Intelligent Run-time Reliability Engineering for Python Software

Yun Peng

Supervisor: Prof. Michael R. Lyu

25 June, 2024

ARISE
Automated Reliable Intelligent Software Engineering

香港中文大學
The Chinese University of Hong Kong

Outline

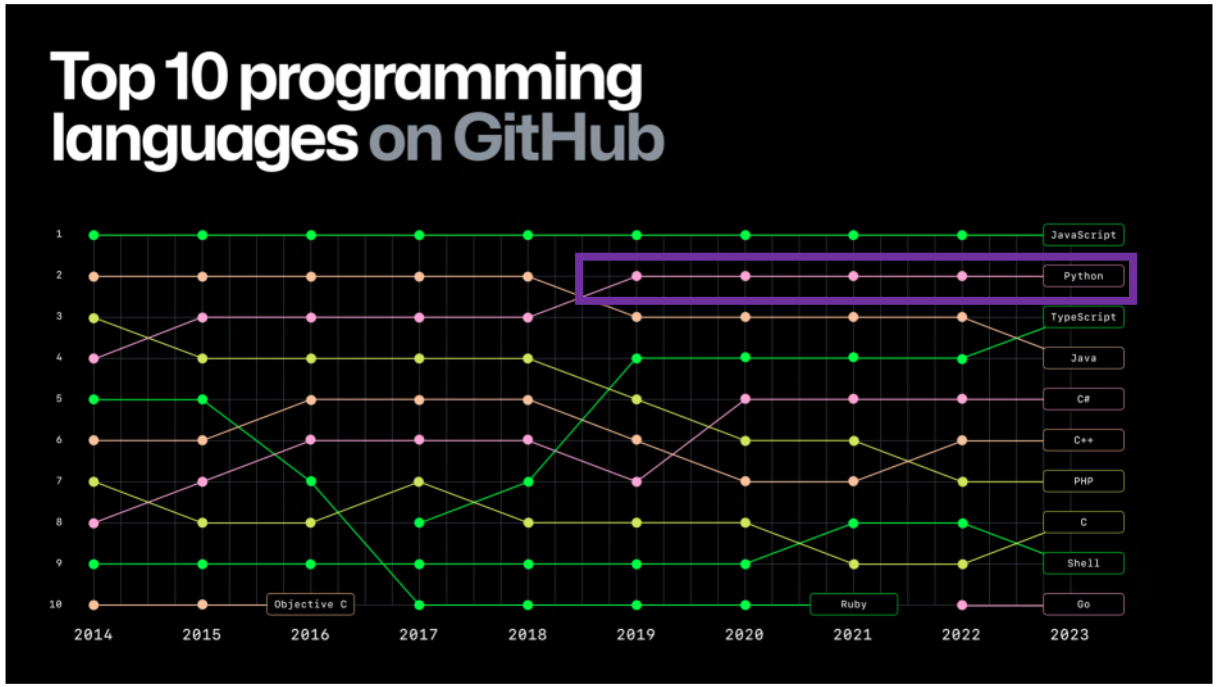
1. • Introduction & Background
2. • Dynamic Type System
3. • Dynamic Run-time Environment
4. • Conclusion & Future Work

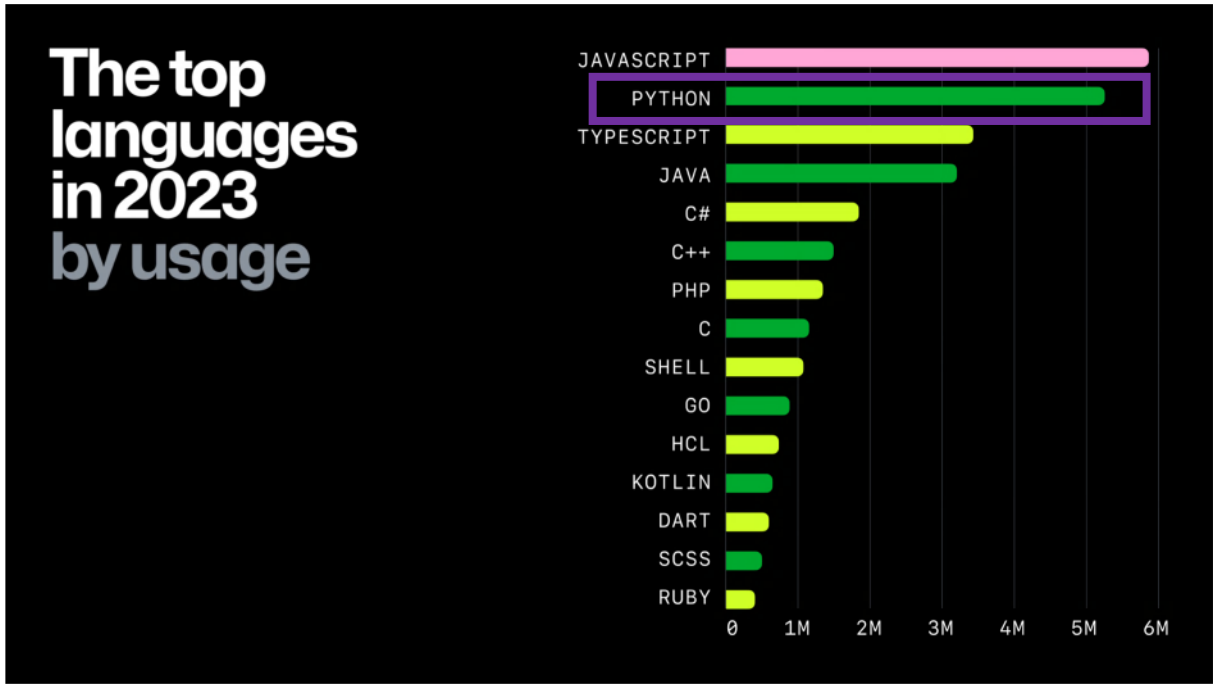
ONE

Introduction & Background

The Popularity of Python Language

- Python has been the 2nd most popular language at GitHub since 2019.
- Python has been used by more than 5 million developers at GitHub in 2023.





GitHub Octoverse: The state of open source and rise of AI in 2023.

Reference: GitHub Octoverse - https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

Dynamic Features → Fast Prototyping

- **Dynamic Type System**

As a dynamic language, Python does not require type declarations in code. The types of variables are determined at run time. This makes it easier to write generic functions.

- **Dynamic Run-time Environment**

Python does not require compilation before execution, making it portable on different platforms and systems.

Dynamic Type System

- Dynamic type system makes it easier for Python developers to **write generic functions**.

```
def add(a, b):
    return a + b
```

Python Code

```
>>> add(1, 2)
>>> 3

>>> add("1", "2")
>>> "12"
```

```
int add(int a, int b){
        return a + b;
}

string add(string a, string b){
        a.append(b);
        return a;
}
```

C++ Code

Reliability Issue #1 – Type Error

- Python allows flexible operations between types.

```
>>> “100” * 2
>>> “100100”


>>> “100” / 2
>>> TypeError
```

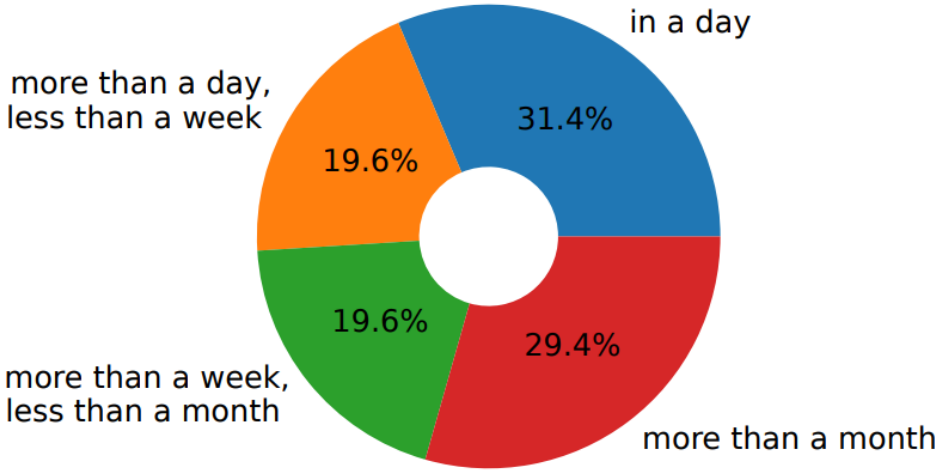
- Python allows heterogeneous data types.

```
def divide(input_list):
    result = input_list[0]
    for item in input_list[1:]:
        result = result / item
    return result

>>> divide([100, 100])
>>> 1

>>> divide([100, “100”])
>>> TypeError
```

Reliability Issue #1 – Type Error

	Type	Attribute	Value	Key	Import
StackOverflow	**31.5%**	19.4%	27.8%	8.3%	13.0%
GitHub	**29.2%**	19.4%	28.2%	12.9%	10.3%

Type errors are the most mentioned Python program errors.

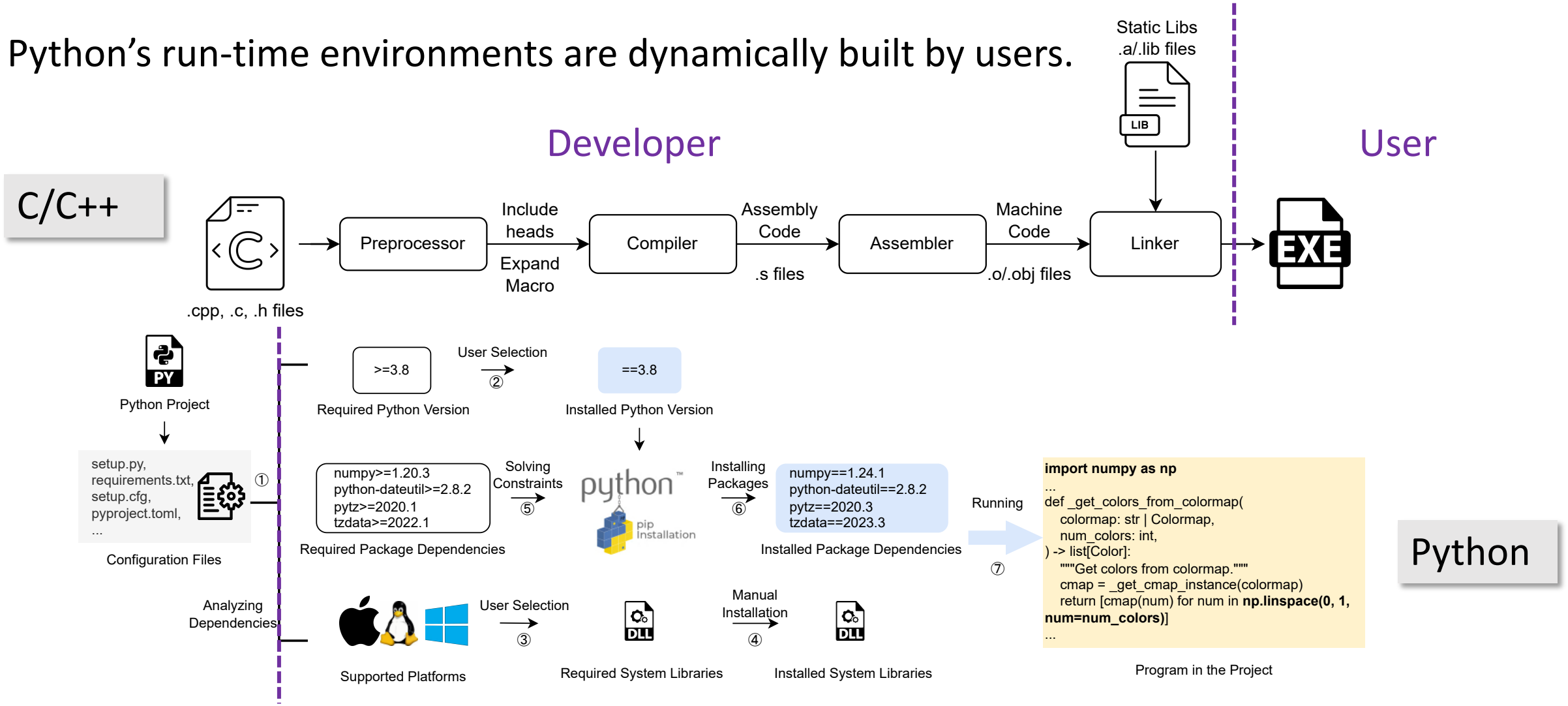


About 50% of type errors cost **more than one week** to be fixed.

Reference: *PyTER: effective program repair for Python type errors*

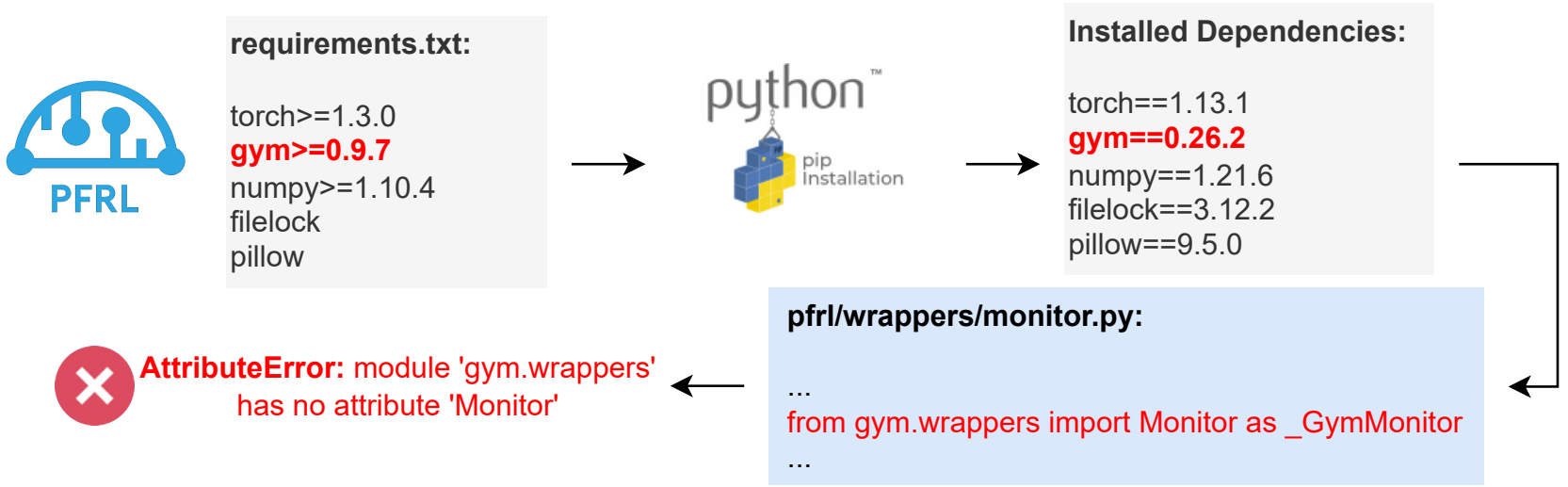
Dynamic Run-time Environment

- Python's run-time environments are dynamically built by users.



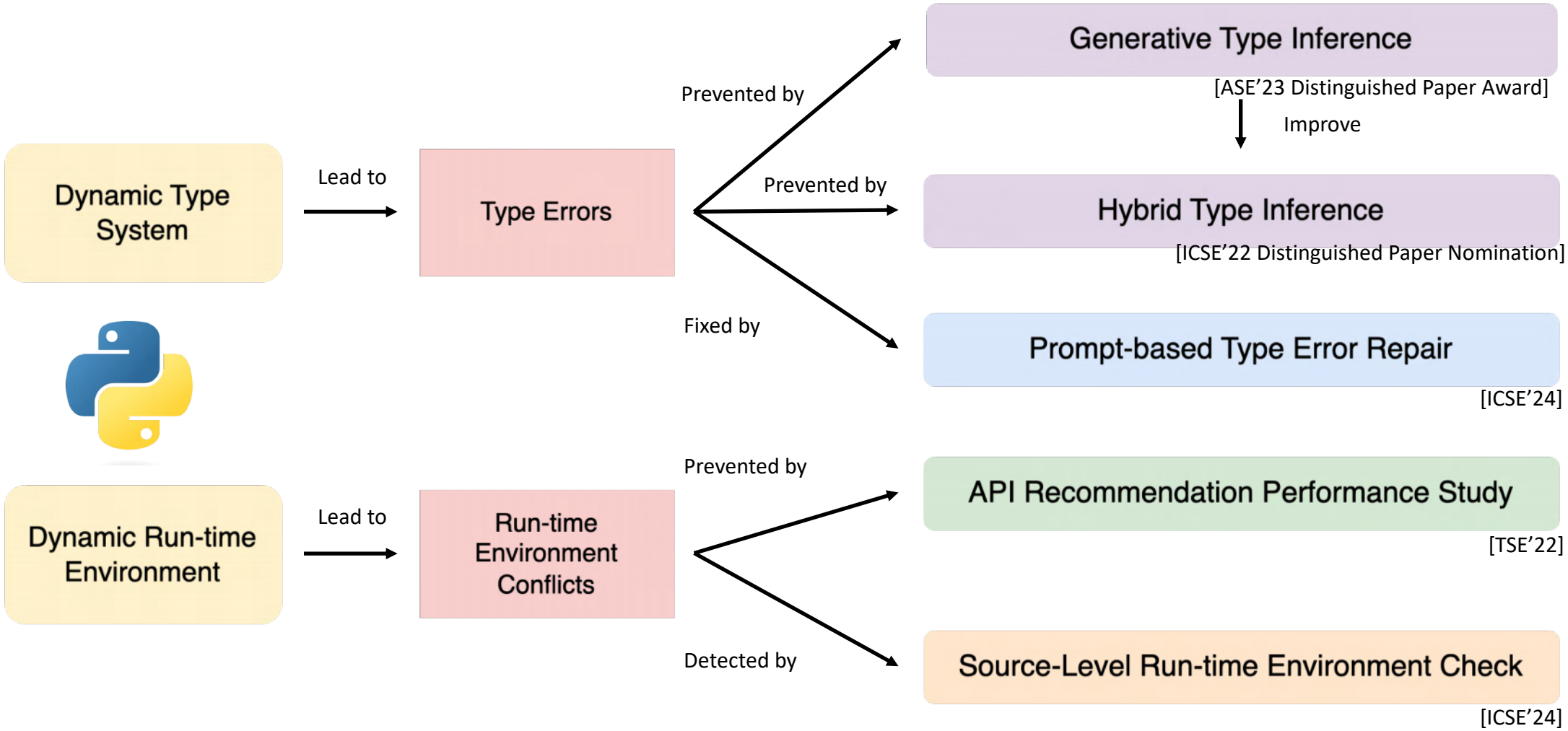
Reliability Issue #2 – Run-time Environment Conflicts

- Run-time environment conflicts occur when the implementations of external APIs used in the Python software **cannot be found or are not the required ones** in the run-time environment.



- Configuration files written by developers are not validated before distribution.

Outline of Thesis



Only first-author publications are listed.

TWO

Dynamic Type System

Improve the Reliability of Dynamic Type System

- Pathway #1 (Prevention): Use type inference to statically get the types of variables so that common static checking techniques can be used to detect potential issues.

Type Checking

Test Case Generation

Type Inference

- Type Inference aims to determine the types of each variable in code.

```
1 def add(num1, num2):
2     a = num1 + num2
3     b = 1 + 2
4     return a + b
```

Parameters:

num1 : ?

num2 : ?

Local Variables:

a : ?

b : ?

Return Value:

add : ?

How to Do Type Inference?

