record dynamic trace with AspectJ
4. Require selected sink reachability when trace-guided
5. Feed concise failure evidence back to the agent

Experimental Setup

Benchmark

20 real-world Java CVEs from CWE-Bench-Java

Vulnerable *and* patched project versions

Manually curated vulnerable locations from patch data

14 CVEs overlap with FAULTLINE

Models & agents

Claude Code with Claude Sonnet 4

Codex with GPT-5 Medium reasoning

Codex with `gpt-oss-20b`

CVE guidance & trace selection generated asynchronously with Claude Sonnet 4

Two Evaluation Settings

No-trace

Prompt

Vulnerability context, but **no** static trace

Validation

Execution *and* the generated success marker

Multi-trace

Prompt

Includes top source–sink traces

Validation

Selected sink reachability

Does adding trace context and a stricter in-loop validation target change *apparent* and *post-hoc* PoC validity?

Measured Metrics

 # Runtime-valid

- Passed the immediate validation loop
- Marker & execution checks
- Trace sink hit when trace-guided

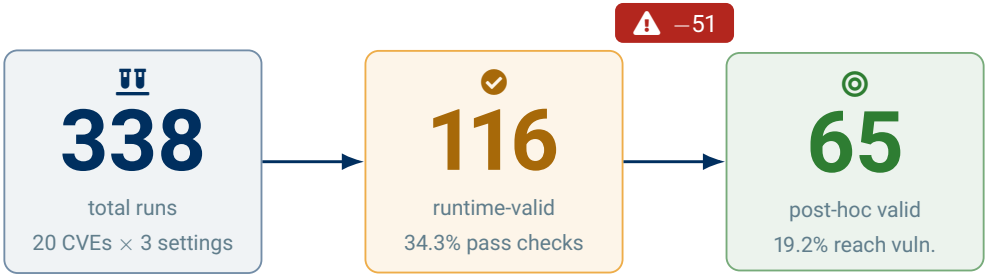
 # Post-hoc valid

- Rerun with AspectJ
- Normalize executed trace entries
- Match against vulnerable file/line range

Trace coverage

- Fraction of selected trace steps reached
- Reported for runtime-valid multi-trace PoCs

Runtime Success Overestimates Exploitation



51 / 116 (44%) runtime-valid candidates never reach the vulnerable code — success markers are **necessary, not sufficient**.

Results by Agent and Model

Agent	Model	# Runtime-valid		# Post-hoc valid		Avg. trace coverage
		No-trace	Multi-trace	No-trace	Multi-trace	
Claude Code	Claude Sonnet 4	17/20	**13/20**	8/20	**9/20**	**57.83%**
Codex	GPT-5 Medium	17/20	**13/20**	**9/20**	8/20	51.88%
Codex	`gpt-oss-20b`	**18/20**	5/20	5/20	4/20	43.18%
FAULTLINE reproduced baseline		5/14 on shared CVEs				

◎ Best value per metric in **bold**; *post-hoc valid* counts only candidates that reach the vulnerable code.

Comparison with FAULTLINE

PoC-Gym

8/14

post-hoc-valid on shared CVEs

FaultLine

5/14

post-hoc-valid on shared CVEs

- 14 CVEs overlap with FaultLine.
- We rerun the public FaultLine PoCs in its evaluation pipeline.
- PoC-Gym uses Java-specific static and dynamic analysis; FaultLine avoids language-specific analysis.

What Fails After Runtime Validation?

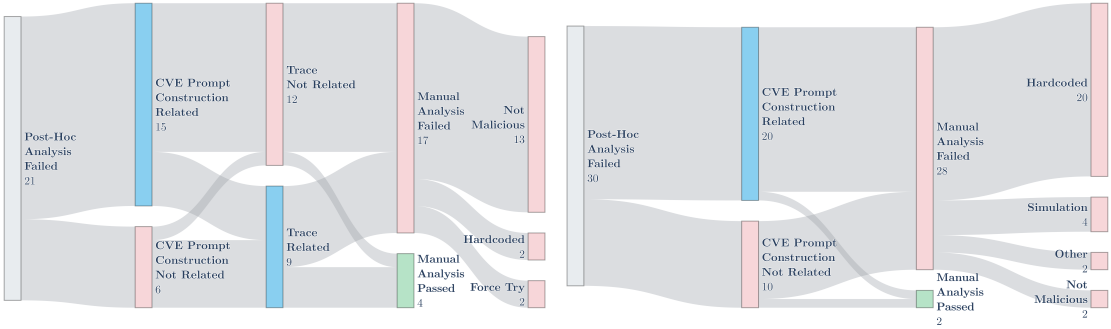
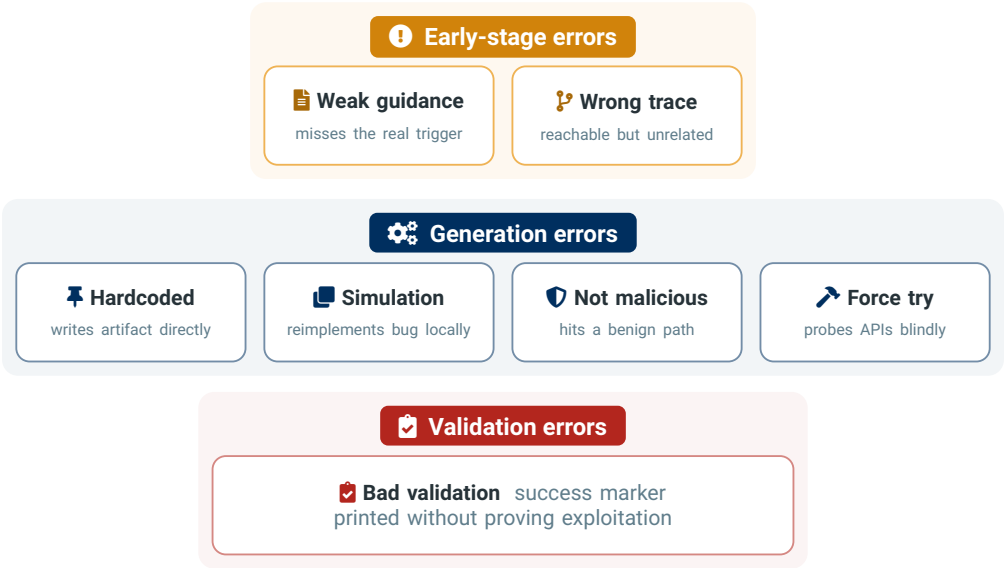


Figure 1: Manual analysis of candidates that pass runtime validation but fail post-hoc trace validation.

Recurring Failure Patterns



Threats to Validity

External validity

20 Java CVEs from `CWE-Bench-Java`.

Results may differ for other languages, ecosystems, vulnerability classes, or newer CVEs.

Stronger future coding agents may change absolute success rates.

Internal validity

LLM-generated guidance and trace ranking can introduce errors before PoC generation starts.

No-trace and multi-trace settings are not isolated ablations.

Agent runs are nondeterministic and depend on model versions and execution environments.

Key Conclusions

LLM agents are useful PoC generators

They synthesize executable Java candidates across real projects.

Runtime validation is too weak on its own

116 runtime-valid candidates become **65** post-hoc-valid after trace validation.

Static and dynamic analysis are complementary

Traces guide generation and give validation a program-grounded target.

Failure analysis should shape future systems

Hardcoding, simulation, weak guidance, and bad validation are recurring reliability risks.