Automated Unit Test Improvement using Large Language Models at Meta

Nadia Alshahwan[*]
Meta Platforms
Menlo Park, USA

Jubin Chheda
Meta Platforms
Menlo Park, USA

Anastasia Finogenova
Meta Platforms
Menlo Park, USA

Beliz Gokkaya
Meta Platforms
Menlo Park, USA

Mark Harman
Meta Platforms
Menlo Park, USA
University College London
London, United Kingdom

Inna Harper
Meta Platforms
Menlo Park, USA

Alexandru Marginean
Meta Platforms
Menlo Park, USA

Shubho Sengupta
Meta Platforms
Menlo Park, USA

Eddy Wang
Meta Platforms
Menlo Park, USA

ABSTRACT

This paper describes Meta's TestGen-LLM tool, which uses LLMs to automatically improve existing human-written tests. TestGen-LLM verifies that its generated test classes successfully clear a set of filters that assure measurable improvement over the original test suite, thereby eliminating problems due to LLM hallucination. We describe the deployment of TestGen-LLM at Meta test-a-thons for the Instagram and Facebook platforms. In an evaluation on Reels and Stories products for Instagram, 75% of TestGen-LLM's test cases built correctly, 57% passed reliably, and 25% increased coverage. During Meta's Instagram and Facebook test-a-thons, it improved 11.5% of all classes to which it was applied, with 73% of its recommendations being accepted for production deployment by Meta software engineers. We believe this is the first report on industrial scale deployment of LLM-generated code backed by such assurances of code improvement.

CCS CONCEPTS

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

KEYWORDS

Unit Testing, Automated Test Generation, Large Language Models, LLMs, Genetic Improvement.

ACM Reference Format:
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 *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3663529.3663839

[*]Author order is alphabetical. The corresponding author is Mark Harman.

1 INTRODUCTION

As part of our overall mission to automate unit test generation for Android code, we have developed an automated test class improver, TestGen-LLM. TestGen-LLM uses two of Meta's[1] Large Language Models (LLMs) to extend existing, human-written, Kotlin test classes by generating additional test cases that cover previously missed corner cases, and that increase overall test coverage.

We reject tests that do not increase coverage, not because we wish to target increased coverage (although that is the useful byproduct). Rather, we use coverage as a proxy for the semantics of the generated tests. That is, we want a fully automated procedure that will discard any generated test that is semantically equivalent to some other existing test. Of course, equivalence is undecidable in general, so we cannot automatically determine whether one test is equivalent to another. Fortunately, test coverage provides a convenient safe and computable under approximation to non-equivalence.

TestGen-LLM is an example of Assured Offline LLM-Based Software Engineering (Assured Offline LLMSE) [6]. That is, unlike other LLM-based code and test generation techniques, TestGen–LLM uses Assured Offline LLMSE to embed the language models, as a service, in a larger software engineering workflow that ultimately recommends fully formed software improvements rather than smaller code snippets. These fully-formed code improvements are backed by verifiable guarantees for improvement and non-regression of existing behavior. A filtration process discards any test case that cannot be guaranteed to meet the assurances.

[1]The two LLMs used by TestGen-LLM were constructed at Meta for general purpose internal use, but they are not the focus of this paper, which is about an LLM-agnostic ensemble approach, its application to test class improvement at Meta and our experience with it at Meta's Test-a-thons. Because details are not relevant to this paper, we do not give details of the two LLMs, simply calling them 'LLM1' and 'LLM2' in this paper.

The filtration process can be used to evaluate the performance of a particular LLM, prompt strategy, or choice of hyper-parameters. For this reason, we include telemetry to log the behavior of every execution so that we can evaluate different choices. However, the same infrastructure can also be used as a kind of ensemble learning approach to find test class improvement recommendations. TestGen-LLM thus has two use cases:

(1) **Evaluation**: To evaluate the effects of different LLMs, prompting strategies, and hyper-parameters on the automatically measurable and verifiable improvements they make to existing code.
(2) **Deployment**: To fully automate human-independent test class improvement, using a collection of LLMs, prompting strategies, and hyper-parameters to automatically produce code improvement recommendations that are backed by
  (a) Detailed automatically-generated documentation that measures the improvement achieved by the new version of the test class;
  (b) Verifiable guarantees that the recommended test class does not regress any important properties of the existing version of the test class.

TestGen-LLM has been used in both of these modes. The evaluation mode was used as a prelude to deployment, allowing us to investigate and tune such choices of LLM, prompt strategy and temperature. It was also used after initial deployment to tune parameters for the subsequent, more widespread, release of the tool to engineers at Meta. The evaluation mode also allows us to report findings for evaluation (See Section 3.3).

Having arrived at sensible parameter choices, based on evaluation, TestGen-LLM was used at Meta to support engineers in various test improving activities, such as test-a-thons, in which a focused team of engineers target a particular aspect of one of Meta's products, in order to enhance existing testing.

Initial planning for TestGen-LLM took place in spring 2023 [5], with initial development in summer and autumn, and evaluation and onward optimization through winter 2023. This paper describes TestGen-LLM and reports our experience in developing and deploying it, through these test-a-thons for Instagram and Facebook. The primary contributions of the paper are:

(1) The introduction of the first example of Assured LLM-based Software Engineering (Assured LLMSE) [6]. In particular, we believe this is the first paper to report on LLM-generated code that has been developed independent of human intervention (other than final review sign off), and landed into large scale industrial production systems with guaranteed assurances for improvement over the existing code base.
(2) In an evaluation on Reels and Stories products for Instagram, 75% of TestGen-LLM test cases generated built correctly, 57% passed reliably, and 25% increased coverage.
(3) A report on the qualitative and quantitative results of development, deployment and evolution at Meta in 2023. When deployed to incrementally improve the coverage of production test classes on Instagram and Facebook more generally, TestGen-LLM was able to improve 10% of all classes to which it was applied and 73% of its test improvements were accepted by developers, and landed into production.
(4) A description of the lessons learned, open problems and research challenges raised by this application of Assured LLMSE to software test improvement.

2 THE TESTGEN-LLM SYSTEM

Like other automated test generation approaches [46], TestGen-LLM filters candidate solutions. It is these filters that provide the assurances that generated tests are fit for purpose. Figure 1 depicts the top level architecture of the TestGen-LLM system.

2.1 TestGen Filters

The first filter simply checks that the candidate code is fully buildable within the existing infrastructure of the app under test. Any code that does not build is immediately discarded and thereby removed from further consideration.

