Names Are All You Need: Effective and Safe Regression Test Selection for Python

YOU WANG, Zhejiang University, China
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany
ZHONGXIN LIU, Zhejiang University, China

Regression test selection (RTS) reduces the cost of regression testing by executing only those tests affected by a code change. Despite extensive study of RTS in statically typed languages such as Java, achieving effective and safe RTS in Python is challenging. Python's dynamic typing makes precise call-graph construction difficult, which can cause call-graph-based RTS to miss affected tests, and hence, compromise safety. Python's eager importing mechanism, in contrast, renders file-level dependency analysis overly conservative. This paper presents NameRTS, the first Python RTS approach based on fine-grained dependency analysis. NameRTS models a Python program as a bipartite graph of code element nodes (e.g., classes, functions, global variables) and name nodes (i.e., identifiers used to reference code elements), with edges capturing definitions and references. RTS is formulated as a reachability problem on this graph: a test is selected if any modified code element is reachable from the names used in that test. This design avoids call-graph construction, enabling a conservative analysis amenable to safety. To control dependency cascades introduced by coarse name matching, NameRTS applies two pruning strategies that leverage prior test executions and context information to refine name matching. To evaluate NameRTS, we construct the first Python RTS dataset with a ground truth indicating which test files are affected by each commit. It includes 500 commits drawn from 10 real-world Python projects. We compare NameRTS with the best-performing baseline, BabelRTS, an RTS technique based on coarse file-level dependencies. On this benchmark, NameRTS skips 69.90% of test files on average, outperforming BabelRTS by 146.5%. It also reduces end-to-end testing time by 45.59%, yielding a 107.7% improvement over BabelRTS. In terms of safety, NameRTS selects all affected tests for 99.6% of commits, with only rare misses in exceptional cases. In contrast, BabelRTS is safe for 76.6% of commits. These results demonstrate the effectiveness of NameRTS, paving the way for more efficient regression testing in Python.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: regression test selection, Python, dependency analysis, program analysis, software testing

ACM Reference Format:
You Wang, Michael Pradel, and Zhongxin Liu. 2026. Names Are All You Need: Effective and Safe Regression Test Selection for Python. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA127 (October 2026), 23 pages. https://doi.org/10.1145/3832218

1 Introduction

Regression testing is a standard practice in software maintenance. It checks whether code changes break existing functionality or introduce new defects, helping developers catch problems early

Authors' Contact Information: You Wang, Zhejiang University, College of Computer Science and Technology and The State Key Laboratory of Blockchain and Data Security, Hangzhou, China, prinzywang@zju.edu.cn; Michael Pradel, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, michael@binaervarianz.de; Zhongxin Liu, Zhejiang University, College of Computer Science and Technology and The State Key Laboratory of Blockchain and Data Security, Hangzhou, China, liu_zx@zju.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA127
https://doi.org/10.1145/3832218

[36, 53, 54, 56]. Unfortunately, regression testing is costly [20, 28]. In continuous integration, the full test suite is often executed on every commit. Bouzenia et al. report that building and testing account for 91.2% of virtual machine time on GitHub Actions [18], highlighting the scale of this cost. On the other hand, Győri et al. show that across 13961 change sets in an open-source ecosystem, only about 7.8% to 17.4% of tests are actually relevant to a typical code change [27], demonstrating that running full test suites is often an order of magnitude more expensive than necessary.

To reduce the cost of regression testing, *regression test selection* (RTS) has been proposed [33, 43, 46]. Instead of executing all tests, RTS analyzes a code change and selects only those tests that are likely to be affected by the change [24, 37, 54]. RTS techniques aim for two objectives: *effectiveness*, by reducing the number of selected tests and end-to-end testing time; and *safety*, by ensuring that all tests that could reveal a regression remain in the selection [45]. Most successful RTS techniques are based on dependency analysis, i.e., selecting a test if it depends on a modified component [54]. To date, the majority of RTS techniques have been developed and evaluated for statically typed languages, most notably Java [26, 40, 57, 58].

Python's flexibility and versatility have made it widely adopted across domains such as scientific computing and data analysis, and it is now considered the lingua franca of artificial intelligence [6–8, 21, 30]. According to a GitHub report [5], Python was the second most popular language in 2025. While RTS has been widely explored in statically typed languages [26, 57], achieving effective and safe RTS in Python faces two fundamental challenges. First, Python's dynamic typing makes it difficult to accurately infer variable types and statically resolve method definitions. As a result, precise and efficient call-graph construction for Python is significantly harder. Existing call graph construction tools for Python face notable completeness and scalability limitations. For example, Bouzenia et al. [17] compare the call graphs produced by state-of-the-art static and dynamic analyzers (PyCG [47] vs. DynaPyt [22]) and find PyCG fails on 11 of 50 real-world projects due to timeouts, memory exhaustion, or crashes, and that even on the 39 successful projects it detects only 49% of the call edges observed by DynaPyt at runtime. Second, Python's eager importing mechanism executes global-scope code across modules even when only a submodule is imported. Specifically, when a submodule is imported (e.g., `sympy.core.numbers`), the Python interpreter implicitly imports and executes the `__init__.py` files of all parent packages (e.g., `sympy/__init__.py` and `sympy/core/__init__.py`), which can in turn trigger additional imports and code execution [4].

These two challenges substantially limit the applicability of existing RTS techniques to Python. Most prior RTS approaches are based on dependency analysis, which can be broadly categorized into function-level and file-level techniques. *Function-level RTS* [16, 39, 49, 57, 58] relies on call graph construction to approximate test reachability. In Python, the completeness and scalability limitations of call graph construction make such approaches unsafe and ineffective. *File-level RTS* [26, 34, 35, 42] constructs dependencies between source files and selects tests whose dependent files are modified. In Python, this approach is fundamentally limited, since Python's eager importing feature inflates dependency scopes and leaves little room for meaningful reduction. Beyond dependency-based techniques, *coverage-based RTS* [32] selects tests based on whether code changes were covered in previous executions. Because eager importing executes substantial global code during test initialization, many tests exhibit broad coverage even without semantic dependence. As a result, changes to the global code force the re-execution of most tests, severely limiting effectiveness.

To enable effective RTS in Python, our core insight is that while constructing a precise and complete call graph for Python is difficult, constructing an over-approximate dependency structure is comparatively easy. By assuming that any reference to a name may reach all code elements defined under that name, we can build a complete, though imprecise, dependency graph without requiring type inference or static call-graph resolution.

Motivated by this insight, this paper introduces NameRTS, the first Python RTS approach that tracks fine-grained dependencies. NameRTS is built on an algorithm we call *name-based dependency propagation*. At its core, NameRTS models a Python project as a bipartite graph of name nodes and code element nodes. When a code element is implemented, it may reference other names that are not local variables, i.e., the names point to code elements defined outside the current one. We treat these references as *external names*. In the graph, definition edges connect each name node to the code elements defined under that name, and usage edges connect each code element to the external names it references. With this graph, NameRTS formulates test selection as a reachability problem. A test is selected if and only if a reachability analysis starting from its external names reaches any modified code element. This avoids relying on call-graph construction or type inference and handles Python's eager importing feature effectively by tracking dependencies at the function level rather than at the file level, giving higher precision while keeping the analysis lightweight.

A downside of a purely name-based approach is that it may trigger dependency cascades: a name may match many code elements that are not actually reachable by the test, and these elements introduce additional names that, in turn, pull in yet more elements. This recursive process can lead to an exponential expansion of reachable code elements, far beyond what the test could ever reach. To reduce this effect, NameRTS applies two pruning strategies. The first identifies names that tend to cause such expansions and prunes their code element nodes unless they were exercised by the test in previous executions. The second applies lightweight, context-aware name-element matching to rule out code elements that are invalid targets of a given name, using the defining file and class of the referencing code element to narrow the set of reachable code elements. Together, these pruning steps cut away paths with no evidence of being reachable by the test, improving precision with a negligible impact on safety in practice.

To evaluate NameRTS, we construct a dataset of 500 commits drawn from 10 open-source Python projects on GitHub, with code sizes ranging from 12k to 426k lines of code. For each commit, we collect the test files that depend on the changed elements by instrumenting the modified code and recording which tests invoke it at runtime, yielding the first Python RTS dataset with a ground truth. We compare NameRTS against the state-of-the-art Python RTS approach BabelRTS [42]. Since BabelRTS is not always safe, we additionally implement EkstaP, a safe file-level RTS baseline based on Ekstazi [26], which was originally proposed for Java. Our results show that NameRTS substantially outperforms both baselines. Across all projects, NameRTS skips 69.90% of test files and reduces end-to-end testing time by 45.59%. This corresponds to improvements of 842.3% and 1371.5% over EkstaP, and 146.5% and 107.7% over BabelRTS, respectively. In terms of safety, NameRTS selects all affected tests for 99.6% of commits, whereas BabelRTS is safe on only 76.6% of commits.

In summary, this paper makes the following contributions:

- We propose NameRTS, the first Python RTS approach based on fine-grained dependency analysis, built on a novel algorithm called *name-based dependency propagation*.
- We curate the first Python RTS dataset with ground truth, covering complex, real-world projects with up to 426k lines of code.
- We conduct an extensive empirical evaluation on this dataset, showing that NameRTS skips 69.90% of test files and reduces testing time by 45.59%, substantially outperforming state-of-the-art baselines while remaining safe on 99.6% of commits.
- We release an open-source package [10] to support reproducibility and future research.