- Static type inference, which is frequently used in compilers.

```
1 def add(num1, num2):
2     a = num1 + num2
3     b = 1 + 2
4     return a + b
```

$$\frac{}{\pi \vdash 1 : \text{int}} \qquad \frac{}{\pi \vdash 2 : \text{int}} \;(\text{Constant})$$

$$\frac{\pi \vdash 1 : \text{int} \quad \pi \vdash 2 : \text{int}}{\pi \vdash 1 + 2 : \text{int}} \;(\text{Add})$$

$$\frac{\pi \vdash 1 + 2 : \text{int}}{\pi \vdash b : \text{int}} \;(\text{Assign})$$

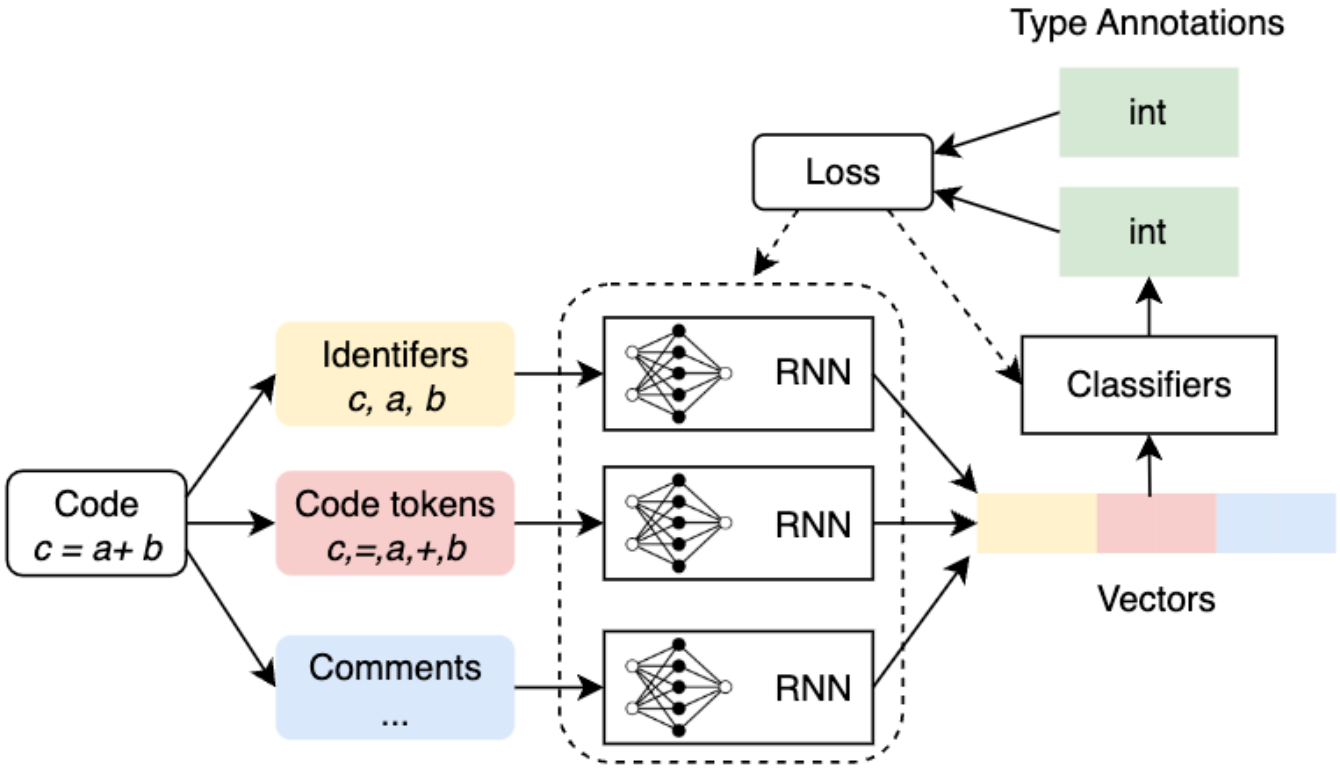
$$\frac{\text{Premise 1, …, Premise N}}{\text{conclusion}}$$

Typing Rule Format

- Very accurate (sound).
- Suffer from the low coverage problem.

How to Do Type Inference?

- Supervised Type Inference.



- Address the low coverage problem.
- Require high-quality type annotations to train, may not be accurate.

How to Do Type Inference?

- Static type inference vs. Supervised type inference.

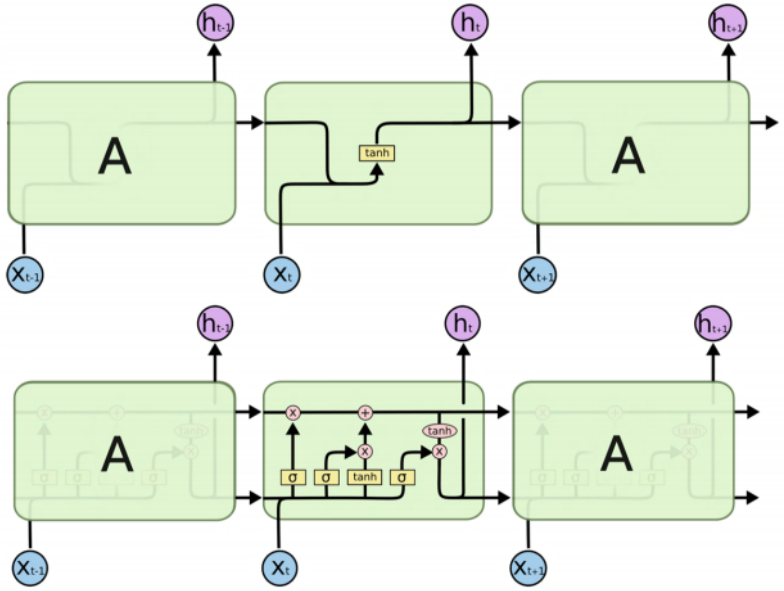
Static

Pyre

A performant type-checker for Python 3

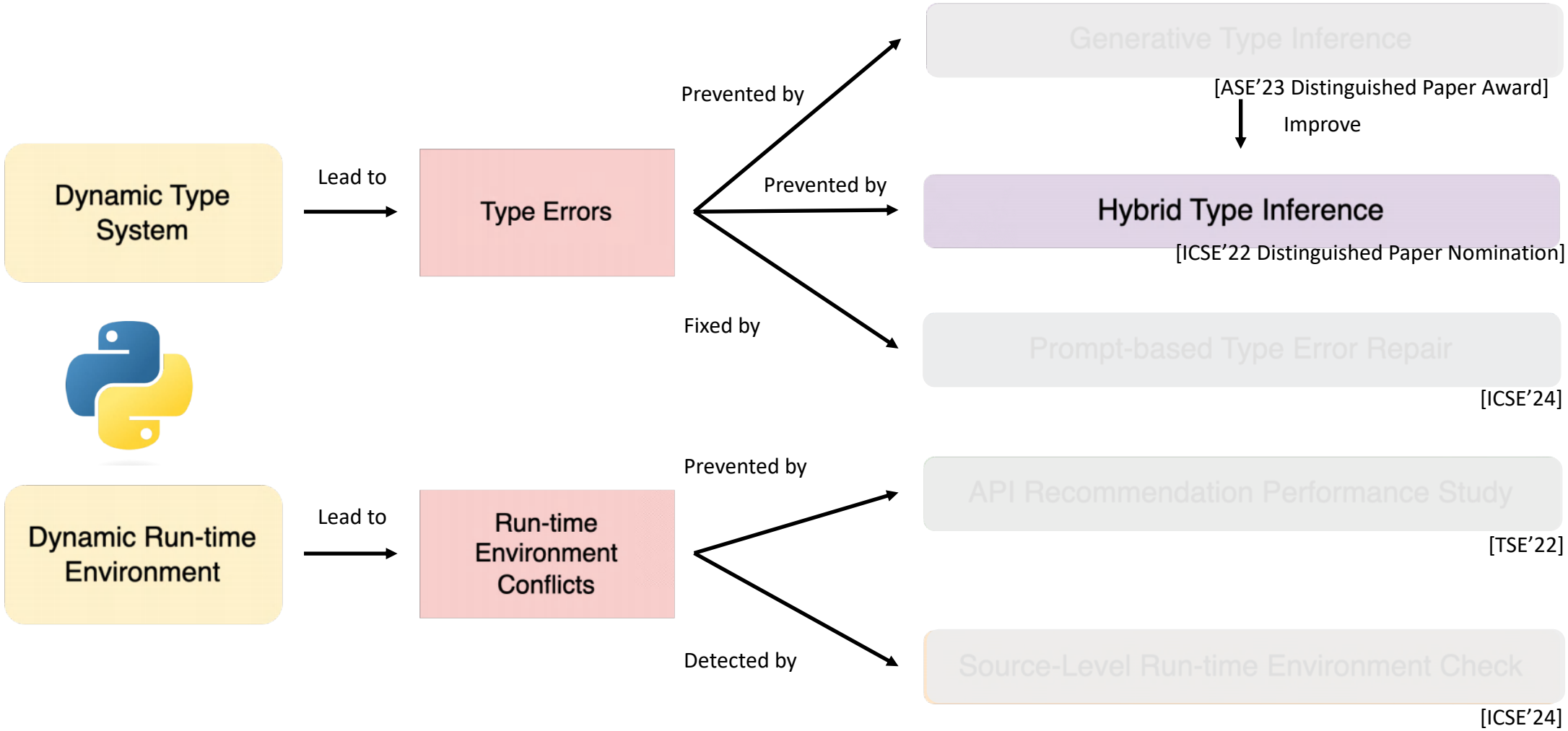
- Very accurate (sound).
- Suffer from the low coverage problem.

Supervised



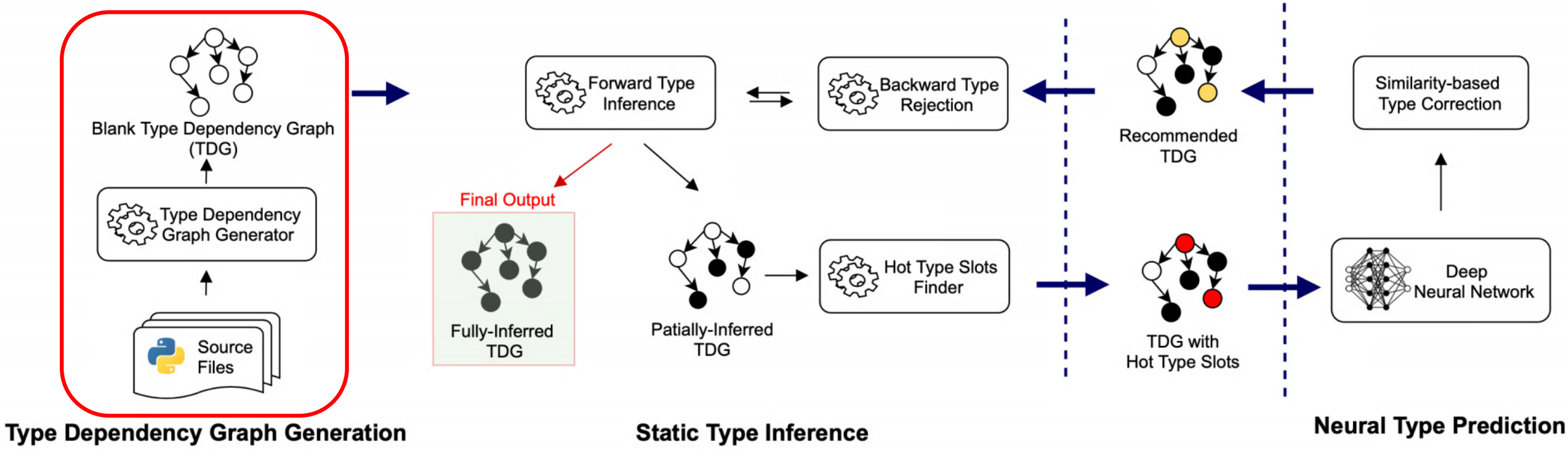
- Address the low coverage problem.
- Require high-quality type annotations to train, may not be accurate.

Outline of Thesis



Only first-author publications are listed.

Hybrid Type Inference - HiTyper



Type Dependency Graph (TDG)

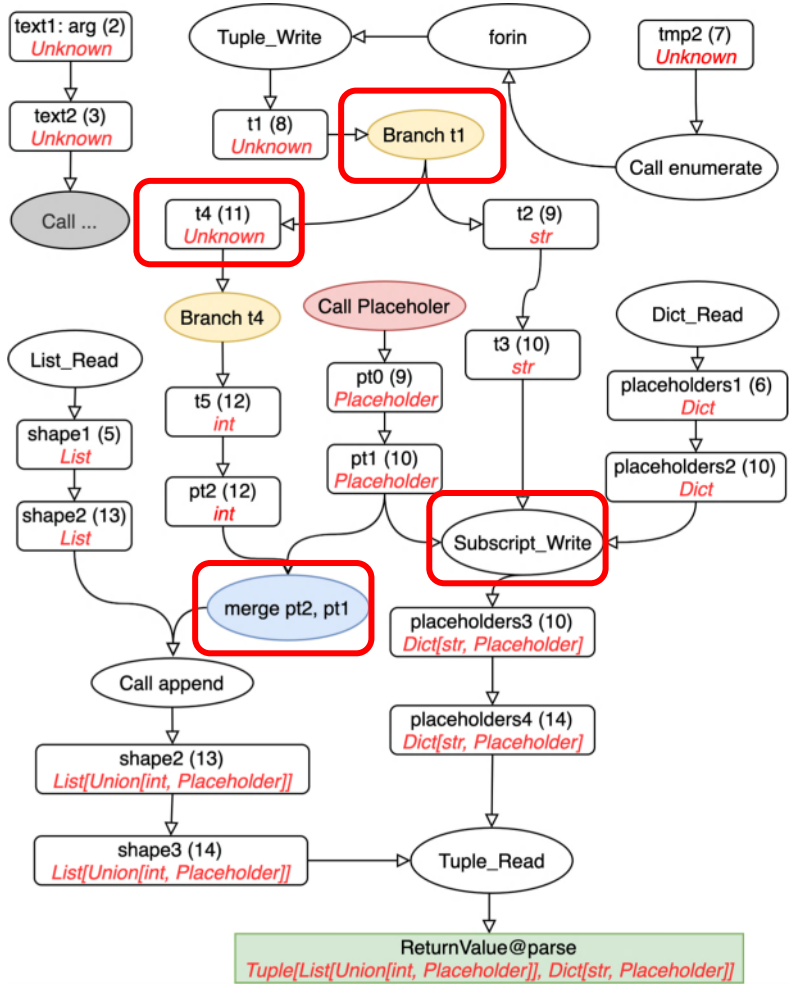
- To bridge static type inference and supervised type inference.

Graph G = (N, E)

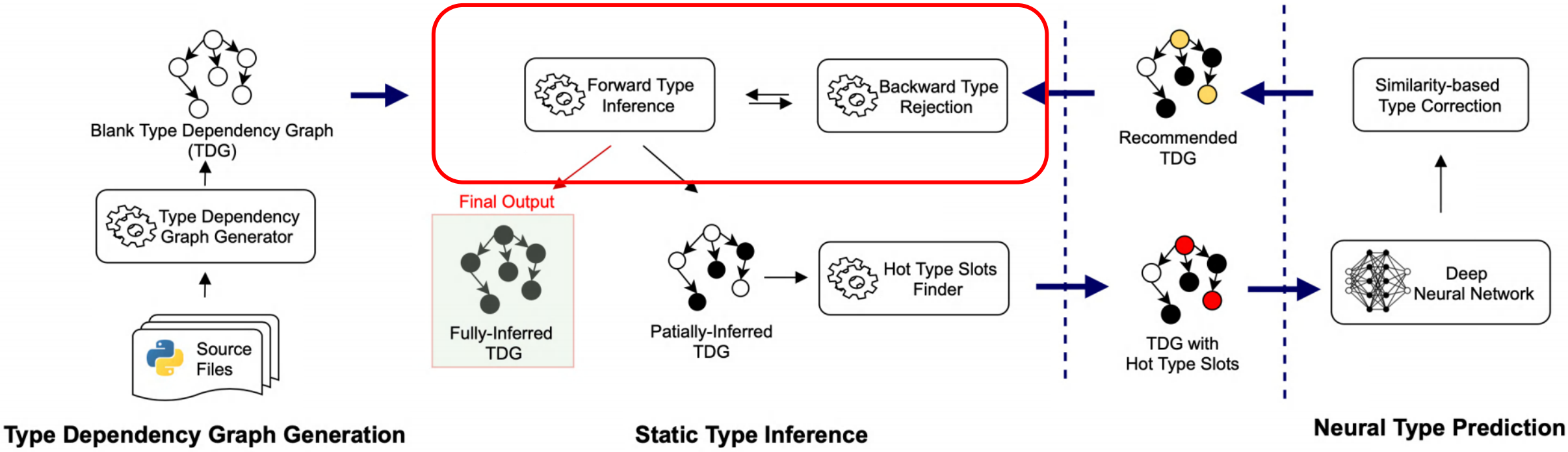
N: nodes E: edges

Four kinds of nodes:

- symbol node / type slot
- **expression node**
- branch node
- merge node



Hybrid Type Inference - HiTyper



Static Inference and Rejection

- To Infer correct types as many as possible and reject incorrect type predictions from deep learning models.

Expressions

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & v \mid c \mid e\ \textbf{blop}\ e \mid e\ \textbf{numop}\ e \mid \\
& e\ \textbf{cmpop}\ e \mid e\ \textbf{bitop}\ e \mid \\
& (e, ..., e) \mid [e, ..., e] \mid \\
& \{e : e, ..., e : e\} \mid \{e, ..., e\} \mid \\
& [e\ \textbf{for}\ e\ \textbf{in}\ e] \mid \{e\ \textbf{for}\ e\ \textbf{in}\ e\} \mid \\
& \{e : e\ \textbf{for}\ e, e\ \textbf{in}\ e\} \mid (e\ \textbf{for}\ e\ \textbf{in}\ e) \mid \\
& e(e, ..., e) \mid e[e : e : e] \mid e.v \\
v \in \mathit{Variables} ::= {} & \textit{all identifiers in code} \\
c \in \mathit{Constants} ::= {} & \textit{all literals in code} \\
\mathit{blop} \in \mathit{Boolean\ Operations} ::= {} & \mathit{And} \mid \mathit{Or} \mid \mathit{Not} \\
\mathit{numop} \in \mathit{Numeric\ Operations} ::= {} & \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mult} \mid \mathit{Div} \mid \mathit{Mod} \mid \\
& \mathit{UAdd} \mid \mathit{USub} \\
\mathit{bitop} \in \mathit{Bitwise\ Operations} ::= {} & \mathit{LShift} \mid \mathit{RShift} \mid \mathit{BitOr} \mid \mathit{BitAnd} \mid \\
& \mathit{BitXor} \mid \mathit{FloorDiv} \mid \mathit{Invert} \\
\mathit{cmpop} \in \mathit{Compare\ Operations} ::= {} & \mathit{Eq} \mid \mathit{NotEq} \mid \mathit{Lt} \mid \mathit{LtE} \mid \mathit{Gt} \mid \mathit{GtE} \mid \\
& \mathit{Is} \mid \mathit{IsNot} \mid \mathit{In} \mid \mathit{NotIn}
\end{aligned}
$$

Basic Format:

$$\pi \vdash e : \theta.$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\textbf{bool}, \textbf{int}, O\}}{\pi \vdash e_1\ \textbf{bitop}\ e_2 : \theta \wedge \widetilde{\theta} \qquad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}}\ (\text{LShift, RShift})$$

Typing Rules

Infer result types for expression.

Type Rejection Rules

Reject operand types for expression.

Static Inference and Rejection

- To Infer correct types as many as possible and reject incorrect type predictions from deep learning models.

Forward Type Inference:

- Start from nodes with no *input* nodes.
- Forward traverse the whole TDG.
- Activate *typing rules* in expression nodes.

Backward Type Rejection:

- Start from nodes with no *output* nodes.
- Backward traverse the whole TDG.
- Activate *type rejection rules* in expression nodes.

Basic Format:

$$\pi \vdash e : \theta.$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\textbf{bool}, \textbf{int}, O\}}{\pi \vdash e_1 \textbf{ bitop } e_2 : \theta \wedge \widetilde{\theta} \qquad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}}$$ (LShift, RShift)

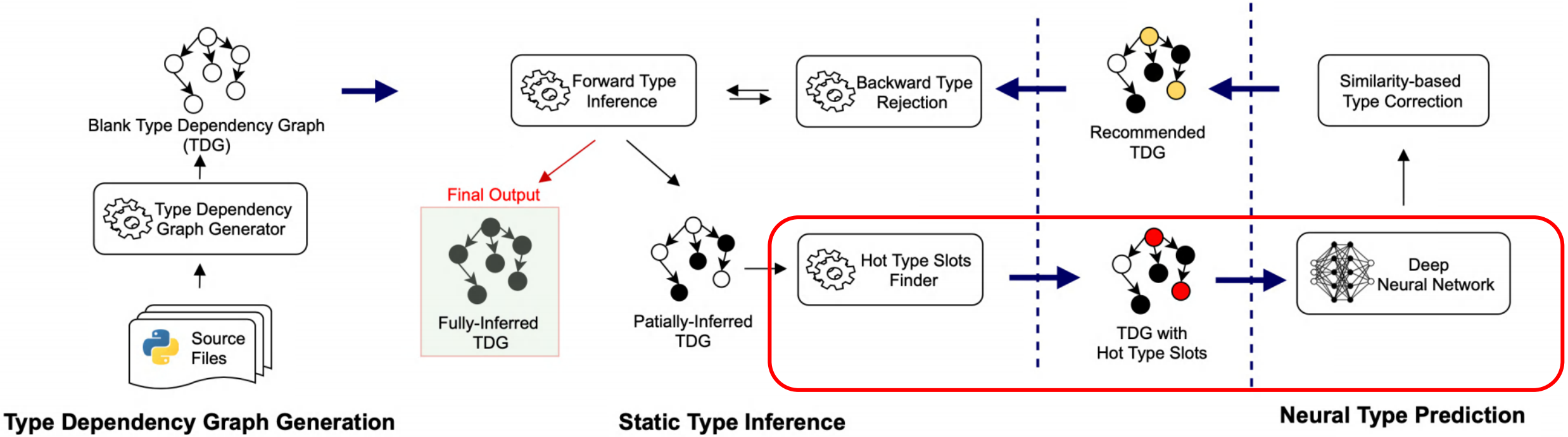
Typing Rules

Infer result types for expression.

Type Rejection Rules

Reject operand types for expression.

Hybrid Type Inference - HiTyper



Neural Type Prediction

Hot Type Slot Finder:

Identify the key variables that hinders the static inference part from inferring other variables.

→ reduce the variables that predicted by DL models.

