What Types of Automated Tests do Developers Write?

Marko Ivanković*§, Luka Rimanić†, Ivan Budiselić†, Goran Petrović†, Gordon Fraser*, and René Just‡§
*University of Passau, Passau, Germany
†Google Switzerland GmbH, Zurich, Switzerland
‡University of Washington, Seattle, WA, USA
§Work done at Google.
marko@ivankovic.me, {rimanic, ibudiselic, goranpetrovic}@google.com, gordon.fraser@uni-passau.de, rjust@cs.washington.edu

***Abstract*—Software testing is a widely adopted quality assurance technique that assesses whether a software system meets a given specification. The overall goal of software testing is to develop effective tests that capture desired program behaviors and reveal defects. Automated software testing is an essential part of modern software development processes, in particular those that focus on continuous integration and deployment. Existing test classifications (e.g., unit vs. integration vs. system tests) and testing best practices offer general conceptual frameworks, but instantiating these conceptual models requires a definition of what is considered a unit, or even a test. These conceptual models are rarely explicated in the literature or documentation which makes interpretation and generalization of results (e.g., comparisons between unit and integration testing efficacy) difficult. Additionally, comparatively little is known about how developers operationalize software testing in modern industrial contexts, how they write and automate software tests, and how well those tests fit into existing classifications. Since software engineering processes have substantially evolved, it is time to revisit and refine test classifications to support future research on software testing efficacy and best practices. This is especially important with the advent of AI-generated test code, where those classifications may be used to automatically classify the types of generated tests or to formulate the desired test output.**

This paper presents a novel test classification framework, developed using insights and data on what types of tests developers write in practice. The data was collected in an industrial setting at Google and involves tens of thousands of developers and tens of millions of tests. The developed classification framework is precise enough that it can be encoded in an automated analysis. We describe our proof-of-concept implementation and report on the development approach and costs. We also report on the results of applying the automated classification to all tests in Google's repository and on what types of automated tests developers write.

I. Introduction

Software testing is generally accepted as an important quality assurance technique in software engineering, but it remains more art than science [1]. While the general concepts of software testing, including high-level distinctions such as unit tests vs. integration tests vs. system tests appear relatively straightforward, the exact details of how each term is defined in practice varies from work to work and often involves significant subjective human judgement. As a result, scientific conclusions about the efficacy, value, and cost of software testing are subject to a specific conceptual model of what is considered a test, which is often not explicated. Arguably, generalizations have become even more difficult as software engineering processes have evolved. For example, tests developed as part of a test-driven-development (TDD) approach are different from those that are developed by test engineers in a dedicated testing phase of a waterfall model or tests developed to reproduce and debug a reported defect.

Perhaps the most common ambiguity is what the definition of a *unit test* is. While existing textbook definitions make sense at a high level, they are not comprehensive or precise enough to be directly instantiated and used in a given industrial or open source contexts. Notably, what exactly constitutes a "unit" is often under-specified. For example, in an object oriented system, is a *unit* a method, a class, a package (i.e., a collection of related classes), or something else? Similarly, is a test that executes more than one unit an integration test, even if one of the units is part of the language's standard library?

Many test-execution frameworks are termed "unit-testing frameworks", such as JUnit (and others in the xUnit family), Jasmine, or Go's framework around the "go test" command. While it seems plausible that the developers of unit-testing frameworks had a particular notion of unit testing in mind when designing and developing them, the tests that framework users write and execute vary widely in complexity and what properties and behaviors they are testing. For example, in the context of Java, software *libraries* such as the Apache Commons [2] libraries commonly use the JUnit framework to test individual methods with input-output pairs. In contrast, software *systems* such as Closure Compiler [3] rely on the JUnit framework for automation and integration into build systems. Many of the JUnit tests in Closure Compiler are arguably neither unit nor integration tests because they run the entire compiler end-to-end, asserting on the compiler's behavior (e.g., error reporting vs. success) as opposed to the correctness of the compiled output or intermediate results.

This ambiguity is further amplified in day-to-day development where the terms testing and unit testing are used somewhat interchangeably, resulting in confusing terminology and a lack of understanding of what software testing entails. Recently, such ambiguity has lead to an entire new set of problems: creating training and evaluation datasets for

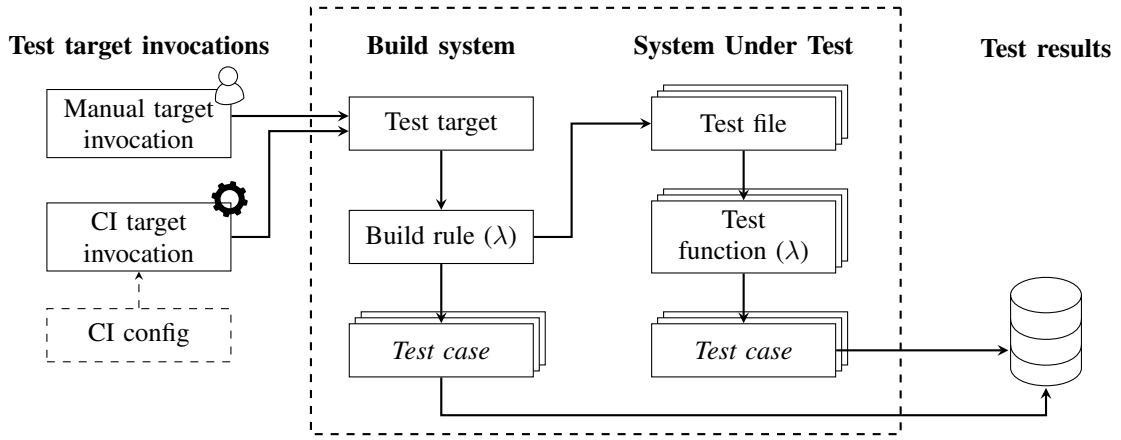

Fig. 1. Individual test cases, either defined in a build-system rule or in a test function. The λ indicates parametrization.