The second filter executes the generated tests, which build by definition, due to their clearing the first filter. Any test that does not pass may reveal a fault. However, it is more likely that the test simply contains an incorrect assertion. We want an entirely automated workflow. Without an automatable test oracle [7], TestGen-LLM cannot automatically determine whether a failing test has found a bug, or whether it merely contains an incorrect test assertion. Therefore, TestGen-LLM discards any test case that does not pass on first execution. The effect of this filter is to preserve only tests that can be used for regression testing [51]. Such tests, since they pass, protect the existing functionality of the system under test against future regressions.

A test that passes on first execution, may only coincidentally pass on the first occasion on which it is executed. More generally, a test that passes on some occasions and fails on others, when executed in an entirely identical environment, is called a 'flaky' test [28]. Flaky tests are one of the most significant problems for industrial software testing [19], and so we clearly do not want TestGen-LLM to introduce flakiness. TestGen-LLM thus uses the 'passes' filter to discard any flaky tests. The filter uses the simple, widely-used and effective approach of repeated execution [10, 28]; a test that does not pass on every one of five executions is deemed flaky. Since the generated test cases are unit level tests, repeated execution is a relatively computationally inexpensive approach to filtering out flaky tests.

A candidate test case that passes through the first two filters is guaranteed to provide reliable signal for regression testing. However, it may simply repeat the test behaviour of one of the existing tests. Such duplication of test effort would be a waste of resources. Furthermore, there will be no meaningful way in which TestGen-LLM could reliably *claim* that the original test class had been *improved* in some *measurable* way. Therefore, the final filter applied to non-flaky passing tests measures their coverage. Any test that does not improve coverage is also discarded.

The candidate test cases that pass through all three filters are thus guaranteed to improve the existing test class, and to provide reliable regression test signal. The pre- and post-processing are steps used to extract test cases and re-construct test classes.

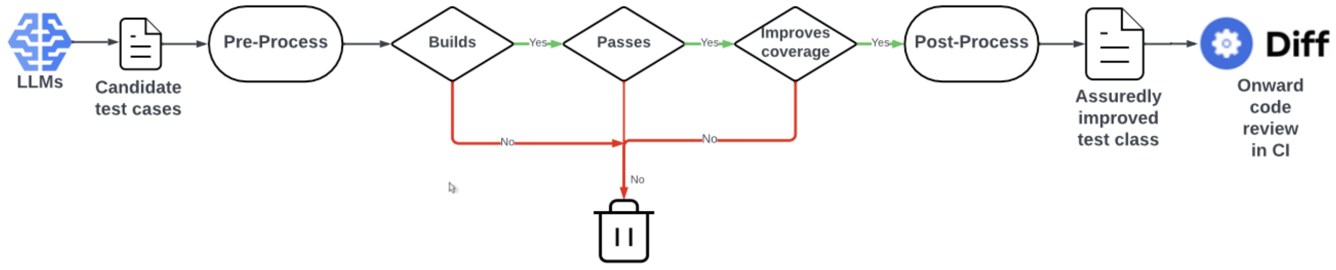

Figure 1: TestGen-LLM top level architecture (an instance of Assured Offline LLMSE [6]).

2.2 Advantages of Code Improvement Style LLM Code Generation

TestGen-LLM uses an approach called 'Assured LLM-based Software Engineering' (Assured LLMSE) [6]. In the remainder of this section, we set the principal advantages we have experienced from our development and deployment of TestGen-LLM, focusing on those we believe likely to carry over to other Assured LLMSE applications, and are not confined merely to test generation.

2.2.1 Measurable Improvement. At Meta, a code change submitted to the Continuous Integration (CI) system is called a 'diff' (short for differential). The diffs that TestGen-LLM generates include a precise measurement of the improvement they offer. For testing, the goal is to increase coverage, especially of corner cases that might have been missed. The diffs submitted by TestGen-LLM document their own coverage improvement to support the claim that they improve the code base.

2.2.2 Verifiable Guarantees of Non-regression. The diffs also include evidence of how they guard against regression. For test generation, TestGen-LLM simply augments an existing test class with additional test cases, retaining all existing test cases and thereby guaranteeing there will be no regression, by construction.

Assured LLMSE can also target operational characteristics, such as performance, for which it will necessarily rely on tests to guard against regression [6]. This was the motivation for our initial focus on test generation: once we have reliable automated regression testing at scale, we can use Assured LLMSE to target operational code properties, such as performance.

2.2.3 Ensemble Approach. Different LLMs have different strengths. Even the same LLM can produce multiple candidate solutions for a given prompt. As our results show, although some prompts, parameters and underlying LLM technologies perform better than others for a given test class, each combination tends to contribute *uniquely* to the overall number of test cases found (See Section 3.3). It is therefore highly advantageous to formulate the problem in such a way that it is amenable to an ensemble-style learning approach [11]. In such an ensemble approach, the best aspects of many LLMs, their prompts and parameters, can also be combined to give an overall improvement recommendation.

The LLM produces code components, not entire programs. Components are composable, and therefore can be provided by multiple different LLMs, working as an ensemble. For the test improvement instance of Assured LLMSE, each component is a test case. Test cases compose very naturally to form a test classes and test suites.

In the more general case of Assured LLMSE [6], LLM recommendations can be defined as code modifications as they typically are for automated repair [14, 15, 22, 31, 48]. Code modifications are also composable, and thereby support the overall ensemble approach to Assured LLMSE.

2.2.4 Helps Humans; Does Not Replace Them. By seeking to improve existing code and tests, rather than constructing them from scratch, TestGen-LLM works in concert with human engineering effort, insights and domain expertise; it does not seek to replace them. TestGen-LLM builds on the positive aspects of language models, while simultaneously sidestepping two of the most pressing concerns, raised in the literature and wider community:

- Whether language models will replace human coders (TestGen-LLM is a support *not a replacement* for humans).
- Whether language model results can be relied upon (TestGen-LLM results come with guarantees that ensure they *can* be relied upon).

More specifically, there has been much discussion on whether language models will replace human coders, with many arguments on all sides, and no shortage of opinions [12]. A debate also continues concerning whether LLMs are beneficial or harmful to education, for example, with arguments against [33], and broadly in favor [20] of their use as educational tools, or as ways to support existing human-led education efforts [30, 38, 42]. However, since TestGen-LLM is designed to be used as a support tool (to help engineers rather than to replace them), it is unnecessary for us to enter into this discussion in this paper. Rather, TestGen-LLM's goal is to provide a *recommender* system [1], that leaves the software engineer in ultimate control of the code that lands into the code base. This also ensures proper engineering oversight and accountability.

