

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4801
AI-assisted Software Engineering

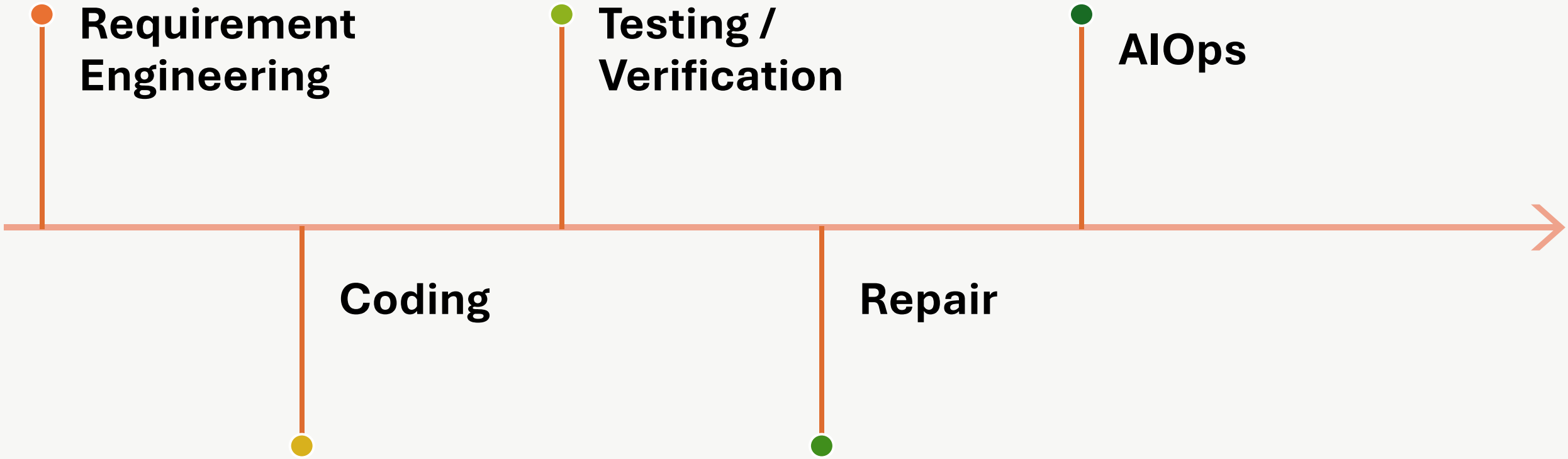
Lecture 13: Verification

Instructor: Jinsheng Ba | 2026 Fall

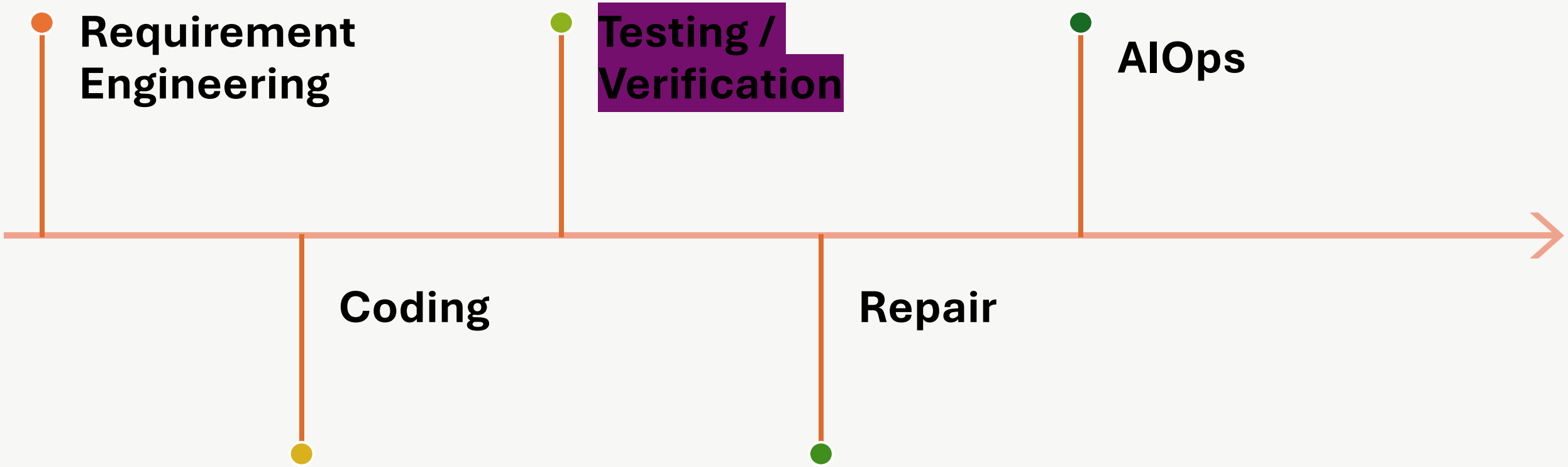
Homework 2

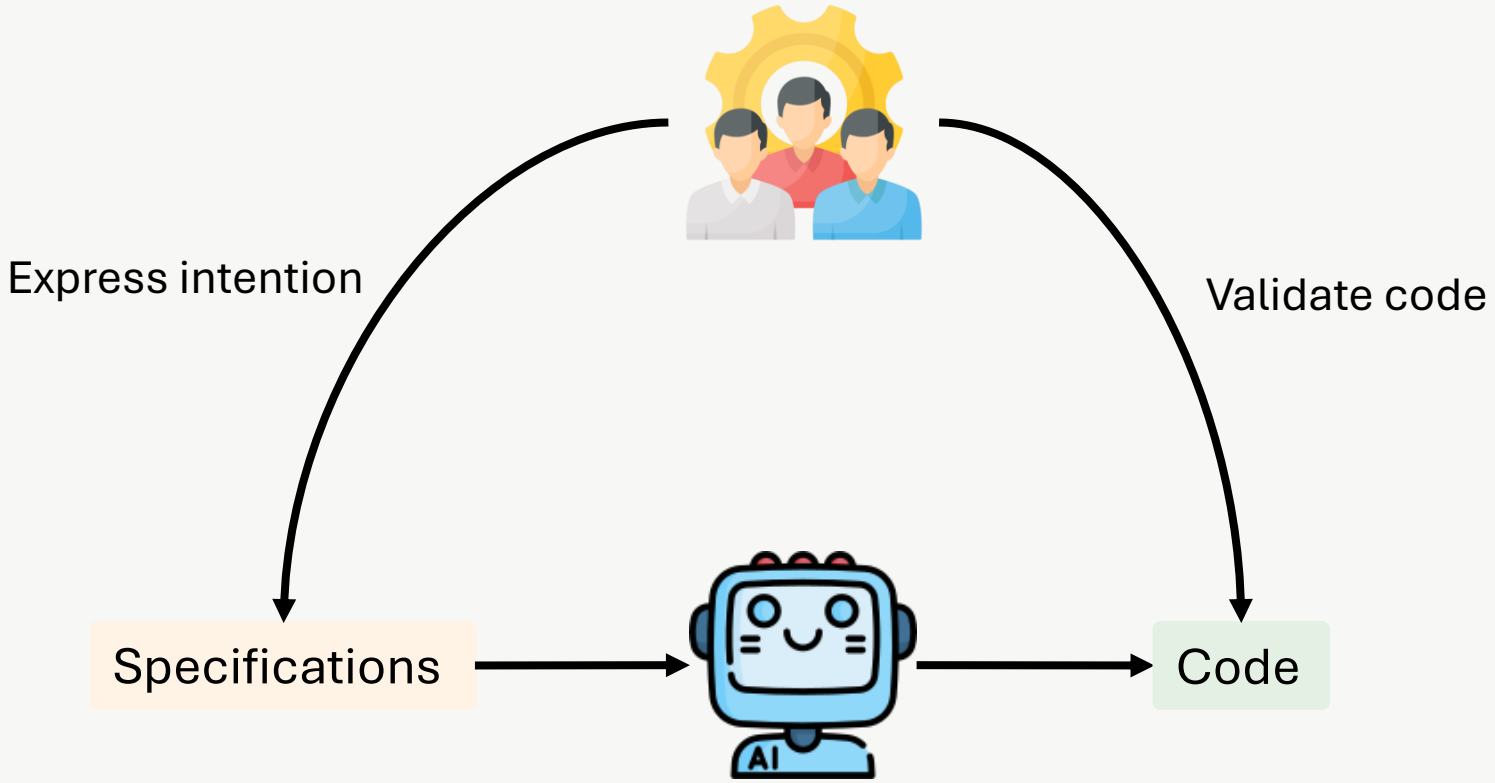
- DDL today

AI-Assisted Software Engineering



AI-Assisted Software Engineering

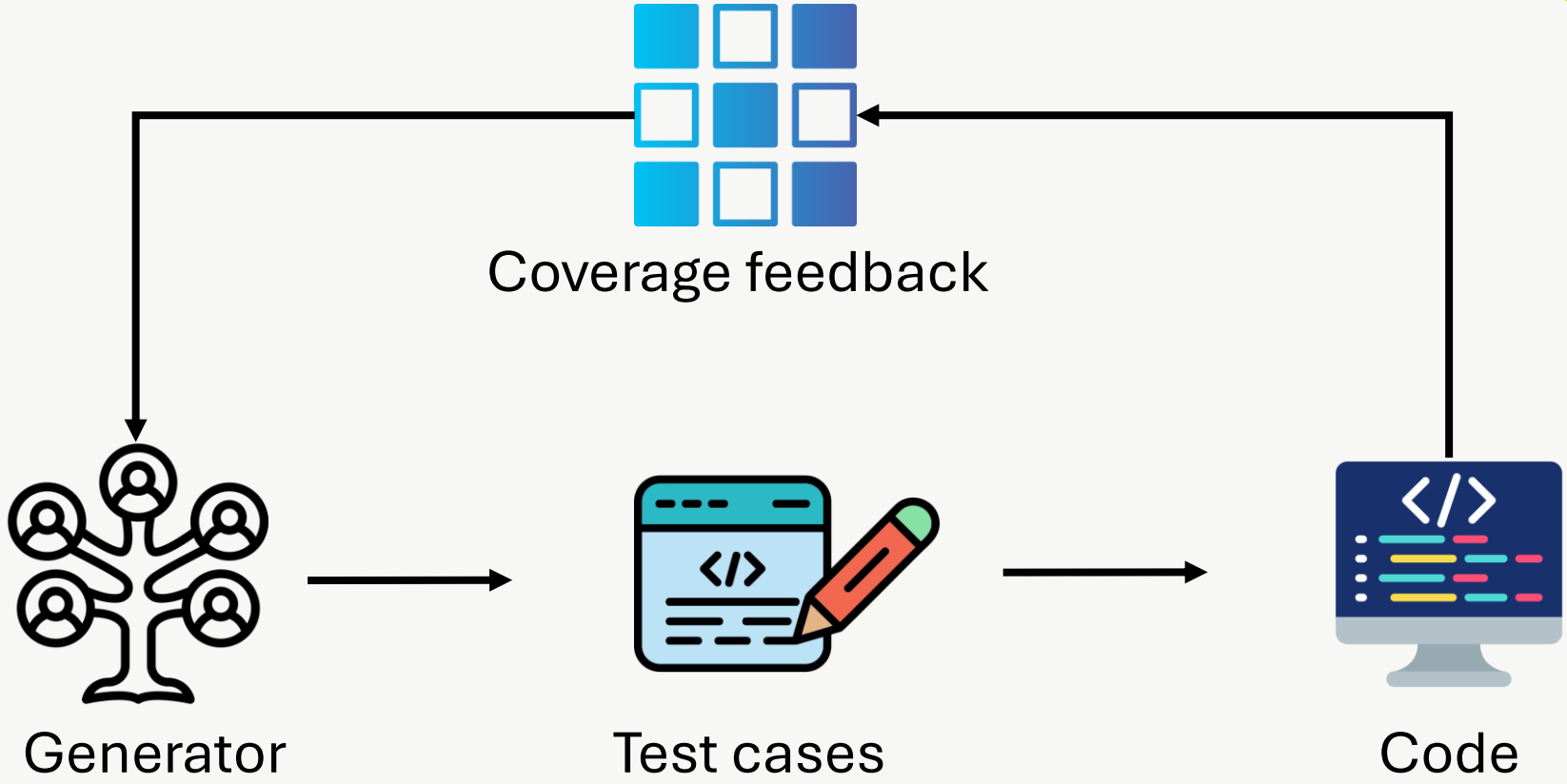




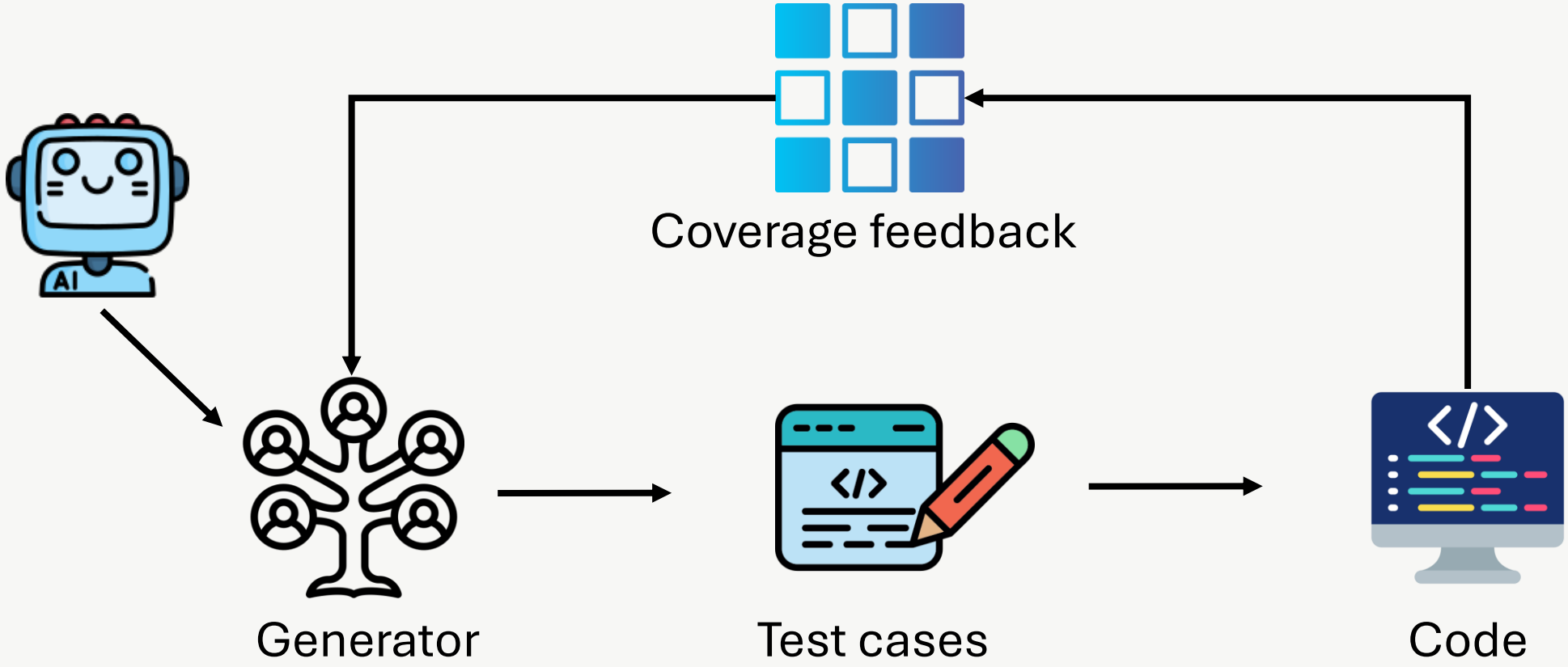
