Neuro-Symbolic Developer Tools for Analyzing, Executing, and Repairing Code

Michael Pradel

University of Stuttgart

Joint work with Beatriz Souza and Islem Bouzenia

SOLA SoftwareLab University of Stuttgart Germany

erc European Research Council

Developers Need Tools

Key feature of humans:

Ability to develop tools

Software development tools, e.g., code completion, bug detection, automated repair

How to build effective developer tools?

Traditional answer:
Symbolic reasoning

- Manually crafted, logic-based rules
- Deterministic, precise reasoning
- Based on formal PL semantics

Traditional answer: Symbolic reasoning

- Manually crafted, logic-based rules
- Deterministic, precise reasoning
- Based on formal PL semantics

Recent answer: Neural reasoning

- Models learned from data (e.g., huge amounts of code)
- Probabilistic reasoning
- Based on “naturalness” of code

Traditional answer: Symbolic reasoning

- Manually crafted, logic-based rules
- Deterministic, precise reasoning
- Based on formal PL semantics

→ **Needs heuristics to be practical**

→ **Fails to understand developer intention**

Recent answer: Neural reasoning

- Models learned from data (e.g., huge amounts of code)
- Probabilistic reasoning
- Based on “naturalness” of code

Traditional answer: Symbolic reasoning

- Manually crafted, logic-based rules
- Deterministic, precise reasoning
- Based on formal PL semantics

→ Needs heuristics to be practical

→ Fails to understand developer intention

Recent answer: Neural reasoning

- Models learned from data (e.g., huge amounts of code)
- Probabilistic reasoning
- Based on “naturalness” of code

→ Easily misses well-known facts and rules

→ Hard to understand and debug

Traditional answer:
Symbolic reasoning

- Manually crafted, logic-based rules
- Deterministic, precise reasoning
- Based on formal PL semantics

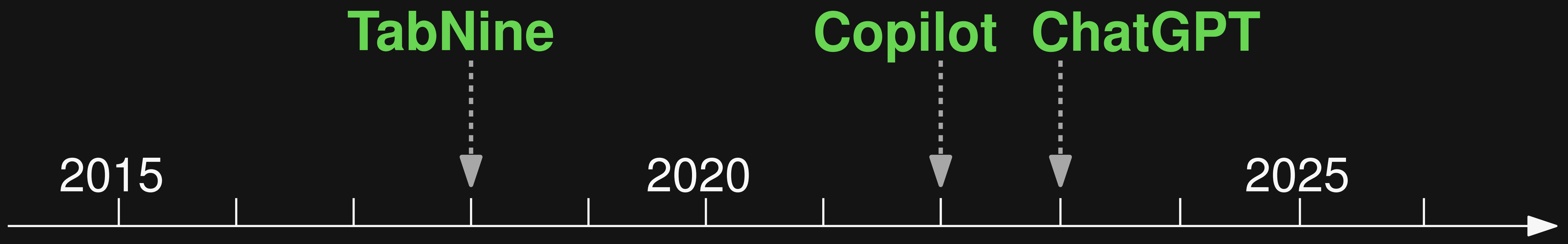
↔

Recent answer:
Neural reasoning

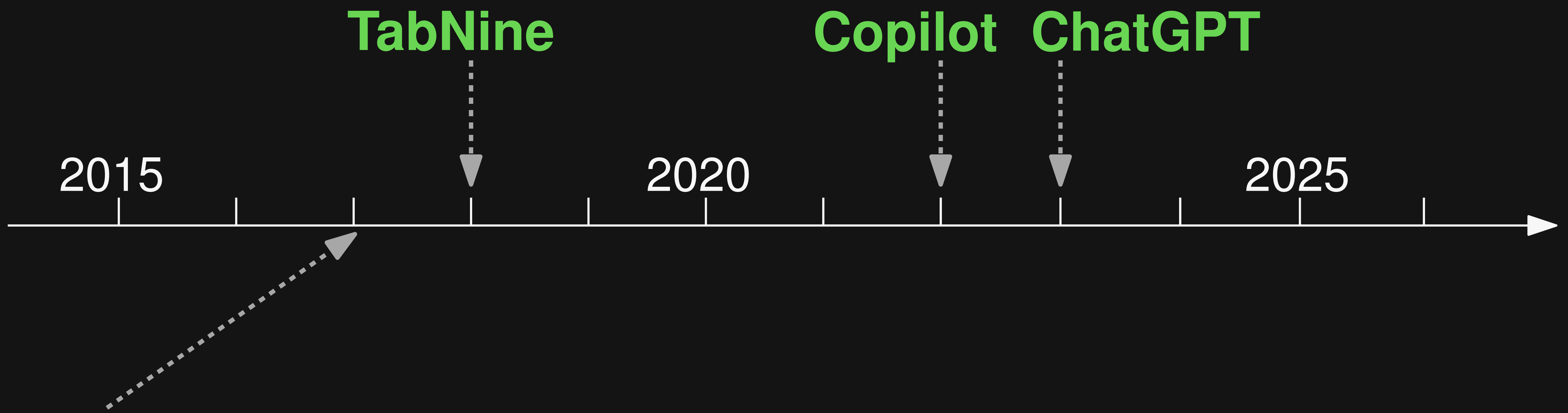
- Models learned from data (e.g., huge amounts of code)
- Probabilistic reasoning
- Based on “naturalness” of code

Get the best of both worlds:
Neuro-symbolic developer tools

A Bit of History



A Bit of History



DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

Example: DeepBugs

```
function setPoint(x, y) { ... }

var x_dim = 23;
var y_dim = 5;
setPoint(y_dim, x_dim);
```

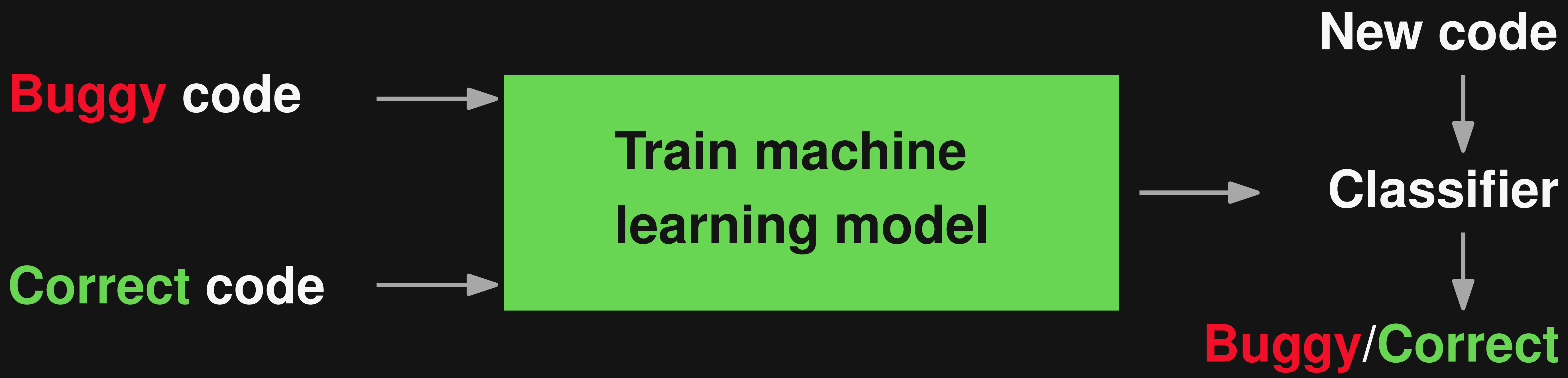
Example: DeepBugs

```
function setPoint(x, y) { ... }

var x_dim = 23;
var y_dim = 5;
setPoint(y_dim, x_dim);
```

Incorrect order of arguments

DeepBugs: Learning to Find Bugs

Train a model to distinguish correct from buggy code



Buggy code

Correct code

Train machine learning model

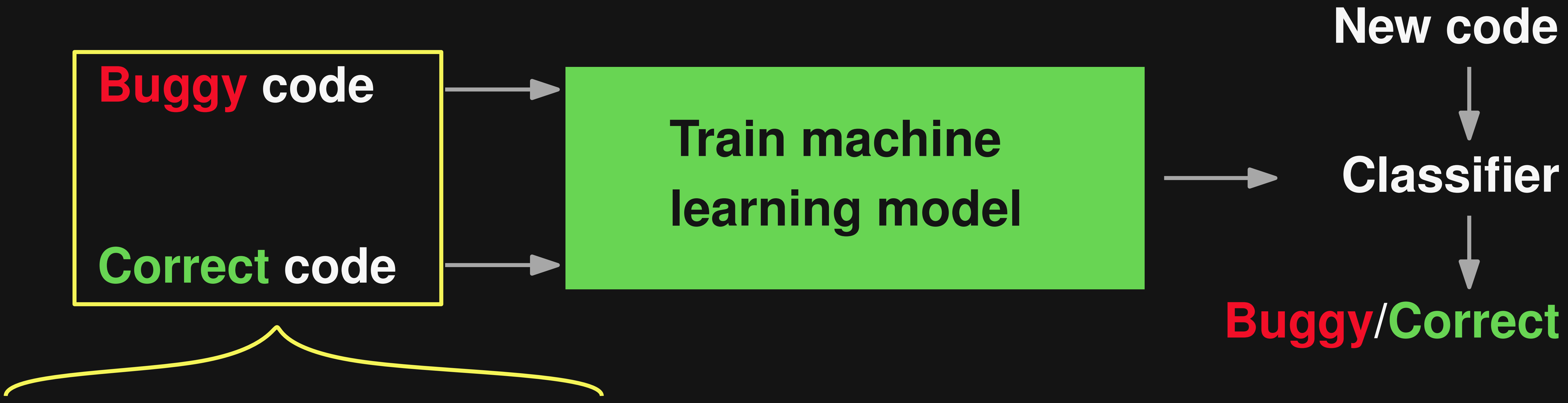
New code

Classifier

Buggy/Correct

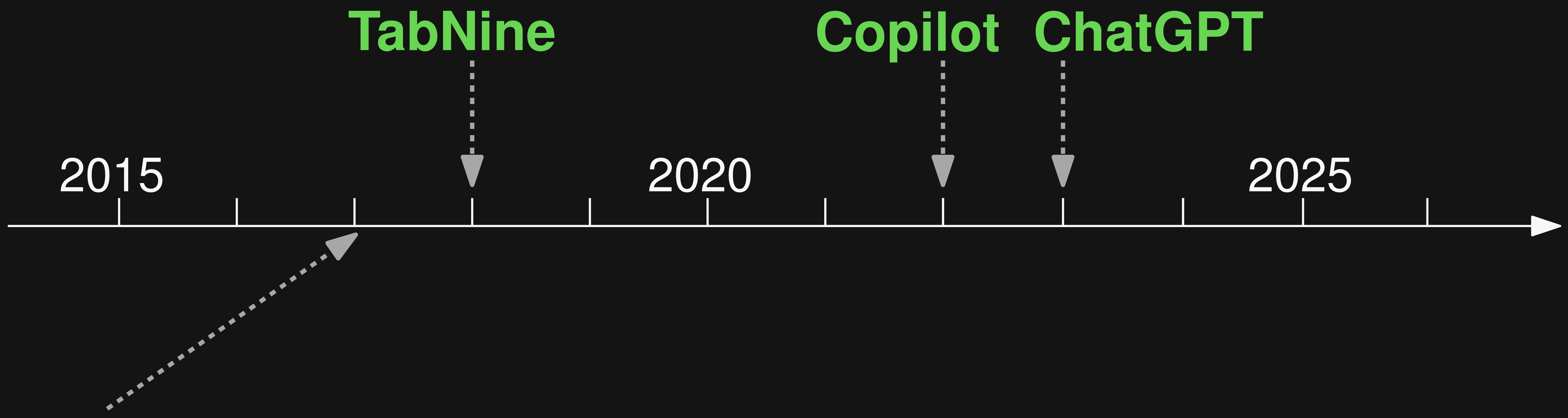
DeepBugs: Learning to Find Bugs

Train a model to distinguish correct from buggy code



Created via program transformations

A Bit of History

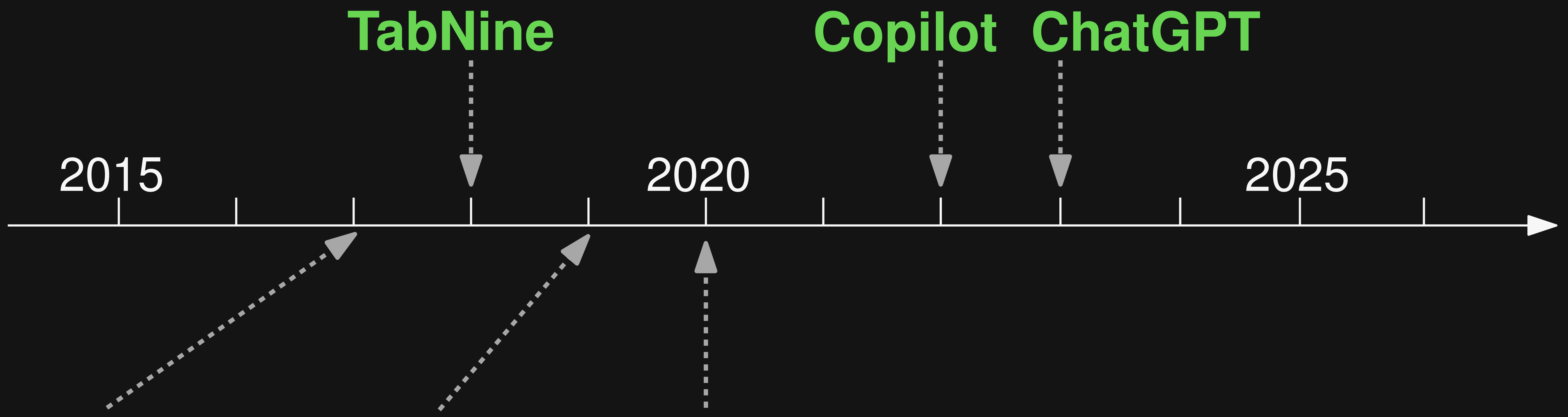


DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

A Bit of History



TabNine

Copilot

ChatGPT

2015

2020

2025

DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE'19) (FSE'20)

