(Auto)Verus
Building Software that You Can Trust

Chenyuan Yang

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Software correctness is critical

Space: Ariane 5

Money: Knight Capital

Medical: Radiation therapy

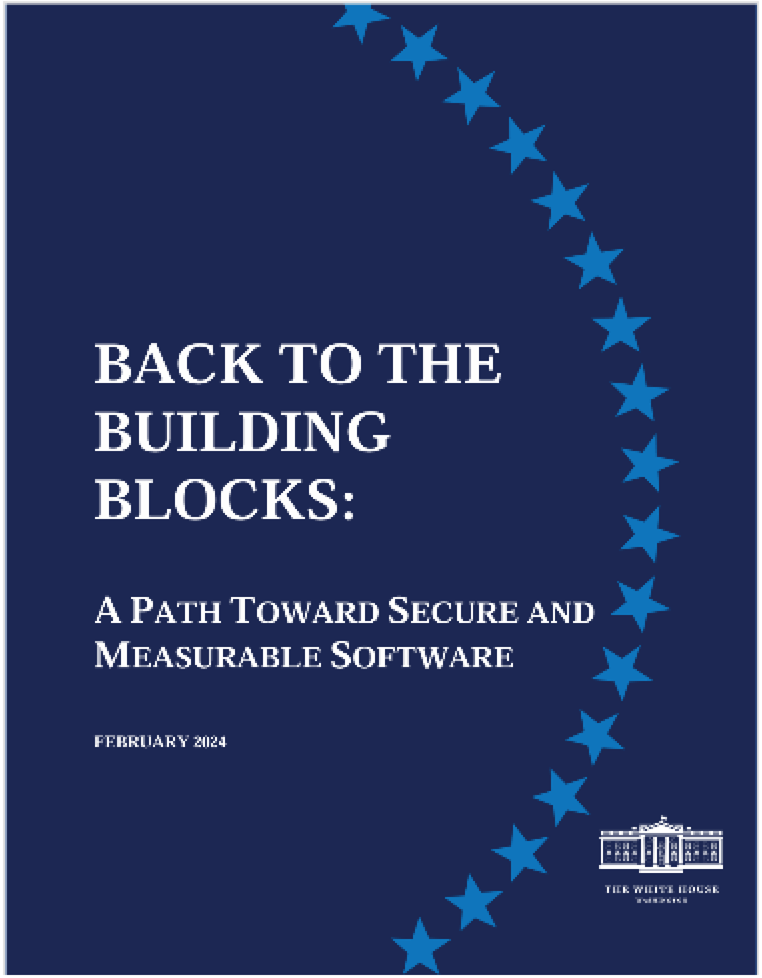
Your online stuff: Amazon crasl

Infrastructure: The north-east US power outage

Transport: American Airlines

[Bad bugs: The worst disasters caused by software fails | New Scientist]

For reliability & security, developers …

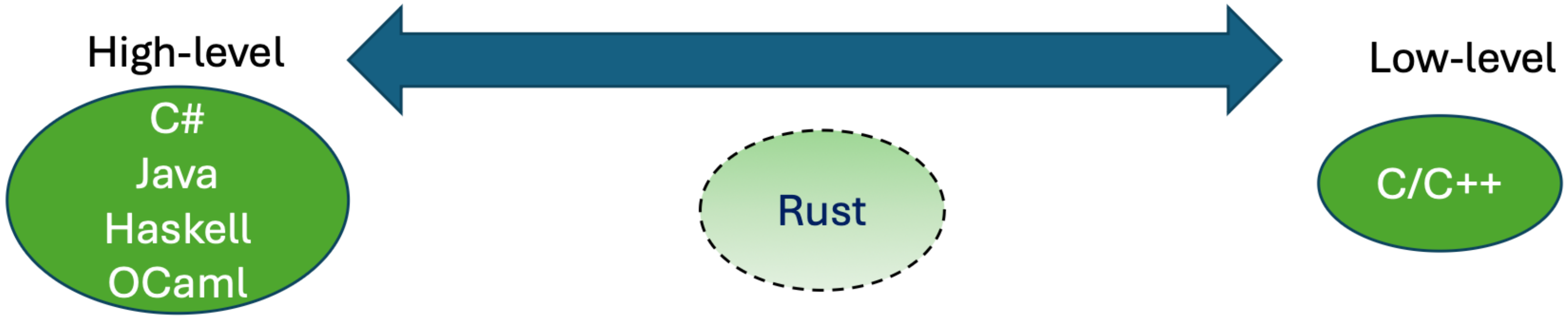


Use memory-safe programming languages, such as Rust …

Use formal verification for the core components…

Why Rust?

- Ownership: Every value has a unique "owner."
- Borrowing: You can borrow a value, but there are strict rules.
  - One mutable borrow OR multiple immutable borrows.
- Lifetimes: The compiler ensures that references don't outlive the data they point to.



Rust borrow example

```rust
fn main() {
    let s1 = String::from("hello");
    // s1's ownership is MOVED to the function
    takes_ownership(s1)
    // This line would cause a compiler error!
    // println!("s1 is {}", s1);
    let s2 = String::from("world");
    // s2 is BORROWED by the function
    borrows_immutably(&s2);
    // s2 is still valid here because it was borrowed
    println!("s2 is still {}", s2);
}
```

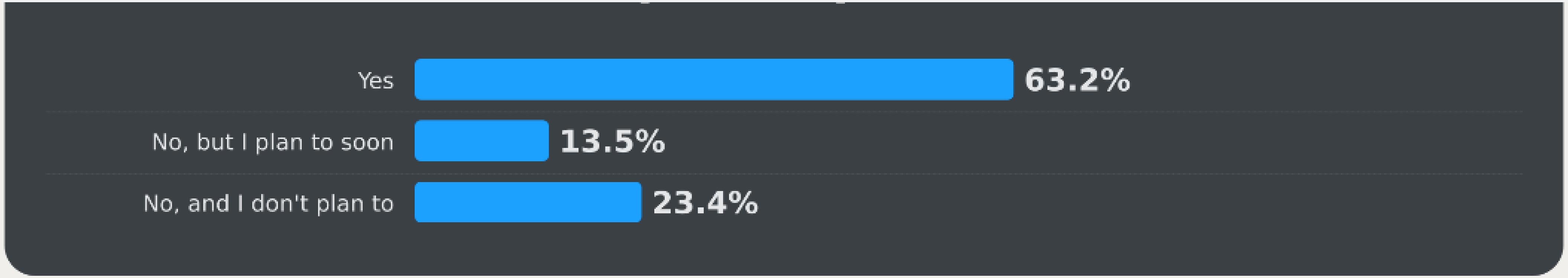
```rust
fn takes_ownership(some_string: String) {
    println!("{}", some_string);
} // `some_string` is dropped here

fn borrows_immutably(some_string: &String) {
    println!("{}", some_string);
} // `some_string` is not dropped
```

For convenience, developers ...

Using AI assistants or coding agents is already the trend!

Do you currently use AI tools in your development process?