In LLM4SE, the roles of human are intention expression and code validation.

Apart from directly generating test inputs and oracles,
we can also generate testing tools!

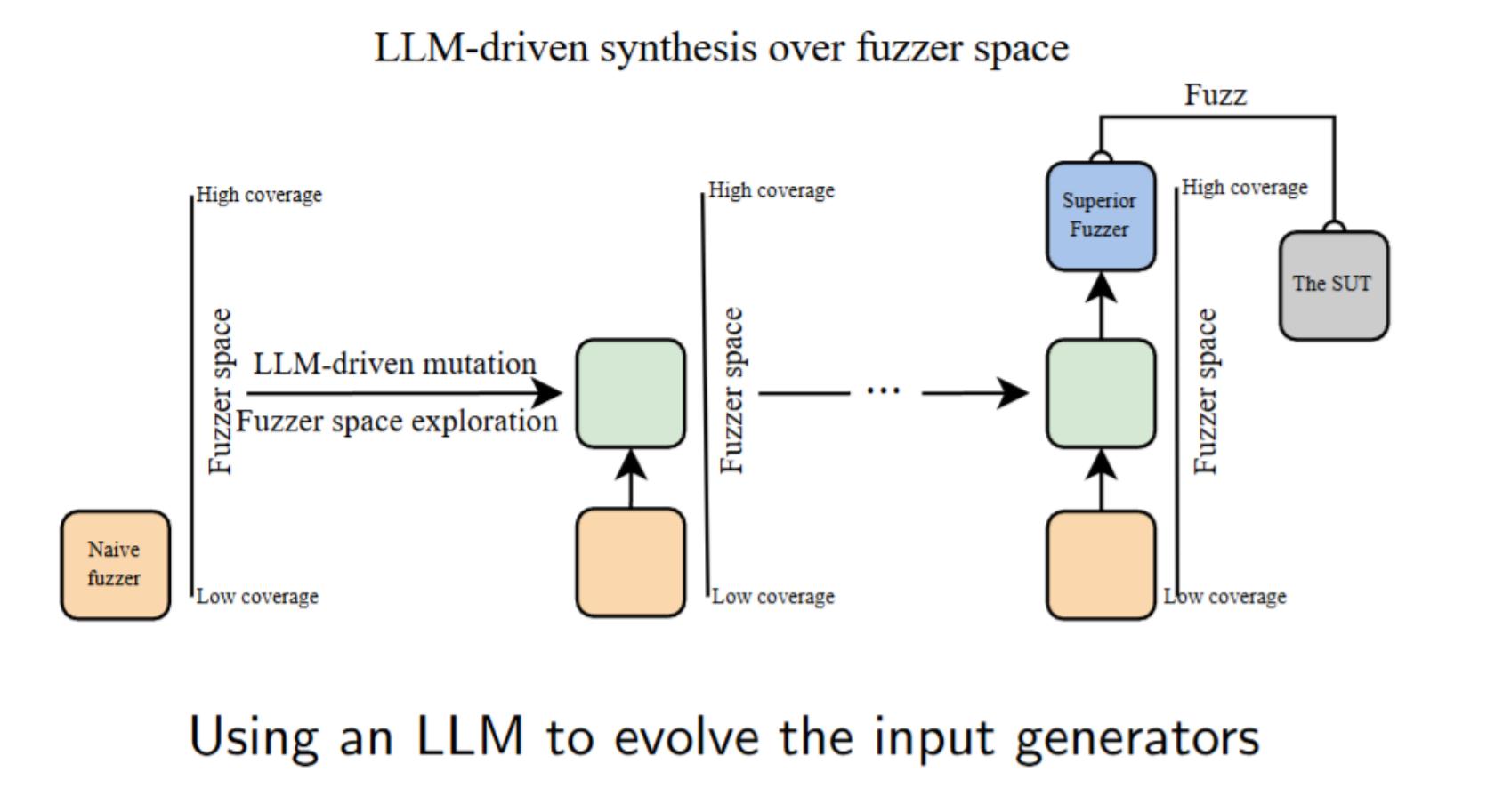
Fuzzing



Generator Synthesis by LLMs

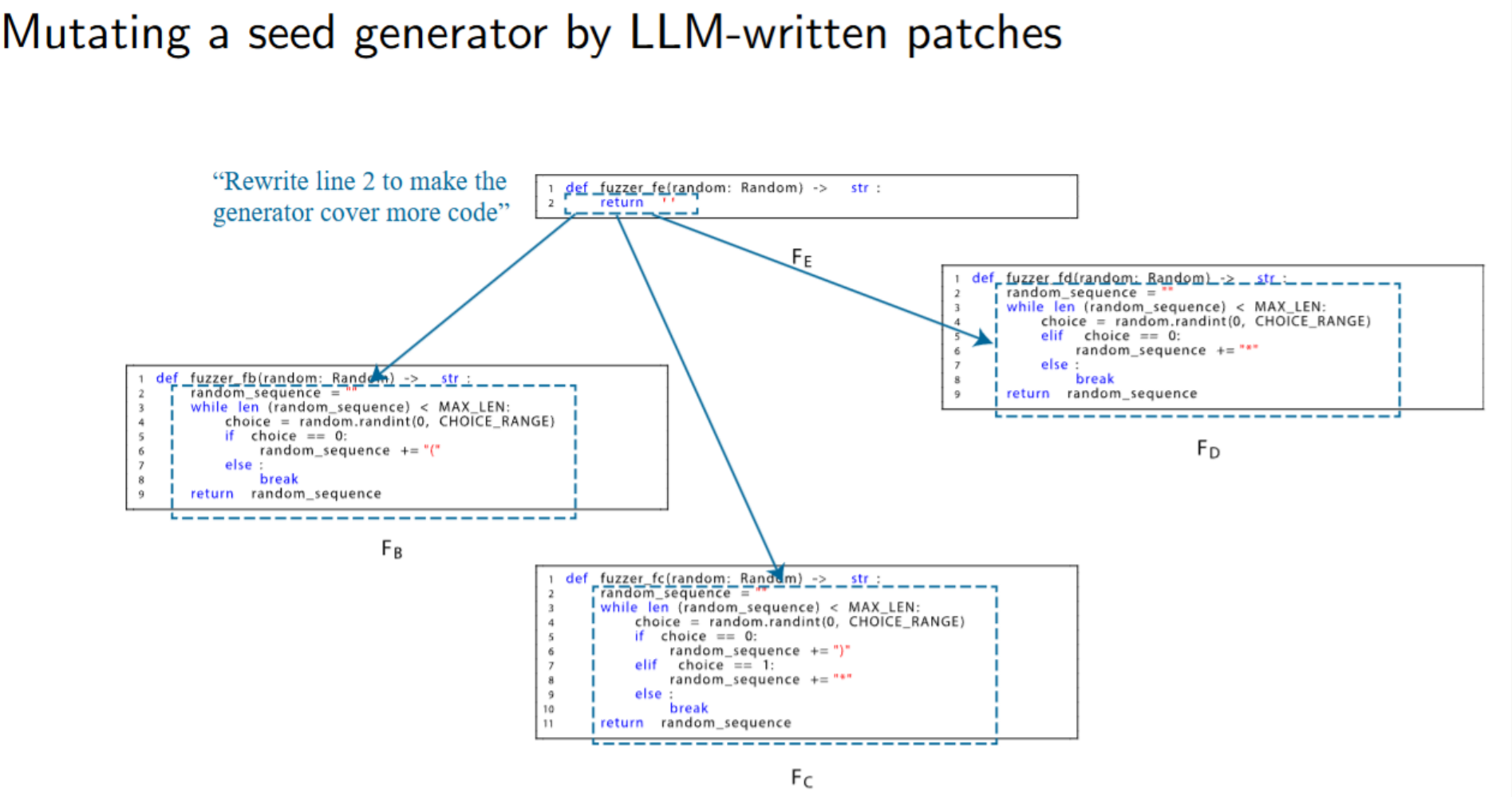


Generator Synthesis by LLMs



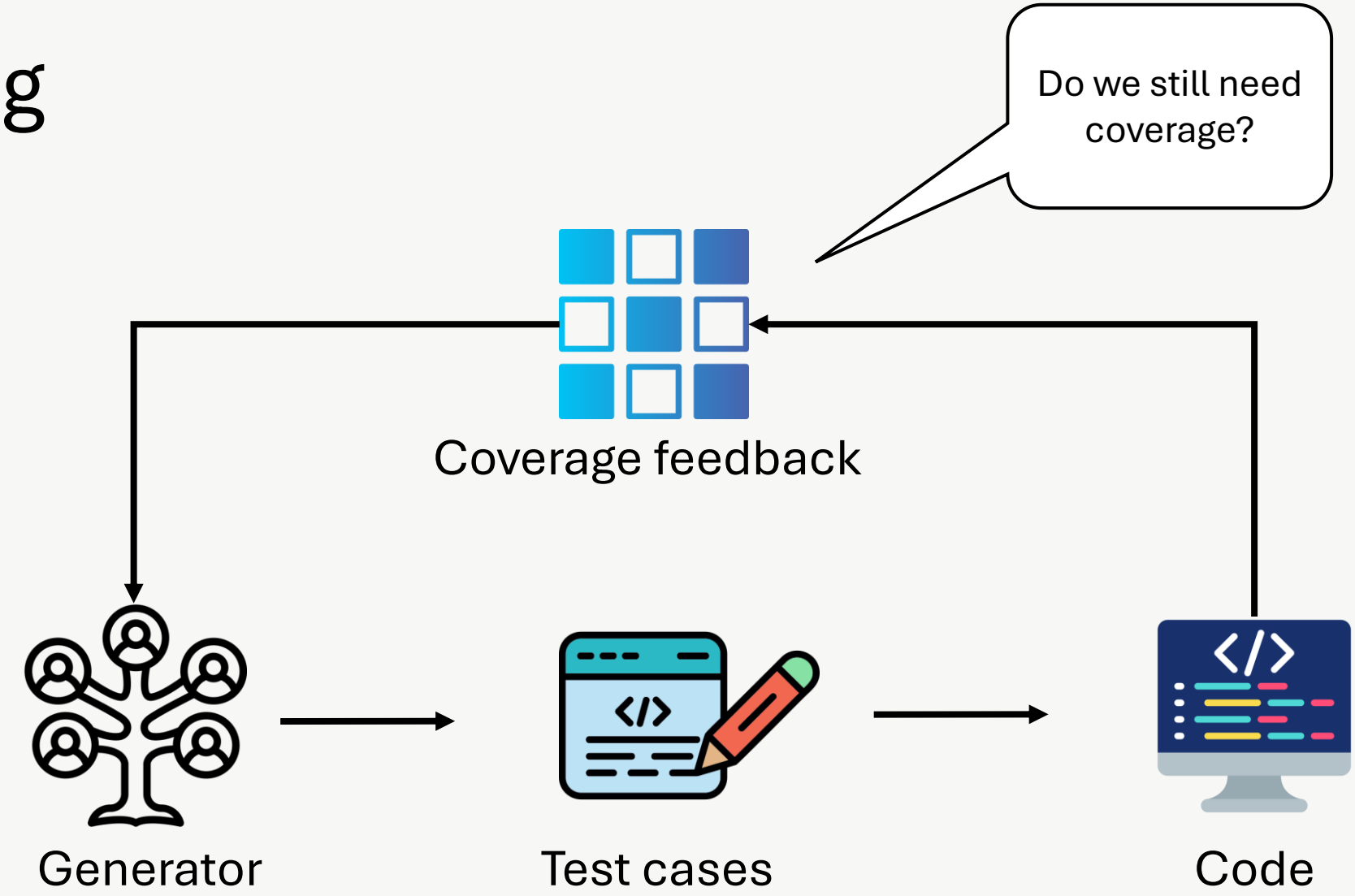
ELFuzz: https://www.usenix.org/system/files/usenixsecurity25-chen-chuyang.pdf

