Are Coding Agents Generating Over-Mocked Tests? An Empirical Study

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Romain Robbes
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI
Bordeaux, France
romain.robbes@labri.fr

Abstract

Coding agents have received significant adoption in software development recently. Unlike traditional LLM-based code completion tools, coding agents work with autonomy (e.g., invoking external tools) and leave visible traces in software repositories, such as authoring commits. Among their tasks, coding agents may autonomously generate software tests; however, the quality of these tests remains uncertain. In particular, excessive use of mocking can make tests harder to understand and maintain. This paper presents the first study to investigate the presence of mocks in agent-generated tests of real-world software systems. We analyzed over 1.2 million commits made in 2025 in 2,168 TypeScript, JavaScript, and Python repositories, including 48,563 commits by coding agents, 169,361 commits that modify tests, and 44,900 commits that add mocks to tests. Overall, we find that coding agents are more likely to modify tests and to add mocks to tests than non-coding agents. We detect that (1) 60% of the repositories with agent activity also contain agent test activity; (2) 23% of commits made by coding agents add/change test files, compared with 13% by non-agents; (3) 68% of the repositories with agent test activity also contain agent mock activity; (4) 36% of commits made by coding agents add mocks to tests, compared with 26% by non-agents; and (5) repositories created recently contain a higher proportion of test and mock commits made by agents. Finally, we conclude by discussing implications for developers and researchers. We call attention to the fact that tests with mocks may be potentially easier to generate automatically (but less effective at validating real interactions), and the need to include guidance on mocking practices in agent configuration files.

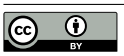
CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software Testing, Mocking, Test Doubles, Coding Agents, LLMs

ACM Reference Format:
Andre Hora and Romain Robbes. 2026. Are Coding Agents Generating Over-Mocked Tests? An Empirical Study. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793302.3793362



MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793362

1 Introduction

Large Language Models (LLMs) have been adopted in multiple software engineering tasks, such as test generation, bug fixing, and refactoring [3, 9, 19, 21, 24, 29, 33, 37, 43, 54, 58]. In the context of software testing, numerous studies have investigated the application of LLMs to generate or enhance test cases [62]. Overall, the results are promising. For example, LLMs can increase line and branch coverage [11, 52, 53], support test-driven development [1, 41], enhance existing human-written tests [4], guide test mutation [23], and generate end-to-end tests [2], to name a few. Unfortunately, LLM-generated test cases may also present quality issues. Recent studies have shown that these test cases may exhibit test smells [5, 45], lack assertions, or behave non-deterministically [4] (i.e., generating flaky tests [38]).

In 2025, LLM-based coding agents like Claude Code [12], GitHub Copilot Agent [25], and Cursor Agent [15] came out and received significant adoption in software development [35]. Unlike traditional LLM-based code completion tools, coding agents work with high *autonomy*; for example, they can invoke external tools, execute code, and complete entire development tasks. Also, unlike such code completion tools, coding agents leave visible *traces* [51] in software repositories, such as authoring or co-authoring commits. Previous work relied on annotations by developers to detect LLM-generated code in software repositories [59, 61]. In contrast, agent-based automation becomes observable through the traces left by coding agents [51], enabling the analysis of their generated code in the wild, within real-world software systems [50].

Among their tasks, coding agents may autonomously generate software tests. However, the quality of these tests remains uncertain. A controversial topic in software testing is the usage of mocking [42]. Commonly, during testing activities, developers are faced with dependencies (e.g., web APIs, databases, file systems, etc.) that make the test harder to implement. In these cases, developers can use mocking to isolate such dependencies, simulating their behavior, making the test fast, repeatable, and deterministic [42, 47]. **Despite the benefits of mocking, its overuse may cause problems: tests can be harder to understand and maintain, and they may provide less assurance that the code is working properly [28]**. That is, using mocks in tests only guarantees success if the mocks match the real implementations. In practice, this is hard to ensure, especially as the real code evolves and possibly gets out of sync with the mocks [28].

While the quality of LLM-generated tests has been explored in prior work [4, 5, 45], the use of mocking remains largely understudied. As a first step in this direction, an early study evaluated OpenAI's GPT-4o for automating mock decisions by comparing its outputs with developer choices in one system, finding that the LLM

often generates more mocks than the developers [49]. Interestingly, as anecdotal evidence, Kent Beck [7] recently stated: "*[LLM] makes some decisions seemingly at random, like, "Oh, let's use a mock for this test even though the actual object is fine. The price of augmented coding is eternal vigilance*".[1] However, to our knowledge, the agent-generated tests and their relation with mocking have not yet been explored. Thanks to the traces left by coding agents in software repositories, we are now able to observe this phenomenon [51].

This paper presents the first empirical study to investigate the presence of mocks in agent-generated tests of real-world software systems. We analyzed over 1.2 million commits made in 2025 in 2,168 TypeScript, JavaScript, and Python repositories. This dataset comprises 48,563 commits by coding agents, 169,361 commits that modify tests, and 44,900 commits that add mocks to tests. We propose three research questions to address the test generation, mock generation, and mock types:

- **RQ1. How frequently do coding agents generate tests?** We find that 60% of the repositories with agent activity also contain agent test activity. Moreover, 23% of commits made by coding agents involve the addition or modification of test files, compared with 13% of commits made by non-agents. Considering all commits containing tests, 7% are made by coding agents, and this proportion increases to 17% in repositories created in 2025.
- **RQ2. How frequently do coding agents generate mocks in tests?** Overall, 68% of the repositories with agent test activity also contain agent mock activity. 36% of test commits made by coding agents add mocks to test code, compared with 26% of test commits made by non-agents. Considering all commits containing mocks, 9% are made by coding agents, and this proportion increases to 19% in repositories created in 2025. Moreover, in repositories with higher agentic activity, coding agents had a higher ratio of mock commits (36%) compared to non-agents (28%).
- **RQ3. What types of mocks are generated by coding agents?** Coding agents predominantly use the *mock* type (95%), whereas non-agents employ a wider variety: *mock* (91%), *fake* (57%), and *spy* (51%). In both cases, *dummy* and *stub* are the least adopted types.

Overall, we find that coding agents are more likely to modify tests and to add mocks than non-coding agents. Based on our findings, we discuss several implications for practitioners and researchers. *First*, the higher likelihood of coding agents modifying test files suggests that these tools are being actively used not only for implementing production code but also for maintaining and expanding tests, highlighting a growing potential for agents to support software testing tasks autonomously. *Second*, the fact that coding agents tend to rely more heavily on mocking may reflect an overuse of isolation techniques, potentially leading to tests that are easier to generate automatically, but less effective at validating real interactions. *Third*, given the tendency of coding agents to rely on mocking, practitioners should explicitly include guidance on mocking best practices and anti-patterns in agent configuration files (e.g., `CLAUDE.md`) to ensure consistent test generation. *Lastly*, the limited diversity of test double types used by coding agents suggests that agent-generated tests may rely too heavily on mocks.

Contributions: The contributions of this study are twofold: (1) we provide the first empirical study to analyze agent-generated tests, particularly exploring the generation of tests and mocks in real-world software systems; and (2) we propose multiple actionable implications for practitioners and researchers.

