

Towards Provably Correct, Real-World Code Generation

Pranjal Aggarwal, Natalie Neamtu

Carnegie Mellon University



**Carnegie
Mellon
University**

Explosion of AI-Written Code

Do Users Write More Insecure Code with AI Assistants?

Neil Perry*
Stanford University

Megha Srivastava*
Stanford University

Deepak Kumar
Stanford University / UC
San Diego

Dan Boneh
Stanford University

ABSTRACT

AI code assistants have emerged as powerful tools that can aid in the software development life-cycle and can improve developer

use them. Such work is important in order understand the practical security challenges introduced by AI-powered code-assistants and the ways users prompt the AI systems to inadvertently cause

*“We find that participants who had access to an AI assistant wrote **significantly less secure** code than those without access to an assistant”*

to be overconfident about security flaws in their code. To better inform the design of future AI-based code assistants, we release our user-study apparatus to researchers seeking to build on our work.

CCS CONCEPTS

• Security and privacy → Human and societal aspects of se-

to an AI programming assistant?

- RQ2: Do users trust AI assistants to write secure code?
- RQ3: How do users’ language and behavior when interacting with an AI assistant affect the degree of security vulnerabilities in their code?

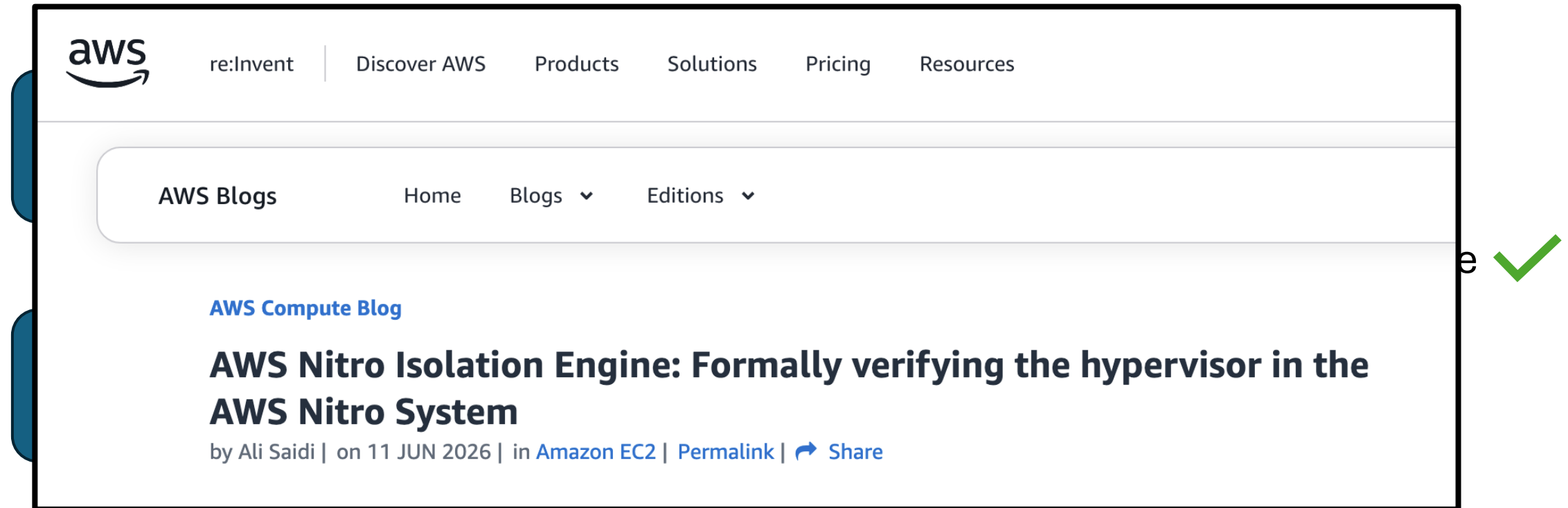
Participants with access to an AI assistant wrote insecure code

Mar 14, 2025, 2:27 AM ET

<http://dx.doi.org/10.1145/3576915.3623157>

Verified Code Generation

How do we use agents to generate code in a trustworthy way?



Verus: Automated Proofs for Rust

- Mathematically prove code is correct/reliable/secure
- Focus on fast, deterministic proof automation for systems code



Result: Scaling verification to large, complex systems

- Memory allocator
- Microkernel
- Distributed KV store
- Persistent memory log
- Cryptographic libraries
- ...

✓ Memory safety ✓ Data race freedom
✓ Functional correctness

How Does Verus Work?

Verus allows the programmer to add specifications and proof annotations:

```
fn vec_update_at_idx<T>(
    v: &mut Vec<T>,
    idx: usize,
    t: T
)
  requires
    idx < v.len(),
  ensures
    v[idx] == t,
{
  // ...
  a
  // ...
}
```

Static check, enforced by verifier
(backed by SMT solver)

Erased at runtime



AI Agents Have Proof Generation Capability

- **VeruSAGE**: benchmark for proof generation on real systems code
 - **849** proofs from 8 large-scale Verus projects
- Opus 4.5 + generic coding agent completes **94%** of tasks at a cost of **\$13.7** per task

Challenges remain for end-to-end verified code generation:

- Formalizing English descriptions into specifications
- Cost, maintainability, verification performance

This Talk

- **Verus-SpecBench**: Benchmark for specification writing
- **AlphaVerus** and **Propose, Solve, Verify**: Reinforcement learning for formal verification

Verus-SpecGym: An Agentic Environment for Evaluating Specification Autoformalization

Anmol Agarwal CMU	Natalie Neamtu CMU	Pranjal Aggarwal CMU	
Seungone Kim CMU	Jannis Limperg* Amazon	Cedric Flamant Amazon	Kanna Shimizu Amazon
Bryan Parno CMU		Sean Welleck CMU	

Propose, Solve, Verify: Self-Play Through Formal Verification

Alex Wilf¹ Pranjal Aggarwal¹ Bryan Parno¹
Daniel Fried¹ Louis-Philippe Morency¹ Paul Pu Liang² Sean Welleck¹

AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement

Pranjal Aggarwal¹ Bryan Parno¹ Sean Welleck¹

Verus-SpecBench

Benchmark of 581 specification writing tasks from Codeforces competition programming problems.

- **Objective, cheap evaluation (test cases)**
- Automated task extraction pipeline
- Agentic evaluation environment (**Verus-SpecGym**)

★ Evaluation

- Data: **Codeforces “hacks”** rule out specifications that are too permissive
- Verifier: (expanded!) Verus feature for **executable specifications**

Evaluating Specifications is Hard

Informal specification

“Output a list of all the **even numbers** in the input.”

autoformalization

faithful?

Formal specification

```
forall |n|  
  output.contains(n) ==>  
    n % 2 && input.contains(n)
```

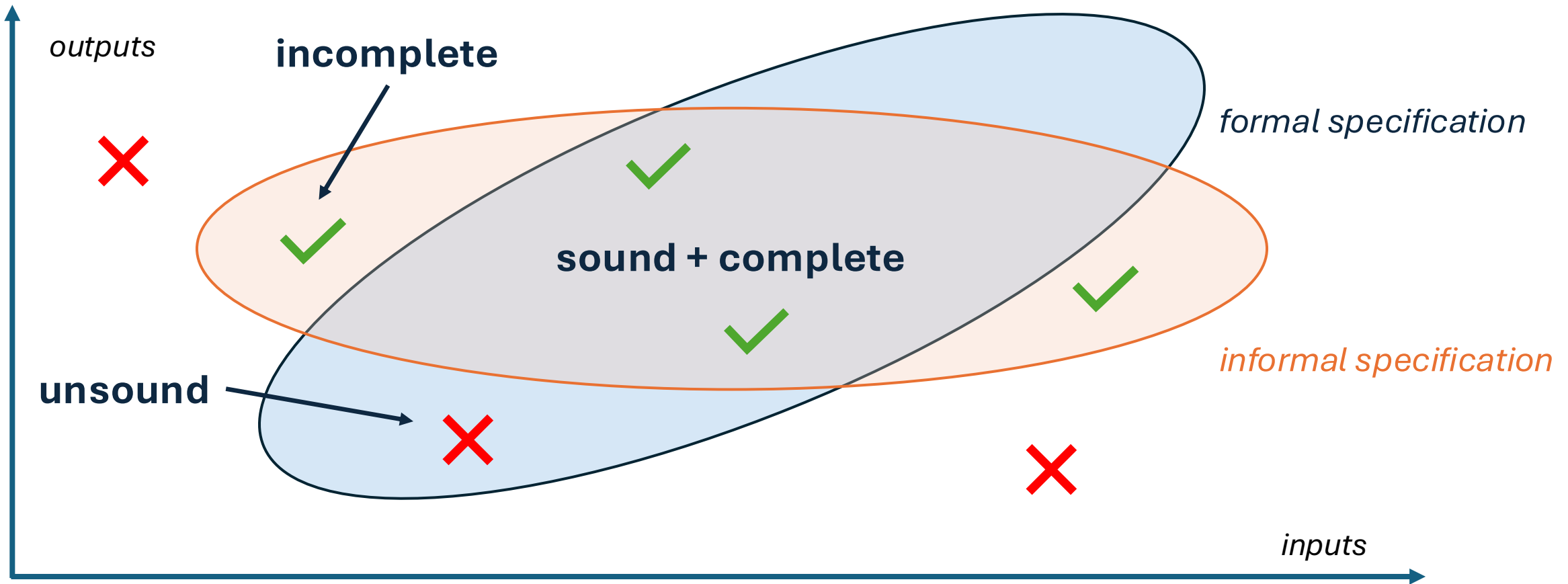


input = [1, 2, 3], output = [2]



input = [1, 2, 3], output = []

Specification Soundness and Completeness



✓ = allowed input, output pair

✗ = disallowed input, output pair

Codeforces Hacks: Soundness Test Cases

- The official Codeforces test suite only provides **completeness** test cases.
- **Hacks**: adversarial test cases for real solutions that pass the official test suite.
 - These become **soundness** test cases.
- Success rate drops **up to 19%** after adding soundness test cases.