Neural type inference with static validation

Example: TypeWriter

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Example: TypeWriter

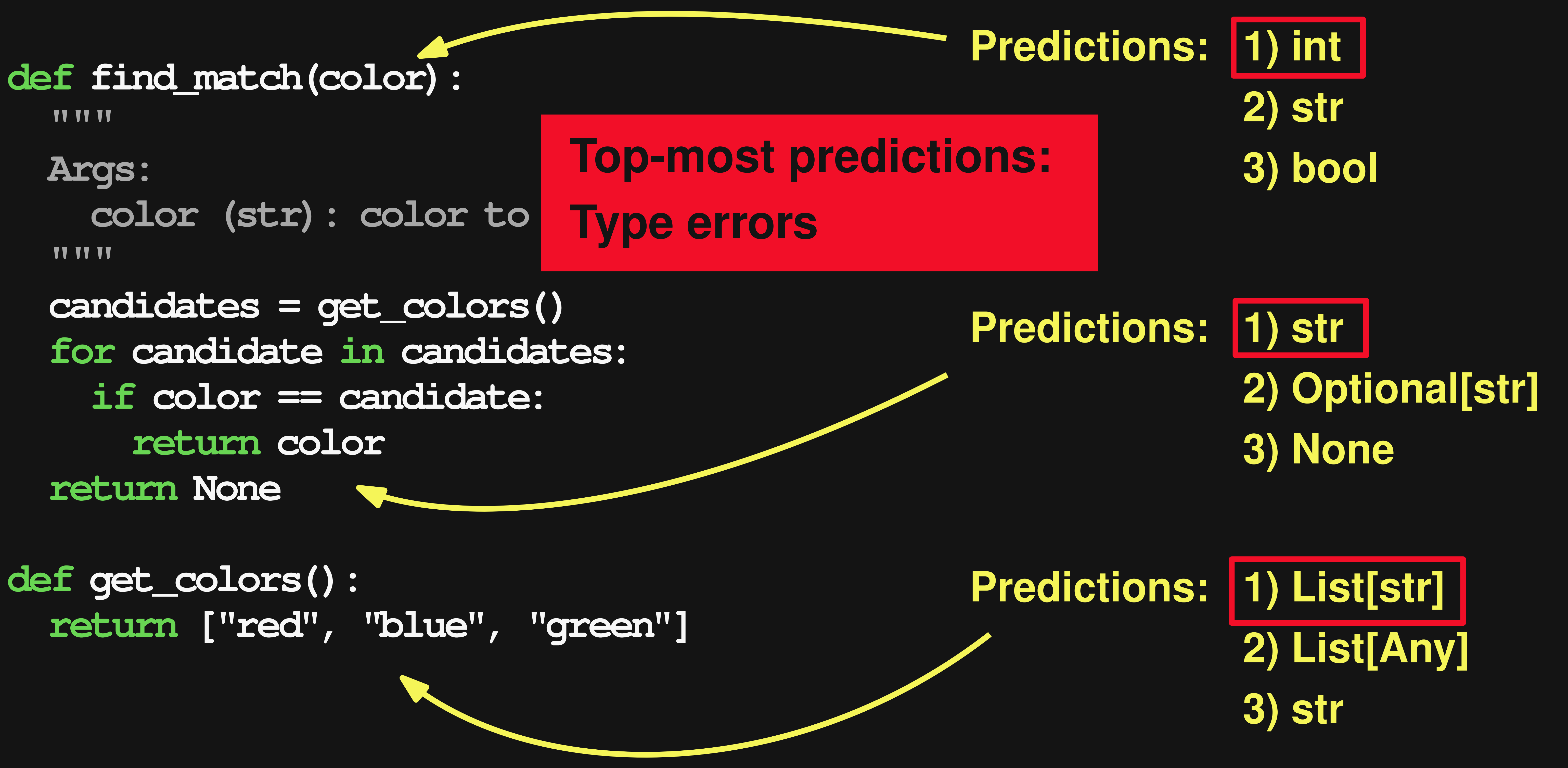
```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Predictions: 1) int
2) str
3) bool

Predictions: 1) str
2) Optional[str]
3) None

Predictions: 1) List[str]
2) List[Any]
3) str

Example: TypeWriter



```
def find_match(color):
  """
  Args:
    color (str): color to
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

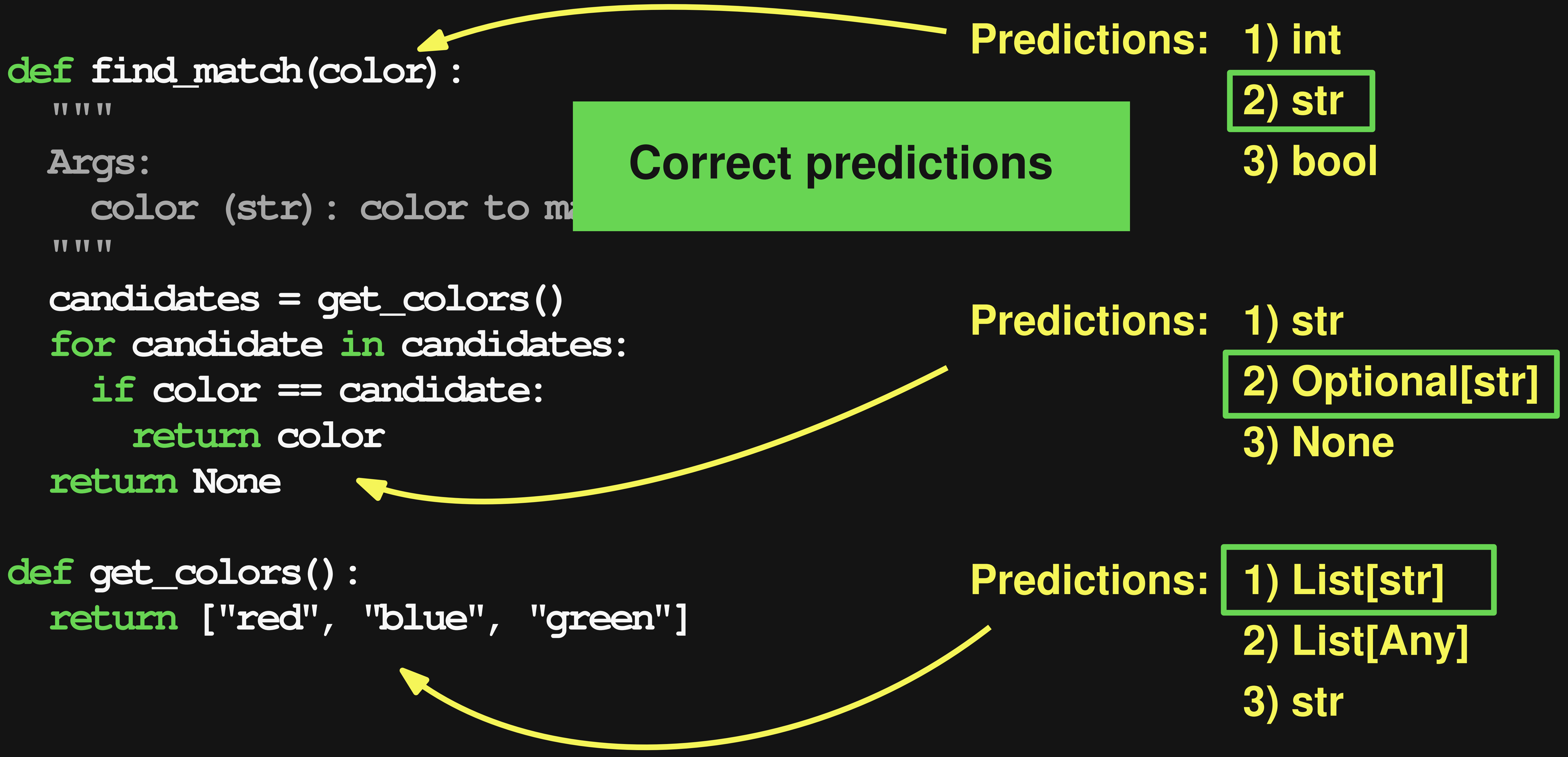
Top-most predictions:
Type errors

Predictions: 1) int
2) str
3) bool

Predictions: 1) str
2) Optional[str]
3) None

Predictions: 1) List[str]
2) List[Any]
3) str

Example: TypeWriter



```
def find_match(color):
  """
  Args:
    color (str): color to m
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

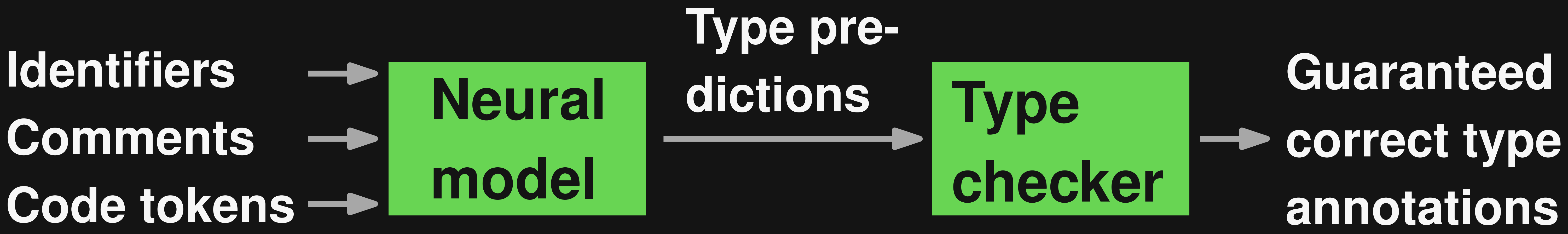
Correct predictions

Predictions: 1) int
[✓ 2) str]
3) bool

Predictions: 1) str
[✓ 2) Optional[str]]
3) None

Predictions: [✓ 1) List[str]]
2) List[Any]
3) str

TypeWiter: Neural Type Prediction



Identifiers

Comments

Code tokens

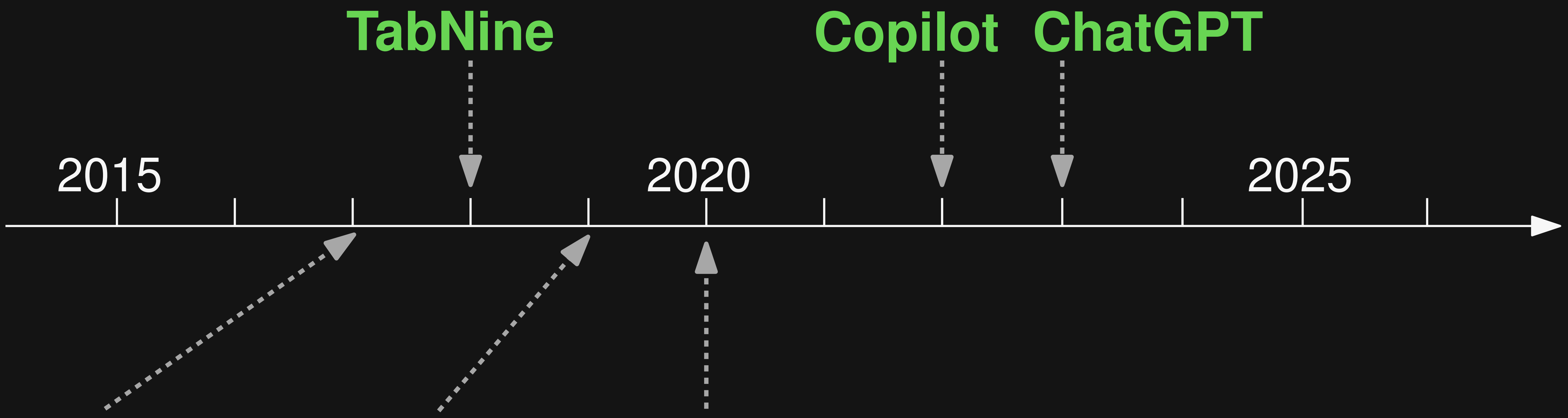
Neural model

Type predictions

Type checker

Guaranteed correct type annotations

This Talk



DeepBugs

(OOPSLA'18)

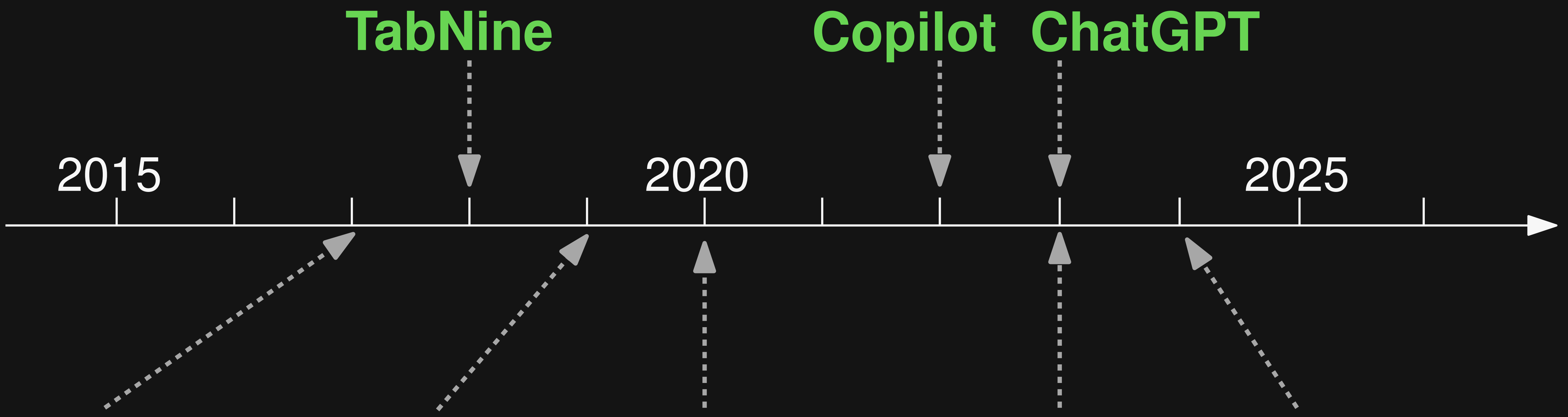
Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE'19) (FSE'20)

Neural type inference with static validation

This Talk



TabNine

Copilot

ChatGPT

2015

2020

2025

DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE'19) (FSE'20)

Neural type inference with static validation

LExecutor

(FSE'23)

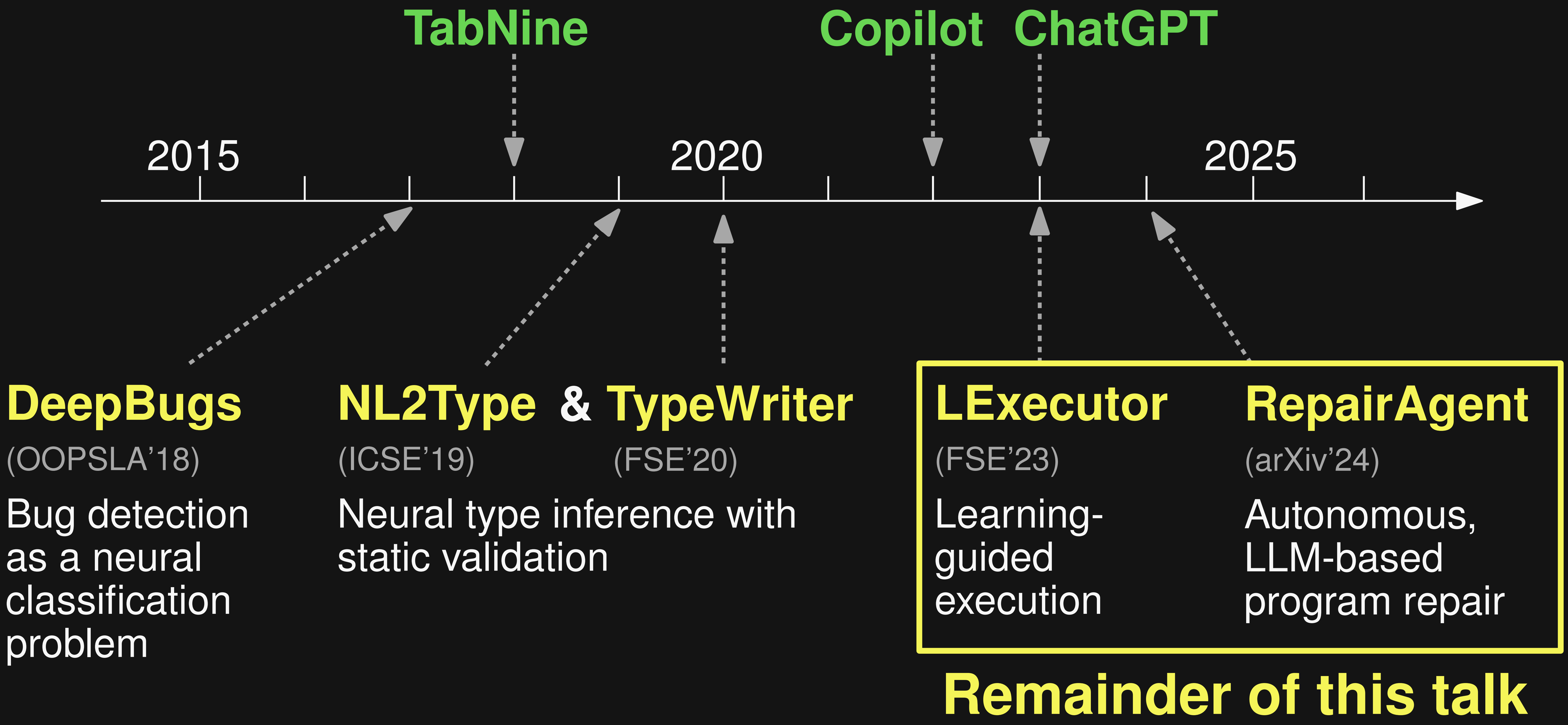
Learning-guided execution

RepairAgent

(arXiv'24)

Autonomous, LLM-based program repair

This Talk



DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

NL2Type & TypeWriter

(ICSE'19) (FSE'20)

Neural type inference with static validation

LExecutor

(FSE'23)

Learning-guided execution

RepairAgent

(arXiv'24)

Autonomous, LLM-based program repair

Remainder of this talk

Motivation

Imagine you want to execute this code:

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Motivation

Imagine you want to execute this code:

Missing variable

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Motivation

Imagine you want to **execute this code:**

Missing function

Missing variable

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]


# ...
```

