Evaluating the Impact of Possible Dependencies on Architecture-level Maintainability

Wuxia Jin*, Dinghong Zhong,Yuanfang Cai, *Member, IEEE,* Rick Kazman, *Senior Member, IEEE*, Ting Liu

Abstract—Dependencies among software entities are the foundation for much of the research on software architecture analysis and architecture analysis tools. Dynamically typed languages, such as Python, JavaScript and Ruby, tolerate the lack of explicit type references, making certain dependencies indiscernible by a purely syntactic analysis of source code. We call these *possible dependencies*, in contrast with the *explicit dependencies* that are directly manifested in source code. We find that existing architecture analysis tools have not taken possible dependencies into consideration. An important question therefore is: *to what extent will these missing possible dependencies impact architecture analysis?* To answer this question, we conducted a study of 499 open-source Python projects, employing type inference techniques and type hint practices to discern possible dependencies. We investigated the consequences of possible dependencies in three software maintenance contexts, including capturing co-change relations recorded in revision history, measuring architectural maintainability, and detecting architecture anti-patterns that violate design principles and impact maintainability. Our study revealed that the impact of possible dependencies on architecture-level maintainability is substantial—higher than that of explicit dependencies. Our findings suggest that architecture analysis and tools should take into account, assess, and highlight the impacts of possible dependencies caused by dynamic typing.

Index Terms—dynamic typing, possible dependency, software architecture, empirical study.

1 Introduction

Dependencies among source code entities are the foundation for many software architecture analyses, including architecture recovery [1][2][3], architectural metrics [4][5], architectural problem or anti-pattern detection [6][7][8], change impact analysis [9], defect prediction [10][11], etc. Many language features may cause syntactic dependencies to be invisible in source code, such as polymorphism, casting, and reflection. This problem is more prominent due to dynamic typing in popular dynamic languages, such as Python, JavaScript, and Ruby. Dynamic typing [12] tolerates the lack of type information, making certain syntactic dependencies invisible in source code. In our research we call the "invisible" syntactic dependencies **possible dependencies** to contrast them with **explicit dependencies** that are directly manifested in source code.

We checked multiple popular commercial architecture analysis tools, such as Lattix Architect [13], Structure101 [14], Understand [15] and DV8 [16], and found that these tools rarely if ever extract possible dependencies when analyzing systems written in dynamic languages. To the best of our knowledge, dependency-based software architecture analysis has almost never taken possible dependencies into consideration. **Thus our motivating research question is: to what extent will these missing possible dependencies impact software architecture analysis**? It is important to explore this question to advance our understanding of possible dependencies, thus better analyzing the architectures of systems written in dynamic languages.

We use a simple example to illustrate the problem and hence our motivation. We implemented two versions of a small system applying a visitor pattern [17] using Java and Python respectively, and then reverse-engineered dependencies among the source files using Understand. We represent these dependencies in two design structure matrix (DSM), as shown in Figure 1. In each DSM, rows and columns denote source files listed in the same order. The numbers along the diagonal indicate self-dependency. Cell (i, j) with "$\times$" denotes that the file in row i syntactically and explicitly depends on the file in column j, as reported by Understand. For example, the "$\times$" in row 2, column 1 of Figure 1(b) means that *Body.py* depends on *CarElement.py*.

In Figure 1(a), *CarElement* is the base class of the concrete element classes listed in rows 2, 3, 4, and 6. Since Java forces explicit type declaration, the visitor interface, *CarElementVisitor*, has to explicitly refer to the four concrete element classes, i.e., the sub-classes of *CarElement*, hence the dependencies in row 5. Similarly, since each concrete element class has to accept a visitor object, they all have to refer to *CarElementVisitor*, hence the dependencies in column 5, forming a dependency cycle highlighted by the blue shading. By contrast, in Figure 1(b) all these dependencies and blue cycle are "invisible" to Understand: *CarElementVisitor.py* takes the same role as *CarElementVisitor.java*, but it doesn't explicitly depend on any of the car elements, and vice versa. Based on the dependencies extracted by

- * *W. Jin is the corresponding author, and with the School of Software Engineering, Xi'an Jiaotong University, China. Email: jinwuxia@mail.xjtu.edu.cn*
- *W. Jin, D. Zhong and T. Liu are with Ministry of Education Key Laboratory of Intelligent Networks and Network Security (MOEKLINNS), Xi'an Jiaotong University, China. Email: zhongdh@stu.xjtu.edu.cn, tingliu@mail.xjtu.edu.cn.*
- *Y. Cai is with the Department of Computer Science, Drexel University, USA. Email: yfcai@cs.drexel.edu*
- *R. Kazman is with the Department of Information Technology Management, University of Hawaii, USA. Email: kazman@hawaii.edu*

Manuscript received X, 2021; revised X, X.

		1	2	3	4	5	6	7	8
1	CarElement.java	(1)				X			
2	Body.java	X	(2)			X			
3	Wheel.java	X		(3)		X			
4	Engine.java	X			(4)	X			
5	**CarElementVisitor.java**		X	X	X	(5)	X		
6	Car.java	X	X	X	X	X	(6)		
7	CarElementPrintVisitor.java		X	X	X	X	X	(7)	
8	CarElementDoVisitor.java		X	X	X	X	X		(8)

(a) The DSM recovered from Java code

		1	2	3	4	5	6	7	8
1	CarElement.py	(1)							
2	Body.py	X	(2)						
3	Wheel.py	X		(3)					
4	Engine.py	X			(4)				
5	**CarElementVisitor.py**					(5)			
6	Car.py	X	X	X	X		(6)		
7	CarElementDoVisitor.py					X		(7)	
8	CarElementPrintVisitor.py					X			(8)

(b) The DSM recovered from Python code

Fig. 1. Explicit dependencies among files in the Java version and Python version

Understand, it appears that the practice of dynamic typing decouples software entities (only 9 dependencies in Python version whereas 26 dependencies in Java one), making the Python implementation appears to be better modularized and easier to maintain than the Java one.

We suspect that this is misleading since the Python and Java version implemented exactly the same pattern with exactly the same functionality. These "invisible" dependencies in Python may still have consequences. For example, if the methods defined in *CarElementVisitor.py* change, other program elements using these methods may have to change accordingly and vice versa, imposing a nontrivial architectural impact.

To quantitatively assess impacts of possible dependencies on architecture-level maintainability, we considered three ways of capturing measuring maintenance difficulty: capturing co-changed files [18][19], measuring architectural maintainability [4][5], and detecting architectural anti-patterns [8][20]. To explore these three measures, we conducted an empirical study using 499 open-source Python projects (41 million SLOC) collected from Github, with diverse sizes and domains. We extended ENRE [21] to extract possible dependencies from source code, employing existing type inference techniques [22][23][24][25] and type hint practices [26]. We extracted file-level explicit and possible dependencies, and mined 1,311,620 change-related and 277,381 issue-related revision commits. Based on this dataset[1], we investigated three research questions (RQs):

RQ1: To what extent will the ability to capture co-change relations be altered when considering possible dependencies?

RQ2: To what extent will possible dependencies impact architectural maintainability scores?

RQ3: To what extent will possible dependencies impact architectural anti-pattern detection?

Our results reveal that: (1) adding possible dependencies significantly improves the precision, recall, and F1 scores of capturing co-change relations;(2) on average, a file involved in possible dependencies requires 30% more maintenance effort than a file involved in explicit dependencies; (3) considering possible dependencies helps reveal previously indiscernible architectural anti-pattern instances, and detect severely problematic files with architectural connections. What is more, among files involved in architectural anti-patterns, those files that have both explicit and possible dependencies should be given higher priority in maintenance activities since they impact change-proneness and bug-proneness the most.

Our study provides strong evidence that **the level of architectural maintainability impact imposed by these possible dependencies is nontrivial, and in fact surprisingly high compared with explicit dependencies**. But the "invisible" nature of possible dependencies makes it more difficult to find them, to understand them, and to change them. Our results thus benefit architecture analysis and tools, providing concrete evidence that they should assess and emphasize the impact of possible dependencies, due to dynamic typing, on architecture-level maintainability .

In summary, this work makes the following contributions:

(1) We contribute a prototype tool that employs type inference techniques and type hint practices to infer possible dependencies in Python code.

(2) We present an empirical study on possible dependencies, revealing the non-trivial impact imposed by possible dependencies on architecture-level maintainability.

(3) We illustrate the benefits of analyzing possible dependencies on architecture analysis of dynamic languages, suggesting how such dependencies can promote co-change prediction, architecture management, and tool enhancement.

(4) We provide a benchmark collected from 499 open source projects for continued research on possible dependencies due to dynamic typing in Python software projects.

This paper is an extended version of our conference paper [27]. We extend our previous work in several ways. First, we improved the possible dependency extraction by employing type hint practices and we formally illustrated our extraction approach. Second, we extended the Python subjects studied—from 105 projects in our original study to 499 projects in the current, more comprehensive study. Third, we re-organized our research questions with a focus on the impact of possible dependencies on architecture-level maintainability, added an investigation of the impact on architecture anti-patterns, and revealed new findings. Fourth, we extended the benchmarks used in dependency verification and co-change capture. Lastly, we improved the work throughout including the key concept definition, study design, potential impact of our study, related work, and threats to validity.

In the rest of this paper: Sections 2 presents the possible dependency extraction approach. Sections 3, 4, and 5 report the study setup, study results, and potential impact of our empirical study. Sections 6 and 7 discuss the threats to validity and related work. Section 8 draws conclusions.

1. https://github.com/xjtu-enre/2022TSEPossibledepData

2 METHODOLOGY

This section illustrates the concepts and the approach proposed in our possible dependency extraction tool, using the Python example shown in Figure 2.

2.1 Key Concepts

The key concepts used in this paper include *entity*, *explicit dependency*, and *possible dependency*.

```
1  import json
2  class ClassE:
3    def m(self):
4      json.dumps("...")
5
6  def f1():
7    obj = ClassE()
8    obj.m()
9
10 class ClassA:
11   def m1(self):
12     print("A.m1")
13   def m2(self):
14     print("A.m2")
15 class ClassB(ClassA):
16   def m1(self):
17     print("B.m1")
18   def m2(self):
19     print("B.m2")
20
21 class ClassC:
22   def m1(self):
23     print("C.m1")
24
25 def f4(r1,r2):
26   r1.m1()
27   r1.m2()
28   r2.m()
```

Fig. 2. A snippet of *test.py*

Entity. A code entity e is an object with a given name or identifier. An entity can be a variable, function, class, module, etc. In Figure 2, entities include *ClassE*, *ClassE.m()*, *f1()*, *obj*, etc.

Dependency. According to Baldwin and Clark's definition [28], e_x depends on e_y if *changes* to e_y may cause e_x to change. A dependency is denoted as $e_x \rightarrow e_y$. A dependency can be a *syntactic dependency* extracted from source code, a *semantic dependency* extracted from the textual information of source code, or a *historical dependency* as recorded in the revision history [29][30][31][32].

Our work focus on syntactic dependencies. Syntactic dependencies are extracted in a static analysis based on an intermediate representation (i.e., AST) of the source code. The use of dynamic language features incurs some syntactically invisible and non-deterministic dependencies that only can be determined at run-time; we call these *possible dependencies*. Considering the inherent non-determinism, we identify two types of **syntactic dependencies** in dynamic languages like Python:

a. Explicit dependency. We define *explicit dependency* as the syntactic dependency relations that are explicitly manifested in source code. An explicit dependency $e_x \rightarrow e_y$ satisfies one of the following conditions: *1)* The depended entity e_y or its parent entity is explicitly imported; *2)* The type of e_y or its parent entity is declared explicitly in its visible scope. For example, Figure 2 shows that $m() \rightarrow json.dumps()$ (Line 4), $f1() \rightarrow ClassE()$ (Line 7), and $f1() \rightarrow ClassE.m()$ (Line 8) are explicit dependencies.

b. Possible dependency. We define *possible* dependency as a syntactic dependency relation that is *not* explicitly manifested due to the use of dynamic features. The targets of *possible dependencies* may be resolved into more than one type, and such dependencies can, in general, only be determined at run-time. We denote a *possible dependency* as $P_i = e_x \rightarrow e_y$, where i is the number of possible types which e_y can be resolved to. In particular, P_1 means entity e_y can only be resolved into one type. $P_{i>1}$ means that the dependency is non-deterministic: for e_y, there are i candidate types. In Figure 2, $f4() \rightarrow r2.m()$ (Line 28) is a P_1 dependency since $r2$ can only be $ClassE$. $f4() \rightarrow r1.m1()$ (Line 26) is a P_2 dependency because $r1$ has two possible types, $ClassA$ and $ClassB$.

The Scope of Our Work. As aforementioned, possible dependencies are related to the usage of dynamic features. Dynamic features in Python include dynamic typing which permits the omission of type declarations, introspection which examines the state of an object at run-time, object changes that can update or change an object at run-time, code generation that executes code at run-time and so on [33][34]. This work will consider possible dependencies due to dynamic typing manifested as a lack of type declarations. Our work is the first step towards shedding a light on the architectural impacts of possible dependencies.

2.2 The Framework Overview

To support our study on the architectural impact of possible dependencies, we design an approach to extract possible dependencies by incorporating the concept of P_i possible dependencies, the type inference technique [35], and type hint practices[2] into an extension of our previous tool named ENRE [21]. The prototype tool that supports extracting possible dependencies is publicly available [3].

Our possible dependency extraction approach assumes that possible dependencies could be extracted from type hints since possible dependencies are due to the lack of type information. Based on this assumption, our approach uses attribute constraints [35] to infer objects' types and thus possible dependencies. In addition, our approach directly extracts type annotations from stub files if they are provided in a Python project. In type hint practice, a Python stub file (with name $*.pyi$) contains only type annotations with empty function bodies. Such type annotations in a stub file are usually maintained by developers for a corresponding source file. Using *django* (version:stable/2.2.x)[4] as an example, Figure 4(b) shows *decorators.py* in this project, and Figure 4(a) shows the corresponding stub file (i.e., *decorators.pyi*) managed in *django-stubs*[5]. We can see that the stub file specifies the type information for function *register*: the type of *models* is *Type[Model]*, the type of *site* is *Optional[Any]*, and *register* returns an object of *Callable* type.