Executing Specifications with exec_spec!

- Due to undecidability, the verifier can neither prove nor disprove **15-80%** of test cases.

```
spec fn odd_seq(in: Seq<u8>, out: Seq<u8>) -> bool {  
  forall |n| out.contains(n) ==> /* ... */  
}
```

```
assert(odd_seq([1, 2, 3], [2]));  
assert(!odd_seq([1, 2, 3], [2]));
```

Verus (proof) assertions



verification results:
0 verified, 2 errors

Executing Specifications with exec_spec!

- Verus feature compiles specification code into runnable Rust code.
 - Supports: bounded integers, bounded quantifiers, spec types (map/set/seq)

```
exec_spec! {  
  spec fn odd_seq(/* ... */) {  
    /* ... */  
  }  
}
```

Verus specification

```
assert!(exec_odd_seq(/* ... */));
```

Rust (runtime) assertion

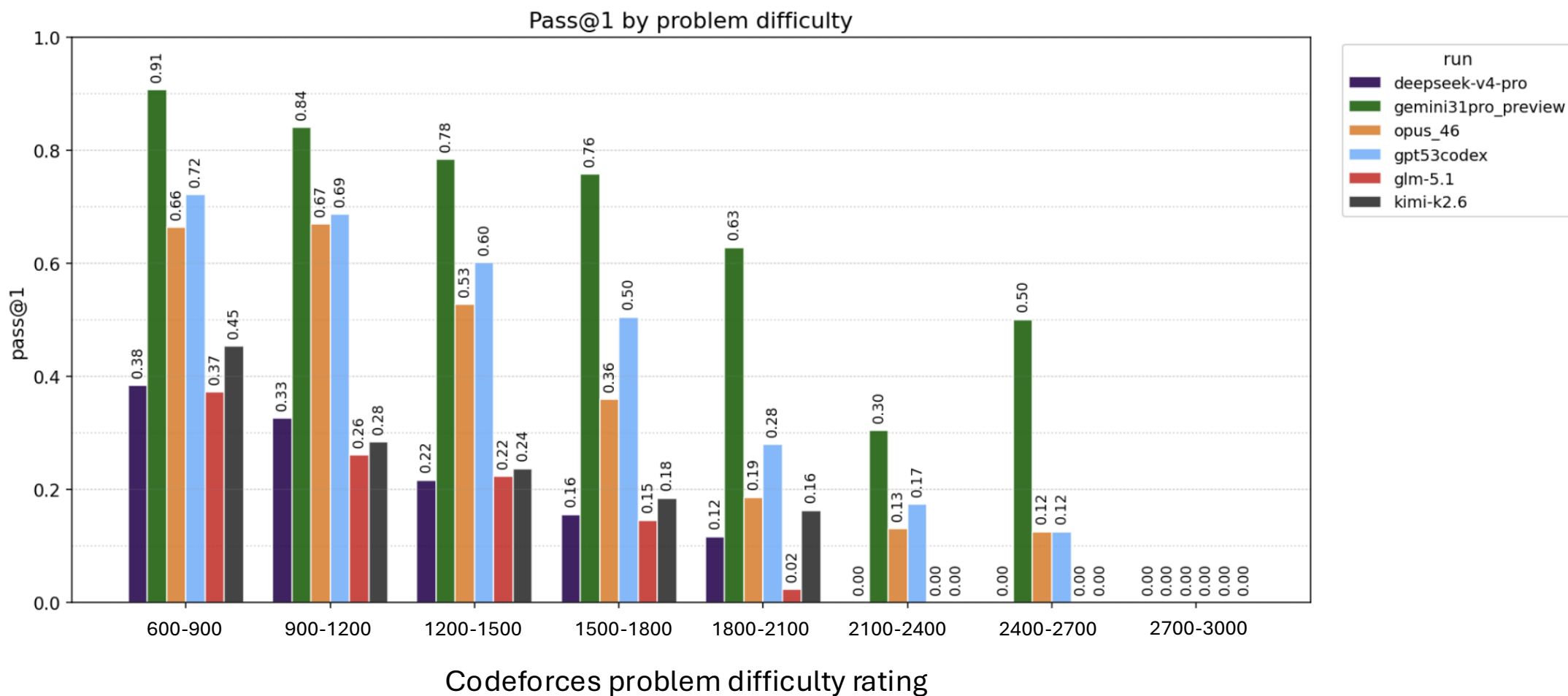
- Upcoming: integration with PBT

Verus-SpecBench Leaderboard

Model	Pass @ 1
gemini-3.1pro	0.778
gpt5.3-codex	0.578
opus4.6	0.511
deepseek-v4pro	0.243
glm-5.1	0.215
kimi-k2.6	0.255

Benchmark repository: <https://github.com/formal-verif-is-cool/verus-spec-gym>

Hardest Problems Still Challenge Best Models

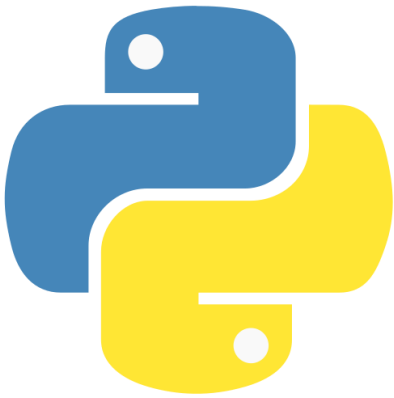


How do we train models to
generate formally verified code?