learning-based artificial intelligence (AI) systems. As AI is increasingly used to generate code, including tests, precise classification rules that can be automated to classify those tests are more important than ever. Our work is motivated by this use case and the ambiguity of existing classifications. In this paper, we propose a new classification that is precise enough to be implemented as an automated analysis. Our proof-of-concept implementation is rule-based and efficient enough that it can analyze tests in real time (e.g., in developers' IDEs). The classification can also be used for other purposes, for example to improve generalizability and comparability of research results or to streamline software testing efforts.

A. Contributions

This paper explores the following research questions:

RQ1: What types of automated tests do developers write?
RQ2: Are test types amenable to automated classification?
RQ3: How common are different test types in practice?

Its main results are as follows:

RA1: Developers encode a wide range of dynamic and static analyses on top of common test automation frameworks (e.g., the xUnit family), implying a broad definition of what constitues a test. These automated tests cover artifacts such as configuration, data, and documentation, going well beyond source-code tests. Based on these observations, we developed a test type classification that covers eight categories of test characteristics.

RA2: It is possible to encode our test type classification in an efficient automated analysis. Reaching a 50% or higher classification rate for each test type category across the entire repository required 15k lines of Go code and approximately 9 engineer-months of effort. The classification pipeline processes millions of tests in about 1 hour of real time on consumer-grade hardware.

RA3: Tests with unorthodox properties, counter to what is commonly described in the software testing literature, are very common in practice. For example, we observed many tests that cover configuration, do not execute the AUT, and favor real dependencies over mocks.

II. Research Setting

Our research was done in an industrial setting at Google, which uses software development and testing practices that are well established and similar to those of other companies of similar size [4], [5], [6], [7], [8], [9].

Google's development environment is based on a monorepo of more than one billion lines of code. It operates a distributed build and test system that is based on an internal, modified version of the Bazel [10] build system.

A. Tests in Bazel

Figure 1 shows a conceptual model of how testing is supported by the Bazel build system. There are two types of test cases: (1) test cases implemented as build rules using the Bazel's Starlark language and (2) test cases implemented in test files as part of the system under test (SUT), usually written in the SUT's programming languages.

Test cases implemented in the build system map to a single build rule but parametrization allows for different instantiations of the same build rule. Instantiating a build rule involves the following steps:

- *Test target*: A Bazel target, defined in a `BUILD` file.
- *Build rule*: Typically one of the common Bazel `*_test` rules. However, users are not limited to the common rules and they can and implement custom rules and macros. One notable property is that a single rule or macro can itself expand into multiple rules. In the context of testing, this allows for easy creation of testing frameworks in the build system itself, where the developer explicitly listing a single rule can result in multiple final test rules being produced. Common examples include testing of multiple web browsers or testing on multiple architectures.
- *Test case*: A set of steps and assertions defined in a build rule. It is possible to write complete tests in Bazel's domain specific language Starlark (a Python dialect). An example of this is a test that simply checks that an artifact under test successfully builds.

Test cases implemented in test files as part of the SUT tend to be hierarchical (and are instantiated after the build rule):

- *Test file*: A single file checked into the repository, typically named `<name>_test.*`.
- *Test function*: A function in a test file that is executed by a testing framework. In gUnit in C++ this would be a function defined using the `TEST` macro, in JUnit in Java it would be a method annotated with `@Test`, etc.
- *Test case*: A single executable test function after parameter expansion, e.g,. parameterized unit tests in Java or table-based tests in Go expand into n individual test cases, where n is the number of test data or table entries.

B. Test results

By default, all tests executed at Google store detailed build and execution logs in a central test result storage system. While it is theoretically possible for a developer to specifically modify multiple parts of their development environment to prevent this, it is exceptionally rarely done. It is safe to assume that more than 99.9% of test executions are available in the central storage system. This includes, but is not limited to, tests executed locally during development, tests executed by the review system during code review, tests executed by continuous integration, tests executed during release verification and tests executed in production (typically against a canary instance during a gradual roll-out process).

C. Scale

At Google, the repository contains millions of test targets and test files. Some test files are re-used across multiple test targets, for example the same test being reused across several Web browsers. Billions of test executions are recorded per day. Any classification that requires a human in the loop for such a large number of tests would be impractical. The change rate of tests would exceed any reasonable efforts, so even if the existing tests could be manually classified, by the time the classification finished the results would be obsolete. This motivates our research into the characteristics of the existing tests and whether it is possible to develop an automated classification for them.

III. Test Type Classification

RQ1: What types of automated tests do developers write?

A. Existing classification frameworks are inadequate

Our ultimate goal is to develop an automated classification system for test types based on a test's implementation and characteristics. This required us to define distinct types of tests and associated rules for classification.