Motivation

Imagine you want to execute this code:

Missing function

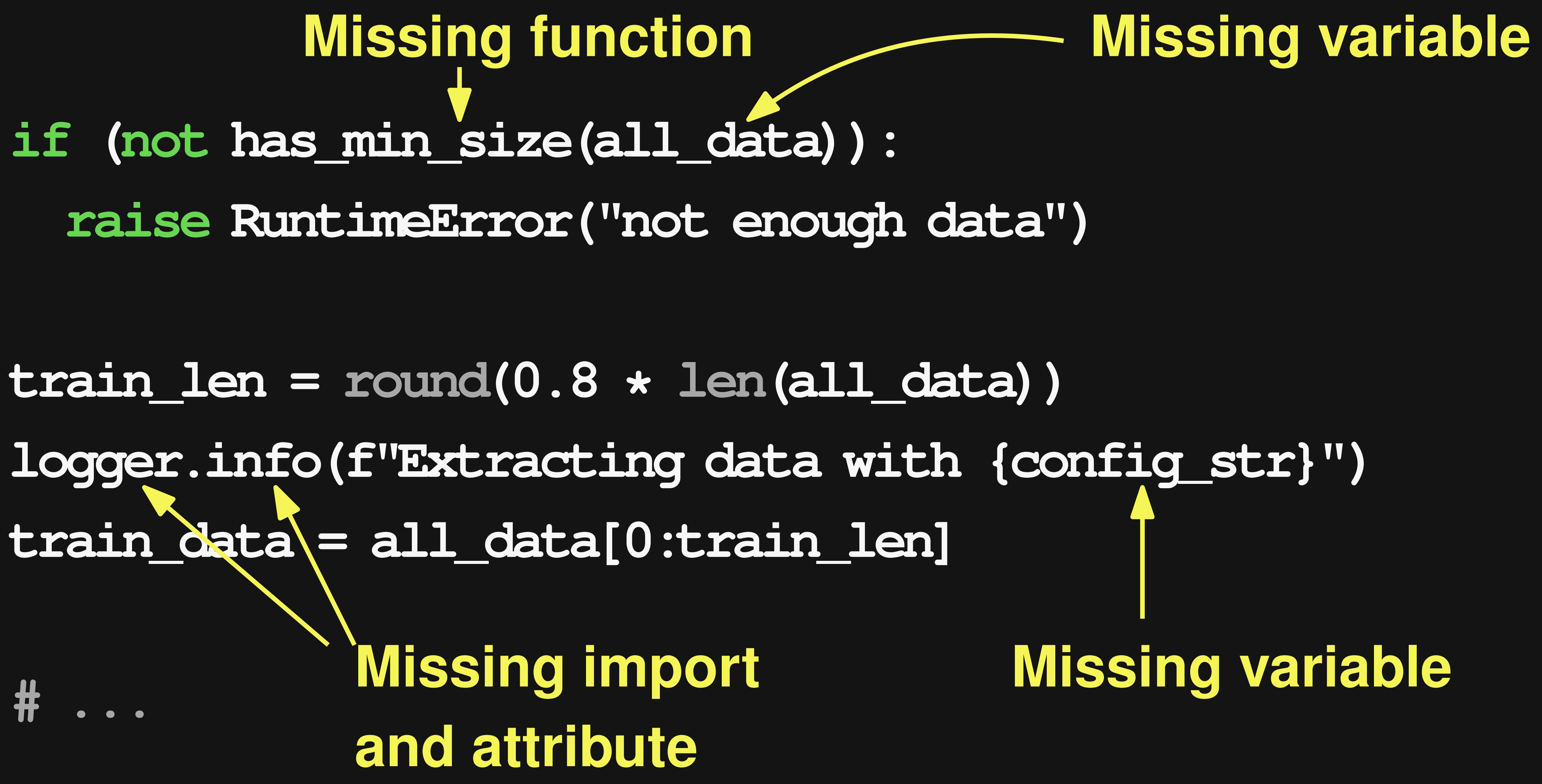
Missing variable

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Missing variable

Motivation

Imagine you want to execute this code:



```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Missing function

Missing variable

Missing import and attribute

Missing variable

Why Execute Incomplete Code?

Enables various dynamic analyses

- Check for exceptions and assertion violations
- Compare two code snippets for semantic equivalence
- Validate static analysis warnings
- Validate and filter LLM-predicted code
- $\langle \textit{Your favorite application here} \rangle$

Executing Ain’t Easy

Lots of incomplete code:

- Code snippets from Stack Overflow
- Code generated by language models
- Code extracted from deep inside complex projects

Executing Ain’t Easy

Lots of incomplete code:

- Code snippets from Stack Overflow
- Code generated by language models
- Code extracted from deep inside complex projects

Can we automatically fill in the missing information?

LExecutor

Learning-guided approach for executing arbitrary code snippets

- Predict missing values with neural model
- Inject values into the execution

Underconstrained execution:
No guarantee that values are realistic

Example: LExecutor

Let’s “lexecute” the motivating example:

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Example: LExecutor

Let’s “lexecute” the motivating example:

Non-empty list

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]


# ...
```

Example: LExecutor

Let’s “lexecute” the motivating example:

Function that returns `True`

Non-empty list

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Example: LExecutor

Let’s “lexecute” the motivating example:

Function that returns `True`

Non-empty list

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Non-empty string

Example: LExecutor

Let’s “lexecute” the motivating example:

Function that returns `True`

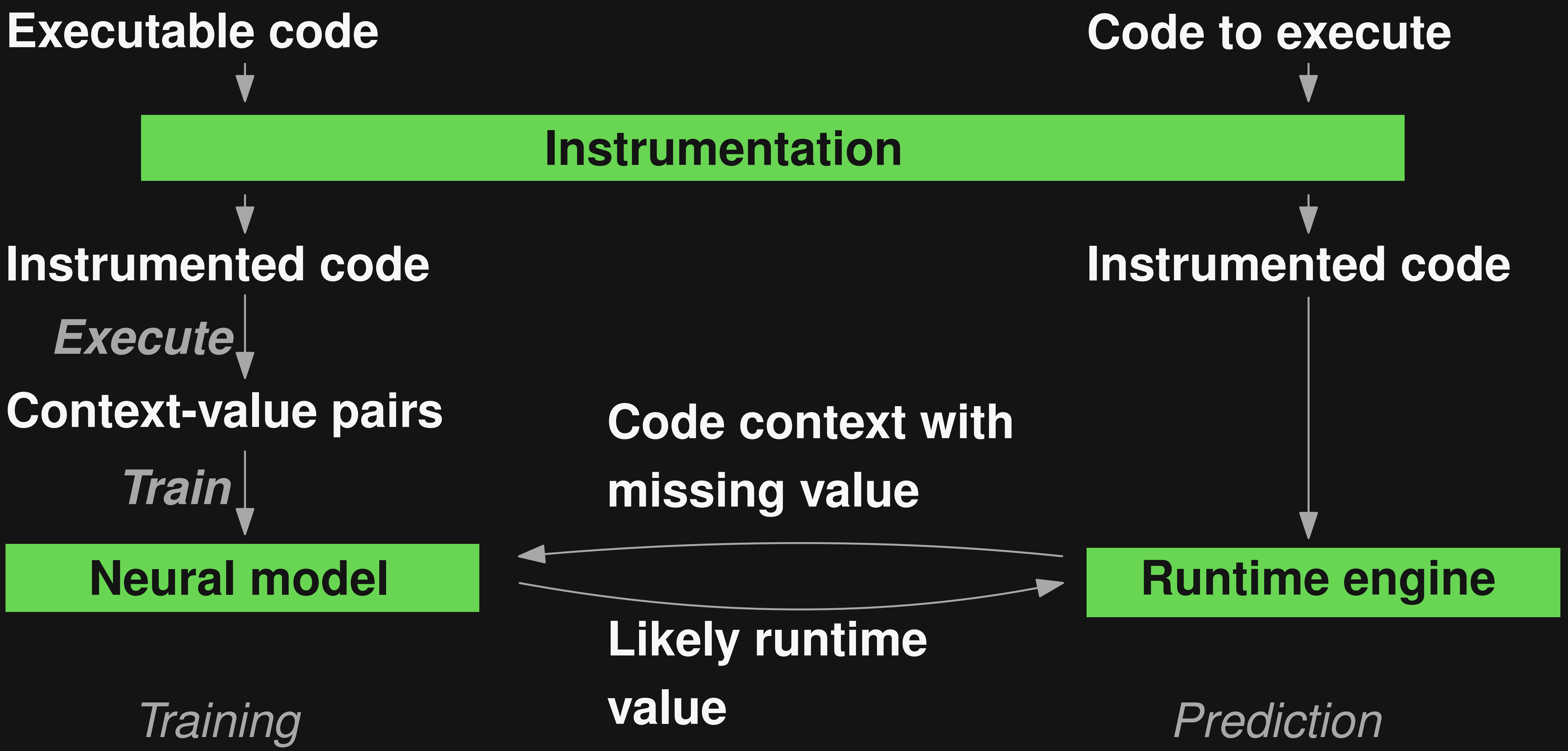
Non-empty list

```
if (not has_min_size(all_data)):
  raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))
logger.info(f"Extracting data with {config_str}")
train_data = all_data[0:train_len]

# ...
```