Figure 3 shows our possible dependency extraction framework employing ENRE, merge-pyi (part of the Pytype[6] tools), and Mypy[7], shown in blue rectangles. Both *Pytype merge-pyi* and *Mypy* can analyze Python code with type hints practice, thus facilitating possible dependency extraction. *Pytype merge-pyi* can merge Python source code and its corresponding stub files which contain type annotations, and *Mypy* infers types from python type-annotated code. The input of this framework is source code, and the output

2. https://www.python.org/dev/peps/pep-0484/
3. https://github.com/xjtu-enre/ENRE-go-python
4. https://github.com/django/django
5. https://github.com/typeddjango/django-stubs/
6. https://github.com/google/pytype
7. http://www.mypy-lang.org

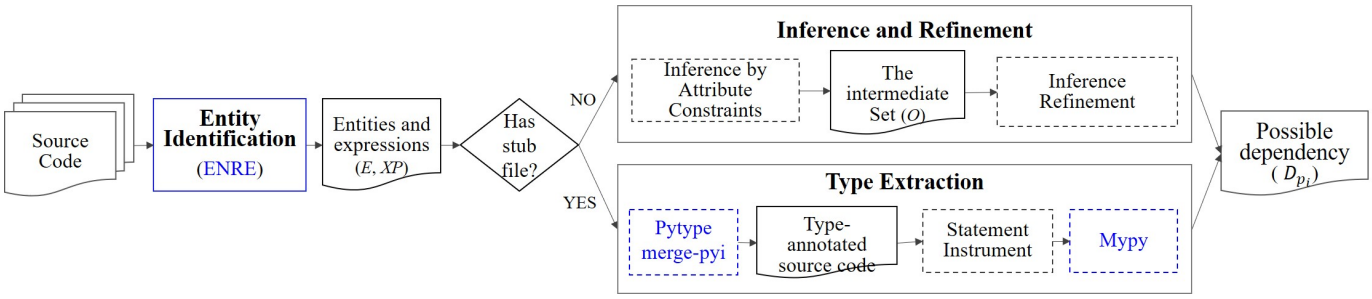


Fig. 3. The possible dependency extraction framework

is a set of possible dependencies, D_{P_i} where P_i denotes the type of possible dependency. This framework contains three components:

1) Entity Identification. This component, provided by ENRE, carries out lexical and syntactic analysis of source code, resolving the code entities $E = \{e\}$ and the expression set $XP = \{xp\}$. An entity is defined using a tuple of attributes, including id,$qualifiedName$,$shortName$, etc. Each *expression*, xp is denoted as ($pythonfile$, $linenumber$,exp, $fromname$, $toname$) that implies possible dependencies.

For example, in Figure 2, Line 26 in *test.py* contains one expression, $xp_1 = (test.py, 26, r1.m1(), f4, m1)$. It indicates one unresolved dependency from $f4$ to $m1$, i.e., $f4 \rightarrow m1$ at Line 26 in $test.py$, and the type of $r1$ has not been resolved.

2) Inference and Refinement. This component processes an expression, $xp \in XP$, if its source code file has no corresponding stub file that contains type hints introduced by PEP484. Following the attribute constraints proposed by Xu et al. [35], this component produces the intermediate structure O that indicates all of the possible dependencies. Then it refines O by the Refinement Inference with different rules. Finally, it extracts a set of possible dependencies, D_{P_i}, from the refined O.

3) Type Extraction. For an expression xp, this component deals with source code that has a corresponding stub file. This component first uses *Pytype merge-pyi* to merge the source code file and the corresponding stub file, producing type-annotated source code. After statically instrumenting xp by adding *reveal_type()* statements into the type-annotated code, this component leverages *Mypy* to extract types of entities contained in xp. Based on xp and the obtained type information, it generates D_{P_i}.

We will introduce Inference and Refinement, and Type Extraction in the following two sections. The notation used in the algorithms is described in Table 1.

TABLE 1
Basic notation

Symbol	Description
e	A code *Entity*.
$e_x \rightarrow e_y$	A *Dependency* from e_x to e_y.
xp	An *Expression* in code.
a	An *Atomic Expression* extracted from xp.
s, $s \in E$	The visible *Scope* of xp or a.
T	A hierarchical structure from xp.
C_a	A set of candidate types of a.
$E = \{e\}$	A set of entities.
XP=$\{xp\}$	A set of expressions.
$O = \{(s, a, C_a)\}$	An intermediate set.
$D_{P_i} = \{e_x \rightarrow e_y\}$	A set of P_i possible dependencies.

2.3 Inference and Refinement

This section will illustrate possible dependency extraction without stub files using the Python example in Figure 2.

2.3.1 Inference by Attribute Constraint

Using E and XP as the inputs, this module follows attribute constraints [35] to infer all of the possible dependencies. This is based on duck typing, i.e., the type of an object is determined by how its attributes are used by others [36]. This module outputs an intermediate data structure, $O = \{(s,\ a,\ C_a)\}$, to record possible dependencies, where $s \in E$ is an entity that denotes code *scope*, a is an *atomic expression* extracted from xp, and C_a denotes the set of candidate types of a.

It first classifies all non-private members of class entities based on their signatures, and then generates f_{sig}, mapping each signature to a set of class members. For example, in Line 10-28 in Figure 2:

$f_{sig}(m1()) = \{ClassA.m1(), ClassB.m1(), ClassC.m1()\}$
$f_{sig}(m2()) = \{ClassA.m2(), ClassB.m2()\}$

For an expression, xp_i, in a scope s, this module first generates a hierarchical structure, T_i, by splitting the exp_i of xp_i into atomic expressions, and denotes their parent-child relations. For example, the result of splitting xp_1 is $T_1 =< a_1,\ a_2 >$, where $a_1 = r1$, $a_2 = r1.m1()$, and a_1 is the parent of a_2, forming a layered structure: if a_1 is in layer i, a_2 is in layer $i+1$.

Then this module resolves each a in T_i: if an atomic expression, a_j, is at the lowest layer of T_i, then its candidate types are denoted as $C_{a_j} = f_{sig}(getName(a_j))$, which returns all the types that match the signature of a_j. If a_j is not at the bottom layer, its C_{a_j} will be resolved as the parents of its children's candidate types.

Consider the above T_1 as an example. The candidate types of $a_2 = r1.m1()$ will first be resolved as $C_{r1.m1()} = C_{T_1(2)} = f_{sig}(m1()) = \{ClassA.m1(), ClassB.m1(), ClassC.m1()\}$. After that, the candidate types of its parent, $a_1 = r1$, will be resolved as $C_{r1} = C_{T_1(1)} = \{ClassA, ClassB, ClassC\}$. Other expressions are processed in the same way. After that, the following elements will be added to O:

① $(f4(), r1, C_{T_1(1)})$, where

$C_{T_1(1)} = \{ClassA, ClassB, ClassC\}$
② $(f4(), r1.m1(), C_{T_1(2)})$, where
$C_{T_1(2)} = \{ClassA.m1(), ClassB.m1(), ClassC.m1()\}$
③ $(f4(), r1, C_{T_2(1)})$, where $C_{T_2(1)} = \{ClassA, ClassB\}$
④ $(f4(), r1.m2(), C_{T_2(2)})$, where
$C_{T_2(2)} = \{ClassA.m2(), ClassB.m2()\}$
⑤ $(f4(), r2, C_{T_3(1)})$, where $C_{T_3(1)} = \{ClassE\}$
⑥ $(f4(), r2.m(), C_{T_3(2)})$, where $C_{T_3(2)} = \{ClassE.m()\}$

Now based on O, we infer the possible dependencies: $f4() \rightarrow r1$ and $f4() \rightarrow r1.m1()$ are P_3 possible dependencies as $|C_{r1}| = |C_{r1.m1()}| = 3$; $f4() \rightarrow r1$ and $f4() \rightarrow r1.m2()$ are P_2 possible dependencies; $f4() \rightarrow r2$ and $f4() \rightarrow r2.m()$ are P_1 possible dependencies.

2.3.2 Inference Refinement

This module refines the intermediate structure O, using the *Scoping Rule* and the *Priority Rule*. Next it extracts a set of possible dependencies, D_{P_i}, from the refined O.

Scoping Rule: Using this rule, this module first groups together atomic expressions within the same scope. In the above example, the elements ① and ③ in O should be grouped together since they indicate that $r1$ is within the scope of $f4()$, and their candidate types should be the same. Their candidate types are refined to be the intersection of their original C_a: $C = C_{T1(1)} \cap C_{T2(1)}$=$\{ClassA, ClassB\}$. Accordingly, the candidates of their children are also updated: $C_{T_1(2)} = \{ClassA.m1(), ClassB.m1()\}$ and $C_{T_2(2)} = \{ClassA.m2(), ClassB.m2()\}$.

Priority Rule: For a candidate type set containing more than one type, if one type inherits from another, we remove the sub-type from the candidate set. This rule follows the Liskov Substitution Principle [37]: if an entity depends on parent/abstract type, then any sub-types can be substituted with each other wherever the parent type is used.

In this example, within candidate set $\{ClassA, ClassB\}$, $ClassB$ inherits from $ClassA$. As result, $C_{T_1(1)}$ and $C_{T_2(1)}$ will be refined as $C_{T_1(1)} = C_{T_2(1)} = \{ClassA\}$, and their children will also be updated: $C_{T_1(2)} = \{ClassA.m1()\}$, and $C_{T_2(2)} = \{ClassA.m2()\}$.

At this point, the set O in this example is updated as follows:

$(\ f4(), r1, C_{T_1(1)} = \{ClassA\}\ )$
$(\ f4(), r1.m1(), C_{T_1(2)} = \{ClassA.m1()\}\ )$
$(\ f4(), r1, C_{T_2(1)} = \{ClassA\}\ )$
$(\ f4(), r1.m2(), C_{T_2(2)} = \{ClassA.m2()\}\ )$

We can now derive a set of P_1 possible dependencies from O: $D_{p_1} = \{f4() \rightarrow ClassA, f4() \rightarrow ClassA.m1(), f4() \rightarrow ClassA.m2(), f4() \rightarrow ClassE, f4() \rightarrow ClassE.m()\}$. We can see that these rules refine some non-deterministic dependencies P_n (less likely to occur at run-time) into P_1 dependencies (more likely to occur) at run-time.

2.4 Type Extraction

In addition to possible dependencies that can be extracted by the above Inference and Refinement component, possible dependencies can also be extracted from stub files that contain type hints (when such stub files are provided) as shown in the Type Extraction component in Figure 3. We introduce Type Extraction using the Python example shown in Figure 4.

1) *Pytype merge-pyi*. Given the source code (*.py) and the stub file (*.pyi) of a Python file, *Pytype merge-pyi* merges the two files and outputs type-annotated source code.

For an example, Figure 4(b) shows *decorators.py* in *django* (version:stable/2.2.x) project, and Figure 4(a) corresponds to its stub file named *decorators.pyi*). *merge-pyi* copies type annotations located in Figure 4(a) into the source code in Figure 4(b). The type-annotated code is shown in Figure 4(c) and the code lines prefixed with + are newly added.

2) Statement Instrument. This module visits the AST (Abstract Syntax Tree) of type-annotated source code to record locations of objects in the code. The data structure of an object location is ($filename$, $objectname$, $startLine$, $endLine$, $startColumn$, $endColumn$), where $filename$ is the file that contains the object, $objectname$ is the identifier of the object, $startLine$ and $endLine$ correspond to the line numbers of the first and last character of the $objectname$, and $startColumn$ and $endColumn$ correspond to the column numbers. Next, this module instruments a statement, *reveal_type(objectname)*, for each object according to the recorded position.

For the object named *models* in Figure 4(c), its location is denoted as ($django/contrib/admin/decorators.py$, $models$,5,5,15,20). In terms of the location, *reveal_type(models)* statement will be added as shown in Figure 4(d).

3) *Mypy*. Mypy analyzes the instrumented source code and outputs the set of types (i.e., $typeSet$) for an object. We used $typeSet$ since an object may be resolved as more than one type. Combining the identified $typeSet$ and the pre-recorded location of an object, we can now deduce ($filename$, $objectname$, $startLine$, $endLine$, $startColumn$, $endColumn$,$typeSet$) for each object.

Recalling *models* in Figure 4(d), *Mypy* identifies the $typeSet$ of *models* as $\{django.db.models.base.Model\}$. Combining $typeset$ and the data structure from *models*, we can get:

b=($django/contrib/admin/decorators.py$, $models$,5,5,15,20,$\{django.db.models.base.Model\}$)

Since *django.db.models.base.Model* is declared in *django/db/models/base.py*, the above b indicates one P_1 possible dependency at file level:

$django/contrib/admin/decorators.py \rightarrow$
$django/db/models/base.py$

3 Study Setup

This section will illustrate the research questions, subjects, the collected data, and evaluation methodology for our empirical study. Since $P_{i>1}$ dependencies are rare and undeterministic [27], we will form a conservative baseline using the union of explicit dependencies and P_1 possible dependencies in this study, which contains the minimal number of "deterministic" syntactic dependencies that should be taken into consideration.

3.1 Research Questions

To evaluate the impact of possible dependencies on architecture-level maintainability, we consider the three

```
#django-stubs/contrib/admin/decorators.pyi
from typing import Any, Callable, Optional, Type
from django.db.models.base import Model
def register(*models: Type[Model], site: Optional[Any] = ...) -> Callable: ...
```

(a) decorators.pyi (stub file)

```
#django/contrib/admin/decorators.py
def register(*models, site=None):
    def _model_admin_wrapper(admin_class):
        if not models:
            raise ValueError('At least one model must be passed to register.')
```

(b) decorators.py (source code)

```
#django/contrib/admin/decorators.py
+ from typing import Any, Callable, Optional, Type
+ from django.db.models.base import Model

+ def register(*models: Type[Model], site: Optional[Any] = ...) -> Callable
    def _model_admin_wrapper(admin_class):
        if not models:
            raise ValueError('At least one model must be passed to register.')
```

(c) decorators.py (type-annotated source code)

```
#django/contrib/admin/decorators.py
from typing import Any, Callable, Optional, Type
from django.db.models.base import Model

def register(*models: Type[Model], site: Optional[Any] = ...) -> Callable
  + reveal_type(models)
    def _model_admin_wrapper(admin_class):
        if not models:
            raise ValueError('At least one model must be passed to register.')
```

(d) decorators.py (source code ready for Mypy)

Fig. 4. An example of Type Extraction in Django

maintenance contexts mentioned in the introduction: capturing co-changed files, architectural maintainability measurements, and detecting architecture anti-patterns. Given these contexts we investigate three research questions (RQs):

RQ1: To what extent will the ability to capture co-change relations be altered when considering possible dependencies? Exploring this question will help understand if possible dependencies are a significant factor in causing co-changes when maintaining source files.

RQ2: To what extent will possible dependencies impact architectural maintainability scores? Answering this question will indicate how possible dependencies impact maintainability scores and to what extent they would cause additional maintenance costs, as compared with explicit dependencies.

RQ3: To what extent will possible dependencies impact architectural anti-pattern detection? This question will advance our understanding regarding how strongly anti-patterns that are detected when considering possible dependencies could impact bug-proneness and change-proneness of a software system. These possible dependencies could therefore reveal highly problematic files that should be given attention during refactoring and maintenance activities.

3.2 Subject Collection and Statistics

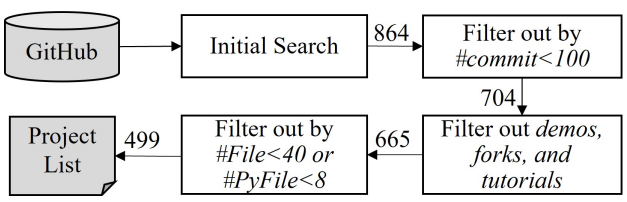


Fig. 5. Project collection and filtering

To minimize the possible bias caused by project domain or size, we queried and chose open-source Python projects sorted by their number of stars from Github using the Github REST API. Figure 5 depicts the project collection and filtering workflow. We initially collected 864 projects with their latest versions. Since our study required a well-managed revision history, we filtered out projects with fewer than 100 commits. We also removed the repeated or forked projects, and tutorial projects if their textual description contains the keywords *tutor|tutorial|course|exercise|guide|note|demo|example|sample*. After this operation, there were 665 projects remaining. We further excluded small projects where the number of files is smaller than 40 or the number of Python files is smaller than 8. In the end we collected 499 projects as the subjects of our study.