There has also been much discussion of the problems of relying on machine-generated code, especially the problem of hallucination [12, 29]. The TestGen-LLM approach in particular (and the overall Assured LLMSE approach more generally), overcomes hallucination by providing automated verifiable guarantees about the semantic properties of the code it recommends. These guarantees mean that the language model itself plays its own role in self accountability, providing at least as strong a semantic guarantee as many human engineers for the recommendations it makes.

2.2.5 Caters Well to LLMs with Limited Token Window Size. The specific formulation we used for test generation (as the problem of extending an existing test class), means that the test generation approach can be effective, even with a very small token window size. This is because the test class is typically much smaller than the class under test. TestGen-LLM can and does use prompts that provide both the test class and the class under test.

Providing the class under test does produce better results (as one would expect, since Retrieval Augmented Generation is known to perform well [12]). Nevertheless, we have also been able to successfully extend test classes by providing solely the existing test class, and omitting the class under test and any other details from the prompt. As our results show, prompts that use solely the test class (and not the class under test) as part of the prompt context, can still find additional tests. Moreover, such prompts found *unique* tests not found by other prompts (see results Section 3.3 for more details).

3 TESTGEN-LLM DEPLOYMENT

In a large company like Meta, it is typically not possible to simply switch on a new technology once developed, and apply it at scale. First, we must perform initial trials, and then cautiously deploy a Minimal Viable Product (MVP), in a well-controlled environment that allows us to gain experience, before migrating the technology to full deployment. If we simply deploy an MVP without first gaining this experience, then relatively small issues in the behavior of the technology can become considerably magnified at scale. The magnifying effect of small inefficiencies, errors or overlooked details, force multiply at this kind of scale. To give a sense of the scale, note that the central repository receives well over 100,000 commits per week [19], while the apps under test have client-side code bases of tens of millions of lines of code each, communicating with back-end server-side infrastructure of hundreds of millions of lines of code.

The deployment workflow thus follows a gradual incremental deployment plan, in which the MVP is gradually evolved and matured from proof of concept to deployed tool/infrastructure, over a series of increasingly larger-scale, and increasingly less tightly constrained trials. In the initial phases, the proto-MVP is applied at a very small scale, and in a tightly-controlled environment. In the later stages, after working in changes from the feedback on these initial deployments, the MVP becomes a more fully-fledged software engineering tool and is deployed freestanding (without detailed human oversight). In this section we describe the process by which we migrated from initial proof of concept to deployment.

3.1 Initial Trial

As an initial trial, we used an initial version of TestGen-LLM to create eight diffs and submitted these into Meta's standard continuous integration code review system.

The initial trial re-enforced the importance of giving individual guarantees per test case, and the value of the improvement assurances. The engineers who reviewed the initial diffs reported that the following two additional features would maximize the ease and speed with which they could review the recommendations from TestGen-LLM:

1. The importance of individual test level guarantees. In the initial MVP, we had only implemented class-level improvement guarantees. That is, the overall test class is improved, but there was no guarantee that each *individual* test case contributes to this improvement. For example, one of the diffs that ultimately landed initially contained four test cases. However, all four were only superficially different. By measuring the individual coverage contributed by each test case, the updated version of TestGen-LLM now automatically weeds out such duplicated test effort. This 'per test case' approach (rather than 'per test class') is slower to compute, but it gives TestGen-LLM the overall ability to more easily mix and match the results from different LLMs and different responses from the same LLM; the ensemble-style approach.

2. Useful to give more coverage details: The TestGen-LLM initial MVP reported only the files for which the improvement suggestion achieves extra coverage. Some of the engineers who reviewed the diffs asked whether the tool could provide full specific coverage information for each file, compared to the original. The TestGen-LLM MVP was therefore updated to report all details of the coverage achieved.

In this way we are able to gain valuable insights on the initial version of the tooling before deploying at wider scale. Based on the lessons learnt from this initial trail, we went on to develop a new version of TestGen-LLM, which was used as part of the next available test-a-thon exercise, in which engineers specifically sought to write new tests and extend existing test cases. We describe this next phase of deployment in the next section.

3.2 Instagram Test-a-thon November 2023

We used the the second version of the TestGen-LLM MVP to generate extra tests as part of the Instagram test-a-thon, which ran from November 15th - 20th 2023. Test-a-thons are regular human-centric activities in which engineers seek allocate specific focused time to writing test cases. In the November Instagram test-a-thon, TestGen-LLM was used in a carefully managed way in order to assess its suitability for wider and less closely-managed use.

Initial calibration: The engineer leading the test-a-thon first identified an initial component of interest for her team, so that she could confirm (or refute) that the diffs generated by the TestGen-LLM MVP would be suitable for consideration. TestGen-LLM produced three diffs for this component, each with a single test class extension for one of three different sub-components. In all three cases, the test class improvements were deemed acceptable. It was found that the verifiable claims for improvement made it easy to accept and land these tests into production. The engineer also greatly appreciated the way in which the generated tests' *coding style* closely mimicked that of the existing human–written test classes for each sub-component. Therefore, based on this initial trial, it was agreed to deploy TestGen-LLM as part of the test-a-thon.

Identifying the test classes to be targeted: During the three days of the test-a-thon, 36 engineers spent significant portions of their time focused on writing additional test cases for specific Instagram products targeted by the test-a-thon. The test classes and products chosen for the test-a-thon were those that had been the subject of recent intensive re-factoring activity.

Whenever a diff was submitted by an engineer as part of the test-a-thon process between 15th and 18th November 2023, TestGen-LLM was executed on the directory in which the test class resided, seeking to extend any of the test classes that reside in that directory. There is a generally close correspondence between directories and sub-components. In particular, all test classes in the directory are typically built using the same build rule and, therefore, TestGen-LLM thus automatically measures the additional coverage achieved over and above all the existing test classes residing in the same directory as the human-written test class.

Simulating a diff time deployment: There are two primary modes in which automated testing can interpose in continuous integration, which we typically call 'diff' time and 'post-land' time [2, 4]. At diff time, the tests are recommended to the engineer at the time they submit a related diff, whereas at post-land time, the test is recommended to the engineer at some arbitrary point after they have landed the relevant diff.

In previous work on deployment of automated testing technology at Meta, we have repeatedly found that diff time deployment is far more effective, because it maximizes relevance [19]. When a test is recommended at diff time, the engineer concerned already has the full context of the existing testing in place, and the code under test. As such, the engineer is in a much better position to quickly and correctly assess the recommended test. This increased relevance typically maximizes the impact of the test recommender system and the signals it provides. For this reason, we seek to prioritize diff time deployment, wherever possible.