Our initial methodology attempted to reuse existing classifications and labels, and consisted of three high-level steps:

1) *Stratification*: Bucket all test targets based on metadata (developer chosen name and tag, and Bazel rule type).
2) *Sampling*: Sample 10 test targets from each labeled bucket and 30 test targets from the bucket of unlabeled test targets. The samples were combined and randomly shuffled at the end of the sampling process.
3) *Classification*: Three developers, each with more than a decade of experience, attempted to independently classify the sampled test targets, without access to the labels of the stratification buckets to avoid bias.

1) Stratification: From the test results storage, we selected all test targets with at least one execution in the month of October 2023 to ensure that the tests we were examining are actually used, and not just dead code in the repository. This resulted in millions of unique test targets. Given the set of all test targets, each with a name and a set of tags, we iteratively defined classification rules that capture large fractions of test targets. We independently classified test targets based on their name, rule type and tags. We based this on the intuition that a developer may indicate the type of the test, e.g., integration test, by using the words "integration test" in the target name; likewise a test rule might indicate it is used to define web tests by adding a "web test" tag. Each metadata type was used to classify targets on its own. This allowed us to cross-check the classification rules and detect errors easily, by making sure that the classifications based on different metadata match.

This iterative process yielded the following seven major categories: (1) Unit test, (2) Integration Test, (3) UI Test, (4) Conformance Test, (5) Config Test, (6) Data Test, and (7) Build Test. We assigned a default category ("Not classified") to all test targets that did not match any of the seven categories.

2) Sampling: In total we sampled 100 targets. From each of the seven categories, we randomly selected 10 targets. From the not-classified category, we randomly sampled 30 targets.

3) Classification: Three developers began to independently classify the sampled targets—following the same order of randomly shuffled targets. The developers had full access to the entire development environment, including the source code of both product and tests, documentation, and other artifacts.

During a first discussion of a little under 25% of all targets, however, it became clear that the categories typically used in the literature, and also captured by test names and tags, are subjective and insufficient. We made two specific observations.

First, there is little to no agreement across the three independent classifications and between those and the bucket labels. For example, one developer considered integration tests to assert on the protocol between units, whereas another developer considered integration tests to simply execute multiple units in the same test. An unresolved question asked whether a unit or integration test has to be a dynamic analysis. Yet another question revolved around UI tests—whether these correspond to using the UI as a test driver vs. testing the UI itself.

Second, the lack of agreement and precise definitions of the classification categories meant that the classification was far too subjective to ever be encoded efficiently and precisely.

Given our observations, we concluded that classifying all 100 targets would not be a useful exercise. Instead, we used these observations and sampled targets to iteratively develop a novel classification framework based on test characteristics.

B. Methodology

Our methodology was informed by our first (failed) attempt: Existing test classifications are ambiguous and subjective. While they capture properties of the tested system and the

used testing frameworks, they capture very little of the test case implementations and characteristics.

1) We decided not to use any existing classification labels.
2) We randomly selected a much larger, 1000 target, sample—to iteratively develop our new classification.
3) We iteratively implemented automated classification rules and refined the classification framework. The automated classification was implemented in the monorepo itself and followed strict production code guidelines that are in place at Google.

The 1000 test targets were sampled uniformly at random from the test result repository, independently of any test target data or metadata. No targets were excluded. Even test targets that are technically outside of the monorepo were included, such as test targets for Google's open source projects.

C. Classification

Our classification is guided by the following 8 questions:

(1) What Artifact Under Test (AUT) is being tested?

- *Code*: Test analyzes an AUT and asserts on functional or non-functional properties. *Example*: Testing a function with input-output pairs, testing for robustness, verifying the absence of vulnerable dependencies, etc.
- *Config*: Test analyzes configuration external to an AUT and asserts on properties of the configuration itself. *Example*: Testing that all deployment configurations specify datacenters that actually exist.
- *Data*: Test analyzes data external to an AUT and asserts on properties of the data. *Example*: Testing that the timezone database shipped with the application does not contain timezones with physically impossible offsets.
- *Documentation*: Test analyzes developer or end-user documentation of an AUT and asserts on properties of the documentation. *Example*: Testing whether documentation exists and/or parses.
- *Hardware*: Test analyzes physical hardware devices or description of hardware in a hardware description language (e.g., VHDL). Note that only tests whose failure can result in an actual physical device being changed are considered to be testing hardware. Tests for firmware would be considered to be testing code instead, unless the firmware test failing could lead to, e.g., a prototype device being sent back for re-soldering. *Example*: Test that asserts on the voltages of the ports of a VHDL architecture.

(2) Is the test executing the AUT?

- *Dynamic*: Test analyzes an AUT by executing it.
- *Static*: Test analyzes an AUT without executing it.

(3) How does the test interact with the AUT?

- *API*: Test controls an AUT's state through *API calls*. *Example*: Calling a function directly from the test code or calling an RPC endpoint.
- *GUI*: Test controls an AUT's state through *GUI interactions*. *Example*: WebDriver tests simulating clicks on a webpage in a real web browser.

(4) Is the test performing a numerical measurement?

- *At build-time*: Test measures some property of the AUT that does not require the AUT to be executed. *Example*: Measuring the size of the compiled binary.
- *At run-time*: Test measures some property of the AUT that requires execution. *Example*: Measuring memory or CPU usage.

(5) How are outputs analyzed?

- *As-is*: Test analyzes an AUT's output as-is. *Example*: Directly asserting on the output of a function under test.
- *Post-processed*: Test analyzes an AUT's output after post-processing. *Example*: A screenshot test partially masking the screenshot before comparing it to an expected image.