Generator Synthesis by LLMs

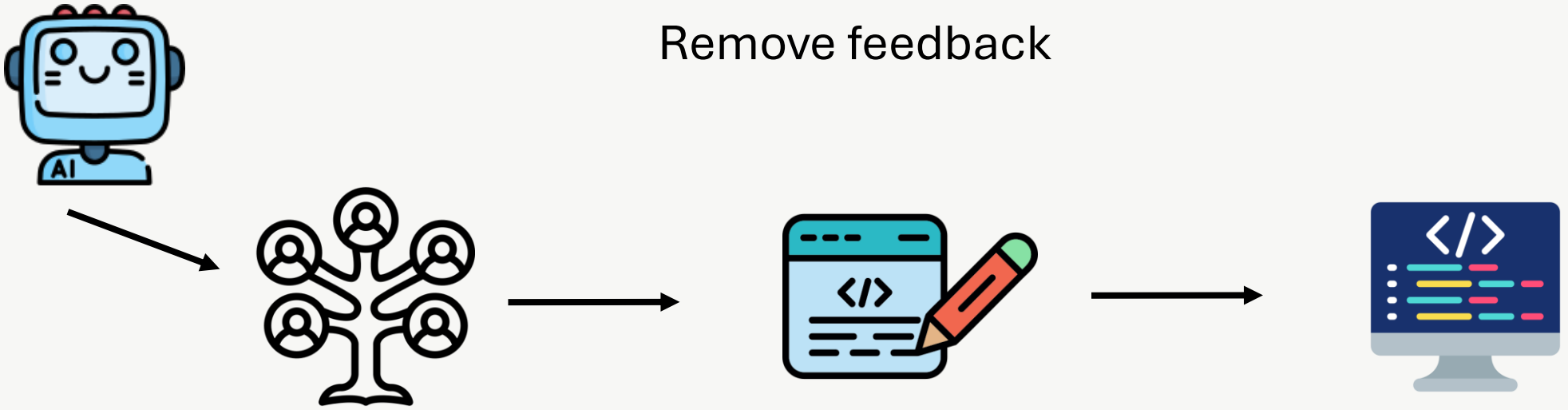


ELFuzz: https://www.usenix.org/system/files/usenixsecurity25-chen-chuyang.pdf

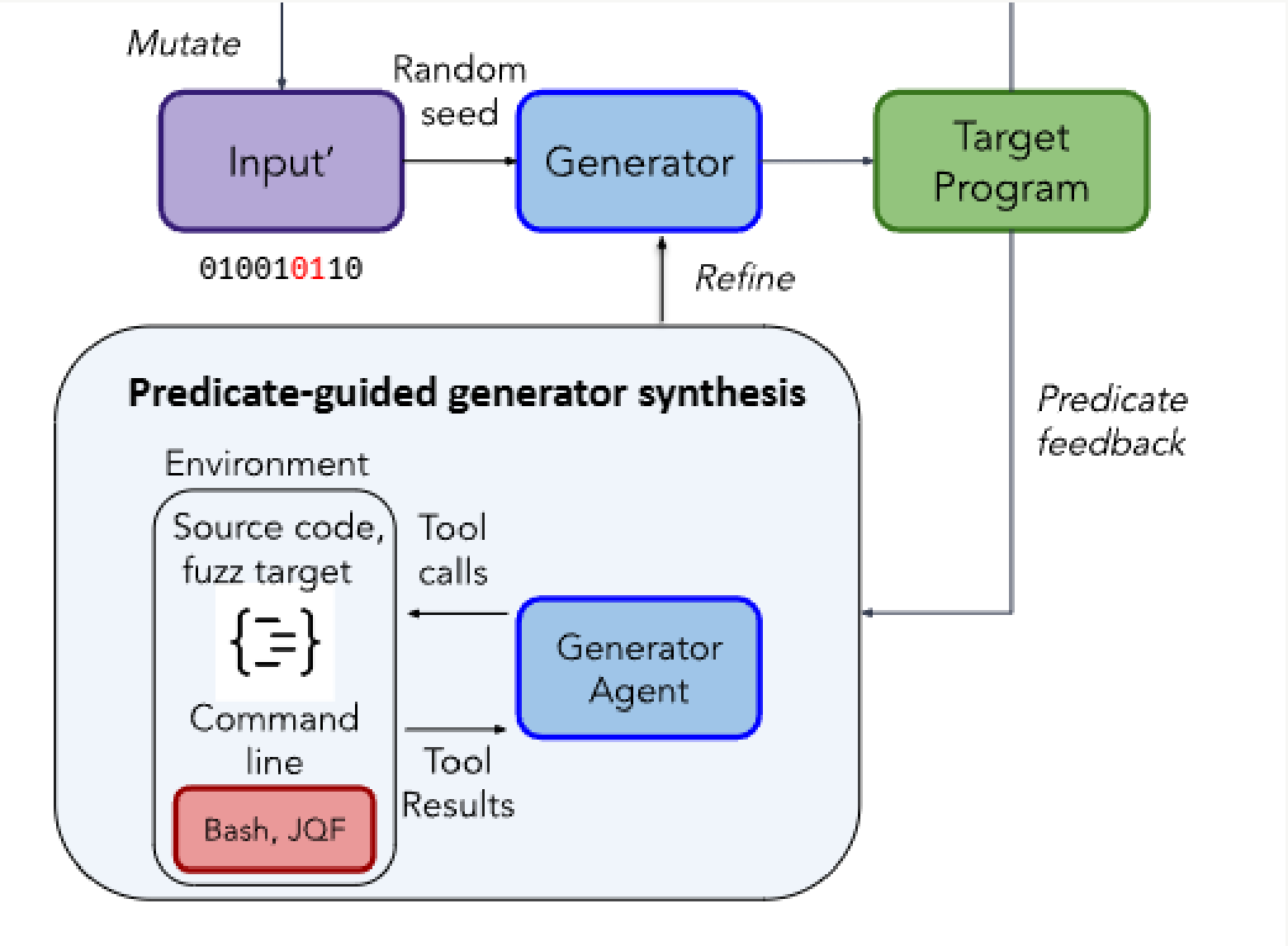
Fuzzing


Do we still need coverage?
Coverage feedback
Generator
Test cases
Code

Remove feedback

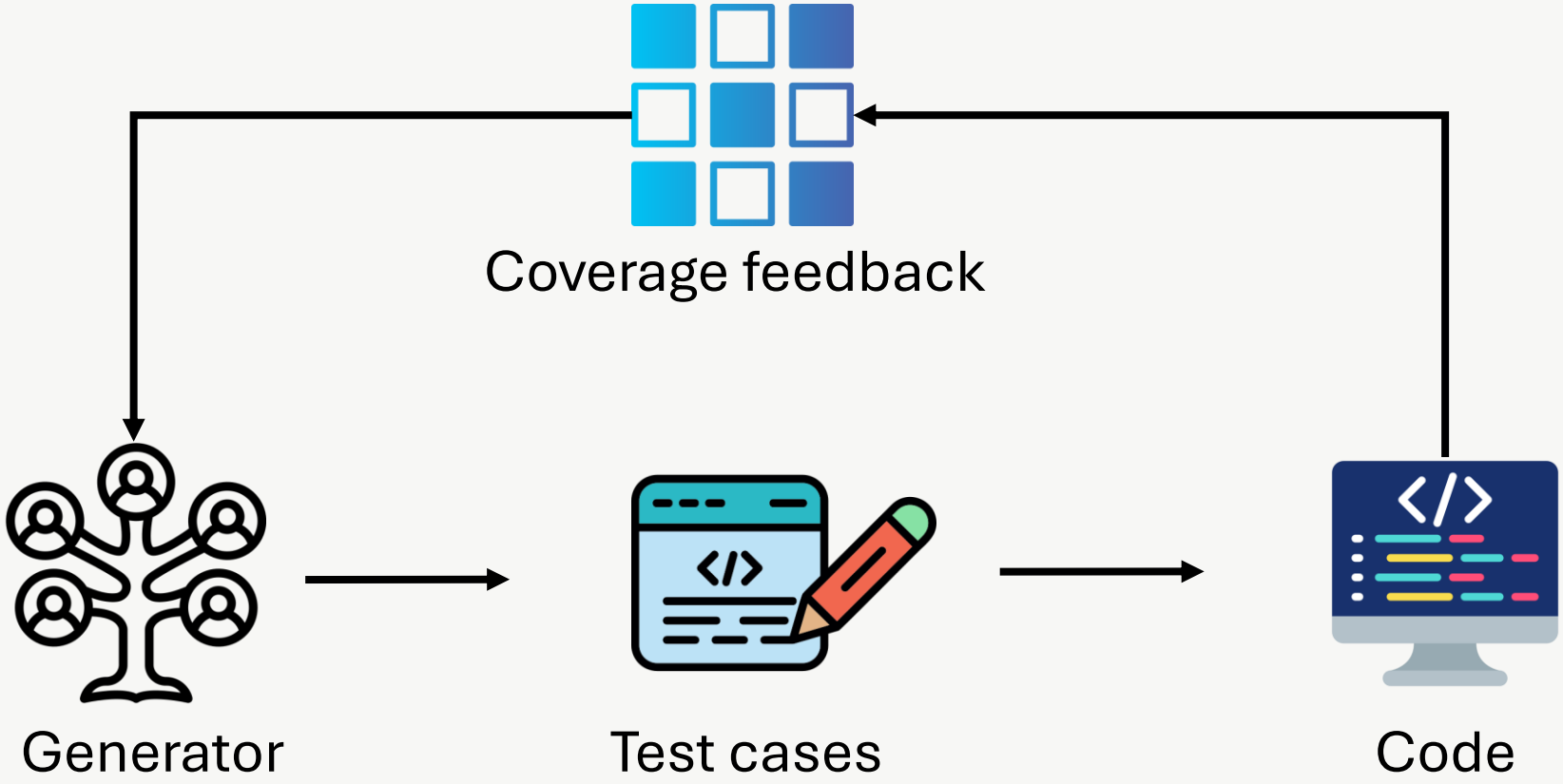


LLMs reason and analyze the target program, then refine the generators aiming to cover unexplored code branches.