Object with a method

Non-empty string

Overview of LExecutor



Executable code

Code to execute

Instrumentation

Instrumented code

Execute

Context-value pairs

Train

Neural model

Instrumented code

Runtime engine

Code context with missing value

Likely runtime value

Training

Prediction

Code Instrumentation

- **Wrap reads of variables, reads of attributes, and function calls**
  - During training: Observe runtime values
  - During prediction: Inject missing values
- **AST-based source-to-source instrumentation**
  - Drop-in replacement for original code
  - Same semantics, except for reads of values

Example

Original code:

```
x = foo()
y = x.bar + z
```

Example

Original code:

```
x = foo()
y = x.bar + z
```

Instrumented code:

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Example

Original code:

```
x = foo()
y = x.bar + z
```

Instrumented code:

Lambda function to postpone the read (to be called by runtime engine)

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Example

Original code:

```
x = foo()
y = x.bar + z
```

Wrapper for variable reads

Lambda function to postpone the read (to be called by runtime engine)

Instrumented code:

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Example

Original code:

```
x = foo()
y = x.bar + z
```

Wrapper for variable reads

Lambda function to postpone the read (to be called by runtime engine)

Instrumented code:

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Wrapper for calls

Example

Original code:

```
x = foo()
y = x.bar + z
```

Wrapper for variable reads

Lambda function to postpone the read (to be called by runtime engine)

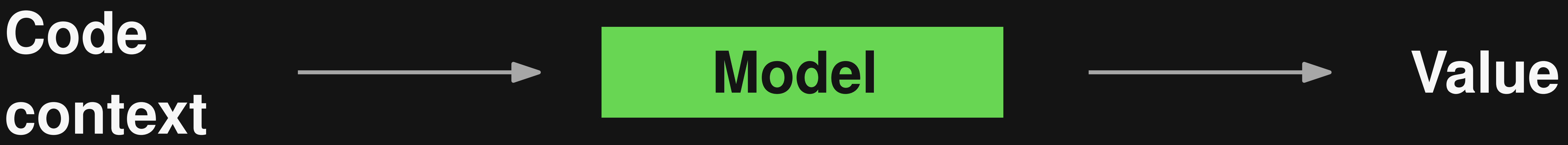
Instrumented code:

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Wrapper for calls

Wrapper for attribute reads

Neural Model: Data Representation

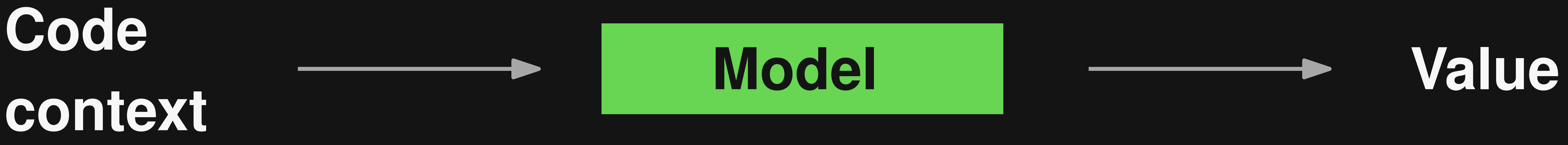


Code context

Model

Value

Neural Model: Data Representation



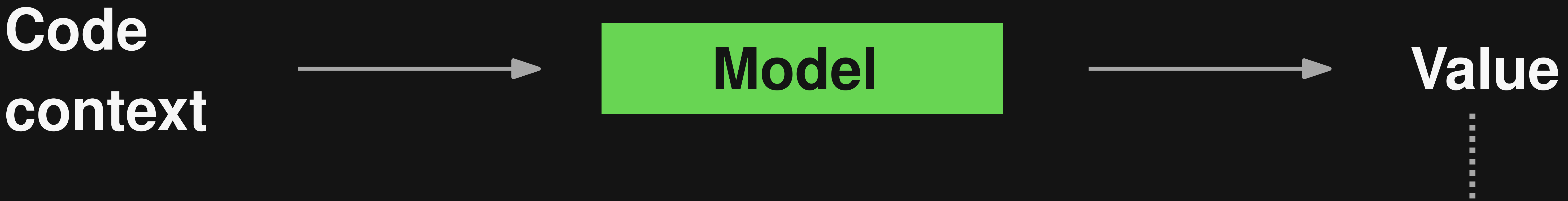
$$n \ \langle sep \rangle \ k \ \langle sep \rangle \ c_{pre} \ \langle mask \rangle \ c_{post}$$

Name used to refer to a value

Kind of value (variable, attribute, or return value)

Code before/after the reference to the value

Neural Model: Data Representation



Concrete values abstracted into 23 classes, e.g.,

- `None`, `True`, `False`
- Negative/zero/positive integer
- Empty/non-empty list
- Callable

Train & Predict

- **Fine-tune a pre-trained CodeT5 model**
- **During prediction:**
  For each use of a value
  - Read value and, if it exists, return it
  - If undefined, query the model and return its prediction

Evaluation

- **Training data**
  - 226k unique value-use events from five projects
- **Code snippets to execute**
  - Open-source functions: 1,000 extracted from five projects
  - Stack Overflow snippets: 462 syntactically correct code snippets in answers to 1,000 Python-related questions

Accuracy

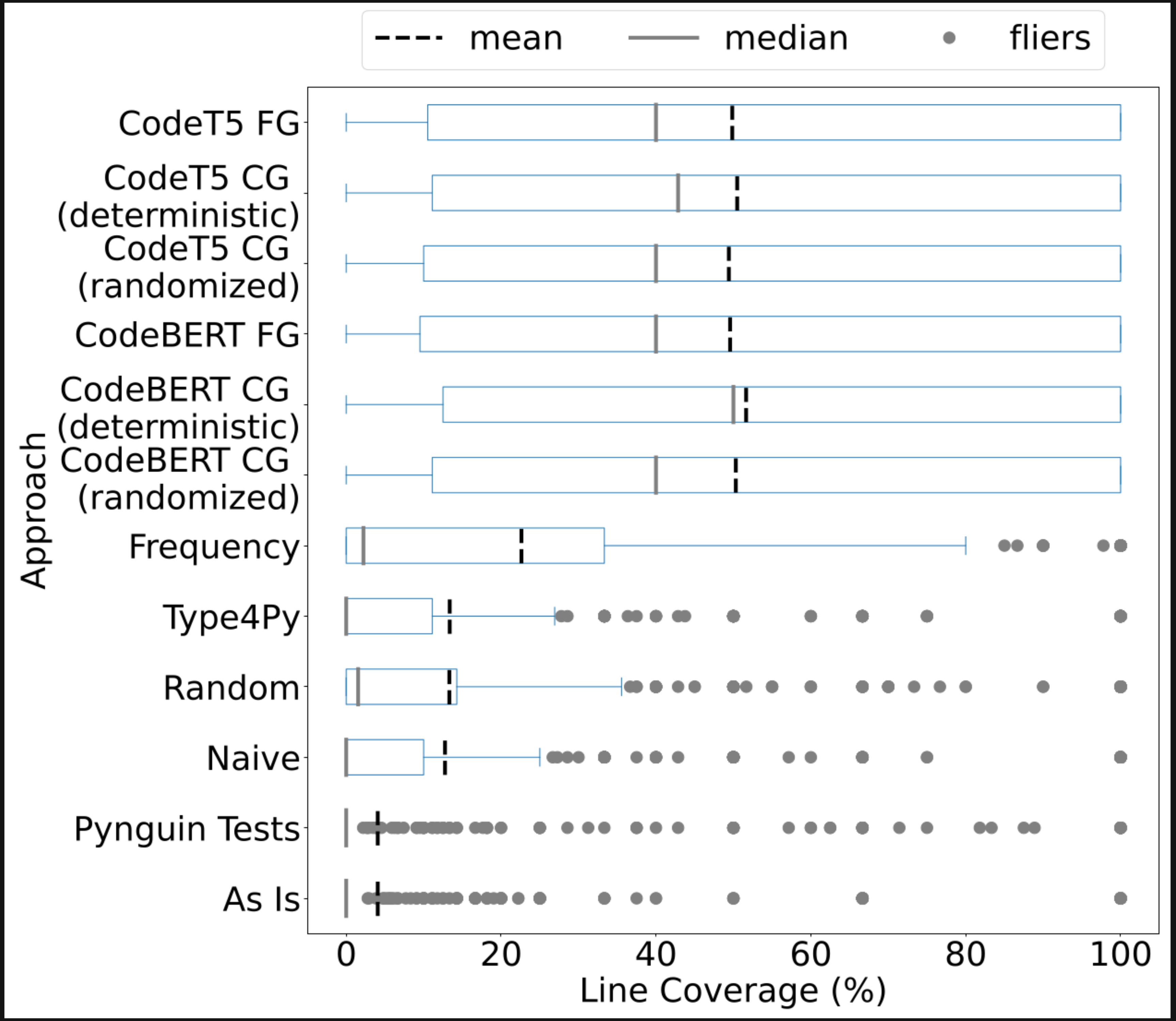
How accurate is the model at predicting realistic values?

23 abstract classes of values

12 abstract classes of values

	Value abstraction			
	Fine-grained		Coarse-grained	
	CodeT5	CodeBERT	CodeT5	CodeBERT
Top-1	80.1%	79.5%	88.1%	87.3%
Top-3	88.4%	94.5%	92.1%	96.5%
Top-5	91.7%	96.8%	94.2%	98.2%

Effectiveness at Covering Code (Open-source functions)



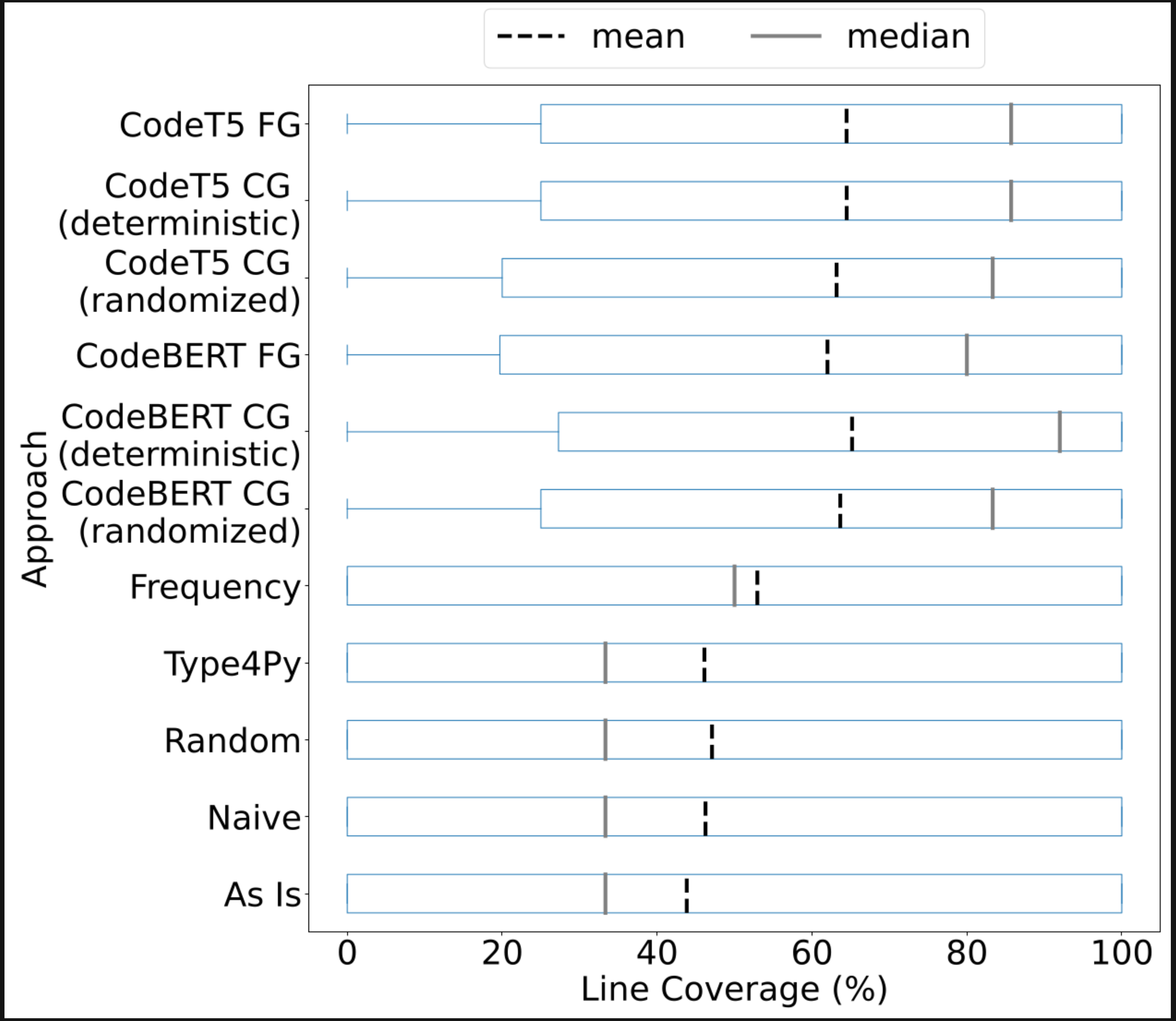
Variants of LExecutor

Neural type prediction

State-of-the-art unit test generator

Just run the code

Effectiveness at Covering Code (Stack Overflow snippets)