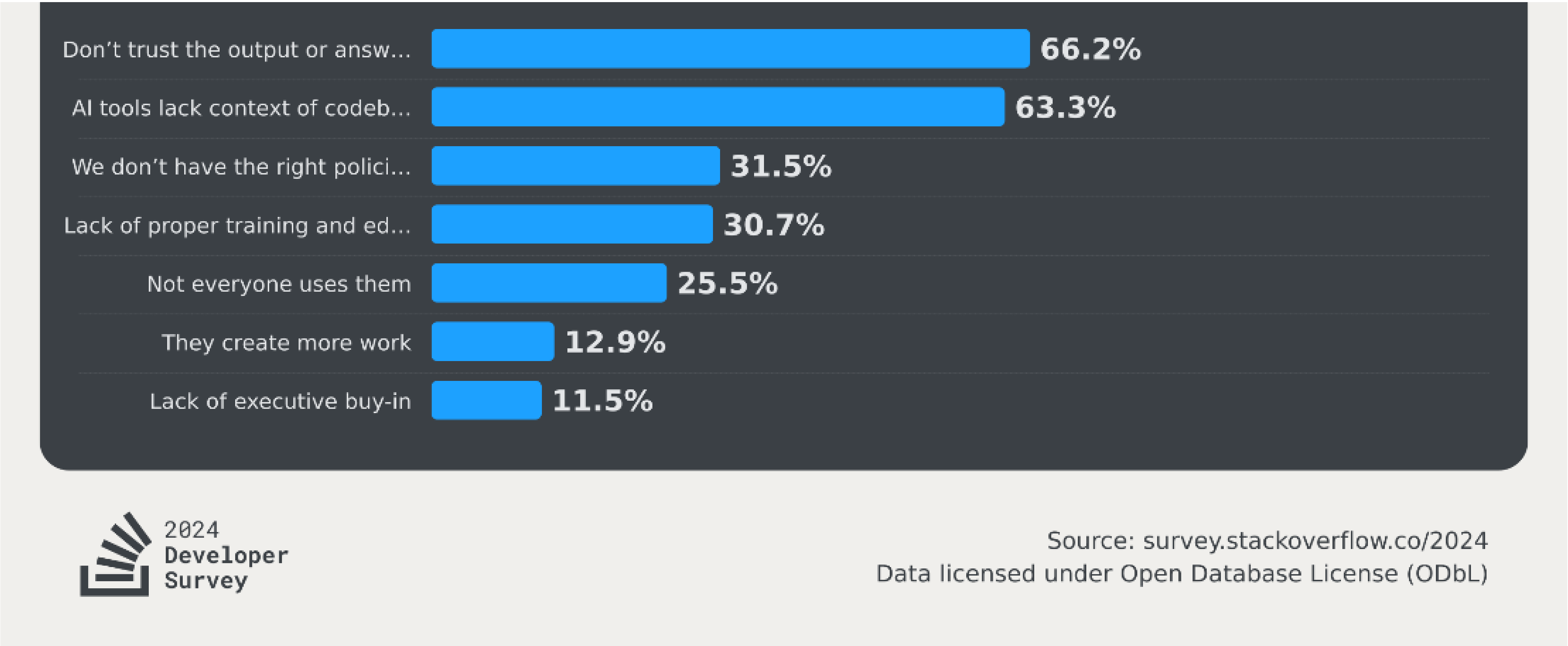


2024 Developer Survey

Source: survey.stackoverflow.co/2024

AI? Reliability & Security?

66.2% Don’t trust the output or answers of AI



How can we trust AI-generated code?

- Software testing
  - To expose bugs in code
  - Active research in AI for testing and testing for AI

But it cannot make sure that there is no bug 🙁

- Software verification
  - To mathematically prove important properties of code
  - To prove that there is no bug!

Why not formally verify software?

- Can I verify the software itself instead of a model of it? ☹️
- Can I not learn a new language to write spec/proof? ☹️
- How long does it take the verifier to run? ☹️
- How fast is the verified software? ☹️

You should try Verus!

- Can I verify the software itself instead of a model of it?
- Can I not learn a new language to write spec/proof?
- How long does it take the verifier to run?
- How fast is the verified software?

*“**Verus** is a tool for verifying the correctness of code written in Rust. Developers write specifications of what their code should do, and Verus statically checks that the executable Rust code will always satisfy the specifications for **all possible executions** of the code”*

GitHub
https://github.com/verus-lang/verus

verus

Verified Rust for low-level systems code

25 Watched 1.4k Starred 85 Forks

Primary language: Rust **License:** MIT license

Verus is already used for various systems projects

- Persistent-memory log, key-value store for Azure Storage
- VeriSMo security module OSDI Best Paper
- Concurrent memory allocator *(CMU)*
- Atmosphere microkernel *(U-Utah)*
- Anvil Cluster Management *(U-Illinois, U-Wisconsin, VMware)* OSDI Best Paper

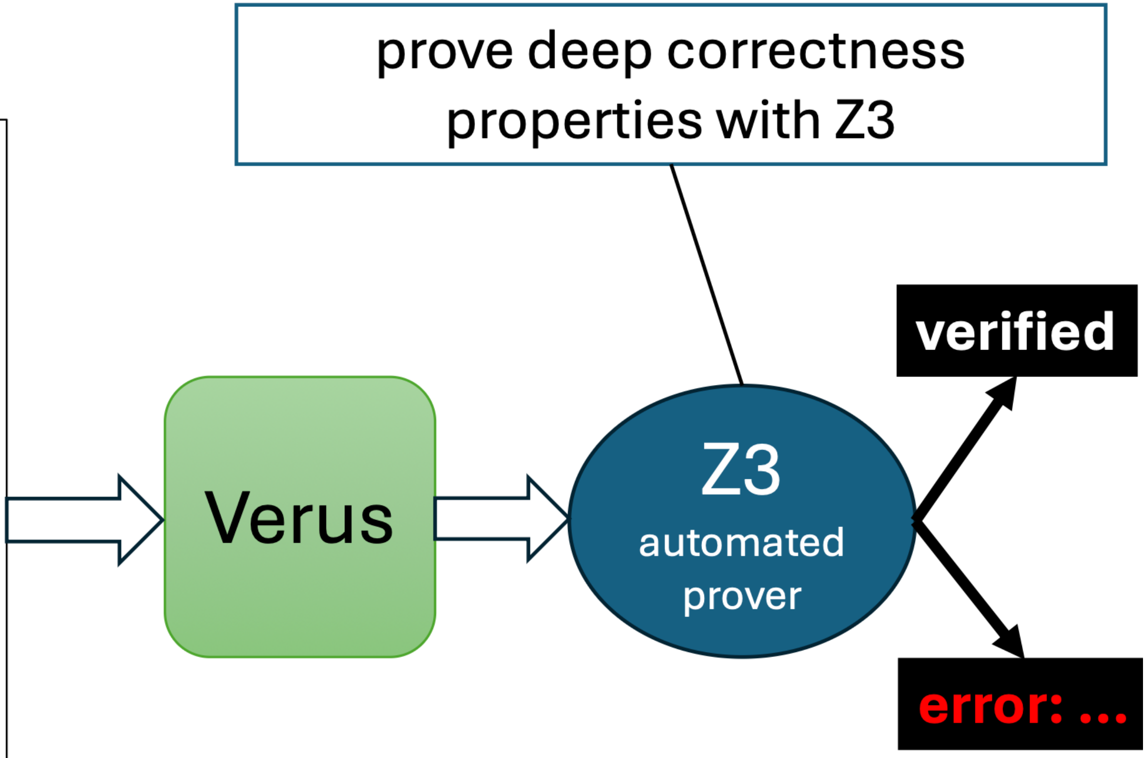
"Verus: A Practical Foundation for Systems Verification" SOSP Best Artifact

"Linear Types for Large-Scale Systems Verification" OOPSLA Best Paper

How Verus works?

express specs, proofs in Rust

```
fn binary_search(v: &Vec<u64>, key: u64)
                                  -> (ret: usize)
   requires
     forall|i:int, j:int| ... ==> v[i] <= v[j],
     ...
   ensures
     key == v[ret], ...
{
   let mut left = 0;
   let mut right = v.len() - 1;
   while left != right
     invariant
       right < v.len(), …
   { … }
   ...
}
```



How Verus works?