Similarity-based Type Correction:

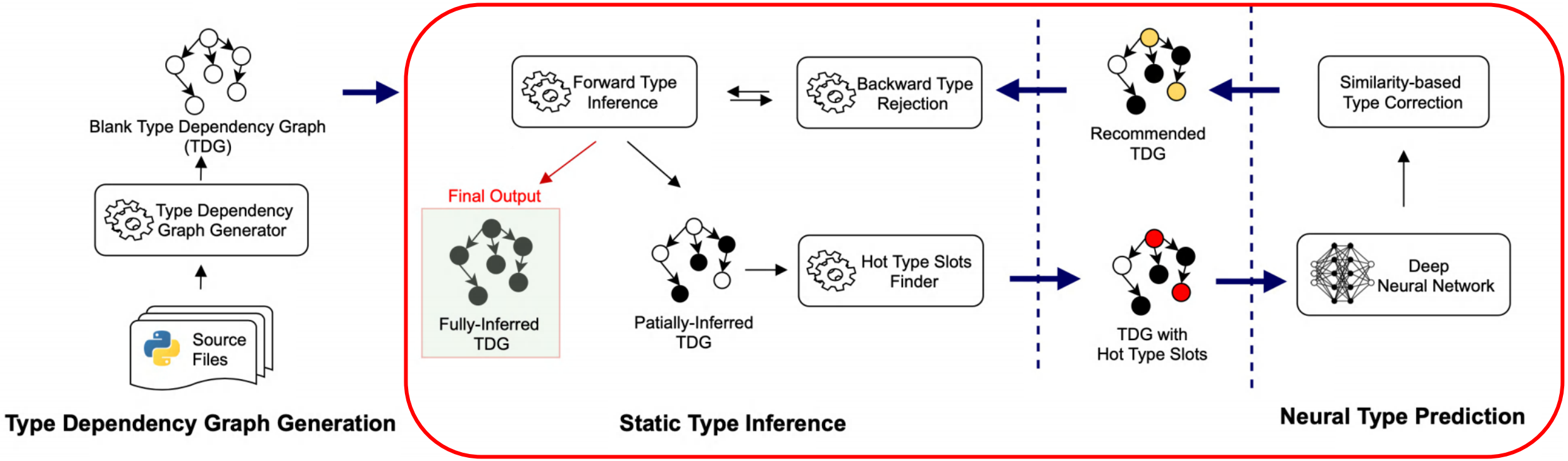
Map the never imported type predictions from DL models into valid types.

→ enhance the ability of predicting user-defined/third-party types.

import torch

prediction: tf.Tensor, *mapping to:* torch.Tensor

Hybrid Type Inference - HiTyper

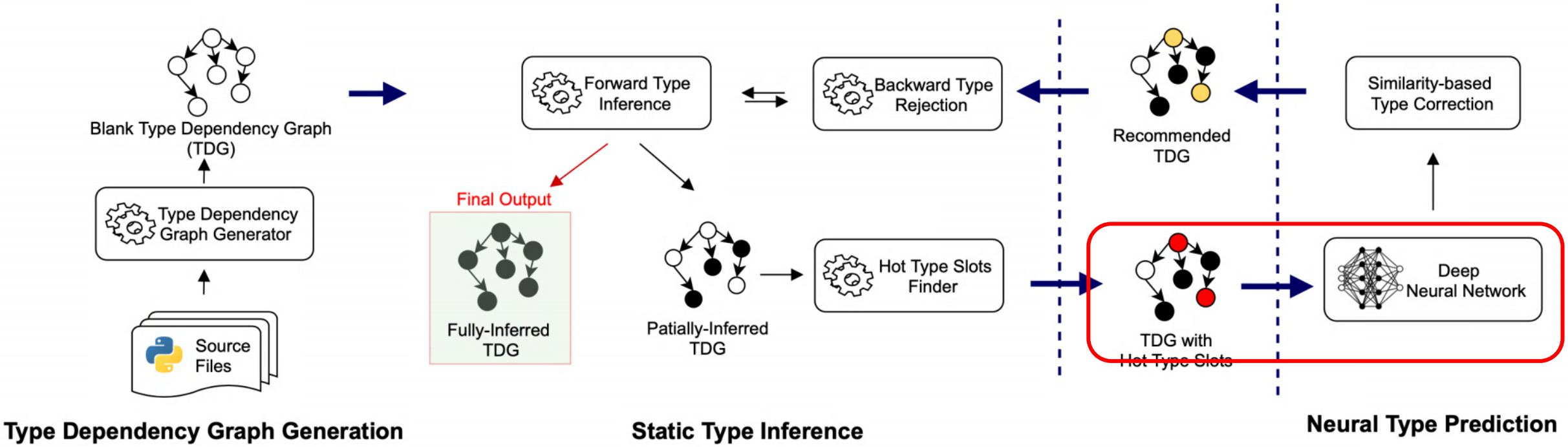


Evaluation

Dataset	Type Category	Approach	Top-1		Top-3		Top-5	
			Exact Match	Match to Parametric	Exact Match	Match to Parametric	Exact Match	Match to Parametric
ManyTypes4Py	Argument	Naive Baseline	0.14	0.16	0.33	0.38	0.43	0.51
		Type4Py	0.61	0.62	0.64	0.66	0.65	0.68
		HiTyper	**0.65**	**0.67**	**0.70**	**0.74**	**0.72**	**0.76**
	Return Value	Naive Baseline	0.07	0.10	0.19	0.28	0.28	0.42
		Type4Py	0.49	0.52	0.53	0.59	0.54	0.63
		HiTyper	**0.60**	**0.72**	**0.63**	**0.76**	**0.65**	**0.77**
	Local Variable	Naive Baseline	0.13	0.17	0.33	0.45	0.47	0.65
		Type4Py	0.67	0.73	0.71	0.78	0.72	0.79
		HiTyper	**0.73**	**0.85**	**0.74**	**0.86**	**0.75**	**0.86**
	All	Naive Baseline	0.13	0.16	0.31	0.40	0.43	0.57
		Type4Py	0.62	0.66	0.66	0.72	0.67	0.73
		HiTyper	**0.69**	**0.77**	**0.72**	**0.81**	**0.72**	**0.82**
Typilus's Dataset	Argument	Naive Baseline	0.19	0.20	0.38	0.42	0.46	0.50
		Typilus	0.60	0.65	0.69	0.74	0.71	0.76
		HiTyper	**0.63**	**0.68**	**0.72**	**0.76**	**0.76**	**0.79**
	Return Value	Naive Baseline	0.11	0.11	0.28	0.31	0.36	0.43
		Typilus	0.41	0.57	0.48	0.62	0.50	0.64
		HiTyper	**0.57**	**0.70**	**0.63**	**0.75**	**0.64**	**0.77**
	All	Naive Baseline	0.17	0.18	0.35	0.39	0.44	0.48
		Typilus	0.54	0.62	0.63	0.70	0.65	0.72
		HiTyper	**0.61**	**0.69**	**0.69**	**0.76**	**0.72**	**0.78**

HiTyper shows great improvement (11% ~ 15%) on **overall type inference** performance,
and the most significant improvement is on **return value inference** (22% ~ 39%).

Limitations of HiTyper



- The performance upper bound of HiTyper depends on the performance of deep learning models used in the framework.
- If deep learning model cannot give correct type predictions, static inference cannot validate and give the final predictions.

A Recent New Approach – Cloze-Style Type Inference

```
1 def add(num1:<mask0>, num2:<mask1>) -> <mask2>:
2     c:<mask3> = num1 + num2
3     d:<mask4> = 1 + 2
4     return c + d
```

Parameters:

<mask0> (num1) : int

<mask1> (num2) : int

Local Variables:

<mask3> (c) : int

<mask4> (d) : int

Return Value:

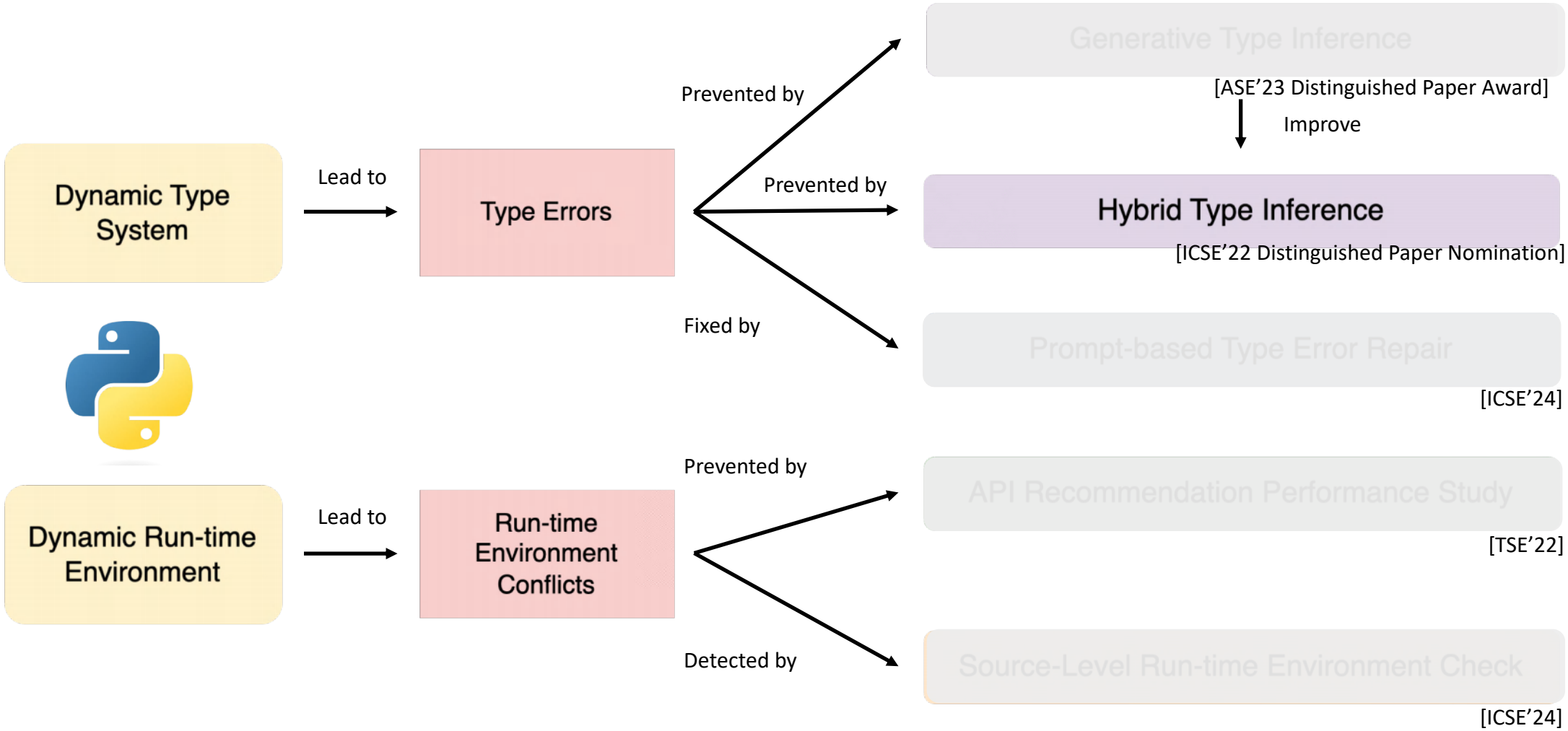
<mask2> (add) : int

A Recent New Approach – Cloze-Style Type Inference

```
1 def add(num1:<mask0>, num2:<mask1>) -> <mask2>:
2     c:<mask3> = num1 + num2
3     d:<mask4> = 1 + 2
4     return c + d
```

- Do not require a high quality training set.
- Lack of static domain knowledge:
  With internal knowledge only in the pre-trained code models.
- Lack of interpretability:
  No idea about how the model reaches the prediction.

Outline of Thesis



Only first-author publications are listed.

Generative Type Inference - TypeGen

Input prompt with static domain knowledge

Python Code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
DATABASES['default'].update(db_from_env)
```

Available User-defined Types:

os.Mapping, os.MutableMapping, os.PathLike, os._AddedDllDirectory, os._Environ, os._wrap_close

Q: What"s the type of the variable DATABASES?

LLM

Output chain-of-thought prompt making predictions

A: First, the variable DATABASES is assigned from a dict. **Second**, the key of the dict is a str. The value of the dict is a dict. **Third**, the keys of the dict are a str and a str. The values of the dict are a str and a function call os.path.join. **Therefore**, the type of the variable DATABASES is `dict[str, dict[str, str]]`.

Let LLMs act like a static type inference tool!

See what static inference sees, think how static inference thinks.

Generative Type Inference - TypeGen

Challenge 1: Lack of static domain knowledge

What knowledge should a model have to infer a type for a variable?
(See what static inference sees)

Knowledge 1: The context of the target variable – Code slicing.

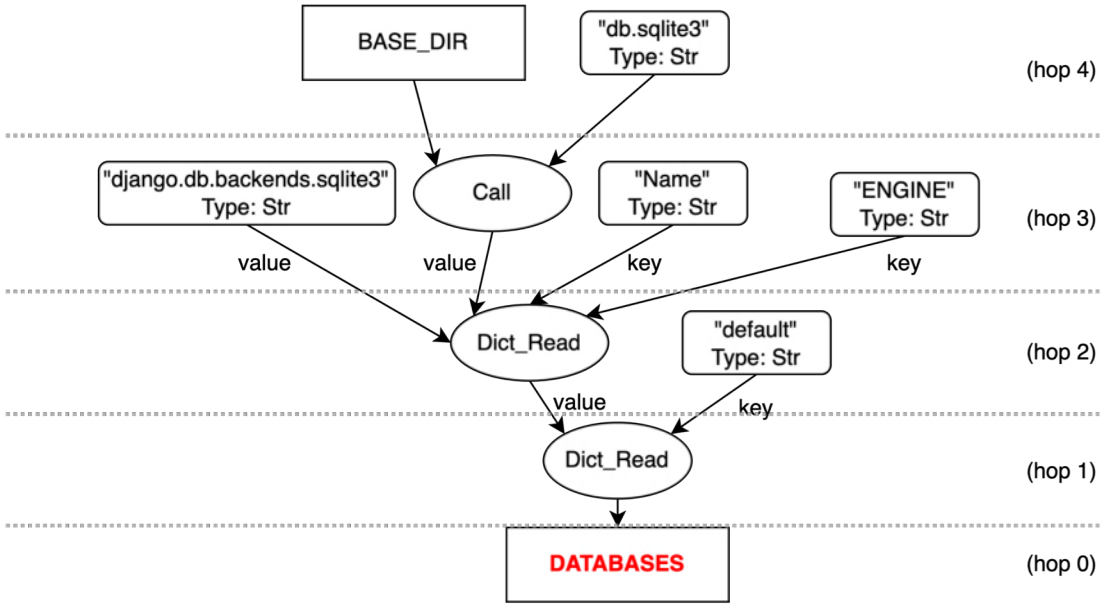
Knowledge 2: The valid type set of the variable – Type hint collection.

Code Slicing

- Remove all statements without data dependencies with the target variable.
- Remove statements with very far data dependencies with the target variable.

```
...
12 import os
...
25 DEBUG = bool( os.environ.get('DJANGO_DEBUG', True) )
27 ALLOWED_HOSTS = ['stepper-v2.herokuapp.com', '127.0.0.1']
...
71 DATABASES = {
72     'default': {
73         'ENGINE': 'django.db.backends.sqlite3',
74         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
75     }
76 }
...
129 db_from_env = dj_database_url.config(conn_max_age=500)
130 DATABASES['default'].update(db_from_env)
...
```

Source Code



Type Dependency Graph

Type Hint Collection

Imported types = third-party types + user-defined types

User-defined types:

- Collect all classes in the current source file.

Third-party types:

- Download top 10,000 popular Python packages in Libraries.io.
- Collect all classes and their paths as a third-party type database.
- Query the database based on the import statements in current source file.

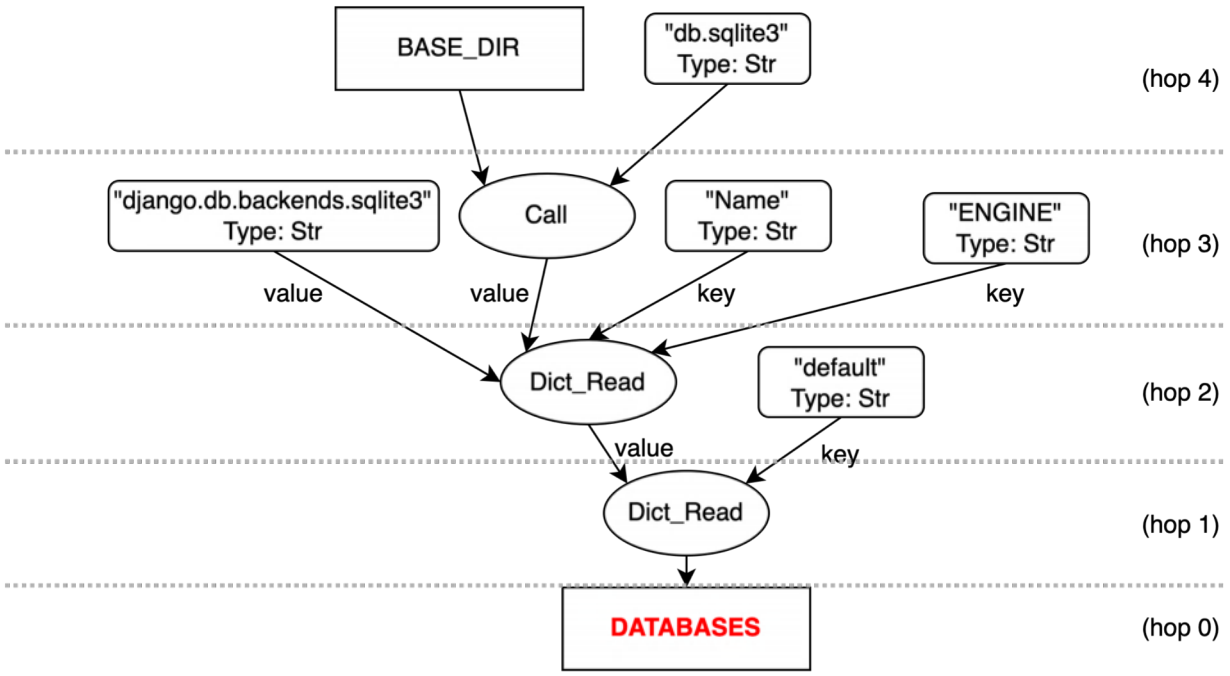
Generative Type Inference - TypeGen

Challenge 2: Lack of Interpretability

How to know/guide the model to reach a type prediction like static inference?
(Think how static inference thinks)

Simulate the inference steps of static inference!

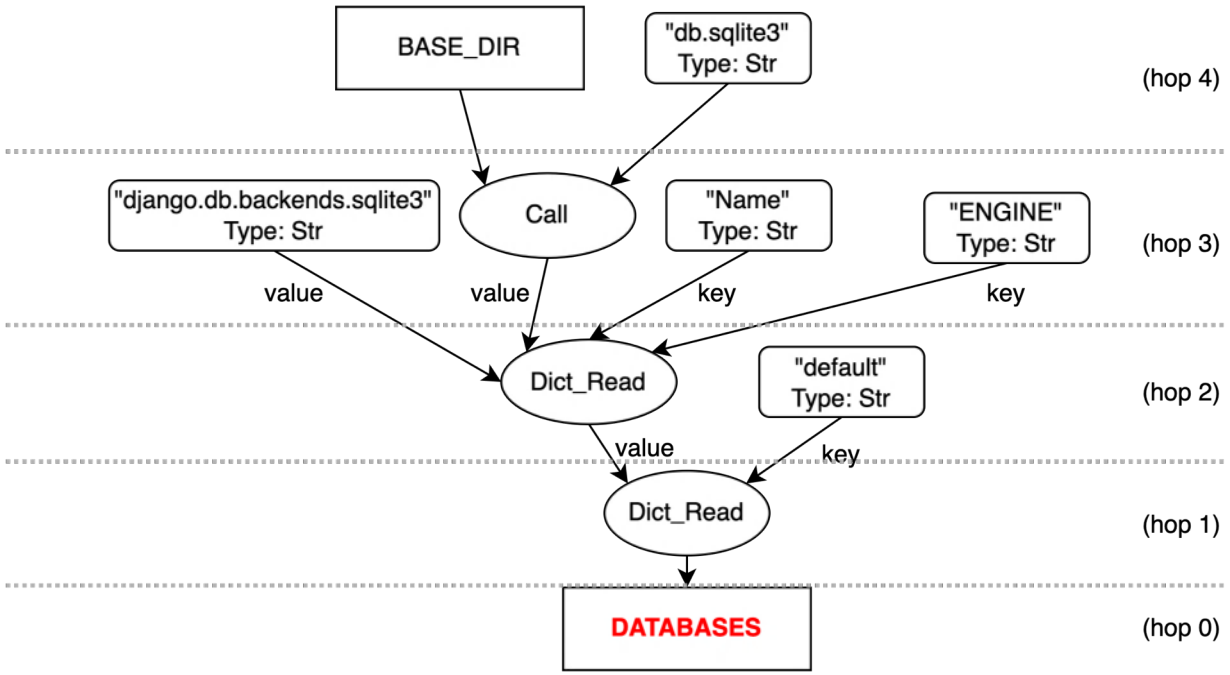
Chain-of-Thought Prompt Generation



Part	Type	Template
DD-RV	Operation→Symbol	The variable/return value of [NAME] is assigned from [OP] operation.
	Symbol→Symbol	The variable/return value of [NAME] is assigned from variable [NAME].
	Type→Symbol	The variable/return value of [NAME] is assigned from [TYPE].
	Operation→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [OP] operation.
	Symbol→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are variable [NAME].
	Type→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [TYPE].
DD-A	Usage	The argument [NAME] is used in [OP]/[NAME].
	Naming	Based on the naming convention, it is reasonable to assume that the type of the argument [NAME] is [GTTYPE].
Con	Conclusion	Therefore, the type of the variable/return value of/argument [NAME] is [GTTYPE].

Translate the Type Dependency Graph into a Chain-of-Thought prompt.

Chain-of-Thought Prompt Generation



- First, the variable DATABASES is assigned from a dict.
- Second, the key of the dict is a str. The value of the dict is a dict.
- Third, the keys of the dict are a str and a str. The values of the dict are a str and a function call os.path.join.
- Therefore, the type of the variable DATABASES is `dict[str, dict[str, str]]`.

Translate the Type Dependency Graph into a Chain-of-Thought prompt.

In-Context Learning

Static Analysis Generated

LLM Predicted

Example Prompt:

①. Code Slice

Python Code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
DATABASES['default'].update(db_from_env)
```

②. Type Hint

Available User-defined Types:

os.Mapping, os.MutableMapping, os.PathLike, os._AddedDllDirectory, os._Environ, os._wrap_close

Q: What"s the type of the variable DATABASES?