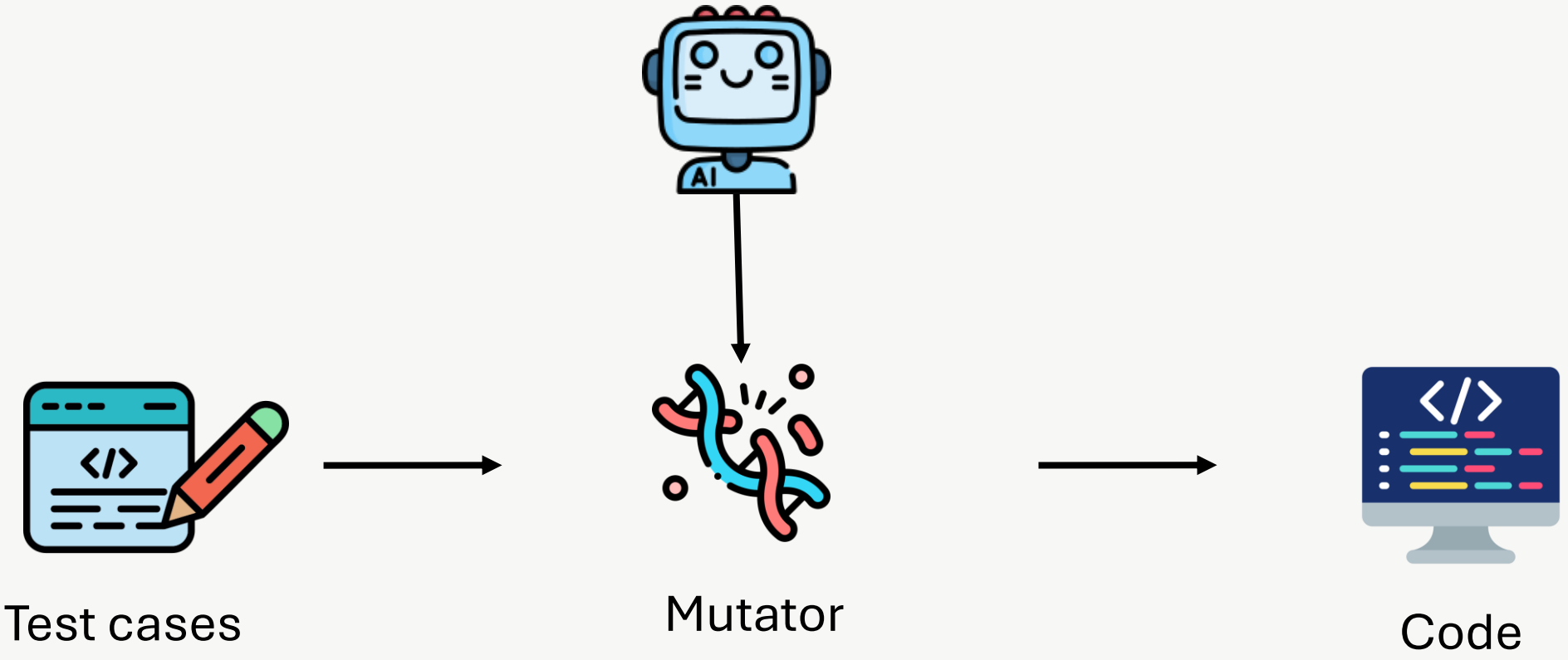


Generators Are All You Need: https://arxiv.org/pdf/2604.01442

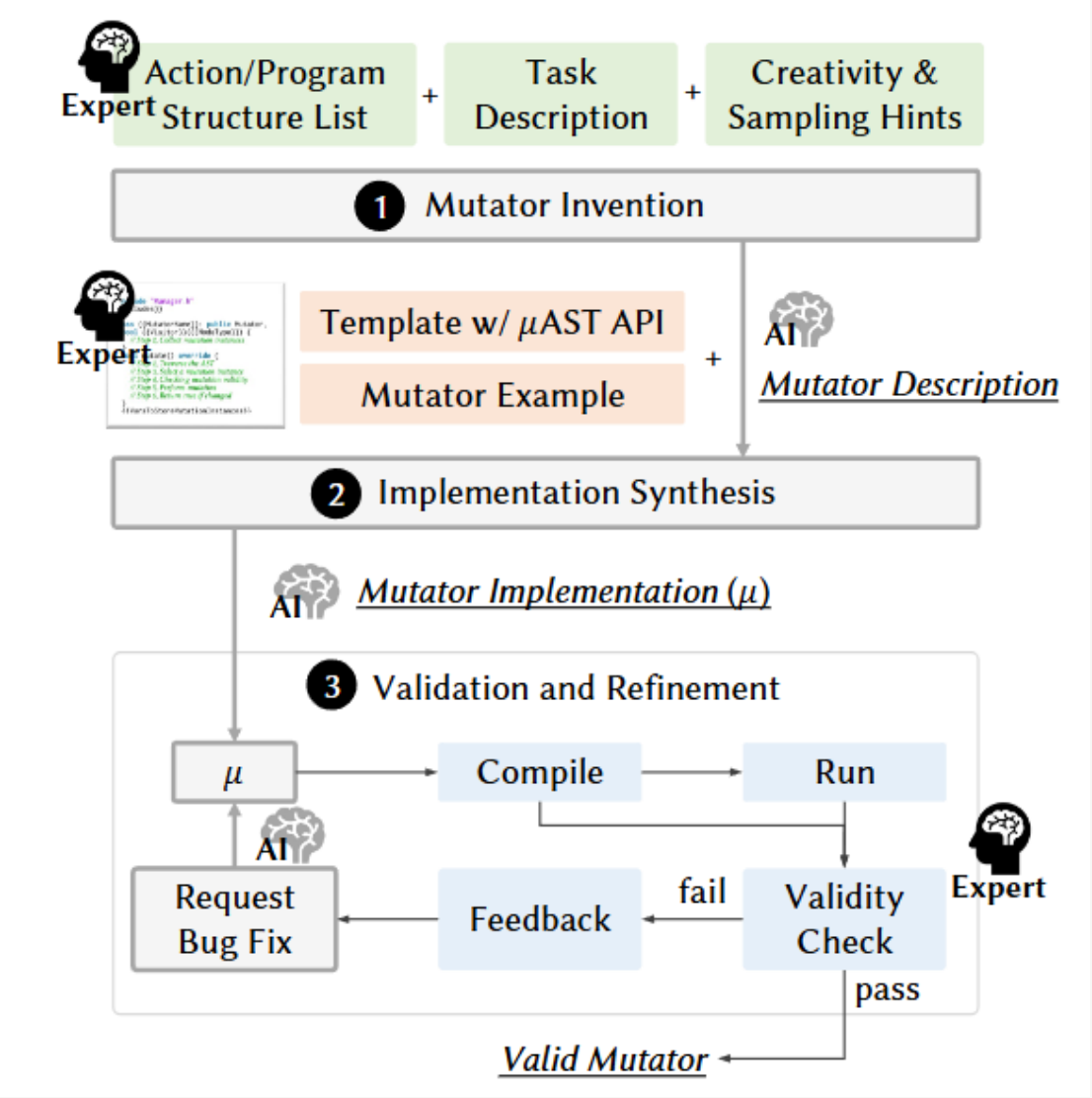
Fuzzing



Fuzzing

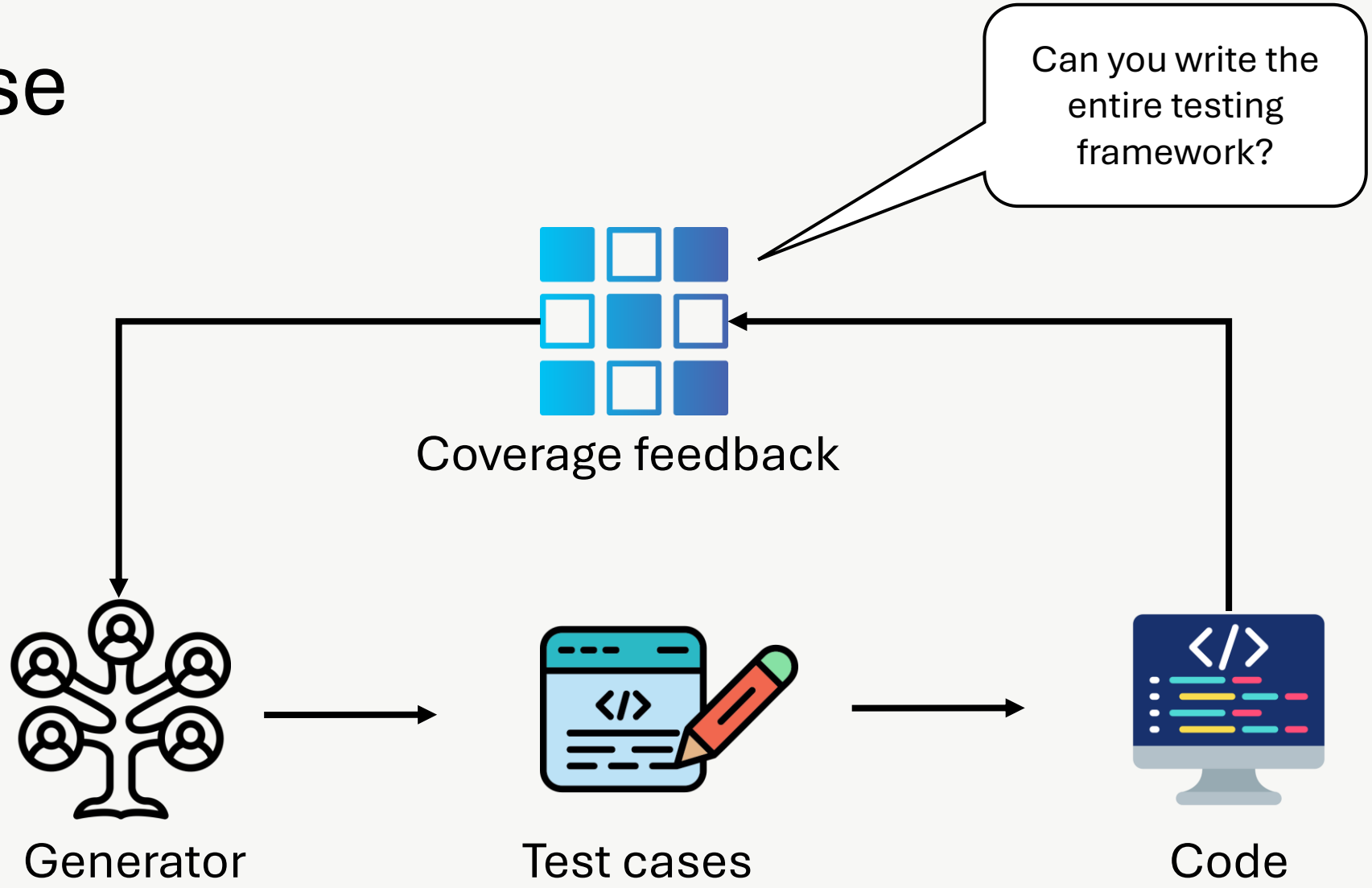


LLMs reason and analyze the target program, then generate the mutators.



MetaMut: https://conngli.github.io/pdfs/metamut_asplos24.pdf

Exercise


Can you write the entire testing framework?
Coverage feedback
Generator
Test cases
Code

LibAFL

`LibAFL`, the fuzzer library

Advanced Fuzzing Library - Slot your own fuzzers together and extend their features using Rust.



`LibAFL` is a collection of reusable pieces of fuzzers, written in Rust, it gives you many of the benefits of an off-the-shelf fuzzer, while being completely customizable. Some highlight features currently include:

- `fast`: We do everything we can at compile time, keeping runtime overhead minimal. Users reach 120k execs/sec in frida-mode on a phone (using all cores).
- `scalable`: `Low Level Message Passing`, `LLMP` for short, allows `LibAFL` to scale almost linearly over cores, and via TCP to multiple machines.
- `adaptable`: You can replace each part of `LibAFL`. For example, `BytesInput` is just one potential form input: feel free to add an AST-based input for structured fuzzing, and more.
- `multi platform`: `LibAFL` runs on *Windows*, *macOS*, *iOS*, *Linux*, and *Android*, and more. `LibAFL` can be built in `no_std` mode to inject `LibAFL` into obscure targets like embedded devices and hypervisors.
- `bring your own target`: We support binary-only modes, like Frida-Mode, as well as multiple compilation passes for sourced-based instrumentation. Of course it's easy to add custom instrumentation backends.