(6) What type of assertions are done on the output of the AUT?

- *Existence*: Test asserts on the existence of an artifact or an output. *Example*: Asserting that the output value of a single function call is not null.
- *Constrained*: Test asserts on necessary correctness conditions. *Example*: Asserting that the output value of a single function call is a positive number.
- *Exact*: Test asserts on sufficient correctness conditions. *Example*: Asserting that the output value of a single function call is exactly the number 16112013.

(7) What type of assertions are done on the implementation or the protocol of the AUT?

- *Existence*: Test asserts that a protocol interaction occurs or that a structural pattern exists. *Example*: Asserting that the binary does not use a specific dependency. Asserting that calling a chain of methods does not throw an error.
- *Constrained*: Test asserts on necessary correctness conditions. *Example*: Asserting that a mocked method was called three or more times.
- *Exact*: Test asserts on sufficient correctness conditions. *Example*: Asserting that a mock was called exactly n times with arguments that exactly match expectations.

(8) How does the test handle dependencies?

- *Mocked dependency*: Test uses mocks. It explicitly scripts the interaction with a dependency.
- *Fake/Test dependency*: Test uses fake implementations. *Example*: A fake in-memory implementation of a database that offers the same API as the real dependency, but offers no consistency and reliability guarantees.

Almost all combinations of question responses are possible, although some are far more likely than others. For example, while it is theoretically possible to construct a test that asserts on outputs without executing the AUT, it would be quite unusual. Instead, one would expect that execution would always be positive if any assertion is positive. Similarly, some attributes are likely to be exclusive, e.g., documentation tests are extremely unlikely to perform run-time measurements. But we do not a priori exclude any combination.

D. Data coding and aggregation

For each question, each option can be answered as:

- *Yes*: We have conclusive positive evidence that the test has a property.
- *No*: We have conclusive negative evidence that the test does not have a property.
- *Inconclusive*: We do not have conclusive positive or negative evidence.

Note that for each question, each option is evaluated independently of other options. This means that it is possible for a question to have more than one answer. For example, "What AUT is being tested?" could be "Code" and "Data" for the same test. Similarly, it is possible for a single test to use both the API and the GUI to test both code and configuration. It is also possible for a test to not use any options in a question. For example, a test that does not perform any measurement would simply have both measurement options set to "No".

We summarize the options for each question as follows:

- If any of the options in the question cannot be conclusively determined, then we summarize the entire question as either "Inconclusive" or "Not classified".
- If all options are conclusively negative, then we summarize the entire question with a grammatically appropriate version of the term "None". For example, if a test is not performing any measurements (neither built-time nor run-time measurement), then question (4) would be summarized as "None".
- If only one option is conclusively positive and all other options are conclusively negative, we summarize the entire question with a grammatically appropriate version of the option. For example, if a test is only testing code, then question (1) would be summarized as "Code".
- If multiple options are conclusively positive, we summarize the entire question as "Mixed", "Multiple", "Both" or similar grammatically appropriate name.

RA1: Developers encode a wide range of dynamic and static analyses on top of common test automation frameworks (e.g., the xUnit family), implying a broad definition of what constitues a test. These automated tests cover artifacts such as configuration, data, and documentation, going well beyond source-code tests.

IV. Automated Classification of Test Types

RQ2: Are test types amenable to automated classification?

Using the sample of 1000 test targets as a reference, we designed and implemented an automated classification pipeline that processes all test targets in the repository.

A. Operationalization of test type classification

Each test target is classified at two levels: the target level itself and optionally the file level, for targets with source test files that are accessible to the pipeline. Many targets do not have any source test files, and very few have source test files in siloed code. The file level classifications are first merged into a unified file-level classification, which is then merged with the target-level classification into a final classification. When merging classifications, there are two types of merge operations: merging two file-level classifications and merging file- and target-level classifications. Because of the hierarchical nature of files and targets, there are subtle differences in how these operations are performed.

TABLE I
Classification merge operation

File	File or Target	File + File	File + Target
Inconclusive	Inconclusive	Inconclusive	Inconclusive
Positive	Positive	Positive	Positive
Negative	Negative	Negative	Negative
Inconclusive	Positive	Positive	Positive
Inconclusive	Negative	Inconclusive	Negative
Positive	Negative	Positive	Error
Negative	Positive	Positive	Error

Table I shows the function for the merge operations. When two files are merged and one of them is inconclusive and the other is negative, we cannot determine the merged value. If future analysis would conclude that the missing value is positive, the merged value would be positive, but if it was negative it would be negative. In contrast, if an inconclusive file is merged with a conclusively negative target, the merged classification must be negative.

If two file classifications are merged, and one is conclusively positive and the other negative, the merged classification is positive. An example that helps demonstrate this is a target that has two files, one for code and one for configuration. The file level classifications will set code and classification to true and false as applicable, but the merged classification should set both to true.

If the unified file classification is being merged with the target, then the conclusively set values must match. For example, if the test files are asserting on the output, the target must transitively also be asserting on the output since it consists of the files. If the classification derived from the target and the merged classification derived from the files conflict, this is an error in the analysis pipeline code and must be fixed. We used this property extensively during development, with continuous tests detecting any such conflicts at code review time and flagging it immediately.