③. COT Prompt

A: First, the variable DATABASES is assigned from a dict. **Second**, the key of the dict is a str. The value of the dict is a dict. **Third**, the keys of the dict are a str and a str. The values of the dict are a str and a function call os.path.join. **Therefore**, the type of the variable DATABASES is `dict[str, dict[str, str]]`.

Target Variable Prompt: +

①. Code Slice

Python Code: [Code]

②. Type Hint

Available User-defined Types: [User-defined types from static analysis]

Q: What's the type of the variable [name]?

A: [To be generated]

Performance of TypeGen

Metric	Category	Approach	Top-1				Top-3				Top-5			
			Arg	Ret	Var	All	Arg	Ret	Var	All	Arg	Ret	Var	All
Exact Match (%)	Supervised	TypeBERT	28.0	38.5	51.1	45.4	34.8	52.6	55.8	51.4	36.5	57.1	58.6	54.1
		TypeWriter	53.3	52.8	-	-	61.1	60.7	-	-	65.8	65.3	-	-
		Type4Py	66.5	56.1	82.0	76.6	72.0	59.2	83.8	79.3	73.8	60.7	84.3	80.1
	Cloze Style	InCoder-1.3B	20.9	20.5	15.1	16.7	21.3	20.8	15.5	17.1	21.3	21.0	15.6	17.2
		InCoder-6.7B	24.1	42.0	18.7	21.9	24.6	42.7	19.1	22.3	24.7	43.1	19.2	22.4
		UniXcoder	55.0	49.2	35.9	40.9	66.9	64.6	42.1	49.0	70.6	69.8	45.2	52.4
		CodeT5-base	51.1	57.6	21.7	30.7	59.3	64.4	28.0	37.4	62.0	66.9	30.7	40.1
		CodeT5-large	56.2	60.2	44.7	48.4	61.6	64.5	50.4	53.9	63.9	66.3	53.4	56.6
	Generative	TYPEGEN	73.1	68.7	82.2	79.2	81.0	77.1	87.9	85.6	82.7	79.1	89.1	87.0
Match to Parametric (%)	Supervised	TypeBERT	29.8	41.4	54.0	48.1	36.0	55.9	58.0	53.5	37.7	60.8	61.2	56.5
		TypeWriter	54.4	54.1	-	-	63.4	63.5	-	-	68.8	69.3	-	-
		Type4Py	68.0	59.0	86.2	80.2	74.1	64.1	88.3	83.3	75.9	66.3	88.8	84.3
	Cloze Style	InCoder-1.3B	22.9	22.8	18.7	19.9	23.3	23.1	19.1	20.3	23.4	23.3	19.2	20.4
		InCoder-6.7B	28.8	51.6	25.0	28.1	29.3	52.1	25.3	28.5	29.4	52.5	25.3	28.6
		UniXcoder	61.9	61.8	44.3	49.3	72.3	76.0	51.2	57.6	75.0	80.1	53.8	60.4
		CodeT5-base	54.8	66.7	27.7	36.6	62.9	74.2	34.4	43.6	65.6	76.4	37.1	46.3
		CodeT5-large	61.4	69.4	55.7	58.0	66.8	74.3	61.2	63.5	68.9	76.2	63.7	65.9
	Generative	TYPEGEN	78.7	75.6	91.2	87.3	84.9	83.0	93.7	91.0	86.1	84.5	94.1	91.7

Performance of TypeGen

Base Model	Approach	Top-1 (△)	Top-3 (△)	Top-5 (△)
GPT-Neo (1.3B)	Zero-Shot	31.5	40.6	42.8
	Standard ICL	44.0 (40%)	50.0 (23%)	50.8 (19%)
	TypeGen	57.0 (81%)	61.5 (51%)	62.8 (47%)
GPT-Neo (2.7B)	Zero-Shot	43.2	50.0	51.9
	Standard ICL	46.6 (8%)	52.3 (5%)	52.8 (2%)
	TypeGen	55.5 (28%)	61.9 (24%)	63.0 (21%)
GPT-J (6.7B)	Zero-Shot	42.4	43.7	43.9
	Standard ICL	50.8 (20%)	54.9 (26%)	55.3 (26%)
	TypeGen	62.7 (48%)	67.3 (54%)	68.4 (56%)
CodeGen (6B)	Zero-Shot	34.7	44.0	45.5
	Standard ICL	54.1 (56%)	60.5 (38%)	61.9 (36%)
	TypeGen	63.7 (84%)	69.1 (57%)	70.8 (56%)
GPT-3.5 (175B)	Zero-Shot	62.0	65.4	66.3
	Standard ICL	69.7 (12%)	74.2 (13%)	75.8 (14%)
	TypeGen	78.9 (27%)	85.0 (30%)	86.2 (30%)
ChatGPT (175B)	Zero-Shot	61.3	66.1	67.5
	Standard ICL	68.0 (11%)	71.8 (9%)	73.1 (8%)
	TypeGen	78.8 (29%)	85.3 (29%)	86.7 (28%)

Ablation	Arg	Ret	Var	Ele	Gen	Usr	All
w/o Code Slice	74.8	77.0	68.8	75.1	75.5	73.9	70.8
w/o Type Hint	76.1	75.9	89.3	94.1	77.2	75.9	85.5
w/o COT Prompt	82.3	78.6	86.4	92.9	70.8	84.3	84.9
TypeGen	83.5	79.4	89.7	94.3	77.8	84.6	87.5

TypeGen is capable of **consistently improving** the zero-shot performance of type inference for language models **with different parameter sizes** and achieves **2x ~ 3x** of improvements made by the Standard ICL setting.

Improve the Reliability of Dynamic Type System

- Pathway #1 (Prevention): Use type inference to statically get the types of variables so that common static checking techniques can be used to detect potential issues.

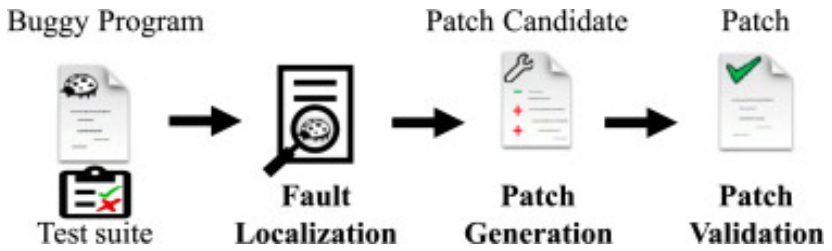
: my[py]

Type Checking

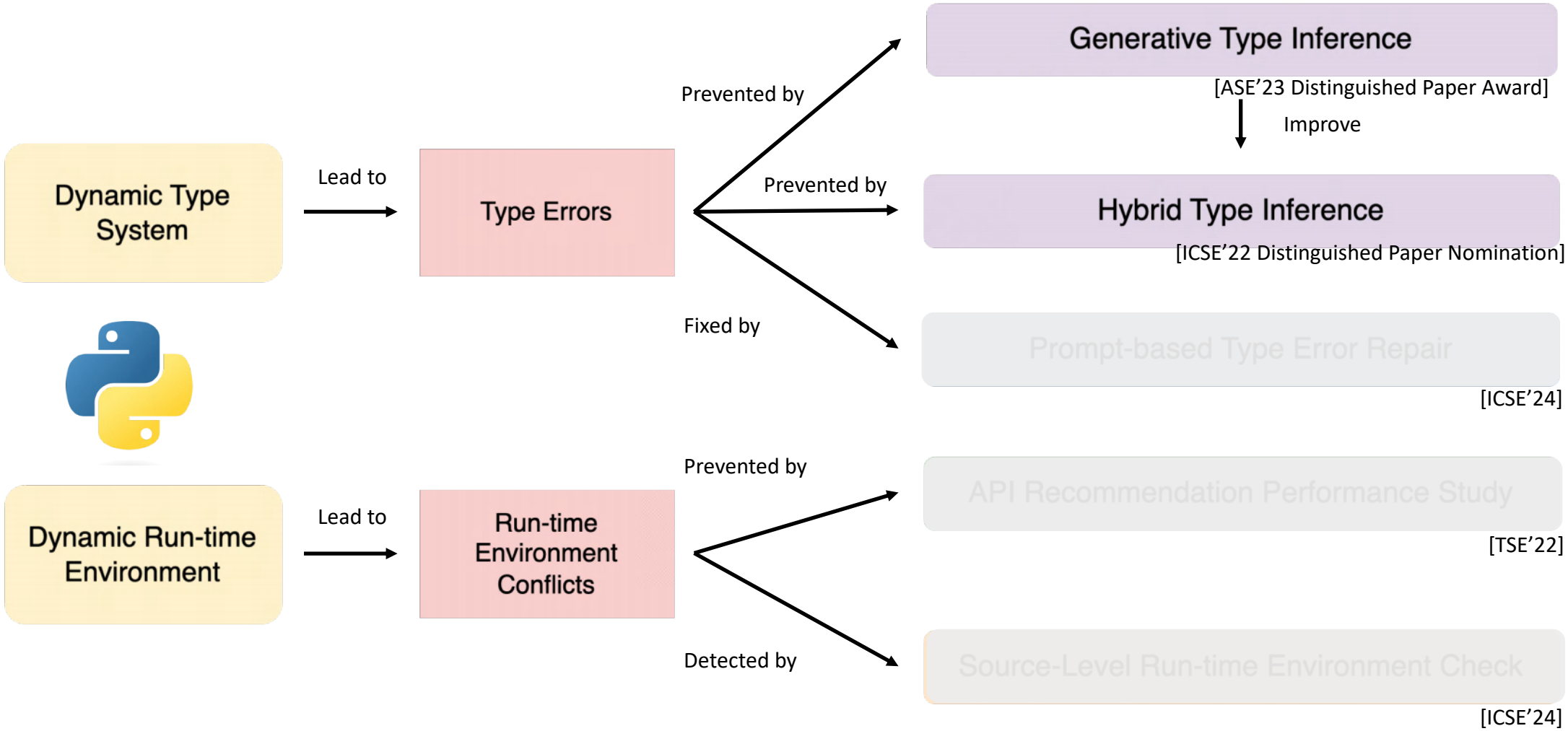
Pynguin

Test Case Generation

- **Pathway #2 (Repair): Implement automatic repair methods to fix issues caused by the dynamic type system.**



Outline of Thesis



Only first-author publications are listed.

How to Fix Existing Type Errors?

- Incorporate domain knowledge into prompt templates → Domain-aware prompts.

Complete mask	`if (fnType != null) {`
line replace	`<mask><mask> .... <mask><mask>`

line after	`<mask><mask> .... <mask><mask>`
	`if (fnType != null) {`
	`<mask><mask> .... <mask><mask>`

Partial mask	`return foundDigit && !hasExp;`
partial after	`<mask><mask> ... <mask> !hasExp;`
partial before	`return <mask><mask> ... <mask>`
Template mask	`primitiveValues.put(double.class, 0);`
method replace	`<mask><mask>...<mask>(double.class, 0);`
parameter replace	`primitiveValues.put(<mask><mask>..<mask>);`
single replace	`primitiveValues.put(double.class, <mask>);`
add parameter	`primitiveValues.put(double.class, 0, <mask>);`
Template mask	`if (endIndex < 0) {`
expression replace	`if (<mask><mask>...<mask>) {`
more ||/&& cond	`if (endIndex < 0 || <mask>..<mask>) {`
replace operator	`if (endIndex <mask> 0 ) {`

Prompt-based program repair

+ Do not need training set.
+ Most effective.
- Require good prompt template design.

Bug Line:

user_pass = '%s:%s' % (unquote(user), unquote(password))

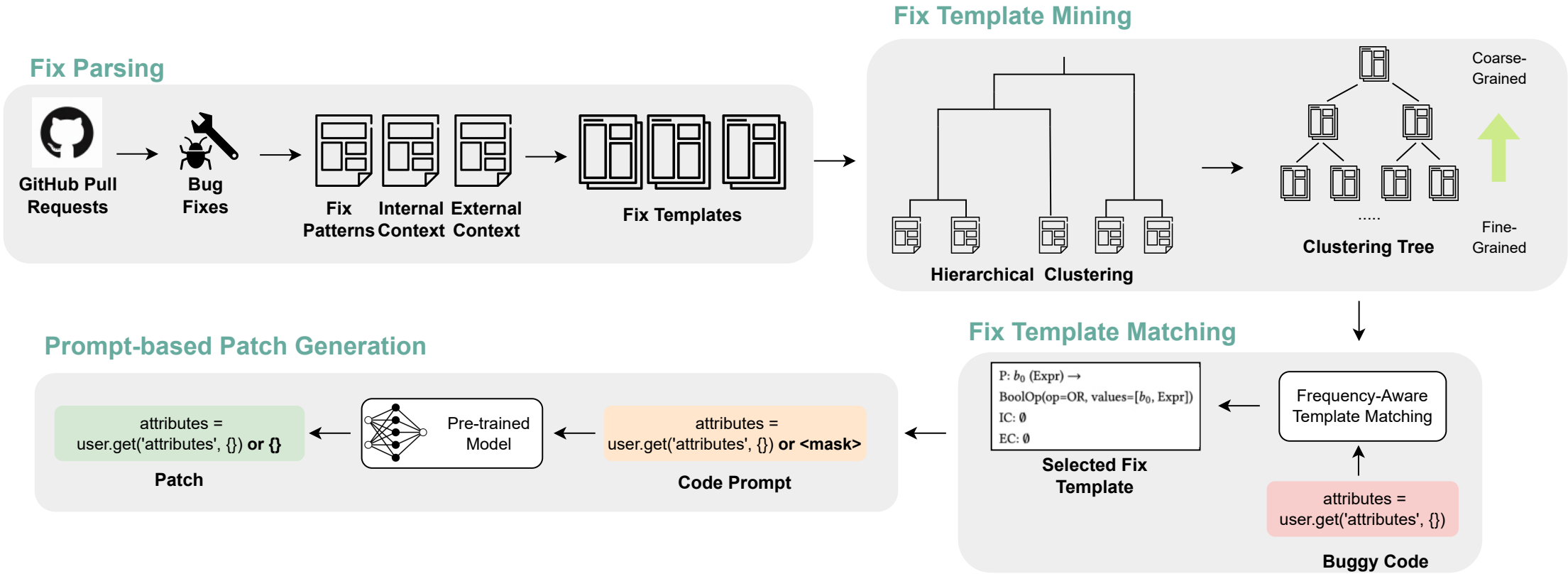
General prompt template:

user_pass = '%s:%s' % (<mask>...<mask>unquote(password))

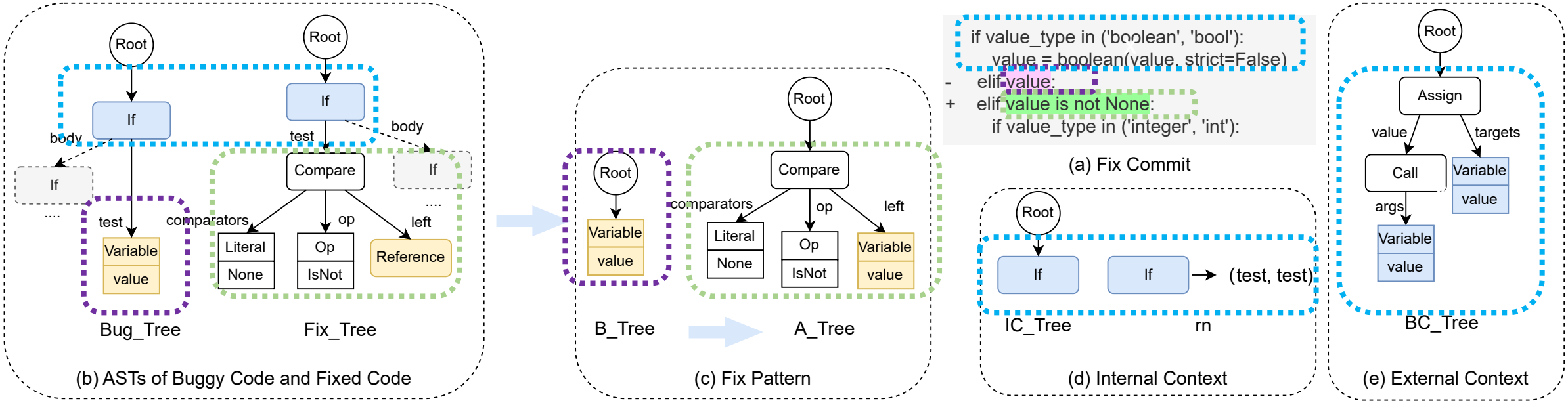
Domain-aware prompt template:

user_pass = <mask>...<mask>('%s:%s' % (unquote(user), unquote(password)))

Domain-Aware Type Error Repair - TypeFix

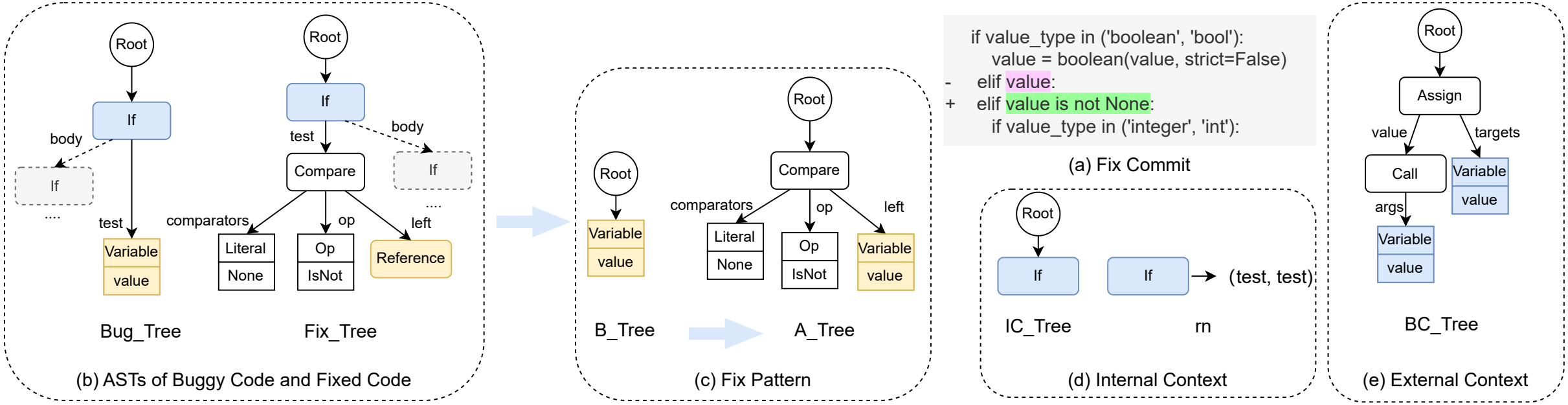


Phase I: Fix Parsing



- **Fix Pattern:** Indicate the exact code change.
- **Internal Context:** Indicate the statement where fix pattern should apply.
- **External Context:** Indicate the location of the statement in internal context.

Phase I: Fix Parsing



- Definition of Template Tree Node

A node is a quadruple (*bt, t, v, i*) where

bt ∈ {Variable, Op, Literal, Builtin, Type, Attribute, Expr, Stmt} is the base type of node,

t is the AST node type, *v* is the value, and *i* is the id.

Phase II: Fix Template Mining

Overall Methodology: Hierarchical Clustering

- Distance of Fix Patterns
  - Defined as 1 minus the rate of same nodes in two template trees.
  - Calculate from the root to leaves, i.e., two nodes can be compared if and only if their parent nodes are the same (top-down).
- Distance of Contexts
  - Defined as 1 minus the rate of same nodes in two template trees.
  - Calculate from leaves to the root, i.e., two nodes can be compared if and only if their children nodes in the leaf-root path are the same (bottom-up).

Phase II: Fix Template Mining

- Abstraction of 2 Nodes *a* and *b* in Fix Patterns or Contexts
  - **Same Node:** *a* and *b* are exactly the same, and they can be reserved for the generalized fix template.
  - **Value Abstraction:** *a* and *b* have the same types but different values. We create a node with the same type and set the value as a special *ABS* token to indicate a hole.
  - **Type Abstraction:** *a* and *b* have the same base types but different types and values. We create a node with the same base type, and set the type and value as a special *ABS* token to indicate a hole.
  - **Node Removal:** *a* and *b* have no common attributes. We directly remove the two nodes.