Variants of LExecutor

Neural type prediction

Just run the code

Summary: LExecutor

Symbolic reasoning

- Execute code using standard PL semantics
- Enables various dynamic analyses

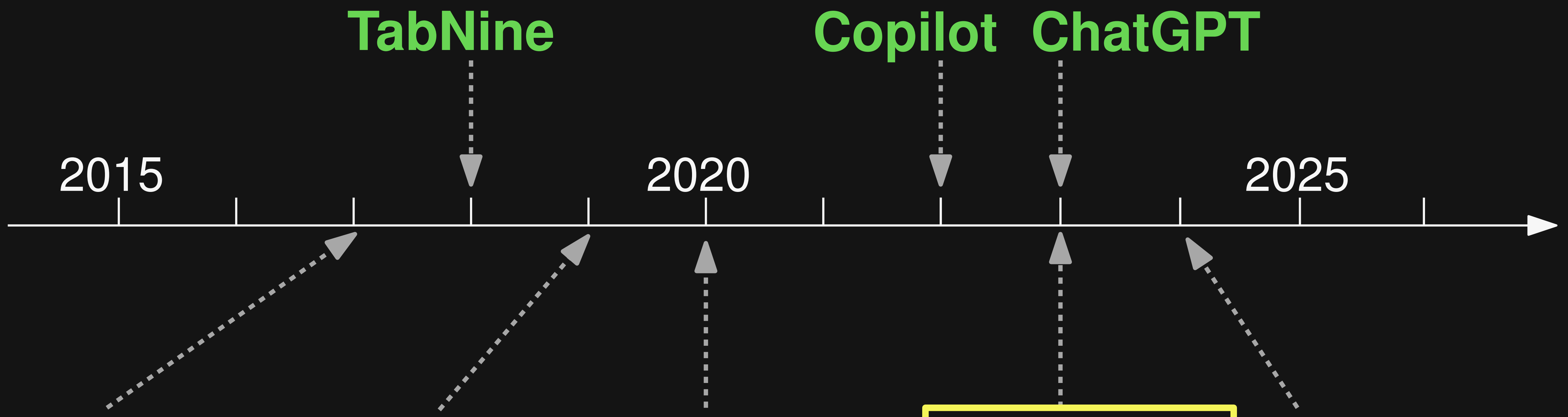
↔

Neural reasoning

- Fill-in missing information on demand during the execution
- Enables execution of otherwise “unexecutable” code

- Paper: “LExecutor: Learning-Guided Execution” (FSE, 2023, Distinguished Paper Award)
- Code: https://github.com/michaelpradel/LExecutor

Timeline



TabNine

Copilot

ChatGPT

2015

2020

2025

DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

NL2Type & TypeWriter

(ICSE'19) (FSE'20)

Neural type inference with static validation

LExecutor

(FSE'23)

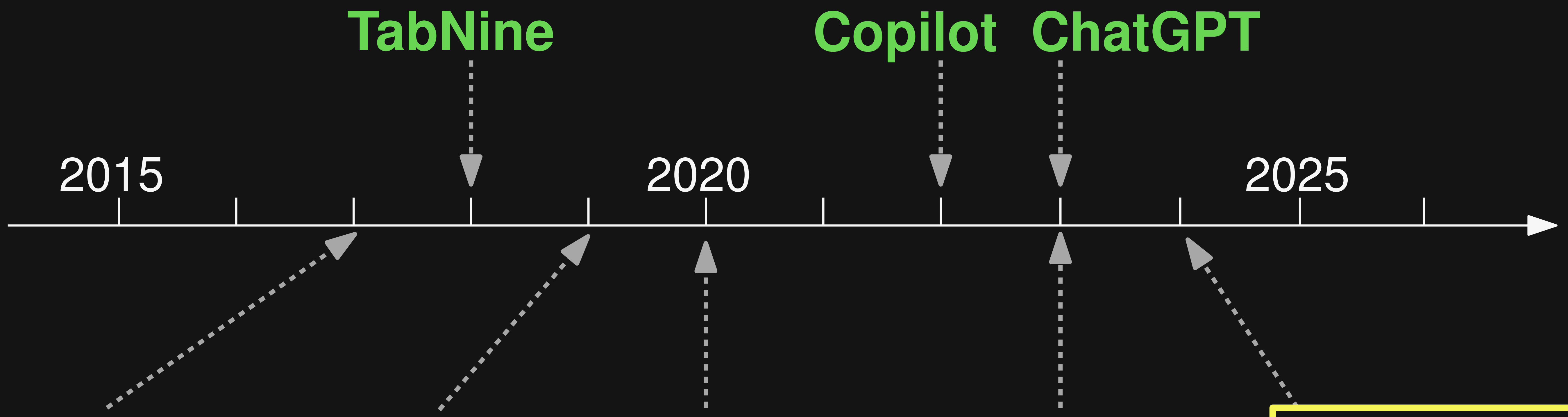
Learning-guided execution

RepairAgent

(arXiv'24)

Autonomous, LLM-based program repair

Timeline



DeepBugs
(OOPSLA'18)
Bug detection as a neural classification problem

NL2Type & **TypeWriter**
(ICSE'19) (FSE'20)
Neural type inference with static validation

LExecutor
(FSE'23)
Learning-guided execution

RepairAgent
(arXiv'24)
Autonomous, LLM-based program repair

Automated Program Repair

- **Problem: Too many bugs, huge effort to fix them**
- **State of the art: Learning-based repair**
  - □ Neural model predicts a fix
  - □ Validation via test suite and/or static analysis

Automated Program Repair

Claire Le Goues, Michael Pradel, Abhik Roychoudhury

Communications of the ACM, 62(12), 2019

Limitations of Current Tools

- **Fixed input: Lines around the bug location**
  - Human developers: Actively gather additional information

- **Fixed algorithm: Repeatedly generate and validate patches**
  - Human developers: Complex interactions with various tools

Examples for current tools:
ChatRepair (Xia et al., 2023) and ITER (Ye et al., 2024)

RepairAgent

New paradigm: LLM takes the decisions autonomously

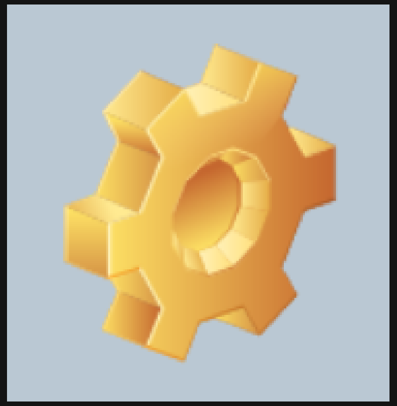
- Agent invokes developer tools:
  LLM mimics human-like actions
- Interaction via natural language:
  LLM mimics human-like reasoning

Example: RepairAgent

LLM agent

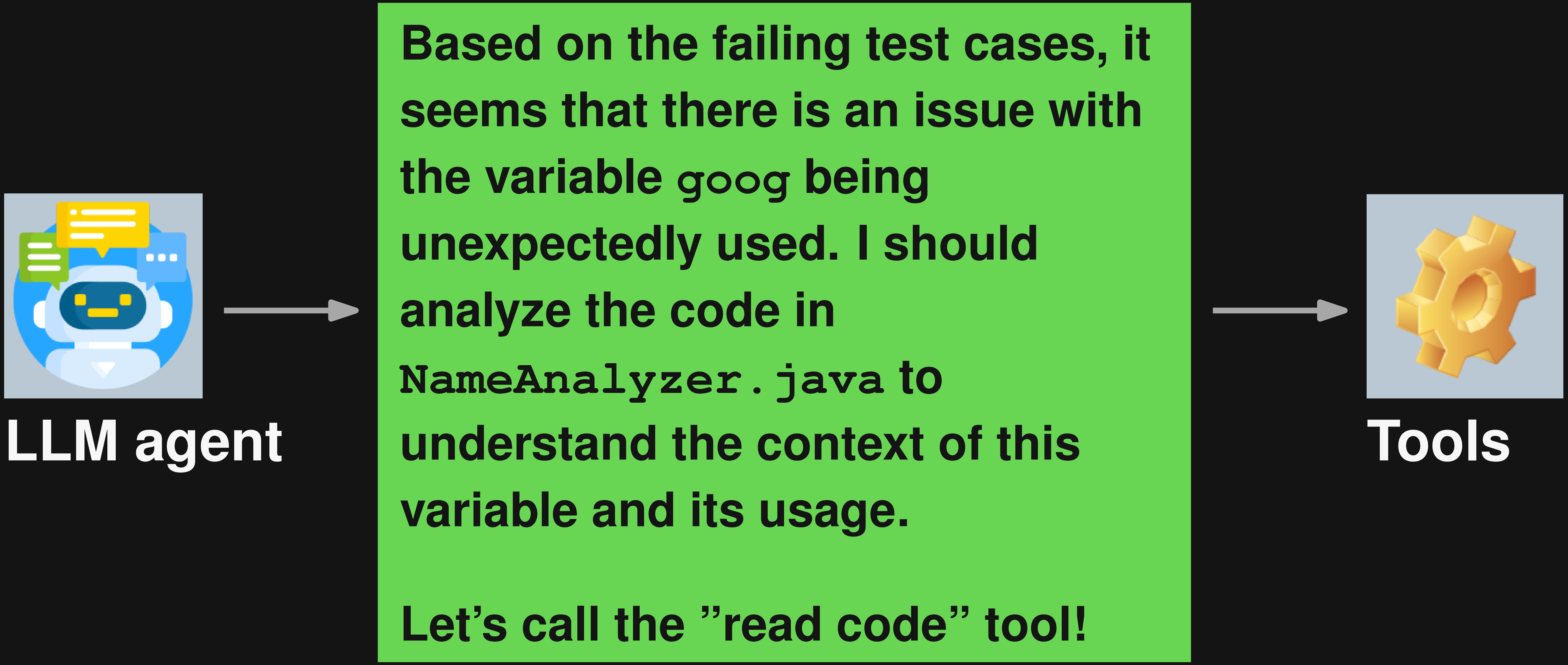
You are an AI assistant specialized in fixing bugs in Java code. Your objective is to autonomously understand and fix the following bug:
[Details of failing test case]

Use the following tools:
[Details on available tools]



Tools

Example: RepairAgent



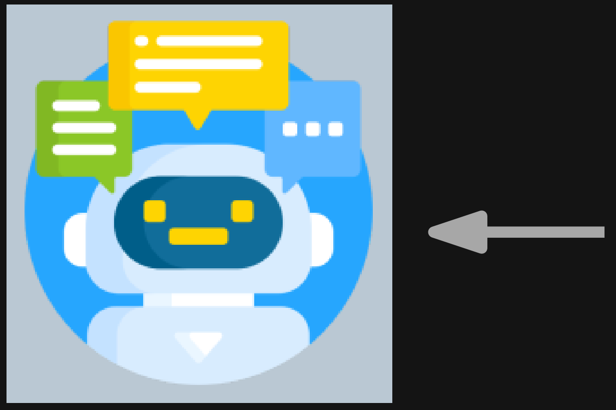
LLM agent

Based on the failing test cases, it seems that there is an issue with the variable `goog` being unexpectedly used. I should analyze the code in `NameAnalyzer.java` to understand the context of this variable and its usage.

Let’s call the ”read code” tool!

Tools

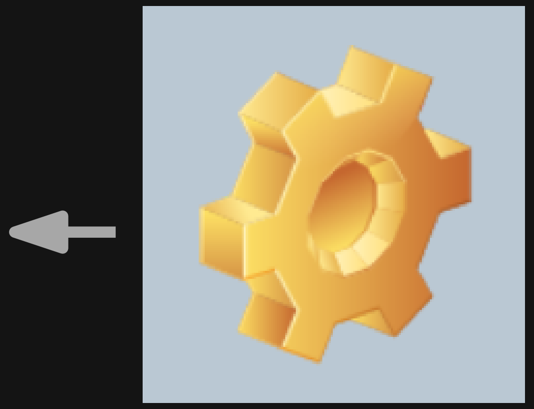
Example: RepairAgent



LLM agent

NameAnalyzer.java:

```
...
} else if (n.isCall()) {
  Node nameNode = n.getFirstChild();
  NameInformation ns =
      createNameInformation(t, nameNode, n);
  if (ns != null && ns.onlyAffectsClassDef) {
    JsName name = getName(ns.name, false);
    if (name != null) {
      refNodes.add(new ClassDefFctNode(
      name, n, parent, parent.getParent()));
    }
  }
...
```



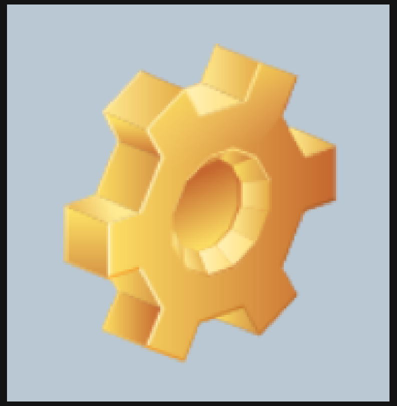
Tools

Example: RepairAgent

LLM agent →

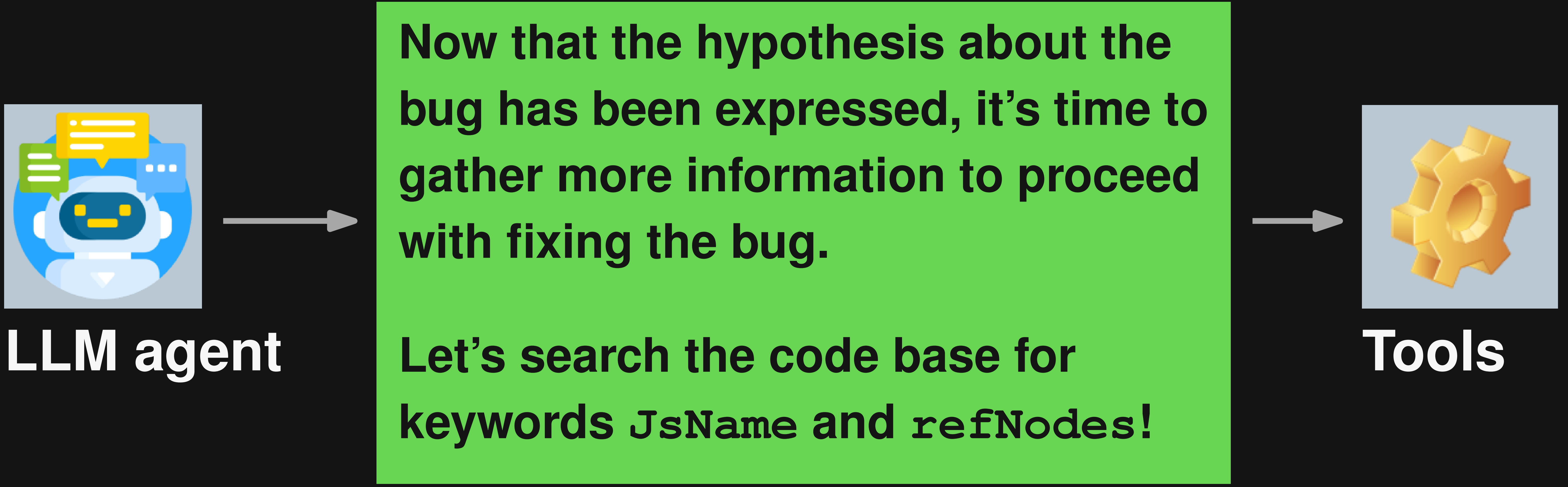
After analyzing the code snippet in `NameAnalyzer.java`, it appears that the issue might be related to the usage of the `JsName` object and the `refNodes` list. The unexpected variable `goog` error could be due to a missing check or incorrect handling of variables.