Through the test-a-thon, we were able to gain experience of diff-time deployment mode, and thereby gain insights and experience on how this technology would play out when deployed in this mode. Although the potential reviewers had technically already landed *some* of the diffs containing test classes extended by TestGen-LLM, they were much more likely to have context, and to have only very *recently* landed test classes in the same directory. However, we cannot claim that this was a *perfect* example of diff time deployment: TestGen-LLM might also have benefited from the fact that this was a test-a-thon, and therefore, there was a greater-than-usual focus on testing and the context in which the tests were being deployed.

How the TestGen-LLM diffs were constructed for the November Instagram Test-a-thon: We constructed the diff summaries and test plans and submitted them manually, but used only information computed directly by TestGen-LLM. Where we included any text in the summary indicating our own interpretation of the results, this was contained in a blue-box "Note" to distinguish this text as human-generated and not machine-generated. In the second Instagram test-a-thon (see Section 3.4) we fully automated the construction of the content and claims made by TestGen-LLM in diff comments and test plans.

3.2.1 Outcomes from the First Instagram Test-a-thon. During the first test-a-thon, 36 engineers landed 105 unit test diffs, of which 16 were generated by TestGen-LLM. In total, TestGen-LLM created 17 diffs. One diff was abandoned because the test case did not include any assertion. 16 landed into production. The test case in the rejected diff attempted to test a partially implemented function, for which the LLM code left a comment indicating that an assertion was to be added as a "TODO". The test case did extend coverage (simply by executing the previously unexecuted method-under-test), but it was rejected by the engineer reviewing the diff because it failed to contain an assertion.

The largest coverage improvement was achieved by a TestGen-LLM diff that covered a method not previously covered, and thus generated a lot of additional coverage, including:

- 28 new files covered that were not previously covered.
- 13 files which were previously partially covered, but for which the improved test class extended coverage
- 3 A/B testing gate keepers (for which generated A/B decision making code was additionally covered).

The smallest coverage improvement was produced by a diff that covered a single additional line (an early `return`) in a file that was already partially covered by the existing test classes.

Anonymized results for top 10 performers among the 36 engineers at the the test-a-thon are shown in Table 1. As can be seen, TestGen-LLM landed in sixth place in rank order by number of tests generated. The table also reveals the way in which TestGen-LLM behaves differently to test engineers. Whereas test engineers will tend to write a whole class of tests in a single diff, TestGen-LLM submits each test as a separate diff.

This is because TestGen-LLM is extending existing test classes, with additional test cases. It also allows the engineers to more easily accept or reject the recommendations, per test case. As a result, TestGen-LLM's number of test cases per diff is always one; a fact which makes it superficially appear to be more productive in terms of diffs. However, since the ranking was computed in terms of the number of test cases landed, the rank position of TestGen-LLM is a fair reflection of its productivity compared to human engineers during the test-a-thon.

Another apparently anomalous result from the table is the number of lines covered by the 17 test cases generated by TestGen-LLM. At first glance, it might appear that TestGen-LLM is able to cover a great deal more than the human engineers. However, this result arose due to a single test case, which achieved 1,326 lines covered. This test case managed to 'hit the jackpot' in terms of unit test coverage for a single test case. Because TestGen-LLM is typically adding to the existing coverage, and seeking to cover corner cases, the typical expected number of lines of code covered per test case is much lower. For example, 6 of the 17 test cases it generated covered only a single extra line. However, in all 17 cases, we manually verified that the generated test did, indeed, cover at least one additional valid corner case, such as an early `return` and/or special processing for special values such as `null` and empty list. The median number of lines of code added by a TestGen-LLM test in the test-a-thon was 2.5. This is a more realistic assessment of the expected additional line coverage from a single test generated by TestGen-LLM.

3.3 Evaluation of LLMs and Prompts

The outcome of the first November Instagram test-a-thon gave us confidence that we had a usable tool in TestGen-LLM. In particular, the manual verification that the tests added valid corner cases, the fact that the test style was appreciated by engineers, and the overall performance relative to human effort (see Table 1), provided the evidence that it was worth developing the tool further, and deploying it more widely.

However, before deploying more widely, we needed to choose suitable defaults for hyper parameters so that engineers could use

Table 1: Results from the First Instagram Test-a-thon, conducted in November 2023. Human test authors are named by the product component on which they worked. The TestGen-LLM tool landed in sixth place overall, demonstrating its human-competitive added value.

rank	test author	No, of tests	lines covered	diffs
1.	Threads Engineer	40	1,047	8
2.	Home Engineer	34	650	6
3.	Business Engineer	34	443	3
4.	Sharing Engineer	33	816	8
5.	Messaging Engineer	18	157	2
6.	**TestGen-LLM**	17	1,460	17
7.	Friends Engineer	12	143	2
8.	Home Engineer	10	273	2
9.	Creators Engineer	10	198	3
10.	Friends Engineer	10	196	5

the tool out-of-the-box without having to consider choices of settings. To tackle this need we switched TestGen-LLM from deployment mode to evaluation mode.

We undertook experiments to determine the the most favorable parameters among different temperatures, language models available, and prompts. We conducted experiments on two products for Instagram, Reels and Stories, to determine the differential performance of the two different language models (LLM1 and LLM2), and also to investigate the unique contribution of each of four prompting strategies. This produced results over 86 Kotlin components with existing human-written test classes (31 for Stories and 55 for Reels) as follows:

(1) 75% of test classes had at least one new test case that builds correctly.
(2) 57% of test classes had at least one test case that builds correctly and passes reliably.
(3) 25% of test classes had at least one test case that builds correctly, passes and increases line coverage compared to all other test classes that share the same build target.

These results are depicted[2] in the Sankey diagram [39] in Figure 2. It was particularly striking that, although 57% of test classes have a test that builds correctly and passes reliably, only 25% of classes had a test that builds reliably, passes non-flakily and adds additional line coverage[3].

The prompts used are set out in Table 2. We wanted to experiment with a variety of different prompting strategies. The prompt `extend_coverage` is the canonical example, which gives maximal information and clear direction to the language model. `corner_cases` was included specifically to focus on corner cases, while `extend_test` was included to investigate the potential to find solutions when only the test class is provided (and not the class under test). Finally, `statement_to_complete` was included

[2]This image was created from the data using the freely-available tool SankeyMatic (`https://sankeymatic.com/build/`).
[3]Unfortunately, although Jacoco is theoretically capable of collecting branch coverage, this is not available at the scale of testing required by Meta, so we currently rely solely on line coverage.