Node: (*bt, t, v, i*)

a: (**Literal, int, 1**, 1024)
b: (**Literal, int, 1**, 2024)
res: (**Literal, int, 1**, -)

a: (**Literal, int,** 1, 1024)
b: (**Literal, int,** 2, 2024)
res: (**Literal, int**, *ABS*, -)

a: (**Literal**, int, 1, 1024)
b: (**Literal**, str, "a", 2024)
res: (**Literal**, *ABS*, *ABS*, -)

a: (**Literal**, int, 1, 1024)
b: (**Op**, add, -, 2024)
res:-

Phase III: Fix Template Matching

- Select Fix Templates
  - We match the ASTs of the buggy programs with the contexts of mined fix templates.
  - For the same type of fix templates, we select the most detailed fix template since it has the most domain knowledge.

- Rank Different Types of Fix Templates
  - Group different fix templates with the same contexts.
  - Rank the fix templates in one group according to the occurrence frequency obtained in the mining phase.
  - Rank the groups according to the abstraction ratio, i.e., the rate of program holes that require LLMs to synthesize.

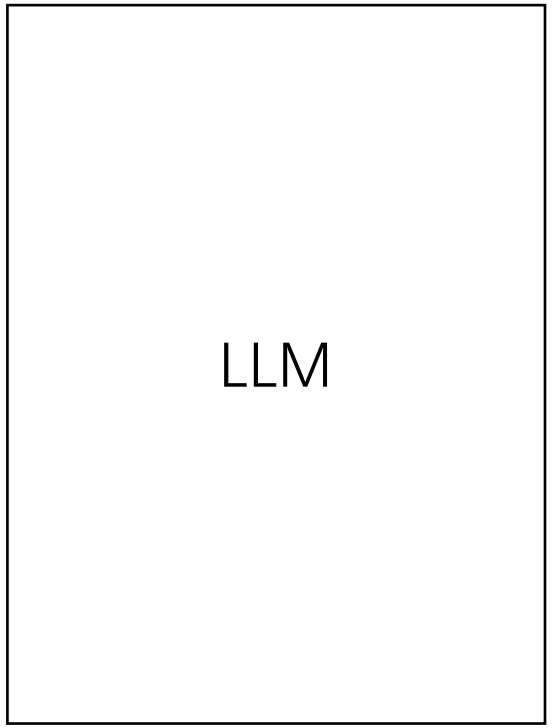
Phase IV: Patch Generation

Bug Line:

user_pass = ’%s:%s’ % (unquote(user), unquote(password))

Domain-aware prompt template:

user_pass = <mask>…<mask>(’%s:%s’ % (unquote(user), unquote(password)))



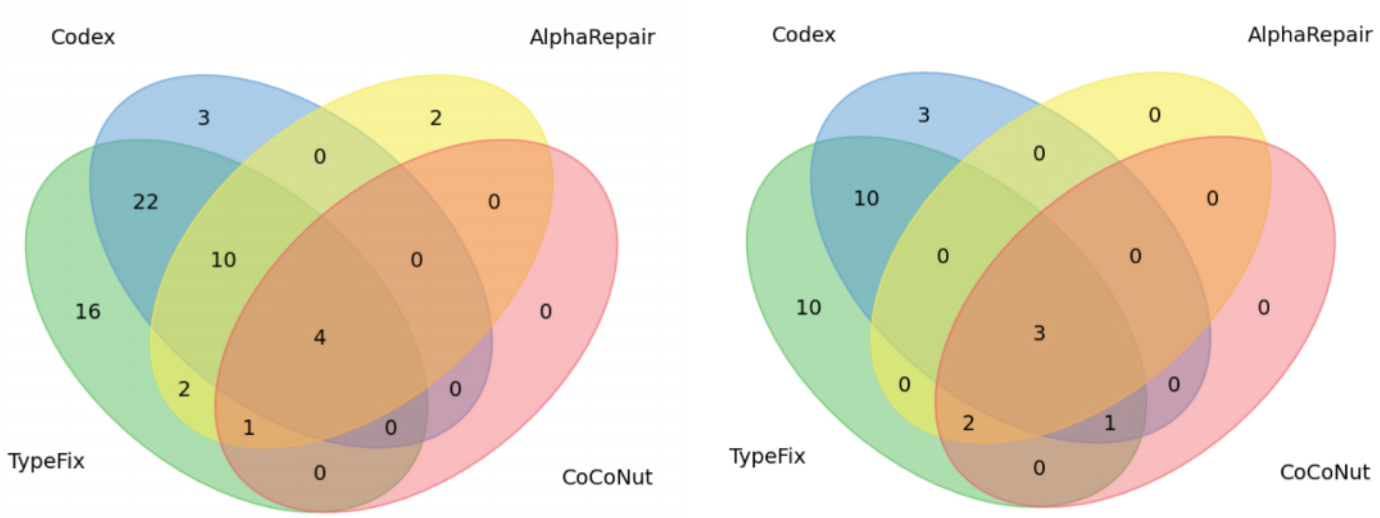
Patch:

user_pass = to_bytes(’%s:%s’ % (unquote(user), unquote(password)))

Evaluation

TypeBugs						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
airflow	14	9/9	4/4	7/7	1/6	0/4
beets	1	0/0	0/1	0/0	0/0	0/0
core	9	7/7	5/7	4/5	4/4	2/3
kivy	1	0/0	0/1	0/0	0/1	0/1
luigi	2	0/2	0/0	0/2	1/2	0/0
numpy	3	0/3	0/2	0/1	0/2	0/0
pandas	48	21/32	17/27	18/19	11/22	3/10
rasa	2	2/2	0/0	2/2	0/0	0/0
requests	4	4/4	4/4	2/2	0/1	0/0
rich	4	2/3	0/1	1/1	0/0	0/0
salt	8	5/8	5/5	4/5	1/5	0/2
sanic	2	0/0	2/2	0/0	0/0	0/0
scikit-learn	7	2/3	2/3	1/2	0/0	0/0
tornado	1	0/0	1/1	0/0	0/0	0/0
Zappa	3	3/3	1/1	0/0	1/3	0/1
Total	109	**55/76**	41/59	39/46	19/46	5/21
Fix Rate (%)	-	**50.5**	37.6	35.8	17.4	4.6

BugsInPy						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
ansible	1	0/0	0/0	0/0	0/0	0/0
fastapi	1	1/1	0/0	1/1	0/0	0/0
keras	7	4/6	1/1	0/3	0/4	0/4
luigi	7	4/5	3/5	3/3	0/0	0/0
pandas	19	4/13	4/6	2/6	3/10	3/8
scrapy	12	10/11	5/7	10/12	1/4	2/4
spacy	1	0/1	0/1	0/0	0/1	0/1
tornado	2	1/1	1/1	1/1	0/1	0/1
youtube-dl	4	2/3	1/1	0/2	1/1	1/1
Total	54	**26/41**	15/22	17/28	5/21	6/19
Fix Rate (%)	-	**48.1**	27.8	31.5	9.3	11.1



Unique type errors fixed by each approach.

TypeFix successfully fixes 55 and 26 type errors in two benchmarks, outperforming state-of-the-art approaches by at least 14 type errors and 9 type errors, respectively. Meanwhile, TypeFix obtains the most unique type error fixes in two benchmarks.

Evaluation

Approach	TypeBugs		BugsInPy	
	#Unique	Coverage	#Unique	Coverage
TypeFix	24	83 (76.1%)	16	40 (74.1%)
PyTER	10	46 (42.2%)	5	18 (33.3%)

Comparison with rule-based approach

TypeFix achieves a template coverage of about 75% on both benchmarks, which is 30% larger than that achieved by fix templates manually defined in PyTER.

	TypeBugs		BugsInPy	
	#Correct	#Plausible	#Correct	#Plausible
No Template	19	41	6	20
Add	+11	+10	+3	+5
Remove	+1	+1	+0	+0
Replace	+6	+8	+4	+9
Insert	+18	+16	+13	+17
Total	55	76	26	41

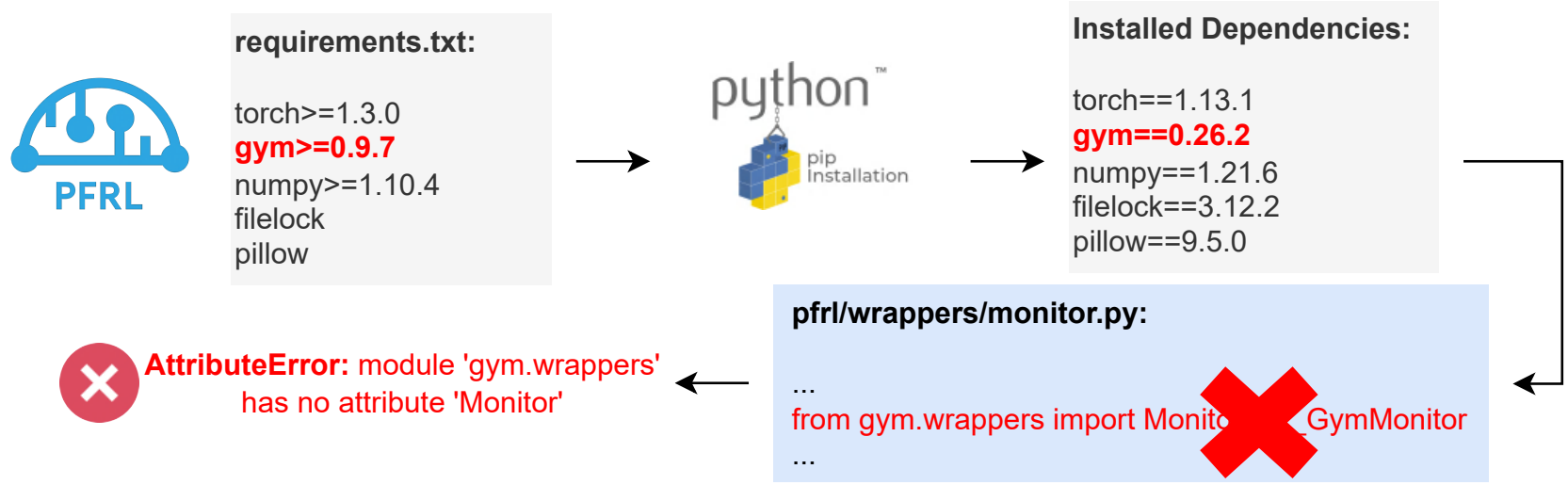
Ablation Results

Ablation results also demonstrate the usefulness of fix templates mined by TypeFix under each category.

THREE

Dynamic Run-time Environment

Improve the Reliability of Dynamic Run-time Environment

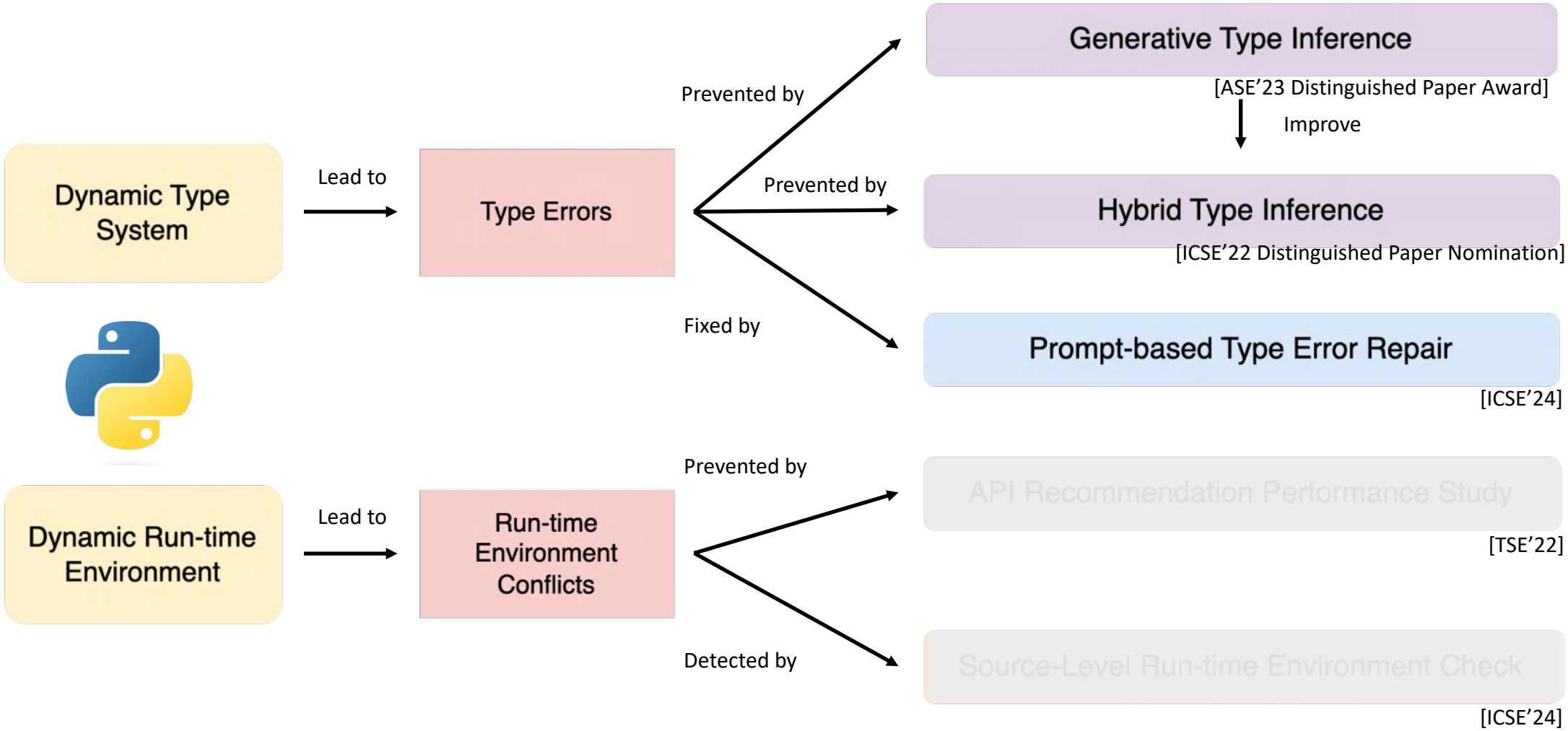


- Pathway #1 (Prevention):

  Run-time environments provide implementations for external APIs used in the code.

  → Provide high-quality API recommendation to avoid API misuse.

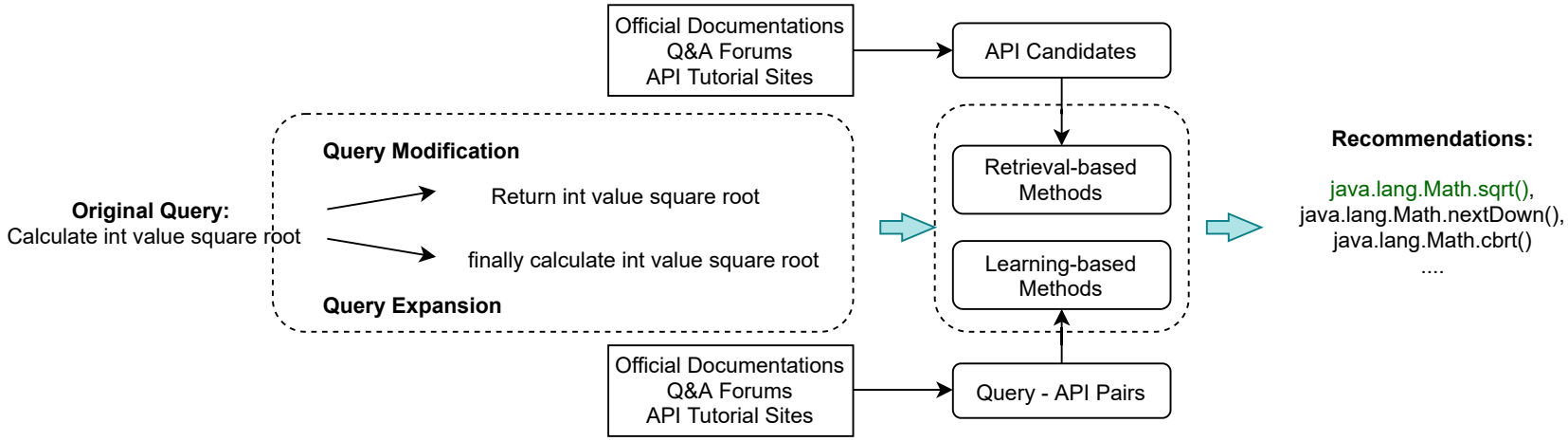
Outline of Thesis



Only first-author publications are listed.

Two Categories of API Recommendation Approaches

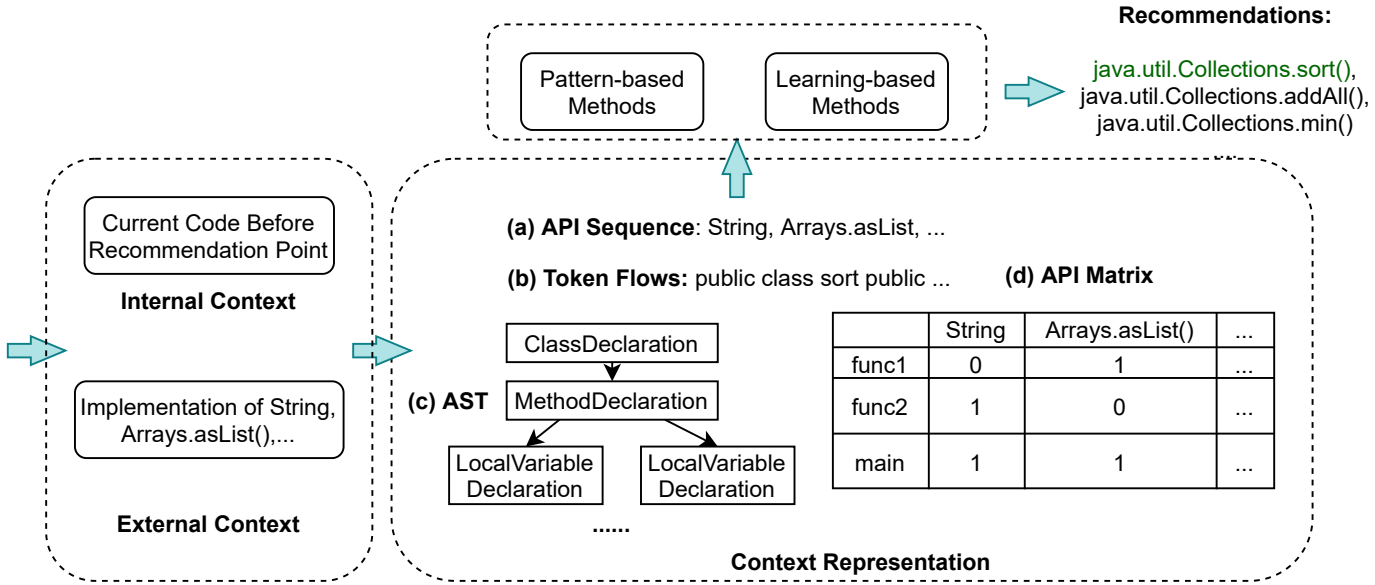
Query-Based:



Code-Based:

```
1 public class Sort {
2  public static void main(String args[]) {
3    String[] strArray =
4       new String[] { "example" };
5    List l = Arrays.asList(strArray);
6    Collections.<Recommendation Point>;
7    ...
8  }
9 }
```

Target Code



APIBench-Q for Query-Based Approaches

- Mining Stack Overflow
  - All posts from Aug 2008 to Feb 2021
  - 1,756,183 Java posts and 1,661,383 Python posts

↓ Format Check

  - 148,938 Java posts and 156,493 Python posts

↓ Keyword Filtering

  - 13,755 posts

↓ Manual Check

  - 1,320 Java queries and 1,925 Python queries

- Mining Tutorial Websites

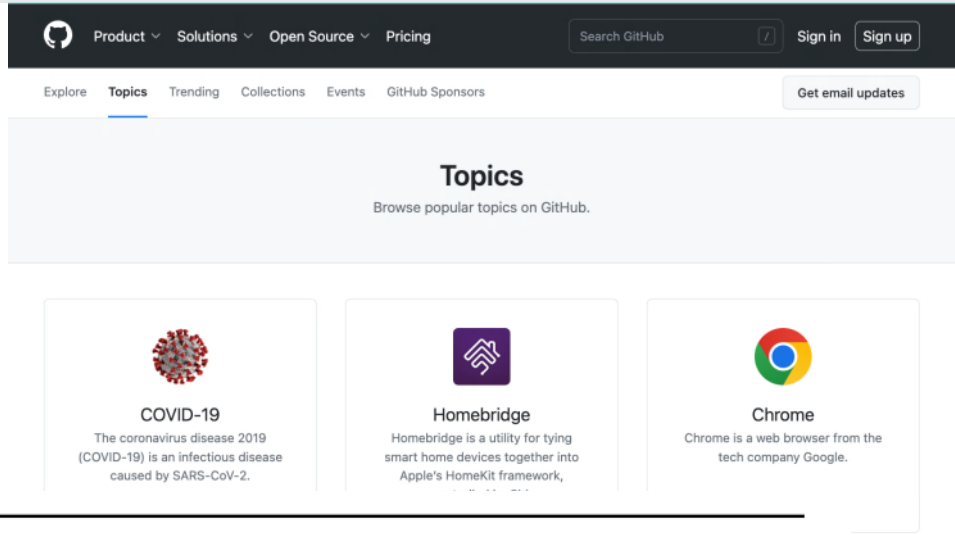
GeeksforGeeks

Kode Java

PL	Stack Overflow			Tutorial Websites		
	Ori.	Exp.	Mod.	Ori.	Exp.	Mod.
Python	1,925	78,157	100,100	2,384	95,360	123,968
Java	1,320	80,343	68,640	5,243	319, 783	272,636

APIBench-C for Code-Based Approaches

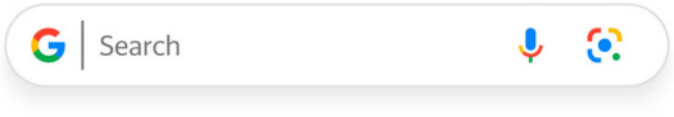
- Mining GitHub
  - General: 1,000 most starred + 1,000 most forked repos at entire Github.
  - Specific Domain: 500 most starred + 500 most forked repos under one topic.