Let’s work under this hypothesis!



Tools

Example: RepairAgent



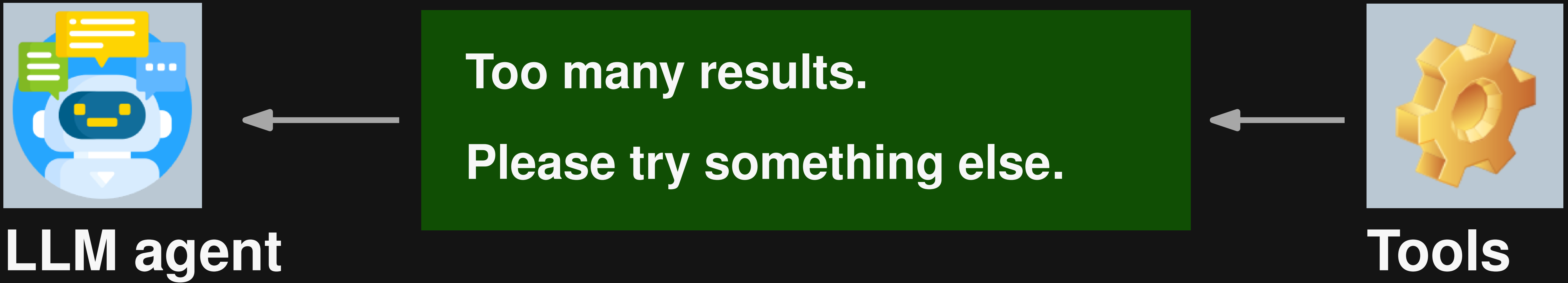
LLM agent

Now that the hypothesis about the bug has been expressed, it's time to gather more information to proceed with fixing the bug.

Let's search the code base for keywords `JsName` and `refNodes`!

Tools

Example: RepairAgent



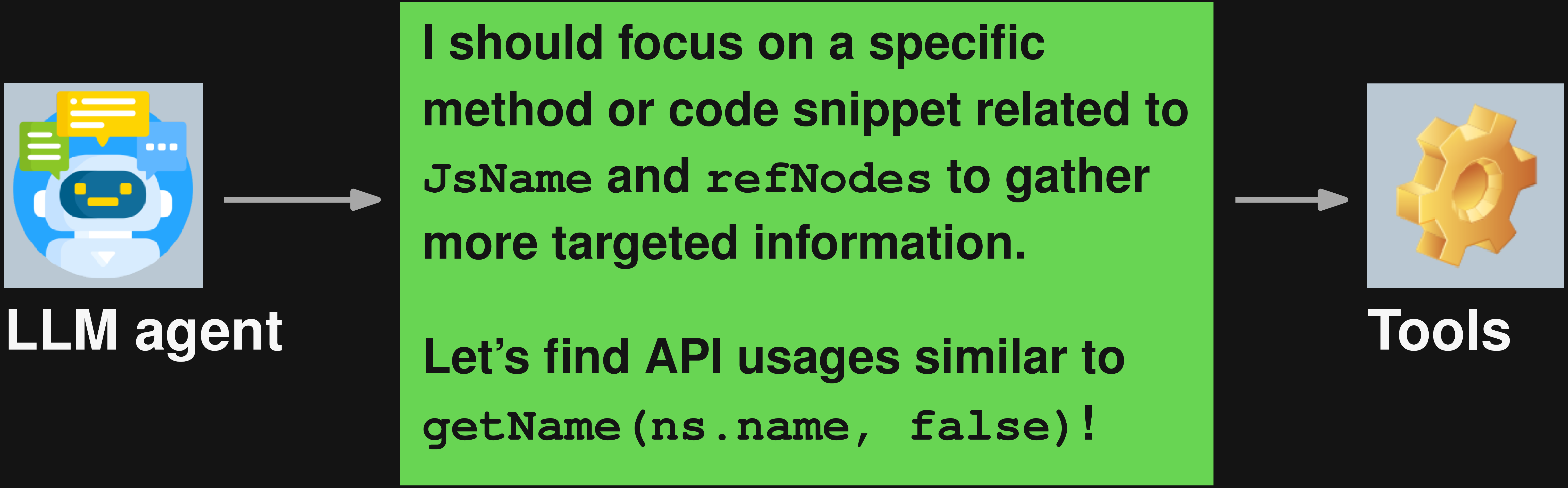
LLM agent

Too many results.

Please try something else.

Tools

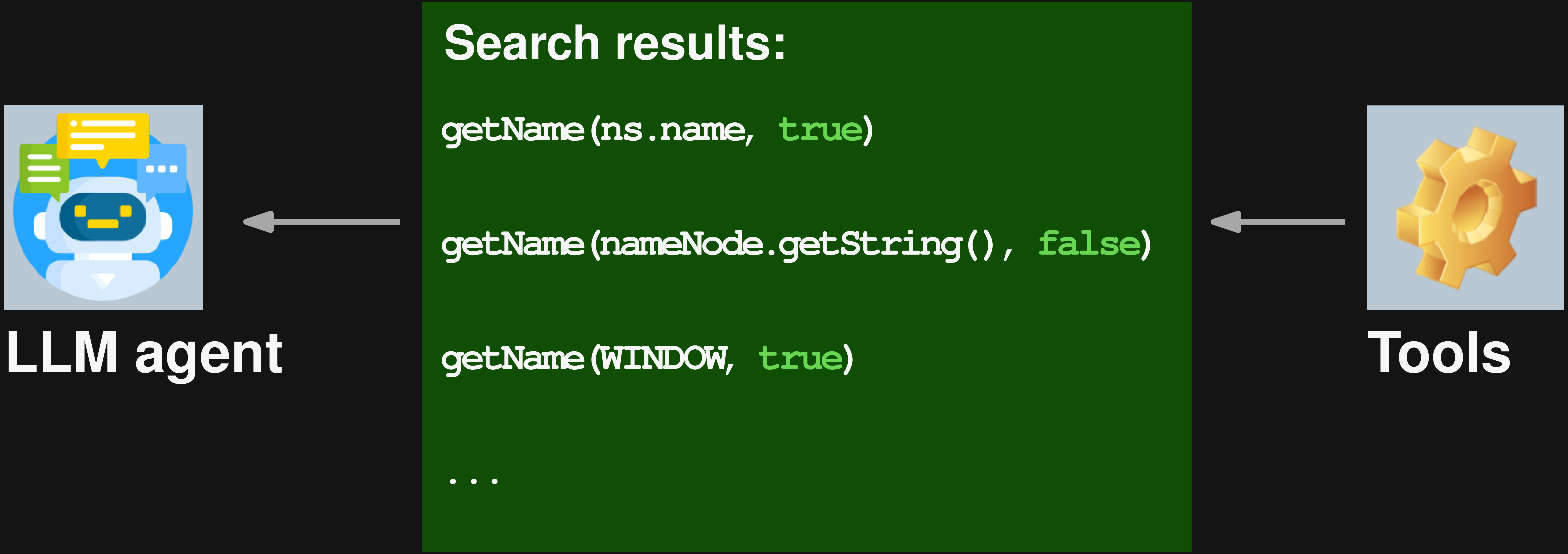
Example: RepairAgent



I should focus on a specific method or code snippet related to `JsName` and `refNodes` to gather more targeted information.

Let’s find API usages similar to `getName(ns.name, false)`!

Example: RepairAgent



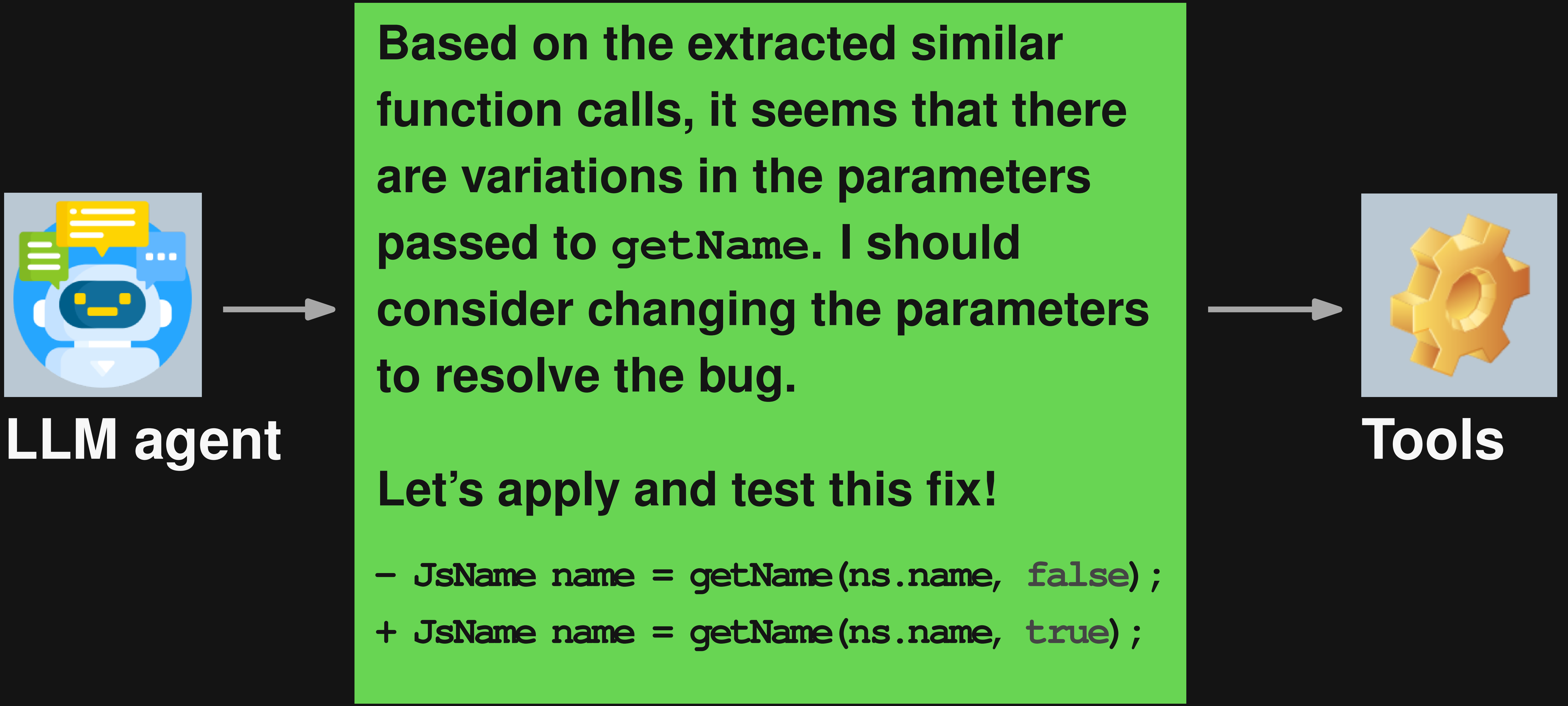
LLM agent

Search results:

```
getName(ns.name, true)

getName(nameNode.getString(), false)

getName(WINDOW, true)

...
```

Tools

Example: RepairAgent



LLM agent

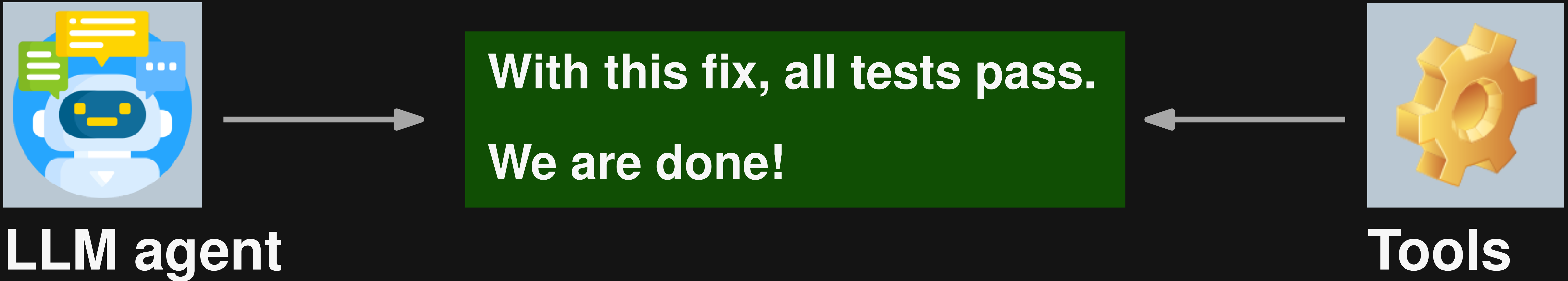
Based on the extracted similar function calls, it seems that there are variations in the parameters passed to `getName`. I should consider changing the parameters to resolve the bug.