2 Study Design

2.1 Case Study

2.1.1 Coding Agents. Coding agents go beyond predictive code completion tools. Unlike such tools, they can autonomously decompose complex goals, invoke external tools (e.g., compilers and search engines), execute code, and refine their outputs with minimal human intervention [35, 50]. These capabilities mark a shift toward intelligent systems that exhibit decision-making, adaptation, and autonomy in software development [35]. In this paper, we focus on three leading coding agents: Claude Code [12], GitHub Copilot [25], and Cursor [15]. These agents are selected based on their growing adoption [35, 50] and their roles as autonomous contributors or co-authors in software repositories.

2.1.2 Programming Languages. We aim to perform a multi-language empirical study so we can have a better overview of test and mock adoption in the wild. We then select the top-3 most used languages on GitHub: Python, JavaScript, and TypeScript [27].

2.2 Initial Set of Repositories

Our goal is to analyze real-world, actively maintained repositories hosted on GitHub. To this end, we rely on the SEART GitHub Search Engine (seart-ghs), a tool that allows researchers to sample repositories to use for empirical studies by using multiple combinations of selection criteria [17]. This tool maintains metadata for all GitHub repositories with at least ten stars. Based on this tool, we selected repositories that meet the following criteria: at least 100 commits, a minimum of 5,000 non-blank lines of code, not being forks, and having at least one commit in the last two months. This process yielded an initial set of 114,098 repositories.

2.3 Detecting Repositories with Agents

From the initial set of repositories, we selected the ones where the main language was Python, TypeScript, or JavaScript and adopted the coding agents Claude, Copilot, or Cursor. To detect the presence of such coding agents, we cloned the repositories and verified the existence of files or directories associated with them. We relied on the patterns presented in Table 1, derived from the official documentations of Claude [13], Copilot [14], and Cursor [16].

Table 1: Coding agent file and directory patterns.

Coding Agent	File and Directory Pattern
Claude	`CLAUDE.md`, `.claudeignore`, `.claude/`, `anthropic/`
Cursor	`CURSOR.md`, `.cursor/`, `.cursorrules`
Copilot	`copilot_instructions.md`, `copilot-instructions.md`, `.copilot-*.md`, `.copilotignore`, `.copilot/`

[1] https://www.linkedin.com/posts/kentbeck_a-problem-im-having-augmented-coding-is-activity-7328422517025452032-Fwxi

These configuration files are used to document commands, instructions, and guidelines for coding agents. For example, the Claude Code documentation states [13]: "`CLAUDE.md` *is a special file that Claude automatically pulls into context when starting a conversation. This makes it an ideal place for documenting: common bash commands, core files and utility functions, code style guidelines, testing instructions [...]*". We found a total of 2,168 repositories with agent files or directories across the three target programming languages.

2.4 Detecting Agent Commits

The presence of a coding agent file or directory in a repository does not necessarily imply that the agent was used to make commits. Therefore, the next step in our study design is to identify repositories containing commits actually made by coding agents. To this end, we analyzed the commits of the selected repositories, searching for commits authored or co-authored by the coding agents Claude, Cursor, or Copilot. The *co-authored* metadata is a GitHub feature that allows a commit to have multiple authors [26]. In this case, developers can attribute a commit to more than one author by adding one or more `Co-authored-by:` trailers to the commit message, thereby making those co-authors visible on GitHub. Although GitHub recommends using the `Co-authored-by:` commit trailer [26], we observed that coding agents may also use variants such as `Co-Authored-By:` (see Figure 1b); therefore, we searched for these trailers in commit messages in a case-insensitive manner.

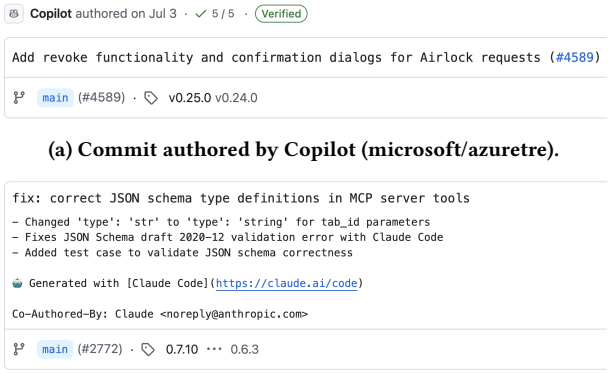
We then identified commits whose *author* or *co-authored-by* metadata contained, in a case-insensitive manner, the substrings *claude*, *cursor*, or *copilot*. In addition, we also looked for traces of all popular coding agents in commit authors or co-authors, including Aider, OpenHands, Devin AI, Google Jules, Cline, Junie, Gemini, CodeRabbit, and Windsurf. All patterns used to detect the presence of agents were derived from official documentation or from a manual assessment of commits authored by coding agents. We manually inspected 500 agent commits and found a precision of 100% (500 out of 500) in the agent commit classification.

We label commits containing these coding agents as *agent commits*, while other commits as *non-agent commits*. Figure 1a presents an example of a commit authored by Copilot in the repository microsoft/azuretre,[2] while Figure 1b shows a commit co-authored by Claude in the repository browser-use/browser-use.[3]

We found a total of 48,563 agent commits in 1,219 repositories.

2.5 Detecting Test Commits

The next step in our design is to detect *test commits*, that is, commits that add or modify test files. Table 2 presents the patterns used to detect the presence of the test files in Python, TypeScript, and JavaScript. For Python, we followed the convention that test files begin with `test_` or end with `_test`, which is used by the testing frameworks Pytest and unittest to automatically detect tests. For TypeScript and JavaScript, test files typically have the suffixes `test.ts` or `spec.ts` (in TypeScript) and `test.js` or `spec.js` (in JavaScript), which are used by popular testing frameworks such as Jest and Mocha to automatically detect tests.[4] In addition, we also consider as test files all files located in directories whose paths contain the substring `test`, such as `test/`, `e2e_tests/`, and `__tests__/`. For TypeScript and JavaScript, we also include directories that end with `spec/`. These directories are commonly used in Python, TypeScript, and JavaScript projects and, depending on the testing framework, may not require files within them to include the `test` prefixes or suffixes. Based on this method, we found a total of 169,361 test commits in 1,779 repositories.



(a) Commit authored by Copilot (microsoft/azuretre).

(b) Commit co-authored by Claude (browser-use/browser-use).

Figure 1: Examples of agent commits.

Table 2: Test file patterns.

Prog. Language	Test File Pattern	Example
Python	`test_*.py` or `*_test.py`	`test_foo.py`
TypeScript	`*test.ts` or `*spec.ts`	`foo.test.ts`
JavaScript	`*test.js` or `*spec.js`	`foo.test.js`

2.6 Detecting Mock Commits

According to Meszaros [42], mocks "*replace an object the system under test (SUT) depends on with a test-specific object that verifies it is being used correctly by the SUT*". In fact, a mock is a particular type of test double [42]. The testing literature proposes five test doubles: dummy, stub, spy, mock, and fake, each one with its isolation characteristics [42]. Dummy is the simplest test double, as its behavior is not predefined nor verified; stub offers predefined implementations; mock and spy verify interactions; and fake provides simplified implementations [22]. Overall, a test double can be defined as "*the broadest term available to describe any fake thing introduced in place of a real thing for the purpose of writing an automated test*".[5] Meszaros recognizes that terminology around the types of test doubles is confusing and inconsistent, hence different people use distinct terms to mean the same thing [42].