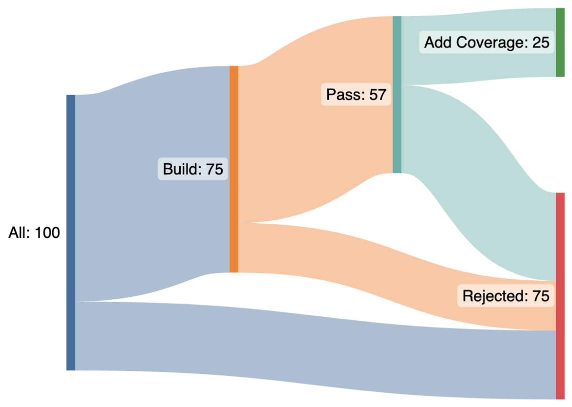

Figure 2: Sankey diagram showing the filtration process outcomes (as percentages of all test cases) from the Experimental Study on Instagram components for Reels and Stories products, using the four prompt strategies from Table 2 and the two language models, LLM1 and LLM2.

to investigate the alternative prompting style of making a statement that should be completed by the language model. This is inspired by the fact that language models are, inherently, predictive models for text completion. As such, it seems reasonable that such a prompt ought to have an advantage.

Using these four prompts and the two language models available, we obtained the following results over the 86 test classes: Using just LLM1, TestGen-LLM was able to find 13 tests (15% of files have at least one new test). Three of the four prompts added unique value to the overall search: `extend_coverage` added 2 unique tests, while `corner_cases` and `extend_test` each added one unique test. Although it did succeed in finding test cases, the prompt `statement_to_complete` failed to add any *unique* test cases, and therefore made no unique individual contribution when using the LLM.

Using just LLM2, TestGen-LLM was able to find 16 tests (19% of files have at least one new test). All four prompts added unique value to the overall search: `extend_coverage` added 5 unique tests, while `statement_to_complete` and `extend_test` each added 2 unique tests. `corner_cases` added 1 unique test.

We also found that generated test cases were most likely to build, pass and extend coverage, with LLM temperature zero. However, the effect size was low compared to the nearest next-best temperatures of 0.2 and 0.5; all had approximately 4% success over all test cases suggested[4]. We therefore set the default temperature to zero.

Based on these results, it was decided to deploy TestGen-LLM for subsequent test-a-thons with default temperature zero, default LLM set to LLM2 and default prompt `extend_coverage`. However, users were free to deploy using other settings when applying the TestGen-LLM tool. In particular, for subsequent Instagram and Facebook test-a-thons, the tool offered the option to perform a temperature

[4]For a given attempt to extend a test class, there can be many attempts to generate a test case, so the success rate per test case is typically considerably lower than that per test class.

sweep over all temperature settings (from 0.0 to 1.0 in steps of 0.1), the option to select a particular prompt and the option to select a particular LLM. The tool also offered an 'LLM ensemble' that combines results from both then-available LLMs.

3.4 Deployment in TestGen-LLM-only Instagram Test-a-thon December 2023

During the period 18th to the 20th of December 2023, the (now further improved) TestGen-LLM tool was run on the same directories that were updated in the November test-a-thon. We chose these because they have been the target of recent human development so they presented a challenge: to improve further on the combination of recent human and previous TestGen-LLM test effort.

Although there was no human intervention in the generation of the diffs by TestGen-LLM, the recommendations were first considered by two of authors before being passed on to engineers. Engineers were also pre-warned that they may be receiving test cases generated by TestGen-LLM, although there was no other pre-training involved other than providing the context that the diffs they would be receiving were generated by an MVP TestGen-LLM deployment.

This deployment was entirely automated, with TestGen-LLM now running automatically on these directories without human intervention. TestGen-LLM automatically generated 42 diffs that were submitted for review. Of the 42 diffs:

- 36 were accepted by the engineer reviewing them.
- 4 were rejected and/or abandoned.
- 2 were withdrawn.

The 2 withdrawn diffs each added coverage, but we recognized that the files covered were unimportant. The reasons for the four rejected diffs were:

(1) Generating tests for trivial methods under test (a getter method).
(2) Failing to follow the 'single responsibility per test case' principle (2 rejected for this reason).
(3) Failing to include an assertion in the test case.

3.4.1 Deployment at the Facebook App Test-a-thon December 2023. In this deployment, we had sufficient confidence to automatically submit recommendations from TestGen-LLM to engineers. There was no engineer pre-training process, no specific test-a-thon expectations, and no additional context provided to the engineers. This gave us a realistic assessment of the engineers' response to LLM-generated test recommendations provided 'out of the box'. Overall, over 50% of the diffs submitted were accepted by developers, a figure which rises to almost 70% of those which received a review by developers. Specifically, of the 280 diffs generated by TestGen-LLM:

- 144 were accepted by the engineer reviewing them.
- 64 were rejected and/or abandoned.
- 61 did not receive a review.
- 11 were withdrawn.

4 QUANTITATIVE RESULTS FROM DEPLOYMENT

This section reports the overall results from the fully freestanding deployment at the November and December Instagram and Facebook app test-a-thons. All deployment data collection took place between 29th Oct 2023 - 29th Dec 2023, during which period TestGen–LLM was deployed (and redeployed) through three different test-a-thons, with evolution an improvement in between each iteration.

In total, over the three test-a-thons, 196 test classes were successfully improved, while the TestGen-LLM tool was applied to a total of 1,979 test classes. TestGen-LLM was therefore able to automatically improve approximately 10% of the test classes to which it was applied. Over all, 73% of TestGen-LLM test improvements were accepted by developers. This is an encouraging result for an automated code improvement recommender system. For example, it compares favorably with previous attempts to deploy automated repair at Meta (where 50% of recommended fix improvements were accepted and landed into production [31]).

Table 3 shows the overall success rate, per test case generation trial, over the two platforms: Facebook and Instagram. In this table, a trial is an attempt to generate a new test case. A trial is only considered successful if the generated test case builds, passes, and increases coverage over all existing test cases. As can be seen, the success rate is similar for both platforms, although slightly higher for Facebook than for Instagram. We believe that this difference may be due to the difference in available training data. There is approximately one order of magnitude more human-written Kotlin test code for Facebook then there is for Instagram.

Table 4 shows the performance over both language models and all four prompts for different temperature settings. A 'successful' trial is one in which a test case is generated that passes all filters (successfully builds, passes reliably, and adds additional coverage over all existing tests, including previous LLM-generated tests). Since the default temperature setting was 0.0, this received by far the greatest number of trials (30,483) during the test-a-thons. However, as can be seen, other temperature settings were used in the deployment, since engineers applying the tool had the ability to select different temperatures.