1) Target-level Classifications: Each target has metadata that can be used for classification. In principle, any data that can be obtained through the Bazel Query Language was available to the classification pipeline. In practice, the data we found most useful was the Bazel rule class, list of source files, name of the test target, the test binary or in Java the test class and the "instantiation stack" (i.e., for rules implemented in Skylark, the call stack of Bazel functions that the target implementation makes).

TABLE II
CONCLUSIVE CLASSIFICATIONS

Category	Repository	Sample
What is being tested	78.4 %	95.1 %
Is the test executing the AUT	76.0 %	93.7 %
Interaction with the AUT	67.3 %	90.3 %
Is the test performing a measurement	66.2 %	89.7 %
How are outputs analyzed	64.5 %	87.1 %
Assertions on the output	61.1 %	87.1 %
Assertions on the implementation	57.3 %	86.5 %
How are dependencies handled	53.3 %	88.8 %
All categories simultaneously	38.4 %	85.0 %

2) Test-file Classifications: In the sample of 1000 targets we managed to extract and access a total of 126 test files that spanned 288 test targets. Note that a single test file can be shared across many different targets (e.g., a web test that is tested across various browsers, or a shell script that tests whether a project builds). These test files were used to guide our development of Abstract Syntax Tree (AST)-based classifiers that automatically classify files.

B. Evaluation

We did not attempt to reach 100% classification in all categories on our sample for two reasons. First, while test results are available, we are unable to access siloed code for a handful of test targets. Second, we expect a certain percentage of tests to be simply too complex to classify in a reasonable amount of time, for developers unfamiliar with the corresponding project. While we did encounter some such tests, where multiple hours of code reading was required to classify them, the overall number was surprisingly small (about 5%). Also note that some categories are harder to classify than others—artifact under test was the easiest, whereas type of assertions on the implementation or the protocol was the hardest.

With these limitations in mind, we chose 85% as our goal for full conclusive classification of test targets in our sample, across all eight categories. While this did leave 150 targets with partially incomplete classifications, we found that it provided us with a sufficient sample that can be effectively used to develop automated classification rules. This dataset is also used to continuously test the automated pipeline. In the future, will both improve the classification and expand the dataset as needed to keep the test coverage at appropriately high levels. After reaching the goal of 85%, we ran the classification pipeline on all test targets in the repository.

Table II shows the percentage of test targets our classification pipeline conclusively classified for each category, as of time of writing. *Repository* corresponds to all test targets in the repository; *Sample* corresponds to the 1000 test targets in our random sample.

Considering all targets in the repository, the pipeline automatically classifies more than 53% of them for each individual question and more than 38% of them for all questions.

For about 20% of all targets in the repository, the classification pipeline identified them as a test target but was unable to extract any information for classification. Given that the sample we used to develop our classification represents much less than 1% of all targets, this is an expected result.

Based on these results, we conclude that it is possible to specify a test classification that is precise enough to be encoded in an automated analysis. While a 53% classification rate is already useful for our use cases (e.g., real-time analysis of tests generated by large language models), we are continuously improving our classification pipeline. An automated classification of 100% of all tests, however, seems unrealistic.

C. Development cost and complexity

We developed the analysis pipeline over a period of 6 months. At any given point, between 1 and 3 developers were actively working on the pipeline code. The total development cost of the pipeline is roughly 9 engineer-months.

The pipeline consists of roughly 15k lines of Go code across about 100 files. This includes tests for the pipeline itself, but excludes test data. The pipeline was developed over a series of about 500 reviewed merge requests.

The quality of this automated pipeline meets deployment and production standards. All merge requests were reviewed by at least one, but usually two, other developers, to ensure that there is agreement on the classification. The code review process requires unanimous agreement of all reviewers and the code author. In case of disagreement, we refined the classification framework and rules, and added corresponding tests, until unanimous consensus was reached.

Additionally, we used strict testing standards, with the classification code being continuously tested during all stages of the development process. The line-based test coverage of the classification code was always kept at or above 90%.

RA2: It is possible to encode our test type classification in an efficient automated analysis. Reaching a 50% or higher classification rate for each test type category across the entire repository required 15k lines of Go code and approximately 9 engineer-months of effort. The pipeline is efficient: it processes millions of tests in about 1 hour of real time on consumer-grade hardware.

V. DISTRIBUTIONS OF TEST TYPES

RQ3: How common are different test types in practice?

The sample analysis and the automated classification of all tests in the repository allow us to investigate how common different test types are in practice. (Section VI speaks to the generalization to open source projects.)

(1) What Artifact Under Test (AUT) is being tested? Table III shows the distribution of the artifacts under test. The first surprising result is the percentage of configuration tests. To the best of our knowledge, nobody has previously reported such results and the literature commonly equates software testing with testing code. That configuration tests outnumber code tests is a finding that encourages further research.

Note that the "hardware" category is missing from the table. This is due to the fact that all hardware test that the pipeline classified were also testing at least some code at the same time—hence, all hardware tests are counted under the "Multiple AUT types" category.

It is also interesting to note that some tests do not test anything (the "Nothing" category). An example of a such a test is a test file specifically created so that it always passes and in effect skips a test if the test execution was requested for unsupported hardware.

(2) Is the test executing the AUT? Table IV shows the percentage of tests that execute the AUT. We found the fact that at least 56% of tests do not execute the AUT to be quite surprising and at odds with the common definition of a test.

Table V shows the percentage of tests that execute the AUT, limiting only to tests that test configuration or code. Unsurprisingly, most tests that test configuration do not execute it. Partially, this can be because some configuration languages are not even strictly speaking executable, however even tests for configuration written in Turing complete languages seem not to need execution in order to assert on configuration properties.