Main Challenge: **We don't have data**

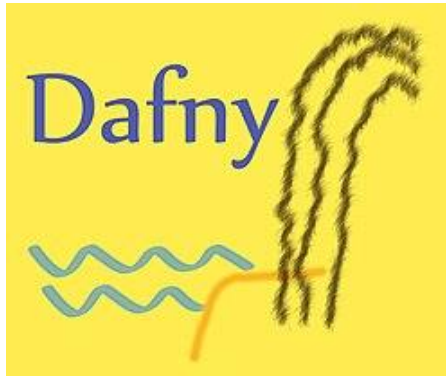
Main Challenge: **We don't have data**

Python



~ Trillion tokens

Dafny



~100M tokens

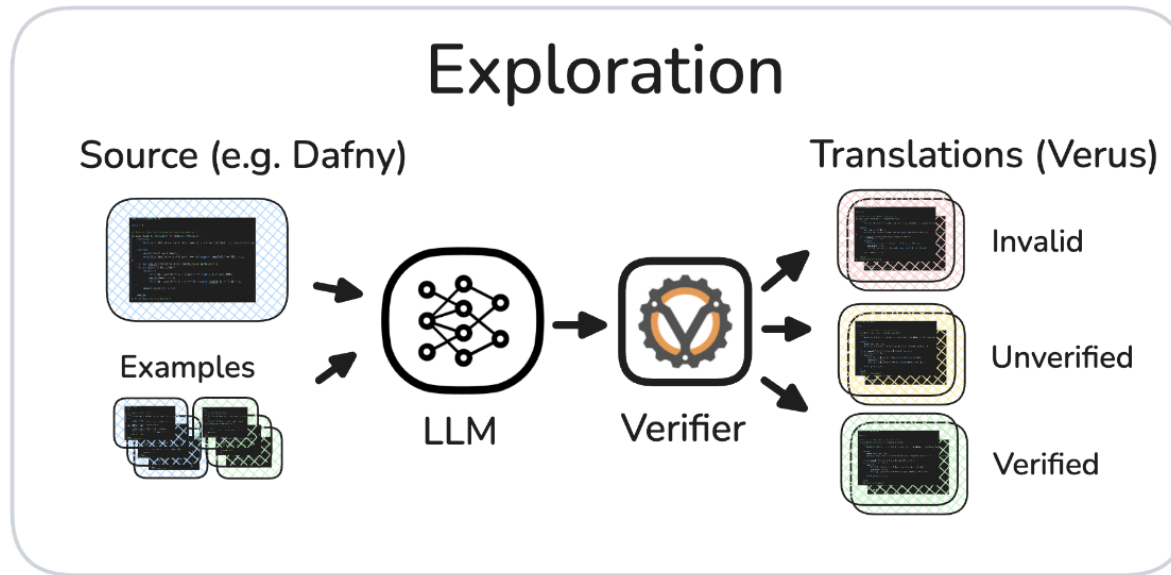
Verus



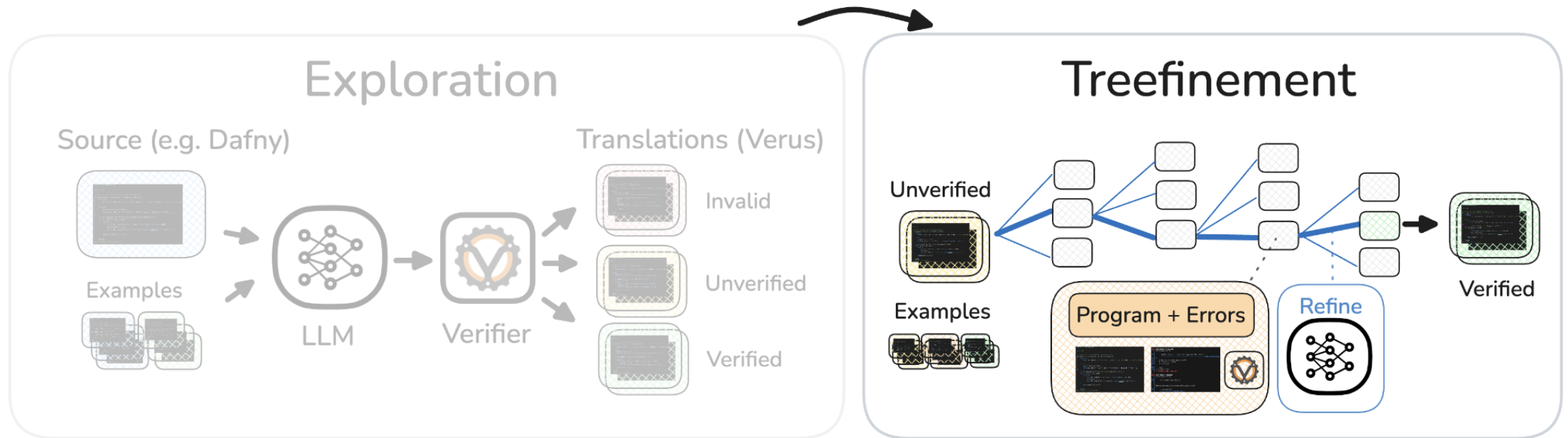
< 10 Repos

AlphaVerus: Iterative Self-Improving Translation

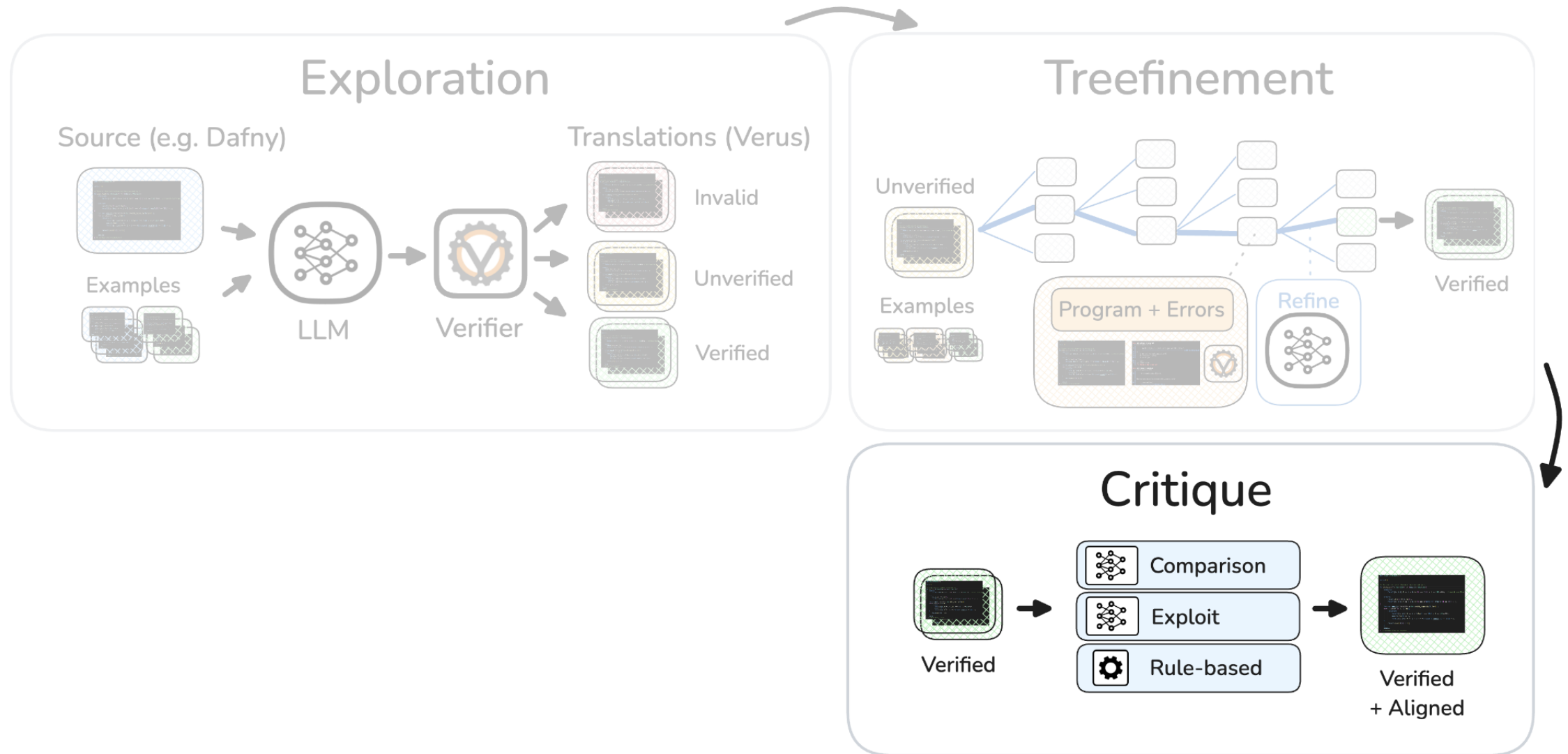
AlphaVerus: Iterative Self-Improving Translation



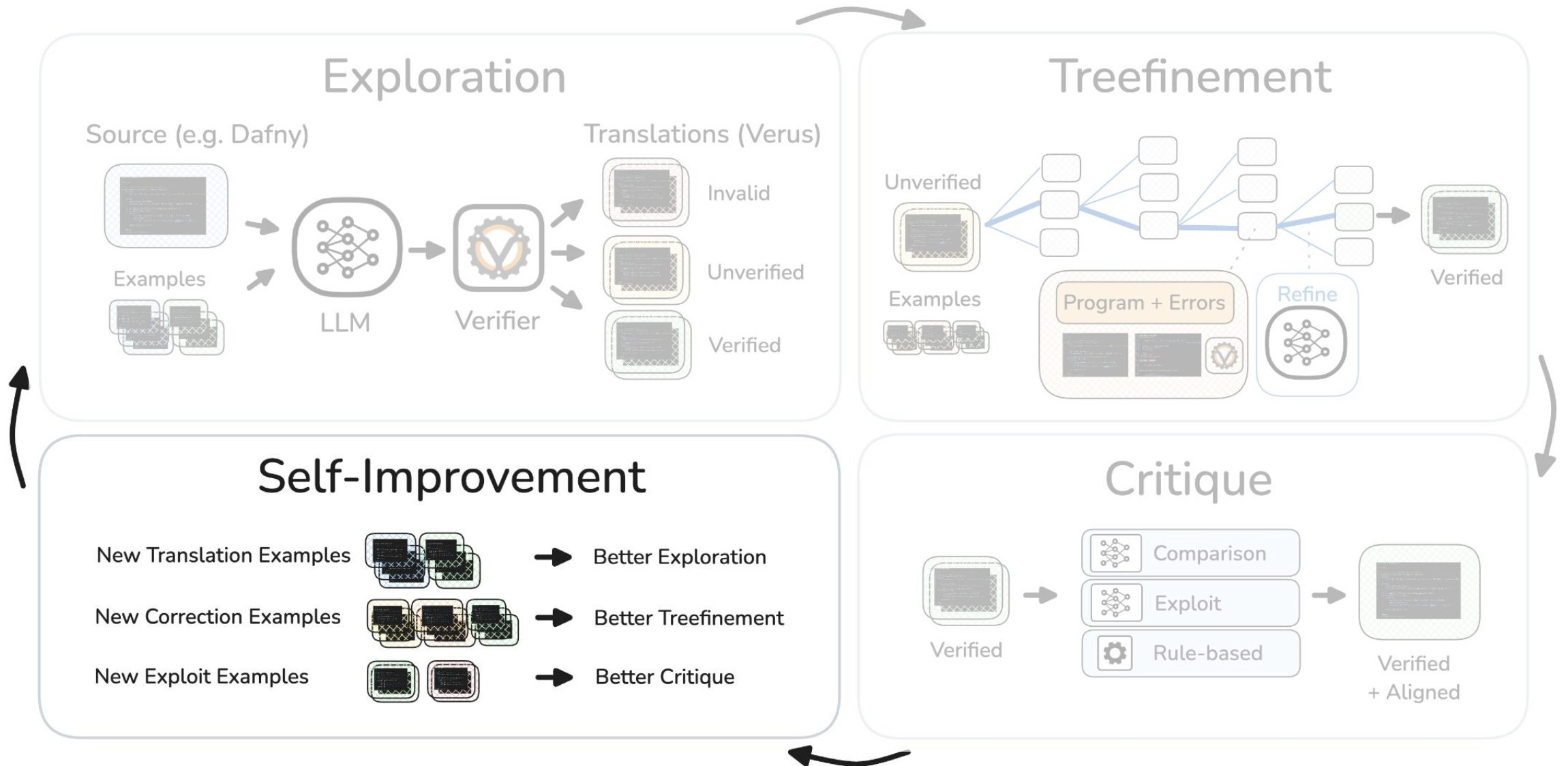
AlphaVerus: Iterative Self-Improving Translation