It is interesting to note that good results were obtained for a temperature of 0.4. It might be tempting to speculate that these results should occasion a change in the default temperature setting. However, care is required in interpreting such results. In particular, the results are based on very different sample sizes. Furthermore, due to these results being obtained from *deployment* mode not *experimental* mode, there are unavoidable confounding factors.

The primary comfounding factor is that tests are generated and deployed *incrementally* in production (as stacks of diffs), accumulating coverage as they are deployed. Therefore, it becomes incrementally harder to increase coverage over additional production trails. Experimental mode is a kind of 'dry run' in which no tests are actually added and therefore additional coverage is always measured with respect to the same baseline.

Coverage growth is inherently logarithmic (over the number of test cases generated). Therefore, as more coverage is achieved, it becomes incrementally harder to achieve further improvements

Table 2: The four primary prompts used in the deployment for the December 2023 Instagram and Facebook app test-a-thons

Prompt name	Prompt Template
extend_test	Here is a Kotlin unit test class: {existing_test_class}. Write an extended version of the test class that includes additional tests to cover some extra corner cases.
extend_coverage	Here is a Kotlin unit test class and the class that it tests: {existing_test_class} {class_under_test}. Write an extended version of the test class that includes additional unit tests that will increase the test coverage of the class under test.
corner_cases	Here is a Kotlin unit test class and the class that it tests: {existing_test_class} {class_under_test}. Write an extended version of the test class that includes additional unit tests that will cover corner cases missed by the original and will increase the test coverage of the class under test.
statement_to_complete	Here is a Kotlin class under test {class_under_test} This class under test can be tested with this Kotlin unit test class {existing_test_class}. Here is an extended version of the unit test class that includes additional unit test cases that will cover methods, edge cases, corner cases, and other features of the class under test that were missed by the original unit test class:

Table 3: Results for the two different platforms. The technology was initially developed by the Instagram Product Performance Organisation within Meta, hence its greater number of overall trials. The highest success rate for the Facebook platform may arise from the fact that there are 10x more examples of human–written test cases for Facebook compared to Instagram.

Platform	Successful trials	Total trials	Success rate
Facebook	490	8,996	0.05
Instagram	831	23,535	0.04

Table 4: Results for different temperature settings. After initial experimentation, zero was chosen as the default, hence its far greater number of overall trials.

Temperature	Successful trials	Total trials	Success rate
0.9	50	1,580	0.03
0.8	18	703	0.03
0.7	12	552	0.02
0.6	7	536	0.01
0.5	4	500	0.01
0.4	16	334	0.05
0.3	5	324	0.02
0.2	3	505	0.01
0.1	4	552	0.01
0.0	1,215	30,483	0.04

from the diminishing number of remaining coverage improvement opportunities. Therefore, any configuration setting (such as temperature) that receives a larger share of overall production deployment trials is at a disadvantage compared to those that receive a smaller share.

Finally, we report the results for the two different language models used in the test-a-thons, overall (Table 5) and per platform (Table 6). Since LLM2 was the default model, it received a far greater number of trials and therefore care is required and interpreting the slight differences in performance between the models. These results do, nevertheless, provide further confirmation of the finding that performance overall is slightly better for the Facebook platform, which enjoys a larger number of existing Kotlin human-written tests, compared to Instagram.

Table 5: Results for the two different LLMs. After initial execution LLM2 was chosen as the default, hence its greater number of overall trials.

LLM	Successful trials	Total trials	Success rate
LLM1	163	3,173	0.05
LLM2	1,157	28,654	0.04

Table 6: Results for the two different platforms and LLMs.

Platform	LLM used	Successful trials	Total trials	Success rate
Facebook	LLM1	47	719	0.07
Instagram	LLM1	116	2,454	0.05
Facebook	LLM2	443	8,146	0.05
Instagram	LLM2	714	20,508	0.03

5 QUALITATIVE OBSERVATIONS FROM DEPLOYMENT

In this section we present observations and lessons learned from deployment of a more qualitative nature. These will form the most immediate future work in the technical development of current TestGen-LLM deployment, while Section 7 presents higher-level directions for future work and open research problems on which we would be interested in collaborating with and/or learning from the wider research community.

Source code analysis and manipulation have always been, and will likely always remain important in Software Engineering [16]. Previous work has successfully used hybrids of static analysis and language models, in which both applications of static analysis and applications of language models benefited [3, 24, 27, 36]. Many of our observations further underscore the potential of static analysis, and highlight the opportunities for combining static analysis with language model inference. In particular, we envisage the following four avenues for static analyses to improve LLM inference:

1. LLM 'self-plagiarism': Since an LLM is, at heart, a probabilistic inference engine, it can be expected that it may produce the same (or similar) responses for the same prompt over multiple samples. We observed that both LLM1 and LLM2 often generated almost identical tests for the same prompt; different in name only. This may also be a byproduct of the default temperature setting (zero), which is the most deterministic.

Anecdotally, it seemed that tests generated were either almost verbatim copies of others previously generated, or very different. They were never just 'somewhat different' on a 'sliding scale' of syntactic and semantic difference; the similarity between generated tests was 'all or nothing'.

Perhaps the nature of the conditional probability sampling makes this behavior highly likely, and thus expected. In later, more mature, versions of TestGen-LLM we doubled down on this observation and included an extra filter to remove previously seen test cases. The observation that similarity between generated tests was 'all or nothing' greatly simplified this filter, because it simply needed to check for syntactic equality of test bodies. Analyses that detect semantically similar code, such as Type 2 and Type 3 clones [45] may also be helpful here.

2. Nuanced coverage reporting: It can happen that TestGen-LLM obtains valuable additional coverage, but not *necessarily* solely for the class under test. It would be easy to automate the process of identifying this case. Sometimes this could be very valuable, because it may test parts of the code that are hard to reach in other ways. Alternatively, where the class under test has little coverage and most of the coverage improvement concerns other units, it may be a sign of inadequate mocking.

TestGen-LLM includes a filter for test flakiness. This tends to reduce any concerns about inadequate mocking. Nevertheless, further automated static analysis and post processing can be applied, based on a more nuanced analysis of the coverage achieved. As the most immediate next step, we plan to flag the issue to code reviewers, so that they are more aware of situations in which generated tests may be playing the role more of integration tests.