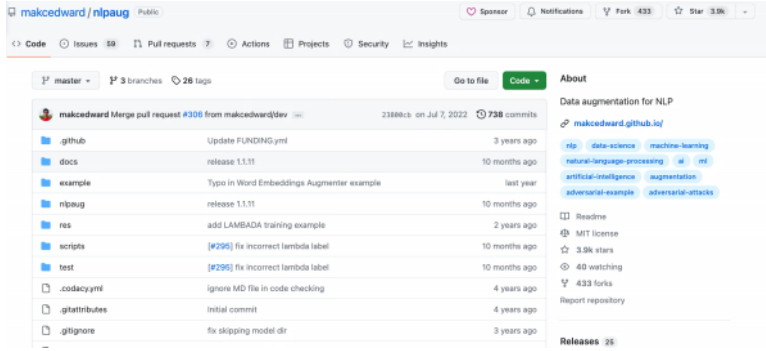


PL	Domain	#Projects	#Files	LOC (per func)	#API (per func)	Total number of APIs (only testset) Standard	User-defined	Popular	LOC Threshold of Short Func	LOC Threshold of Long Func
Python	General	899	230,064	15.24	5.55	1,363,240	1,747,878	54,244	8.875	54.875
	ML	323	46,556	13.89	6.08	629,437	339,821	125,377	12.65	46.05
	Security	126	15,785	18.98	6.72	111,393	64,809	3,613	6	86.5
	Web	568	82,771	14.14	5.05	369,114	241,602	11,832	7.35	51.625
	DL	307	39,577	14.58	6.25	413,295	220,228	76,654	11.675	52.525
Java	General	935	1,056,790	11.16	4.06	5,164,481	3,808,124	36,178	6.26	19.2
	Android	377	87,468	8.24	2.91	517,461	267,141	75,069	7.28	16.8
	ML	52	41,377	12.82	4.77	194,013	136,963	0	7.52	19.74
	Testing	55	23,618	9.93	3.98	105,577	55,241	22	6.44	15.68
	Security	58	20,445	12.35	5.32	125,558	74,471	1,243	6.88	20.78

Query-Based Baselines

Approach	Category/ Data Source	PL	Venue	Year
Query Reformulation				
Google Prediction Service [22]	Query expansion, modification	Any	-	2021
NLPAUG [40]	Query expansion, modification	Any	-	2021
SEQUER [8]	Query expansion, modification	Any	ICSE	2021
NLP2API [56]	Query expansion	Java	ICSME	2018
Query-Based API Recommendation				
RACK [58]	Official documentation, Stack Overflow	Java	ICSE	2016
KG-APISumm [38]	Official documentation, Wikipedia	Java	FSE	2019
Naive Baseline	Official documentation	Any	-	2021
DeepAPI [23]	Official documentation	Java	FSE	2016
Lucene [16]	Official documentation	Any	-	2021
BIKER [27]	Official documentation, Stack Overflow	Java	ASE	2018





Effectiveness of Query-based Approaches

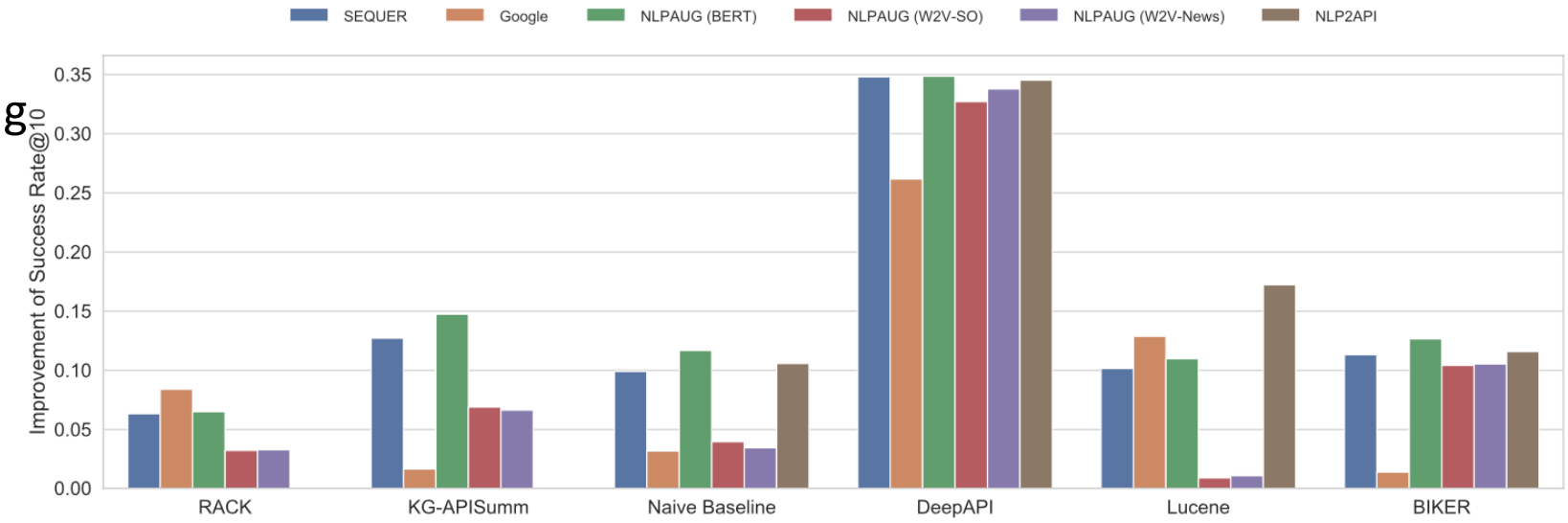
Class Level vs. Method Level

Baseline	Level	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
RACK	Class	0.17	0.30	0.35	0.41	0.17	0.23	0.24	0.24	0.25	0.17	0.24	0.26	0.28
KG-APISumm	Class	0.19	0.33	0.40	0.50	0.19	0.25	0.26	0.27	0.28	0.19	0.24	0.27	0.31
Naive Baseline	Class	0.07	0.13	0.16	0.21	0.07	0.10	0.10	0.10	0.11	0.07	0.09	0.10	0.13
	Method	0.02	0.03	0.04	0.05	0.01	0.02	0.03	0.03	0.03	0.07	0.09	0.10	0.13
DeepAPI	Class	0.19	0.27	0.29	0.30	0.19	0.22	0.23	0.23	0.23	0.17	0.22	0.23	0.24
	Method	0.05	0.09	0.10	0.11	0.05	0.07	0.07	0.07	0.07	0.17	0.22	0.23	0.24
Lucene	Class	0.15	0.21	0.24	0.29	0.15	0.17	0.18	0.17	0.19	0.12	0.15	0.16	0.20
	Method	0.04	0.08	0.10	0.14	0.04	0.06	0.06	0.06	0.07	0.12	0.15	0.16	0.20
BIKER	Class	0.33	0.51	0.59	**0.67**	0.33	0.41	0.41	**0.39**	**0.44**	0.27	0.32	0.35	**0.42**
	Method	0.12	0.23	0.29	**0.37**	0.12	0.16	0.18	**0.18**	**0.19**	0.27	0.32	0.35	**0.42**

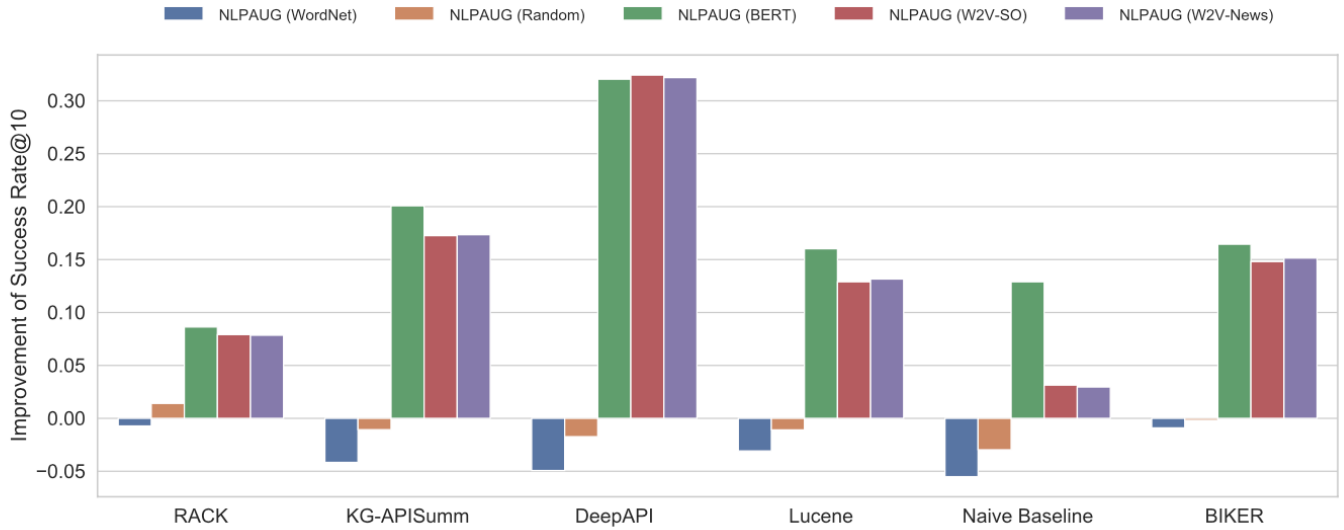
Finding: Existing approaches fail to predict 57.8% method-level APIs that could be successfully predicted at the class level. **Accurately recommending the method-level APIs** still remains a great challenge.

Impact of Query Reformulation Techniques

Finding: Query reformulation techniques are **quite effective** in helping query-based API recommendation approaches give the correct API by adding an average boost of 27.7% and 49.2% on class-level and method-level recommendations, respectively.



(a) Query Expansion

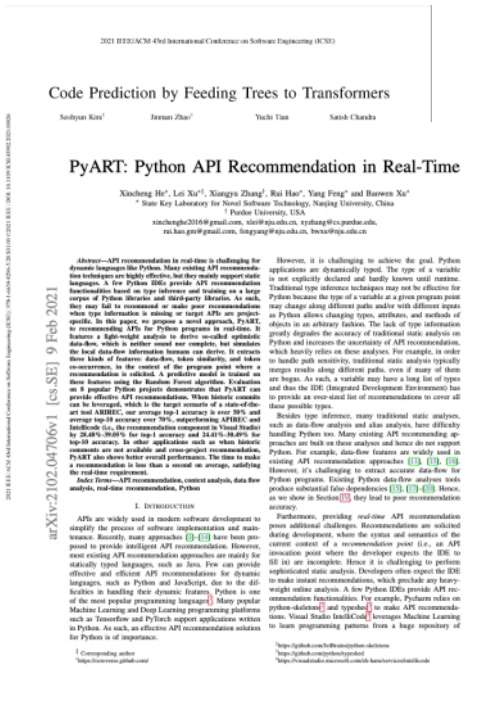


(b) Query Modification

Finding: Query expansion is **more stable and effective** to help current query-based API recommendation approaches give correct APIs than query modification.

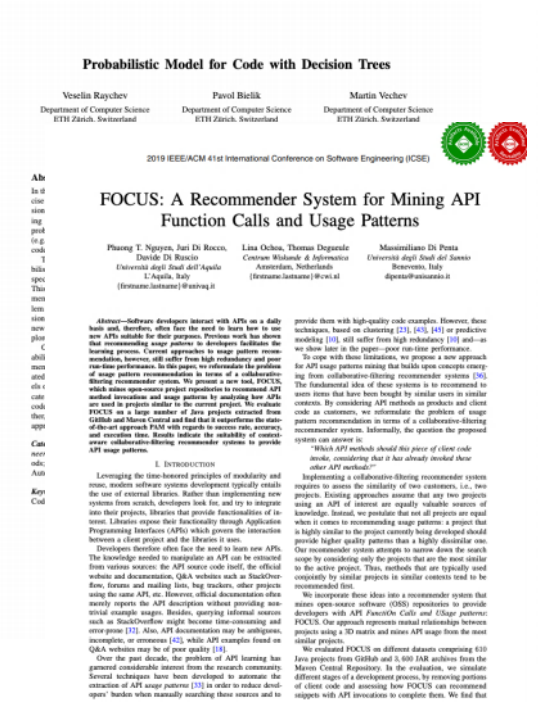
Code-Based Baselines

Approach	Representation	PL	Venue	Year
Practical IDE				
PyCharm [30]	Code tokens	Python	-	2021
Visual Studio Code [43]	Code tokens	Python	-	2021
Eclipse [17]	Code tokens	Java	-	2021
IntelliJ IDEA [29]	Code tokens	Java	-	2021
Approach in Academia				
TravTrans [32]	AST	Python	ICSE	2021
PyART [24]	Token flow Data flow	Python	ICSE	2021
Deep3 [59]	AST, DSL	Python	ICML	2016
FOCUS [49]	API Matrix	Java	ICSE	2019
PAM [18]	API sequence	Java	FSE	2016
PAM-MAX	API sequence	Java	FSE	2016



Code Prediction by Feeding Trees to Transformers

PyART: Python API Recommendation in Real-Time



Probabilistic Model for Code with Decision Trees

FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns

Parameter-Free Probabilistic API Mining across GitHub

Cross-Domain Performance

General: multiple domains

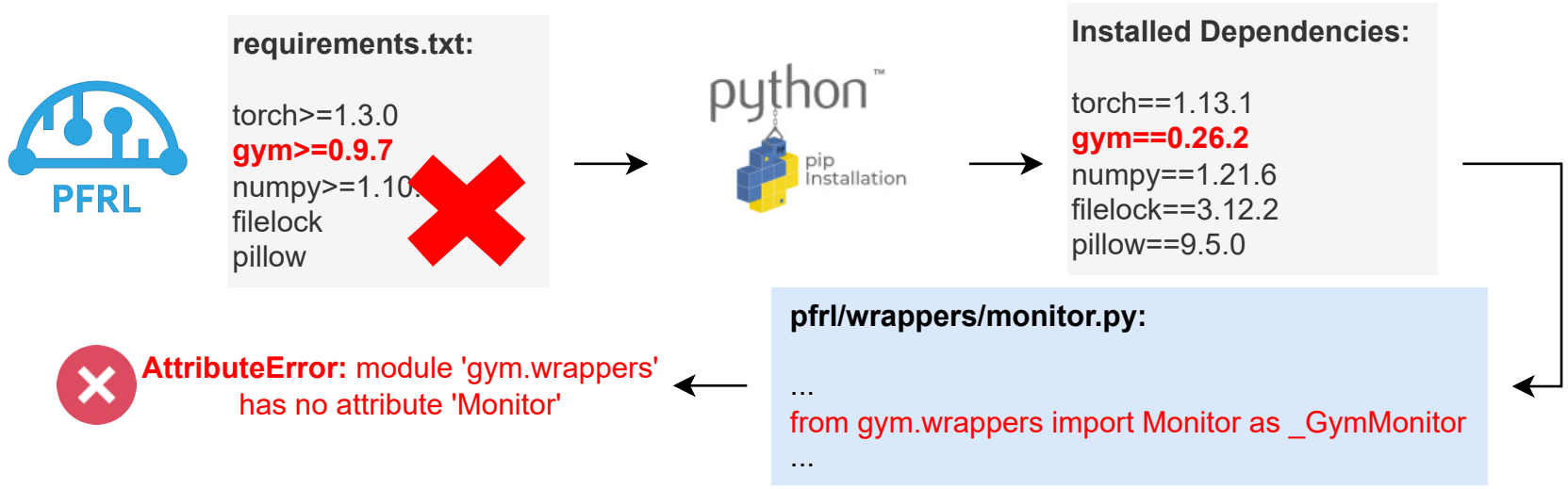
Training Domain	TravTrans				Deep3				PyART			
	ML	Security	Web	DL	ML	Security	Web	DL	ML	Security	Web	DL
ML	0.64	0.58	0.53	**0.71**	0.42	0.41	0.36	**0.48**	0.39	0.35	0.40	**0.40**
Security	0.40	**0.54**	0.54	0.39	0.31	**0.51**	0.42	0.29	0.36	**0.48**	0.47	0.36
Web	0.54	0.63	**0.64**	0.51	0.33	0.42	**0.46**	0.31	0.42	0.47	**0.50**	0.40
DL	0.66	0.58	0.50	**0.68**	0.44	0.39	0.33	**0.44**	0.43	0.36	0.38	**0.45**
General	0.72	0.76	0.78	0.74	0.55	0.65	0.62	0.57	0.44	0.44	0.46	0.46

Finding: Training on **multiple domains** helps the current approaches to recommend APIs in different single domains, and the performance is **even better than** only training on the certain single domain.

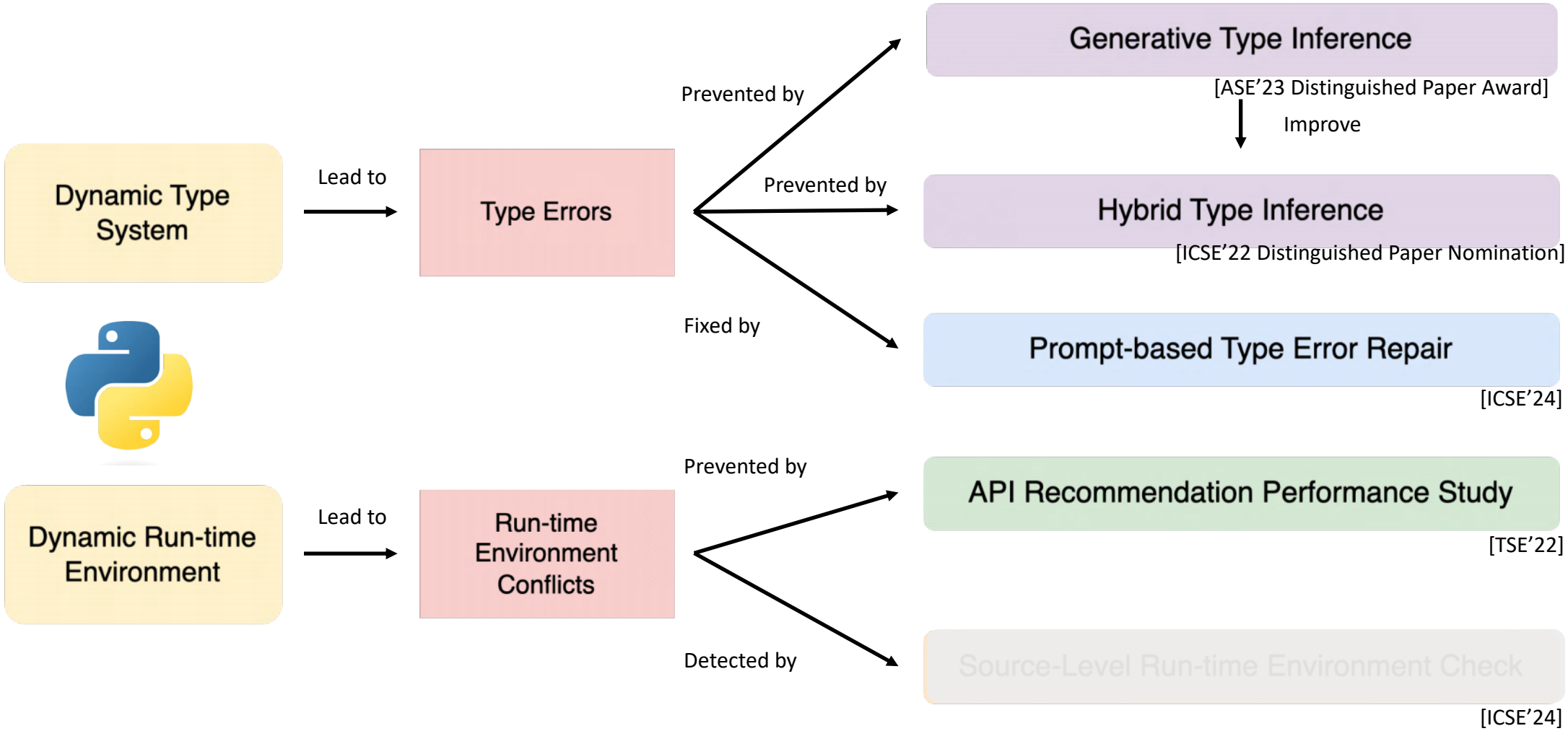
Improve the Reliability of Dynamic Run-time Environment

- Pathway #1 (Prevention): Provide accurate API recommendation to avoid external API misuse.