Developers tend to use the terms mocks and test doubles interchangeably [47]. In this context, Robert Martin [40] mentions: "*The word mock is sometimes used in an informal way to refer to the whole family of objects that are used in tests*".[6]

[2] https://github.com/microsoft/azuretre/commit/a266b50733
[3] https://github.com/browser-use/browser-use/commit/b2da2fec89
[4] https://create-react-app.dev/docs/running-tests/#filename-conventions
[5] https://github.com/testdouble/contributing-tests/wiki/Test-Double
[6] https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html

Thus, in this paper, we use the term mock due to its popularity, but our analysis encompasses the full family of test doubles, that is, dummy, stub, spy, mock, and fake.

2.6.1 Mock Usage. There are two approaches to using mocks in test code: (1) employing mocking frameworks or (2) manually creating them in the source code [47].

JavaScript and TypeScript provide multiple mocking frameworks such as Sinon.JS,[7] Jest Mock Functions,[8] and Jasmine Spy.[9] Examples of APIs provided by such mocking frameworks include `mock`, `spyOn`, `spyOnProperty`, and `fake`, to name a few. In Python, the unittest testing framework provides unittest.mock,[10] while Pytest provides monkeypatch[11] and pytest-mock.[12] As expected, such frameworks include multiple APIs to create mocks, such as `MagicMock`, `mocker`, `mock`, `spy`, and `stub`. In addition, Pytest provides the `monkeypatch`[13] fixture for mocking functionality in tests, while unittest provides `patch`,[14] which acts as a function decorator, class decorator, or a context manager.

In the second approach, developers manually create mocks in test code [46], including classes, methods, and variables with mocked implementations. For example, a search in the microsoft/vscode repository for the five test double types returns thousands of occurrences, such as `dummyDisplayObj`, `RemoteServiceStub`, `spyEncryptionService`, `MockLabelService`, and `_fakeAuthPopulate`.[15]

Thus, our approach to automatically detecting mock usage aims to capture both framework-based and manually created mocks. Specifically, to determine whether a block of code uses a test double, we first parse the code to extract all identifiers and then check whether any of them include the terms *dummy*, *dummies*, *stub*, *mock*, *spy*, *spies*, or *fake*, in a case-insensitive manner. In addition, we also verify the presence of the terms *monkeypatch* and *monkey_patch*. Moreover, since *patch* is a common term that may appear in contexts unrelated to test doubles, we only capture its occurrences when used in Python decorators (i.e., `@patch`, `@mock.patch`, or `@unittest.mock.patch`) or Python context managers (i.e., `with patch`, `with mock.patch`, or `with unittest.mock.patch`), which are standard ways to invoke the `patch` API according to the official unittest documentation. Identifiers containing such mock-related terms are referred to as *mock identifiers*, e.g., `dummyDisplayObj`.

We manually inspected 100 randomly selected mock commits of 10 repositories (10 commits per repository), and found a precision of 94% (94 in 100) in the identifier-based mock detection.

This approach is in line with prior studies, which also rely on code identifiers to detect the presence of test doubles [22, 47]. Therefore, we can capture the presence of test doubles in any code constructor, including classes, methods, functions, parameters, arguments, attributes, calls, and variables.

2.6.2 Mock Commits. The next step in our study is to identify commits that actually add mocks to test code; for brevity, we refer to these as *mock commits*. It is well known that commits often include unrelated modifications—also referred to as tangled changes [20]. For instance, a single commit may simultaneously add a new test and refactor an existing one. To mitigate potential noise introduced by such tangled commits, we focus on identifying mock commits that explicitly introduce new mock identifiers.

Specifically, for each test commit, we examine the test files that were added or modified. First, we extract all mock identifiers appearing in the added and deleted lines of these files, yielding two sets: *added mock identifiers* and *deleted mock identifiers*. Next, we determine the *new mock identifiers*, defined as the mock identifiers that appear only in the added lines (i.e., not present among the deleted ones). Finally, we classify a test commit as a *mock commit* if it introduces at least one *new mock identifier*. Based on this method, we found 44,900 mock commits in 1,381 repositories.

Figure 2 presents three examples of mock commits. First, we see a mock commit in the repository mlflow/mlflow that added the manually created test double `DummyModel` (Figure 2a).[16] Next, in Figure 2b, we observe a mock commit that added two mock APIs (`MagicMock` and `patch` of unittest.mock) and the alias `mock_evaluate`.[17] Lastly, in the repository prebid/prebid.js we see a mock commit that added the mock API `stub` of the Sinon.JS mocking framework (Figure 2c).[18]

```
318 + def test_download_artifacts_with_run_id_and_artifact_path(tmp_path):
319 +     class DummyModel(mlflow.pyfunc.PythonModel):
320 +         def predict(self, context, model_input: list[str]) -> list[str]:
321 +             return model_input
322 +
323 +     with mlflow.start_run() as run:
324 +         mlflow.pyfunc.log_model(name="model", python_model=DummyModel())
325 +     mlflow.artifacts.download_artifacts(
326 +         run_id=run.info.run_id, artifact_path="model", dst_path=tmp_path
327 +     )
```

(a) New mock identifier: `DummyModel` (mlflow/mlflow, 6ba3cefcee).

```
192 + def test_input_is_optional_if_trace_is_provided():
193 +     with patch("mlflow.evaluate") as mock_evaluate:
194 +         mlflow.genai.evaluate(
195 +             data=pd.DataFrame({"outputs": ["Paris"], "trace": [MagicMock()]}),
196 +         )
197 +
198 +         mock_evaluate.assert_called_once()
```

(b) New mock identifiers: `mock_evaluate`, `MagicMock`, and `patch` (mlflow/mlflow, 6920a3a779).

```
132 +     beforeEach(function () {
133 +       sandbox = sinon.createSandbox();
134 +
135 +       // Stub generateUUID to return a fixed value for testing
136 +       sandbox.stub(utils, 'generateUUID').returns('test-generated-uuid');
137 +
138 +       // Stub config.getConfig for bidderTimeout
139 +       sandbox.stub(config, 'getConfig').withArgs('bidderTimeout').returns(30);
140 +     });
```

(c) New mock identifier: `stub` (prebid/prebid.js, 5e57caa8ce).

Figure 2: Examples of mock commits.

[7] https://sinonjs.org
[8] https://jestjs.io/docs/mock-functions
[9] https://jasmine.github.io/tutorials/spying_on_properties
[10] https://docs.python.org/3/library/unittest.mock.html
[11] https://docs.pytest.org/en/stable/how-to/monkeypatch.html
[12] https://pytest-mock.readthedocs.io/en/latest/index.html
[13] https://docs.pytest.org/en/stable/reference/reference.html#pytest.MonkeyPatch
[14] https://docs.python.org/3/library/unittest.mock.html#the-patchers
[15] GitHub search: https://github.com/search?q=repo%3Amicrosoft%2Fvscode+language%3ATypeScript+dummy+OR+stub+OR+spy+OR+mock+OR+fake&type=code

[16] Mock commit: https://github.com/mlflow/mlflow/commit/6ba3cefcee
[17] Mock commit: https://github.com/mlflow/mlflow/commit/6920a3a779
[18] Mock commit: https://github.com/prebid/prebid.js/commit/5e57caa8ce

2.7 Dataset Overview

Table 3 summarizes the analyzed data. Overall, we analyzed 1,254,878 commits made in 2025 in 2,168 repositories. Among these, 48,563 were agent commits from 1,219 repositories, 169,361 were test commits from 1,779 repositories, and 44,900 were mock commits from 1,381 repositories. To perform the repository analysis, we adopted PyDriller [48] and GitEvo [31]. Our scripts and dataset are publicly available at: https://doi.org/10.5281/zenodo.17427638.

Table 3: Summary of commits and repositories.