2 Motivating Example

Figure 1 presents a project illustrating how NameRTS operates. The project contains five files. The test files `test_1.py` and `test_2.py` import the function `compute` from `module_B.py`. The function

module_A.py
```python
class A:
    def get_value(self, i):
        return i
```

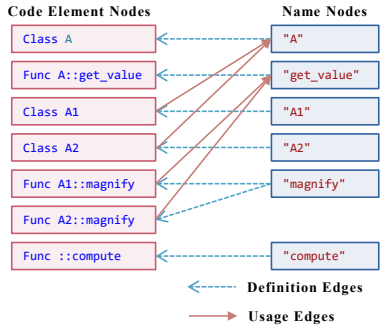
module_A_ext.py
```python
from module_A import A
class A1(A):
    def magnify(self, i):
        return self.get_value(i) + 10
class A2(A):
    def magnify(self, i):
        return self.get_value(i) * 10
```

module_B.py
```python
def compute(f, i):
    return f(i)
```

test_1.py
```python
from module_B import compute
from module_A_ext import A1
def test_1():
    a1 = A1()
    assert compute(a1.magnify, 1) == 11
```

test_2.py
```python
from module_B import compute
from module_A import A
def test_2():
    a = A()
    assert compute(a.get_value, 1) == 1
```

(a) Example project.



(b) Bipartite name-element graph.

```
init_test_1 = {"A1", "compute", "magnify"}
"A1"                -> class A1         -> "A"
"A"                 -> class A
"compute"           -> Func compute
"magnify" (+"A1")   -> Func A1::magnify -> "get_value"
"get_value" (+"A")  -> Func A::get_value

init_test_2 = {"A", "compute", "get_value"}
"A"                 -> class A
"compute"           -> Func compute
"get_value" (+"A")  -> Func A::get_value
```

(c) Propagation procedure for test files.

Fig. 1. Motivating example of name-based dependency propagation.

`compute` takes a callable and its input value, and invokes the callable with that input, representing a function-based callback pattern. In `test_1.py`, the test imports class `A1` from `module_A_ext.py`, constructs an instance, and passes its method `magnify` together with the integer `1` to `compute`. Class `A1` inherits from `A`, and `A1::magnify` invokes `A::get_value` through `self`, capturing typical object-oriented behavior and dynamic method dispatch. In contrast, `test_2.py` imports `A` directly and exercises only `A::get_value`. Since `magnify` has multiple implementations, i.e., in `A1` and `A2`, the example contains reachable and unreachable method definitions with the same name.

Call-graph-based RTS selects more tests than necessary in this scenario. For example, PyCG [47], a state-of-the-art static call graph builder for Python, treats the first argument of `compute` as a callable and therefore adds call edges from `compute` to multiple methods that are passed as arguments. In this example, the call graph contains edges from `compute` to both `A::get_value` and `A1::magnify`. This means a change to `A1::magnify` selects both `test_1.py` and `test_2.py` even though only `test_1.py` can reach that method at runtime. File-level RTS faces a similar issue. A change to `A2::magnify`, which no test executes, would force `test_1.py` to be selected, because it imports the modified file `module_A_ext.py`.

Our idea is to model this example project as a bipartite graph shown in Figure 1b. Code element nodes represent classes, including `A`, `A1` and `A2`, as well as functions, including `A::get_value`, `A1::magnify`, `A2::magnify`, and `compute`. Name nodes correspond to the names through which these code elements are referenced. Definition edges connect each name to the code elements defined under that name. Usage edges connect each code element to the external names it references. For example, the name "`A1`" is defined by class `A1`, and class `A1` references "`A`" as its superclass. These edges capture how definitions and name references are interrelated in the program. To determine which code elements `test_1.py` depends on, NameRTS starts from external names

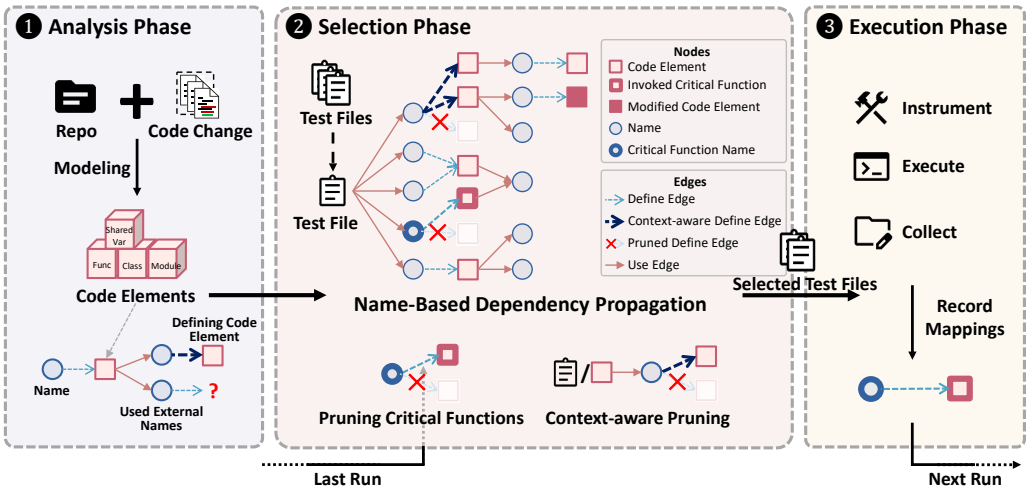


Fig. 2. Overview of NameRTS.

visible in the test and performs name-based dependency propagation. It follows definition edges to find candidate code elements for each name, then follows usage edges to extract newly referenced names, repeating until no new names are reached. Figure 1c shows the propagation steps. In this example, an attribute (e.g., `A1::magnify`) is reachable only when its defining class (e.g., `A1`) is reachable. The final reachable set consists of code elements that the test may use. A change to any element in this set, such as `A1::magnify`, causes `test_1.py` to be selected for re-execution. A change outside this set, such as `A2::magnify`, does not. Thus, this graph avoids the unnecessary selection that file-level RTS produces when any part of `module_A_ext.py` changes. In contrast to call-graph-based RTS, NameRTS does not select `test_2.py` when `A1::magnify` is modified.

This example illustrates three key advantages of name-based dependency propagation for Python RTS. First, the approach is lightweight and amenable to safety. Instead of precise name resolution, NameRTS relies on simple name matching and conservatively considers all possible definitions of a referenced name. Compared to call-graph construction, this over-approximate strategy avoids complex type inference, is easier to implement, and simplifies ensuring completeness and safety. Second, despite its simplicity, the approach enables effective test selection with fine-grained dependencies. Dependencies are tracked primarily at the function level, yielding finer granularity than file-level RTS. Moreover, the analysis is fast enough to be performed separately for each test file, avoiding the conflation of dependencies across different tests that occurs in call-graph-based RTS, as illustrated in the motivating example above. The same mechanism naturally extends to global variables, enabling variable-level dependency tracking and preventing unnecessary test selection caused by modifications to global variable definitions. Third, while name-based dependency propagation is conservative and may propagate spurious dependencies, resulting in unnecessary test selection, this limitation can be significantly mitigated through targeted pruning. NameRTS restricts propagation to code elements with evidence of reachability or contextual validity, which improves precision while affecting safety only in a negligible way in practice.

3 Approach

NameRTS is a regression test selection technique for Python that determines which tests may be affected by a given code change using *name-based dependency propagation*. For each test file,

NameRTS identifies whether it may reach any modified code element by propagating dependencies through a lightweight name-element graph. As illustrated in Figure 2, NameRTS operates in three phases: analysis, selection, and execution. The analysis phase extracts fine-grained code elements and their name dependencies, and maintains the metadata required for dependency propagation. The selection phase performs propagation for each test file and applies pruning strategies to eliminate paths with no evidence of reachability. The execution phase runs the selected tests and records runtime metadata for use in subsequent runs. Before applying NameRTS to a new commit, a one-time *initialization run* is required. This run applies all three phases and executes the full test suite to establish the metadata needed for *incremental runs* performed after every code change.

3.1 Used Metadata

NameRTS maintains metadata defining the dependency information needed for *name-based dependency propagation*. These metadata are incrementally updated across runs to capture fine-grained dependencies between names and code elements.

Code Elements. In the analysis phase, the target project is parsed into four types of code elements: `Module`, `Class`, `Function`, and `SharedVariable`. Each code element is represented as $e = \langle name, type, checksum, usedNames, file, optional(defClass)\rangle$, including its unqualified identifier (*name*), code element type (*type*), the hash of its bytecode (*checksum*), the set of external names it references (*usedNames*), and its defining file. For `Function` and `SharedVariable` elements, an additional attribute `defClass` specifies their defining class, if any. `SharedVariable` includes global variables and class static variables, which are initialized at import time and serve as shared state.

Name-element graph. To formalize name-based dependency propagation, NameRTS models a bipartite name-element graph consisting of *code element nodes* E, one for each code element derived above; and *name nodes* N, which are all *name* fields of code elements. Directed edges are defined as two sets, i.e., *definition edges* and *usage edges*. Definition edges $Def = \{ (n, e) \mid n \in N \land e \in E \land e.name = n \}$ map each name to code elements defined under that name, indicating that resolving name n may yield code element e. Usage edges $Use = \{ (e, n) \mid n \in N \land e \in E \land n \in e.usedNames \}$ capture the external names referenced by each code element, indicating that e depends on the definition of n. Propagation over these edges captures how names lead to code elements and how code elements introduce further name dependencies.