- **Pathway #2 (Detection): Detect compatibility issues in the configuration files before software usage.**



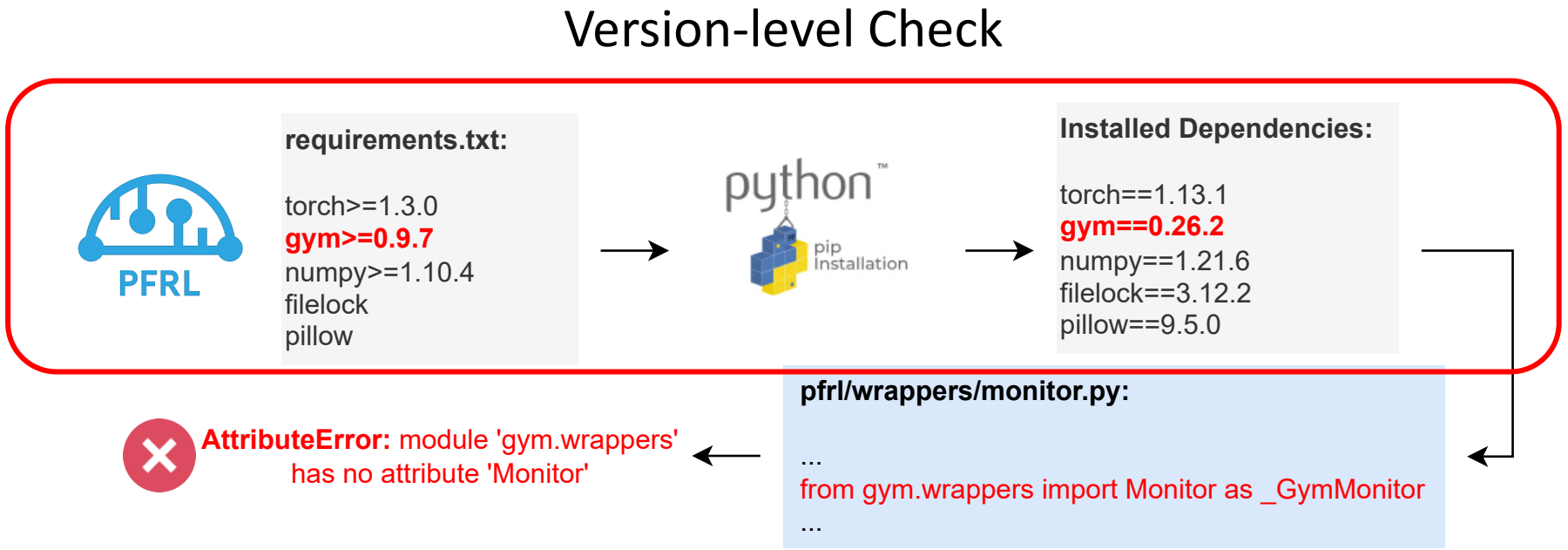
Outline of Thesis



Only first-author publications are listed.

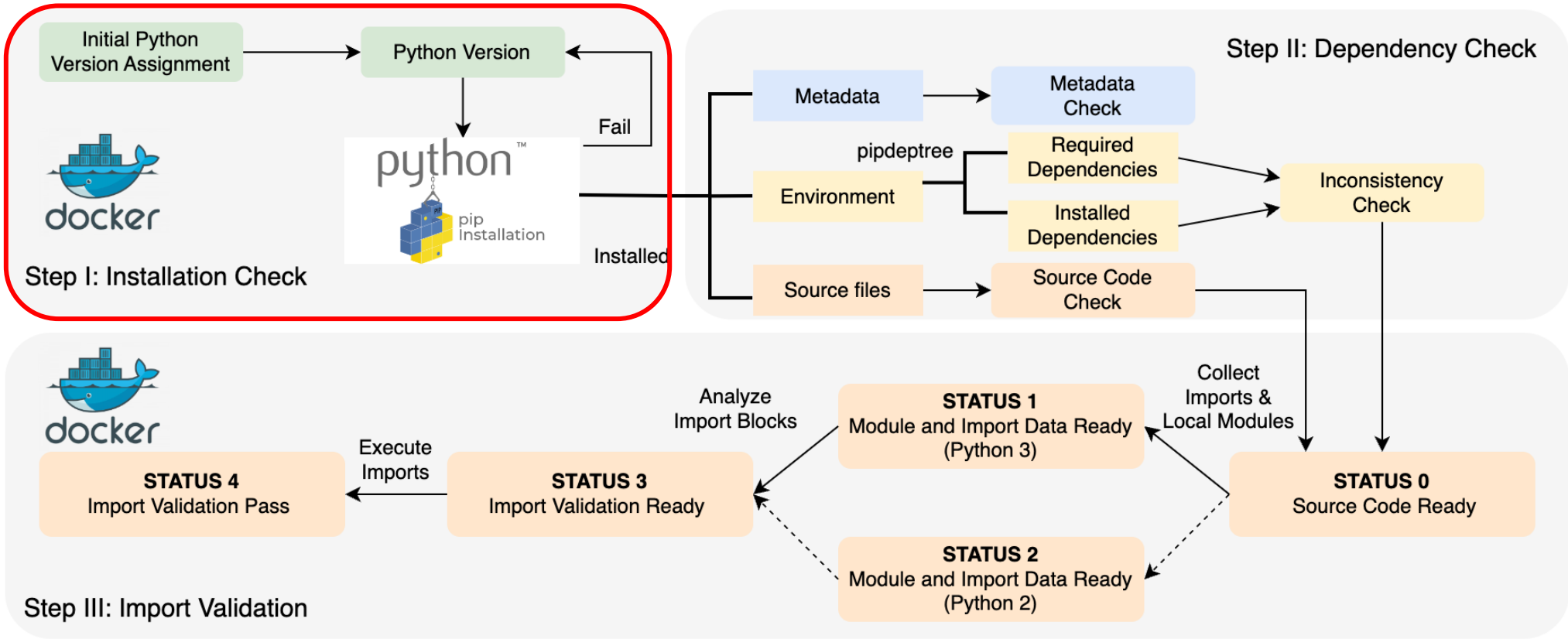
Source-level Run-time Environment Conflict Check

- Version constraints defined in configuration files **are not reliable.**
- We should validate whether the **import statements** in source code can be successfully executed.



Source-level Run-time Environment Conflict Check

- **Installation Check:** Assign the correct Python version and check whether a package can be successfully installed based on configuration files.



Installation Check

- Python Version Assignment

Examine the classifiers set by developers in PyPI

↓ Fail

Choose the latest Python version released 180 days before the package release time

↓ Fail

Copy the Python version from other releases of the same package

↓ Fail

Choose commonly used Python versions 2.7, 3.6, 3.10, etc

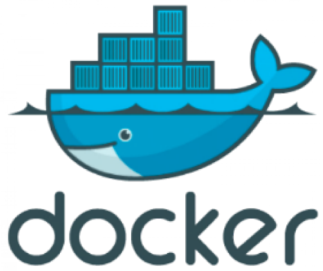
↓ Fail

Try all Python versions

Programming Language

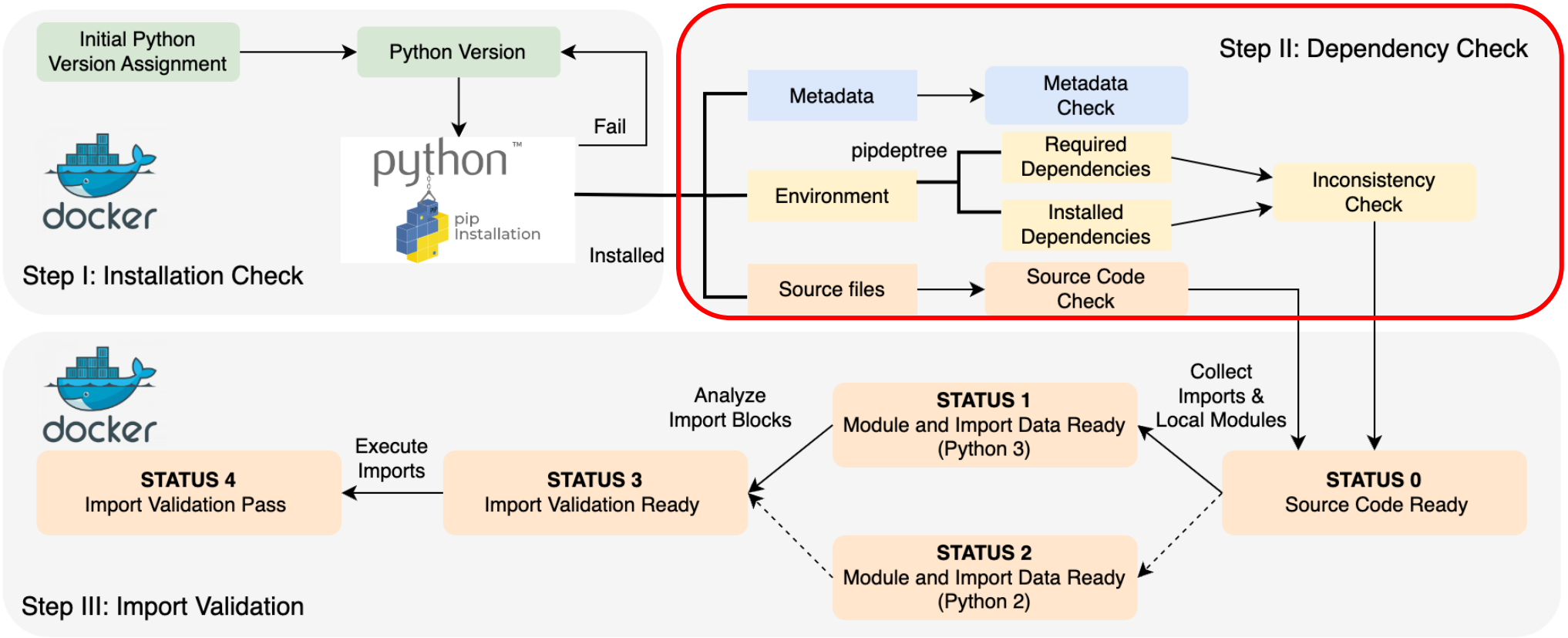
- C++
- Python :: 3
- Python :: 3.8
- Python :: 3.9
- Python :: 3.10

PyPI Classifiers



Source-level Run-time Environment Conflict Check

- **Dependency Check:** Check potential conflicts between indicated versions in configurations and installed versions via pip.

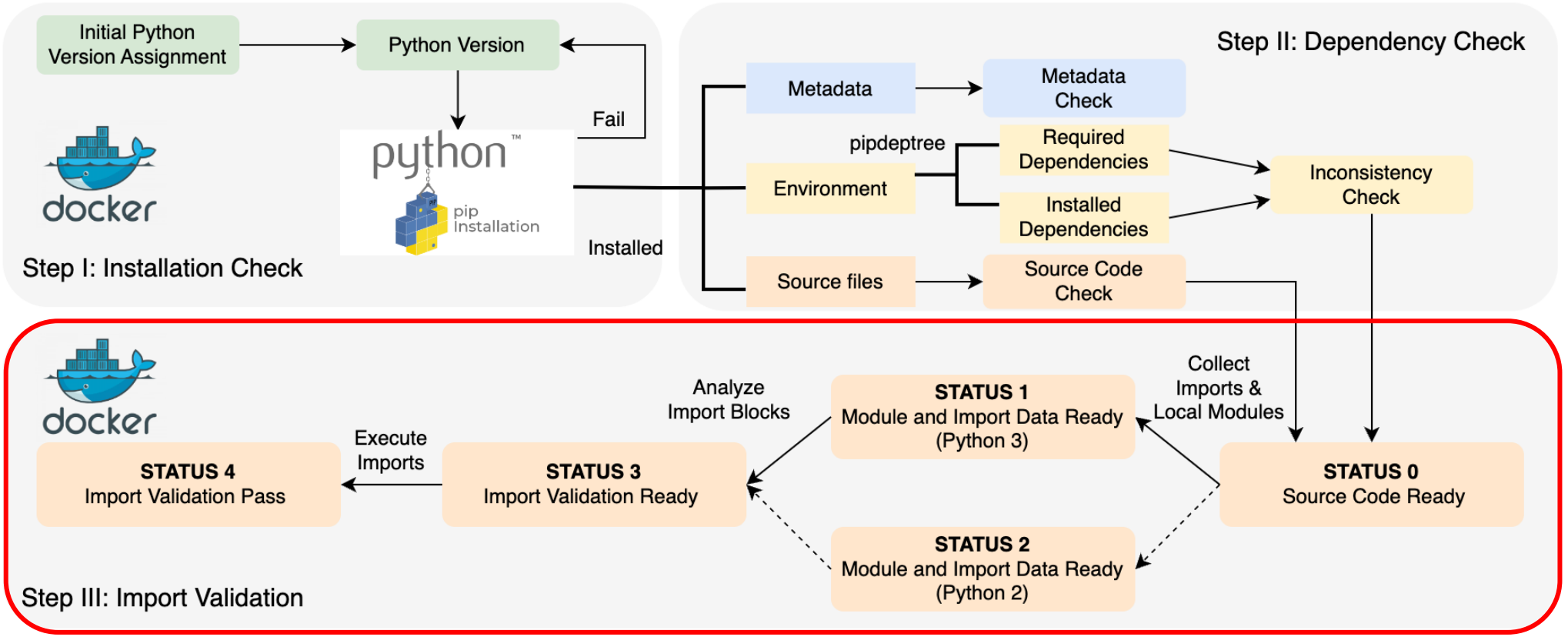


Dependency Check

- **Metadata Check**
  - Existence of file <package>-<version>.dist-info.
  - Top modules in file top_level.txt.
- **Run-time Environment Check**
  - Solve the valid dependencies in configurations provided by developers.
  - Collect the installed dependencies in the run-time environment built by Installation Check.
  - Check inconsistences between the required dependencies and installed dependencies.
- **Source Files Check**
  - Locate the source files based on the modules provided in the configurations.
  - Check the syntax of all source files.

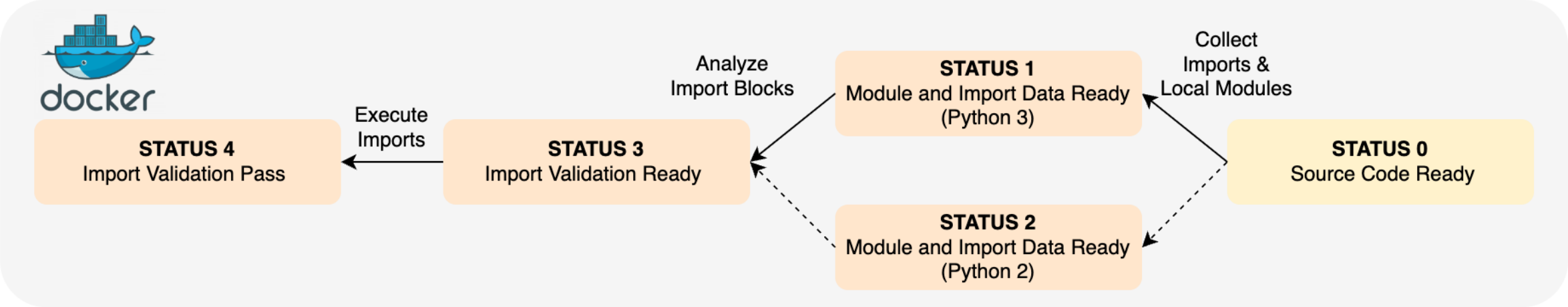
Source-level Run-time Environment Conflict Check

- **Import Validation:** check potential conflicts between import statements in source code and the installed run-time environment.



Import Validation

- Imports
  - Internal Imports: introduce local modules within the project.
  - **External Imports: require third-party packages from the run-time environment.**
- Collect Local Modules
  - All Python files and sub-directories with __init__.py file in the same directory.
  - Image files such as .so and .pyd.



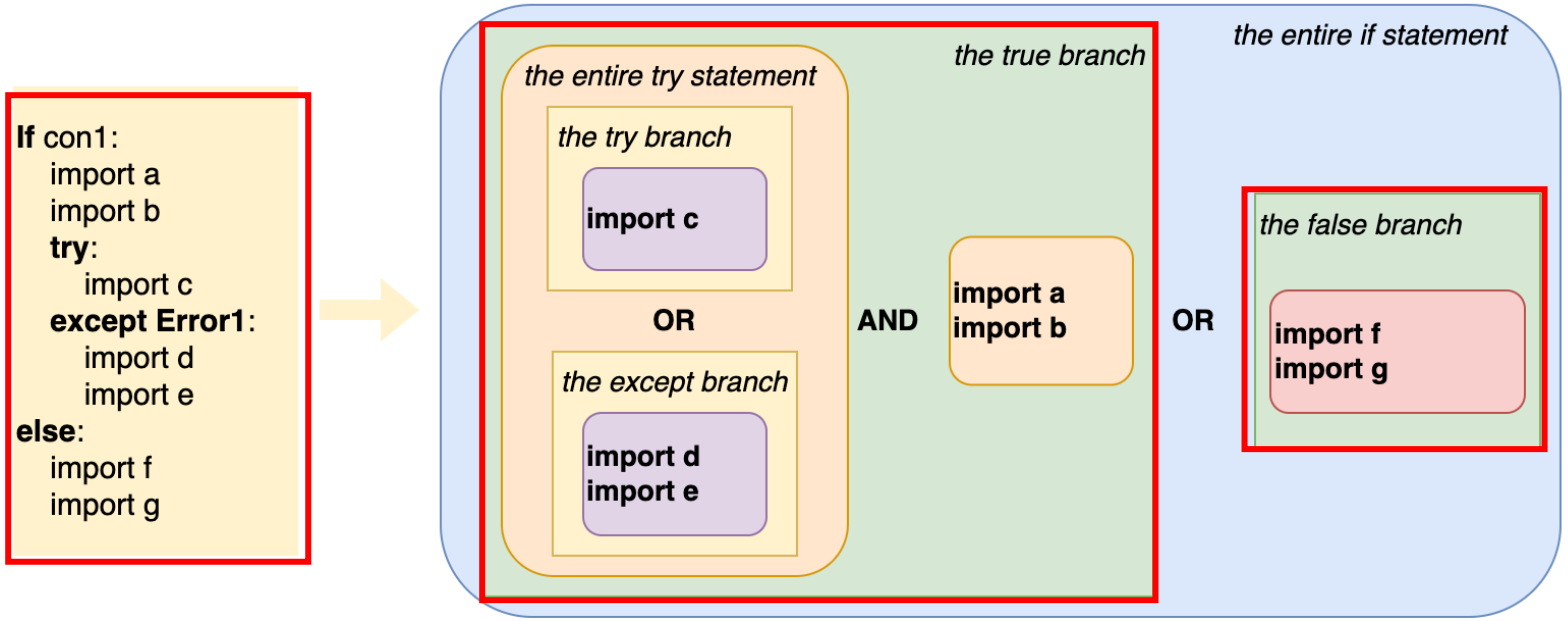
Import Validation

- Import Blocks
  - Developers employ different methods to handle different run-time environments, such as using if-else statements and try-except statements to incorporate import statements.

```
if sys.platform.startswith("java"):
    import platform

    os_name = platform.java_ver()[3][0]
    if os_name.startswith("Windows"):  # "Windows XP", "Windows 7", etc.
        system = "win32"
    elif os_name.startswith("Mac"):  # "Mac OS X", etc.
        system = "darwin"
    else:  # "Linux", "SunOS", "FreeBSD", etc.
        # Setting this to "linux2" is not ideal, but only Windows or Mac
        # are actually checked for and the rest of the module expects
        # *sys.platform* style strings.
        system = "linux2"
else:
    system = sys.platform
```

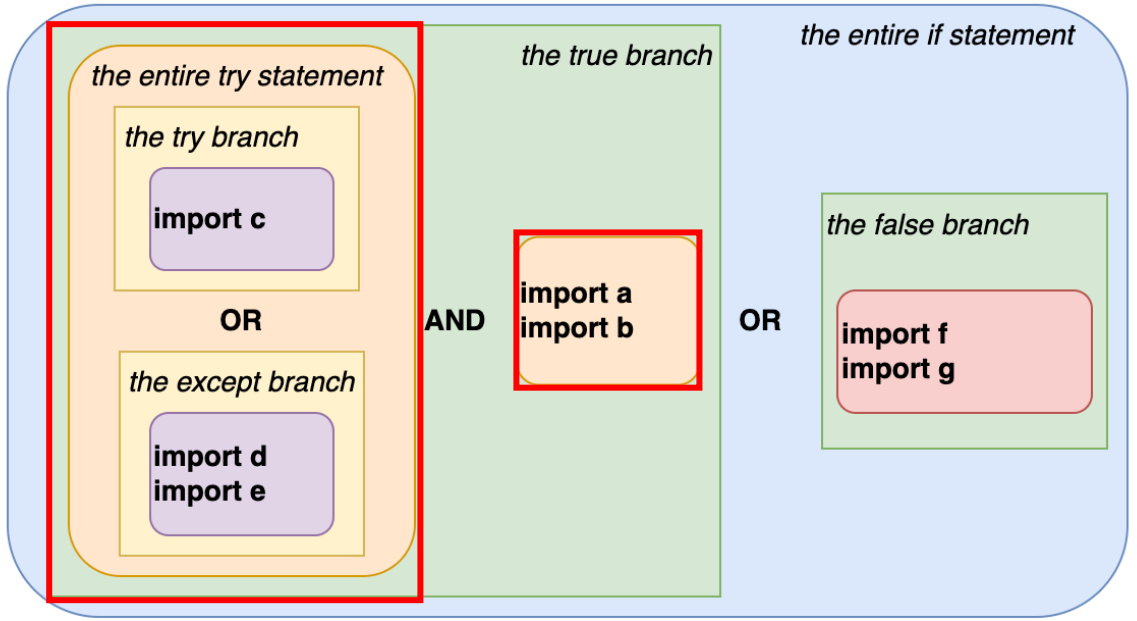
Block Analysis

- Block analysis aims to reformulate an import block to a boolean expression, so that we know whether an import block is successfully executed.