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)




{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right



    {
        let mid = left + (right - left) / 2;
        if v[mid] < key {
            left = mid + 1;
        } else {
            right = mid;
        }
    } …
```

How should we describe the functionality of binary_search?

Pre-Condition `requires`

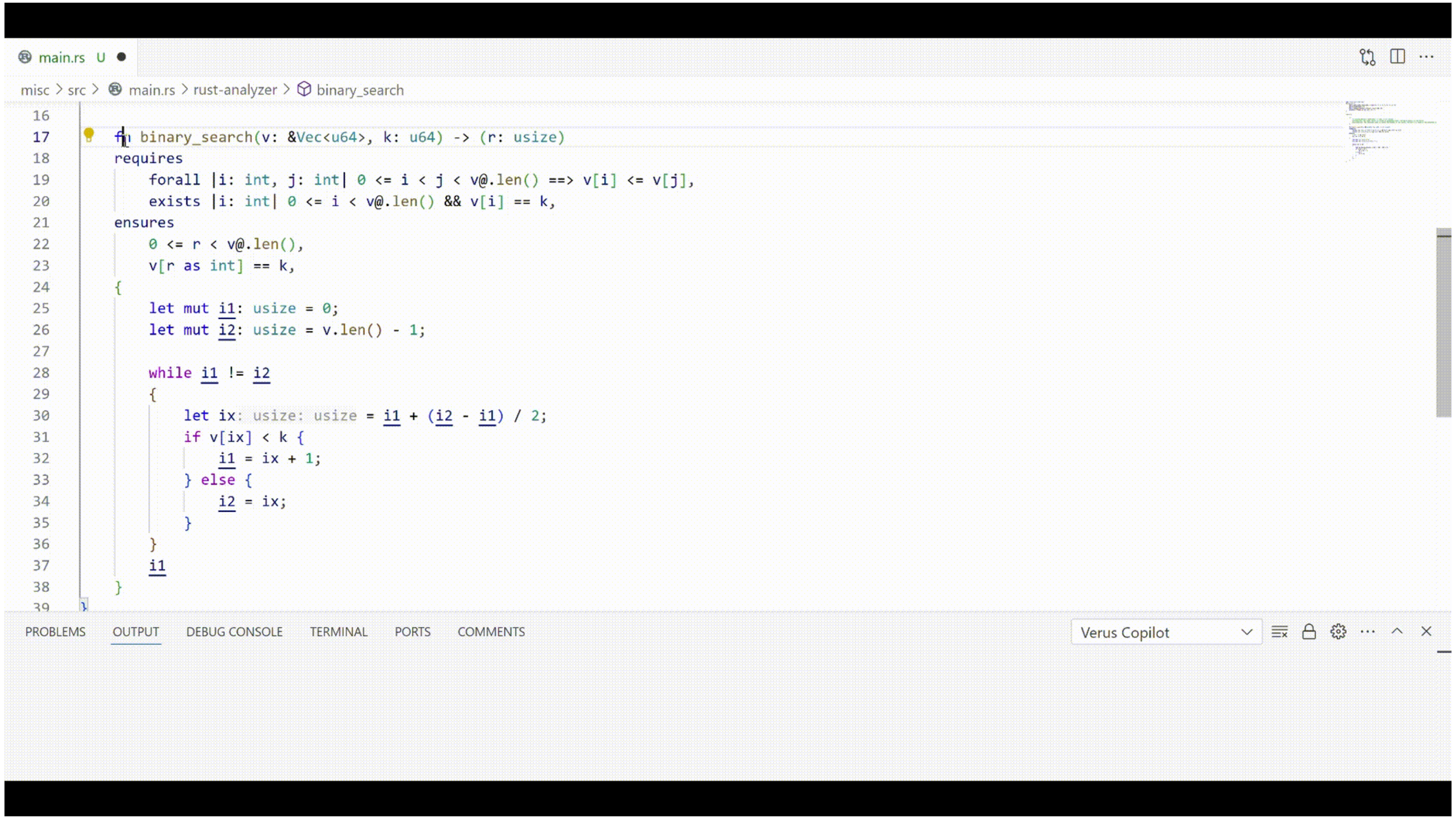
Post-Condition `ensures`

*"The input **v** is a sorted vector and it has the value **key** we want to find"*

*"The output **ret** is a valid index **i** such that **v[i]** equals the **key**"*

But …

- Can AI generate Verus specifications and proofs for me?



Can GPT-4 prove binary-search?

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)
requires
    forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
    exists |i: int| 0 <= i < v@.len() && key == v[i],
ensures
    ret < v.len(), key == v[ret as int],
{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right
    {
        let mid = left + (right - left) / 2;
        if v[mid] < key {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    left
}
```

"You are a Verus expert. Please add proof annotations for the following code, so that Verus can prove the function Implementation satisfies the function specification."

??

Lack of knowledge: syntax

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)
requires
    forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
    exists |i: int| 0 <= i < v@.len() && key == v[i],
ensures
    ret < v.len(), key == v[ret as int],
{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right
      invariant
        exists |i: usize| 0 <= i < v@.len() && key == v[i],
        …
    { … }

}
```

Verus error:
mismatched type, expecting `int' yet getting `usize'

Lack of skills: loop invariants

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)
requires
    forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
    exists |i: int| 0 <= i < v@.len() && key == v[i],
ensures
    ret < v.len(), key == v[ret as int],
{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right
      invariant
        forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
        exists |i: int| left <= i <= right && key == v[i],
    {
        let mid = left + (right - left) / 2;
        if v[mid] < key {
            left = mid + 1;
        } else {
            right = mid;
        }
    } …
```

Missing invariant:
right < v@.len()

Lack of skills: loop invariants

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)
requires
    forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
    exists |i: int| 0 <= i < v@.len() && key == v[i],
ensures
    ret < v.len(), key == v[ret as int],
{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right
      invariant
        forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
        exists |i: int| left <= i <= right && key == v[i],
    {
        let mid = left + (right - left) / 2
        if v[mid] < key {
            left = mid + 1;
        } else {
            right = mid;
        }
    } …
```

Missing invariant:
right < v@.len()

Verus error:

1. Function postconditions not satisfied
2. invariant not satisfied at the end of loop
3. Precondition, mid < v@.len(), of v[mid] not satisfied

Lack of strategy: debugging, prioritization, …