Somewhat more surprisingly, only a little over half of all code tests need to execute the code under test. Since we used the sampled targets to develop the classification directly, we can report the types of tests we directly observed that do not execute the AUT. The majority of such tests come in the form of static analysis that is using the testing infrastructure for the benefits of integration with the developer workflow and ease of use. The canonical example would be a linter that reads the source code under test and reports a passing test if the code does not have any reported errors, and a failing test otherwise. Most textbook definitions of testing would exclude such analysis, however we have found in our analysis that developers in practice do refer to such analyses as tests. We found this an important distinction between theory and practice.

(3) How does the test interact with the AUT? Table VI shows how tests interact with the AUT. Taking into account the observations around what AUTs are being tested and whether the AUTs are executed, it is not surprising that most tests do not interact with the AUT at all. Even if only code tests are observed, tests that do not interact with the AUT are still the most numerous. We can also see that about 3 to 4 times as many tests written by developers use the API compared to the GUI, and a small number of tests use both.

(4) Is the test performing a numerical measurement? Table VII shows the percentage of tests that perform a measurement. Most tests do not perform a measurement; only about 2% of all tests do. About 1% of tests measure a build-time property and about 1% measure a run-time property; only 0.3% measure both types properties.

(5) How are outputs analyzed? Table VIII shows how the outputs of the AUT are analyzed. As expected, given the findings about other categories such as AUT execution, most tests do not analyze the output of the AUT. Note that many of these tests do assert on the implementation or the protocol.

TABLE III
WHAT IS BEING TESTED?

Tested	Repository	Sample
Configuration	42.5 %	45.2 %
Code	29.5 %	39.9 %
Multiple AUT types	4.0 %	5.2 %
Data	0.7 %	1.2 %
Documentation	0.2 %	0.2 %
Nothing	1.4 %	1.9 %
Not classified	21.6 %	4.7 %

TABLE IV
IS THE TEST EXECUTING THE AUT?

Execution	Repository	Sample
No	56.1 %	66.1 %
Yes	19.6 %	26.4 %
Not applicable	0.2 %	1.2 %
Not classified	24.1 %	6.3 %

TABLE V
IS THE TEST EXECUTING THE AUT (CONFIGURATION OR CODE)?

Execution	Repository	Sample
What is being tested: Configuration		
No	79.4 %	88.7 %
Yes	8.2 %	6.9 %
What is being tested: Code		
No	40.0 %	41.6 %
Yes	52.5 %	57.1 %

TABLE VI
HOW DOES THE TEST INTERACT WITH THE AUT?

Interaction	Repository	Sample
No interaction	60.4 %	70.2 %
API	3.7 %	14.4 %
GUI	2.9 %	4.1 %
Both API and GUI	0.0 %	0.4 %
Not applicable	0.2 %	1.2 %
Not classified	32.8 %	9.7 %

TABLE VII
IS THE TEST PERFORMING A NUMERICAL MEASUREMENT?

Measurement	Repository	Sample
No measurement	64.8 %	86.0 %
Build-Time	0.6 %	1.2 %
Run-Time	0.2 %	1.0 %
Both Build-Time and Run-Time	0.3 %	0.3 %
Not applicable	0.2 %	1.2 %
Not classified	33.9 %	10.3 %

TABLE VIII
HOW ARE OUTPUTS ANALYZED?

Outputs	Repository	Sample
Not analyzed	62.5 %	75.2 %
As is	1.8 %	11.1 %
Post-processed	0.0 %	0.1 %
Both	0.0 %	0.1 %
Not applicable	0.2 %	1.4 %
Not classified	35.5 %	12.1 %

TABLE IX
WHAT ASSERTIONS ARE DONE ON THE OUTPUT OF THE AUT?

Assertions	Repository	Sample
No assertions	60.5 %	74.1 %
Exact match	0.3 %	7.8 %
Mixed	0.0 %	2.4 %
Constrained	0.0 %	1.2 %
Existence	0.0 %	0.2 %
Not applicable	0.2 %	1.4 %
Not classified	39.0 %	12.9 %

TABLE X
WHAT ASSERTIONS ARE DONE ON THE PROTOCOL OR IMPLEMENTATION OF THE AUT?

Assertions	Repository	Sample
Existence	37.4 %	42.8 %
Mixed	8.5 %	13.5 %
No assertions	2.4 %	13.3 %
Constrained	7.1 %	11.2 %
Exact match	1.6 %	4.5 %
Not applicable	0.2 %	1.2 %
Not classified	42.8 %	13.5 %

TABLE XI
HOW DOES THE TEST HANDLE DEPENDENCIES?

Dependencies	Repository	Sample
Real dependencies	48.3 %	79.9 %
Fakes	2.0 %	3.6 %
Mocks	2.3 %	3.3 %
Both fakes and mocks	0.5 %	0.8 %
Not applicable	0.2 %	1.2 %
Not classified	46.7 %	11.2 %

Of the tests that do analyze the output, almost all assert on the output directly, without manipulating it in any way. Only a tiny fraction (0.1%) of all test, post-processes the output. In our sample, all tests that post-process the output were using screenshots to verify visual outputs and all of them used masking to modify the screenshots.

(6) What type of assertions are done on the output of the AUT? Table IX shows what assertions are done on the output. Notably, there is a clear preference towards exact-match assertions.