For each project we collected its revision history from the version control system Git. The revision history covers a time range from the beginning to the latest version of a project. We only considered the commits in which a source file is modified; commits that only added files are ignored since we are focused on co-change and maintenance activities. An issue-related commit is a commit labeled with an issue ID.

Table 2 summarizes the demographic information of the projects that we analyzed: #*File*—total number of files; #*LoC*—total lines of code; #*PyFile* and #*PyLoc*—Python files and Python LoC only; #*AvgLoCPerPyFile*—average LoC per Python file, i.e., $\frac{\#PyLoC}{\#PyFile}$; #*Commit*—total number of commits in the revision history; #*Committer*—total number of contributors; #*IssueCommit* and #*IssueCommitter*—issue-related commits and committers as recorded in the revision history. Figure 6 shows the violin plots with the distribution of these attributes. *Min*, *Median*, and *Max* in Table 2 correspond to the minimum, median, and maximum values over all projects.

Table 3 lists the domain[8] distribution of our collected projects. For example, the first row in Table 3 shows that 25.9% (129 out of 499) projects belong to the Software Development domain. We observe that our subjects cover diverse domains; the top 3 popular application domains include Software Development (25.9%), Web and Internet Development (25.3%), and Scientific and Numeric (21.4%).

8. https://legacy.python.org/about/apps/

TABLE 2
The investigated Python projects

#Subject	Statistic	#File	#LoC	#Commit	#IssueCommit	#Commiter	#IssueCommitter	#PyFile	#PyLoC	#AvgLoCPerPyFile
499	Min	41	473	100	0	1	0	8	473	9.27
	Medium	182	264,98	1,160	138	83	28	134	16,866	126.96
	Max	8,224	2,728,799	498,49	12,717	2,837	1,971	3636	774,125	3303.69
	Sum	224,642	41,126,375	1,311,620	277,381	82,079	40,412	132310	21,446,298	/

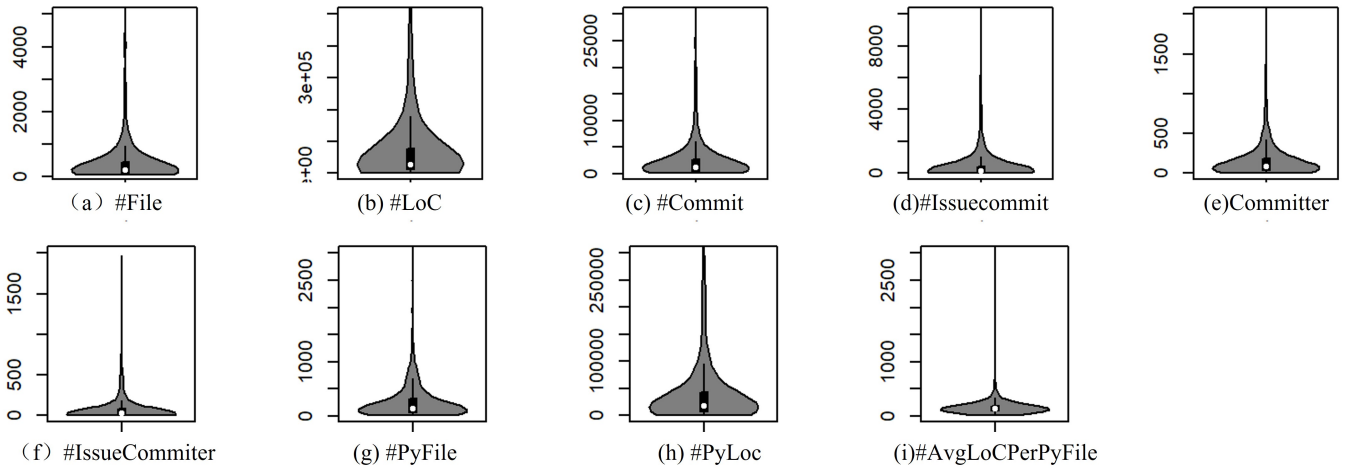


Fig. 6. The distribution of projects

TABLE 3
The distribution of project domain

Domain	Percentage
Software Development	25.9% (129)
Web and Internet Development	25.3% (126)
Scientific and Numeric	21.4% (107)
Desktop GUIs	5.6% (28)
Game and 3D Graphics	3.4% (17)
Database Access	2.8% (14)
Image Processing and Computer Vision	2.4% (12)
Network Programming	2.0% (10)
Education	1.6% (8)
Others	9.6% (48)

3.3 Dependency Extraction and Verification

This section will introduce dependency extraction, dependency verification, and the statistics of these dependencies extracted from the subjects.

3.3.1 Dependency Extraction

We used the Understand tool to extract explicit dependencies from the source code of each project. Understand can identify explicit dependencies such as method calls, variable references, and module or file importing. This tool has been widely used in both academic research [38][39][40] and industry [13][14].

We then used the possible dependency extraction approach in Section 2 to detect possible dependencies from source code.

3.3.2 Dependency Verification

We verified the possible dependencies by collecting benchmarks from execution traces. The process is as follows:

1) We used the possible dependency extraction approach in Section 2 to detect expressions that imply possible dependencies and to resolve these dependencies. The results are simply denoted as $D = \{d\}$ where d=($pythonfile$, $linenumber$, exp, $fromname$, $toname$, $typeSet$). For example in *docutils*[9] (v0.12), $d = (peps.py, 130, pep\_type = para.astext(), peps.py, astext, \{nodes\})$. This means that $pep\_type = para.astext()$ is an expression at line 130 in $peps.py$. $peps.py$ calls $para.astext()$, thus $fromname$ is $peps.py$ and $toname$ is $astext()$. $typeSet = \{nodes\}$ is the resolved result, meaning that $para$ is resolved as the type of $nodes$. This record d indicates one P_1 dependency is extracted, i.e., $peps.py \rightarrow nodes.astext()$ since $|typeSet| = 1$.

2) We collected dependencies from execution traces. We did this in two ways. We relied on MonkeyType[10] to execute test cases included in projects. MonkeyType, published by Instagram, uses the *sys.setprofile* hook provided by Python to interpose among function calls, dumping call traces into a SQLite database. In addition we executed test cases included in various projects and traced executions by a Python module called *trace*[11]. Employing MonkeyType and the *trace* module, we extracted the dependencies $T = \{t\}$ from these executions, where t=($pythonfile$, $linenumber$, exp, $fromname$, $toname$, $typeSet$), the same representation as d.

3) We generated P_1 and P_n possible dependency benchmarks by filtering T. For $t \in T$, if the $(filename, linenumber, exp)$ is equal to that of $d \in D$, add t into P_n $benchmarks$. Add t into P_1 $benchmarks$ if

9. https://github.com/docutils-mirror/docutils
10. https://github.com/instagram/monkeytype
11. https://docs.python.org/3.7/library/trace.html

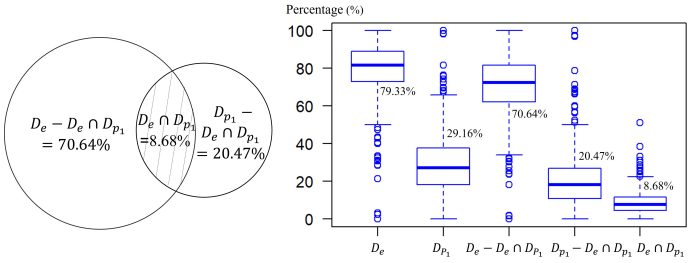
the $|typeSet| = 1$, i.e., $P_1 benchmarks \subseteq P_n benchmarks$. We used $(filename, linenumber, exp)$ for filtering instead of $(filename, exp)$ since some expressions with the same string and located in the same Python file may be located in different code lines or blocks, perhaps implying distinctive dependencies.

4) We evaluated the accuracy of possible dependencies. If the $typeSet$ of d is completely equal to or is the base-class set of $typeSet$ of t in the benchmarks, we considered that the possible dependency indicated by d is verified to be correct. Since the benchmarks were collected by filtering the execution traces against the expressions that imply possible dependencies, the value of the recall is equal to that of the precision. Thus we used P_1 $accuracy$ (or P_n $accuracy$) to denote the precision or recall against P_1 $benchmarks$ (or P_n $benchmarks$).

We selected 8 projects for detailed evaluation. As summarized in Table 4, 1217 P_1 $benchmarks$ and 1868 P_n $benchmarks$ were collected in total. Against 1217 P_1 $benchmarks$, 1184 P_1 dependencies were resolved correctly, i.e., P_1 $accuracy$ is $1184/1217 = 0.97$. P_n $accuracy$ is 0.86 against P_n $benchmarks$. The incorrect results are sometimes caused by the possible dependencies between source code and Python libraries, which our tool is unable to detect. P_n $accuracy$ is lower than P_1 $accuracy$ since it is difficult for execution traces to cover all possible dependencies. Thus, we formed a conservative baseline using the union of explicit dependencies and P_1 possible dependencies in the following study.

3.3.3 Statistics of the Collected Dependencies

We summarized the P_1 possible dependencies and explicit dependencies extracted from the 499 studied projects. The dependencies extracted among methods or classes can be aggregated into dependencies among files. Since many architecture-level analyses use file pairs as atomic units of analysis, we will observe the distribution of file-level P_1 possible dependencies.



(a) The set of dependencies (b) The boxplots of dependencies

Fig. 7. The distribution of file-level dependencies

Figure 7(a) depicts the aggregated file-level P_1 possible dependencies, D_{P_1}, and how they overlap with explicit dependencies, D_e, averaged over all 499 projects. Figure 7(b) depicts the ranges of their percentages and overlaps, labeled with the mean values of all percentage scores. It shows that, of all the baseline dependencies containing $D_e \cup D_{P_1}$, 20.47% of them only have possible dependencies, i.e., $D_{P_1} - D_e \cap D_{P_1}$.

The results show that file-level P_1 possible dependencies (D_{P_1}) represent a significant portion (20.47% on average) of all uniquely resolvable file-level dependencies($D_e \cup D_{P_1}$). For the following study, we use $D_e \cup D_{P_1}$ as a baseline since many architecture-level analyses use file pairs as atomic units of analysis.

3.4 Evaluation Setup

Based on the subjects with their revision history and dependencies, we investigated the impact of possible dependencies on architecture-level maintainability by studying the three research questions in Section 3.1. That is, we evaluate the impact of possible dependencies on historical co-change capturing (RQ1), the impact on architectural maintainability measurements (RQ2), and the impact on architectural anti-pattern detection (RQ3).

3.4.1 The Setup for RQ1

Much prior research has leveraged revision history, focusing on how files are committed together, as a benchmark for understanding co-change relations [41], change impact analysis [42][43], and identifying problematic modules [44][5]. The underlying assumption is that related files are more likely to be changed together. The question in the context of our research is: *to what extent will the ability to capture co-change relations be altered when considering possible dependencies?* If this ability is significantly improved, it would indicate that possible dependencies are an important factor causing co-changes. The evaluation process for RQ1 is as follows:

1) Benchmark Generation.

We collected file-level co-change relations as benchmarks from every project's revision history. As defined in the work of [30][45], co-change coupling is a measure of the degree to which two entities co-evolve or change together, based on their evolution history. To mitigate possible bias in our study, we enriched the co-change relation collection process and obtained two kinds of benchmarks:

Co-change Benchmark A. According to prior work [40][46], we consider a pair of files having a co-change relation if they were committed at the same time. Prior work often used arbitrary thresholds to determine "meaningful" co-change relations [40], while it is unclear if a particular threshold is generalizable. To conduct a rigorous study, we assembled 20 co-change benchmarks (using 20 different threshold settings), $Benchmark_i$, $i = 1, 2, ..., 20$, from the revision history of each project, each containing a set of file pairs that have changed together at least i times. The larger the i, the more times two files were committed together, and hence the stronger their relationship. From $Benchmark_1$ to $Benchmark_{20}$, the co-change relations become less random and more meaningful.

Co-change Benchmark B. Existing studies [47][48][49] have argued that co-changed files were not only those committed at the same time but also co-changed files could be committed during a short time interval. For instance, if a change to $file_1$ is typically followed by a change to $file_2$, the two files would be considered as co-change files. Based on this assumption, we followed the work of [49] to collect co-change benchmarks by setting two parameters, *time distance* and *commit distance*. *time distance* is the maximal duration where files in commits within this time restriction

TABLE 4
The Benchmarks and verification results

Project	P_1 *benchmarks*	P_1 *correct*	P_1 *accuracy*	P_n *benchmarks*	P_n *correct*	P_n *accuracy*
glance	15	12	0.80	18	12	0.67
docutils	375	359	0.96	513	360	0.71
pythonvisitor	12	12	1.00	12	12	1.00
python-patterns	17	17	1.00	21	17	0.81
algorithms	14	13	0.93	15	14	0.93
pygorithm	18	18	1.00	48	48	1.00
mypy	217	209	0.96	681	597	0.88
mimesis	549	544	0.99	560	555	0.99
summary	1,217	1,184	0.97	1,868	1,615	0.86

TABLE 5
Co-change Benchmark A and Benchmark B

Benchmark A				
1	2	3	4	5
23346372	10799168	6590282	4404330	3027070
6	7	8	9	10
2158212	1464692	1069782	808434	656658
11	12	13	14	15
517092	438404	379060	325922	270842
16	17	18	19	20
227164	197722	165742	132144	104234
Benchmark B				
67778				

will be considered as co-changing. *commit distance*, is the minimal number of common commits where $file_1$ and $file_2$ were revised within the time constraint. When setting *time distance*=0 and *commit distance*=1, the generated benchmarks are same as the $Benchmark_1$ in Benchmark A. As suggested by Bird et al. [50], for each project, we configured the threshold of *time distance* as the third quartile value in the distribution of time intervals between continuous commits and set the threshold of *commit distance* as 20.

Benchmark Summary. For all benchmarks, we used 301 of the 499 projects as subjects because we can obtain all the required benchmark data from their revision histories. The other 198 projects were excluded as several benchmarks were unavailable given the rigorous parameter settings. The collected co-change relation benchmarks are summarized in Table 5. For example, the total number of co-change file pairs is 23,346,372 in $Benchmark_1$ of Benchmark A, and the number for Benchmark B is 67,778. It is obvious that benchmark size becomes smaller with the larger i of $Benchmark_i$ contained in Benchmark A.

2) Co-change Capturing. We measured the ability of a set of dependencies, D, to reflect co-change relations in the benchmarks using *Precision*, *Recall*, and F_1 scores:

Precision ($P(D)$)—the percentage of file pairs in the dependency set that are also in the co-change benchmark;

Recall ($R(D)$)—the percentage of file pairs within the co-change benchmark that are captured by the dependency set;

F_1 score ($F_1(D)$)—the harmonic average of the precision and recall.

The larger the $P(D)$, $R(D)$ and $F_1(D)$ values, the stronger the potential of D to reflect co-change relations. We thus computed the precision, recall, and F_1 scores of D_e and $D_e \cup D_{P_1}$, for each project against each of the 20 benchmarks. If the scores using $D_e \cup D_{P_1}$ is larger than those using D_e, we could conclude that considering possible dependencies can improve the potential of capturing co-change relations. Figure 8, Figure 9, and Table 7 show the measurement results.