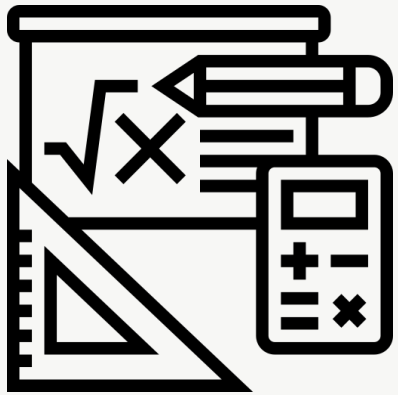
https://aflplus.plus/libafl-book/

Testing

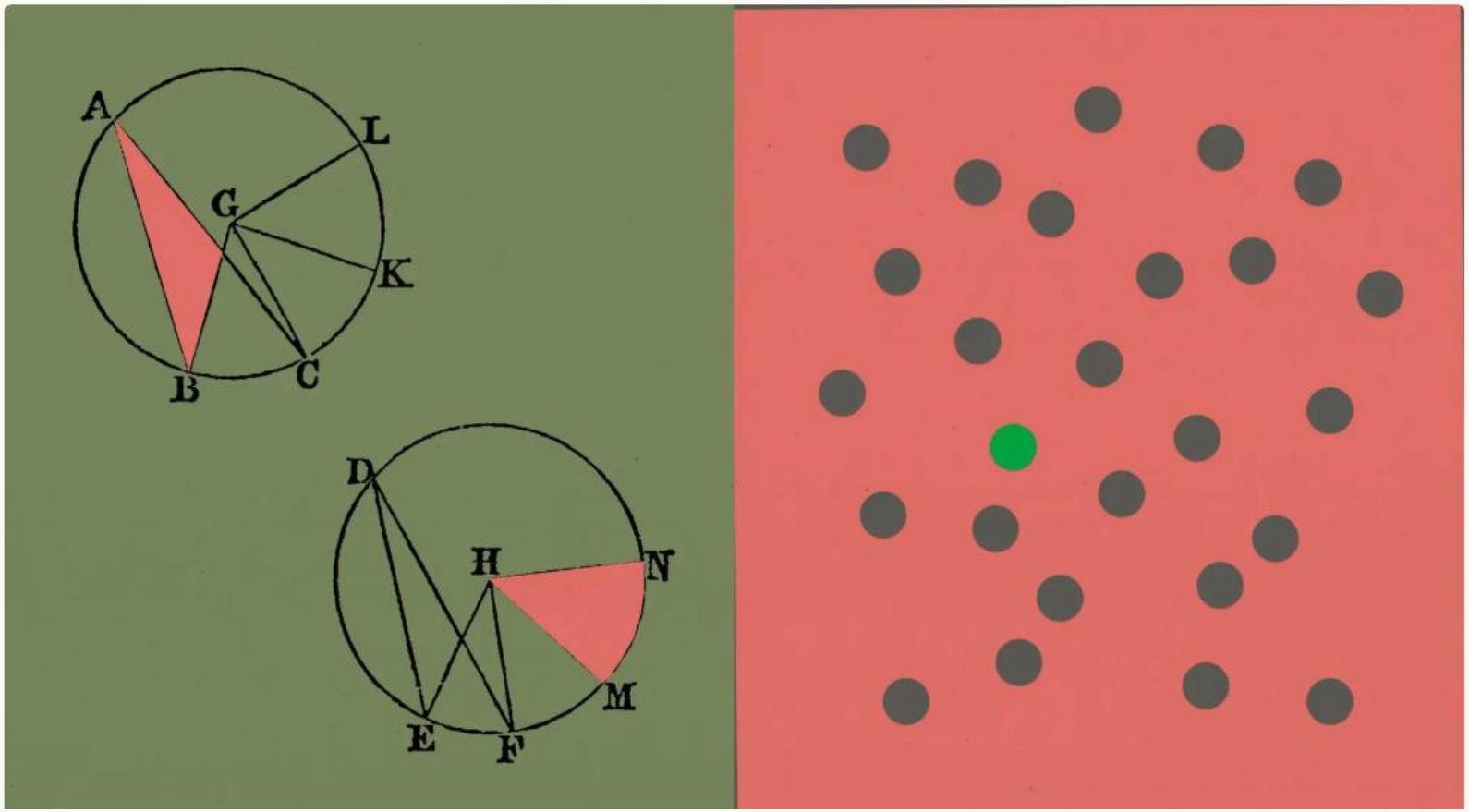
Verification

Let’s see how to validate code

Program Verification



LLMs are making formal software verification feasible at a much larger scale



https://www.anthropic.com/research/formal-verification

Fuzzing

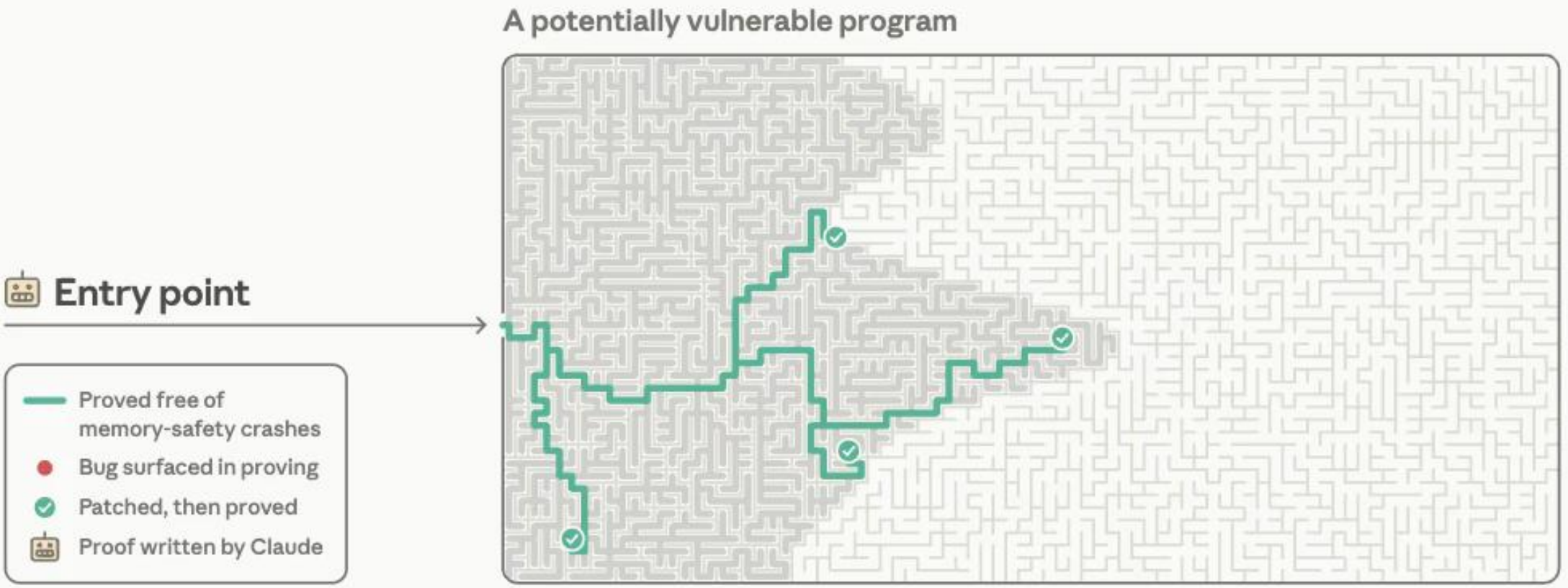
Huge numbers of inputs through one entrypoint



https://www.anthropic.com/research/formal-verification

Formal verification

A test only a proof can pass — every path from this entry point, not the whole program



The proof relies on the formal model of the chip and operating system being right; that model is provided and being open-sourced too.

https://www.anthropic.com/research/formal-verification

Testing is Not Verification

80%

Dev Effort in Testing

Edsger W. Dijkstra famously said:
"Program testing can be used to show the presence of bugs, but never to show their absence!“
Formal verification uses mathematical proofs to guarantee correctness for **all possible inputs**.

Hoare Logic: Pre & Post Conditions

- **Pre-condition (P):** The state of the program before execution.
- **Command (C):** The code block being verified.
- **Post-condition (Q):** The guaranteed state after execution.

$$\{ P \} C \{ Q \}$$

If P is true before C runs, and C successfully terminates, then Q is guaranteed to be true afterward.

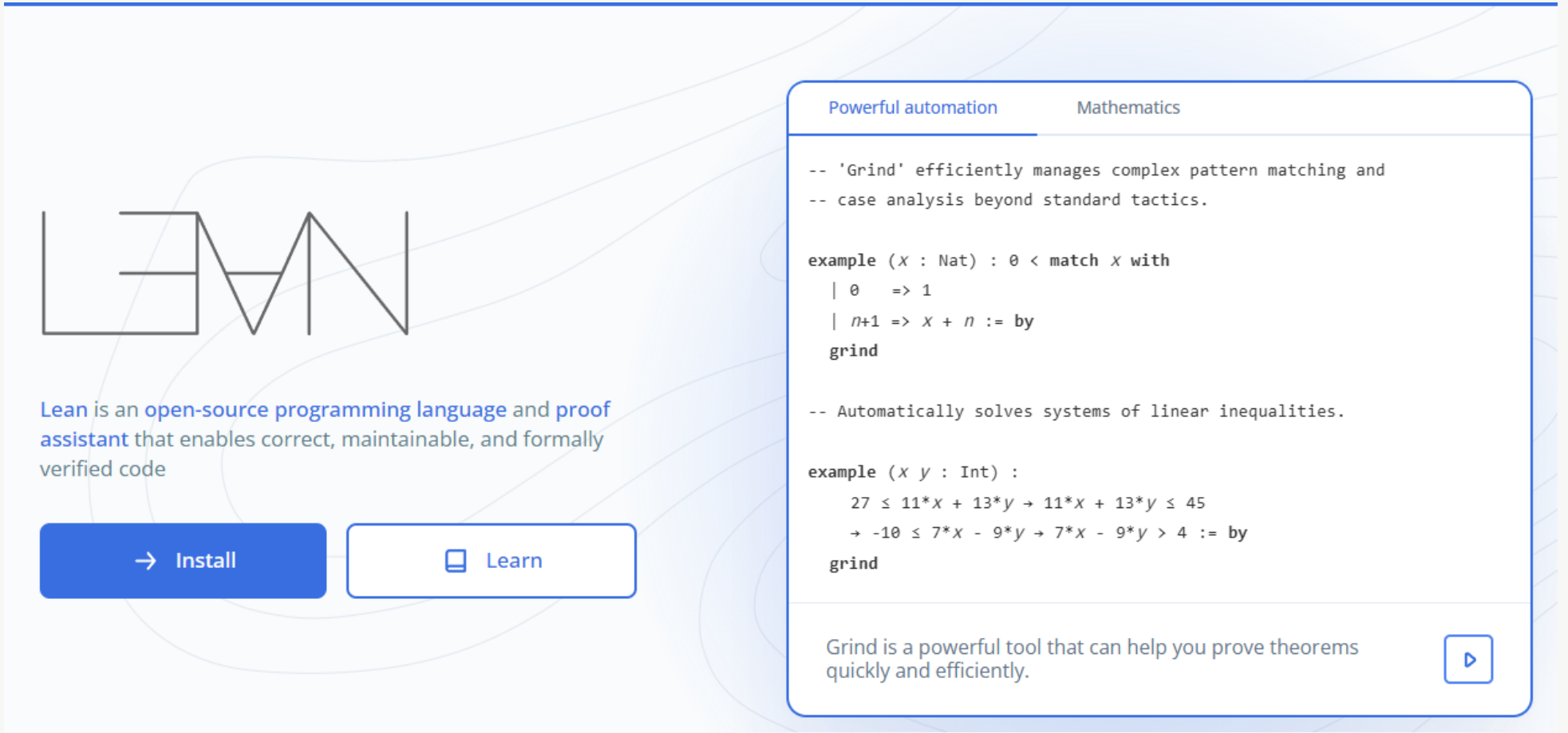
Example

- **Precondition (P):** $n \geq 0$(The input must be non-negative).
- **Postcondition (Q):** $s = \frac{n(n+1)}{2}$(The final sum matches the mathematical formula).

```
i := 0;
s := 0;
while i < n do
    i := i + 1;
    s := s + i;
```

A Practical Tool

Lean 4



https://lean-lang.org/

```
-- Precondition: every element appears 1 or 2 times; exactly one is unique
def FindSingleNumber_precond (nums : List Int) : Prop :=
  let numsCount := nums.map (fun x => nums.count x)
  numsCount.all (fun c => c = 1 ∨ c = 2) ∧ numsCount.count 1 = 1
```

How to derive precondition?

```
-- Implementation: find the unique element that appears exactly once
def filterlist (x : Int) (nums : List Int) : List Int :=
  nums.filter (fun y => y == x)

def FindSingleNumber (nums : List Int)
    (h : FindSingleNumber_precond nums) : Int :=
  match nums.filter (fun x => (filterlist x nums).length == 1) with
  | x :: _ => x
  | []     => 0
```

How to map the implementation?

```
-- Postcondition: result is the unique element; others appear twice
def FindSingleNumber_postcond (nums : List Int) (result : Int)
    (h : FindSingleNumber_precond nums) : Prop :=
  nums.length > 0 ∧ (filterlist result nums).length = 1 ∧
  ∀ (x : Int), x ∈ nums → x = result ∨ (filterlist x nums).length = 2
```

How to derive postcondition?

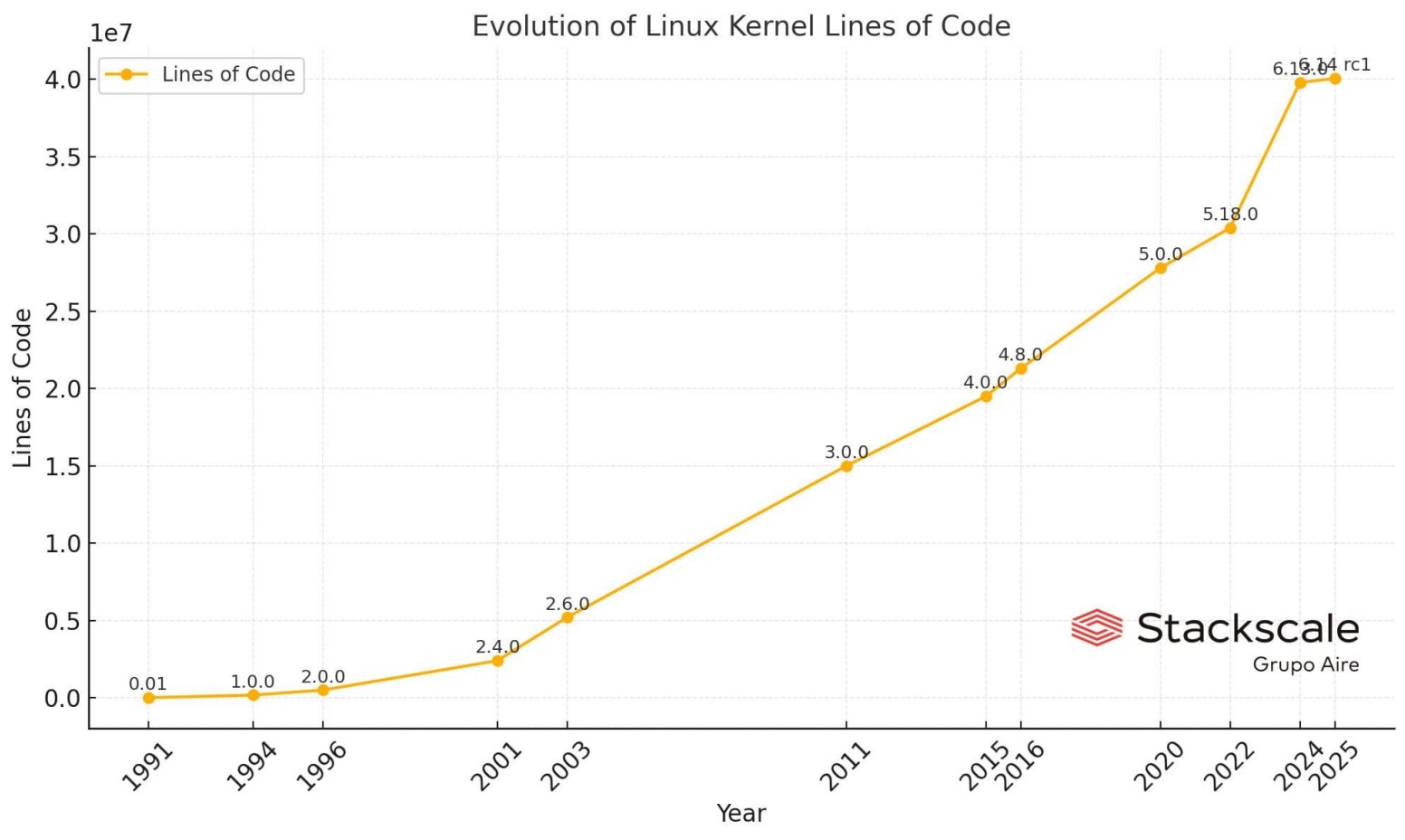
```
-- Verification goal (proof to be synthesized automatically)
theorem FindSingleNumber_spec (nums : List Int)
    (h : FindSingleNumber_precond nums) :
    FindSingleNumber_postcond nums (FindSingleNumber nums h) h := by
  sorry
```

How to verify the implementation?

https://arxiv.org/pdf/2603.19329

Challenges

Real-world code is too complex to be verified



https://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code/

We Have Large Language Models

Opportunities



1. Specification

Generating formal pre/post conditions from natural language descriptions.