```
fn binary_search(v: &Vec<u64>, key: u64) -> (ret: usize)
requires
    forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
    exists |i: int| 0 <= i < v@.len() && key == v[i],
ensures
    ret < v.len(), key == v[ret as int],
{
    let mut left: usize = 0;
    let mut right: usize = v.len() - 1;
    while left != right
      invariant
        forall |i: int, j: int| 0 <= i < j < v@.len() ==> v[i] <= v[j],
        exists |i: int| left <= i <= right && key == v[i],
    {
        let mid = left + (right - left) /
        if v[mid] < key {
            left = mid + 1;
        } else {
            right = mid;
        }
    } …
```

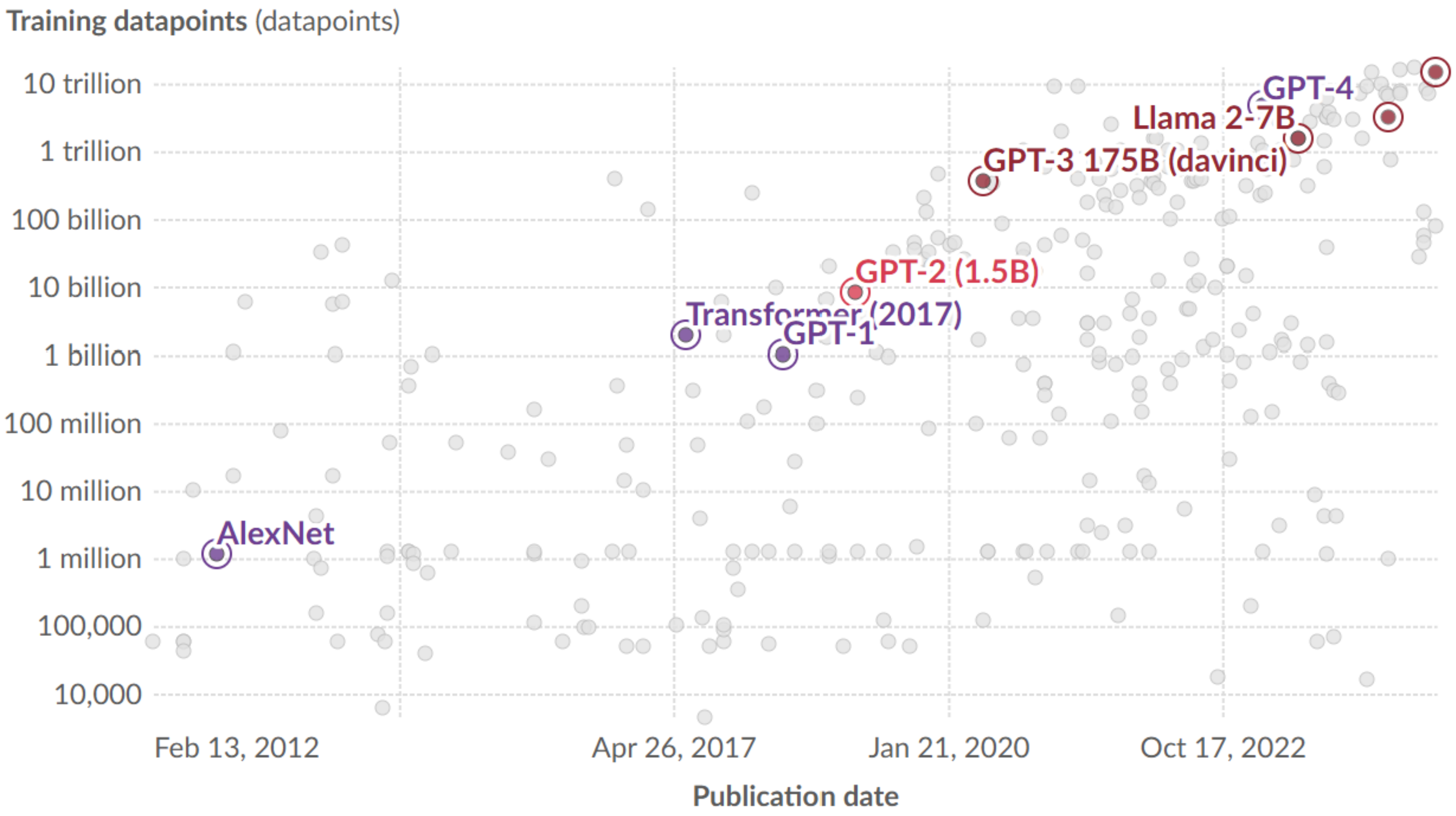
Verus error:

1. Function postconditions not satisfied
2. invariant not satisfied at the end of loop
3. Precondition, mid < v@.len(), of v[mid] not satisfied

How to teach AI proof knowledge, skills, strategies?

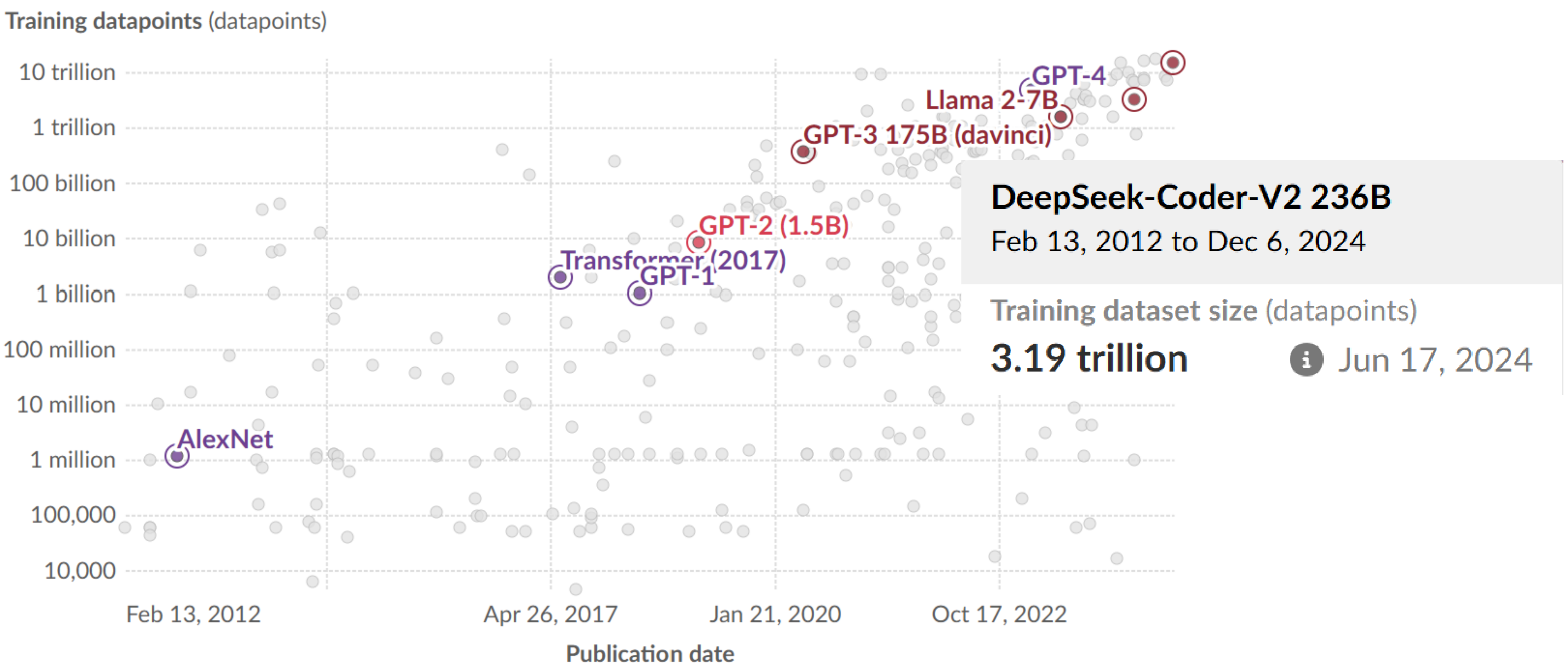
Not enough data!

The amount of data used to train models



OurWorldinData.org/artificial-intelligence

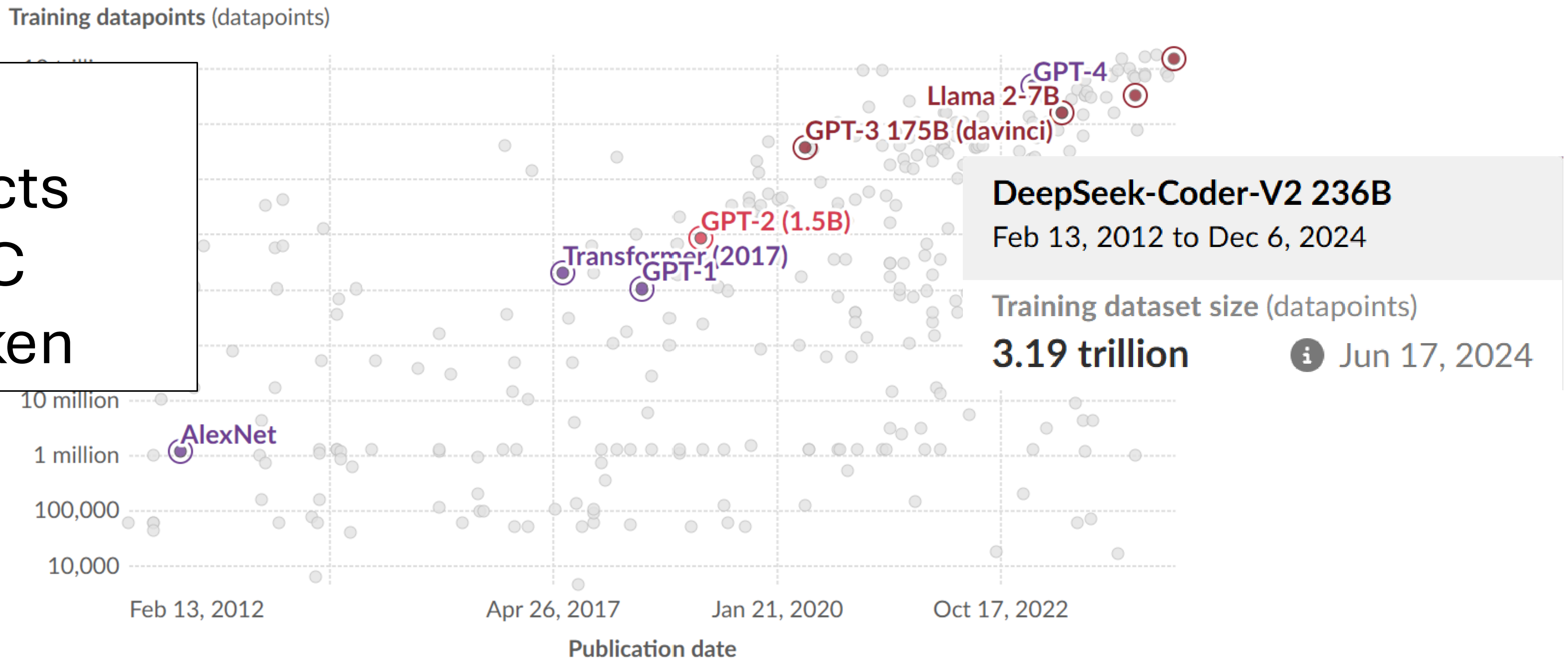
The amount of data used to train models



OurWorldinData.org/artificial-intelligence

The amount of Verus data available

Verus
~ 10 projects
~ 100K LoC
~ 500K Token



OurWorldinData.org/artificial-intelligence

AutoVerus:
Automated Proof Generation for Rust Code

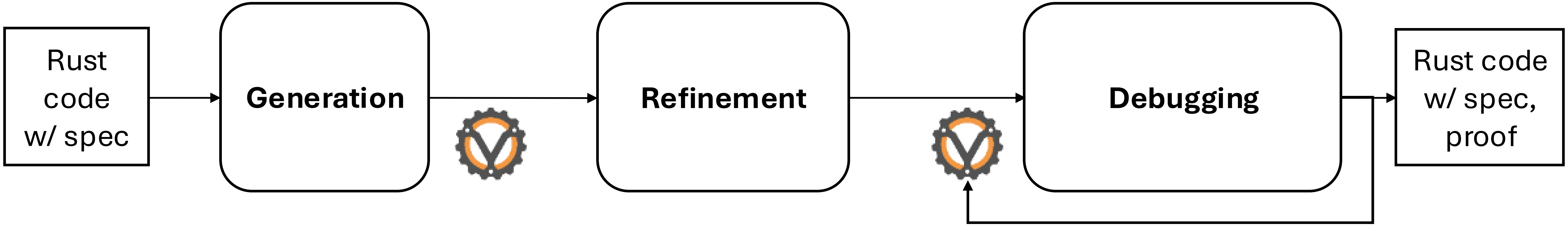