3.4.2 The Setup for RQ2

We consider if maintainability measures are improved after considering possible dependencies and assess if these changes allow us to better reveal the true maintainability level of a project. If the answer is positive, we will try to figure out why possible dependencies make such a contribution. The evaluation process is as follows:

1) Maintainability Measurement. Using the Decoupling Level (DL) and Propagation Cost (PC) metrics [4][5] as two state-of-the-art architectural maintainability metrics, we observe how a project's scores change based on D_e and $D_e \cup D_{P_1}$. Both DL and PC are calculated based on file-level dependencies. DL assesses how well files are decoupled into independent modules [5]. PC measures how tightly coupled a system is by calculating how changes may propagate through direct and indirect dependencies among files [4]. The higher the DL (or the lower the PC), the better the architectural maintainability [5]. Table 8 lists the results.

2) Correlation Calculation. We next consider which DL/PC scores better reflect the actual maintainability level of a project: scores based on D_e, or scores based on $D_e \cup D_{P_1}$, as shown in Table 8. Following the work of Mo et al. [5], we assessed which set of DL and PC scores are better correlated with actual maintenance effort, as reflected by six measures mined from each project's revision history. If the scores based on the consideration of possible dependencies are more strongly correlated with the ground-truth based on revision history, we assume that these scores more accurately reflect the true maintainability level of the projects.

Using a project's history as its ground-truth, we used the six maintainability measures from prior work [5]:

Change/Bug Commit Overlap Ratio (i.e., CCOR and BCOR)—the extent to which a file is modified by different bug-fixing or change commits;

Change/Bug Commit Fileset Overlap Ratio (CCFOR and BCFOR)—the extent to which a file is committed by different committers;

Change/Bug Pairwise Committer Overlap (CPCO and BPCO)—the likelihood that two committers need to communicate based on whether they committed changes to the same set of files.

For these measures of a project to be reliable, it requires that the code repository is well managed—with bug issues

and change issues properly tagged——and with a sufficient amount of history. Based on these criteria we conducted correlation analysis between dependency-based DL (and PC) scores and the history-based maintainability ground-truth. Since the score distribution presents outliers, we used the Spearman correlation coefficient—which is robust with respect to outliers and is a non-parametric version of the Pearson correlation coefficient—to analyze the correlation [51]. The stronger the correlation, the better these scores reflect a project's true maintainability. Table 9 lists the correlation analysis results, indicating that the consideration of possible dependencies better reflects the ground-truth of maintainability levels, as we will discuss in Section 4.2.2.

3) Potential Cause Analysis. We further investigate why the maintainability scores without considering possible dependencies appear to be better than they actually are, as will be revealed in Sections 4.2.1 and 4.2.2. We compare the maintenance effort of $file_{P_1}$ (i.e., files involved in P_1 dependencies) vs. $file_e$ (i.e., files involved in explicit dependencies) measured based on the revision history of projects. We hypothesize that, if the maintenance effort of $file_{P_1}$ is greater, it means that a file with possible dependencies incur higher maintenance cost than that of explicit dependencies, and thus missing possible dependencies could distort the maintainability scores.

By mining the revision history of each project, we used K_{cmt}, K_{loc}, K_{author}, K_{issue}, $K_{issueCmt}$, and $K_{issueLoc}$ to comprehensively assess the relative maintenance cost of a $file_{P_1}$ as compared with that of a $file_e$, from 6 dimensions, similar to the work of Mo et al. [5]. The six measures are defined as follows:

$$K_{cmt} = \frac{\#commit\ of\ a\ file_{P_1}\ (avg.)}{\#commit\ of\ a\ file_e\ (avg.)}$$
$$K_{loc} = \frac{\#changeLoc\ of\ a\ file_{P_1}\ (avg.)}{\#changeLoc\ of\ a\ file_e\ (avg.)}$$
$$K_{author} = \frac{\#author\ of\ a\ file_{P_1}\ (avg.)}{\#author\ of\ a\ file_e\ (avg.)}$$
$$K_{issue} = \frac{\#issue\ of\ a\ file_{P_1}\ (avg.)}{\#issue\ of\ a\ file_e\ (avg.)}$$
$$K_{issueCmt} = \frac{\#issueCmt\ of\ a\ file_{P_1}\ (avg.)}{\#issueCmt\ of\ a\ file_e\ (avg.)}$$
$$K_{issueLoc} = \frac{\#issueLoc\ of\ a\ file_{P_1}\ (avg.)}{\#issueLoc\ of\ a\ file_e\ (avg.)}$$

where, #*commit*—the number of commits made to change a file; #*changeLoc*—the total lines of changed code when modifying a file; #*author*—the number of developers for maintaining a file; #*issue*—the number of issues that a file gets involved; #*issueCmt*—the number of commits of a file for fixing issues; #*issueLoc*—the total lines of code changed to a file for fixing issues. The bigger these measures, the more maintenance cost spent on a file.

For a project, if one of the 6 scores is bigger than 1, it means that files involved in possible dependencies incur more maintenance effort, measured in a given dimension, than those involved in explicit dependencies. Figure10 depicts the boxplots of these scores on our subjects.

3.4.3 The Setup for RQ3

Architecture anti-patterns [20] are defined as the connections among files that violate design principles and impact bug-proneness and change-proneness, leading to severe maintenance effort. Since anti-patterns are defined and detected based on code dependencies, we study to what extent and how the architecture anti-patterns detected by $D_e \cup D_{P_1}$ are different from those by D_e. Then we verify the anti-patterns detected by $D_e \cup D_{P_1}$ by investigating whether the affected files (i.e., those files involved in anti-patterns) require more maintenance effort than non-affected files. If the answer is positive, we will try to distinguish the most influential files by categorizing the affected files from the perspective of dependency type. The most influential files often incur the greatest maintenance costs, thus deserving special attention. The evaluation setup is as follows:

1) Anti-pattern Detection. Our study considered the six architecture anti-patterns defined by Mo et al. [8], including Unstable Interface (UIF), Modularity Violation Group (MVG), Unhealthy Inheritance Hierarchy (UIF), Crossing (CRS), Clique (CLQ), and Package Cycle (PKG). Table 6 describes these anti-patterns.

Detection of these anti-patterns has been automated by the DV8 tool [16]. We referred to the work of Mo et al. [20] to configure the parameters used in detecting anti-patterns. The parameter values for detection based on D_e are consistent with those based on $D_e \cup D_{P_1}$.

2) Change Qualifying. We first counted the number of anti-pattern instances (i.e., $\#Instance$) and the proportion of source files affected by these anti-patterns (i.e., *affected file*%) for the detection based on D_e and $D_e \cup D_{P_1}$ respectively. Table 10 lists the results averaged on the studied projects. Each row corresponds to the result of each architecture anti-pattern.

We then use *django* as an example to observe how possible dependencies change the anti-pattern detection results. We manually inspected *django* to pinpoint the difference between anti-pattern instances identified using $D_e \cup D_{P_1}$ versus those identified using D_e. As compared with instances identified using D_e, we categorize the instances identified using $D_e \cup D_{P_1}$ into five types: **Type_same**, **Type_larger**, **Type_smaller**, **Type_new**, and **Type_partial**. To formally illustrate these types, we use *Instance*$(D_e \cup D_{P_1}, i)$ to denote an instance detected by $D_e \cup D_{P_1}$, use *Instance*(D_e, i) to denote an instance detected by D_e, and label their affected file sets as *affectedfileSet*$(D_e \cup D_{P_1}, i)$ and *affectedfileSet*(D_e, i), respectively. The formal definitions of the five types are as follows:

Type_same means that *Instance*$(D_e \cup D_{P_1}, i)$ is the same as the corresponding *Instance*(D_e, i). That is,

$$\textit{affectedfileSet}(D_e \cup D_{P_1}, i) = \textit{affectedfileSet}(D_e, i)$$

Type_larger means that *Instance*$(D_e \cup D_{P_1}, i)$ is derived from the corresponding *Instance*(D_e, i), but with more files involved. That is,

$$\textit{affectedfileSet}(D_e \cup D_{P_1}, i) \supsetneq \textit{affectedfileSet}(D_e, i)$$

Type_smaller means that *Instance*$(D_e \cup D_{P_1}, i)$ is derived from the corresponding *Instance*(D_e, i), but with fewer files involved. That is,

$$\textit{affectedfileSet}(D_e \cup D_{P_1}, i) \subsetneq \textit{affectedfileSet}(D_e, i)$$

Type_new means that *Instance*$(D_e \cup D_{P_1}, i)$ is newly revealed, invisible using D_e only. That is,

$$\textit{affectedfileSet}(D_e \cup D_{P_1}, i) \cap \textit{affectedfileSet}(D_e, i) = \emptyset$$

Type_partial means that *Instance*$(D_e \cup D_{P_1}, i)$ is partially overlapping with that using D_e, and vice versa. That is,

TABLE 6
The architecture anti-patterns studied in this work

Anti-pattern	Definition
Unstable Interface (UIF)	refers to the design rule file that is structurally depended by and is changed frequently with many other files.
Unhealthy Inheritance (UHI)	refers to the problematic hierarchical structure where a parent class depend on one of its children, or a client of a class depends on both the base class and its children.
Crossing (CRS)	is a set of files where a file has both high fan-in and high fan-out with other files.
Clique (CLQ)	connects a group files whose structural dependencies form a strongly connected graph so that changes to any files would propagate to any other files within this group.
Modularity Violation Group (MVG)	describes two structurally independent modules that should evolve independently but actually co-change frequently in revision history.
Package Cycle (PKG)	means the two packages structurally depend on each other, violating the hierarchical package structure of a software system.

$$\textit{affectedfileSet}(D_e \cup D_{P_1}, i) \cap \textit{affectedfileSet}(D_e, i) \neq \emptyset$$
$$\text{and } \textit{affectedfileSet}(D_e \cup D_{P_1}, i) \nsubseteq \textit{affectedfileSet}(D_e, i)$$
$$\text{and } \textit{affectedfileSet}(D_e \cup D_{P_1}, i) \nsupseteq \textit{affectedfileSet}(D_e, i)$$

Table 12 shows these types of anti-pattern instances detected by $D_e \cup D_{P_1}$ in *django*. We will explain the results for each kind of ant-pattern through examples in Section 4.3.1.

3) Anti-pattern Verification. Following the work of Mo et al. [20], we further investigate whether the affected files detected by $D_e \cup D_{P_1}$ are more error-prone and change-prone than non-affected files, and how the results are different from that of D_e only. The assumption is: if the files in anti-patterns are truly error-prone and change-prone, these files may be problematic and may have substantial impacts on maintenance. For this reason developers must be aware of these architectural connections when making changes or fixing bugs [20].

We first measure the change-proneness and error-proneness of a file by computing its $\#issue$, $\#author$, $\#changeCmt$, $\#changeLoc$, $\#issueCmt$ and $\#issueLoc$—the same six measures used in RQ2. In terms of each measure, for each project we compute the $Increase(D_e)$ and $Increase(D_e \cup D_{P_1})$, where $Increase(D_e)$ (or $Increase(D_e \cup D_{P_1})$) is the average rate of the error- or change-proneness measurement of an affected file compared to that of a non-affected file in anti-patterns detected by D_e (or $D_e \cup D_{P_1}$). If the value of $Increase$ is larger than 100%, the files involved in the detected anti-patterns are the truly problematic portion that consumes more maintenance effort.

Then we compare $Increase(D_e \cup D_{P_1})$ with $Increase(D_e)$ by computing $\Delta(increase) = \frac{Increase(D_e \cup D_{P_1}) - Increase(D_e)}{Increase(D_e)}$. A positive value of $\Delta(increase)$ indicates that the supplementing of possible dependencies helps identify more problematic and architecturally connected files that incur substantial maintenance costs. Figure 11 demonstrates the results.

4) Prioritizing Affected Files. Existing design flaw detection tools [14][52] tend to report a large number of problematic files, sometimes covering half of the files in a project. So many highlighted files overwhelm users and actually hinder them in pinpointing the truly flawed files [53]. It is essential to distinguish and prioritize the most influential files that require substantial maintenance effort. Developers thus could pay more attention to these influential files during anti-pattern remediation. Given that possible dependencies incur more maintenance cost than explicit dependencies, as will be shown in Section 4.2.3, we conjecture that possible dependencies might provide a clue for such prioritization, which will be tested in this RQ3. Furthermore, since existing studies [54] showed that files with larger sizes incur more maintenance cost, we will conduct the Wilcoxon Sign-Rank Test [55] to demonstrate the uniqueness of the prioritization dimension provided by possible dependencies, different from the dimension provided by file size.

For each kind of anti-pattern, we first classify the affected files into three categories, $file_e$ (i.e., a file involved in explicit dependencies only), $file_{P_1}$ (i.e., a file involved in possible dependencies only), and $file_{P_1 \cap e}$ (i.e., a file involved in both explicit and possible dependencies). Then we evaluate the averaged $\#author$, $\#issue$, $\#changeCmt$, $\#changeLoc$, $\#issueCmt$, and $\#issueLoc$, i.e., the six measures, for the affected $file_e$ files, the affected $file_{P_1}$ files, and the affected $file_{P_1 \cap e}$ files. Table 14 illustrates the six measurements of $file_e$, $file_{P_1}$, and $file_{e \cap p_1}$ participating in anti-patterns, averaged over all projects.

4 Study Results

Following the evaluation methodology in Section 3.4, we will analyze the evaluation results corresponding to each research question.

4.1 RQ1: The Impact on Capturing Co-changed Files

4.1.1 Results

Considering the two sets of collected co-change benchmarks (i.e., Benchmark A and Benchmark B), we analyze the precision, recall, and F1 measurements of using explicit dependencies (D_e) *vs.* using the combination of explicit and possible dependencies ($D_e \cup D_{P_1}$). The results are shown in Figure 8, Figure 9, and Table 7.