2. Prove

Finding complex loop invariants that "explain" why a loop is correct.



3. Repair

Fixing programs or proofs based on feedback from the formal verifier.

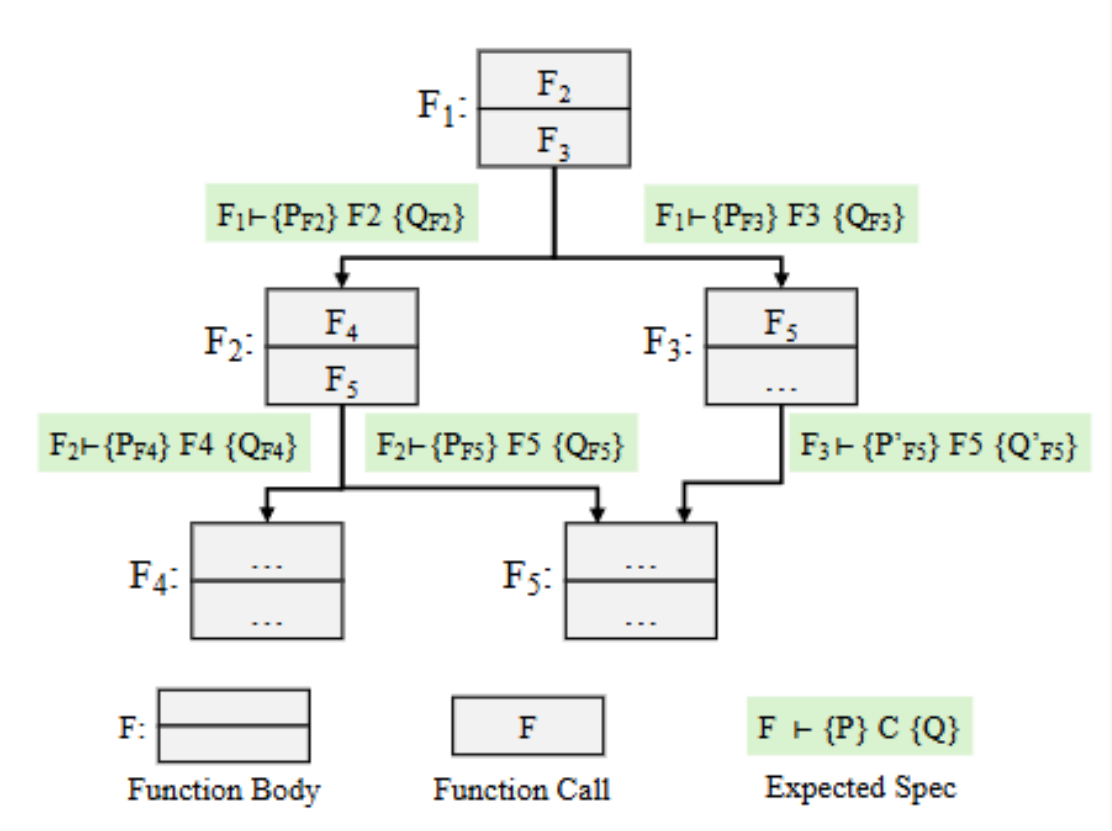
Documents

To reverse the effect of the Average filter after decompression, output the following value:

```
Average(x) + floor((Raw(x-bpp)+Prior(x))/2)
```

https://www.rfc-editor.org/info/rfc2083/

Top-Down Inference



FM-Agent: https://arxiv.org/pdf/2604.11556v1

Top-Down Inference

Pre-condition:
- "s" is a non-empty string slice.
- "gnu_extensions" reflects the active language mode.

Post-condition:
- Returns Some(token_kind) if "s" is a reserved keyword.
  The keyword set covers C standard keywords for C89 through C23, ...
- Returns None for all other identifiers.

FM-Agent: https://arxiv.org/pdf/2604.11556v1

Turning Tests Into Proofs

To address this for the code in the open-source libraries that we sought to prove, we take the approach of automatically converting their preexisting compilable test suites into a spec by encoding the rule "this test suite always passes," using a general model of what it means to execute any RISC-V-compiled binary on Linux. Historically, proving such specs was intractable, given the inherent complexity of writing a proof that can encompass and make an argument over all possible behaviors and interactions of tens of thousands of assembly instructions over any input (and the fact that proofs tend to be extremely large as a result). This is where today's LLMs come in: churning through millions of lines of code in pursuit of a well-defined goal is a task that LLMs are extremely well suited for.

https://www.anthropic.com/research/formal-verification

The Gap

- LLMs are super powerful at solving problems
- The remaining huge gap is human intention and specifications

Opportunities



1. Specification

Generating formal pre/post conditions from natural language descriptions.



2. Prove

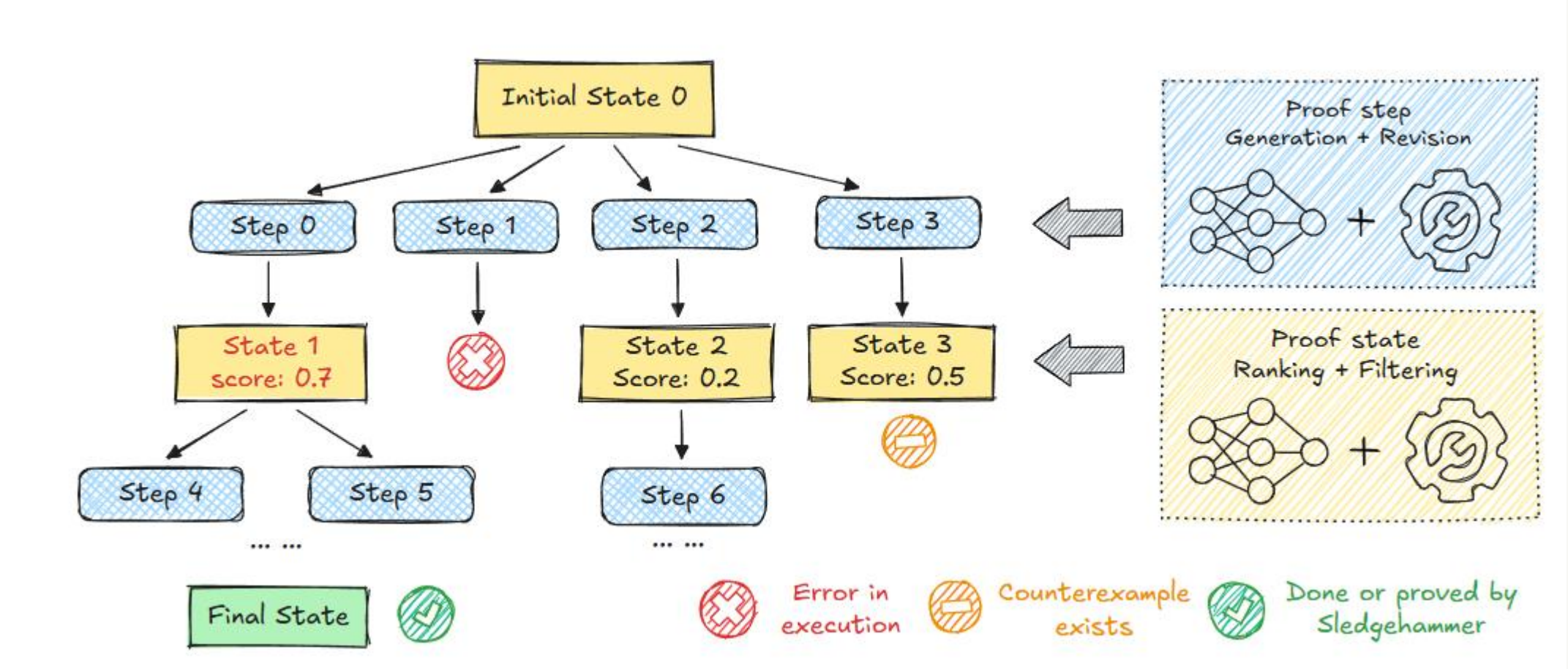
Finding complex loop invariants that "explain" why a loop is correct.



3. Repair

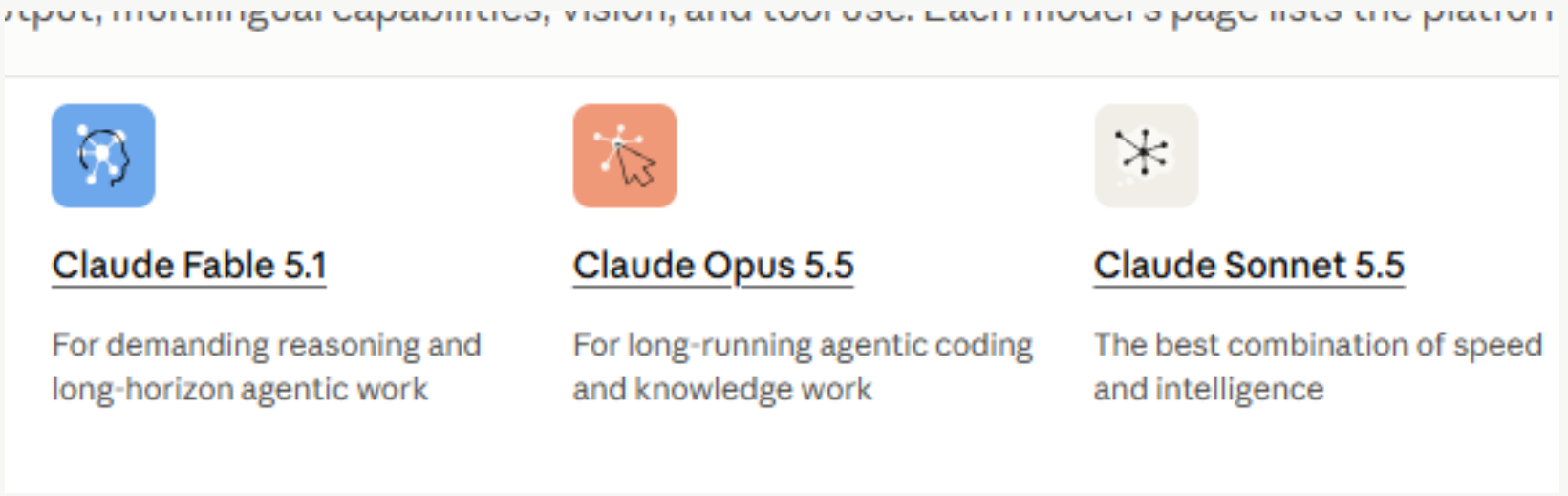
Fixing programs or proofs based on feedback from the formal verifier.