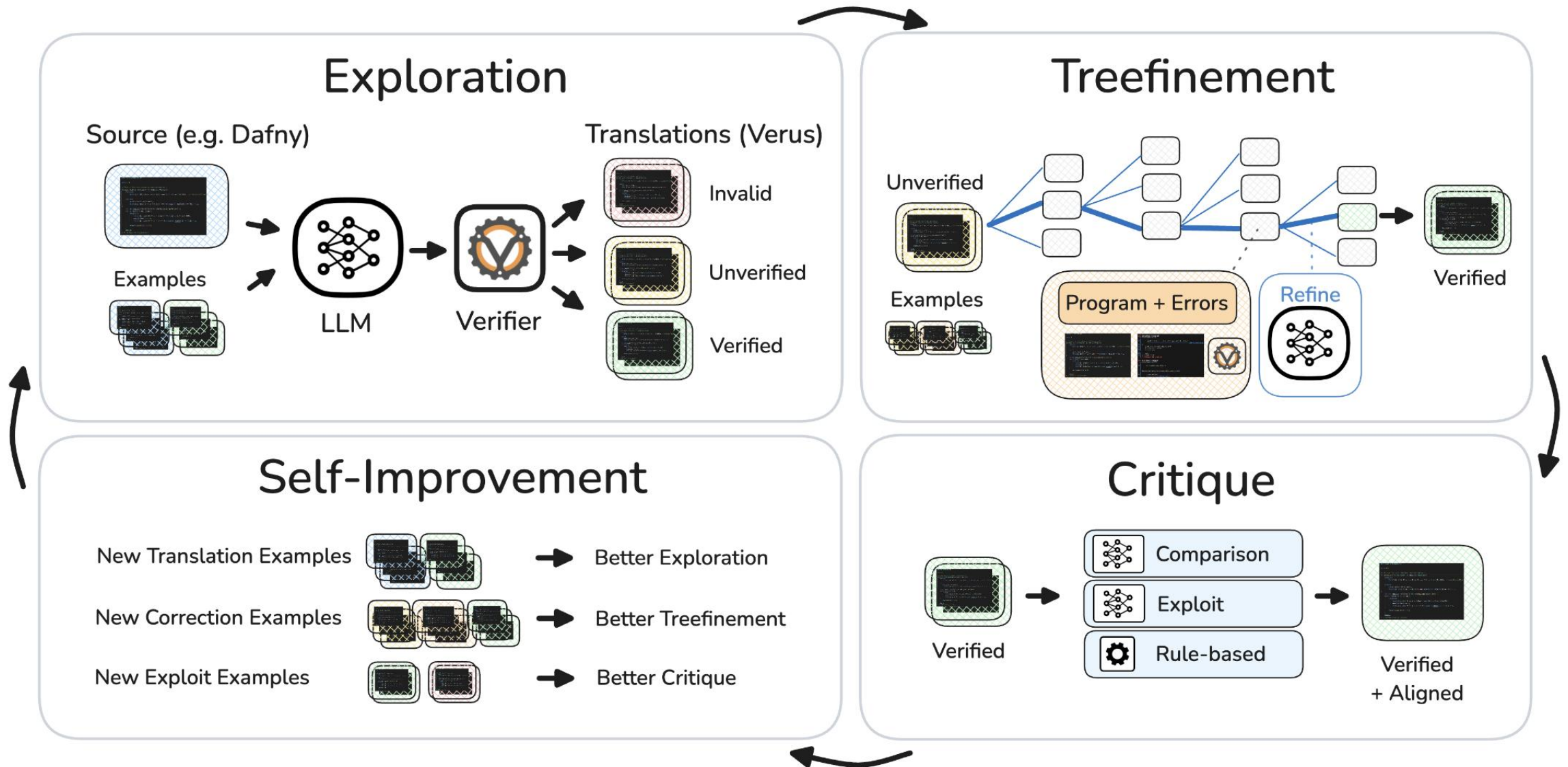
AlphaVerus: Iterative Self-Improving Translation



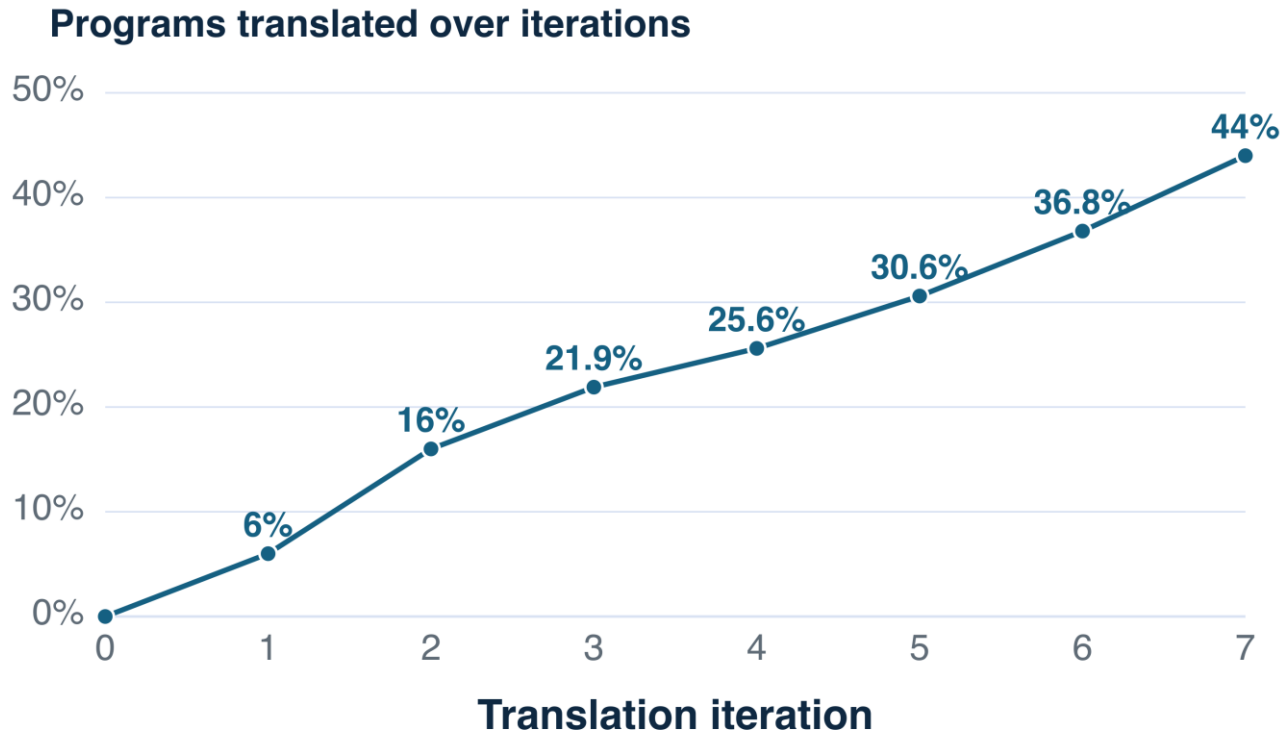
AlphaVerus: Iterative Self-Improving Translation