3. Highlighting test need: Sometimes the generated tests included a "TODO" (e.g., to write the assertion; see, for example, Section 3.2.1). We did not land these tests into production. However, in some cases, they indicated considerable potential coverage wins. In one case, a series of such tests were generated for the class under test, each of which covered a non-trivial function in the class under test that was not previously covered.

Although these test cases did not include any assertions, they could nevertheless add value simply by covering these functions and using the so-called 'implicit oracle' [7] (that the code should not raise an exception). Furthermore, such cases might be useful as hints to human test writers, and may also create the initial template for such a follow-up human-written test. After all, the task 'add a suitable assertion to this generated test' involves considerably less human effort than the task 'write a brand new test case from scratch'.

4. Re-prompting: Sometimes, newly-generated tests covered a subset of the lines of the method under test, that had not previously been covered at all. This could be a situation where TestGen-LLM should automatically recognize and re-prompt the LLM to try and achieve further coverage for this method. If it can: fine. If not, TestGen-LLM should flag this situation to the engineer. The human engineer will likely more easily fill in the gaps, now they have a starting test template to work from.

6 RELATED WORK

Software test generation is one of the most widely-studied topics within the more general area of what might be termed 'Large Language Model-based Software Engineering' (LLMSE) [12]. Wang et al. [49] presented a literature review of 102 papers on testing, debugging and repair, while Fan et al. [12] present a general survey, across all software engineering applications, including software testing.

Although both surveys confirm the prevalence of LLM-based test generation in the literature, no previous paper has tackled the problem of extending existing test classes, nor reporting results on the provision of measurable assurances for both the improvement and the absence of regressions. The primary technical novelty of the present paper is to introduce this test extension application, as a specific example of Assured LLMSE [6], while the main contribution is the experience report describing its development and deployment at Meta, where it has been applied to Facebook and Instagram. We believe this is the first instance of industrial deployment of Assured LLMSE.

Results vary quite widely in the literature for coverage achieved by LLM-based test generation. For example, Siddiq et al. [44] reported that by generating tests from scratch (rather than seeking to improve on existing tests) it is possible to achieve 80% coverage on the small examples in the 'HumanEval' data set using CodeX [32].

However, they also report that generation from scratch achieved no more than 2% coverage on the EvoSuite SF110 data set [13]. Naturally, one might expect that the coverage achievable would depend partly on the size of the system, since larger systems have more scope for complex interactions, and deeply nested code that is harder to cover. Schafer et al. [43] also report high statement level coverage (70%), but also for relatively smaller systems (25 packages, ranging in size from 25 lines of code to 3,100 lines of code). This indicates the importance of studying and reporting experience from deployment on large complex industrial software systems, as a complement to results on such smaller systems, benchmarks and open source software.

Notwithstanding the high degree of variability for coverage reported in the present literature, our empirical findings are broadly consistent with previous results reported on larger open source systems. We found that approximately approximately 57% of Kotlin test cases generated are executable (with 25% improving coverage). This compares relatively favorably with recently reported results. For example Nie et al. [37] report 29% of tests generated using TeCo are executable, while Yuan et al. [52] report approximately one third Chat-GPT-generated tests are executable with suitable prompt engineering. However, the language model technologies used, the training set, fine tuning another characteristics play a crucial role [12].

TestGen-LLM embeds the language model within a wider software engineering process that filters candidates, in order to provide the assurances that guard against hallucination, and otherwise sub-optimal results, that might accrue from the unfettered application of language models without such filtering. In this regard, TestGen-LLM is a hybrid approach combining traditional software engineering and software testing with language models as an engine for code generation. Previous authors have also considered hybrid forms of LLM applications in testing, for example, hybridizing with Search Based Software Testing (SBST) [26], Mutation Testing [35] and Fuzzing [21].

A wide variety of LLMs have been used in previous work on problems relating to software testing, including BART, CodeBert, ChatGPT, CodeX, CodeT5, and T5 [12, 49]. Meta released public versions of LLaMA and CodeLlama, including source code, in February and August 2023 respectively. Llama is a general purpose LLM with a variety of model sizes ranging from 7 billion to 65 billion parameters [47]. CodeLlama is a model more specifically trained on software and has model sizes 7B, 13B, and 34B (and, as of 29th Jan 2024, 70Bn) [41]. These two freely available LLMs, LLaMA and CodeLlama, have also been used in previous work on testing topics by other researchers [34, 50]. Although the results presented here are based on TestGen-LLM using two internal LLMs built by Meta, the Assured LLMSE design is LLM-agnostic and allows for an arbitrary number of LLMs to each contribute test cases to the extended test class.

TestGen-LLM's approach to test improvement draws inspiration from previous research on Genetic Improvement[40] and automated repair [15]. Genetic Improvement treats existing software code as 'genetic material' to be mutated and recombined to improve existing code according to measurable improvement criteria. Many approaches to automated repair adopt a generate-and-test approach, in which multiple candidate solutions are generated using a cheap generation technology, and subsequently filtered and discarded according to evaluation criteria. Like automated repair, TestGen-LLM uses a generate-and-test approach; filtering out those candidates that do not meet well-defined semantic criteria, such as passing reliably. Like genetic improvement, TestGen-LLM treats code as genetic material to be mixed and matched.

However, unlike Genetic Improvement, TestGen-LLM uses an 'ensemble' of language models and configurations, rather than genetic programming. By contrast, in its original formulation [25], Genetic Improvement envisaged Genetic Programming as the core technology for creating candidate code variations. Much of the work on automated program repair has also used similar computational search techniques [15]. However, the advent of LLMs provides us with an additional route to achieve the same goal. Essentially, our approach can be thought of as a form of Search Based Software Engineering (SBSE)[17, 18], in which the search is over a set of candidate test class improvements, which are evaluated using a generate-and-test search process, and for which the core technology for generation is based on language models.

7 FUTURE WORK AND OPEN PROBLEMS

There are many avenues for future work and open problems concerning automated test improvement using Assured LLMSE. In this section we outline three such open problems.

1. Assessing improvement: The measurement of improvement is clearly a key factor in any Assured LLMSE [6]. We have taken the simple approach of measuring line coverage as a proxy for improvement, but it is merely an expedient proxy for 'improvement'. However, line coverage is a stringent requirement for success. Were we to relax the requirement to require only that test cases build and pass, then the success rate rises to 57%. Future work will therefore consider other test improvement criteria. Mutation coverage [23] would likely be the best performing criterion. This is because strong mutation coverage has been empirically demonstrated to outperform other forms of coverage [9]. However, it is challenging to deploy such computationally demanding techniques at the scale we would require [8].
2. Application–aware probability distribution resolution : More research is required to define application-specific techniques for transforming the LLM probability distribution into code [6].