Import Graph. The import graph is a mapping $IG : s \mapsto \{ s' \mid s' \text{ is imported in } s \}$ from each source file s to the set of files it directly imports. This information is used to resolve transitive module dependencies.

Accessed Names. NameRTS maintains a mapping $AN : t \mapsto \{ n \mid n \in N \}$ that associates each test file t with the set of names it accesses at runtime, including those obtained indirectly (e.g., via reflection). This metadata captures implicit runtime dependencies not visible in static analysis.

3.2 Analysis Phase

The goal of the analysis phase is to construct the bipartite name-element graph described in Section 3.1. This includes deriving code elements, identifying the external names they reference, and establishing the definition and usage edges that connect code element nodes and name nodes. In addition, the analysis phase determines the set of code elements visible during test execution for each test file, and detects modified code elements for later use.

Constructing code elements. To extract the four types of code elements, NameRTS performs a source-level static analysis.

Extracting used external names. The extraction process of used external names is shared across all four code element types. NameRTS extracts these names by compiling the project and inspecting the Python bytecode. For each snippet from which a code element is constructed, NameRTS identifies the corresponding bytecode range by mapping bytecode instructions to source line numbers and selecting those whose line numbers fall within the snippet. It then reads identifier operands from the instructions flagged by `hasname` [3] in Python's `opcode` module, which reference entries in the `co_names` table. This bytecode-based approach avoids the need to trace variable definitions to determine whether an accessed identifier refers to a local variable or an external element, as local variables are stored separately in the `co_varnames` table and are therefore naturally excluded [1].

Constructing `Function` elements. The analysis constructs a `Function` code element for each top-level function or method definition and extracts its external names using the bytecode-based procedure described above. Any function or class defined inside a function body is treated as part of that enclosing function and does not form a separate code element.

Constructing `Class` elements. For each class definition, including its inheritance clause but excluding method definitions and static class variable definitions, the analysis constructs a `Class` code element. The used external names of a `Class` element primarily include its superclass names, enabling NameRTS to determine when members inherited from those superclasses may be reachable.

Constructing `Module` elements. The analysis constructs `Module` code elements to capture import-time behavior with side effects that may influence test execution. For each module, the used external names are extracted from all top-level statements executed at import time (including compound constructs, such as `if`, `for`, or `try`) that contain function calls whose return values are not consumed. Such statements are treated as contributing to import-time side effects. For example, in Figure 3, lines 1–2 form a `Module` element, yielding the external names $c1$ and $func$.

Constructing `SharedVariable` elements. Global variables and class static variables are initialized at import time and constitute shared state that may be accessed across multiple contexts. The analysis constructs a `SharedVariable` code element for each such variable, extracting used external names from top-level statements executed at import time that define or update that variable. For example, in Figure 3, lines 3–6 define the global variable a and form its `SharedVariable` element, yielding the external names $c2$, $A1$, and $A2$.

```
1 if c1:
2     func()
3 if c2:
4     a = A1()
5 else:
6     a = A2()
```

Fig. 3. Example Code with `Module` and `SharedVariable` elements

Each constructed code element becomes a node in the name-element graph. A definition edge conceptually connects the code element to the name under which it is defined, and its extracted external names give rise to usage edges to the corresponding name nodes. The resulting conceptual graph forms the basis for name-based dependency propagation in the selection phase.

Determining visible code elements for each test file. Since a test can only reach code elements defined in files it imports, we use file-level visibility as an initial filter to eliminate provably irrelevant code elements. NameRTS constructs the import graph IG by statically analyzing import statements in source files to extract file-level import relationships. To handle Python's import semantics, when a submodule is imported, NameRTS also records the `__init__.py` files of all its parent packages as imported files. Given this graph, NameRTS computes, for each test file, the set of source files it transitively depends on by performing a graph traversal starting from the test file. From this set, NameRTS derives $Visible[t]$, the set of code elements whose defining files are reachable from t in the import graph and are therefore visible during the execution of t.

Detecting modified code elements. To detect modified code elements, NameRTS computes a checksum for each code element from its Python bytecode and compares it against the previously

Algorithm 1: Name-based Dependency Propagation

Input: $\mathcal{T}$: set of all test files;
AN: Accessed Names metadata;
Def: set of definition edges (n, e);
Use: set of usage edges (e, n);
M: set of modified code elements;
$Visible$: mapping $t \mapsto$ visible code elements.
Output: $\mathcal{T}'$: set of affected test files.

1 $\mathcal{T}' \leftarrow \emptyset$
2 **foreach** $t \in \mathcal{T}$ **do**
3 *reachable_names* $\leftarrow$ external names used in t
4 $Modules_t \leftarrow \{ m \in Visible[t] \mid m.type = \text{MODULE} \}$
5 **foreach** $m \in Modules_t$ **do**
6 *reachable_names* $\cup =$ $m.usedNames$
7 *reachable_names* $\cup =$ $AN[t]$
8 *reachable_elements* $\leftarrow \emptyset$
9 *work_stack* $\leftarrow$ *reachable_names*
10 **repeat**
11 **while** *work_stack not empty* **do**
12 $n \leftarrow$ *pop(work_stack)*
13 $E \leftarrow \{ e \mid (n, e) \in Def \wedge e \in Visible[t] \}$
14 $E \leftarrow \{ e \in E \mid e.class = null \vee e.class \in reachable_names \}$
15 prune E with Algorithm 2
16 *reachable_elements* $\cup =$ E
17 **foreach** $e \in E$ **do**
18 $names \leftarrow \{ n' \mid (e, n') \in Use \}$
19 add new *names* to *reachable_names* and *work_stack*
20 refill *work_stack* with names of `Function` and `SharedVariable` elements defined in classes
21 **until** *no new names are added*
22 **if** $\exists e \in$ *reachable_elements s.t.* $e \in M$ **then**
23 $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{t\}$
24 **return** $\mathcal{T}'$

Algorithm 2: Code Element Pruning

Input: n: name being propagated;
$type(n)$: name type (*non-attr*, *sure-attr*, or *amb-attr*);
ctx: optional context (defining file or class);
E: candidate code element nodes associated with n;
IC: Invoked Critical Functions;
reachable_names: current set of reachable name nodes.
Output: E: pruned set of code element nodes.

1 **if** n *corresponds to a critical function* **then**
2 $E \cap = IC[t]$
3 **if** $type(n) =$ NON-ATTR **then**
4 **if** *definition of* n *exists in* $ctx.file$ **then**
5 **return** *definition of* n
6 **else**
7 $def_n \leftarrow$ explicit import target of n
8 **return** def_n *if exists, otherwise* E
9 **if** $type(n) =$ SURE-ATTR **then**
10 **return** $\{ e \in E \mid e.class \in hierarchy(ctx.class) \}$
11 **return** E

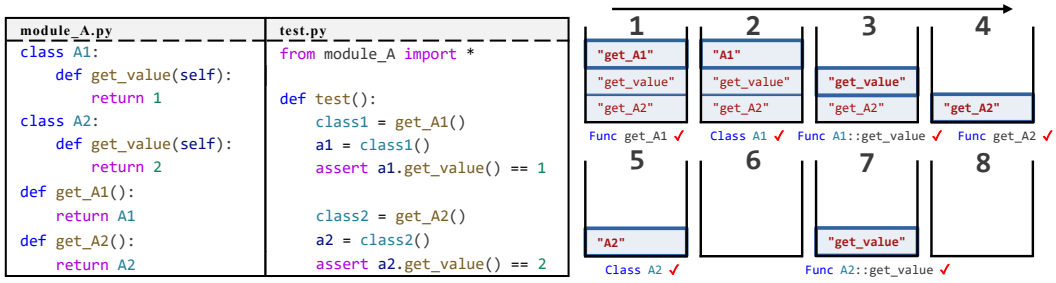
stored checksum. Before computing the checksum, NameRTS removes or normalizes non-semantic information in the bytecode that does not reflect actual code changes, such as virtual memory addresses or other compilation-specific metadata. Since comments are not preserved in Python bytecode, they naturally do not affect the computed checksums. The modified set M contains all code elements whose checksums have changed or are not present in the previous run.

The analysis phase runs during both the initialization run and the incremental runs. In the initialization run, it analyzes all source files to construct code elements and their metadata, which are cached. In the incremental runs, it loads the cached results, re-analyzes only modified files, and updates the cache accordingly. Modified code elements are identified only during subsequent runs.

3.3 Selection Phase

In the selection phase, NameRTS performs name-based dependency propagation for each test file with respect to the current code change, realized as a reachability computation over the bipartite name-element graph described in Section 3.1. A test file is selected for re-execution if it can reach any modified code element node. Algorithm 1 outlines the overall procedure.

For each test file t, NameRTS first initializes the set *reachable_names* with all external names referenced by code elements defined in t (line 3). It then incorporates import-time behavior by



(a) Example project. (b) Evolution of the *work_stack*.

Fig. 4. Example of name-based dependency propagation.

adding the external names referenced by the `Module` elements visible to t (line 4–6). The set is further augmented using *Accessed Names* metadata, which records names implicitly accessed during previous executions of t (line 7). These names form the initial frontier of reachable name nodes.