An agent framework supporting LLM through prompts, workflow, compiler & formal methods

OOPSLA 2025

How to teach LLMs to write Verus proof?

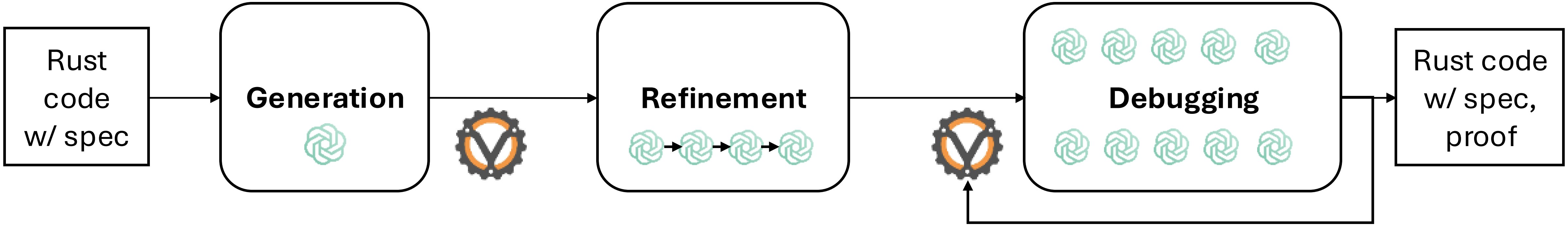
- A workflow that mimics human experts' methodology

"The proof development of human experts is an iterative process of repeatedly running Verus, checking and prioritizing Verus errors, developing and editing proof to fix them." – Interview of multiple co-authors of the Verus paper



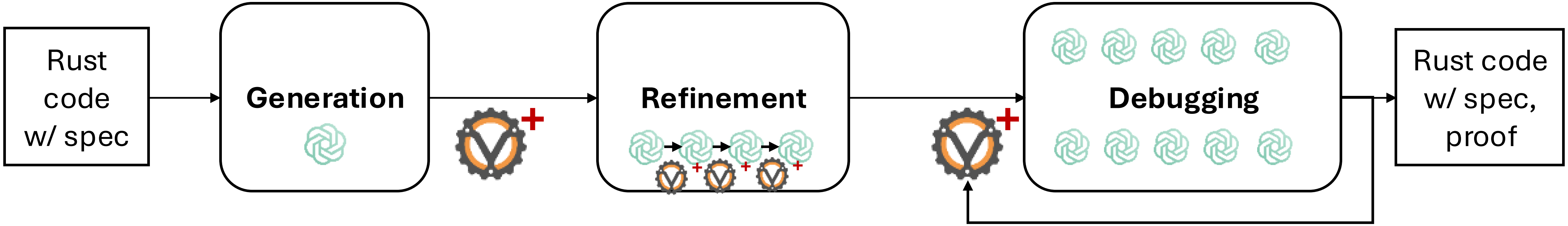
How to teach LLMs to write Verus proof?

- A workflow that mimics human experts' methodology
- An extensible network of GPT agents w/ Verus knowledge, skills

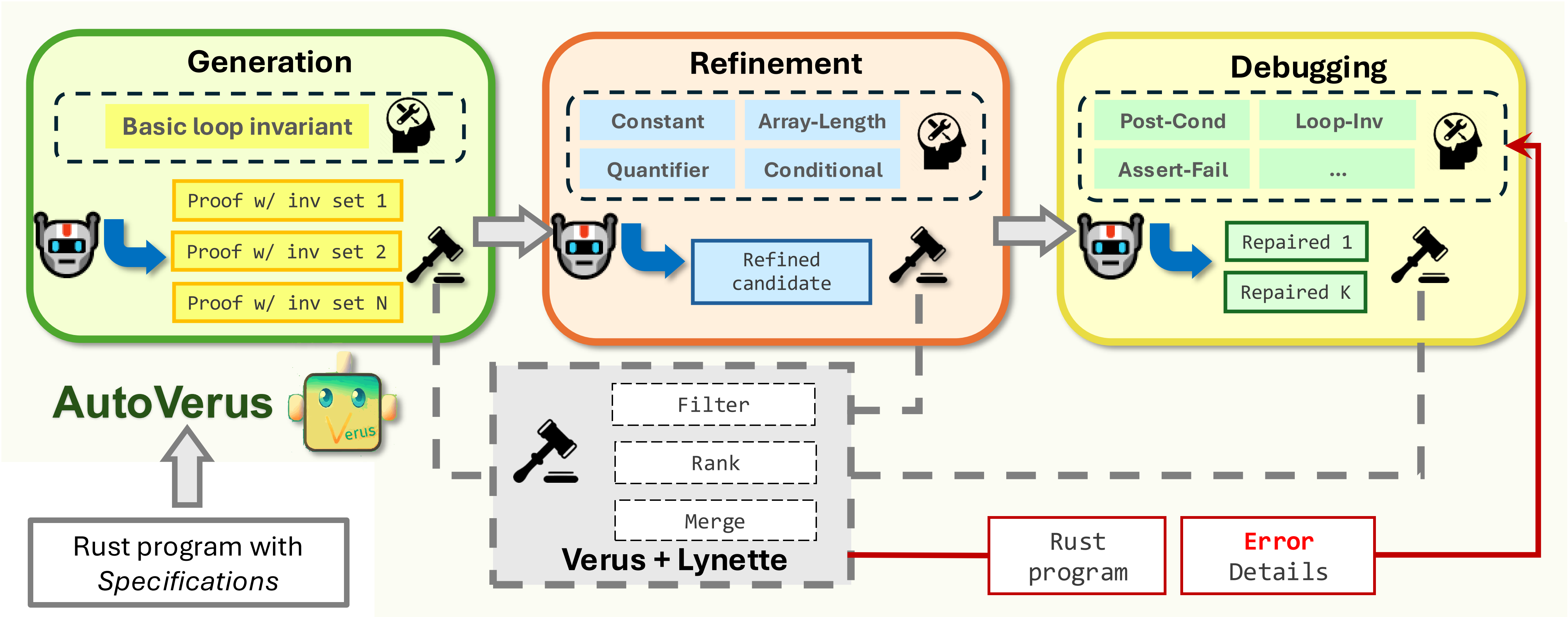


How to teach LLMs to write Verus proof?

- A workflow that mimics human experts' methodology
- An extensible network of GPT agents w/ Verus knowledge, skills
- Proof discipline, ranking, merging, … supported by formal methods



AutoVerus



Generation
Basic loop invariant
Proof w/ inv set 1
Proof w/ inv set 2
Proof w/ inv set N
Refinement
Constant
Array-Length
Quantifier
Conditional
Refined candidate
Debugging
Post-Cond
Loop-Inv
Assert-Fail
...
Repaired 1
Repaired K
AutoVerus
Verus
Filter
Rank
Merge
Verus + Lynette
Rust program with *Specifications*
Rust program
Error Details

Lack of Discipline

- LLM may cheat on the verification