AlphaVerus: Iterative Self-Improving Translation

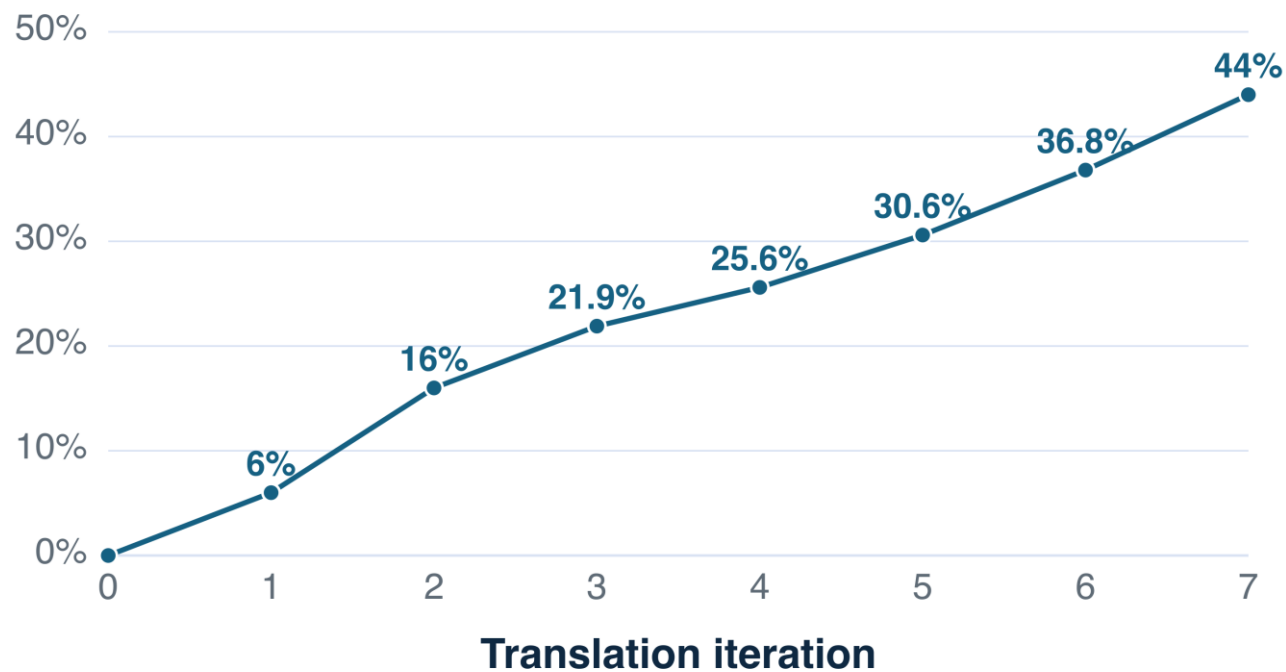


AlphaVerus: Results

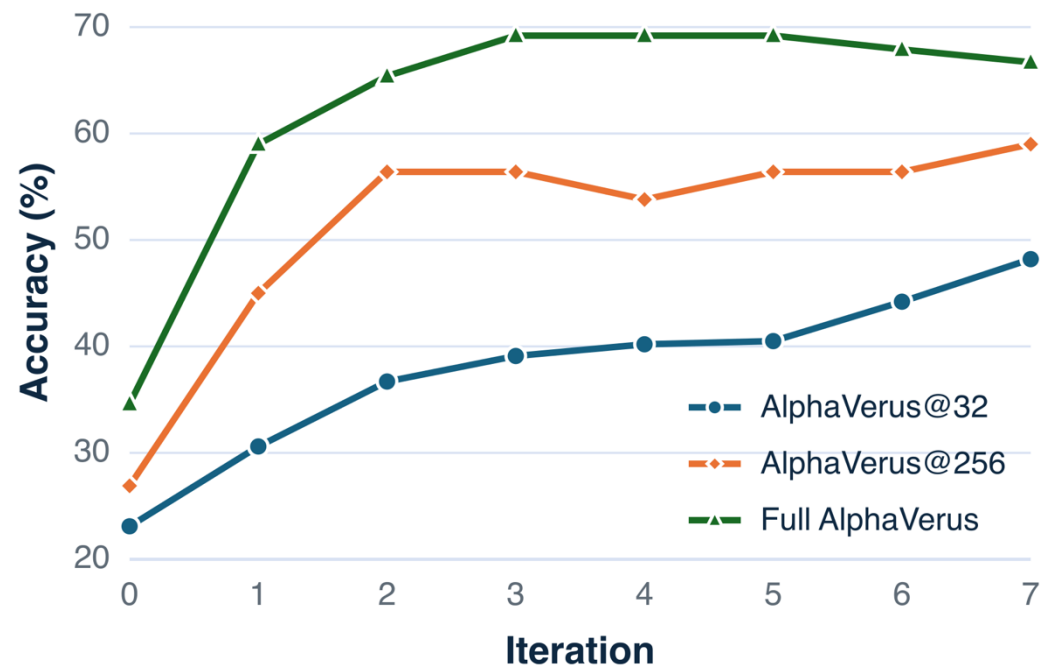


AlphaVerus: Results

Programs translated over iterations

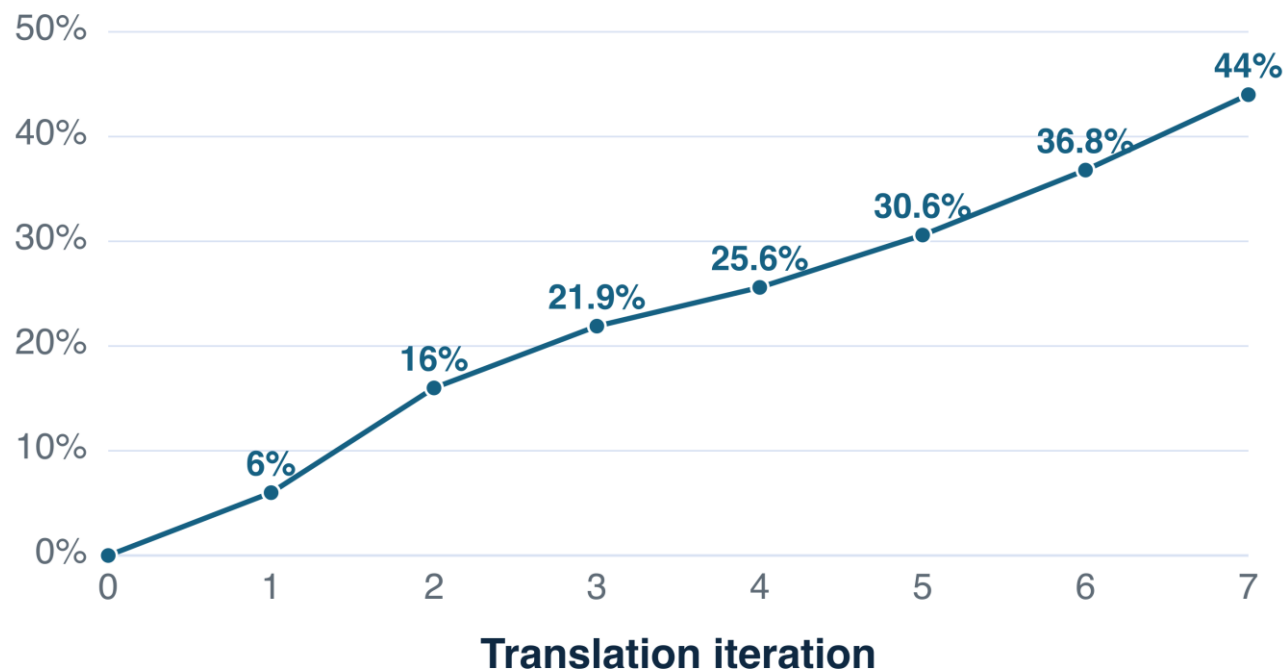


Verified HumanEval accuracy

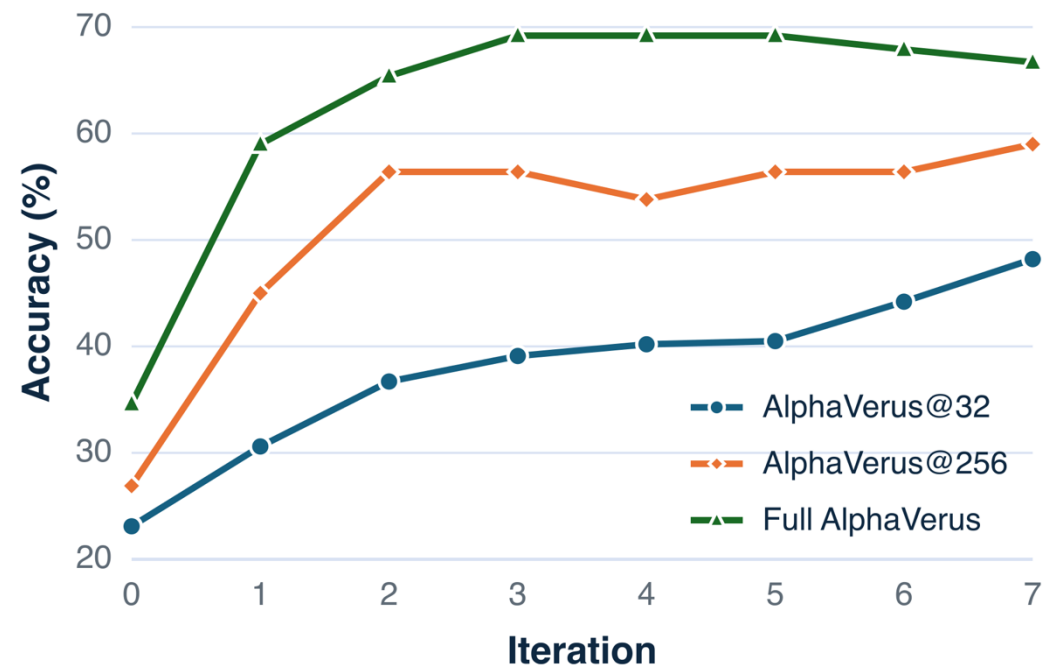


AlphaVerus: Results

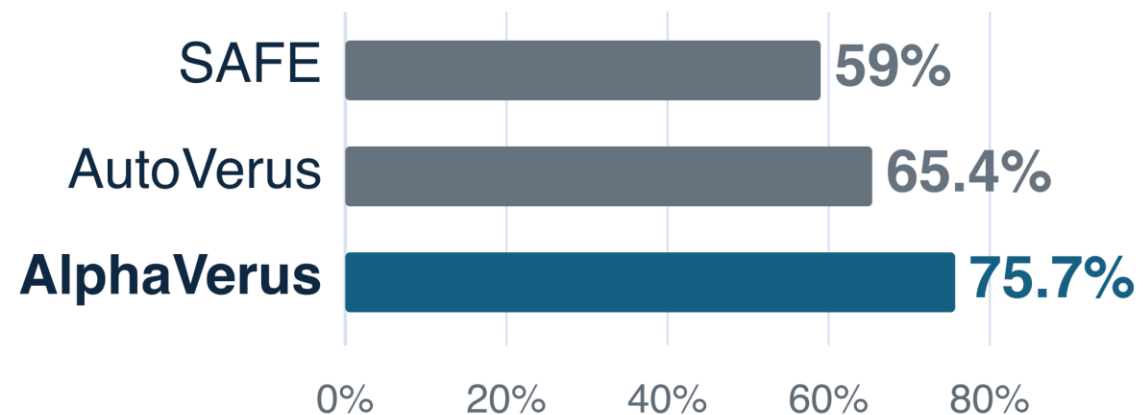
Programs translated over iterations



Verified HumanEval accuracy



Proof-annotation generation on MBPP



What if we don't have the
Source Programs?

Propose-Solve-Verify: **Self-Play Through Formal Verification**

Propose-Solve-Verify: **Self-Play Through Formal Verification**

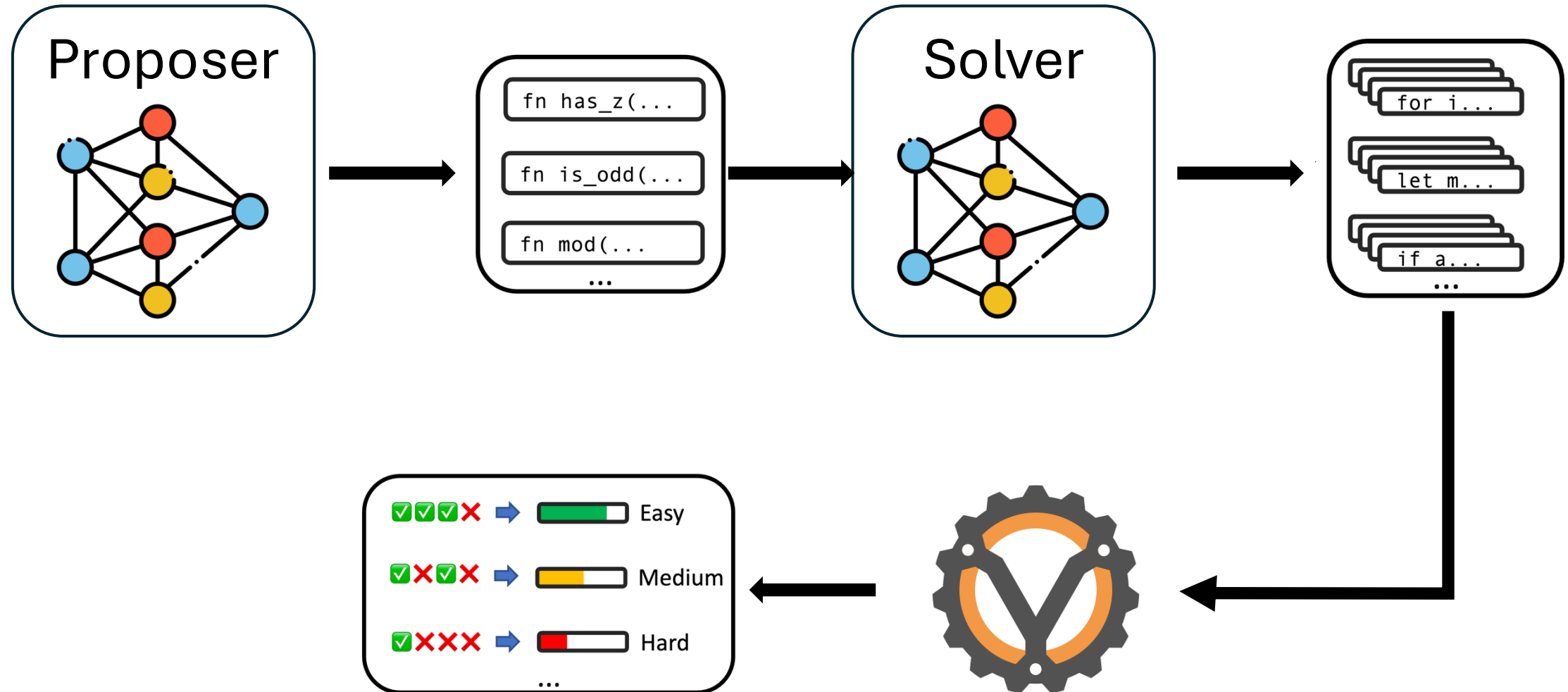
Key Idea:

A **proposer LLM** generates novel specifications

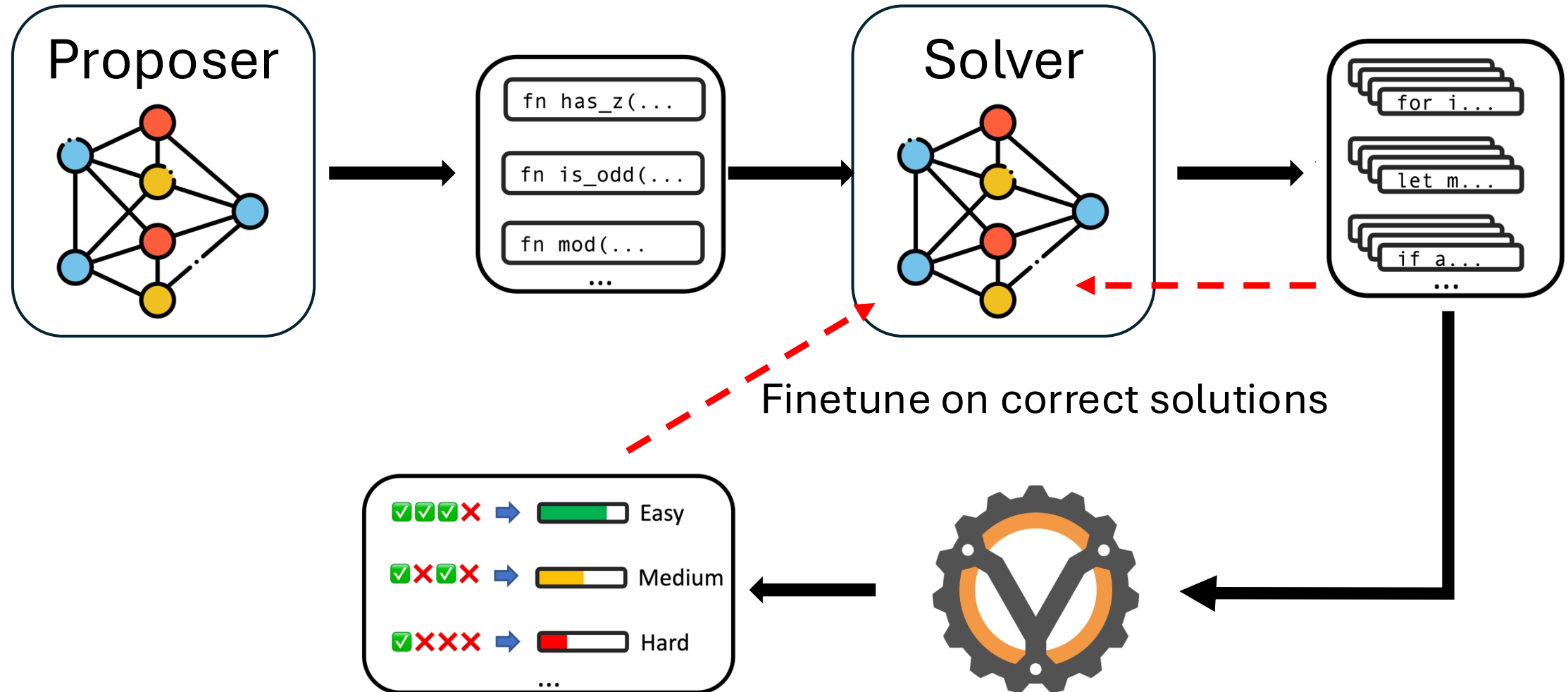
A **solver LLM** write code and proof for those specs!

and both **improve over time**

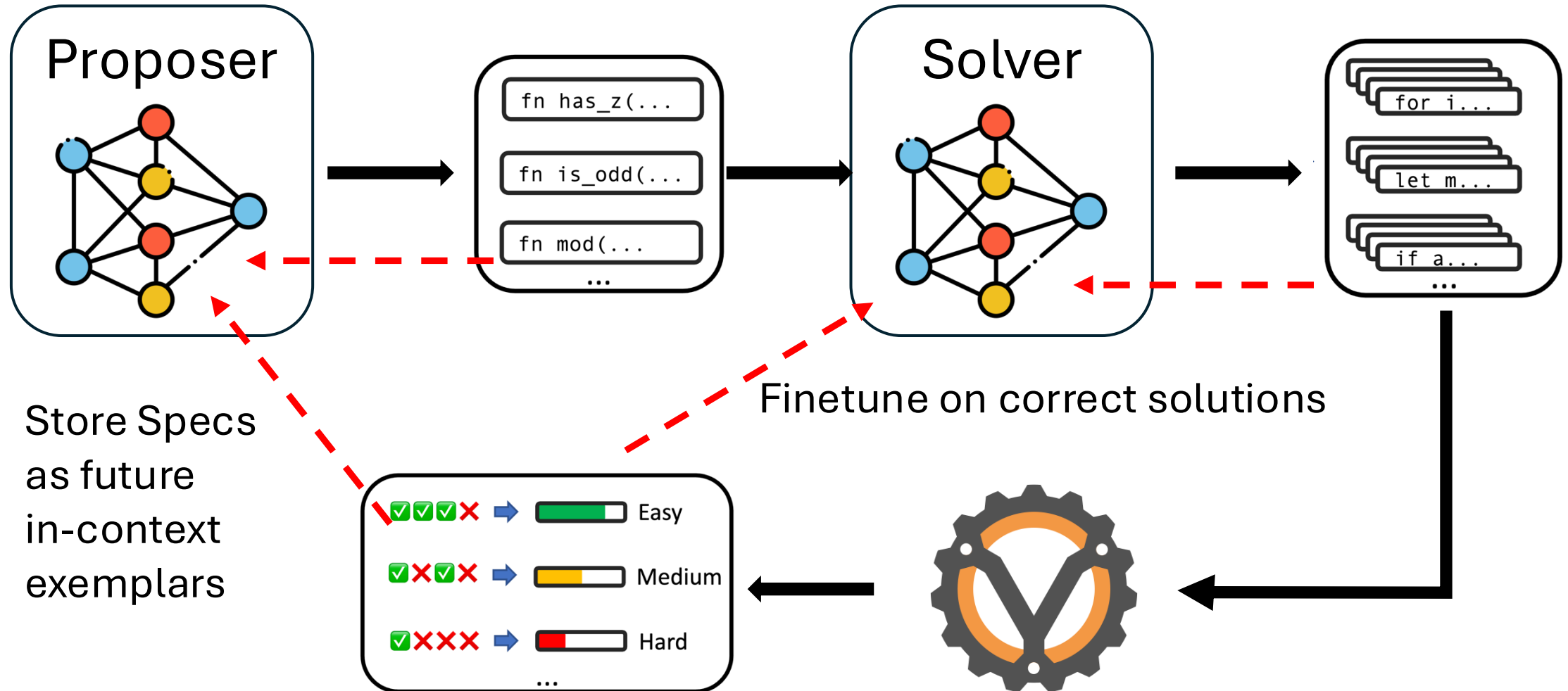
Propose-Solve-Verify



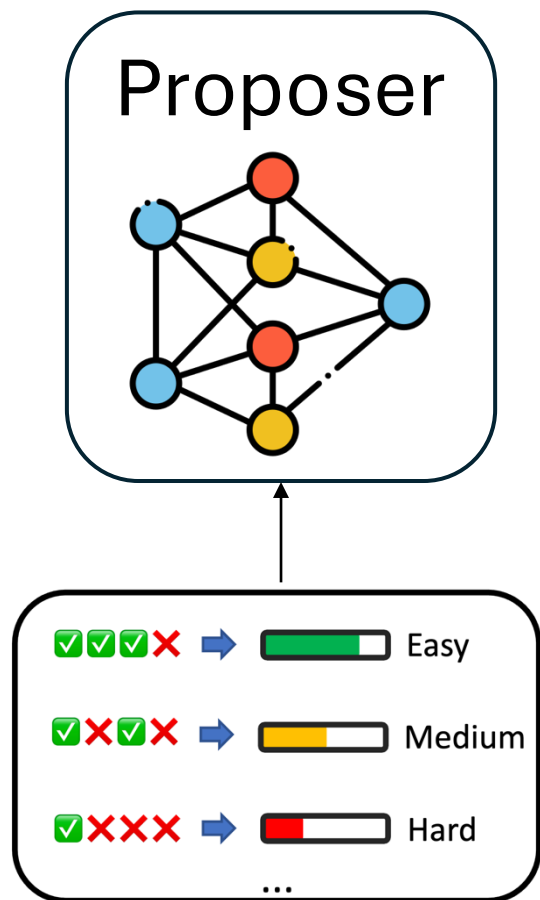
Propose-Solve-Verify



Propose-Solve-Verify



Propose-Solve-Verify



1. Difficulty-Aware Spec Generation

2. Ensure Diversity of generated Specs

Proposer In-Context Learning

I would like you to output a problem that is of {difficulty} level. Here are some examples of problems and how difficult they were for the model.