Figure 8(a) presents the precision scores, i.e., $P(D_e)$ and $P(D_e \cup D_{P_1})$ for the investigated projects using diverse co-change benchmarks in Benchmark A. It shows that $P(D_e \cup D_{P_1}) > P(D_e)$ except for the first five points in the curves. We conducted the Wilcoxon Sign-Rank Test [56] to observe whether $P(D_e \cup D_{P_1})$ scores are significantly bigger than $P(D_e)$ scores on the investigated projects. The P-value is less than 0.01, indicating the statistical significance of the comparison. We then used $\Delta P(D_e \cup D_{P_1}) = \frac{P(D_e \cup D_{P_1}) - P(D_e)}{P(D_e)}$ to compute the improvement of $D_e \cup D_{P_1}$

TABLE 7
The improvement of the Precision, Recall, and F_1 scores when adding P_1 dependencies (on Benchmark A)

Benchmark	1	2	3	4	5	6	7	8	9	10	11
$\Delta P(D_e \cup D_{P_1})$	-1.97%	-1.71%	-1.00%	-0.47%	-0.06%	0.18%	0.04%	0.45%	0.23%	0.48%	0.39%
$\Delta R(D_e \cup D_{P_1})$	21.13%	20.89%	22.12%	22.79%	23.21%	23.55%	23.74%	24.39%	24.59%	25.14%	25.17%
$\Delta F_1(D_e \cup D_{P_1})$	19.92%	18.64%	18.45%	17.55%	16.35%	14.97%	13.40%	12.41%	10.91%	9.95%	8.80%
Benchmark	12	13	14	15	16	17	18	19	20	range	
$\Delta P(D_e \cup D_{P_1})$	0.71%	1.12%	1.37%	1.80%	2.33%	2.78%	3.06%	3.21%	3.34%	-1.97%–3.34%	
$\Delta R(D_e \cup D_{P_1})$	25.87%	26.57%	26.60%	26.99%	28.01%	27.48%	27.90%	27.40%	28.49%	20.89%–28.49%	
$\Delta F_1(D_e \cup D_{P_1})$	8.21%	7.77%	7.20%	6.98%	7.04%	6.80%	6.70%	6.40%	6.30%	6.30%–19.92%	

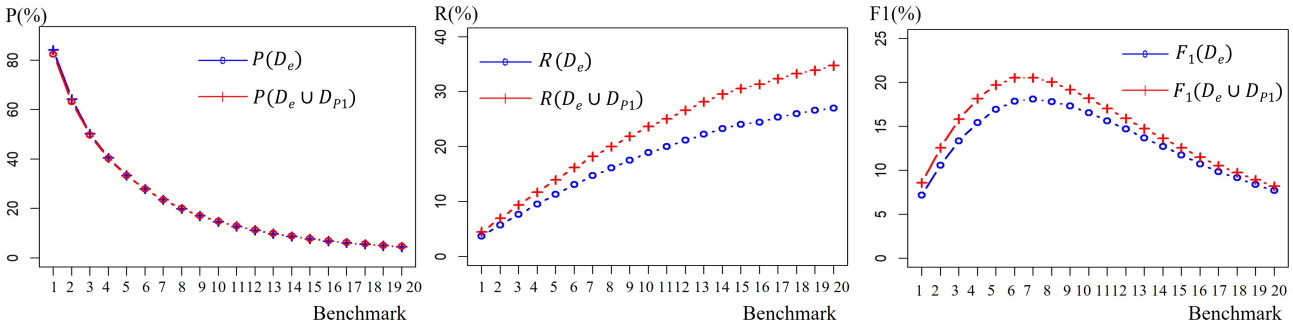


Fig. 8. The precision, recall, and F_1 measurements against 20 benchmarks (on Benchmark A)

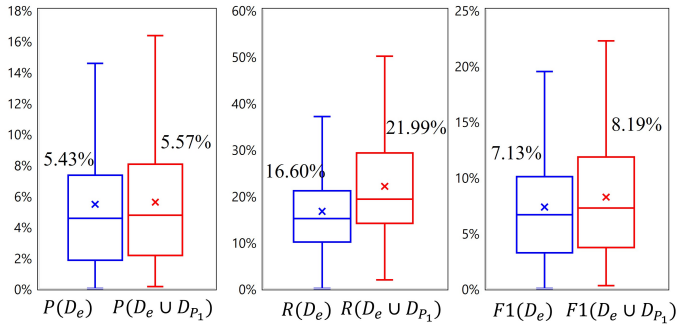


Fig. 9. The precision, recall, and F_1 measurements (on Benchmark B)

when compared with D_e for each benchmark. As illustrated in Table 7, except for the first five points in the curves, the improvement ranges from 0.04% to 3.34%.

Figure 8(b) presents recall scores, showing that adding possible dependencies always improves recall with P-value less than 0.01: $R(D_e \cup D_{P_1}) > R(D_e)$. Similarly, we used $\Delta R(D_e \cup D_{P_1}) = \frac{R(D_e \cup D_{P_1}) - R(D_e)}{R(D_e)}$ to compute the improvement of $D_e \cup D_{P_1}$ when compared with D_e for each benchmark. As shown in Table 7, the values of $\Delta R(D_e \cup D_{P_1})$ are 20.89%–28.49%, again indicating that the combination of D_e and D_{P_1} significantly improves the ability to reflect co-change relations.

Figure 8(c) illustrates the F_1 scores: $F_1(D_e \cup D_{P_1}) > F_1(D_e)$, again with P-value<0.01. The values of $\Delta F_1(D_e \cup D_{P_1})$ range from 6.30% to 19.92%, as listed in Table 7. The observation of the improvement of F_1 score is consistent with that of precision and recall scores.

Similarly, the measurement results on Benchmark B are visualized in Figure 9, labeled with the averaged values. Consistent with the statistical test on Benchmark A, the P-values here are also smaller than 0.01. On average, $\Delta P(D_e \cup D_{P_1}) = 2.58\%$, $\Delta R(D_e \cup D_{P_1}) = 32.47\%$, and $\Delta F1(D_e \cup D_{P_1}) = 14.87\%$. In summary, Figure 9 suggests the same results on Benchmark B as Benchmark A: the precision, recall, and F1 scores are improved when adding possible dependencies for capturing co-change relations.

4.1.2 *Answering RQ1*

When adding P_1 dependencies to explicit dependencies, the precision, recall and F_1 scores of capturing co-change relations are significantly improved. For the precision, both the denominator and numerator in this formula are increased when adding P_1 dependencies, while the value is enhanced by 0.04%–3.34% (except for the first 5 values) on 20 diverse benchmarks in Benchmark A, by 2.58% for Benchmark B. The recall improvements range from 20.89%–28.49% on Benchmark A and the improvement is 32.47% on Benchmark B. For the F_1 scores, the improvements range from 6.30%–19.92% on Benchmark A and the improvement is 14.87% on Benchmark B. These results indicate that a significant portion of co-changes reflects possible dependencies among files.

4.2 RQ2: The Impact on Maintainability Measures

4.2.1 *Maintainability Scores based on D_e vs. $D_e \cup D_{P_1}$*

Recall that we computed the dependency-based maintainability scores, DL and PC, using D_e and $D_e \cup D_{P_1}$ respectively. The larger the DL scores and the smaller the PC scores, the higher the maintainability levels of a project [5]. Table 8 lists the measures for all projects. We observe that, after considering D_{P_1}, the mean DL score decreased from 87.23% to 80.51%, and the mean PC score increased from 9.96% to 16.16%. This analysis shows that the apparent maintainability levels decrease after considering possible dependencies.

TABLE 8
The DL/PC value based on D_e and $D_e \cup D_{P_1}$

Statistic	DL		PC	
	D_e	$D_e \cup D_{P_1}$	D_e	$D_e \cup D_{P_1}$
Min	62.54%	51.01%	0.31%	0.31%
25th PT	82.06%	72.56%	4.20%	7.14%
50th PT	89.44%	82.45%	7.70%	13.27%
75th PT	95.18%	90.40%	13.06%	21.81%
Max	100%	99.99%	25.51%	43.75%
Mean	87.23%	80.51%	9.96%	16.16%

TABLE 9
Spearman correlations between DL/PC and Ground-truth

	DL			PC		
	D_e	$D_e \cup D_{P_1}$	Δ(%)	D_e	$D_e \cup D_{P_1}$	Δ(%)
CCOR	-0.624	-0.738	18.20	0.641	0.751	17.16
BCOR	-0.623	-0.796	27.89	0.565	0.728	28.95
CCFOR	-0.615	-0.760	23.55	0.586	0.734	25.31
BCFOR	-0.735	-0.837	13.91	0.641	0.762	18.77
CPCO	-0.712	-0.727	2.09	0.695	0.727	4.69
BPCO	-0.754	-0.831	10.24	0.700	0.759	8.45
Mean	/	/	15.98	/	/	17.22

4.2.2 Correlations between Maintainability Scores and Ground-truth

Table 9 displays the correlation results with all the P-values less than 0.01, suggesting that the correlated relationships between maintainability scores and ground-truth are statistically significant. For example, the correlation between the DL calculated using D_e and the CCOR measure is -0.624, and this correlation is increased to -0.738 when D_{P_1} is taken into consideration. The Δ column represents the improvement of correlation when comparing the scores calculated using $D_e \cup D_{P_1}$ vs. using D_e only. It is clear that, after supplementing D_e with D_{P_1}, the DL and PC scores have stronger correlations with the ground-truth maintainability measures: improving by 15.98% for DL and 17.22% for PC on average.

In addition, Table 9 shows that the Δ value regarding CPCO is less than those regarding the other five measures. One possible reason is that each of the six metrics assesses the maintainability of a project based on different assumptions, as explained in Section 3.4.2. To be rigorous, our experiments employed all six maintainability measures for a more comprehensive evaluation. The results indicate that their Δ values are consistent despite slight differences in the correlation strengths.

In general, these results show that adding possible dependencies decreases the maintainability scores, and this makes these scores more closely reflect a project's true maintainability, which is what we would desire of such measures.

4.2.3 Maintenance Cost of $file_{P_1}$ vs. $file_e$

Figure 10 shows the boxplots of K_{cmt}, K_{loc}, K_{author}, K_{issue}, $K_{issueCmt}$, and $K_{issueLoc}$ for each project. Consider the first boxplot as an example. The K_{cmt} scores are larger than 1 for at least 75% of the projects. The statistical significance tests consistently showed P-values less than 0.01. Similar results can be observed in the other five boxplots. On average, for all projects, $K_{cmt} = 1.29$, $K_{loc} = 1.36$, $K_{author} = 1.16$, $K_{issue} = 1.30$, $K_{issueCmt} = 1.37$, $K_{issueLoc} = 1.30$. The results show that, on average, the maintenance cost spent on a $file_{P_1}$ is $\frac{(1.29-1)+(1.36-1)+(1.16-1)+(1.30-1)+(1.37-1)+(1.30-1)}{6} = 30\%$ larger than that spent on a $file_e$.

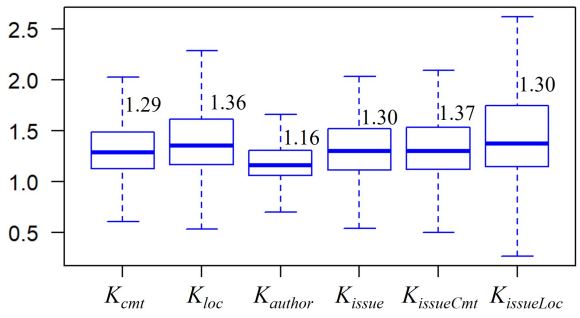


Fig. 10. The distribution of K_{cmt}, K_{loc}, K_{author}, K_{issue}, $K_{issueCmt}$, and $K_{issueLoc}$ for all projects

4.2.4 Answering RQ2

These results show that, without P_1 dependencies, architectural maintainability scores appear to be better than they actually are. When P_1 dependencies are included, the scores are decreased, more accurately reflecting the real maintainability of the system. The accuracy is improved by 15.98% or 17.22%. We observe that files involved in possible dependencies incur about 30% more maintenance costs than those in explicit dependencies, on average. One potential implication is that possible dependencies (invisible to source code) are more difficult to understand and to change, consequently consuming more maintenance effort.

4.3 RQ3: The Impact on Architecture Anti-pattern Detection

4.3.1 Anti-pattern Changes

Table 10 lists the number of anti-pattern instances (*#Instance*) and the percentage of affected files (*affectedfile%*) detected based on D_e and $D_e \cup D_{P_1}$, averaged over all projects. For example, the average number of UIF instances based on D_e and $D_e \cup D_{P_1}$ is 7.6 and 10.2, respectively. Table11 shows the Wilcoxon Sign-Rank Test to check whether *#Instance* detected by D_e is significantly smaller than that by $D_e \cup D_{P_1}$ over all projects.

The results in Table 10 and Table 11 indicate that, except for the MVG anti-pattern, both *#Instance* and *affectedfile%* significantly increase after taking into account D_{P_1} (P-value$<$0.001). This suggests that seemingly problematic but isolated files turn out to in fact be architecturally connected (through possible dependencies), and these connections have an impact on a software system. On average, the MVG anti-pattern appears to decrease slightly in terms of *affectedfile%* while the decrease is statistically significant (P-value$<$0.001) on all individual projects. Prior work reported that "unnamed" coupling is an undetectable symptom of modularity violation [57]. Our results suggest that possible dependencies can explain a portion of "unnamed" coupling incurred by dynamic language feature. Possible dependencies couple together different "modules" that evolve together. Consequently, a consideration of possible dependen-

TABLE 10
The number of instances and the percentage of affected files of anti-patterns detected from D_e and $D_e \cup D_{p_1}$

	D_e		$D_e \cup D_{P_1}$	
	#Instance	*#affectedfile%*	*#Instance*	*#affectedfile%*
UIF	7.6	49.00%	10.2	53.70%
UHI	4.3	8.10%	6.5	13.30%
CRS	8.4	24.20%	12.5	34.30%
CLQ	1.3	4.30%	1.7	13.90%
PKG	7.8	18.30%	14.8	32.20%
MVG	13.2	33.80%	13.2	33.20%

TABLE 11
The P-values of the Wilcoxon Sign-Rank Tests among #Instance or #affectedfile% of anti-patterns detected from D_e and $D_e \cup D_{p_1}$

	#Instance		*#affectedfile%*	
	before<after	before>after	before<after	before>after
UIF	<0.001	1	<0.001	1
UHI	<0.001	1	<0.001	1
CRS	<0.001	1	<0.001	1
CLQ	<0.001	1	<0.001	1
PKG	<0.001	1	<0.001	1
MVG	0.911	0.09	1	<0.001

Note: The P-values in the column *before<after* indicate that whether the results before considering D_{P_1} is significantly smaller than that after this consideration. P-values in *before>after are the vice versa.*

cies leads to a smaller (and maybe more accurate) number of files affected by MVG anti-pattern.

4.3.2 Qualifying Changes

We will show, in the analysis of the *django* project, how possible dependencies lead to anti-pattern changes, and we have categorized the detected instances. This categorization was introduced in Section 3.4.3. Table 12 summarizes these types of anti-pattern instances detected by $D_e \cup D_{P_1}$ in *django*. We now explain the results for each kind of ant-pattern.

1) UIF anti-pattern. The value of #Type_same in the UIF column in Table 12 indicates that all of 42 UIF instances detected by D_e are same included in the instances detected by $D_e \cup D_{P_1}$. Indicated by #Type_new, other 4 UIF instances detected by $D_e \cup D_{P_1}$ cannot be detected by D_e. The results indicate that supplementing possible dependencies can uncover the originally invisible UIF instances. Appendix A Figure 12(a) shows one of those 4 newly-detected UIF instances.

2) UHI anti-pattern. As shown in the UHI column in Table 12, 49 UHI instances are detected by $D_e \cup D_{P_1}$ while 46 UHI instances are detected by D_e, in total. 28 instances revealed by $D_e \cup D_{P_1}$ are same as those revealed by D_e.

TABLE 12
Categorizing anti-pattern instances detected by considering D_{P_1} in *django*

	UIF	UHI	CRS	PKG	CLQ	MVG
#Total	46	49	67	85	10	63
#Type_same	42	28	28	60	8	50
#Type_larger	0	18	33	16	2	0
#Type_smaller	0	0	0	0	0	12
#Type_new	4	3	6	9	0	0
#Type_partial	0	0	0	0	0	1

TABLE 13
The changes of CLQ instances after considering D_{P_1} in *django*

$D_e \cup D_{P_1}$		D_e			
CLQid	CLQsize	CLQid	CLQsize	Overlapped size	Difference size
1	224	1,2,8	127	127	97
2	9	3	8	8	1
3	8	4	8	8	0
4	6	5	6	6	0
5	2	6	2	2	0
6	2	7	2	2	0
7	2	9	2	2	0
8	2	10	2	2	0
9	2	12	2	2	0
10	2	11	2	2	0

Among the remaining 21 instances revealed by $D_e \cup D_{P_1}$, 18 instances are derived from the instances revealed by D_e. These derived instances contain the same base class files as those revealed by D_e, but connect a larger number of files depending on both parent class files and child class files. The additional 3 UHI instances are newly revealed by $D_e \cup D_{P_1}$ while invisible in D_e, one of which is illustrated in Appendix A 12(b).