Decomposition: CoT

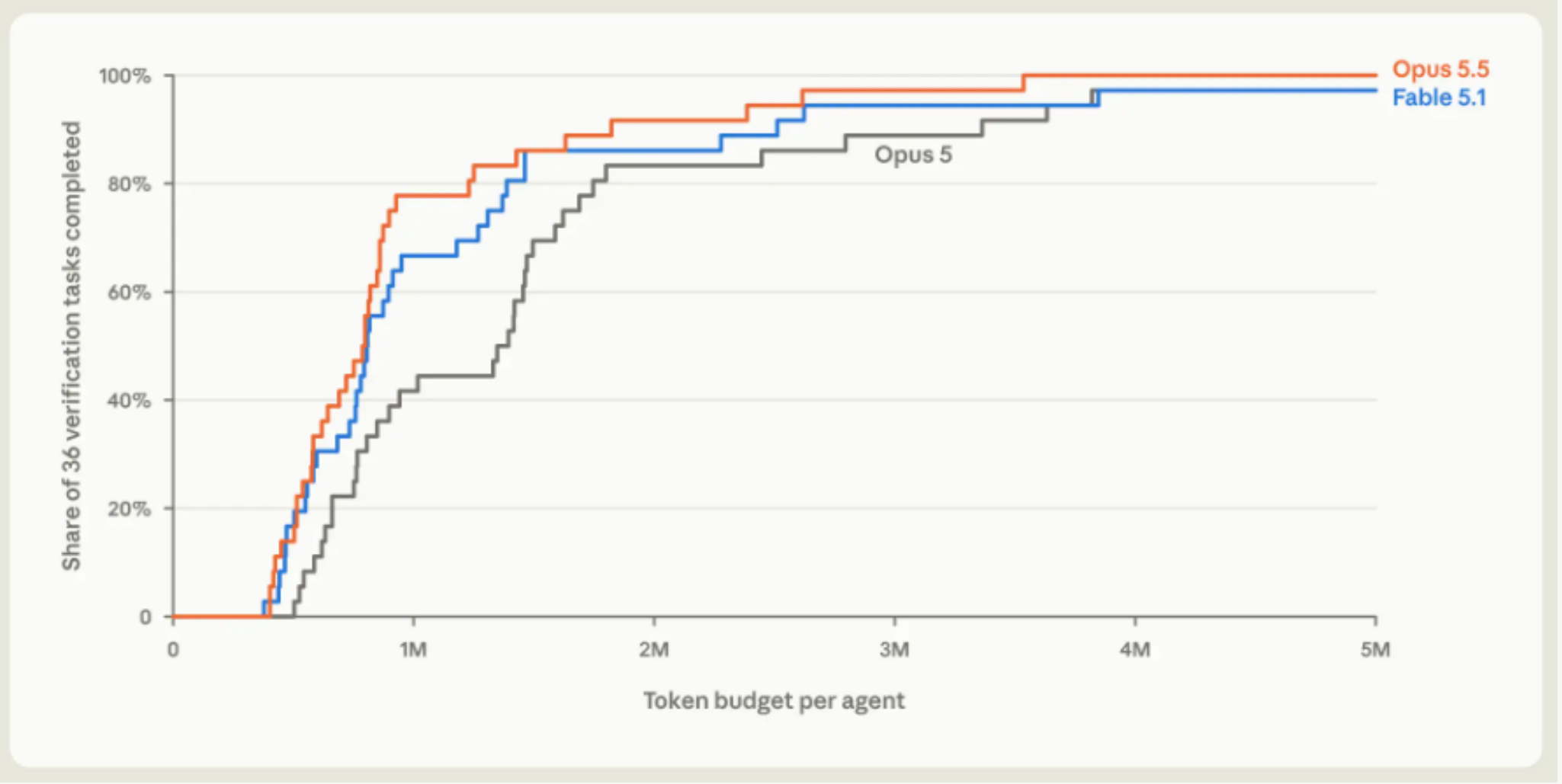


https://arxiv.org/pdf/2603.19715

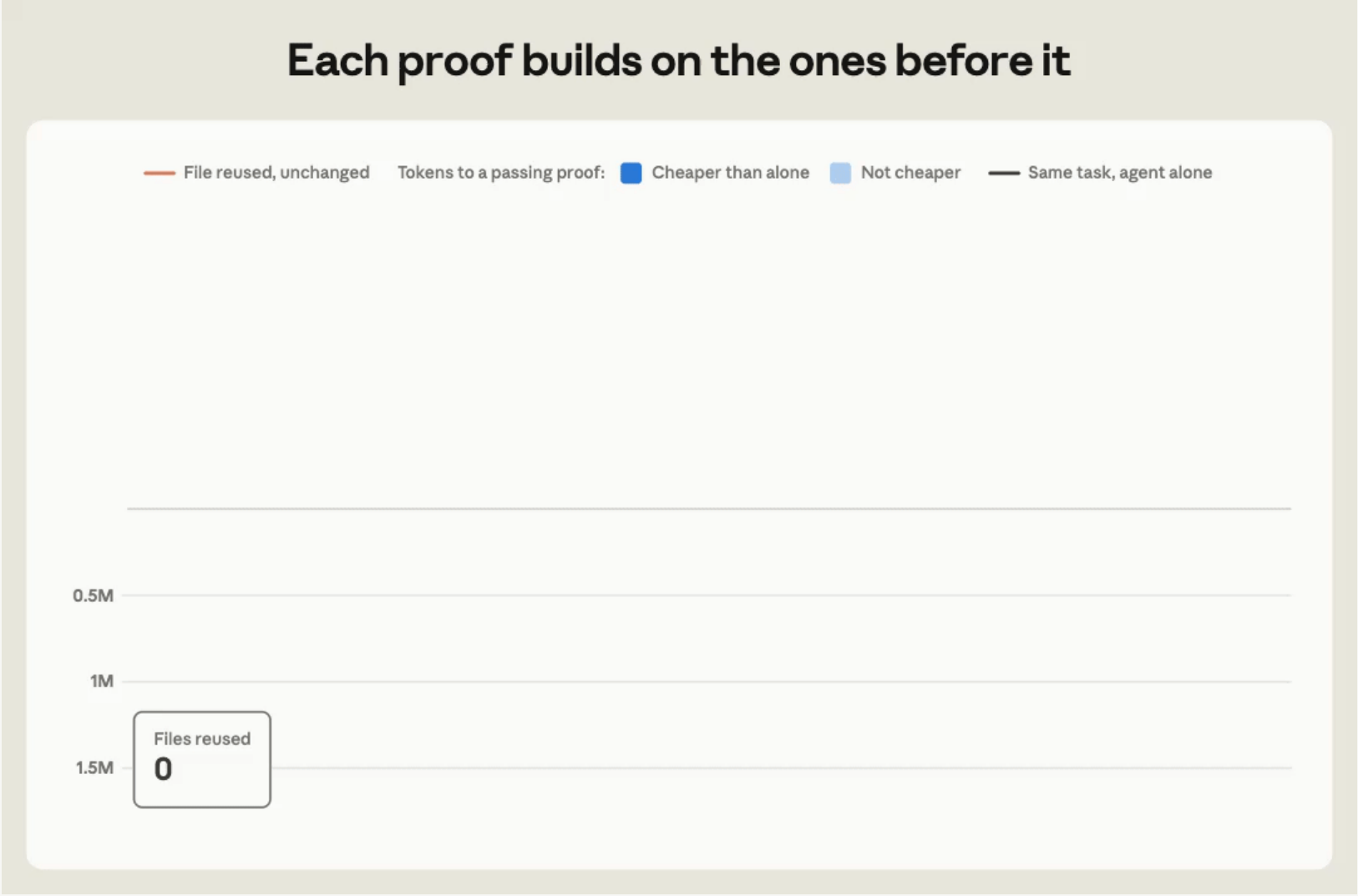
LLMs


Claude Fable 5.1
For demanding reasoning and long-horizon agentic work
Claude Opus 5.5
For long-running agentic coding and knowledge work
Claude Sonnet 5.5
The best combination of speed and intelligence

Token Consumption



Proof Reuse


Each proof builds on the ones before it
File reused, unchanged
Tokens to a passing proof:
Cheaper than alone
Not cheaper
Same task, agent alone
0.5M
1M
1.5M
Files reused
0

Opportunities



1. Specification

Generating formal pre/post conditions from natural language descriptions.



2. Prove

Finding complex loop invariants that "explain" why a loop is correct.



3. Repair

Fixing programs or proofs based on feedback from the formal verifier.

The "Verify-and-Repair" Loop

1. **Propose:** LLM generates a candidate invariant.
2. **Check:** SMT Solver attempts to prove it.
3. **Feedback:** If it fails, the solver provides a *counterexample*.
4. **Repair:** LLM uses the counterexample to refine the invariant.

LTL Syntax

Let Σ be a finite alphabet. A *word* w over Σ is an infinite sequence of letters $a_0a_1a_2a_3....$ with $a_i \in \Sigma$ for all $i \geq 0$, and a language is a set of words. A *finite word* is a finite sequence of letters. As usual, the set of all words (finite words) is denoted $\Sigma^\omega(\Sigma^*)$.

Definition 1 (LTL syntax). LTL formula φ over a set Ap of atomic propositions are constructed by the following syntax:

$$\begin{aligned}\varphi ::= tt \mid ff \mid a \mid \neg a \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi U\varphi \mid \varphi W\varphi \mid \varphi R\varphi \mid \\ \varphi M\varphi \mid \varphi \rightarrow \varphi \mid \varphi \iff \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \oplus \varphi\end{aligned} \tag{1}$$

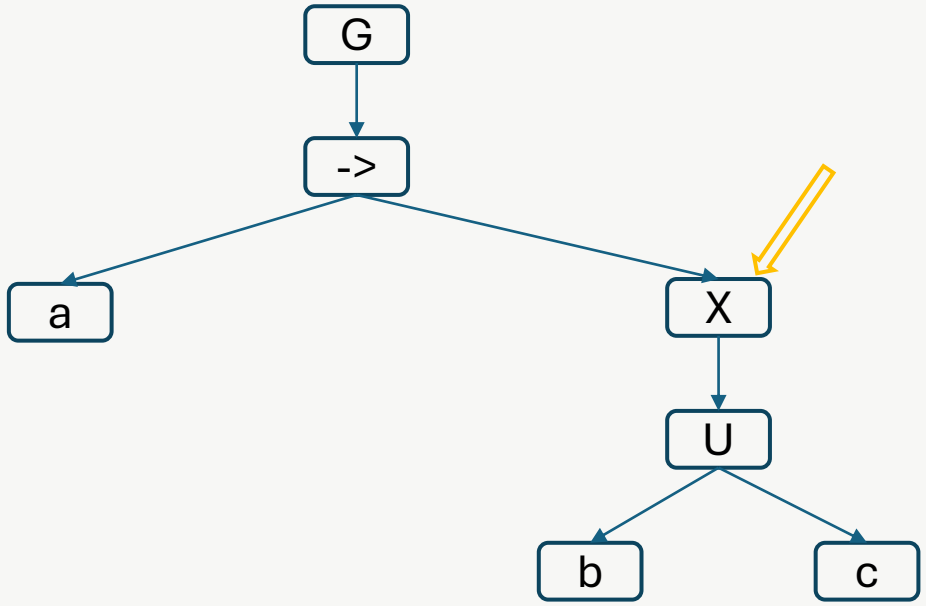
where $a \in Ap$ is an atomic proposition and X, F, G, U, W, R, M are the next, finally, globally, (strong) until, weak until, (weak) release, and strong release operators, respectively.

Depending the number of operands, the operators can be classified into:

$$\begin{aligned}&\textit{unary operator} ::= X \mid F \mid G \mid \neg \\ &\textit{binary operator} ::= U \mid W \mid R \mid M \mid \rightarrow \mid \iff \mid \wedge \mid \vee \mid \oplus\end{aligned} \tag{2}$$

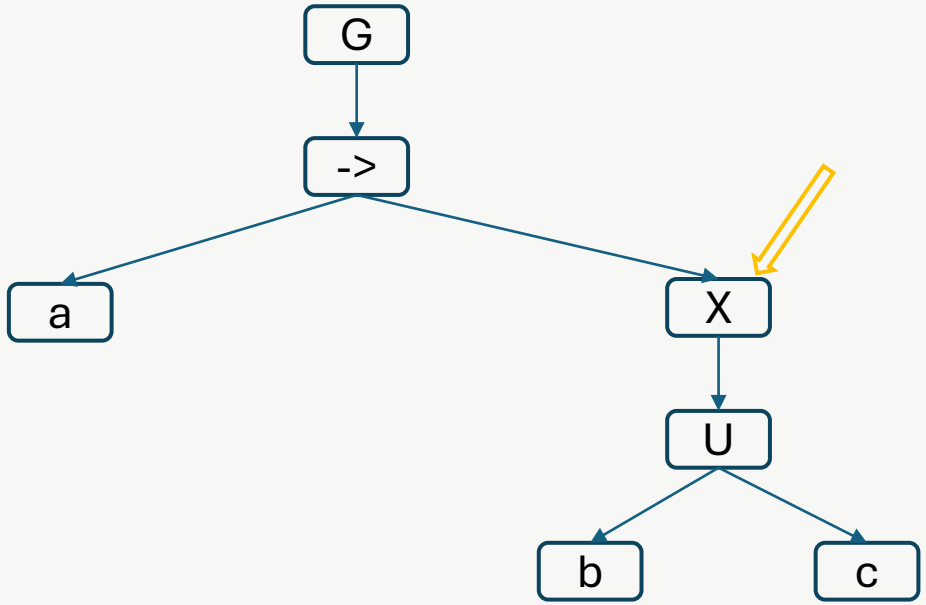
Relax: G(a -> X(b U c))

- If a counterexample found, we relax the formula aiming to accept the counterexample.
- Randomly select an operator and change it.
- For example: X -> F
- “a d b c” is accepted by G(a -> F(b U c)), not G(a -> X(b U c)).