After initialization, NameRTS iteratively expands *reachable_names* until a fixed point is reached (lines 8-20). A working stack, *work_stack*, is initialized with all current name nodes in *reachable_names* (line 9). For each name node n popped from the stack, NameRTS follows definition edges by collecting all code element nodes e such that $(n, e) \in Def$ and e is visible to t (line 12-13). The resulting candidates are then pruned. Specifically, NameRTS filters out code elements whose defining class is not present in the *reachable_names* set (line 14). A class outside *reachable_names* is considered inaccessible, implying that its attributes are not reachable. The remaining code element nodes are recorded as reachable (line 16). For each recorded code element node e, NameRTS follows usage edges by adding newly reached name nodes n' such that $(e, n') \in Use$ to both *reachable_names* and *work_stack* (line 17–19). This alternation of traversing definition and usage edges continues until *work_stack* becomes empty. Because new class names may appear during propagation, NameRTS then refills the *work_stack* with the names of `Function` and `SharedVariable` elements defined in classes to ensure that attribute-related code elements depending on newly reachable classes are not prematurely excluded (line 20). If no new names are added during this iteration, propagation has converged. Finally, if any recorded reachable code element node belongs to the modified set M, the corresponding test file is marked as affected and selected for execution (lines 22–23).

Figure 4 illustrates the propagation procedure using an example. Figure 4a shows the example project, while Figure 4b depicts the evolution of the *work_stack*, which maintains the frontier of name nodes to be processed. Initially (state 1), the *work_stack* contains the external names referenced by `test.py`, namely `get_A1`, `get_value`, and `get_A2`. When `get_A1` is popped, NameRTS resolves it to the function `get_A1`, whose body references the class name `A1`. Thus `A1` is pushed to the stack (state 1-2). Subsequent steps proceed similarly. Notably, when name `get_value` is processed in state 3-4, only `A1::get_value` is considered reachable, since class name `A2` is not yet accessible and `A2`'s attributes are therefore pruned. After `A2` becomes reachable and the stack is exhausted (state 6), NameRTS refills the *work_stack* with attribute names, causing name `get_value` to be reconsidered and `A2::get_value` to be correctly included (states 7–8).

3.4 Execution Phase

In the execution phase, NameRTS runs test files and collects runtime metadata to support future test selection. During the initialization run, the full test suite is executed, whereas in incremental runs only the selected tests are executed.

Augmenting Accessed Names via Dynamic Analysis. NameRTS assumes that if the Python interpreter can access a name during test execution, its definition may be used. However, not all uses appear as direct name accesses, which might cause the approach to miss some dependencies. We address two types of indirect name accesses by dynamically analyzing the executing regression tests: operator overloading and reflection. For operator overloading, rather than matching every possible overloaded operator, NameRTS records all operator-related names (e.g., `__add__`) under a special key "*" in a map AN (for "accessed names"), meaning they are shared by all test files. For reflection, NameRTS focuses on introspection, which inspects program structure or object attributes at runtime, e.g., `builtins.getattr`. The approach monkey-patches common built-in introspection functions to identify accessed names based on their parameters or return values, and adds these names to $AN[t]$ for the currently executing test file. Names accessed during module import are likewise recorded under the shared key "*", so they are treated as reachable to all test files. Dynamic code execution through `eval` or similar mechanisms is not tracked, as it is rare and prohibitively expensive to analyze safely [11, 38].

Dynamic Import Events. While standard `import` statements are analyzed in the analysis phase to construct the import graph IG, imports performed through the `importlib` API require dynamic handling, as the module name is often stored in a variable whose value may be constructed at runtime and is hard to resolve statically. NameRTS intercepts imports triggered through the `importlib` API by monkey-patching `importlib.import_module`, and augments IG with the captured dynamic import events. These dynamically observed imports are recorded and incorporated into IG in the analysis phase during all future incremental runs.

3.5 Pruning Mechanisms

The approach explained so far conservatively assumes that any definition with a specific name is a dependency. In practice, this conservative approach often causes the set of reachable code elements to grow far beyond what a test can actually reach. A single name node may pull in large portions of the graph, which recursively trigger further expansions. This dependency cascade inflates the reachable code element set and reduces the precision of selection. To control this effect, NameRTS integrates two pruning mechanisms into the dependency propagation procedure. Both refine the candidate set of code element nodes that are considered valid successors of a name node and discard paths that have no evidence of being reachable in a given test.

Pruning critical functions. Some name nodes correspond to many code element nodes. For example, a single method name may correspond to method implementations in many different classes. When propagation reaches such a name node, it may induce a large dependency cascade that is unlikely to reflect the behavior of an individual test. To mitigate this effect, NameRTS identifies functions whose name nodes possibly lead to many other nodes and requires runtime evidence before the propagation is allowed to proceed through them.

Critical functions are identified during the initialization run, where NameRTS performs dependency propagation for all tests. For each name node, NameRTS aggregates outgoing name nodes reachable through all of its candidate code elements. Names with the largest aggregated expansions are primary sources of dependency cascades. NameRTS selects the top N such names as *critical function names*, where N is a configurable parameter. All functions defined under these names are treated as *critical*. In the initialization and subsequent runs, before executing tests, NameRTS inserts lightweight instrumentation at the entry points of all critical functions. Each instrumentation probe records the first invocation of the function during the execution of a test file, and ignores subsequent invocations, ensuring negligible runtime overhead. If a critical function is called during module import, the probe records the event under the shared key "$*$". In the selection phase, when

propagation encounters a critical function, NameRTS prunes the corresponding code element node unless an invocation was observed for the current test (line 1-2 in Algorithm 2). We preserve two exceptions and never prune these functions even without runtime evidence: newly added functions and methods whose overriding implementations in subclasses were removed, since both can introduce new invocations that were not observed in prior executions.

Pruning with context-aware name-element matching. The second pruning mechanism refines dependency propagation by exploiting the context in which names are used. For many used external names, contextual information enables more precise name-element matching than trivial name-based matching, which prevents propagation to spurious defining code element nodes.

During the analysis phase, NameRTS classifies each used external name into one of three categories based on its bytecode-level access pattern:

- *non-attr:* names accessed without a dot operation (i.e., not via `LOAD_ATTR`). These are typically explicitly imported or defined in the current module, making definitions straightforward to resolve.
- *sure-attr:* attribute accesses on `self` or `cls`. Their definitions can be reliably resolved within the enclosing class, its superclasses, or its subclasses.
- *amb-attr:* attribute accesses on objects other than `self` or `cls`. These may refer to attributes of arbitrary objects, including modules, and are therefore *ambiguous* in origin.

In the selection phase, when a name node becomes reachable, NameRTS records the context of the code element node that produced it, including its defining file $ctx.file$ and, if any, its defining class $ctx.class$. Pruning proceeds according to the name category. For *non-attr* names, NameRTS attempts to locate its definition within the same file $ctx.file$ (Algorithm 2, lines 5-6). If none is found, it traces explicit import statements to locate the imported definition. When such a definition is found, only that code element node is retained. Otherwise, all candidate elements are kept to ensure safety (Algorithm 2, lines 8–9). For *sure-attr* names, NameRTS retains code element nodes whose defining classes lie in the hierarchy of the referencing class $ctx.class$ (Algorithm 2, line 11). These matching steps can be precomputed to avoid repeated lookups during propagation. For *amb-attr* names, NameRTS preserves all candidate code element nodes to maintain safety.

3.6 Special Handling of Decorators

Decorators in Python are functions executed at definition time, wrapping a class or function and binding the returned object to the original name [2]. Since decorators are typically executed during module import, they need to be handled with care. NameRTS distinguishes between two kinds of decorators. *Functional decorators* modify or extend the behavior of the decorated object. They are treated as external names used by the decorated function or class, since their effects manifest only when the decorated object is invoked. *Registry decorators*, by contrast, register the decorated object for later use. NameRTS identifies such decorators heuristically by matching configurable keywords commonly used in registration patterns (e.g., `register`, `router`). Because such decorators determine when and how the registered objects are invoked, their behavior is difficult to fully capture through static analysis. To remain safe, for classes decorated in this way, NameRTS conservatively adds them to $AN[*]$. For functions, NameRTS instruments them and monitors their execution during test runs. If a decorated function is executed by a test t, it is added to $AN[t]$. Finally, if a registry decorator itself is modified or newly introduced, all test files are marked as affected, as the registration behavior of the system may have changed.

Table 1. Dataset summary. Test Files is the average number of test files, Test Time is the full-suite execution time. NNodes and CENodes are the numbers of name nodes and code element nodes, respectively.

Project	Head	kLoC	Test Files	Test Time(s)	NNodes(k)	CENodes(k)
sympy	d854b09	426	611.4	1391.1	24.4	60.2
sklearn	cc526ee	230	243.7	725.5	11.3	41.3
matplotlib	d05b43d	156	102.0	431.0	10.0	39.0
dask	8639b6e	123	163.0	867.9	6.9	16.8
xarray	97fb90b	112	68.3	726.2	6.3	12.7
sphinx	eda953e	106	139.2	133.0	5.2	14.5
pylint	d17bb06	74	81.0	102.7	2.9	7.1
seaborn	2386036	35	34.1	562.2	2.6	5.6
pvlib	b4916e1	28	54.7	56.6	2.2	5.5
loguru	940b7cf	12	48.2	76.9	1.0	1.7
Avg.		130	154.6	507.3	7.3	20.4

4 Evaluation

To evaluate NameRTS, we investigate the following research questions:

- **RQ1:** How effective is NameRTS in reducing tests and testing time while remaining safe?
- **RQ2:** What is the computational overhead introduced by NameRTS?
- **RQ3:** How much do the two pruning mechanisms contribute to the effectiveness?
- **RQ4:** How does the choice of the parameter N for identifying critical functions affect effectiveness?