3) CRS anti-pattern. As for CRS anti-pattern, from Table 12, we observe that among 67 instances in $D_e \cup D_{P_1}$: 6 instances are newly detected while invisible in D_e; 28 instances are the same as the corresponding instances in D_e; the remaining 33 instances are derived from those in D_e only, with the same crossing center files but involving a larger number of fan-in/fan-out files due to possible dependencies. Appendix A Figure 12(c) illustrates one CRS instance which was originally undetectable.

4) PKG anti-pattern. Similarly, we analyze the PKG instances in $D_e \cup D_{P_1}$ and those in D_e, as listed in Table 12. Among the 85 instances in $D_e \cup D_{P_1}$, 60 instances are Type_same, 9 instances are Type_new, and the other 16 instances are Type_larger, presenting the same package-level cycles as those in D_e but infecting more files in cycles. Appendix A Figure 12(d) depicts one Type_new PKG instance, where possible dependencies create a new relation between two packages, producing a new package cycle.

5) CLQ anti-pattern. Similarly, Table 12 shows that 10 CLQ instances are detected in $D_e \cup D_{P_1}$, where 8 instances are Type_same and 2 instances are Type_larger. Table 13 shows the inspection results. In Table 13, CLQid labels an CLQ instance, CLQsize denotes the number of files involved in an instance, Overlappedsize counts the files both taking part in CLQ i by $D_e \cup D_{P_1}$ and CLQ j by D_e, and Differencesize counts the files that participate in CLQ i by $D_e \cup D_{P_1}$ but not in CLQ j by D_e.

Table 13 indicates that 8 CLQ instances (CLQid =3,4,..., 10) in $D_e \cup D_{P_1}$ are the same as those (CLQid =4,5,..., 11) in D_e. The 3 separate CLQ instances (CLQid = 1,2,8) by D_e are merged into one larger instance (CLQid = 1) in $D_e \cup D_{P_1}$ with 224 files. Obviously, possible dependencies link together the originally separate instances, growing into a larger clique. CLQid=2 in $D_e \cup D_{P_1}$ is derived from CLQid=3 by D_e with one more file (i.e., *models.py*) involved due to possible dependencies, as shown in Appendix A Figure 12(e). In total, the number of CLQ instances decreases from 12 to 10 after considering possible dependencies but

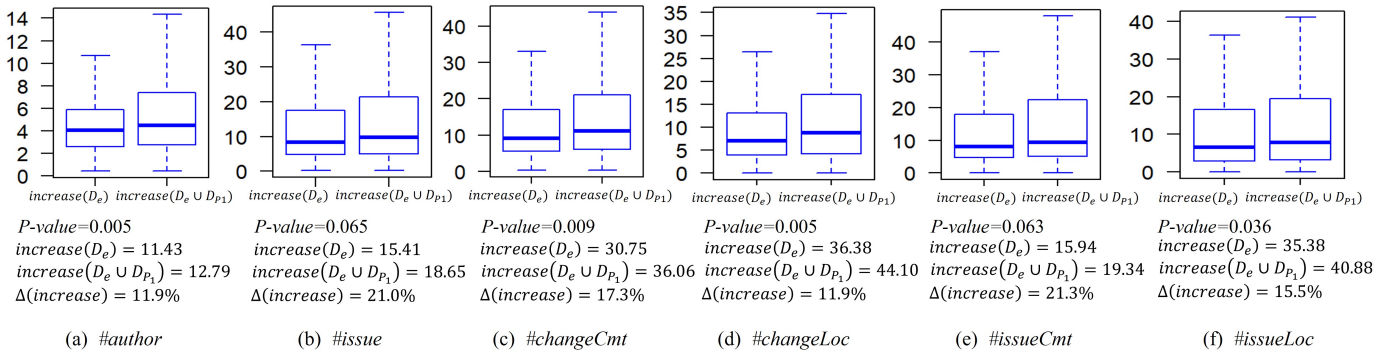


Fig. 11. The distribution of $Increase(D_e)$ and $Increase(D_e \cup D_{P_1})$ in terms of the six error- or change-proneness measures for all projects

more files are involved.

6) MVG anti-pattern. As listed in Table 12, $D_e \cup D_{P_1}$ and D_e respectively detect 63 MVG instances in *django*. Among 63 instances in $D_e \cup D_{P_1}$, 50 instances are the same as those in D_e; 12 instances are derived from those in D_e, but fewer files are affected after considering possible dependencies; the remaining instance is Type_partial, i.e., different and partially overlapped with that of D_e. Appendix A Figure 12(f) shows a MVG instance detected by D_e. Here we see how several files that were previously categorized as being in a MVG instance are no longer considered to be in violation after considering possible dependencies.

To sum up, our quantitative analysis of all projects and the case study of *django* show that originally indiscernible anti-patterns can be revealed by possible dependencies. Except for the MVG anti-pattern, considering possible dependencies also leads to anti-patterns connecting more problematic files, thus imposing a larger scope impact on software architecture than what appears by considering D_e only. The exception is that fewer files participate in MVG instances after supplementing possible dependencies. The reason is that possible dependencies can reveal a portion of "unnamed coupling" between files that have been co-changed in revision history, which has been indicated by the RQ1 results.

4.3.3 Anti-pattern Verification

Recall that we used $increase(D_e)$ and $increase(D_e \cup D_{P_1})$ to evaluate the anti-patterns detected by D_e and $D_e \cup D_{P_1}$. Figure 11 depicts the distributions of these values for all projects in terms of the six error- or change-proneness measures, such as *#author*, *#issue*, *#changeCmt*, *#changeLoc*, *#issueCmt*, and *#issueLoc*. Each sub-figure is labeled with the summary results. Considering $\#author$ in Figure 11 (a), $increase(D_e \cup D_{P_1}) = 12.79$ means that a file affected by anti-patterns detected based on $D_e \cup D_{P_1}$ requires 12.79-1=11.79 times higher developer efforts to maintain (i.e., such files are 11.79 times more error-/change-prone) than a non-affected file on average. $increase(D_e) = 11.43$ is the corresponding result when anti-patterns are detected based on D_e, averaged on all projects. $\Delta(increase) = 11.9\%$ means the value of $increase$ based on $D_e \cup D_{P_1}$ is 18.8% larger than that based on D_e. *P-value*=0.005 describes the significance level of the same statistical test as RQ1 and RQ2, checking whether $increase(D_e \cup D_{P_1})$ is larger than $increase(D_e)$.

From Figure 11 we observe that both $increase(D_e)$ and $increase(D_e \cup D_{P_1})$ are larger than 100%, meaning that anti-pattern files detected by D_e or $D_e \cup D_{P_1}$ are more change- or error-prone than other files in a project. We also see that the average value of $increase(D_e \cup D_{P_1})$ is bigger than that of $increase(D_e)$ for all six error- or change-proneness measurements. Except for Figure 11(b) and (e), all the P-values are less than 0.05. The P-values in Figure 11(b) and (e) are larger than 0.06 but still < 0.1. In general, the average value over all $\Delta(increase)$ values is 16.5%, i.e., $\frac{11.9\%+21.0\%+17.3\%+11.9\%+21.3\%+15.5\%}{6} = 16.5\%$.

Our analyses suggest that, for anti-patterns identified by $D_e \cup D_{P_1}$, the affected files consume more maintenance effort than non-affected files, as indicated by $increase(D_e \cup D_{P_1}) > 100\%$. More interestingly, possible dependencies reveal more problematic files with architectural connections, i.e., the increase rate of maintenance effort spent on affected file by $D_e \cup D_{P_1}$ is higher than that of D_e only, on average.

4.3.4 Prioritizing the Affected Files

Among all the files (i.e., $file_{e \cup P_1}$) affected by anti-patterns, we compared the maintenance cost of $file_e$, $file_{P_1}$, and $file_{e \cap P_1}$. The statistical P-values are less than 0.01, demonstrating that the maintenance cost of $file_{e \cap P_1}$ is larger than that of $file_{P_1}$, and the maintenance cost of $file_{P_1}$ is larger than that of $file_e$ with a high confidence level. Concretely, Table 14 illustrates the six measurements of $file_e$, $file_{P_1}$, and $file_{e \cap p_1}$ participating in anti-patterns, averaged over all projects. For example, the *#authornum* row shows that, among the files affected by UIF anti-pattern, the values of $\#author$ metric for $file_e$, $file_{P_1}$, and $file_{P_1 \cap e}$ are 5.9, 7.0, and 8.1, respectively. That is, $file_{P_1 \cap e}$ requires more maintenance effort than $file_{P_1}$ and $file_e$, with respect to $\#author$. Similar results can also be observed in other five anti-patterns, i.e., UHI, CLQ, PKG, CRS, and MVG. Regarding the other five maintenance measures, as shown in the remaining rows of Table 14, the results are consistent with those regarding *#authornum*.

The above results indicate the average cost of maintenance of files affected by each anti-pattern: $file_{P_1 \cap e} > file_{P_1} > file_e$. We now test whether the LoC% (i.e., $\frac{Loc\ of\ a\ file}{total\ Loc\ of\ a\ project}$) of $file_{P_1 \cap e}$ is significantly larger than

TABLE 14
The average maintenance measures of $file_e$, $file_{P_1}$, and $file_{P_1 \cap e}$ in UIF, UHI, CLQ, CRS, PKG, and MVG anti-patterns

		UIF	UHI	CLQ	CRS	PKG	MVG
#authornum	$file_e$	5.9	7.2	8.7	6.7	6.1	8.0
	$file_{P_1}$	7.0	7.9	9.2	7.8	7.1	9.3
	$file_{P_1 \cap e}$	8.1	8.6	10.2	8.7	8.1	10.4
#issuenum	$file_e$	8.6	12.2	13.4	10.3	9.1	12.1
	$file_{P_1}$	11.0	13.8	14.8	12.4	10.8	14.7
	$file_{P_1 \cap e}$	13.0	15.5	16.5	14.3	12.6	16.8
#changecmt	$file_e$	27.7	37.3	48.1	32.4	29.9	40.9
	$file_{P_1}$	35.6	42.5	52.1	39.3	36.5	50.2
	$file_{P_1 \cap e}$	44.4	49.0	60.4	47.8	44.3	59.1
#changeloc	$file_e$	923.2	1114.9	1388.4	1242.6	847.7	1573.7
	$file_{P_1}$	1123.3	1271.5	1479.3	1144.6	1070.3	1433.1
	$file_{P_1 \cap e}$	1425.0	1434.1	1730.3	1383.7	1298.7	1646.0
#issueCmt	$file_e$	9.5	13.5	15.0	11.5	10.1	13.5
	$file_{P_1}$	12.2	15.3	16.5	13.8	12.0	16.3
	$file_{P_1 \cap e}$	14.5	17.2	18.5	16.0	14.0	18.8
#issueLoc	$file_e$	686.0	695.4	762.6	568.7	448.6	642.5
	$file_{P_1}$	1796.5	844.6	898.9	717.7	611.6	805.9
	$file_{P_1 \cap e}$	2381.0	882.8	1031.6	837.3	702.9	917.9

TABLE 15
The P-values of the Wilcoxon Sign-Rank Tests among file sizes (Loc%) of $file_{P_1 \cap e}$,$file_{P_1}$, and $file_e$

Anti-pattern	$Loc\%of file_{P_1 \cap e} > Loc\%of file_{P_1}$	$Loc\%of file_{P_1} > Loc\%of file_e$
UIF	**0.07**	**0.01**
UHI	0.48	0.22
CRS	**0.07**	0.12
CLQ	0.34	0.39
PKG	0.22	**0.03**
MVG	0.21	0.25

that of $file_{P_1}$ and $file_e$. We reported the statistical P-values in Table 15.

The P-values larger than 0.1 in Table 15 show that file sizes of $file_{P_1 \cap e}$ are not significantly larger than those of $file_{P_1}$ and $file_e$, and file sizes of $file_{P_1}$ are not significantly larger than those of $file_e$. Among all the 12 P-values, there exist 4 values smaller than 0.1. It indicates that in most cases, the files in $file_{P_1 \cap e}$—those files most difficult to maintain—are not strongly correlated with the size in terms of LOC. That is, the more influential affected files indicated by possible dependencies are not necessarily larger-size files.

These observations demonstrate that, among affected files, the files involved in both explicit and possible dependencies (i.e., $file_{e \cap P_1}$) are most influential. They significantly impact change-proneness and bug-proneness, and thus deserve special attention. A consideration of possible dependencies thus provides a valuable insight into prioritizing maintenance effort.

4.3.5 Answering RQ3

Our study shows that considering possible dependencies 1) helps reveal previously indiscernible anti-patterns, or 2) leads to size increases of anti-patterns (except for the MVG anti-pattern), thus revealing connections among more problematic files and reflecting a larger-scope impact than what is revealed by considering D_e only. The differences observed in the MVG anti-patterns occur because possible dependencies couple together co-changed modules, revealing that some of the files that were previously understood to be in modularity violation relationships were not in fact so related. The addition of possible dependencies makes the detection of the MVG pattern more accurate. We also verified the anti-patterns identified by $D_e \cup D_{P_1}$ are truly problematic, consuming more maintenance effort than non-affected files. The increased rate of maintenance cost is 16.5% higher than that detected by D_e only. A more interesting finding is that the affected files involved in both explicit and possible dependencies are most influential. They significantly impact change-proneness and bug-proneness relative to other affected files. This observation provides an alternative and unique consideration for prioritizing anti-pattern refactoring: users should pay special attention to the affected files within the overlap between explicit and possible dependencies.

5 Discussion: Potential Impacts

Our study reveals the substantial impact of possible dependencies on architecture maintainability, indicating that architecture analysis and analysis tools should assess and even emphasize the impact of such dependencies.

1) Change Prediction. Our results from RQ1 indicate that possible dependencies are an important factor causing co-changes among files. We envision that adding in possible dependencies will improve the performance of prediction-related research for dynamic languages, such as the prediction of future code changes [9][58] and defect localization [10][59][60]. Some methods of change impact analysis [43][61] rely on co-change relations from a long-period revision history. Since our study shows that possible dependencies (extracted from source code) can capture a portion of co-changes, we believe that considering possible dependencies would contribute to an earlier and more accurate prediction of an architectural-level change influence. We will verify this assumption in the future.

2) Architecture Management. Our results suggest that design metrics, such as cohesion [62], coupling [63], and maintainability [5] measures, should account for possible dependencies. Otherwise, these measures may be inaccurate, for dynamic languages. As shown in RQ2, after considering possible dependencies, architectural maintainability scores are decreased, more accurately reflect reality.

Since source files related to possible dependencies take more maintenance cost than explicit dependencies as indicated in RQ2, we envision that the awareness of possible dependency related changes could help better understand and manage architectural change or erosion [64][61] during software evolution and maintenance.