Let’s apply and test this fix!

```
- JsName name = getName(ns.name, false);
+ JsName name = getName(ns.name, true);
```

Tools

Example: RepairAgent



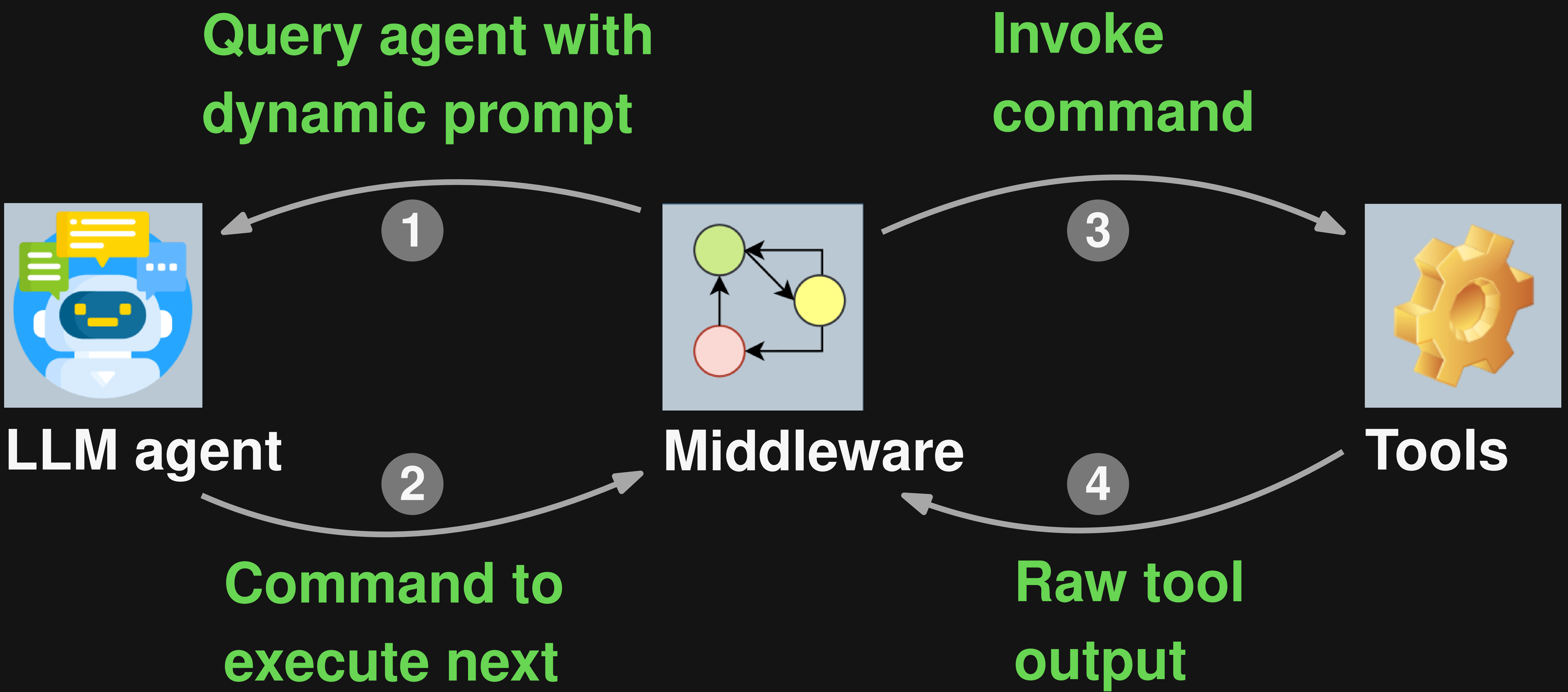
LLM agent

With this fix, all tests pass.

We are done!

Tools

Overview of RepairAgent



LLM agent

Middleware

Tools

1 Query agent with dynamic prompt

2 Command to execute next

3 Invoke command

4 Raw tool output

Dynamic Prompt

Prompt isn’t fixed, but changes over time

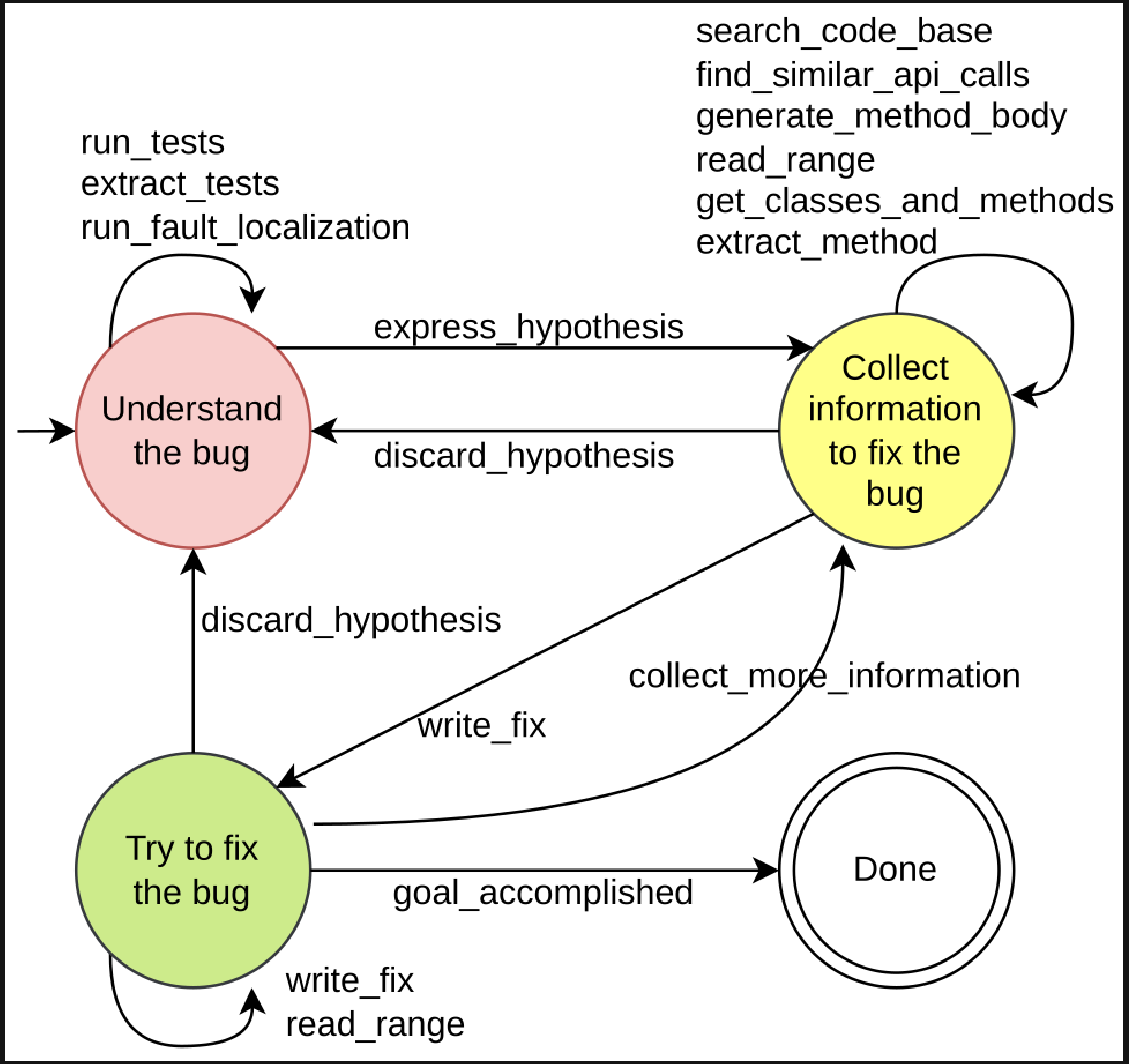
- Static parts:
  - Role, goals, guidelines, output format
- Dynamic parts:
  - Available tools
  - Gathered information
  - Last executed command and its result

Tools

Category	Tools
Read and extract code	Read range of lines
	Get classes and methods
Search and generate code	Search code base by keyword
	Predict method body via LLM
Testing and patching	Run test suite
	Apply a patch
Control	Express or discard hypothesis
	Declare success

See paper for complete list of 14 tools.

Guidance via Finite State Machine



Evaluation

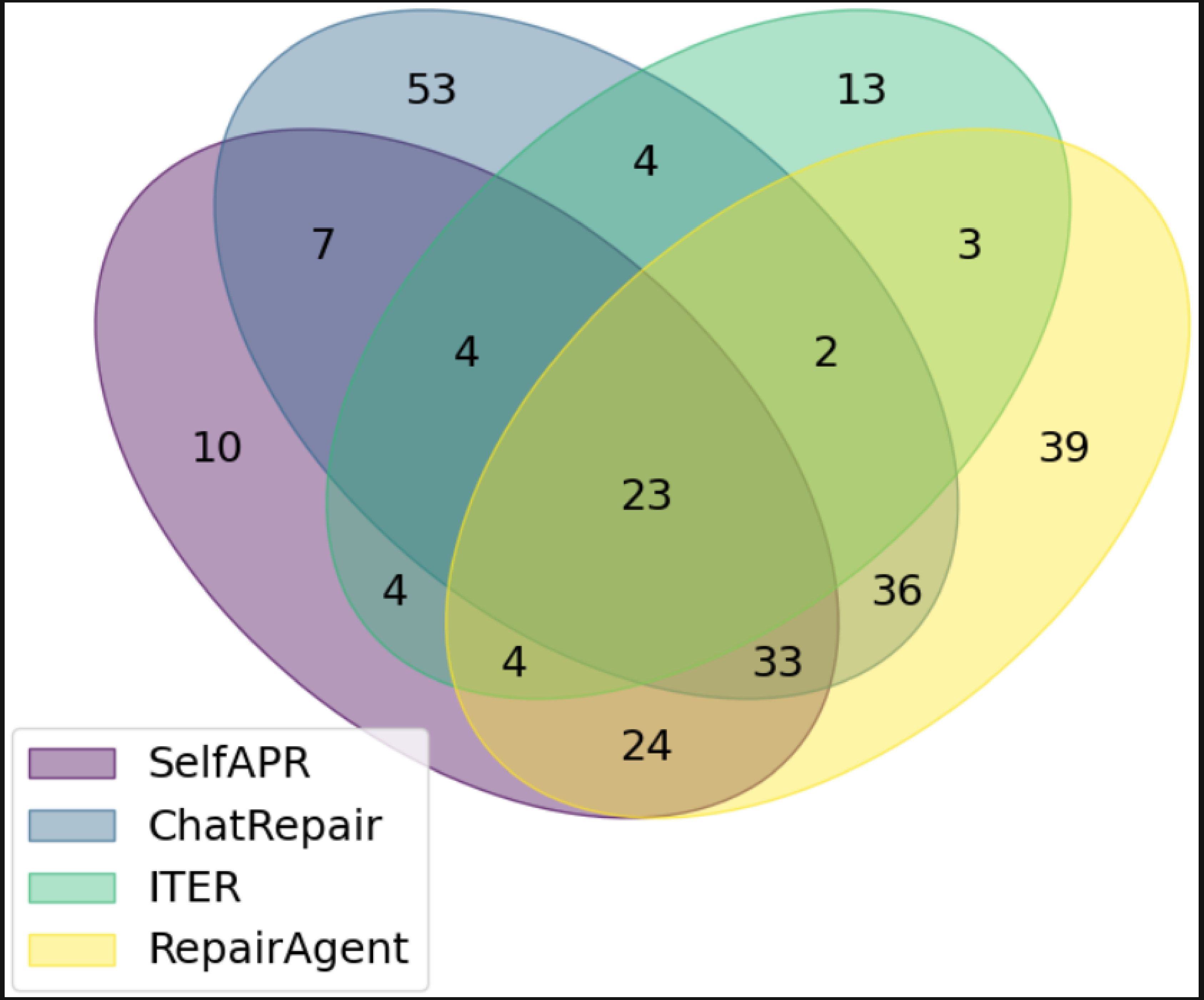
- OpenAI’s GPT-3.5-0125
- All 835 bugs from Defects4J v1.2 and v2
  - Including multi-line, multi-file bugs
- Measures of success
  - Plausible fixes
  - Correct fixes
  - Cost per bug

Effectiveness

Correct bug fixes:

Bug type	RepairAgent	ChatRepair	ITER	SelfAPR
Single-line	110	133	36	83
Multi-line	46	29	14	24
Multi-file	3	0	4	3
Total	164	162	57	110

Effectiveness



Examples

```
  if (cfa != null) {
    for (Node finallyNode : cfa.finallyMap.get(parent)) {
-     cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+     cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
    }
  }
```