Problem 1: EASY

{problem spec}

Problem 2: HARD

{problem spec}

...

Now it's your turn! Please describe your reasoning, then output a new problem that is {difficulty}.

Propose-Solve-Verify **outperforms** baselines

Outperforms both
AlphaVerus and Rejection
Finetuning in:

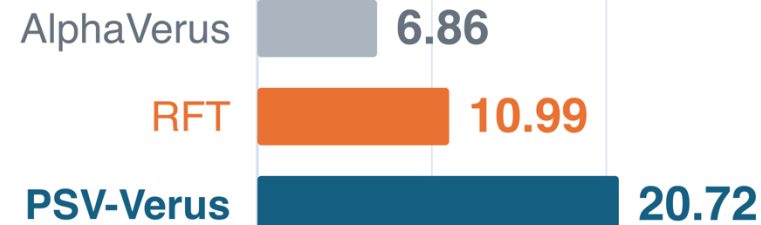
a.) Transfer Learning

b.) Test-Time Training

Average pass@1 (%)

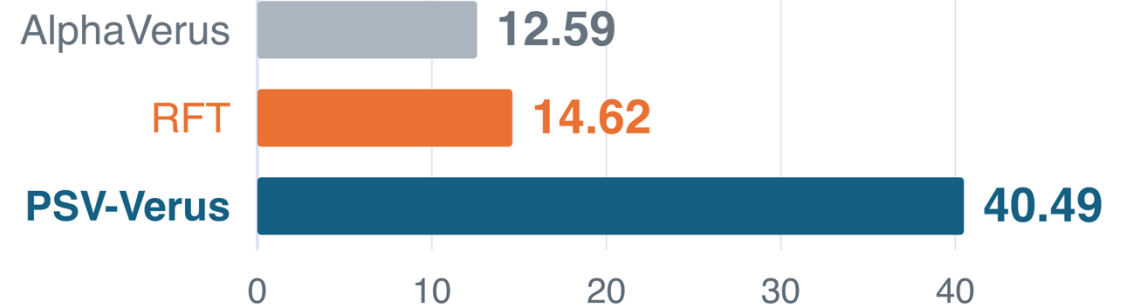
Transfer learning

Average over MBPP + HumanEval



Test-time training

Average over Dafny2Verus + MBPP + HumanEval



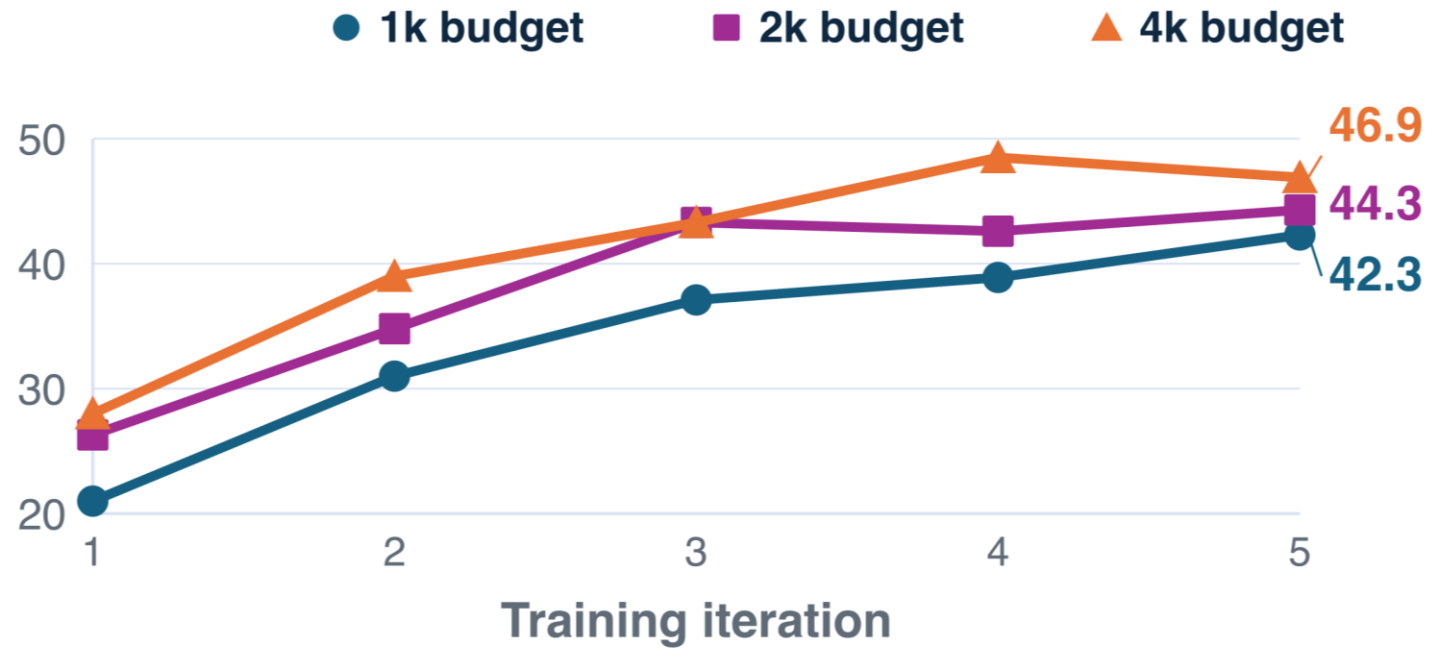
Propose-Solve-Verify: **Scaling Helps**

Performance scales with
increasing:

A.) Iterations

B.) Generated Specs
per iteration

Pass@1 generally rises with more iterations

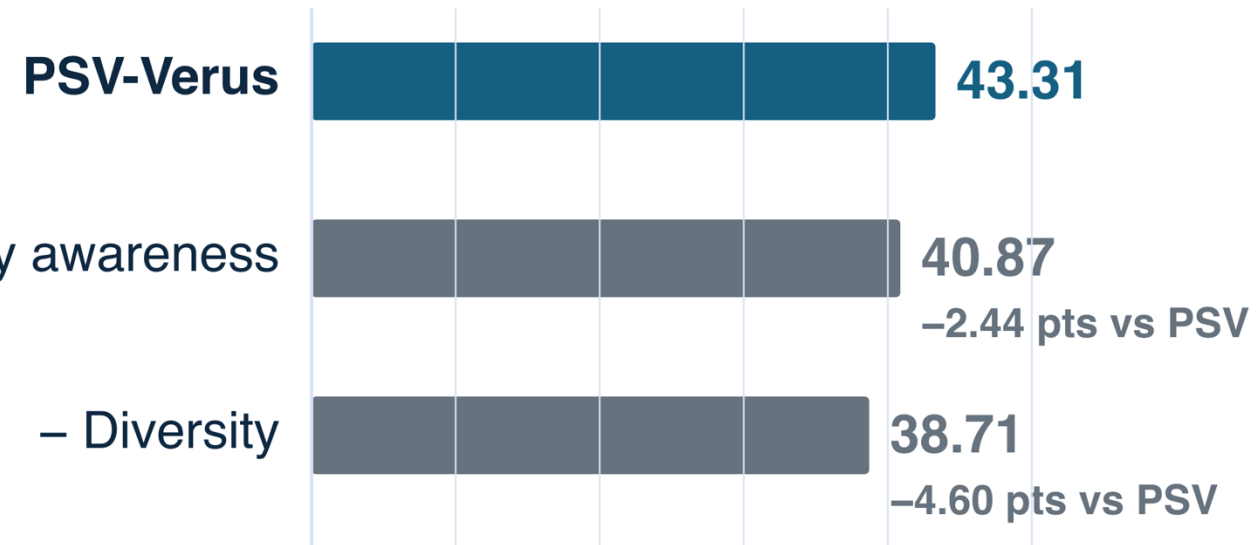


Propose-Solve-Verify: Ablations

We show importance of:

A. Difficulty -Aware Proposer

B. Diversity of generated Specs



What if we make the system
even more general?

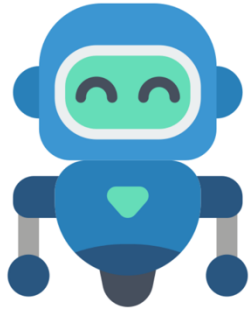
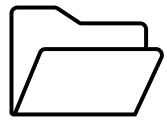
Multi-Agent Simulations (MAS) for Formal Code Generation

Computer environment

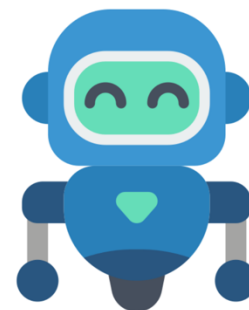
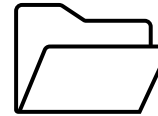


Multi-Agent Simulations (MAS) for Formal Code Generation

Computer environment



Proposer Agent

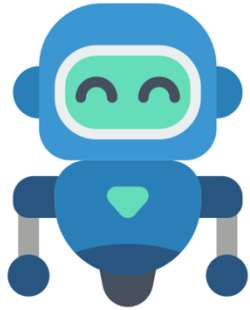
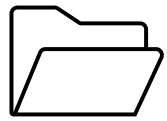


Solver Agent

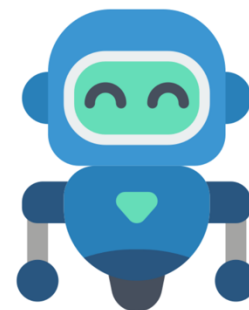
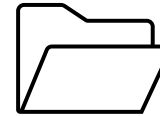


Multi-Agent Simulations (MAS) for Formal Code Generation

Computer environment



Proposer Agent



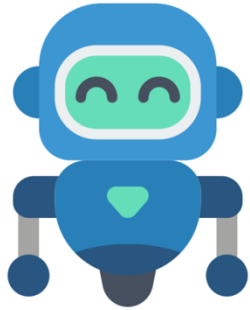
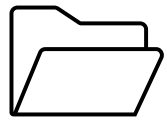
Solver Agent



Only Incentive: Stay Alive!