3) Refactoring Prioritization. Our study of architecture anti-pattern detection indicates that the flawed files within the overlap between explicit and possible dependencies are most bug-prone and change-prone. Prior design flaw detection tools, such as SonarQube [52] and Structure101 [14], tend to report a large number of flawed files, sometimes covering almost 50% of the files in a project. Recent work, such as *ArchRoot* [65] and *Active Hotspots* [53] focus on pinpointing the truly problematic ones, the minimal number

of problematic file sets. Complementary to that research, our study of RQ3 provides a unique alternative for maintenance prioritization: the flawed files within the overlap between explicit and possible dependencies are most influential and incur severe maintenance issues. We also envision that paying attention to such files could promote a more appropriate and effective alignment of maintenance effort.

4) Tool Improvement. Even though type inference techniques have been widely studied [22][23][24][25], we have checked that popular commercial architecture analysis tools, such as Lattix [13], Structure101 [14], DV8 [16], have not employed these techniques and possible dependencies. We reported study results to the architects of DV8 and Depends [66]. They verified our findings and decided to include possible dependencies as enhancements to their tools.

5) Other Suggestions. On one hand, these results suggest that developers should be aware of code objects with possible dependencies when changing code, and they can highlight such objects by type hints introduced in Python PEP484 [26]. On the other hand, static languages like JAVA and C++ support dynamic features, such as JAVA reflection [67] and the usage of dynamic bindings [68]. Prior research [69][70] of dynamic bindings shows that including dynamic information improves the performance of architecture recovery, a technique to reconstruct the module view of the software architecture. Inspired by these work, we will generalize our study on static languages.

6 Threats to Validity

This section describes the threats that could affect the validity of our studies in this paper.

Reliability. The first threat is related to the imprecision of dependency extraction tools. We acknowledge the existence of various tools for dependency identification, such as the Dependency Finder [71], Sourcetrail [72], and Depends [66]. We chose a commercial tool named Understand to detect explicit dependencies since it is widely accepted in industry [13][14][39]. We provided a dependency extraction approach based on ENRE [21] to detect possible dependencies by employing type inference techniques, which are also utilized in the empirical study by Chen et al. [12]. It is difficult to verify the large number of possible dependencies for all 499 projects. To mitigate this threat, we adopted two ways to verify possible dependencies. On one hand, as shown in Section 3.3.2, we automatically collected benchmarks from execution traces of eight projects of the studied subjects to evaluate 1217 P_1 possible dependencies (the accuracy is 97% indicated by Table 4). On the other hand, we randomly sampled 1000 dependencies from all the 133,100 file-level P_1 possible dependencies for a manual verification. Two authors of this paper participated in this sanity check, achieving more than 90% accuracy. All our study in RQ1, RQ2 and RQ3 only considered the P_1 dependencies and explicit dependencies at file level to form a conservative baseline for the study. Our results would only be strengthened if $P_{i>1}$ dependencies could be further resolved in the future.

Another threat is the domain analysis of our subjects related to Table 3. We recruited five research volunteers (including one PhD student and four other graduate students) to label the domain of each Python project, following the application domain categories introduced in Python official community (https://legacy.python.org/about/apps/). To mitigate any subjectivity threat, they cross-validated their results and reached consensus after an independent labeling step.

To be rigorous, we reported all of the correlation analyses and comparison analyses with statistical significance indicated by P-values. Due to the existence of outliers, we employed the Spearman correlation coefficient in RQ2. We compared two populations in all studies through the Wilcoxon signed-rank test using paired samples. We adopted this statistical test to check whether one data vector is significantly larger or smaller than another one, along with showing the mean distribution.

Construct Validity. First, we focused our analysis on the modular structure view of the software architecture. The dependency-based architectural metrics (DL and PC for RQ2) and anti-pattern detection (for RQ3) used in this work are based on a software module view. One threat is that our study does not generalise to all views of the software architecture like component-connector structures [73]. The modular structure is most commonly used by software practitioners [70]. Thus we choose this view for our study.

Second, different techniques may produce different observations. To mitigate this threat, we used the state-of-the-art architecture analysis techniques and diverse metrics with comprehensive ground-truth measures in this study. In the future, we will employ other analysis techniques for further validation.

Other threats are about the extent of influence of parameter settings. We used co-change history to explore RQ1. Prior work often used arbitrary thresholds to determine co-change relations [40]. It is unclear if a particular threshold setting is generalizable. To mitigate this threat, our study used 20 co-change benchmarks (i.e., 20 different threshold settings), $Benchmark_i, i = 1, 2, ..., 20$, each containing file pairs that have changed together at least i times. Moreover, architecture anti-pattern detection for RQ3 involves parameter settings. The change to the configuration would produce different anti-pattern instances. To partially mitigate this threat, we used the default settings recommended by the prior work [39][46].

Internal Validity. Our finding in RQ2 shows that the affected files, which involve possible dependencies, consume more maintenance costs than other files. We explored the correlation between source files involved in possible dependencies and software maintainability. However, we are not claiming causal relations. The study of how possible dependencies incur maintenance effort will be our future work.

Another threat is that other factors (i.e., file size) may lead to the observation, i.e., files involved in possible dependencies have a higher maintenance cost than those with explicit dependencies. For example, files with more lines of code always require more effort to maintain [54]. We mitigated this threat by conducting the Wilcoxon Sign-Rank Test shown in Table 11, indicating little correlation between the anti-pattern affected files with more maintenance costs and their file sizes.

External Validity. The main threat to external validity is that our study on various Python projects may produce inconsistent observations. To generalize our study, we extended our dataset to 499 popular open-source Python projects collected from Github. These projects exhibit diverse sizes and domains, as depicted in Figure 6 and Table 3. Besides, our study focused on the impact of possible dependencies on architecture maintainability by contrasting with explicit dependencies. We have not investigated the relation between code practices and the impact of possible dependencies, which is future research.

Second, our study may not generalize to all the dynamic features of dynamic programming languages. Besides enabling the lack of type declarations, Python supports other kinds of dynamic features such as introspection, object changes, code generation, and meta-abstraction [33][74][75]. Our study is, however, the first step towards understanding possible dependencies created due to dynamic typing. The resolution of possible dependencies related to dynamic features is challenging for static program analysis. An exploration of such features through using run-time tracing would be another feasible direction, as indicated by the work of [33]. Moreover, other dynamic languages like Ruby and JavaScript also lack strong typing, the same feature that our work has studied. We are confident that our study of this feature could be extended to these languages.

7 Related Work

Code dependencies are fundamental to architecture analysis and other software analysis tasks like malware detection [76][77], reuse detection [78], and test case generation [79]. Our work focus on dependency-based software architecture analysis and this section will introduce the related work.

7.1 Co-change Prediction

Co-change prediction techniques identify the code artifacts needed to be modified together by a given change. The work of Badri [18] and Petrenko [19] predicted the affected code based on dependency graph statically extracted from source code. Poshyvanyk et al. [31] and Wiese et al. [80] used conceptual or semantic coupling to reason about the effects of a change to a software system. Zimmermann [81] and Rolfsnes [42] identified potentially relevant items based on history or evolutionary coupling. Empirical studies [29][40] reported that syntactic dependencies (extracted from code syntax), semantic dependencies (extracted from code textual information), and history dependencies (extracted from revision history) created the different but complimentary structures of a software system. Other work [82] leveraged hybrid features such as evolutionary and semantic information for co-change prediction. Different from those work, our study treated the co-change files mined from revision history as benchmarks to analyze how likely possible dependencies would capture co-change relations.

7.2 Maintainability Measurement

Much prior work focus on measuring the maintainability at code level. Maintainable source code should present higher cohesiveness within code elements and lower coupling between elements. LCOM, designed by Chidamber and Kemerer [83], computes the lack of cohesiveness among class method members. Briand et al. [84] designed a group of cohesion metrics to indicate the error-prone and defect-prone software elements. CBO, also proposed by Chidamber [83], are the most influential coupling metrics. CBO counts the number of other classes that are coupled to a class.

Instead of focusing on code implementation level, architecture modularity/maintainability measures aim to compare design alternatives and indicate architecture decay [5]. The work of Maccormack [4] presented the Propagation Cost (PC) to measure how tightly source files are coupled together based on direct and indirect dependencies among files. On the contrary, Decoupling Level (DL) [5] evaluates how likely a software system can be decoupled into small and independently separate modules. The larger DL and the smaller PC will indicate a more maintainable software architecture. Both PC and DL assume that code dependency structure can embody software design decisions, and thus aid assessing architecture quality.

Our work utilized the DL and PC to observe the impact of possible dependency on architecture maintainability and measures.

7.3 Architecture Anti-pattern Detection

Researchers have proposed various methods to detect architecture-level violations and smells that cause software architecture decay or erosion, adversely impacting software internal quality. Marinescu et al. [85] designed metric-based rules to identify design flaws, using cohesion and complexity metrics at class level. The work by Bertran et al. [86] combined code-level metrics with architecture-level metrics to pinpoint the smells such as "God Class" that lead to architecture decay.

The studies by Le et al. [87] categorized four types of architectural smells, including concerned-based smells, dependency-based smells, interface-based smells, and coupling-based smells, in terms of their symptoms, source, and consequence. Arcan tool [88][89] detects dependency-based architecture smells such as cycle dependency, unstable dependency, and hub-like dependency using subgraph pattern-based techniques. Mo et al. [8][20] identified problematic anti-patterns that violate design principles and impact maintainability through the combination of syntactic dependencies and revision history records.

Recent work [53] pointed out that existing design flaw detection tools tend to report a large number of problems, hindering users from pinpointing the truly flawed files. To address this issue, Qiong et al. [53] detected a few dominating *Active Hotspots* which accumulate long-term maintenance cost, by tracking the interactions among files in software evolution. *ArchRoot* [65] captures the the minimal number of design rule spaces to capture the large number of a project's bug-prone files with high maintenance effort. Liu et al. [90] investigated the most problematic *root* files in architecture anti-patterns. Their results revealed that active and overlapping root files among multiple anti-patterns should be prioritized during design problem resolution.

Our work exploited the dependency-based anti-patterns to explore how possible dependencies would reveal problematic file groups that incur severe architecture-level maintainability issues. We also explored an alternative of identifying the most severe flawed files that should having maintenance priority.

7.4 Language Dynamic Features

Dynamic languages such as Python, JavaScript, and Ruby, support dynamic typing code objects. Nanz et al. [91] and Ray et al. [92] revealed that dynamic languages are more prone to run-time defects due to the lack of static type checking. Kleinschmager et al. [93] presented evidence that static typing can help fix type errors. To mitigate type-related errors, much research has explored type inference by static code analysis [22][23][24][25]. The work of Aycock et al. [94] and Rigo et al. [95] resolved types of objects with unknown types based on objects of known types through data flow analysis. Xu et al. [35] leveraged type hints implied by code conventions such as variable names, code comments, and class attribute access. Recent work proposed neural network frameworks for type inference [96][25]. Our work has leveraged existing type inference techniques to detect possible dependencies and explored their impact on architectural maintainability.

Recent research has also studied dynamic features of programming languages, such as duck typing, introspection/reflection, object changes, code generation, and library loading. Milojkovic et al. [97] studied how often duck typing is used in Smalltalk. Åkerblom et al. [98] inspected execution traces to measure the degree to which the method calls are monomorphic or polymorphic at run-time in Python. Åkerblom et al. [33] observed the usage of these features. Wang et al. [99] demonstrated the correlation between frequent usage of dynamic features and change-proneness of related code. Chen et al. [12] focused on the misuse practice of dynamic typing in Python by employing type inference techniques. A recent work [100] detected patterns that are followed by non-trivial type annotation practices, and explored the features of type-annotated modules. Our study also investigated dynamic typing, but for a different purpose.

8 Conclusion

In our study of 499 open-source Python projects we explored the architecture-level maintainability impact of possible dependencies caused by the lack of type information in dynamic languages. Our study presented strong evidence that the architectural impact of possible dependencies is nontrivial, and in fact surprisingly high compared with explicit dependencies.

Our results benefit software research and practice by: 1) providing empirical evidence of the impact of dynamic typing on design structures, and 2) suggesting how automated architecture analysis tools can be improved—augmented with possible dependencies related to dynamic typing. These results also suggest the need to improve coding practices in dynamic languages by making developers more aware of the impacts of their dynamic typing choices, thus potentially leading to the adoption of practices that result in more maintainable software.

Appendix
Architecture Anti-pattern Cases

This section depicts the anti-pattern examples detected by considering P_1 possible dependencies in *django* project. This part is a supplement for Section 4.3.2.

We used the DSM structure to visualize anti-pattern instances. In the DSM in Figure 12(a), each row and each column corresponds to a Python file in the same order; 'X' in cell (i, j) rendered in black color denotes the explicit dependency only; '$\divideontimes$' in blue color means that both explicit dependency and possible dependency exist between the two files; '$+$' in red color denotes the possible dependency only; the number in one cell means the number of co-changes between two Python files. For instance, the "$\divideontimes$,5" in the cell $(2, 1)$ indicates that there exist both explicit and possible dependencies from *migration.py* to *state.py*, and the two python files have co-changed 5 times in the examined revision history.

1) UIF instance. Figure 12(a) shows one newly-detected UIF instances in *django*. *state.py* is an unstable interface file that is detected by $D_e \cup D_{P_1}$ but is missed by D_e. Without possible dependencies, the history influence scope of *state.py* is 8 since only 8 files depend on and co-changed with *state.py*. This anti-pattern instance is unable to be detected based on D_e because the history influence scope of *state.py* is less than 10, the minimal threshold as configured. After considering possible dependencies, 3 more files depend on and co-changed with *state.py*, surpassing the UIF detection threshold. That is, supplementing possible dependencies can uncover the originally invisible UIF instances.

2) UHI instance. In Figure 12(b), "inherit" indicates that *forms.py* is a parent class file and *models.py* is a child class file. Besides the original dependencies from the two files, *utils.py* and *edit.py*, to the child class file, supplementing possible dependencies produces new connections from these two files to the parent class file, thus constructing a UHI anti-pattern.

3) CRS instance. Figure 12(c) illustrates one CRS instance with *base/operations.py* as the crossing center. After supplementing possible dependencies, *base/operations.py* has 4 fan-in files and 4 fan-out files, creating this CRS anti-pattern that was originally undetectable.

4) PKG instance. Figure 12(d) depicts one PKG instance. Without considering possible dependencies, the package *django/db/models* unidirectionally depends on the package *django/core*. Possible dependencies create the opposite relation between the two packages, consequently producing a new package cycle.e

5) CLQ instance. In Figure 12(e), one more file (i.e., *models.py*) involved in this CLQ anti-pattern due to possible dependencies, making this instance become larger when considering possible dependencies.