Commits	TS	JS	Python	Total
All commits	835,781	98,389	320,708	1,254,878
Agent commits	32,728	3,892	11,943	48,563
Test commits	94,747	6,428	68,186	169,361
Mock commits	23,838	1,561	19,501	44,900
Repositories				
With agent files or directories	1,392	242	534	2,168
With agent commits	773	117	329	1,219
With test commits	1,149	143	487	1,779
With mock commits	890	89	402	1,381

2.8 Research Question

2.8.1 RQ1. How frequently do coding agents generate tests? First, we investigate how frequently coding agents add or modify test files. We examine the overlap between agent/non-agent commits and test/non-test commits in all 1,254,878 commits. To assess whether these two categorical variables are statistically associated, we applied a Chi-squared test of independence. **Rationale:** This analysis is a first step toward better understanding the extent to which coding agents contribute to testing activities compared to non-agents.

2.8.2 RQ2. How frequently do coding agents generate mocks in tests? To better understand how coding agents generate mocks, we divide this research question into two parts: (1) at the commit level and (2) at the repository level. *First*, at the commit level, we explore the overlap between agent/non-agent test commits and mock/non-mock commits across all 169,361 test commits. To assess whether these two categorical variables are statistically associated, we also applied a Chi-squared test of independence. *Second*, at the repository level, we group commits belonging to the same repository. We categorize repositories based on their level of agentic activity: (1) repositories with lower agentic activity (i.e., between 10 and 49 agent commits) and (2) repositories with higher agentic activity (i.e., at least 50 agent commits). In both cases, we select repositories with at least 10 test commits to ensure a minimum level of testing activity. We make no assumption about the presence of mocks in test commits, as this is the phenomenon we aim to investigate. To assess the mock differences between agents and non-agents within the *same* repository, we applied a paired Wilcoxon test and Cliff's delta to compute the effect size. **Rationale:** This analysis aims to better understand the extent to which coding agents contribute to mocking generation as compared to non-agents. It is important to note that a mock overuse can make the test harder to understand and maintain, and they may provide less assurance that the code is working properly [28], but the ratios of mocking usage can vary across repositories. This motivated our repository-level analysis, which performs a paired comparison of mock generation by agents and non-agents within the *same* repository.

2.8.3 RQ3. What types of mocks are generated by coding agents? In our final research question, we explore the presence of different mock types, i.e., dummy, stub, spy, mock, and fake [42], in mock commits. Here, we consider repositories that contain at least one mock commit made by coding agents, totaling 496 repositories. **Rationale:** We aim to explore the types of mocks created by agents and non-agents. This analysis helps us understand whether mock types are used evenly or tend to concentrate on specific types.

3 Results

3.1 RQ1. Coding Agents and Tests

In this analysis, we explore all 1,254,878 commits of the 2,168 repositories. Table 4 presents the contingency table for the agent commits and test commits. Overall, we detected 48,563 agent commits (in 1,219 repositories), of which 11,035 are test commits, resulting in a test commit ratio of 23%. By comparison, non-coding agents had a test commit ratio of only 13% (i.e., 158,326 out of 1,206,315). Those 11,035 agent test commits are distributed across 729 repositories, indicating that 60% (729 out of 1,219) of the repositories with agent activity also contain agent test activity.

Table 4: Contingency table of agent commits versus test commits (all repositories).

	Test commit	Non-test commit	Total
Agent commit	11,035	37,528	48,563
Non-agent commit	158,326	1,047,989	1,206,315
Total	169,361	1,085,517	1,254,878

We applied the Chi-squared test of independence to assess whether the two categorical variables are statistically associated. The test revealed a significant association between them (χ^2 = 3,683.06, df = 1, p < 0.001). The standardized residuals presented in Figure 3 show that the cell corresponding to agent commits and test commits is substantially overrepresented (55.35), whereas the remaining cells are under- or overrepresented to a lesser extent. Overall, these results demonstrate a strong deviation from independence between the two variables, suggesting that agent commits are more likely to add or modify test files compared to non-agent commits.

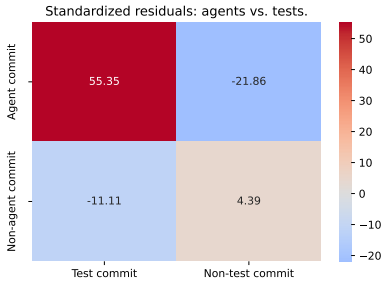


Figure 3: Standardized residuals for the contingency table of agent commits versus test commits (all repositories).

Observation 1: 60% of the repositories with agent activity also contain agent test activity. 23% of commits made by coding agents involve the addition or modification of test files, compared with 13% of commits made by non-agents.

Table 5 presents the ratio of test commits made by coding agents. We observe a small variation across languages, with Python repositories showing a higher ratio (27%) compared to JavaScript/TypeScript (21%). Among the coding agents, Copilot presents the highest proportion of test commits (27%), followed by Claude (24%) and Cursor (16%). It is worth noting, however, that Cursor accounts for the smallest number of agent commits (1,688) in our dataset.

Table 5: Ratio of test commits made by coding agents.

Category	Group	Agent		
		Commits	Test Commits	Ratio (%)
All	–	48,563	11,035	23%
Language	Python	11,943	3,249	27%
	JS/TS	36,620	7,786	21%
Coding Agent	Copilot	5,986	1,616	27%
	Claude	31,370	7,396	24%
	Cursor	1,688	278	16%
	Other	9,746	1,851	19%

Interestingly, Table 4 also shows that agent test commits account for 7% of all test commits (i.e., 11,035 out of 169,361). For comparison, Table 6 presents the contingency table for repositories created in 2025. In this case, we notice that agent test commits account for 17% of all test commits (i.e., 4,526 out of 26,654). Thus, repositories created in 2025 present a higher proportion of agent test commits than the overall dataset (17% vs. 7%).

Table 6: Contingency table of agent commits versus test commits (only repositories created in 2025).

	Test commit	Non-test commit	Total
Agent commit	4,526	18,654	23,180
Non-agent commit	22,128	20,1744	22,3872
Total	26,654	220,398	247,052

Observation 2: Considering all commits containing tests, 7% are made by coding agents. For repositories created in 2025, this proportion increases to 17%.

3.2 RQ2. Coding Agents and Mocks

To better understand how coding agents generate mocks, we conduct two complementary analyses: (1) at the commit level and (2) at the repository level.

3.2.1 Commit-Level Analysis. In this first analysis, we focus on the 169,361 test commits. Table 7 presents the contingency table for the agent test commits and mock commits. Overall, we detected 11,035 agent test commits (in 729 repositories), of which 3,934 are mock commits, resulting in a mock commit ratio of 36%. By comparison, non-coding agents presented a mock commit ratio of 26% (i.e., 40,966 out of 158,326). Those 3,934 agent mock commits are distributed across 496 repositories, indicating that 68% (496 out of 729) of the ones with agent test activity also contain agent mock activity.

Table 7: Contingency table of agent test commits versus mock commits (all repositories).

	Mock commit	Non-mock commit	Total
Agent test commit	3,934	7,101	11,035
Non-agent test commit	40,966	117,360	158,326
Total	44,900	124,461	169,361

As in RQ1, we applied the Chi-squared test of independence. In this case, the test also revealed a significant association (χ^2 = 505.5, df = 1, p < 0.001). The standardized residuals presented in Figure 4 show that the cell corresponding to agent test commits and mock commits is substantially overrepresented (18.64). Overall, these results suggest that agent test commits are more likely to add mocks compared to non-agent test commits.