4.1 Experimental Setup

Dataset construction. Python projects can grow to hundreds of thousands of lines of code, yet prior RTS studies for Python consider only projects up to 60k lines [32, 42]. To provide a more comprehensive assessment of NameRTS, we curate a new dataset of 10 open-source Python projects. We first define the following selection criteria: (i) Python is the primary language of the project; (ii) the project is compatible with Pytest; (iii) executing the full test suite requires more than 30 seconds; and (iv) the project contains more than 20 test files. Network-dependent projects are excluded because their test execution times tend to be unstable. Based on these criteria, we first consider the projects used by the SWE-bench benchmark [31, 52], which represent widely used, actively maintained Python systems, and identify seven projects that satisfy all criteria. Then, to increase diversity, we randomly sample additional projects from GitHub repositories with more than 1,000 stars and more than 500 commits, retaining the first three sampled projects that meet the same criteria. These projects are summarized in Table 1. They contain, on average, 130k lines of Python code, with sizes ranging from 12k to 426k. For each project, we collect 50 consecutive commits, following the setup used in prior work [57]. Each commit must build successfully, as a commit that fails to build cannot execute any tests. We also require that each commit modifies at least one Python source file. Commits that only touch test files or non-Python files are filtered out, ensuring that our evaluation focuses on source-code changes relevant to RTS. The complete list of selected commit hashes is available in our replication package [10].

Ground truth extraction. To compute a ground truth for test selection, we instrument every modified function and execute each test file to determine whether the instrumented functions are invoked. For modified global variables and class static variables, we use a language server to trace their references until functions are reached and then instrument these functions. A language server is suitable here because ground truth extraction requires only a one-time, precise resolution of

actual usages. Although individual language server queries are fast, applying such analysis in RTS would require resolving call targets or references at a large number of program locations on every commit, resulting in prohibitive cumulative overhead. Therefore, we use the language server only for offline ground truth collection. One complication is that a function may execute during module import for initialization. Because of Python's eager importing feature, such functions often run even when the test never uses their initialized state. To avoid such false positives, we manually inspect invocations that occur only during import and not during test execution, and discard test files that do not actually use the initialized states. To the best of our knowledge, this is the first Python RTS dataset that provides a ground truth.

Used metrics. RTS evaluation focuses on effectiveness and safety. Following prior work [42], effectiveness is evaluated along two dimensions. *Test reduction* measures how many test files are skipped on average, computed as TestR = $(T_{\text{all}} - T_{\text{sel}})/T_{\text{all}}$, where T_{all} and T_{sel} denote the total and selected test files across all commits. As a complementary metric to test reduction, we also measure *precision*, which quantifies how many selected test files are actually affected according to the ground truth, computed as Precision = $T_{\text{aff}}/T_{\text{sel}}$, where T_{aff} denotes the number of selected test files that are affected. Moreover, *time reduction* measures savings in end-to-end testing time, including initialization (if any), analysis, selection, and test execution, computed as TimeR = $(t_{\text{orig}} - t_{\text{rts}})/t_{\text{orig}}$, where t_{orig} and t_{rts} denote the cumulative testing time without and with RTS. Safety is evaluated using *safe rate*, which measures the fraction of commits for which all tests that are affected according to the ground truth are selected, computed as SafeR = $C_{\text{safe}}/C_{\text{total}}$, where C_{safe} and C_{total} denote the number of safe commits and the total number of commits.

Implementation details. Pytest fixtures are injected into test functions through parameters, which are treated as local variables and therefore do not appear in *usedNames*. This would cause NameRTS to miss dependencies introduced by fixtures. To avoid this, we add the names of injected fixtures to the *usedNames* of the receiving functions so they participate in dependency propagation. NameRTS monitors the invocation of critical functions during test execution, while we observe that shared global state may cause missing invocations. For example, one test may initialize a global variable, preventing subsequent tests from invoking the same initialization code. To avoid such interference, we implement isolated test execution atop `pytest-xdist`: each test file runs in a separate process while preserving overall serial order. This yields consistent runtime metadata and safe dependency tracking. One project, *pylint*, is incompatible with `pytest-xdist`, so isolation is disabled for it. Our results show that disabling isolation for *pylint* does not lead to substantial safety loss. Isolation is enabled only for NameRTS and disabled for all baselines. Unless otherwise noted, the parameter N for identifying critical functions is set to 500.

4.2 RQ1: Effectiveness and Safety

Approach. We evaluate NameRTS against two baselines: BabelRTS [42] and EkstaP. Existing RTS techniques can be categorized into file-level and function-level approaches [54]. BabelRTS is a file-level RTS technique and represents the state of the art for Python. It relies on static file-level dependencies but ignores implicit imports that may affect program behavior, which leads to unsafe test selection. To compare NameRTS with a safe file-level RTS, we implement an additional baseline, *EkstaP*, following the file-level RTS paradigm exemplified by Ekstazi [26] for Java. EkstaP reuses the import graph infrastructure of NameRTS and accounts for implicit parent-package imports and dynamic imports via `importlib`, as described in Section 3.4. Rather than faithfully reproducing Ekstazi in Python, EkstaP is designed to provide a strong and safe file-level RTS baseline with as much reduction as possible while preserving safety. We do not include pytest-rts [32], a coverage-based RTS technique that selects test functions based on previously observed coverage, because

Table 2. Comparison with baseline RTS techniques (RQ1). GT denotes the test reduction achieved by the ground truth for each project.

Project	NameRTS				EkstaP				BabelRTS				GT
	TestR	Precision	TimeR	SafeR	TestR	Precision	TimeR	SafeR	TestR	Precision	TimeR	SafeR	TestR
sympy	73.51%	45.47%	32.66%	100.0%	0.00%	12.05%	-3.08%	100.0%	12.86%	12.32%	14.16%	88.0%	87.95%
sklearn	89.98%	84.52%	66.81%	100.0%	15.83%	10.06%	12.42%	100.0%	15.74%	10.05%	13.52%	100.0%	91.53%
matplotlib	54.43%	47.98%	37.01%	100.0%	0.00%	21.86%	-3.52%	100.0%	97.63%	46.28%	95.71%	14.0%	78.14%
dask	51.82%	63.69%	29.15%	100.0%	0.00%	30.69%	-3.34%	100.0%	6.70%	32.35%	2.51%	98.0%	69.31%
xarray	66.28%	77.06%	49.11%	100.0%	3.93%	27.05%	0.52%	100.0%	5.74%	26.80%	6.08%	66.0%	74.01%
sphinx	50.47%	47.35%	9.89%	98.0%	2.00%	23.94%	-3.15%	100.0%	10.37%	26.02%	6.36%	80.0%	76.54%
pylint	79.75%	51.10%	30.30%	98.0%	6.00%	11.06%	-1.54%	100.0%	60.84%	16.65%	21.76%	48.0%	89.60%
seaborn	82.88%	98.29%	75.68%	100.0%	26.55%	22.91%	20.73%	100.0%	11.96%	19.04%	5.99%	98.0%	83.18%
pvlib	94.33%	61.29%	78.59%	100.0%	9.90%	3.85%	5.47%	100.0%	46.24%	6.39%	41.79%	98.0%	96.53%
loguru	55.56%	89.72%	46.66%	100.0%	9.97%	44.28%	6.47%	100.0%	15.49%	46.58%	11.60%	76.0%	60.13%
Avg.	**69.90%**	**66.65%**	**45.59%**	99.6%	7.42%	20.77%	3.10%	100.0%	28.36%	24.25%	21.95%	76.6%	80.69%

our ground truth and safety metric are defined at the test-file level, preventing a fair comparison. Moreover, prior work [42] shows that BabelRTS achieves substantially better performance than pytest-rts, so we focus on BabelRTS as a stronger baseline. At present, there is no function-level RTS technique specifically designed for Python. Function-level RTS techniques developed for other languages rely on call graph construction. In Python, call graph analysis suffers from scalability and precision limitations [17], making porting such techniques non-trivial and unreliable.

Results. Table 2 reports the performance of NameRTS compared with the baselines. On average, NameRTS skips 69.90% of test files and reduces end-to-end testing time by 45.59%, achieving 86.63% of the maximum test reduction indicated by the ground truth (69.90% out of 80.69%), while attaining a precision of 66.65%, substantially higher than EkstaP (20.77%) and BabelRTS (24.25%). Although the precision is not perfect, only 19.31% of test files are affected by a typical code change on average according to the ground truth, allowing NameRTS to still achieve test reduction close to the ground truth. NameRTS's test reductions hold across projects of very different scales. For example, it eliminates 73.51% of test files in *sympy* (426k LoC) and 94.33% in *pvlib* (28k LoC). Several projects show more modest reductions, yet NameRTS still removes more than half of their tests. A key factor affecting the effectiveness of NameRTS on these projects is that certain critical functions execute during module import. For such functions, NameRTS must conservatively assume that any test importing the defining file may access the functions. This effect is most visible when changes occur inside global code that includes side-effecting calls. In *matplotlib*, for instance, five out of fifty commits modify the function `_get_executable_info`, which is executed when `matplotlib/testing/compare.py` is imported. For these commits, NameRTS eventually selects all tests.