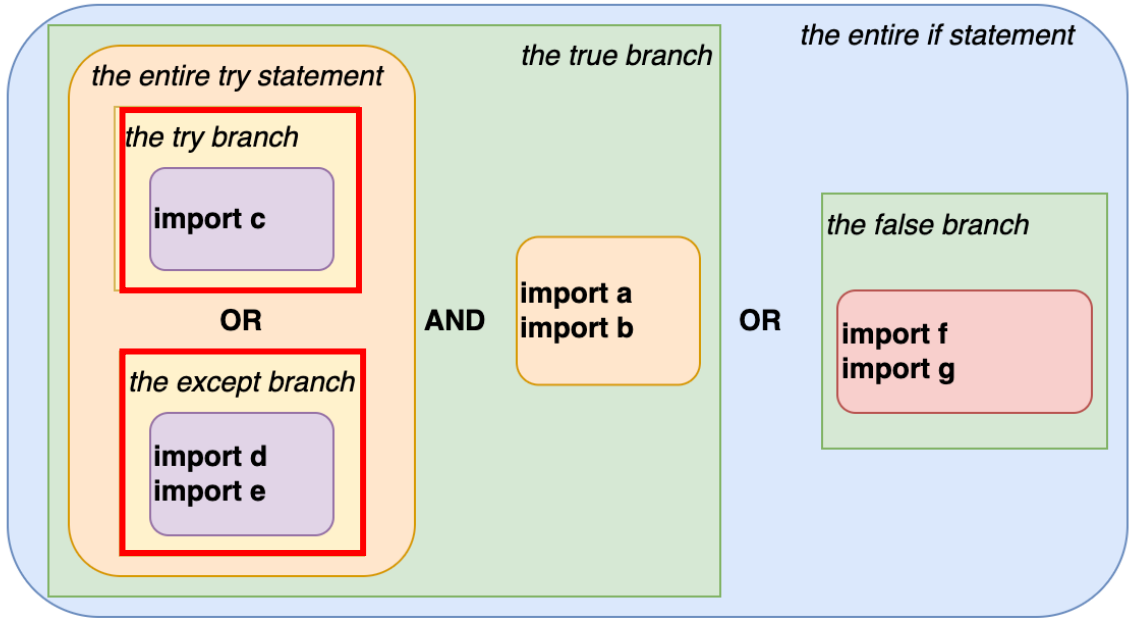
Block Analysis

```
If con1:
    import a
    import b
    try:
        import c
    except Error1:
        import d
        import e
else:
    import f
    import g
```



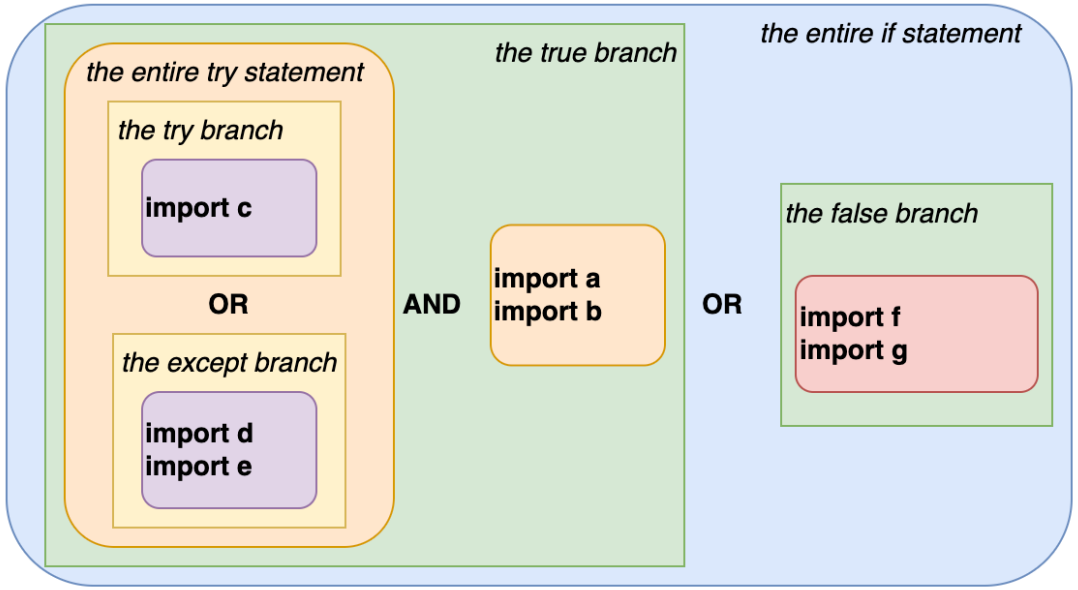
Block Analysis

```
If con1:
    import a
    import b
    try:
        import c
    except Error1:
        import d
        import e
else:
    import f
    import g
```



Block Analysis

```
If con1:
    import a
    import b
    try:
        import c
    except Error1:
        import d
        import e
else:
    import f
    import g
```



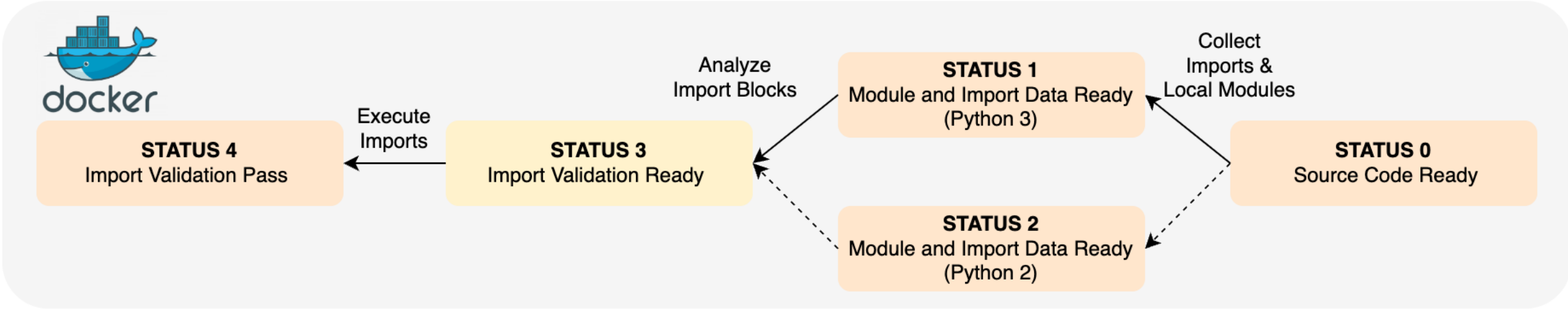
Boolean Expression

```
(
    (a and b) and
    (
        c or
        (d and e)
    )
) or
(f and g)
```

Import Validation

- Execute imports to validate boolean expression:

  ((a **and** b) **and** (c **or** (d **and** e))) **or** (f **and** g)



Detected Run-time Environment Conflicts

- All Libraries
  - 8,282 packages and 338,069 releases on PyPI Platform.
- Installed Libraries (pass the Installation Check)
  - 7,830 (95%) packages and 303,377 (90%) releases.
- Validated Libraries (pass all checks)
  - 5,371 (65%) packages and 131,720 (39%) releases.

Detected Run-time Environment Conflicts

- Incomplete Configuration
  - Missing configuration files – 281
  - Missing required libraries for setup – 3,318
  - **Missing Python versions – 55,138 (16%)**
  - **Missing required libraries for direct imports – 142,521 (42%)**

Finding: Developers tend to provide **inadequate** configurations for the usage of libraries, especially for **Python versions and direct imports in source code**.

Detected Run-time Environment Conflicts

- Incorrect Configuration
  - Dependency conflicts in setup – 6,318
  - Incorrect Python versions – 4,155
  - Other run-time errors in setup – 3,464
  - Inconsistent configurations with metadata – 592
  - Inconsistent version numbers with release dates – 12,018
  - **Missing required modules for indirect imports – 11,023**
  - **Inconsistent modules in direct imports with installed dependencies – 6,678**
  - **Other run-time errors in imports - 8,178**

Finding: Developers make mistakes in writing configurations since **19% of configuration issues are incorrect configurations**. What's more, about 50% of incorrect configuration issues can only be detected by Import Validation, indicating the **importance of source-level validation**.

Detected Run-time Environment Conflicts

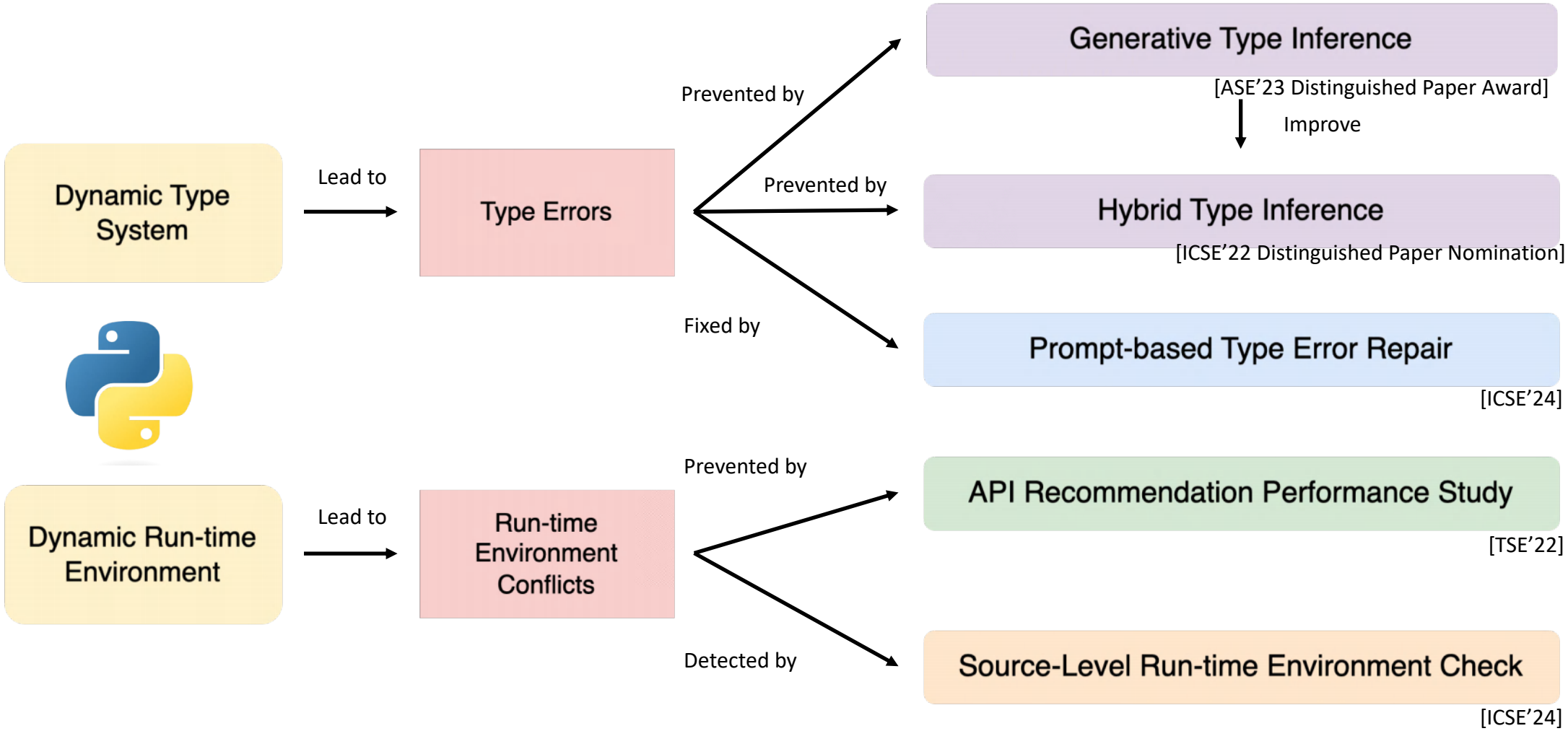
- Incorrect Code
  - Missing source code – 2,588
  - Parsing error – 431
  - **Multiple version control failure - 15,507 (5%)**

Finding: Incorrect configurations **can hardly be handled** by the multiple version control logic in source code, as there are 5% of library releases suffering from import block failures.

FOUR

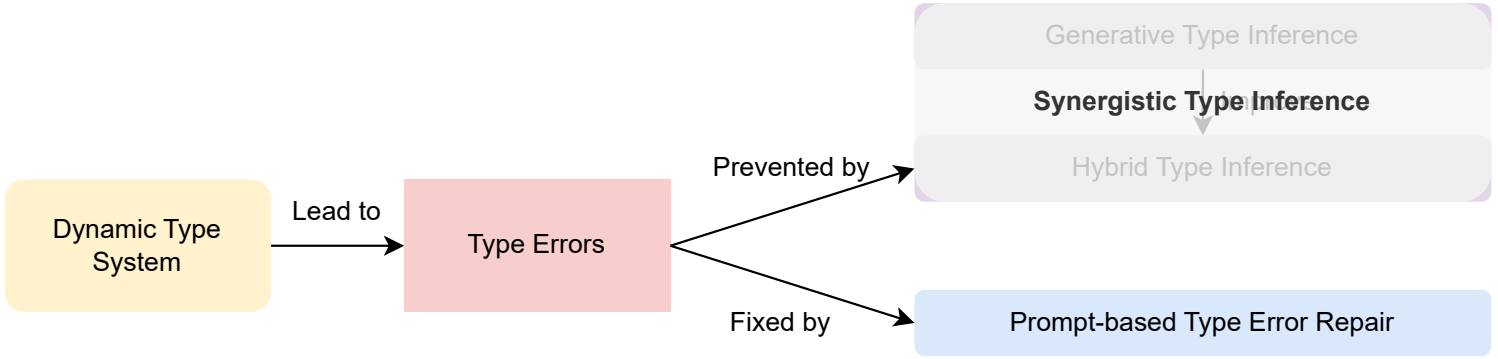
Conclusion & Future Work

Conclusion



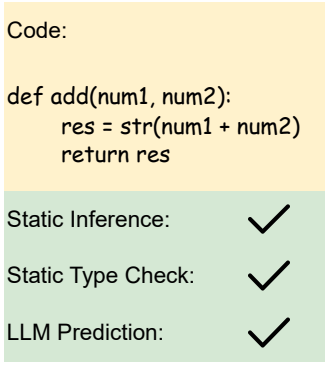
Only first-author publications are listed.

Future Work

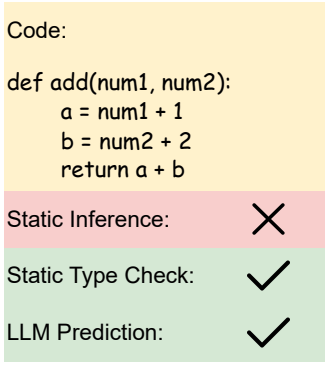


Synergistic Type Inference:

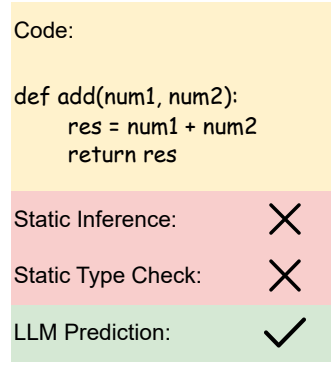
- LLMs and static inference handle different kinds of variables
- LLMs get instant feedbacks from type checkers and improve the predictions
- Single-variable inference → multiple-variable inference



(1) Static Inferred

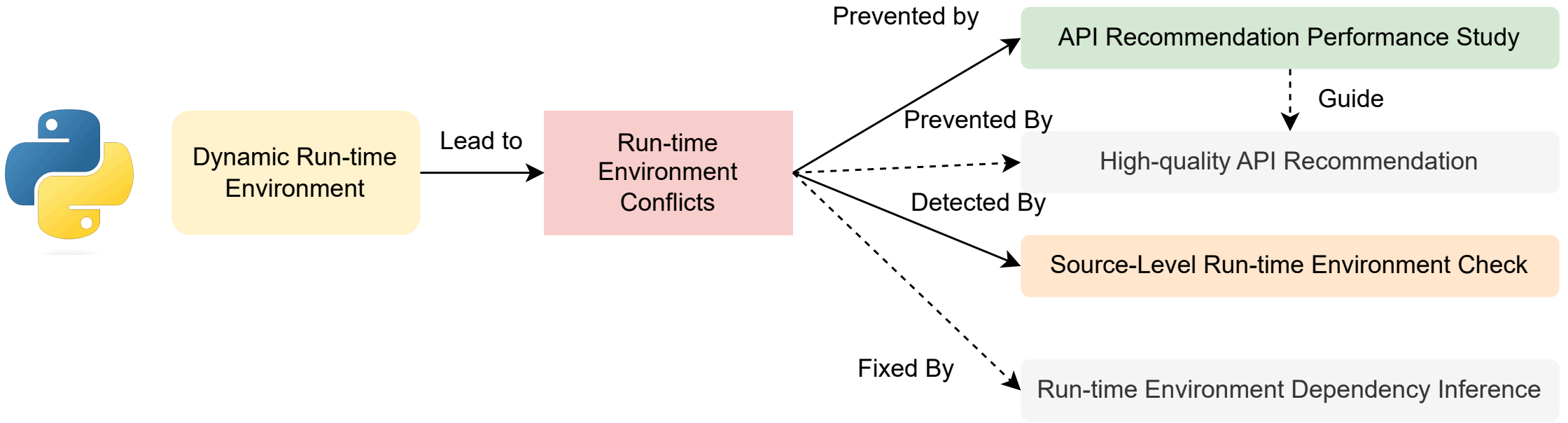


(2) Context Sensitive



(3) Context Insensitive

Future Work



High-quality API Recommendation:

- Make use of query reformulation techniques.
- Train the deep learning model on multiple-domain data.

Run-time Environment Dependency Inference:

- Infer correct configurations of run-time environments based on the source code of software.

List of Publications

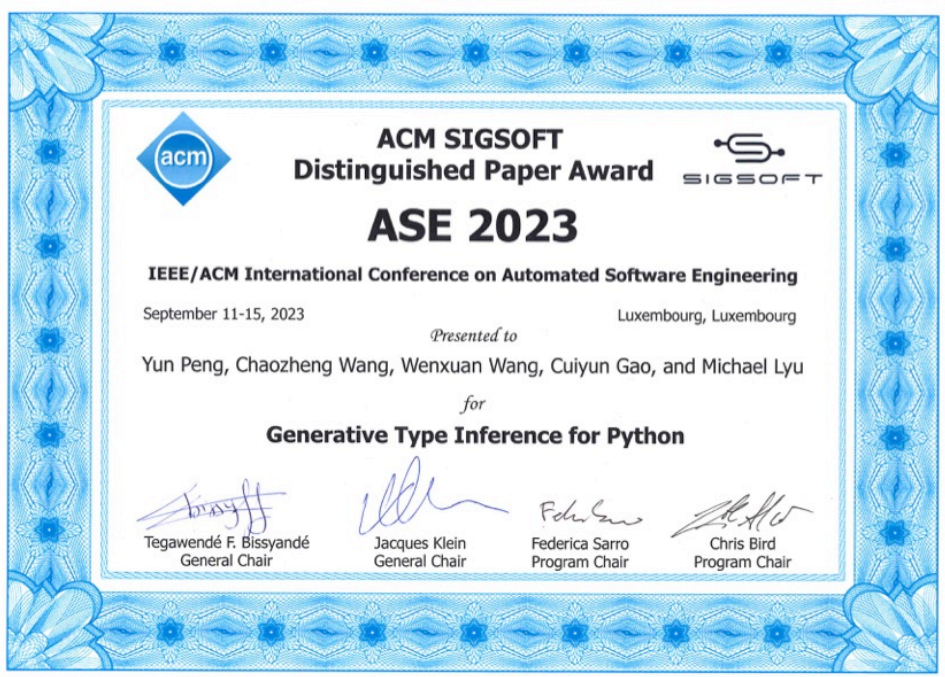
- **[ICSE'24] Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem.**
- **[ICSE'24] Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors.**
- [FSE'24] Less Cybersickness, Please: Demystifying and Detecting Stereoscopic Visual Inconsistencies in Virtual Reality Apps.
- [LLM4Code] Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context.
- **[ASE'23] Generative Type Inference for Python.**
- [ASE'23 Industry Challenge] REEF: A Framework for Collecting Real-World Vulnerabilities and Fixes.

List of Publications

- [TSE'23] Prompt Tuning in Code Intelligence: An Experimental Evaluation.
- [TSE'23] API Usage Recommendation via Multi-View Heterogeneous Graph Representation Learning.
- **[TSE'22] Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We?**
- [FSE'22] No More Fine-tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence.
- **[ICSE'22] Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python.**

Awards

- ACM SIGSOFT Distinguished Paper Award (ASE'23).
- ACM SIGSOFT Distinguished Paper Award Nomination (ICSE'22).
- Distinguished Paper Award of Industry Challenge Track (ASE'23).



☆ Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, Michael Lyu

NOMINATED FOR DISTINGUISHED PAPER

DOI Pre-print Media Attached

Thank you!

ARISE
Automated Reliable Intelligent Software Engineering

香港中文大學
The Chinese University of Hong Kong