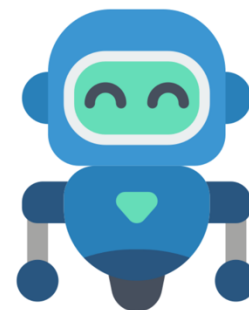
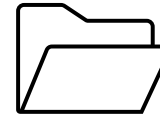
Multi-Agent Simulations (MAS) for Formal Code Generation

Computer environment



Proposer Agent

Reward: $is_{solved} + \alpha * time$



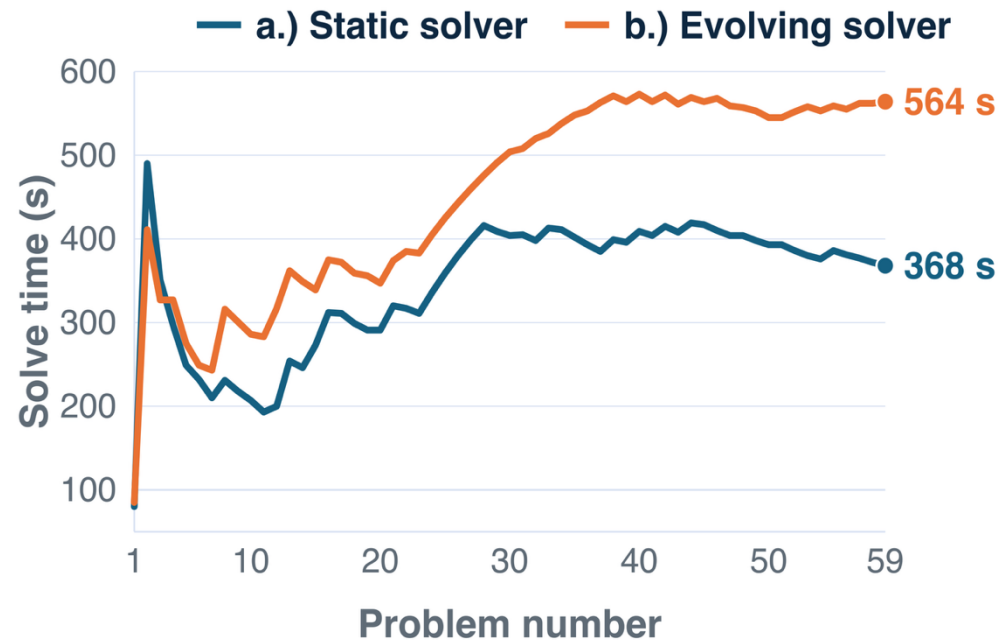
Solver Agent

Reward: $is_{solved} - \alpha * time$

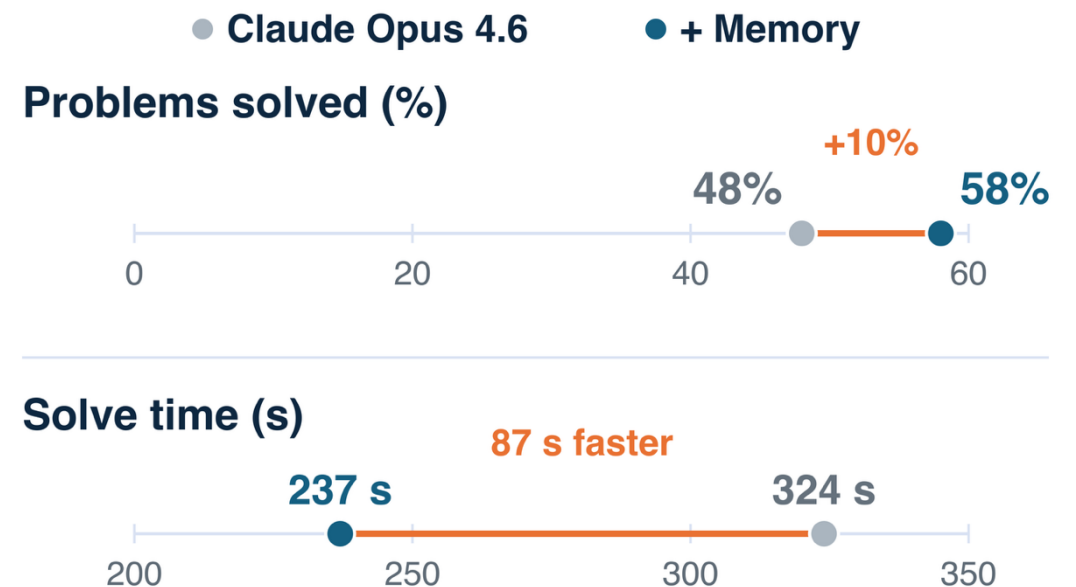


MAS for Formal Code Generation

Proposer improved creating harder specs over time

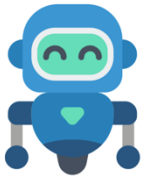


Solver improved, solving more downstream questions faster



MAS for Formal Code Generation : Ongoing Work

Computer environment



Agent 1

MAS for Formal Code Generation : Ongoing Work

Computer environment



MAS for Formal Code Generation : Ongoing Work

Computer environment



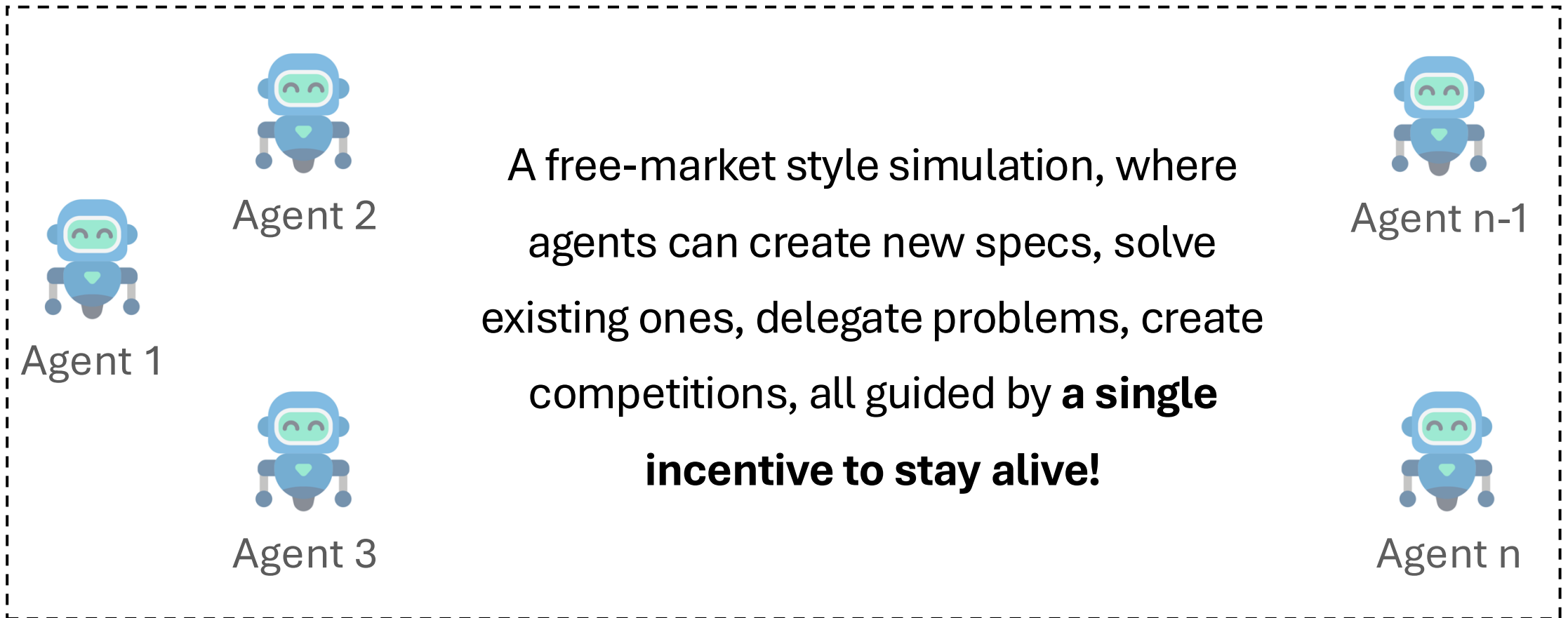
MAS for Formal Code Generation : Ongoing Work

Computer environment



MAS for Formal Code Generation : Ongoing Work

Computer environment



Acknowledgements



Alex Wilf



Anmol Agarwal



Bryan Parno



Cedric Flamant



Daniel Fried



Jannis Limperg



Kanna Shimizu



Louis-Philippe
Morency



Natalie Neamtu



Paul Pu Liang



Pranjal Aggarwal



Sean Welleck



Seungone Kim



AlphaVerus



PSV



VerusSpecGym

Thank you!

This material is based upon work supported by the National Science Foundation CISE Graduate Fellowships under Grant No. 2313998, the National Science Foundation under Grant Nos. DMS-2434614 and DMS-2502281, AFRL and DARPA under Agreement FA8750-24-9-1000, the CyLab Future Enterprise initiative, and gifts from Amazon, the Beneficial AI Foundation, Convergent Research, VMWare, OpenAI Researcher Program, Microsoft, as well as a grant of API credits from Microsoft Azure and Gemini. Pranjal is supported by a SoftBank Group-Arm Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these entities.