```
while (i < v.len())
    invariant
        ...
        result.len() <= i,
        ...
{
    ...
}
```

error: invariant not satisfied after loop

```
while (i < v.len())
    invariant
        ...
        result.len() <= i,
        ...
{
    ...
    assume(result.len() <= i);
}
```

`assume` makes the verification pass, but by adding assumption

Lack of Discipline

- LLM may change the executable code

```
while (i < v.len())
{
  if (v[i] <= e) {
    result.push(v[i]);
  }
  i = i + 1;
}
```

→

```
while (i < v.len())
{
  if (v[i] > e) {
    result.push(v[i]);
  }
  i = i + 1;
}
```

We need discipline!

- We DO want to unleash the creativity of LLM

But

- We do NOT want LLM to make arbitrary changes to the input
- We need an effective way to search among many creative outputs

How to add discipline to LLM?

Using Verus + Lynette to

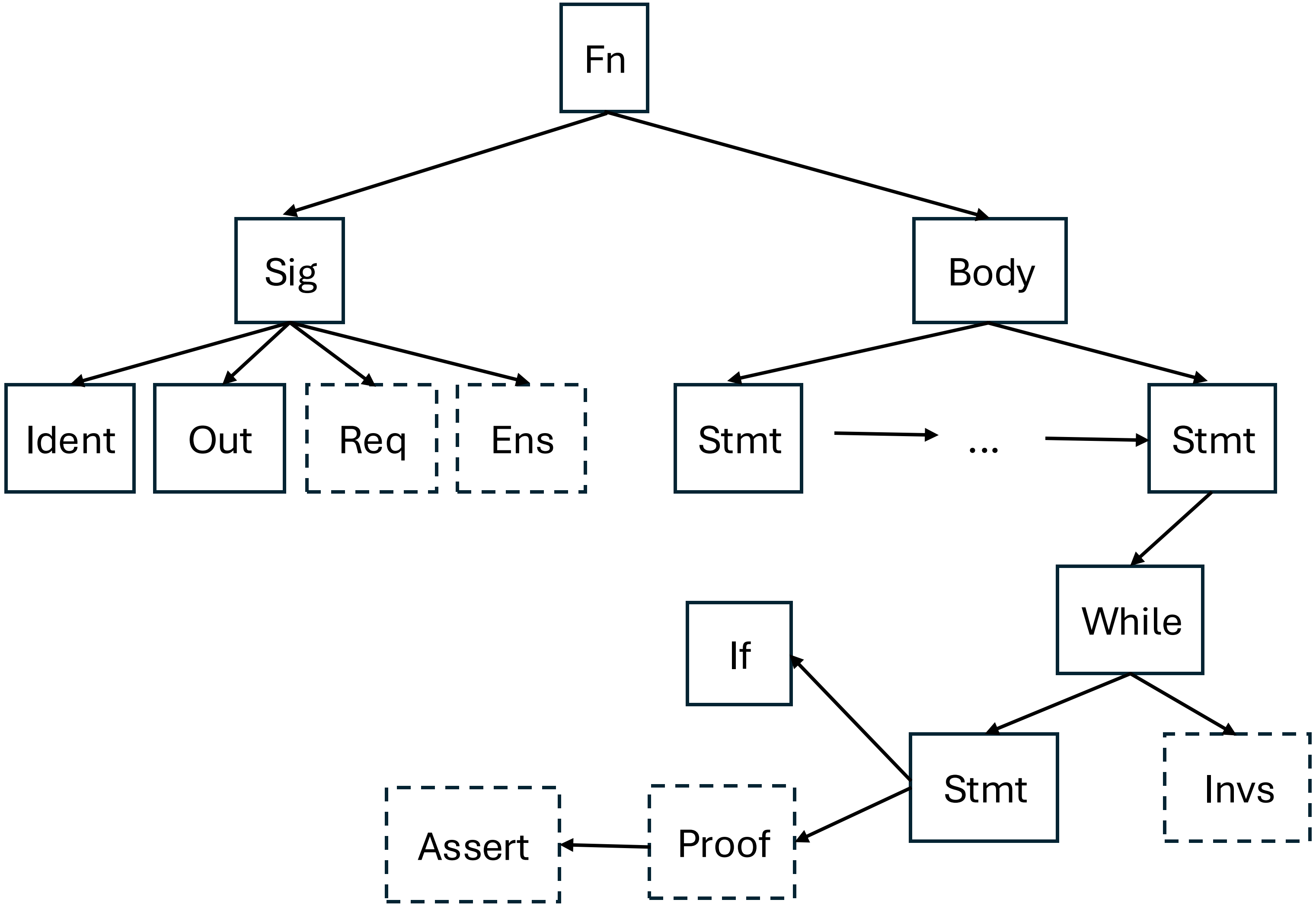
- Rank
- Merge
- Filter

all LLM outputs

The exact ranking/merging/filtering policy is skipped from this talk

Discipline - Lynette, the Verus Source Forger