Regarding safety, NameRTS is safe for 99.6% of all commits, i.e., it is unsafe for only two commits. One unsafe commit appears in *pylint*[1], where two test files are missed. This is because shared state left in the global `astroid.MANAGER` cache by an earlier test causes the resolution logic to short-circuit and never reach the critical function `AggregationsHandler.handle`. The code element node is incorrectly pruned, preventing dependency propagation from reaching the modified function. This test interference occurs only because our isolator cannot be applied to *pylint*, as discussed in Section 4.1. NameRTS also misses one test in *sphinx*[2]. Specifically, *sphinx* and *docutils* interact through a callback protocol, in which *docutils* explicitly invokes `dispatch_visit` defined in *sphinx*.

[1]https://github.com/pylint-dev/pylint/commit/7d6f3f230e038eabee2efc75628329fe74aa943e
[2]https://github.com/sphinx-doc/sphinx/commit/c76c2ad63c172e5ebab8bdb965286cfde5ca4491

Because this invocation is hard-coded in the external library, i.e., *docutils*, and NameRTS does not analyze library code, the callback cannot be inferred.

BabelRTS skips 28.36% of test files and yields a 21.95% reduction in end-to-end testing time. Its safety, however, is inconsistent. For 6 of the 10 projects, the Safe Rate falls below 90%. The most common safety issues stem from BabelRTS ignoring implicit parent package imports. The safety degradation is most pronounced in *matplotlib* and *pylint*, where BabelRTS achieves high test reduction (97.63% and 60.84%) but attains safe rates of only 14.00% and 48.00%. In *matplotlib*, the unsafety is caused by BabelRTS assuming that package sources are located under the project root, but *matplotlib* places its code under `lib/matplotlib`, leading to missed dependencies. In *pylint*, unsafety instead arises from extensive runtime plugin loading via *importlib*, which BabelRTS's static-only analysis cannot capture. EkstaP, which incorporates these missing dependency edges and handles implicit and dynamic imports safely, achieves 7.42% test reduction and 3.10% time reduction. For 3 of the 10 projects, EkstaP cannot skip any test. By analyzing EkstaP's import graph, we find that Python's eager importing causes severe inflation of file-level dependencies: on average, a single test file imports 83.27% of source files, and 80.92% of source files are imported by every test file. As a result, file-level RTS leaves little opportunity for meaningful test reduction, and the overhead of EkstaP's required initialization run to capture dynamic imports can outweigh its benefits, leading to limited or even negative time reduction. These results highlight that while file-level RTS performs well in languages like Java [26, 35], it is substantially less effective in Python.

Answer to RQ1: NameRTS skips 69.90% of test files, reduces testing time by 45.59%, and is safe on 99.6% of commits. BabelRTS is often unsafe, while EkstaP remains safe but reduces only 7.42% of tests and 3.10% of time. Compared with this safe, file-level baseline, NameRTS provides an order of magnitude higher test and time reductions.

4.3 RQ2: Computational Overhead

Approach. NameRTS introduces overhead during analysis, selection, and execution. RQ2 quantifies this overhead by breaking it down into three components: *initialization overhead*, which refers to the one-time cost of the initialization run; *selection overhead*, which measures the cost of identifying affected tests on each commit and covers the analysis and selection phases; and *runtime overhead*, which captures the additional execution time introduced by instrumentation, such as probes inserted into critical functions. Together with actual test execution time, these components form the end-to-end testing cost when NameRTS is enabled. To measure the runtime overhead introduced by NameRTS, we record test execution time with and without instrumentation for the same selected tests, and take their difference as the runtime overhead.

Results. Table 3 reports the overhead introduced by NameRTS. The *RunAll* column shows the time required to execute the full test suite across 50 commits. Under the *Time* column, the *Test*, *Runtime*, *Select*, and *Init* subcolumns denote the actual test execution time and the runtime, selection, and initialization overhead, each as a percentage of *RunAll*. The *Selected* column reports the percentage of tests selected by NameRTS. On average, NameRTS introduces 10.04% overhead relative to RunAll, consisting of 4.45% runtime overhead, 3.22% selection overhead, and 2.38% initialization overhead. Initialization overhead remains stable at roughly 2% across projects. This is expected because initialization includes one full test-suite execution, while RunAll includes fifty. For relatively simpler projects, such as *seaborn* and *loguru*, the cost of static analysis is negligible, so initialization overhead is dominated by the test execution. Selection overhead varies with project scale and the relative cost of test execution. Large and structurally complex projects incur higher selection overhead. For instance, *sympy*, the largest project in our dataset with 426k lines of code, has one of the highest selection overheads at 7.43%. *sphinx* and *xarray* have similar code sizes (106k

Table 3. Breakdown of the overhead of NameRTS relative to full test-suite execution (RQ2).

Project	RunAll(s)	Time (% of RunAll)				
		Test	Runtime	Select	Init	Total
sympy	69555.2	46.22%	10.64%	7.43%	3.05%	67.34%
sklearn	36277.0	27.04%	1.31%	2.33%	2.51%	33.19%
mpl	21548.0	53.83%	3.92%	2.95%	2.30%	63.00%
dask	43395.8	62.73%	4.32%	1.58%	2.22%	70.85%
xarray	36311.8	41.87%	5.72%	1.04%	2.25%	50.88%
sphinx	6647.6	66.95%	12.15%	8.24%	2.77%	90.11%
pylint	5132.8	63.12%	1.02%	3.35%	2.21%	69.70%
seaborn	28110.5	21.72%	0.20%	0.43%	1.96%	24.31%
pvlib	2832.3	12.17%	3.56%	3.16%	2.52%	21.41%
loguru	3845.3	48.07%	1.65%	1.66%	1.96%	53.34%
Avg.	25365.6	44.37%	4.45%	3.22%	2.38%	54.41%

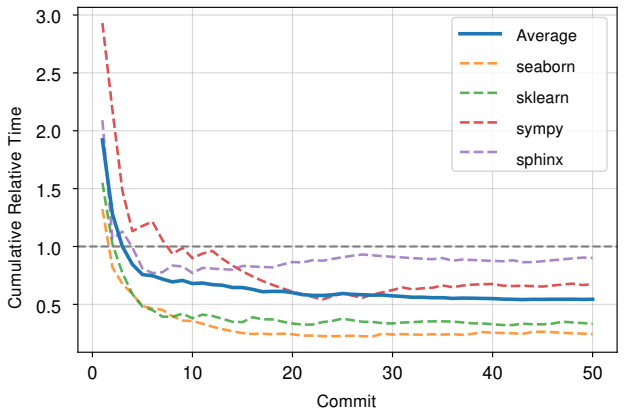


Fig. 5. Cumulative Relative Testing Time

and 112k LoC), yet *sphinx* has far faster tests (6647.6s vs. 36311.8s), making its selection overhead proportionally larger (8.24% vs. 1.04%). Runtime overhead originates primarily from instrumentation, in particular the probes inserted into critical functions. Projects whose tests invoke certain functions intensively incur higher runtime overhead. For example, *sympy* and *xarray* are computation-heavy mathematical libraries and show runtime overheads of 10.64% and 5.72%. In contrast, for projects dominated by lightweight operations, runtime overhead is negligible. For instance, *seaborn* shows only 0.2% runtime overhead.

To understand at what point in time enabling NameRTS pays off, Figure 5 shows the *cumulative relative testing time*, i.e., the ratio between testing time incurred up to a given commit under NameRTS and the time required to run all tests for the same set of commits. We present four representative projects: two where NameRTS achieves strong reductions (*seaborn* and *sklearn*) and two where the gains are comparatively smaller (*sympy* and *sphinx*). The *Average* curve denotes the mean across all ten projects. Overall, cumulative relative testing time falls below 1.0 within five commits on average, and within ten commits even for projects where NameRTS provides more modest benefits, meaning that NameRTS 's benefits outweigh its initialization overhead quickly.

Table 4. Ablation of pruning mechanisms in NameRTS (RQ3).

Project	NameRTS		w/o CF		w/o NEM		w/o CF&NEM	
	TestR	TimeR	TestR	TimeR	TestR	TimeR	TestR	TimeR
sympy	73.51%	32.66%	40.99%	20.59%	73.42%	28.15%	15.93%	-19.29%
sklearn	89.98%	66.81%	76.06%	57.59%	89.92%	65.49%	75.92%	55.63%
mpl	54.43%	37.01%	32.04%	18.24%	47.55%	29.37%	30.10%	15.98%
dask	51.82%	29.15%	14.15%	1.70%	48.80%	21.09%	8.44%	-6.22%
xarray	66.28%	49.11%	37.45%	25.99%	65.34%	46.76%	33.55%	23.15%
sphinx	50.47%	9.89%	9.99%	-16.53%	48.10%	6.08%	9.93%	-20.84%
pylint	79.75%	30.30%	67.21%	16.59%	79.41%	28.90%	66.86%	15.10%
seaborn	82.88%	75.68%	81.18%	75.53%	82.88%	75.32%	80.01%	74.20%
pvlib	94.33%	78.59%	93.86%	78.30%	92.43%	77.85%	91.81%	78.49%
loguru	55.56%	46.66%	31.69%	26.66%	55.48%	45.89%	31.60%	26.53%
avg.	**69.90%**	**45.59%**	48.46%	30.47%	68.33%	42.49%	44.42%	24.27%
Δ vs. NameRTS			30.67%↓	33.17%↓	2.24%↓	6.79%↓	36.46%↓	46.75%↓