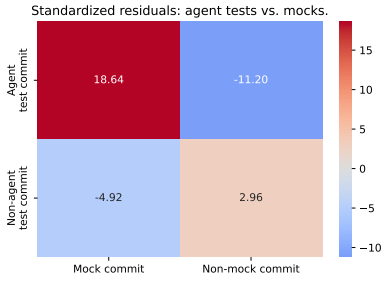


Figure 4: Standardized residuals for the contingency table of agent test commits versus mock commits (all repositories).

Observation 3: Overall, 68% of the repositories with agent test activity also contain agent mock activity. 36% of test commits made by coding agents add mocks to test code, compared with 26% of test commits made by non-agents.

Table 8 presents the ratio of mock commits made by coding agents in tests, separated by language and by agent. We observe minimal differences between Python (37%) and JavaScript/TypeScript (35%) repositories. Likewise, the ratios are similar across coding agents, with Cursor showing the highest proportion (37%), followed by Copilot (36%) and Claude (34%). It is worth noting, however, that Cursor contributes the smallest number of agent test commits (only 278) in our dataset.

Observation 4: Python, JavaScript, and TypeScript repositories present similar ratios of mock commits made by coding agents. Likewise, the ratios are comparable across different coding agents.

In Table 7, we notice that agent mock commits account for 9% of all mock commits (i.e., 3,934 out of 44,900). For comparison, Table 9 presents the contingency table for repositories created in 2025. In

Table 8: Ratio of mock commits made by coding agents.

Category	Group	Agent		
		Test Commits	Mock Commits	Ratio (%)
All	–	11,035	3,934	36%
Language	Python	3,249	1,214	37%
	JS/TS	7,786	2,720	35%
Coding Agent	Cursor	278	103	37%
	Copilot	1,616	582	36%
	Claude	7,396	2,501	34%
	Other	1,851	806	44%

this case, we notice that agent mock commits account for 19% of all mock commits (i.e., 1,529 out of 7,855). Thus, repositories created in 2025 present a higher proportion of agent mock commits than the overall dataset (19% vs. 9%).

Table 9: Contingency table of agent test commits versus mock commits (repositories created in 2025).

	Mock commit	Non-mock commit	Total
Agent test commit	1,529	2,997	4,526
Non-agent test commit	6,326	15,802	22,128
Total	7,855	18,799	26,654

Observation 5: Considering all commits containing mocks, 9% are made by coding agents. For repositories created in 2025, this proportion increases to 19%.

3.2.2 Repository-Level Analysis. In this second analysis, we focus on the repository level by grouping commits belonging to the same repository. We categorize repositories based on their level of agentic activity: (1) repositories with lower agentic activity (i.e., between 10 and 49 agent commits) and (2) repositories with higher agentic activity (i.e., at least 50 agent commits). In both cases, we only include repositories that contain at least 10 test commits to avoid those without minimal test activity. Table 10 summarizes the median frequency of mock commits per repository for both categories.

First, Table 10a presents the analysis for the 282 repositories with lower agentic activity. On the median, coding agents made 5 test commits per repository, of which 1 was a mock commit, resulting in a mock ratio of 27%. In contrast, non-agents produced a mock ratio of 24%. We applied a paired Wilcoxon test to compare the mock ratios between agents and non-agents within the same repository.[19] The test revealed a significant difference with p-value = 0.0029. However, the effect size was negligible (Cliff's delta = 0.002), indicating that the observed difference is statistically significant, but not meaningful.

Second, Table 10b details the analysis for the 179 repositories with higher agentic activity. In this case, on the median, coding agents made 30 test commits per repository, of which 10 were a mock commit, resulting in a mock ratio of 36%. In contrast, non-agents produced a mock ratio of 28%. Again, we applied a paired Wilcoxon test to compare the mock ratios between agents and non-agents. The test revealed a significant difference with p-value < 0.001 and, in this case, a small effect size (Cliff's delta = 0.252).

Table 10 also presents the median values per programming language. We find no notable difference between Python and JavaScript/TypeScript. For example, the mock commit ratios are 36% for Python and 35% for JavaScript/TypeScript in repositories with higher agentic activity.

Observation 6: In repositories with higher agentic activity, coding agents had a higher ratio of mock commits (36%) compared to non-agents (28%).

3.2.3 Examples of Repositories with High Frequency of Agent Mock Commits. Table 11 presents repositories with a high frequency of agent mock commits. Category *All* in Table 11 presents the top-10 repositories with the most mock commits made by coding agents across the entire dataset. The top-1 is promptfoo/promptfoo (148 agent mock commits), a tool for testing LLM applications. As an example of an agent mock commit, we show commit 0cdc01a,[20] which added multiple mock identifiers to tests, such as `MockedFunction` and `createMockTargetProvider`. The second repository is drivly/ai (138), an agentic workflow platform. As an example, we present commit 4eb5ea0,[21] which states "*Add comprehensive unit tests for functions.do SDK*" and includes diverse mock identifiers, as `mockFetch`, `mockApiInstance`, and `consoleSpy`. The third repository is liam-hq/liam (73), a tool to automatically generate ER diagrams from databases. As an example, we show commit 695e79f,[22] which mentions "*Add comprehensive test coverage for all routing scenarios*" and includes multiple occurrences of `createMockState`.

We also investigated the presence of agent mock commits in repositories maintained by top organizations on GitHub. For this purpose, we collected the top 100 organizations with the most stars from Gitstar Ranking.[23] Among them, we identified 24 repositories with at least one mock commit made by code agent: Microsoft (12), Home-Assistant (3), Redis (2), Azure (2), GitHub (1), Apache (1) MUI (1), Ethereum (1), and Cloudflare (1). Table 11 presents the top-10 repositories with the most agent mock commits in the top organizations. We notice important repositories such as home-assistant/core (61 agent mock commits) and microsoft/vscode (14).

In home-assistant/core, we present commit 4ca1ae6,[24] which adds multiple mocks as `FAKE_STATIONS` and `mocked_stations`. In microsoft/vscode, we show commit 8222326,[25] which includes mocks as `createMockServiceConfig` and `mockConfigurationTrue`. Lastly, in apache/superset, we present commit 7d0a472,[26] which add mocks as `mockGetBootstrapData` and `mockMatchMedia`.

Observation 7: Coding agents made commits with mocks in a wide range of repositories, including 24 maintained by popular organizations such as Microsoft and Redis.

[19] Normality tests indicated that the data were not normally distributed. Both Shapiro-Wilk and D'Agostino tests yielded p-values < 0.001, thus, we used the non-parametric Wilcoxon test.

[20] https://github.com/promptfoo/promptfoo/commit/0cdc01aca0
[21] https://github.com/drivly/ai/commit/4eb5ea0415
[22] https://github.com/liam-hq/liam/commit/695e79f3eb
[23] https://gitstar-ranking.com/organizations
[24] https://github.com/home-assistant/core/commit/4ca1ae61aa
[25] https://github.com/microsoft/vscode/commit/822232673f178e733b0
[26] https://github.com/apache/superset/commit/7d0a472d1e401df6a92b

Table 10: Median frequency of mock commits per repository.

(a) Repositories with lower agentic activity (between 10 and 49 agent commits and at least 10 test commits).