(7) What type of assertions are done on the implementation or the protocol of the AUT? Table X shows what assertions are done on the protocol or the implementation. In comparison to assertions on output, there is a clear preference for existence and constraint assertions.

(8) How does the test handle dependencies? Table XI shows how tests handle dependencies, indicating that the use of real dependencies is by far most common. Note that at Google avoiding the use of fakes and mocks in favor of more realistic and effective tests is common [4].

RA3: Tests with unorthodox properties, counter to what is commonly described in the software testing literature, are very common in practice. For example, we observed many tests that cover configuration, do not execute the AUT, and favor real dependencies over mocks.

VI. GENERALIZATION TO OPEN SOURCE PROJECTS

Given that we developed heuristics for automated classification based on the 1000 target sample which consists almost entirely of proprietary Google code, we expect that generalizing the automation to open source projects requires additional effort, which we leave for future work.

However, Google's code repository does contain a large number of so-called "third-party packages" which are either imported from open source, exported to open source, or some combination of the two. Development on these packages happens either externally or internally, with various levels of involvement from Google's developers. Testing in these packages depends significantly on the package's development model. When the package is primarily developed in open source and uses a build system incompatible with Google's internal build system, test targets must be explicitly defined when the package is imported, which is often done using a language-specific build rule that simply globs all of the package's test files, but this is inherently limited to simpler tests. Conversely, when the package is primarily developed internally in Google's repository and exported to open source, tests can utilize more of the facilities of Google's build system, and look more similar to what is found in the proprietary part of the repository.

While these third-party packages are not a representative sample of open source projects, they still make up a large repository of open source code that our pipeline is already able to analyze. While the test targets in these third-party packages make up only a small fraction of all test targets in the repository, this is still hundreds of thousands of targets, and analyzing classification results provides useful directions for future generalization.

When classifying what is being tested on the test targets in the third-party package, we see two significant differences compared to the full repository. First, our automation is unable

to classify 48.1% of third-party test targets, while on the full repository, only 12.6% of targets remain unclassified. Second, the automation classifies only 1.3% third-party tests as configuration tests, while 42.5% of tests in the full repository are configuration tests. Instead, third-party tests are dominated by 45.6% code tests, whereas the full repository only has 29.5% code tests.

We conjecture that these findings are a result of a combination of effects, which is supported by manual inspection of a few dozen third-party packages:

- Our pipeline's ability to classify tests currently relies on target-level properties to a large degree. On the full repository, our automation is only able to extract at least one test source file for 23% of targets, meaning that all the classification signal on the remaining 77% of targets comes from target-level properties. Even for the targets that do have extracted source files, 84% of code or configuration classification is influenced by target-level properties. Conversely, for third-party test targets, 30% have extracted source files, and even there, 52% of code or configuration classification is influenced by target-level properties. The fact that third-party packages predominantly use simpler build rules that do not carry sufficient signal for conclusive classification contributes significantly to a much larger fraction of unclassified targets.
- Third-party packages tend to contain more libraries, and production projects are much less common than in Google's full repository. As such, configuration tests are expected to be less common, and even when they exist, our automation is much less likely to be able to classify them if they do not use standard frameworks and build rules that dominate internal configuration testing.

Further improving automated classification on open source projects requires dedicated effort on supporting different build systems and more complete source code analysis. We leave a deeper investigation for future work.

VII. Discussion

A. Implications

Our analyses and findings have two major implications.

First, a large number of tests the developers actually write are using test-automation frameworks and infrastructure in ways the framework authors may not have intended. We conjecture, based on our findings and experience, that this is due to developers finding such frameworks and infrastructure convenient and familiar. Given our findings, it seems worthwhile for framework authors to keep a broad notion of software testing in mind when developing them.

Second, a large number of tests are outside of the commonly researched areas. As an example, while test adequacy techniques such as code coverage [7] and mutation testing [11], [12] are very well researched for code tests, they are almost nonexistent for configuration tests, even though we have found configuration tests to be predominant. Future research into actual and desirable properties of such tests is needed.

B. Limitations

This qualitative research requires significant expertise across multiple programming languages, software system domains, test-automation frameworks, build systems, etc. Five out of six authors have a decade or more experience in software development, and four have a decade or more experience specifically in software testing and analysis. While we do acknowledge that nobody can be an expert in all of software engineering, we believe that our research team represents a diverse set of viewpoints and covers a wide range of software engineering expertise.

This paper is limited to automated, repeatedly executed tests. Manual testing, exploratory testing, user acceptance testing and other non-automated or non-repeated tests may not fit into the classification we propose, although some generalization is possible.

The research presented in this paper was conducted in a single industrial setting, at Google. While developers, programming languages, and tools all form a large and diverse sample, it may still be too company-specific. We encourage other researchers and practitioners to replicate our study in their context.

C. Future Work

Improving the classification rules to reach 90% precision and recall on the full dataset with millions of test targets is our next step. Once we reach that milestone, we will be able to cross-reference the individual test runs stored in the test result storage and explore material differences in execution rates, pass rates, coverage, mutation testing score and other test metrics using the classification presented in this paper to refine the analysis.