Restrict: G(a -> X(b U c))

- If no counterexample found after a period, we restrict the formula aiming to expose more counterexamples.
- Randomly select an operator and change it.
- For example: X -> G



Relationship

We let $\mathcal{L}(\varphi) := \{w \in (2^{Ap})^{\omega} : w \models \varphi\}$ denote the language of LTL formula φ. For any LTL formulas φ and ψ, we have the following relations:

$$
\begin{aligned}
&\mathcal{L}(ff) \subset \mathcal{L}(G\varphi) \subset \mathcal{L}(X\varphi) \subset \mathcal{L}(F\varphi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi M \psi) \subset \mathcal{L}(\varphi R \psi) \subset \mathcal{L}(\varphi U \psi) \subset \mathcal{L}(\varphi W \psi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(\varphi \oplus \psi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi M \psi) \subset \mathcal{L}(\varphi R \psi) \subset \mathcal{L}(\varphi \rightarrow \psi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi \Longleftrightarrow \psi) \subset \mathcal{L}(\varphi \rightarrow \psi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt) \\
&\mathcal{L}(ff) \subset \mathcal{L}(G\varphi) \subset \mathcal{L}(\varphi) \subset \mathcal{L}(F\varphi) \subset \mathcal{L}(tt)
\end{aligned}
\tag{3}
$$

Relax and Restrict

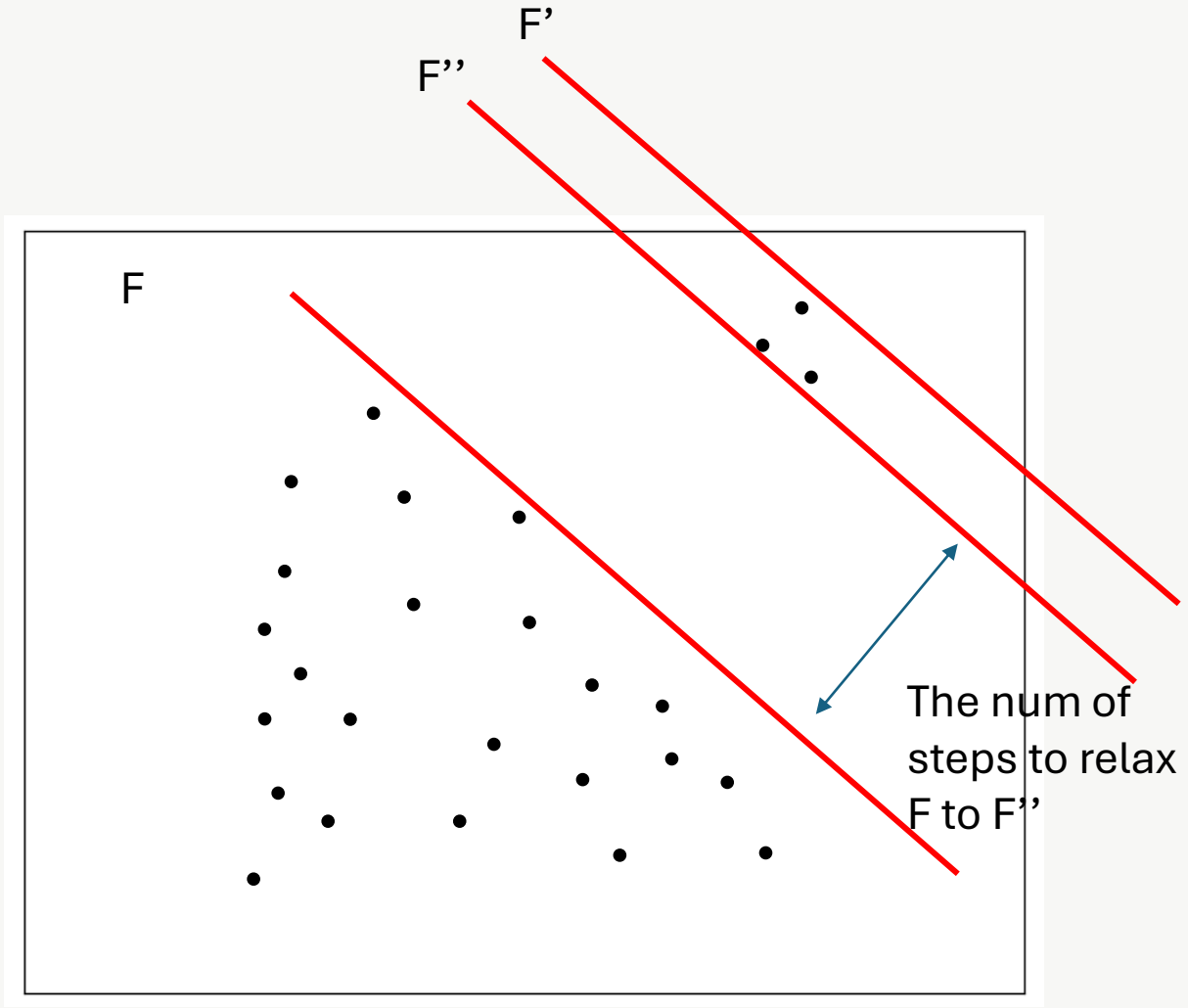
Relax: 

$$\mathcal{L}(ff) \subset \mathcal{L}(G\varphi) \subset \mathcal{L}(X\varphi) \subset \mathcal{L}(F\varphi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi M \psi) \subset \mathcal{L}(\varphi R \psi) \subset \mathcal{L}(\varphi U \psi) \subset \mathcal{L}(\varphi W \psi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(\varphi \oplus \psi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi M \psi) \subset \mathcal{L}(\varphi R \psi) \subset \mathcal{L}(\varphi \rightarrow \psi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi \iff \psi) \subset \mathcal{L}(\varphi \rightarrow \psi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(\varphi \wedge \psi) \subset \mathcal{L}(\varphi) \subset \mathcal{L}(\varphi \vee \psi) \subset \mathcal{L}(tt)$$
$$\mathcal{L}(ff) \subset \mathcal{L}(G\varphi) \subset \mathcal{L}(\varphi) \subset \mathcal{L}(F\varphi) \subset \mathcal{L}(tt)$$

Restrict: 

Anomaly Detection

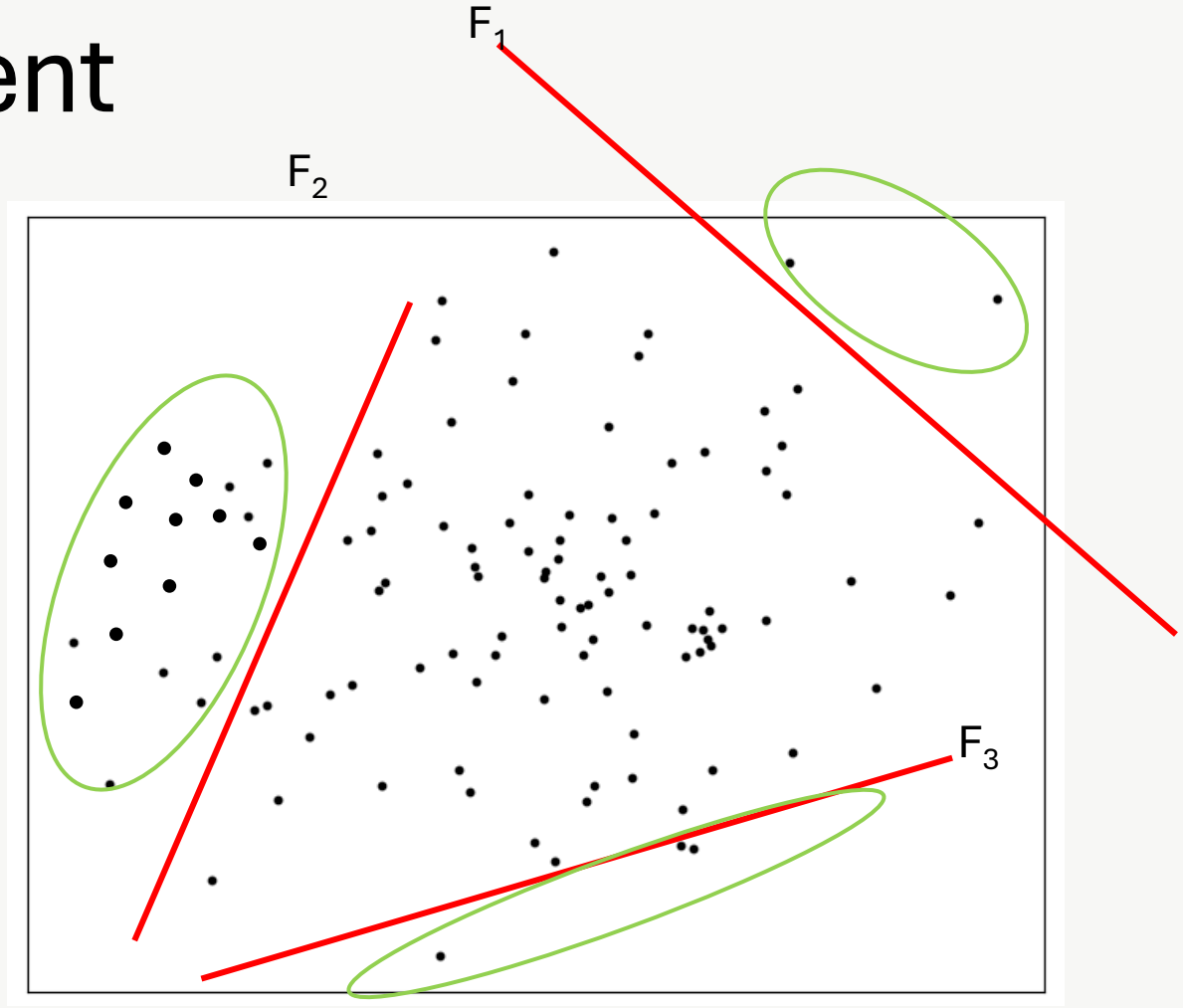
- For each LTL property F, we gradually refine it to F’ which satisfy all traces. In this process, we record F’’ which is the last LTL property that the traces that violate F still violate.
- It provides a logic-based deviation measurement.



Event trace space of a model
Each dot is an event trace

LTL Property Refinement

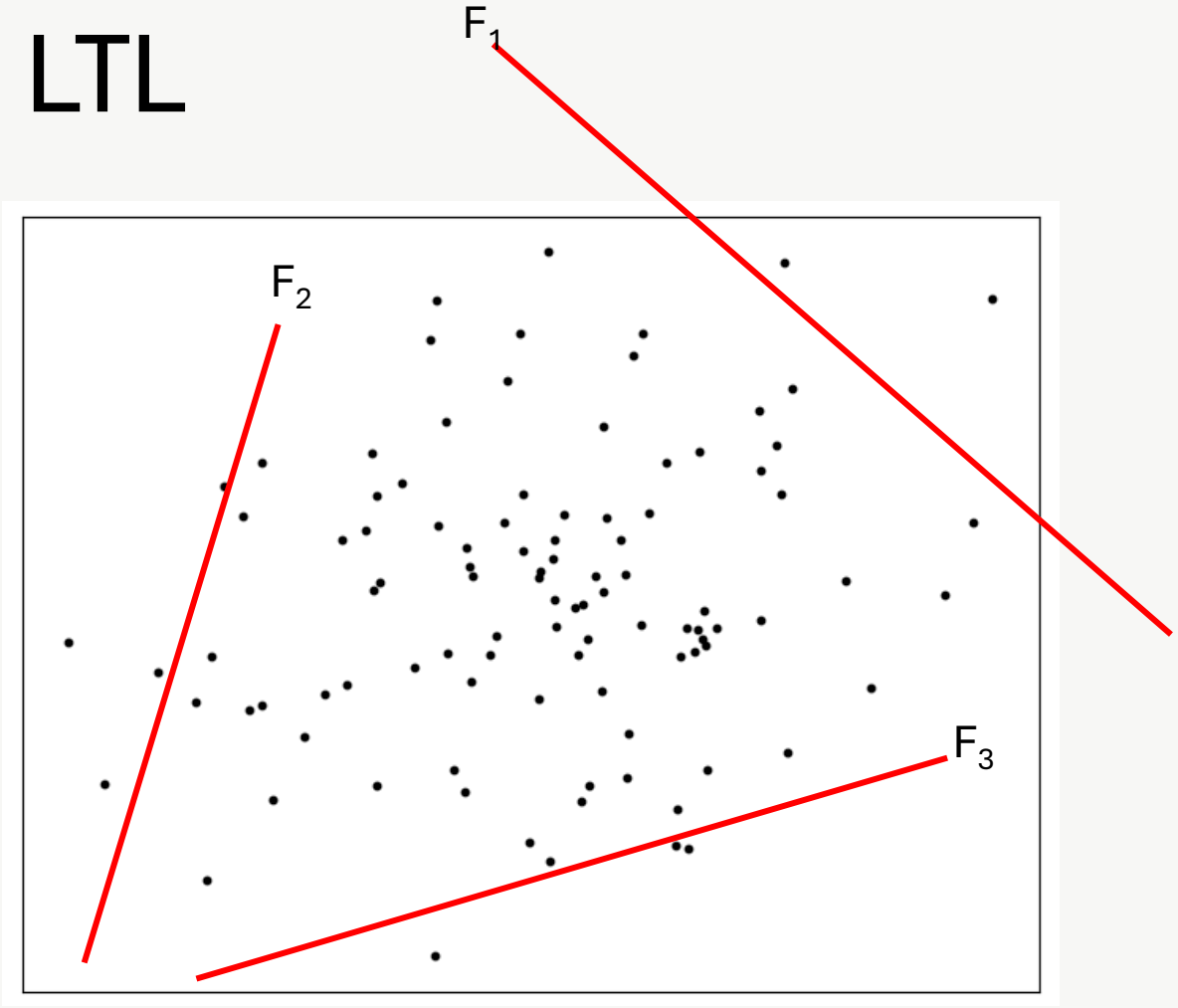
- If no suspicious anomaly behaviors, we relax the LTL properties to the LTL properties that less than X% traces violate.
- If refinement fails, remove this LTL property.



Event trace space of a model
Each dot is an event trace

Fuzzing/Verification -> LTL

- Guide LTL property generation towards 1) the boundary of discovered event traces, 2) covering more APs.
- Use verification to look for counter examples



Event trace space of a model
Each dot is an event trace



香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Questions

Next class: Validation Exercise