Language	#Repositories	Agent			Non-Agent			p-value	Effect-Size
		Test Commits	Mock Commits	%	Test Commits	Mock Commits	%		
JS/TS	195	4	1	25%	64	12	24%	–	–
Python	87	8	2	27%	64	11	24%	–	–
All	282	5	1	27%	64	12	24%	0.002	negligible

(b) Repositories with higher agentic activity (at least 50 agent commits and at least 10 test commits).

Language	#Repositories	Agent			Non-Agent			p-value	Effect-Size
		Test Commits	Mock Commits	%	Test Commits	Mock Commits	%		
JS/TS	123	28	9	35%	42	7	22%	–	–
Python	56	33	12	36%	105	24	30%	–	–
All	179	30	10	36%	56	13	28%	<0.001	small

Table 11: Top-10 repositories with the most mock commits across the entire dataset and within top organizations.

Category	Repository	Language	Agent		
			Test Commits	Mock Commits	Ratio
All	promptfoo/promptfoo	TypeScript	232	148	0.64
	drivly/ai	TypeScript	395	138	0.35
	liam-hq/liam	TypeScript	511	73	0.14
	czlonkowski/n8n-mcp	TypeScript	221	67	0.30
	chatman-media/timeline-studio	TypeScript	144	66	0.46
	calcom/cal.com	TypeScript	130	64	0.49
	home-assistant/core	Python	87	61	0.70
	Kilo-Org/kilocode	TypeScript	102	58	0.57
	dyoshikawa/rulesync	TypeScript	189	57	0.30
	ar-io/ar-io-node	TypeScript	106	45	0.42
Top Organizations	home-assistant/core	Python	87	61	0.70
	microsoft/fluidframework	TypeScript	43	16	0.37
	microsoft/vscode	TypeScript	78	14	0.18
	redis/agent-memory-server	Python	55	12	0.22
	microsoft/vscode-pull-request-github	TypeScript	14	8	0.57
	home-assistant/supervisor	Python	11	7	0.64
	cloudflare/workers-sdk	TypeScript	19	7	0.37
	apache/superset	TypeScript	20	7	0.35
	microsoft/omnichannel-chat-sdk	TypeScript	15	7	0.47
	Azure/LogicAppsUX	TypeScript	11	5	0.45

3.3 RQ3. Types of Generated Mocks

In our final research question, we explore the presence of different mock types [42] (namely dummy, stub, spy, mock, and fake) within mock commits. For this analysis, we consider only repositories that contain at least one mock commit made by coding agents, totaling 496 repositories. Figure 5 shows the distribution of mock types across the 496 repositories. For example, 19% of the repositories contain at least one agent mock commit with a *dummy*, compared to 40% for non-agent commits.

The most commonly adopted type is *mock* for both agents and non-agents, which seems to reflect the fact that developers tend to use the terms mocks and test doubles interchangeably [47]. Moreover, *dummy* and *stub* are the least adopted types, which are the simplest test doubles. We recall that, in theory, a *dummy* has no predefined or verified behavior, whereas a *stub* only provides predefined implementations [22]. It is worth noting that non-agent mock commits are more evenly distributed across types, particularly *mock* (91%), *fake* (57%), and *spy* (51%). In contrast, agent mock commits are more concentrated in the *mock* type (95%), followed by *spy* (33%) and *fake* (32%). This suggests that coding agents tend to generate less diverse test doubles as compared to non-agents.

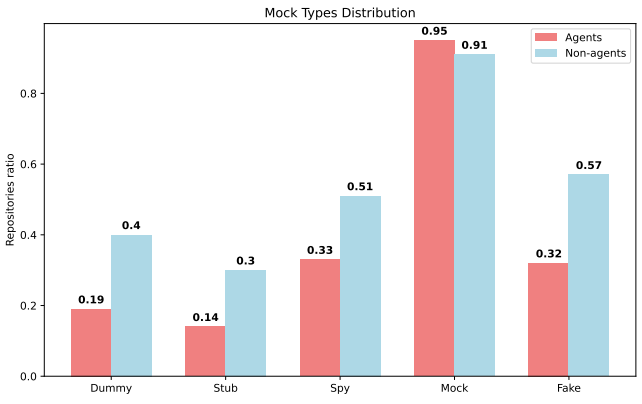


Figure 5: Distribution of mock types.

Observation 8: Coding agents predominantly use the *mock* type (95%), whereas non-agents employ a wider variety: *mock* (91%), *fake* (57%), and *spy* (51%).

4 Discussion and Implications

4.1 Coding agents are likely to add/modify tests

We detected that agent commits are more likely to add or modify test files compared to non-agent commits (RQ1). Overall, 60% of the repositories with agent activity also contain agent test activity, and 23% of commits made by coding agents involve the addition or modification of test files, compared with 13% of commits made by non-agents. The Chi-squared test of independence showed that the condition agent commits and test commits is overrepresented.

Implication: The higher likelihood of coding agents modifying or adding test files suggests that these tools are being actively used not only for implementing production code but also for maintaining and expanding tests. This indicates a growing potential for agents to support software testing tasks autonomously, which may influence how practitioners instruct coding agents to create tests and review the automatically generated tests.

4.2 Coding agents are likely to add mocks

We detected that agent test commits are more likely to add mocks to tests compared to non-agent test commits (RQ2). Overall, 68% of the repositories with agent test activity also contain agent mock activity, and 36% of test commits made by coding agents add mocks, compared with 26% of test commits made by non-agents. The Chi-squared test of independence showed that the condition agent test commits and mock commits is overrepresented. Even within the same repository, we found that coding agents tend to add proportionally more mocks (36%) than non-agents (28%). Using mocks in tests only guarantees success if the mocks match the real implementations. In practice, this is hard to ensure, especially as the real code evolves and possibly gets out of sync with the mocks [28].

Implication: Coding agents tend to rely more heavily on mocking when generating or modifying tests. This may reflect an overuse of isolation techniques, potentially leading to tests that are easier to generate automatically, but less effective at validating real interactions [28]. Practitioners should review such agent-generated tests to ensure appropriate use of mocks and adequate test realism. This also opens opportunities for future research to assess whether such mocks impact test quality, maintainability, and coverage.

We recall that developers can customize the coding agent configuration files (e.g., `CLAUDE.md`), including commands, guidelines, and test instructions [13, 14, 16]. These configuration files can also contain mock-related information, such as mock instructions, mocking best practices, mock anti-patterns, mock examples, and mock tools. For example, the `CLAUDE.md` file of the repository browser-use/browser-use mentions: "*Never mock anything in tests, always use real objects!! The only exception is the LLM [...]*".[27] We found only five agent mock commits in this repository; 3 are indeed related to LLMs,[28] but 2 not seem to be in the scope of LLMs.[29] Likewise, the `copilot-instructions.md` of home-assistant/core contains: "*Mock external APIs - Use fixtures with realistic JSON data*".[30] Interestingly, this repository is among the ones with the most agent mock commits (61). The file `AGENTS.md` of apache/superset is very specific on what should be used:"*Mock patterns: Use MagicMock() for config objects, avoid AsyncMock for synchronous code*".[31] Table 12 details the frequency of test and mock occurrences in coding agent configuration files according to the GitHub Code Search. It shows that the presence of test-related instructions is relatively common, whereas mock-related instructions are less frequent.

Table 12: Frequency of test and mock occurrences in coding agent configuration files on GitHub (October 2025).

File	#	With test	With mock
`CLAUDE.md`	112k	102k	13k
`copilot-instructions.md`	44k	27k	7k
`CURSOR.md`	4.8k	1.3k	200

Implication: Given the tendency of coding agents to rely on mocking, practitioners should include guidance on mocking best practices and anti-patterns in agent configuration files to ensure more consistent test generation. Future research can leverage coding agent configuration files to explore whether the provided instructions are actually reflected in the agent-generated code.