VIII. Related Work

The classical literature offers many categorizations related to testing, although these typically focus on the process of how the tests are derived rather than properties of the resulting tests (e.g., white-box vs. black-box testing, verification vs. validation). The most common classification of tests themselves is in terms of test levels, usually matching the different levels of the classical V-model, with the lowest level defined as unit or module tests. What exactly constitutes a unit, however, is usually only fuzzily defined: Myers [1] defines a unit test as targeting individual subprograms, subroutines, or procedures, whereas integration tests result from integrating multiple modules. Similarly, Beizer [13] loosely defines unit tests as targeting the smallest testable piece of software that can be controlled by a test driver, component tests as targeting integrated components, whereas integration tests consider combinations of components. Pezze and Young [14] refer to unit tests as targeting the "smallest possible work assignment", while Ammann and Offutt [15] interpret a unit as the one or more contiguous program statements with a name that other parts of the software use to call it. Along the same lines, Tarlinder [16] attempts to extrapolate a definition of unit tests, while still keeping the definition of a unit of work subjective

("a method, class, or cluster of classes that implement a single logical operation, which is accessible through a public interface"). The definition itself refers to other properties such as their automation, the fact that they do not just execute code but also check some property of the execution, they are fast, repeatable, and run in isolation. Besides these classifications in the literature, researchers tend to also provide implicit classifications in terms of the type of system targeted (e.g., web vs. desktop vs. mobile), or interface accessed by the tests (e.g., API vs. GUI. vs. command line). However, to the best of our knowledge there is no comprehensive classification of tests and their attributes.

Since the classical distinction between unit and integration tests has been an integral part of established IEEE [17] or ISTQB [18] standards for decades, more recently researchers have started questioning whether this classification is still valid. A prerequisite for investigating the classification is a means to label tests automatically, which is usually done using basic heuristics; for example, Trautsch and Grabowski [19] classify Python tests based on the number of module imports, assuming that integration tests need to import multiple modules and unit tests only a single one. Trautsch et al. distinguish between unit and integration tests based on whether a test covers classes from one or more Java packages. Orellana et al. [20] rely on the Maven build system in which unit tests and integration tests are executed by different test plugins. Kanstren et al. [21] calculate the number of methods that are covered by a test, assuming that unit tests will cover fewer methods than integration or system tests.

Given such a coarse classification, a large discrepancy between what developers classify as unit tests and the IEEE/ISTQB definitions of unit vs. integration tests has been reported by Trautsch and Grabowski [19] on Python tests. Trautsch et al. [22] also investigated whether unit tests and integration tests based on the IEEE/ISTQB definitions differ in terms of the faults they find on Java systems, measured using mutation analysis. Interestingly, they found that they do not, which counters the expectation that different types of tests should capture different types of bugs. Consequently, there is a mismatch between developer practice and the classical definitions, which our extensive taxonomy confirms.

IX. Conclusion

Software testing is generally accepted as an important quality assurance technique and is widely deployed across the industry. However, precise definitions of what software testing is and how different tests should be classified is lacking.

This paper proposes a novel classification framework for automated tests. The framework was developed using a sample of 1000 developer-written tests. The resulting classification is precise enough that it can be encoded in an automated analysis. This allows for automated, continuous classification of entire repositories. To demonstrate this, we implemented a proof of concept pipeline and ran it across all tests in Google's repository, fully classifying 38% of them.

This allowed us to make several novel observations on what types of tests developers write:

- Production-grade projects in a continuous integration and continuous deployment environment tend to have more configuration tests than code tests.
- Many tests are the result of developers finding test-execution frameworks and infrastructure convenient and using them for analyses that would not typically be considered "testing", with more than half of all tests never even executing the project under test.

These initial observations show a critical gap between the areas of testing that is currently being researched and what developers write in practice. Closing this gap and developing, for example, test adequacy techniques, for previously less researched tests would be a valuable direction to take.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] The Apache Software Foundation, "Apache Commons Project," https://commons.apache.org/, Last accessed Feb. 2025.

[3] Google Open Source, "Closure Compiler," https://github.com/google/closure-compiler, Last accessed Feb. 2025.

[4] H. Wright, T. D. Winters, and T. Manshreck, *Software Engineering at Google*. O'Reilly Media, 2020.

[5] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[6] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2018, pp. 181–190.

[7] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 955–963.

[8] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 10, pp. 3900–3912, 2021.

[9] M. Ivankovic, G. Petrovic, Y. Kulizhskaya, M. Lewko, L. Kalinovcic, R. Just, and G. Fraser, "Productive coverage: Improving the actionability of code coverage," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024, pp. 58–68.

[10] Google Open Source, "Bazel," https://bazel.build/, Last accessed: Feb. 2025.

[11] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2014, pp. 654–665.

[12] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921.

[13] B. Beizer, *Software testing techniques*. dreamtech Press, 2003.

[14] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[15] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[16] A. Tarlinder, *Developer testing: Building quality into software*. Addison-Wesley Professional, 2016.

[17] I. ISO, "Ieee, systems and software engineering–vocabulary," *IEEE computer society, Piscataway, NJ*, vol. 8, no. 9, 2010.

[18] I. Glossary, "International software testing qualification board," 2016.

[19] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 207–218.

[20] G. Orellana, G. Laghari, A. Murgia, and S. Demeyer, "On the differences between unit and integration testing in the travistorrent dataset," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 451–454.

[21] T. Kanstrén, "Towards a deeper understanding of test coverage," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 1, pp. 59–76, 2008.

[22] F. Trautsch, S. Herbold, and J. Grabowski, "Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects," *Journal of Systems and Software (JSS)*, vol. 159, p. 110421, 2020.