```
fn remove_all_greater(v: Vec<i32>, i: T)
  -> (result: Vec<i32>)
  requires
    forall |k1:int,k2:int|
      0<=k1<k2<v.len() ==> v[k1] != v[k2]
  ensures
    forall |k:int| 0 <= k < result.len()
==>
      result[k] <=
e&&v@.contains(result[k]),
    forall |k:int| 0 <= k < v.len()
      &&
v[k]<=e==>result@.contains(v[k]),
{
  let mut i: usize = 0;
  let vlen = v.len();
  let mut result: Vec<i32> = vec![];
  while (i < v.len())
  {
    if (v[i] <= e) {
      result.push(v[i]);
    }
    i = i + 1;
    proof {
      assert(...);
      ...
    }
  }
  result
}
```



Discipline - Lynette, the Verus Source Forger

- Detecting "unsafe" changes
  - AST-level comparison
  - Bottom-line: Generate same executable code
    - By erasing all ghost code
    - Then comparing the rest of the code
  - Conditional
    - Spec function
    - Pre/post condition
    - Assumption

Expertise – Error-Fix Action Table

Error Type
Function postcondition not satisfied
Function precondition not satisfied
Function precondition not satisfied (Vector Length Violation)
Loop invariant not satisfied at end of loop body
Loop invariant not satisfied before the loop body
Assertion failed
Arithmetic overflow/underflow
Type error
Misc. verus syntax error
…

Expertise – Error-Fix Action Table

Error Type	Fix Actions
Postcondition not satisfied	Add the proof blocks related to the post-condition at the exit point
	Modify the existing loop invariants
Precondition not satisfied	Add the assertions related to the pre-condition just before the invocation of the function
Precondition not satisfied - Vector Length	add loop invariants/asserts for the array: 1. an invariant that specify the array length (i.e., A.len() == ...); 2. an invariant about the array index not under bound (e.g., k >= 0)
Invariant not satisfied at end of loop body	the end of the loop
Invariant not satisfied before loop	nt before the loop body
	e it correct
	Add the failed invariant to all the loops before the failed loop
	Delete the failed loop invariant
Assertion failed	Add the necessary assertions before the failed assertion
	Add appropriate loop invariants to ensure the assertion holds true
	Fix the assertion error for the following code by using existing lemma functions
	Fix the assertion error for the following code by creating the helper proof functions
...	...

One LLM agent for each fix action

Repair: Post-Condition Not Satisfied

Your mission is to **fix the post-condition not satisfied error** for the following code. Basically, you should add the proof blocks related to the post-condition at the exit point, or modify the existing loop invariants to make them work for the post-condition

There are two general fixes for the “post-condition not satisfied” error

```
pub fn filter(x: &Vec<u64>, y: &mut Vec<u64>)
requires
    old(y).len() == 0,
ensures
    y@ == x@.filter(|k:u64| k%3 == 0),
{
    let mut i: usize = 0;
    let xlen = x.len();

    while (i < xlen)
    invariant
        i <= xlen,
        y@ == x@.take(i as int).filter(|k:u64| k%3 ==
0),
    {
        if (x[i] % 3 == 0) {
            y.push(x[i]);
        }
        i = i + 1;
    }
}
```

→

```
pub fn filter(x: &Vec<u64>, y: &mut Vec<u64>)
requires
    old(y).len() == 0,
ensures
    y@ == x@.filter(|k:u64| k%3 == 0),
{
    let mut i: usize = 0;
    let xlen = x.len();

    while (i < xlen)
    invariant
        i <= xlen,
        y@ == x@.take(i as int).filter(|k:u64| k%3 ==
0),
    {
        if (x[i] % 3 == 0) {
            y.push(x[i]);
        }
        i = i + 1;
    }