4.3 Recent repositories contain a higher proportion of agent test and mock commits

In RQs 1 and 2, we also analyzed the proportion of test and mock commits in repositories created more recently, in 2025. We found that repositories created in 2025 present a higher proportion of agent test commits than the overall dataset (17% vs. 7%). Likewise, repositories created in 2025 present a higher proportion of agent mock commits than the overall dataset (19% vs. 9%). Given these trends, it is plausible that in the near future, a substantial portion of tests (and mocks) will be generated by coding agents.

Implication: The increasing share of test and mock commits by coding agents in recently created repositories suggests that agent-generated testing is becoming more common. This opens room for future research to investigate how the increasing presence of coding agents affects the evolution of testing practices over time.

4.4 Coding agents use less diverse test doubles

The testing literature proposes five test doubles: dummy, stub, spy, mock, and fake, each one with its characteristics [42]. In RQ3, we found that agents predominantly use the *mock* type (95%), whereas non-agents employ a wider variety: *mock* (91%), *fake* (57%), and *spy* (51%). In both cases, *dummy* and *stub* are the least adopted types.

Implication: The narrow use of test double types by coding agents may suggest limited diversity in their testing strategies. Practitioners should be aware that agent-generated tests may rely on a single form of test double (i.e., mock), potentially missing opportunities to simulate different isolation scenarios. Practitioners can provide more fine-grained instructions in agent configuration files about the rationales behind the usage of each test double.

[27] https://github.com/browser-use/browser-use/blob/0e925468bf/CLAUDE.md
[28] https://github.com/browser-use/browser-use/commit/cedb0d9cd4
[29] https://github.com/browser-use/browser-use/commit/c16bfa4f65
[30] https://github.com/home-assistant/core/blob/fa86148df0/.github/copilot-instructions.md
[31] https://github.com/apache/superset/blob/98fba1eefe/AGENTS.md

5 Threats to Validity

Detection of test and mock commits: To detect test commits, we analyzed multiple test file and directory patterns, which were extracted from the official documentation of popular testing frameworks, such as Pytest, unittest, Jest, Sinon.JS, and Jasmine. Our solution to detect mock commits was inspired by prior studies [22, 47], which also rely on code identifiers to detect the presence of test doubles in test code. Thus, based on the fact that we rely on official documentation and established solutions to detect tests and mocks, we minimize the risks of false positive test and mock commits.
Detection of agent commits: We started the process to detect agent commits by selecting repositories that contained agent-related files or directories associated with Claude, Copilot, or Cursor, which are three leading agents [35]. These three agents work by authoring or co-authoring commits. Next, we analyzed commit authors and co-authors to identify the presence of multiple coding agents, as described in Section 2.4. This step was necessary because repositories may employ multiple coding agents (e.g., Claude and Aider), each leaving distinct traces in their commits. Therefore, we minimize the risks of false negatives, that is, not finding real agent commits.
Agent commits with co-authors: Developers can attribute a commit to more than one author by adding `Co-authored-by:` trailers to the commit message [26]. Although GitHub recommends using the `Co-authored-by:` [26], we observed that coding agents may also use variants such as `Co-Authored-By:`, as reported in Section 2.4. Thus, we searched for these trailers in commit messages in a case-insensitive manner to reduce the risks of false negatives. Moreover, when a commit includes multiple co-authors, it becomes less clear how individual contributions are distributed. To minimize the risks, we reported multiple views of the data in Tables 5 and 8.
Generalization of the results. We analyzed more than 1.2 million commits made in 2025 across 2,168 repositories written in three programming languages (Python, JavaScript, and TypeScript). Moreover, we mined commits to identify the usage of numerous coding agent tools, including Claude, Copilot, Cursor, Aider, OpenHands, and Devin AI. Despite these observations, our findings – as usual in empirical software engineering – cannot be directly generalized to repositories written in other languages or using other agents.

6 Related Work

6.1 Coding Agents

Since coding agents are very new [51], few empirical studies of them exist (benchmark evaluations of coding agents, or studies of code completion LLMs are out of scope for this paper). Cursor, a coding agent integrated in a variant of Visual Studio Code [15], has been both popular and available for longer than other leading agents. It has been the focus of two studies. A controlled experiment by Becker et al. [8] focused on estimated and actual task completion time. Sixteen developers worked on 246 tasks in several large projects. The developers estimated that they were 20% more productive using Cursor. Surprisingly, the experiment found that task completion time *increased* by 19%, contrary to developer perception. The second study is an observation study by Kumar et al. [34], where 19 developers also used Cursor, working on 33 issues. They observed various collaboration strategies between developers and agents, from complete "one-shot" delegation, where developers send the entire task to the agent, up to more collaborative processes, where the task is divided into simpler subtasks that the agent works on.

Agent interaction logs are explored by Bouzenia and Pradel [10]. These logs come from less established tools (OpenHands [60], Autocoderover [63], and RepairAgent [9]) and are extracted from the SWE-bench benchmark, so they do not reflect real usage. The study analyzed 120 interaction logs submitted to the benchmark, comparing logs from successful and unsuccessful tasks across a variety of factors, such as the number of interaction steps and tokens used, the type of actions they took, and anti-patterns in failing logs.

6.2 Software Tests and Test Doubles

Ideally, test suites should have good quality to catch more bugs and protect against regressions [6, 18, 30, 32, 39]. When creating tests, developers may find dependencies that make the test harder to implement [47]. In this scenario, they can use test doubles (also known as mocks) to emulate the dependencies' behavior, contributing to making the test fast, isolated, and deterministic [22, 42, 47, 55, 56]. There are five types of test doubles: dummy, stub, spy, mock, and fake [42]. There exists a vast literature exploring mocking practices mainly in Java [36, 44, 47, 55, 56], but also in Android [22] and Python [57]. In the context of LLMs and mocking, an early study evaluated OpenAI's GPT-4o for automating mock decisions in Apache Dubbo. In a controlled experiment, the study compared model outputs with developer choices, finding that the model tends to generate more mocks than developers [49]. In our research, we leverage the traces left by agents in real-world software systems to study mock generation in 2,168 repositories.

7 Conclusion

This paper presented an empirical study to investigate the presence of mocks in agent-generated tests of real-world software systems. We analyzed over 1.2 million commits made in 2025 in 2,168 TypeScript, JavaScript, and Python repositories. Overall, we find that coding agents are more likely to modify tests and to add mocks to tests than non-coding agents. Finally, we concluded by discussing multiple implications for researchers and practitioners, such as the need to review agent-generated tests to ensure appropriate use of mocks and the call for guidance on mocking best practices and anti-patterns in agent configuration files.

As future work, we plan to conduct a more qualitative investigation of agent-generated tests and mocks, for example, to assess their quality and type, verify whether some mocks could be avoided, and identify whether they introduce harmful testing practices. Finally, we plan to explore the content of coding agent files (e.g., `CLAUDE.md`) to better understand their role in software testing and mocking.

Acknowledgments

This research was supported by CNPq, CAPES, FAPEMIG, and the French State. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), www.ines.org.br, CNPq grants: 408817/2024-0 and 403304/2025-3. This study also received financial support from the French State in the framework of the Investments for the Future programme IdEx université de Bordeaux.

References

[1] Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. 2024. TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved? *arXiv preprint arXiv:2412.02883* (2024).