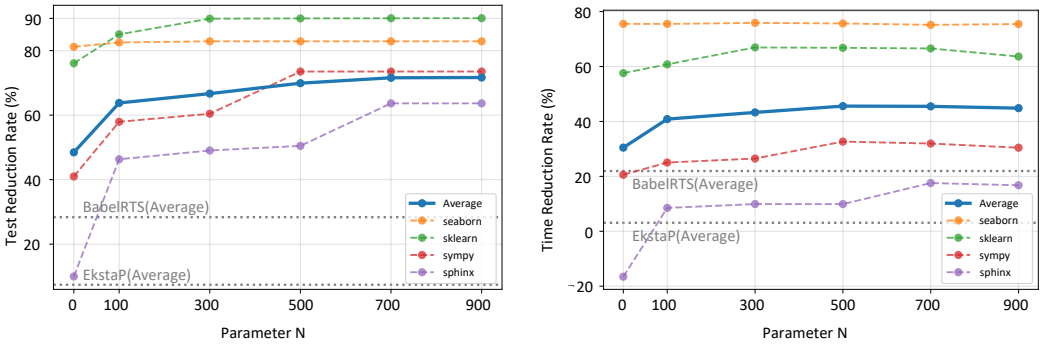
Answer to RQ2: NameRTS introduces an average overhead accounting for 10.04% of the full test-suite execution time. The overhead consists of 2.38% initialization overhead, 3.22% selection overhead, and 4.45% runtime overhead. Enabling NameRTS pays off after five commits, on average.

4.4 RQ3: Impact of Pruning Mechanisms

Approach. NameRTS incorporates two pruning mechanisms to filter unreachable code element nodes and avoid dependency cascades: critical-function pruning (CF) and context-aware name-element matching (NEM). To evaluate their contributions to effectiveness, we run three variants of NameRTS by disabling each mechanism individually (w/o CF, w/o NEM) and both together (w/o CF&NEM), and then compare their performance with that of NameRTS.

Results. Table 4 summarizes the results. Pruning critical functions has the largest impact. Removing CF reduces test reduction by 30.67% and time reduction by 33.17%, relatively. This confirms that our strategy for identifying critical functions enables NameRTS to more accurately identify the code elements that are truly affected. Removing NEM causes decreases of 2.24% and 6.79% in test and time reduction, respectively. NEM primarily improves time reduction by removing code element nodes that are not reachable, thus preventing propagation from exploring their subsequent paths and thereby reducing selection overhead. This benefit appears most noticeably in projects with relatively time-consuming selection phases. For example, in *sympy*, where selection accounts for 7.43% of the RunAll time, NEM reduces the selection overhead by 36.48%, which improves overall time reduction by an additional 4.27 percentage points. In *dask*, removing NEM reduces the time reduction by 27.65%, because NEM not only reduces the selection overhead but also helps NameRTS avoid the tests that are relatively slow. When both pruning mechanisms are removed, test reduction and time reduction drop by 36.46% and 46.75%, respectively. Even without any pruning, NameRTS still reduces more tests and time than EkstaP and BabelRTS, thanks to its fine-grained dependency analysis that exploits information unavailable to file-level RTS.

Answer to RQ3: Removing critical-function pruning causes the largest decline, reducing test and time reduction by 30.67% and 33.17%. Removing context-aware name-element matching leads to smaller but still substantial declines. Disabling both mechanisms causes the largest drop, showing that the two pruning mechanisms are essential for NameRTS.



(a) Test reduction. (b) Time reduction.

Fig. 6. Parameter sensitivity analysis (RQ4).

4.5 RQ4: Sensitivity to the Parameter N

Approach. NameRTS identifies the top-N critical function names during the initialization run and monitors the invocation of the corresponding functions during subsequent test execution. To study how the choice of N affects the effectiveness of NameRTS, we conduct a sensitivity analysis over different values of N. We evaluate NameRTS with N set to 0, 100, 300, 500, 700, and 900.

Results. Figure 6 summarizes the results for four representative projects, chosen to cover both strong and relatively weaker effectiveness of NameRTS. The reported average is computed over all ten projects. Overall, test reduction increases monotonically as N grows. Using N=500 yields a relative improvement of 44.24% in test reduction over N=0. Beyond this point, larger values of N bring only marginal additional gains, with N=900 improving test reduction by only 2.52% over N=500. Different projects saturate at different values of N, beyond which further increases provide nearly no returns. For example, *seaborn* exhibits nearly identical test reduction across all evaluated values of N from 100 to 900. In contrast, *sklearn* reaches saturation around N=300, while *sympy* and *sphinx* peak at N=500 and N=700, respectively. Time reduction shows a different trend. On average, NameRTS achieves the best time reduction when N=500. Although test reduction continues to increase for larger values of N, time reduction starts to decrease because the additional runtime overhead of monitoring more functions outweighs the benefit of skipping additional tests. The best value of N varies across projects. For instance, *sphinx* achieves a time reduction of 17.57% at N=700, which is notably higher than the 9.89% observed at N=500. Across all evaluated values of N, including N=0, NameRTS consistently outperforms the file-level baselines shown in the figure, indicating that its effectiveness is robust to the choice of N. In our evaluation, we adopt N=500 as a uniform setting across all projects to balance effectiveness and runtime overhead while keeping the configuration simple.

Answer to RQ4: N=500 provides a good balance between effectiveness and overhead, yielding near-saturated test reduction and the best average time reduction. Increasing N improves test reduction, but with diminishing returns. While larger N can further reduce tests, the added runtime overhead eventually limits time reduction.

5 Threats to Validity

Threats to internal validity. Our pruning of critical functions relies on runtime invocation information from previous executions to decide whether a function is reachable by a given test. This use of dynamic information is safe because, before propagation reaches a modified code

element, the executed code remains unchanged, and previously observed invocations remain valid. Once propagation reaches a modified element, the test is already identified as affected, and any subsequent behavioral changes do not influence the decision. Our implementation of NameRTS may not capture all Python language features. We address several constructs that influence RTS safety or effectiveness, such as decorators and operator overloading, but less common patterns may not be fully supported. To reduce this risk, we evaluate NameRTS on a dataset of 500 commits drawn from ten large projects so that widely used language features are represented. NameRTS is also extensible. Additional names can be injected into the *usedNames* sets of individual code elements or into the initial name sets of test files to accommodate project-specific patterns when needed. NameRTS does not model fully dynamic code execution mechanisms such as `eval` and `exec`. Such mechanisms are well known to be difficult to analyze efficiently [11, 38], and resolving them precisely would counteract the goal of RTS. Although this limitation may affect safety in principle, we did not observe any missed tests caused by these mechanisms in our dataset, which suggests that such cases are uncommon in practice. Future work could explore lightweight ways to approximate reflective behavior without incurring the high cost of full dynamic analysis, enabling safer RTS for reflection-heavy code.

Threats to external validity. Our dataset may not fully represent real-world development practices. To mitigate this threat, we select ten large, popular open-source projects from various application domains. For each project, we follow prior work [57] and collect fifty consecutive commits. While fifty commits may not reflect behavior across the full history of a project, this number balances evaluation cost and generalizability.

6 Related Work

RTS for statically typed languages. RTS has been studied most extensively in statically typed languages, particularly Java. Most existing techniques rely on dependency analysis, which selects tests that depend on modified program components. Based on the granularity at which dependencies are tracked, these techniques can be broadly categorized into file-level and function-level approaches. File-level RTS [26, 35, 50] tracks dependencies between test files and source files, and selects a test if any file it depends on has changed. Ekstazi [26] and STARTS [35] derive file dependencies through dynamic and static analysis respectively. Ekstazi# [50] instruments C# assemblies to capture file-level dependencies, enabling RTS for .NET. Our results show that file-level RTS is far less effective for Python because eager importing inflates dependency scopes. Function-level RTS [23, 25, 29, 39, 40, 48, 49, 51, 55, 57–59] aims to improve precision by tracking dependencies at the level of functions, typically relying on call graph construction to approximate test reachability. RTS++ [25] analyzes LLVM IR to build call graphs across revisions and selects tests whose dependency closures reach modified functions. HyRTS [58] applies function-level analysis selectively to changed files, falling back to file-level analysis elsewhere to balance precision and overhead. Wang et al. [51] and Liu et al. [40] identify semantics-modifying changes and select tests only for such changes, excluding refactorings and other non-semantic edits. JcgEks [57] combines dynamic file-level selection with call graph analysis augmented by runtime information to determine whether tests can reach modified methods. These approaches depend on constructing precise call graphs. Because Python is dynamically typed, building high-quality call graphs is significantly harder, making such techniques difficult to port directly. In contrast, NameRTS avoids call graph construction while still achieving high precision.

RTS for dynamically typed languages and language-agnostic RTS approaches. A few studies explore RTS for dynamically typed languages, where precise static dependency analysis is more challenging. NodeSRT [19] addresses challenges in JavaScript by instrumenting all functions to collect runtime

call edges across Node.js and browser environments. Pytest-rts [32] selects tests based on previously observed coverage at the test-function level, whereas our evaluation operates at the test-file level. This mismatch prevents a fair comparison, so we do not include pytest-rts as a baseline. Compared to NodeSRT and pytest-rts, NameRTS instruments only critical functions, striking a balance between test reduction and runtime overhead. BabelRTS [42] performs language-agnostic RTS via regex-based rules to extract file-level dependencies. Other work explores language-agnostic learning-based RTS [12–15, 41] that predicts affected tests from historical features, such as coverage or failure patterns, or predicts when to entirely skip regression testing [44]. These methods do not offer safety guarantees, whereas NameRTS focuses on analysis-based selection with minimal safety loss.