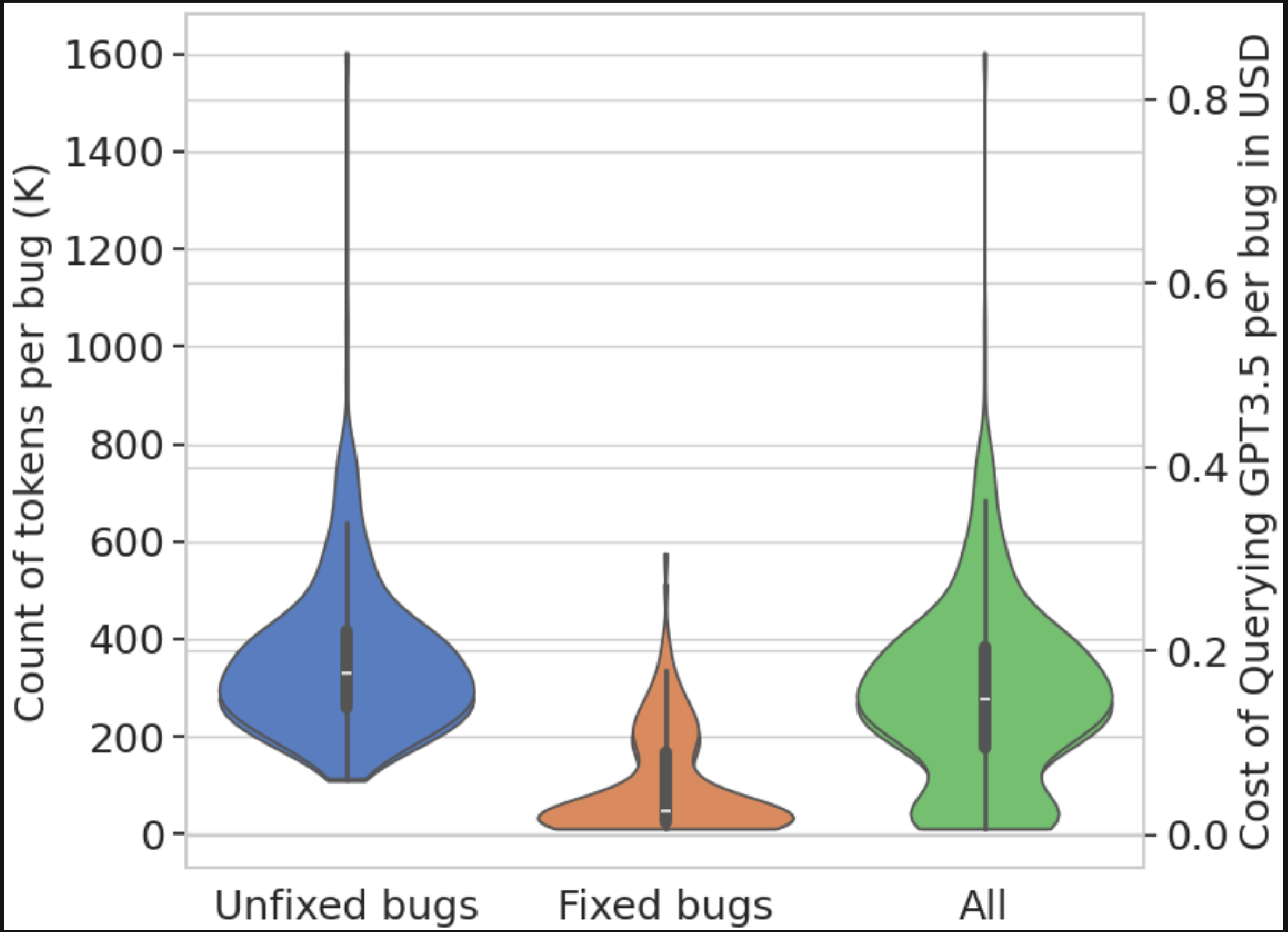
Found this field by searching the code base

Examples

```
  Separator sep = (Separator) elementPairs.get(0);
+ if (sep.iAfterParser == null &&
+     sep.iAfterPrinter == null) {
    PeriodFormatter f = toFormatter(...);
    sep = sep.finish(f.getPrinter(), f.getParser());
    return new PeriodFormatter(sep, sep);
+ }
```

Found condition via LLM-based code completion

Costs



Avg. per bug:

270k tokens,

USD 0.14

Summary: RepairAgent

Symbolic reasoning

- Test executions
- FSM-based guidance
- Static code search

↔

Neural reasoning

- LLM-driven decision making
- LLM-based code completion
- NL as “glue language”

- Paper: “RepairAgent: An Autonomous, LLM-Based Agent for Program Repair” (arXiv, 2024)

Conclusions and Open Challenges

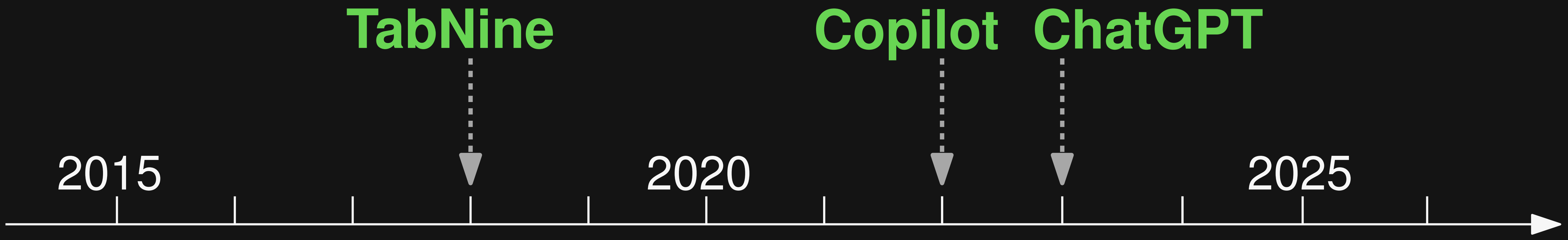
Neuro-symbolic developer tools are here to stay

- **LExecutor: Learning-guided execution**
  - Future work: Dynamic analysis applications
- **RepairAgent: Autonomous, LLM-based repair**
  - Future work: Autonomous agents for other SE tasks
- **General open challenge: Better interfaces between neural and symbolic reasoning**

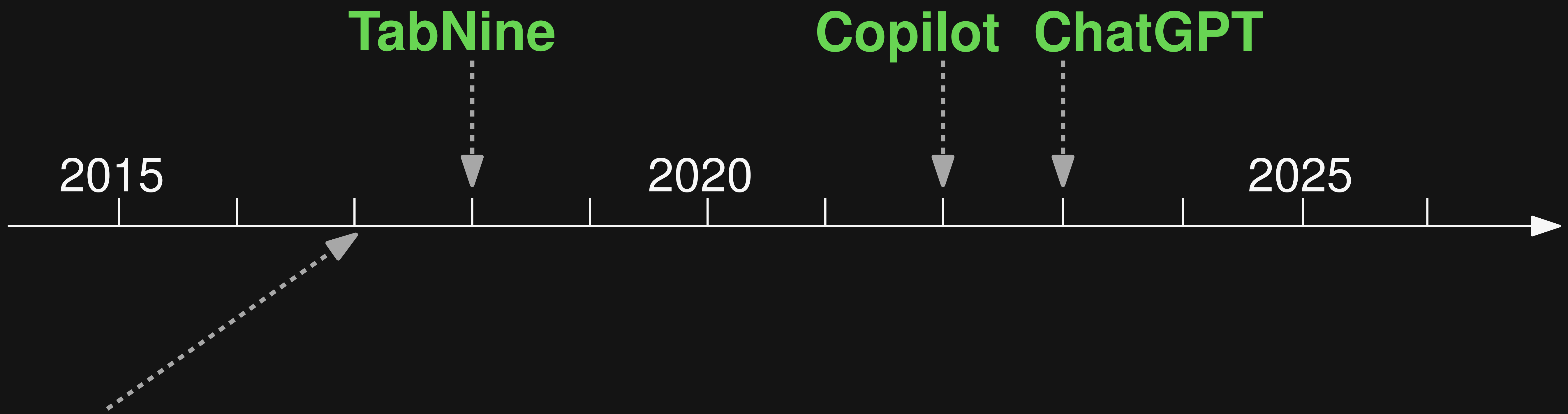
Template

- aa

A Bit of History



A Bit of History



TabNine

Copilot

ChatGPT

2015

2020

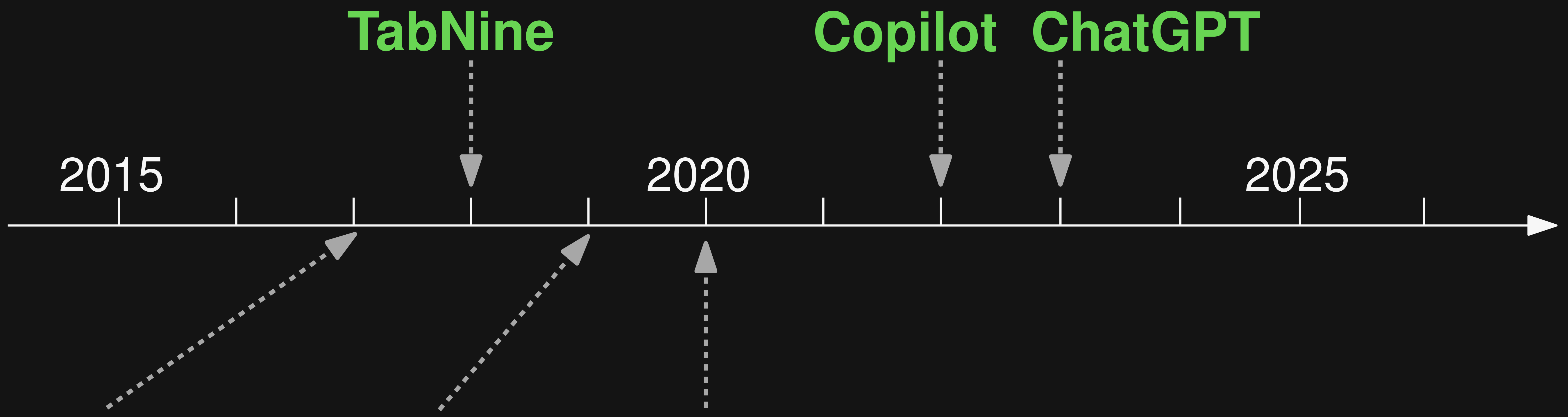
2025

DeepBugs

(OOPSLA’18)

Bug detection as a neural classification problem

A Bit of History



TabNine

Copilot ChatGPT

2015

2020

2025

DeepBugs

(OOPSLA'18)

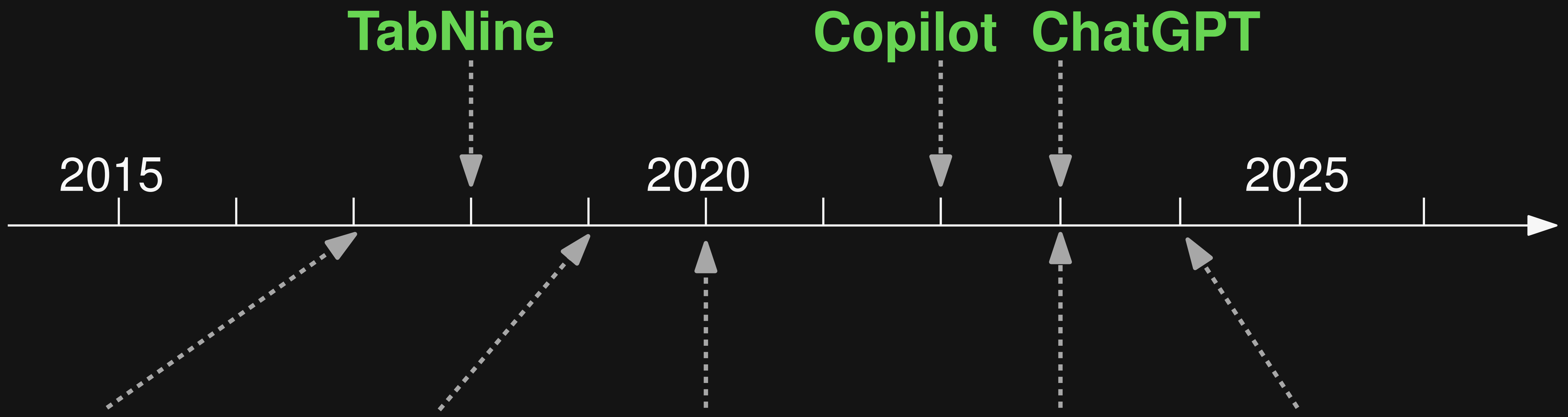
Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE'19) (FSE'20)

Neural type inference with static validation

A Bit of History



TabNine

Copilot ChatGPT

2015

2020

2025

DeepBugs

(OOPSLA’18)

Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE’19) (FSE’20)

Neural type inference with static validation

LExecutor

(FSE’23)

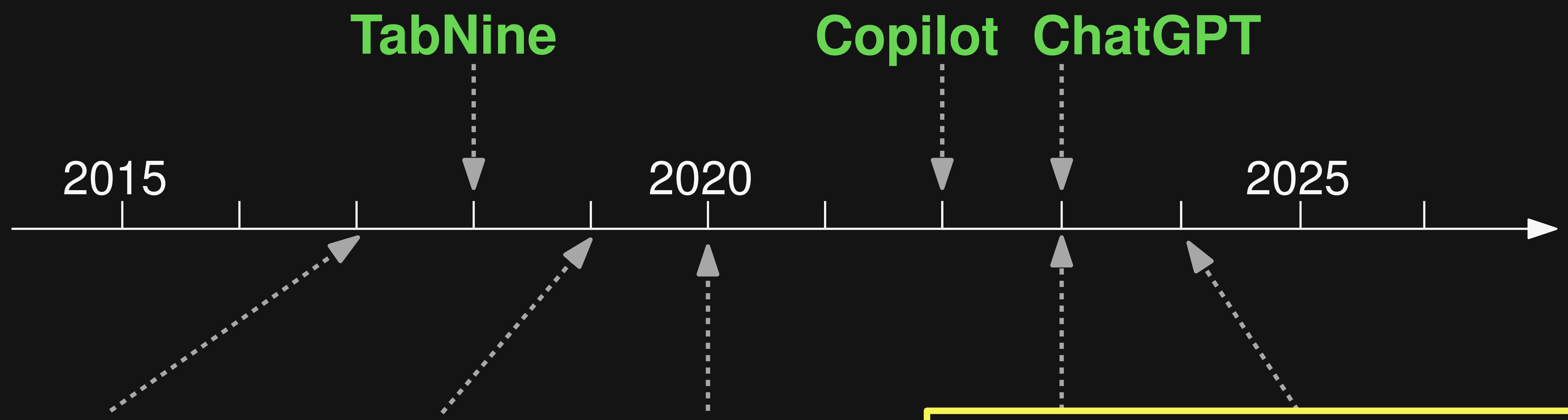
Learning-guided execution

RepairAgent

(arXiv’24)

Autonomous, LLM-based program repair

A Bit of History



DeepBugs

(OOPSLA'18)

Bug detection as a neural classification problem

NL2Type & **TypeWriter**

(ICSE'19) (FSE'20)

Neural type inference with static validation

LExecutor

(FSE'23)

Learning-guided execution

RepairAgent

(arXiv'24)

Autonomous, LLM-based program repair

Remainder of this talk