[2] Parsa Alian, Noor Nashid, Mobina Shahbandeh, Taha Shabani, and Ali Mesbah. 2024. Feature-Driven End-To-End Test Generation. *arXiv preprint arXiv:2408.01894* (2024).

[3] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *International Symposium on Empirical Software Engineering and Measurement*. 1–7.

[4] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *International Conference on the Foundations of Software Engineering*. 185–196.

[5] Victor Anthony Alves, Cristiano Santos, Carla Bezerra, and Ivan Machado. 2024. Detecting Test Smells in Python Test Code Generated by LLM: An Empirical Study with GitHub Copilot. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 581–587.

[6] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4925–4946.

[7] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

[8] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv preprint arXiv:2507.09089* (2025).

[9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[10] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. *arXiv preprint arXiv:2506.18824* (2025).

[11] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *International Conference on the Foundations of Software Engineering*. 572–576.

[12] Claude Code. January, 2026. https://docs.claude.com/en/docs/claude-code/overview.

[13] Claude Code: Best practices for agentic coding. January, 2026. https://www.anthropic.com/engineering/claude-code-best-practices.

[14] Copilot: Best practices for using GitHub Copilot to work on tasks. January, 2026. https://docs.github.com/en/copilot/tutorials/coding-agent/get-the-best-results.

[15] Cursor Agent. January, 2026. https://cursor.com/features.

[16] Cursor Rules. January, 2026. https://cursor.com/docs/context/rules.

[17] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.

[18] Francisco Dalton, Márcio Ribeiro, Gustavo Pinto, Leo Fernandes, Rohit Gheyi, and Baldoino Fonseca. 2020. Is exceptional behavior testing an exception? an empirical assessment using java automated tests. In *International Conference on Evaluation and Assessment in Software Engineering*. 170–179.

[19] Juri Di Rocco, Phuong T Nguyen, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2025. DeepMig: A transformer-based approach to support coupled library and code migrations. *Information and Software Technology* 177 (2025), 107588.

[20] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 341–350.

[21] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[22] Mattia Fazzini, Chase Choi, Juan Manuel Copia, Gabriel Lee, Yoshiki Kakehi, Alessandra Gorla, and Alessandro Orso. 2022. Use of test doubles in android testing: An in-depth investigation. In *International Conference on Software Engineering*. 2266–2278.

[23] Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. 2025. Mutation-guided llm-based test generation at meta. *arXiv preprint arXiv:2501.12862* (2025).

[24] Roar Elias Georgsen. 2023. *Beyond Code Assistance with GPT-4: Leveraging GitHub Copilot and ChatGPT for Peer Review in VSE Engineering*. Technical Report. EasyChair.

[25] GitHub Copilot Coding Agent. January, 2026. https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent.

[26] GitHub: Creating a commit with multiple authors. January, 2026. https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors.

[27] GitHub: The state of open source software. January, 2026. https://github.blog/news-insights/octoverse/octoverse-2024.

[28] Google Testing Blog: Don't Overuse Mocks. August. https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html.

[29] Andre Hora. 2024. Predicting Test Results without Execution. In *International Conference on the Foundations of Software Engineering*. 542–546.

[30] Andre Hora. 2024. Test polarity: detecting positive and negative tests. In *International Conference on the Foundations of Software Engineering (FSE)*. 537–541.

[31] Andre Hora. 2026. GitEvo: Code Evolution Analysis for Git Repositories. In *International Conference on Mining Software Repositories (MSR 2026)*. In press.

[32] Andre Hora and Gordon Fraser. 2025. Exceptional Behaviors: How Frequently Are They Tested?. In *International Conference on Automation of Software Test (AST)*. IEEE, 70–79.

[33] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).

[34] Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson Murphy-Hill. 2025. Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild. *arXiv e-prints* (2025), arXiv–2506.

[35] Hao Li, Haoxiang Zhang, and Ahmed E Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. *arXiv preprint arXiv:2507.15003* (2025).

[36] Mengzhen Li and Mattia Fazzini. 2024. Automatically removing unnecessary stubbings from test suites. In *Conference on Software Testing, Verification and Validation*. IEEE, 233–244.

[37] Jenny T Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. 2023. Can GPT-4 Replicate Empirical Software Engineering Research? *arXiv preprint arXiv:2310.01727* (2023).

[38] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.

[39] Diego Marcilio and Carlo A Furia. 2021. How Java programmers test exceptional behavior. In *International Conference on Mining Software Repositories (MSR)*. IEEE, 207–218.

[40] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[41] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-driven development and llm-based code generation. In *International Conference on Automated Software Engineering*. 1583–1594.

[42] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[43] Mauricio Monteiro, Bruno Castelo Branco, Samuel Silvestre, Guilherme Avelino, and Marco Tulio Valente. 2023. End-to-End Software Construction using ChatGPT: An Experience Report. *arXiv preprint arXiv:2310.14843* (2023).

[44] Shaikh Mostafa and Xiaoyin Wang. 2014. An empirical study on the usage of mocking frameworks in software testing. In *International Conference on Quality Software*. IEEE, 127–132.

[45] Wendkûuni C Ouédraogo, Yinghua Li, Kader Kaboré, Xunzhu Tang, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. 2024. Test smells in LLM-Generated Unit Tests. *arXiv preprint arXiv:2410.10628* (2024).

[46] Harry Percival. 2014. *Test-driven development with Python: obey the testing goat: using Django, Selenium, and JavaScript*. O'Reilly Media, Inc.

[47] Gustavo Pereira and Andre Hora. 2020. Assessing mock classes: An empirical study. In *International Conference on Software Maintenance and Evolution*. IEEE, 453–463.

[48] Pytest. January, 2026. https://docs.pytest.org.

[49] Hanbin Qin. 2025. To Mock or Not to Mock: Divergence in Mocking Practices Between LLM and Developers. In *International Conference on Software Engineering: Companion Proceedings*. IEEE, 239–240.

[50] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2025. Agentic Much? Adoption of Coding Agents on GitHub. *Under submission* (2025).

[51] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2026. Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity. In *International Conference on Mining Software Repositories (MSR 2026)*. In press.

[52] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[53] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[54] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *Asia-Pacific Software Engineering Conference*. IEEE, 151–160.

[55] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? an empirical study on mocking practices. In *International Conference on Mining Software Repositories (MSR)*. 402–412.
[56] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing Java systems: Why and how developers use them, and how they evolve. *Empirical Software Engineering* 24 (2019), 1461–1498.
[57] Fabian Trautsch and Jens Grabowski. 2017. Are there any unit tests? An empirical study on unit testing in open source Python projects. In *International Conference on Software Testing, Verification and Validation*. IEEE, 207–218.
[58] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting Code Coverage without Execution. *arXiv preprint arXiv:2307.13383* (2023).
[59] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. *arXiv preprint arXiv:2402.16480* (2024).
[60] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).
[61] Tao Xiao, Youmei Fan, Fabio Calefato, Christoph Treude, Raula Gaikovina Kula, Hideaki Hata, and Sebastian Baltes. 2025. Self-admitted GenAI usage in open-source software. *arXiv preprint arXiv:2507.10422* (2025).
[62] Quanjun Zhang, Chunrong Fang, Siqi Gu, Ye Shang, Zhenyu Chen, and Liang Xiao. 2025. Large Language Models for Unit Testing: A Systematic Literature Review. *arXiv preprint arXiv:2506.15227* (2025).
[63] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.