6) MVG instance. Figure 12(f) shows a MVG instance detected by D_e. The module including the first file has frequently co-changed with another module composed by the other files in this DSM. No syntactically explicit dependencies exist between the two modules. After adding

		1	2	3	4	5	6	7	8	9	10	11	12
1	django/db/migrations/state.py	(1)	5	5	4	6	7	3	10	10	25	3	22
2	django/core/management/commands/migrate.py	※,5	(2)	※,11	15	X,13	X,4	3	+,1	2	※,6	X,1	3
3	django/db/migrations/executor.py	※,5	11	(3)	1	※,6	X,4	1	※,9	4	4	4	3
4	django/core/management/commands/makemigrations.py	X,4	15	1	(4)	※,11	X,3	※,7	+		※,9		2
5	django/db/migrations/loader.py	+,6	13	6	11	(5)	※,15	13	※,8	5	13	1	3
6	django/db/migrations/graph.py	※,7	4	4	3	15	(6)	3	※,7	3	12	1	3
7	django/db/migrations/questioner.py	+,3	3	1	7	X,13	3	(7)	2	3	7	1	1
8	django/db/migrations/migration.py	+,10	1	9		8	7	2	(8)	5	8	5	3
9	django/db/migrations/operations/fields.py	X,10	2	4		5	3	3	5	(9)	13	9	X,26
10	django/db/migrations/autodetector.py	※,25	6	4	9	13	※,12	※,7	※,8	X,13	(10)	3	X,19
11	django/db/migrations/operations/special.py	※,3	1	4		1	1	1	5	9	3	(11)	10
12	django/db/migrations/operations/models.py	X,22	3	3	2	3	3	1	X,3	X,26	19	10	(12)

(a) An UIF instance is newly detected by $D_e \cup D_{P_1}$ while it was originally invisible by D_e

		1	2	3	4
1	django/forms/forms.py	(1)			
2	django/forms/models.py	Inherit	(2)		
3	django/contrib/admin/utils.py	+	X	(3)	
4	django/views/generic/edit.py	+	X		(4)

(b) An UHI instance is newly detected by $D_e \cup D_{P_1}$ while it was originally invisible by D_e

		1	2	3	4	5	6
1	django/db/backends/base/features.py	(1)	6	10	2	11	6
2	django/db/models/expressions.py	6	(2)	X,15	※,13	X,7	6
3	django/db/models/sql/compiler.py	X,10	※,15	(3)	X,9	X,11	6
4	django/db/models/fields/__init__.py	2	X,13	9	(4)	X,7	5
5	django/db/backends/base/operations.py	X,11	+,7	+,11	+,7	(5)	17
6	django/db/backends/sqlite3/operations.py	6	X,6	6	※,5	X,17	(6)

(c) A CRS instance is newly detected by $D_e \cup D_{P_1}$ while it was originally invisible by D_e

		1	2	3	4	5	6	7	8	9	10
1	django/db/models/base.py	(1)		X	X			+		X	+
2	django/db/models/expressions.py		(2)					+		X	+
3	django/db/models/options.py	X		(3)				+		X	+
4	django/db/models/query.py	X	X	X	(4)	X		+		X	+
5	django/db/models/aggregates.py		※			(5)		+		X	+
6	django/db/models/lookups.py		X				(6)	+		X	+
7	django/db/models/__init__.py	X	X		X	X	X	(7)		X	+
8	django/core/paginator.py	+			+				(8)		
9	django/core/exceptions.py									(9)	
10	django/core/__init__.py										(10)

(d) A PKG instance is newly detected by $D_e \cup D_{P_1}$ while it was originally invisible by D_e

		1	2	3	4	5	6	7	8	9
1	django/contrib/gis/db/backends/oracle/models.py	(1)				+				
2	django/contrib/gis/db/models/aggregates.py		(2)			※				
3	django/contrib/gis/db/models/lookups.py			(3)	X	X				
4	django/contrib/gis/db/backends/oracle/operations.py	X	X		(4)	X		X		
5	django/contrib/gis/db/models/fields.py			X		(5)				X
6	django/contrib/gis/db/models/sql/conversion.py						(6)	X		
7	django/contrib/gis/db/backends/base/operations.py					X		(7)	X	
8	django/contrib/gis/db/models/functions.py					X	X	X	(8)	X
9	django/contrib/gis/db/backends/postgis/operations.py			X		X		X		(9)

(e) A CLQ instance detected by $D_e \cup D_{P_1}$ involves one more files than the corresponding instance by D_e

		1	2	3	4	5	6	7	8	9	10	11	12	13
1	django/db/models/sql/where.py	(1)	10	8	8	8	9	6	8	6	12	8	14	11
2	django/db/models/query_utils.py	+,10	(2)	X,3	15	6	11	X,7	6	4	10	6	X,11	20
3	django/db/models/lookups.py	8	X,3	(3)	2	1	X,6	X,2	2	4	3	4	X,8	3
4	django/db/models/fields/related.py	8	X,15	2	(4)	X,13	10	X,13	5	5	8	7	X,61	X,45
5	django/db/backends/__init__.py	8	6	1	13	(5)	8	5	5	6	11	5	25	33
6	django/db/models/expressions.py	+,9	X,11	6	10	8	(6)	X,8	13	3	8	2	X,13	13
7	django/db/models/__init__.py	6	X,7	X,2	X,13	5	X,8	(7)	X,5	2	5	4	X,14	X,13
8	django/db/models/aggregates.py	8	6	2	5	5	X,13	X,5	(8)	3	5	3	X,3	4
9	django/contrib/gis/db/backends/postgis/operations.py	6	4	4	5	X,6	X,3	X,2	3	(9)	5	3	7	6
10	django/db/models/sql/subqueries.py	+,12	X,10	3	8	11	8	X,5	5	5	(10)	6	10	X,29
11	django/db/models/sql/datastructures.py	8	6	4	7	5	2	X,4	3	3	6	(11)	5	9
12	django/db/models/fields/__init__.py	14	X,11	8	61	25	X,13	X,14	3	7	10	5	(12)	X,27
13	django/db/models/query.py	11	X,20	3	X,45	33	X,13	X,13	X,4	6	X,29	9	X,27	(13)

(f) A MVG instance detected by $D_e \cup D_{P_1}$ involves three less files (as highlighted in blue rows) than the corresponding instance by D_e

Fig. 12. Anti-pattern examples in *django*

possible dependencies, 3 originally affected files (rendered in blue) becomes un-affected by this MVG instance. The reason is that some co-changed files that have no explicit dependencies now show possible dependencies.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (2018YFB1004500), National Natural Science Foundation of China (62002280, 61721002, 61833015, 61902306), the Fundamental Research Funds for the Central Universities, China Postdoctoral Science Foundation (2020M683507, 2019TQ0251, 2020M673439), and Youth Talent Support Plan of Xi'an Association for Science and Technology (095920201303). This work was also partially supported by the United States National Sciences Foundation grants 1835292, 1823177, 1817267, and 1816594.

REFERENCES

[1] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 27–36.

[2] M. Chatterjee, S. K. Das, and D. Turgut, "Wca: A weighted clustering algorithm for mobile ad hoc networks," *Cluster computing*, vol. 5, no. 2, pp. 193–204, 2002.

[3] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 197–208.

[4] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006.

[5] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016, pp. 499–510.

[6] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012, pp. 167–178.

[7] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 662–665.

[8] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2015, pp. 51–60.

[9] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[10] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 531–540.

[11] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[12] Z. Chen, Y. Li, B. Chen, W. Ma, L. Chen, and B. Xu, "An empirical study on dynamic typing related practices in python systems," in *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. –.

[13] Lattix, "https://www.sdcsystems.com/tools/lattix-software/lattix-architect/," 2004-2022.

[14] Structure101, "https://structure101.com/," 2004-2022.

[15] S. Understand, "https://scitools.com/," 1996-2022.

[16] ArchDia, "https://archdia.com," 2004-2022.

[17] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[18] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*. IEEE, 2005, pp. 9–pp.

[19] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 10–19.

[20] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[21] W. Jin, Y. Cai, R. Kazman, Q. Zheng, D. Cui, and T. Liu, "Enre: a tool framework for extensible entity relation extraction," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 2019, pp. 67–70.

[22] J. Palsberg, "Object-oriented type inference," in *Proc. OOPSLA'91*, 1991, pp. 146–161.

[23] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 5673. Springer-Verlag, August 2009.

[24] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 1859–1866.

[25] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: inferring javascript function types from natural language information," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 304–315.

[26] Python, "https://www.python.org/dev/peps/pep-0484/," 2001-2022.

[27] W. Jin, Y. Cai, R. Kazman, G. Zhang, Q. Zheng, and T. Liu, "Exploring the architectural impact of possible dependencies in python software," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1–13.

[28] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*. MIT press, 2000, vol. 1.

[29] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 692–701.

[30] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 190–198.

[31] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical software engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[32] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on software engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[33] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad, "Tracing dynamic features in python programs," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 292–295.

[34] L. Wei, Z. Chen, W. Ma, C. Lin, X. Lei, and B. Xu, "An empirical study on the characteristics of python fine-grained source code change types," in *IEEE International Conference on Software Maintenance Evolution*, 2017.

[35] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 607–618.

[36] P. docs, "https://docs.python.org/3/glossary.html#term-duck-typing," 2001-2022.

[37] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[38] Z. Codabux and B. J. Williams, "Technical debt prioritization using predictive analytics," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 704–706.

[39] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele, "Experiences applying automated architecture analysis tool suites," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 779–789.

[40] D. Cui, T. Liu, Y. Cai, Q. Zheng, Q. Feng, W. Jin, J. Guo, and Y. Qu, "Investigating the impact of multiple dependency structures on software defects," in *Software Engineering, 2019. ICSE 2019. Proceedings. 41th International Conference on*. IEEE, 2019, pp. –.

[41] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 51.

[42] T. Rolfsnes, L. Moonen, and D. Binkley, "Predicting relevance of change recommendations," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 694–705.

[43] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[44] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 488–498.

[45] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120–137, 2017.

[46] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, 2019.

[47] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol, "Extracting change-patterns from cvs repositories," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 221–230.

[48] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 325–334.

[49] D. Sas, P. Avgeriou, R. Kruizinga, and R. Scheedler, "Exploring the relation between co-changes and architectural smells," *SN Computer Science*, vol. 2, no. 1, pp. 1–15, 2021.

[50] C. Bird, T. Menzies, and T. Zimmermann, *The art and science of analyzing software data*. Elsevier, 2015.

[51] L. Myers and M. J. Sirois, "Spearman correlation coefficients, differences between," *Encyclopedia of statistical sciences*, vol. 12, 2004.

[52] SonarQube, "https://www.sonarqube.org/," 2008-2022.

[53] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, and H. Fang, "Active hotspot: an issue-oriented model to monitor software evolution and degradation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 986–997.

[54] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.

[55] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, 2019.

[56] A. Kaur and R. Kumar, "Comparative analysis of parametric and non-parametric tests," *Journal of computer and mathematical sciences*, vol. 6, no. 6, pp. 336–342, 2015.

[57] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 411–420.

[58] H. Kagdi, "Improving change prediction with fine-grained source code mining," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 559–562.

[59] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 46–57.

[60] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[61] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 95–105.

[62] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[63] W. Jin, T. Liu, Y. Qu, Q. Zheng, D. Cui, and J. Chi, "Dynamic structure measurement for distributed software," *Software Quality Journal*, vol. 26, no. 3, pp. 1119–1145, 2018.

[64] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 235–245.

[65] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: a new model for representing and analyzing software architecture," *IEEE Transactions on Software Engineering*, 2018.

[66] Multilang-depends, "https://github.com/multilang-depends/depends," 2018-2022.

[67] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE, 2017, pp. 507–518.

[68] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 264–280, 2000.

[69] A. E. Hassan, Z. M. Jiang, and R. C. Holt, "Source versus object code extraction for recovering software architecture," in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, 2005, pp. 10–pp.

[70] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

[71] D. Finder, "https://depfind.sourceforge.io/," 2002-2022.

[72] Sourcetrail, "https://www.sourcetrail.com/," 2014-2022.

[73] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[74] Python, "https://docs.python.org/3/reference/datamodel.html," 2001-2022.

[75] Y. Peng, Y. Zhang, and M. Hu, "An empirical study for common language features used in python projects," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 24–35.

[76] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[77] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[78] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[79] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2016.

[80] I. S. Wiese, R. Ré, I. Steinmacher, R. T. Kuroda, G. A. Oliva, C. Treude, and M. A. Gerosa, "Using contextual information to predict co-changes," *Journal of Systems and Software*, vol. 128, pp. 220–235, 2017.

[81] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[82] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 119–128.

[83] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

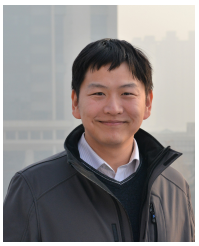
[84] L. C. Briand, J. W. Daly, and J. Wuest, "A unified framework for cohesion measurement," in *Software Metrics, IEEE International Symposium on*. IEEE Computer Society, 1997, pp. 43–43.
[85] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.
[86] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1090–1093.
[87] D. Le, D. Link, Y. Zhao, A. Shahbazian, C. Mattmann, and N. Medvidovic, "Toward a classification framework for software architectural smells," *Techincal Report csse. usc. edu*, 2017.
[88] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 282–285.
[89] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 433–437.
[90] J. Liu, W. Jin, Q. Feng, X. Zhang, and Y. Dai, "one step further: investigating problematic files of architecture anti-patterns," in *2021 IEEE 32st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 1–12.
[91] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 778–788.
[92] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.
[93] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 153–162.
[94] J. Aycock, "Aggressive type inference," *language*, vol. 1050, p. 18, 2000.
[95] A. Rigo and S. Pedroni, "Pypy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 944–953.
[96] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.
[97] N. Milojkovic, M. Ghafari, and O. Nierstrasz, "It's duck (typing) season!" in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 312–315.
[98] B. Åkerblom and T. Wrigstad, "Measuring polymorphism in python programs," in *ACM SIGPLAN Notices*, vol. 51, no. 2. ACM, 2015, pp. 114–128.
[99] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of python dynamic features on change-proneness." in *SEKE*, 2015, pp. 134–139.
[100] W. Jin, D. Zhong, Z. Ding, M. Fan, and T. Liu, "Where to start: Studying type annotation practices in python," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 529–541.

Wuxia Jin received the PhD degree in Computer Science from Xi'an Jiaotong University, Xi'an, China. She is an assistant professor of the School of Software Engineering Institute, Xi'an Jiaotong University. Her research interests include software analysis, software architecture, and software evolution.

Dinghong Zhong is a master student, majored in the Software Engineering, supervised by Wuxia Jin. His research direction includes code dependency analysis and typing for dynamic languages.

Yuanfang Cai is a Professor at Drexel University. Her primary research interests are software design, software architecture, software evolution, technical debt detection, and software economics. She has created a number of influential concepts, methods, and tools, including Design Rule Space, Decoupling Level, Modularity Violation, Hotspot patterns, and the definition, detection, and quantification of architectural debt, She has authored about 100 publications and four patents. Dr. Cai is currently serving on program committees and organizing committees for multiple top conferences, and serves as an Associate Editor and editorial board member for top journals in the area of software engineering. The tools and technologies from Dr Cai's research, including the DV8 tool suite, have been licensed and adopted by multiple multinational corporations.

Rick Kazman is a Professor at the University of Hawaii and a Visiting Researcher at the Software Engineering Institute of Carnegie Mellon University. His research interests are software architecture, design and analysis, visualization, and software engineering economics. He is the author of over 250 publications, co-author of three patents and seven books, including Software Architecture in Practice, Technical Debt in Practice, and Designing Software Architectures: A Practical Approach. His research has been cited over 26,000 times according to Google Scholar.



Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. He visited the School of Electrical and Computer Engineering, Cornell University from 2016 to 2017. Since 2010, he has been with Xi'an Jiaotong University. Currently, he is a professor at the School of Cyber Science and Engineering. His research interests include cyber-physical system and software engineering.