7 Conclusion

This paper presents NameRTS, a regression test selection approach for Python that addresses the key challenges of dependency analysis in Python RTS. NameRTS is built on *name-based dependency propagation*, a conservative dependency analysis framework based on direct name-element matching, which deliberately over-approximates dependencies to ensure safety. This design reflects the fundamental requirement of RTS to remain safe, especially in dynamic languages where precise static dependency analysis is inherently difficult to achieve. On top of this foundation, NameRTS applies two pruning strategies to eliminate false dependencies introduced by coarse name-element matching, improving precision while preserving safety. To enable rigorous evaluation, we curate the first Python RTS dataset with a precise ground truth. Experimental results show that NameRTS is substantially more effective than existing Python RTS approaches, advancing the practicality of regression test selection for Python.

8 Data-Availability Statement

Our code and data are available on GitHub [10] and archived on Zenodo [9].

Acknowledgments

This research/project is supported by the National Natural Science Foundation of China (No.92582107), the Fundamental Research Funds for the Central Universities (No.226-2025-00067), and the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] 2025. 3. Data model - Python 3.14.0 documentation. https://docs.python.org/3/reference/datamodel.html
[2] 2025. Compound statements – Python 3.14.0 documentation. https://docs.python.org/3/reference/compound_stmts.html#function-definitions
[3] 2025. dis - Disassembler for Python bytecode - Python 3.14.0 documentation. https://docs.python.org/3/library/dis.html#dis.hasname
[4] 2025. The import system – Python 3.14.0 documentation. https://docs.python.org/3/reference/import.html#regular-packages
[5] 2025. Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/
[6] 2025. Technology | 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025/technology#most-popular-technologies
[7] 2025. TIOBE Index - TIOBE. https://www.tiobe.com/tiobe-index/
[8] 2025. Tools and Trends - The State of Developer Ecosystem in 2025. https://devecosystem-2025.jetbrains.com/tools-and-trends
[9] 2026. Artifact for "Names Are All You Need: Effective and Safe Regression Test Selection for Python". Zenodo. doi:10.5281/zenodo.21279428
[10] 2026. NameRTS. GitHub repository. https://github.com/ZJU-CTAG/NameRTS

[11] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing dynamic features in python programs. In *Proceedings of the 11th working conference on mining software repositories*. 292–295. doi:10.1145/2597073.2597103

[12] Khaled Walid Al-Sabbagh, Miroslaw Staron, Miroslaw Ochodek, Regina Hebig, and Wilhelm Meding. 2020. Selective regression testing based on big data: Comparing feature extraction techniques. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops*. 322–329. doi:10.1109/ICSTW50294.2020.00058

[13] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 142–151. doi:10.1145/2597073.2597084

[14] Maral Azizi and Hyunsook Do. 2018. ReTEST: A cost effective test case selection technique for modern software development. In *2018 IEEE 29th International Symposium on Software Reliability Engineering*. 144–154. doi:10.1109/ISSRE.2018.00025

[15] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12. doi:10.1145/3377811.3380369

[16] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. 2017. Test case selection in industry: An analysis of issues related to static approaches. *Software Quality Journal* 25, 4 (2017), 1203–1237. doi:10.1007/s11219-016-9328-4

[17] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A benchmark of executable python software. *Proceedings of the ACM on Software Engineering* 1 (2024), 338–358. doi:10.1145/3643742

[18] Islem Bouzenia and Michael Pradel. 2024. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3623303

[19] Yufeng Chen. 2021. NodeSRT: a selective regression testing tool for Node. js application. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*. 126–128. doi:10.1109/ICSE-Companion52605.2021.00055

[20] Pavan Kumar Chittimalli and Mary Jean Harrold. 2009. Recomputing coverage information to assist regression testing. *IEEE Transactions on Software Engineering* 35, 4 (2009), 452–469. doi:10.1109/TSE.2009.4

[21] Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. 2025. NoCode-bench: A Benchmark for Evaluating Natural Language-Driven Feature Addition. *CoRR* abs/2507.18130 (2025). doi:10.48550/arXiv.2507.18130

[22] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 760–771. doi:10.1145/3540250.3549126

[23] Daniel Elsner, Severin Kacianka, Stephan Lipp, Alexander Pretschner, Axel Habermann, Maria Graber, and Silke Reimer. 2023. BinaryRTS: Cross-language regression test selection for C++ binaries in CI. In *2023 IEEE Conference on Software Testing, Verification and Validation*. 327–338. doi:10.1109/ICST57152.2023.00038

[24] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information and Software Technology* 52, 1 (2010), 14–30. doi:10.1016/j.infsof.2009.07.001

[25] Ben Fu, Sasa Misailovic, and Milos Gligoric. 2019. Resurgence of regression test selection for C++. In *2019 12th IEEE Conference on Software Testing, Validation and Verification*. 323–334. doi:10.1109/ICST.2019.00039

[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 211–222. doi:10.1145/2771783.2771784

[27] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *2018 IEEE 29th International Symposium on Software Reliability Engineering*. 112–122. doi:10.1109/ISSRE.2018.00022

[28] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology* 2, 3 (1993), 270–285. doi:10.1145/152388.152391

[29] Simon Hundsdorfer, Roland Würsching, and Alexander Pretschner. 2025. RustyRTS: Regression Test Selection for Rust. In *2025 IEEE Conference on Software Testing, Verification and Validation*. 338–348. doi:10.1109/ICST62969.2025.10988992

[30] Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic Software Issue Resolution with Large Language Models: A Survey. *CoRR* abs/2512.22256 (2025). doi:10.48550/arXiv.2512.22256

[31] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues?. In *International Conference on Learning Representations*.

[32] Eero Kauhanen, Jukka K Nurminen, Tommi Mikkonen, and Matvei Pashkovskiy. 2021. Regression test selection tool for python in continuous integration process. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*. 618–621. doi:10.1109/SANER50967.2021.00077

[33] James Law and Gregg Rothermel. 2003. Whole program path-based dynamic impact analysis. In *25th International Conference on Software Engineering, 2003. Proceedings.* 308–318. doi:10.1109/ICSE.2003.1201210

[34] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 583–594. doi:10.1145/2950290.2950361

[35] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering.* 949–954. doi:10.1109/ASE.2017.8115710

[36] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989.* 60–69. doi:10.1109/ICSM.1989.65194

[37] Hareton KN Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990.* 290–301. doi:10.1109/ICSM.1990.131377

[38] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology* 28, 2 (2019), 1–50. doi:10.1145/3295739

[39] Yingling Li, Junjie Wang, Yun Yang, and Qing Wang. 2019. Method-level test selection for continuous integration with static dependencies and dynamic execution rules. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security.* 350–361. doi:10.1109/QRS.2019.00052

[40] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 664–676. doi:10.1145/3597926.3598086

[41] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice.* 91–100. doi:10.1109/ICSE-SEIP.2019.00018

[42] Gabriele Maurina, Walter Cazzola, and Sudipto Ghosh. 2025. BabelRTS: Polyglot Regression Test Selection. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3554403

[43] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes* 29, 6 (2004), 241–251. doi:10.1145/1029894.1029928

[44] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 553–565. doi:10.1145/3460319.3464840

[45] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551. doi:10.1109/32.536955

[46] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. doi:10.1145/248233.248262

[47] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering.* 1646–1657. doi:10.1109/ICSE43902.2021.00146

[48] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *Proceedings of the ACM on Programming Languages* 3 (2019), 1–29. doi:10.1145/3360613

[49] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. 2016. Change-based test selection: an empirical evaluation. *Empirical software engineering* 21, 5 (2016), 1990–2032. doi:10.1007/s10664-015-9405-5

[50] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for. net. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 848–853. doi:10.1145/3106237.3117763

[51] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *Proceedings of the 40th international conference on software engineering.* 233–244. doi:10.1145/3180155.3180254

[52] You Wang, Michael Pradel, and Zhongxin Liu. 2026. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. In *2026 IEEE/ACM 48th International Conference on Software Engineering.*

[53] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. 1997. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering.* 264–274. doi:10.1109/ISSRE.1997.630875

[54] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120. doi:10.1002/stvr.430

[55] Maruf Hasan Zaber. 2021. *Towards Parallelization of Regression Test Selection.* Master's thesis. University of California, Irvine.

[56] Chengming Zhang, Haoye Wang, Chuyang Xu, Jiakun Liu, Kui Liu, and Zhongxin Liu. 2026. Can test cases generated by large language models facilitate automated program repair? *Empirical Software Engineering* 31, 3 (2026), 68. doi:10.1007/s10664-026-10802-w

[57] Guofeng Zhang, Luyao Liu, Zhenbang Chen, and Ji Wang. 2024. Hybrid Regression Test Selection by Integrating File and Method Dependences. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1557–1569. doi:10.1145/3691620.3695525

[58] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*. 199–209. doi:10.1145/3180155.3180198

[59] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *2019 IEEE/ACM 41st International Conference on Software Engineering*. 430–441. doi:10.1109/ICSE.2019.00056

Received 2026-01-30; accepted 2026-04-16