    proof {
        assert(y@ == x@.filter(|k:u64| k%3 == 0));
    } // Added by AI
}
```

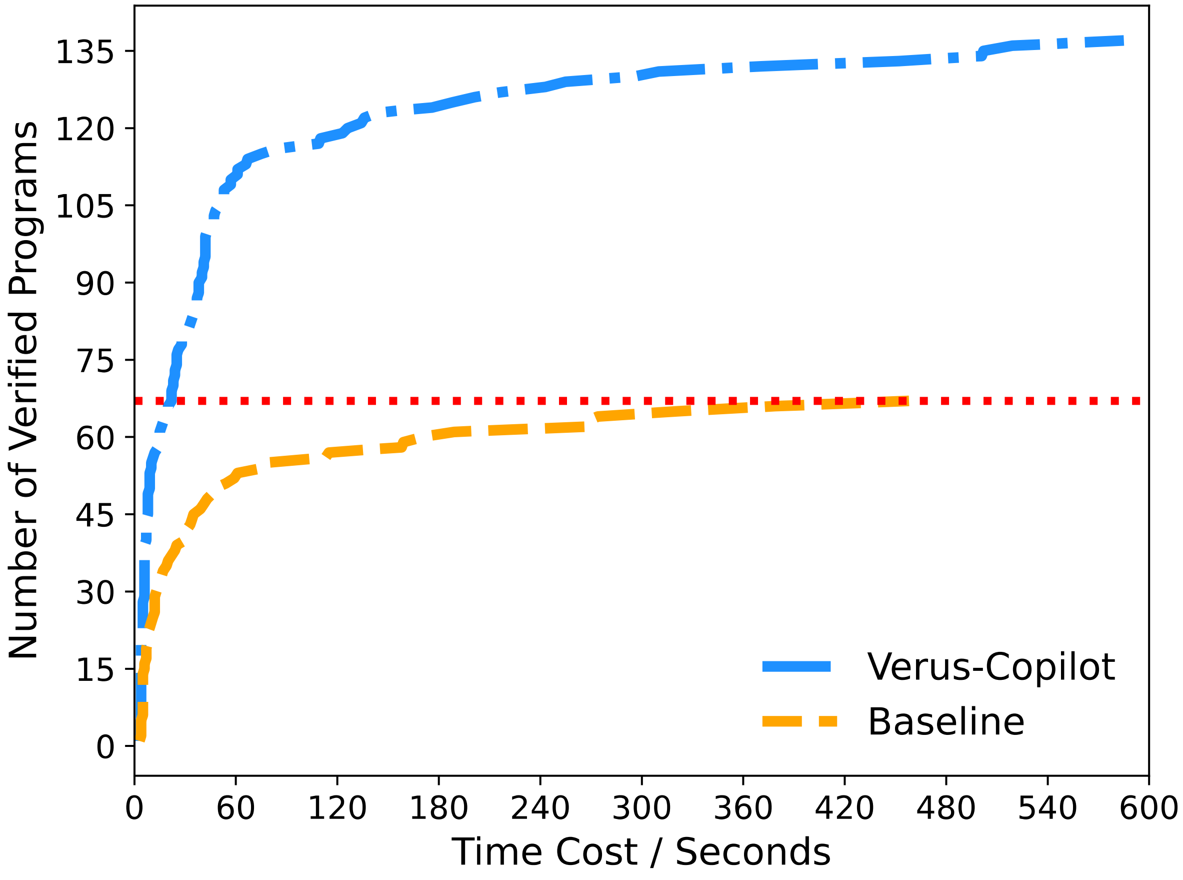
Benchmark Construction: Verus-Bench

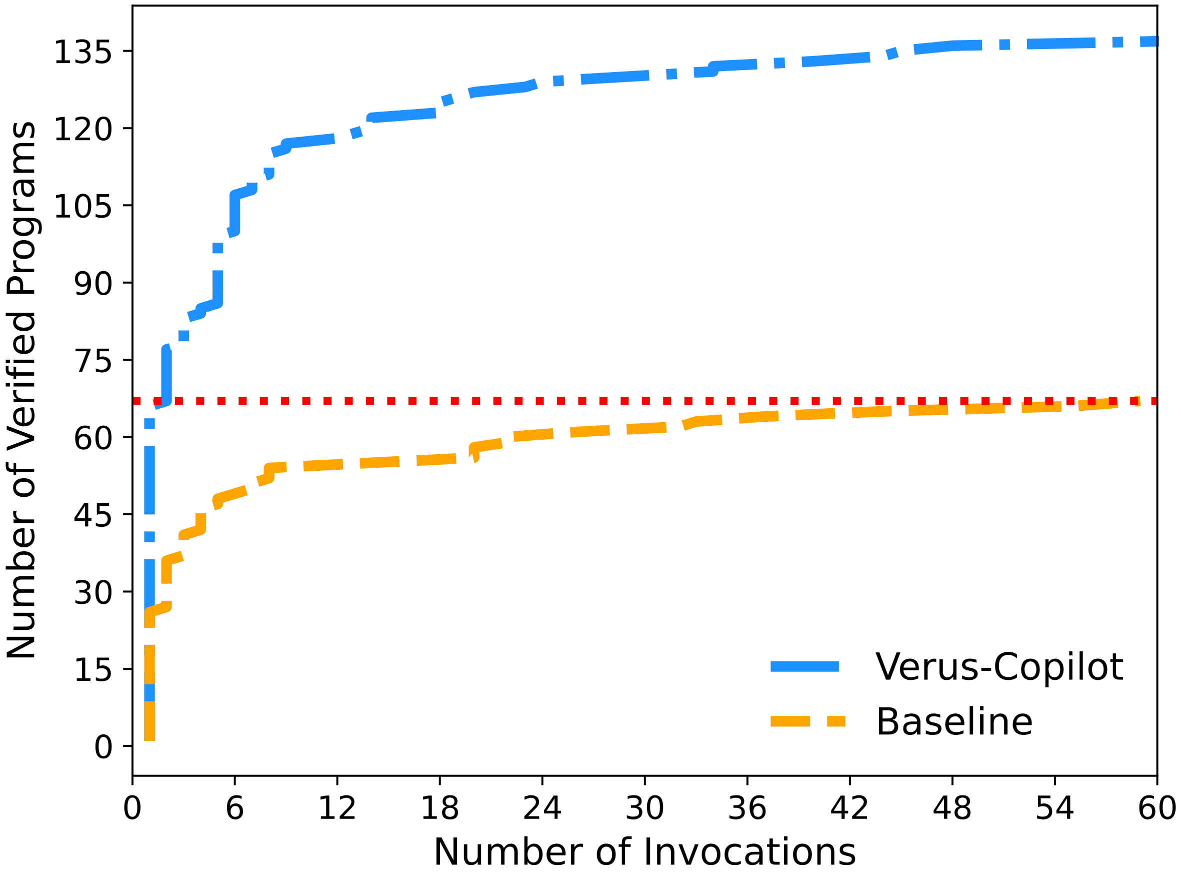
- No **existing** Verus proof generation benchmark
- We translated three verification-related benchmark in other languages (C, Dafny) into Verus
  - CloverBench[1], Diffy[2], MBPP[3]
  - Misc is collected from Verus tutorials
- The **first** benchmark designed for Verus proof generation

Benchmark Sources	CloverBench	Diffy	MBPP	Misc	Total
# of Proof Tasks	11	38	78	23	150
Executable LOC	175	951	1,333	390	2,849
Specification LOC	80	265	700	207	1,252

[1] Sun, Chuyue, et al. "Clover: Clo sed-Loop Ver ifiable Code Generation." International Symposium on AI Verification. Cham: Springer Nature Switzerland, 2024.
[2] Chakraborty, Supratik, Ashutosh Gupta, and Divyesh Unadkat. "Diffy: Inductive reasoning of array programs using difference invariants." , CAV 2021,
[3] Misu, Md Rakib Hossain, et al. "Towards ai-assisted synthesis of verified dafny methods." Proceedings of the ACM on Software Engineering 1.FSE (2024)

Results Based on Time and Invocation





Much better performance than directly invoking LLMs

File Edit Selection View Go Run ···

misc (Workspace)

main.rs U

misc > src > main.rs > binary_search

```rust
/*
Binary search implementation in Rust.

Preconditions:
- The input vector `v` must be sorted in non-decreasing order.

Postconditions:
- If the element `x` is found in the vector, the function returns `Some(i)` where i is a valid index of the vector and points to x.
- If the element `x` is not found, the function returns `None`.
*/

verus!{
fn binary_search(v: Vec<u64>, x: u64) -> (result: Option<usize>)

{
    let mut low: i32 = 0;
    let mut high: usize = v.len();

    // Return Some(i) if x is found, else return None
    return None;
}
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Filter

Verus Copilot

Recap

- Verus could make sure that AI-generated code is 100% correct!
  - Precondition
  - Postcondition
  - Proof annotation like loop invariants, assertions, etc
- AI could also help you to complete the proof!
  - AutoVerus: https://github.com/microsoft/verus-proof-synthesis
  - Verus-Copilot: https://github.com/microsoft/verus-copilot-vscode
    - VSCode extension