3. LLMs are dedicated followers of fashion: LLMs are 'fashion followers' that mimic existing test writing styles, adopting (and often very faithfully replicating) the mode of expression prevalent in the test class itself and, more generally, in the code-base on which they have been trained. This is a natural consequence of the probabilistic nature of the language model. Often this 'fashion following' is a very desirable characteristic. Our engineers strongly appreciated the way TestGen-LLM followed different assertion styles (standard JUnit style and also bespoke assertion styles, written by engineers as utilities specifically for the component under test). In particular, where there was a bespoke style, TestGen-LLM tests use the corresponding utilities. Furthermore, the generated tests also followed existing naming conventions, commenting styles, and overall test structure. It would be highly challenging to define algorithms for replicating these bespoke styles using a more rule-based approach, but using an LLM, this useful behaviour simply comes naturally. Nevertheless, fashion following also means that the language model can pick up deprecated coding habits, so fixing this is a topic for future work.

8 CONCLUSIONS

This paper introduced TestGen-LLM which has been used to land test cases in production at Meta. The paper described the evolution of TestGen-LLM from proof of concept, through minimal viable product, to deployed test support tool. The primary TestGen-LLM characteristic of interest is the way in which TestGen-LLM guards against LLM hallucination: it submits, for human review, only test cases that it can guarantee improve on the existing code base. We believe this is the first report of Assured Large Language Model Software Engineering deployed at scale in industry.

ACKNOWLEDGMENTS

We wish to thank the many AI and developer infrastructure teams at Meta for their work on the LLMs, build, test and continuous integration systems, without which TestGen-LLM would not be possible. We also want to thank the Instagram and Facebook platform organizations, and leadership for their support, and the many Meta engineers who reviewed the code produced by TestGen-LLM, providing valuable feedback and insights on its development, deployment and evolution.

REFERENCES

[1] Adomavicius and Tuzhilin. 2005. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering* 17 (2005).

[2] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Virtual.

[3] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2023. Improving Few-Shot Prompts with Relevant Static Analysis Products. arXiv:2304.06815.

[4] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook (keynote paper). In 10^{th} *International Symposium on Search Based Software Engineering (SSBSE 2018)*. Montpellier, France, 3–45. Springer LNCS 11036.

[5] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST 2023)*. IEEE, 1–10.

[6] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering (keynote paper). In $2^{nd.}$ *ICSE workshop on Interoperability and Robustness of Neural Software Engineering (InteNSE)* (Lisbon, Portugal). To appear.

[7] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525.

[8] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.

[9] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 597–608.

[10] Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 982–994. https://doi.org/10.1145/3510003.3510194

[11] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. 2020. A survey on ensemble learning. *Frontiers of Computer Science* 14 (2020), 241–258.

[12] Angela Fan, Beliz Gokkaya, Mitya Lyubarskiy, Mark Harman, Shubho Sengupta, Shin Yoo, and Jie Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *ICSE Future of Software Engineering (FoSE 2023)*. To Appear.

[13] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[14] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (2013), 421–443.

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[16] Mark Harman. 2010. Why Source Code Analysis and Manipulation Will Always Be Important (Keynote Paper). In 10^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*. Timisoara, Romania.

[17] Mark Harman and Bryan F. Jones. 2001. Search Based Software Engineering. *Information and Software Technology* 43, 14 (Dec. 2001), 833–839.

[18] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. 2012. Search Based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1 (November 2012), 11:1–11:61.

[19] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis (keynote paper). In 18^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*. Madrid, Spain, 1–23.

[20] Will Douglas Heaven. 2023. ChatGPT is going to change education, not destroy it. *MIT Technology review* (April 2023).

[21] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting Greybox Fuzzing with Generative AI. arXiv:2306.06782.

[22] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. arXiv:2303.18184 [cs.SE]

[23] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September–October 2011), 649 – 678.

[24] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with LLMs. *arXiv preprint arXiv:2303.07263* (2023).

[25] William B. Langdon and Mark Harman. 2015. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation (TEVC)* 19, 1 (Feb 2015), 118–135.

[26] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. (2023).

[27] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting static analysis with large language models: A ChatGPT experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2107–2111.

[28] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In 22^{nd} *International Symposium on Foundations of Software Engineering (FSE 2014)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne Storey (Eds.). ACM, Hong Kong, China, 643–653.

[29] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. arXiv:2305.12138.

[30] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM, Toronto ON Canada, 931–937. https://doi.org/10.1145/3545945.3569785

[31] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Montreal, Canada.

[32] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

[33] Laura Meckler and Pranshu Verma. 2022. Teachers are on alert for inevitable cheating after release of ChatGPT. *The Washington post* (December 2022).

[34] Seungjun Moon, Yongho Song, Hyungjoo Chae, Dongjin Kang, Taeyoon Kwon, Kai Tzu-iunn Ong, Seung-won Hwang, and Jinyoung Yeo. 2023. Coffee: Boost Your Code LLMs by Fixing Bugs with Feedback. *arXiv preprint arXiv:2311.07215* (2023).

[35] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2023. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. *arXiv e-prints* (2023), arXiv–2308.

[36] Ruba Mutasim, Gabriel Synnaeve, David Pichardie, and Baptiste Rozière. 2023. Leveraging Static Analysis for Bug Repair. arXiv:2304.10379 [cs.SE]

[37] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. arXiv:2302.10166.

[38] David Noever and Kevin Williams. 2023. Chatbots As Fluent Polyglots: Revisiting Breakthrough Code Snippets. arXiv:2301.03373.

[39] Ethan Otto, Eva Culakova, Sixu Meng, Zhihong Zhang, Huiwen Xu, Supriya Mohile, and Marie A Flannery. 2022. Overview of Sankey flow diagrams: focusing on symptom trajectories in older adults with advanced cancer. *Journal of geriatric oncology* 13, 5 (2022), 742–746.

[40] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. https://doi.org/doi:10.1109/TEVC.2017.2693219

[41] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

[42] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V.1*. ACM, Lugano and Virtual Event Switzerland, 27–43. https://doi.org/10.1145/3501385.3543957

[43] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. arXiv:2302.06527.

[44] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. arXiv:2305.00418.

[45] Jeffrey Svajlenko and Chanchal K Roy. 2020. A Survey on the Evaluation of Clone Detection Performance and Benchmarking. *arXiv preprint arXiv:2006.15682* (2020).

[46] Deepika Tiwari, Long Zhang, Martin Monperrus, and Benoit Baudry. 2022. Production Monitoring to Improve Test Suites. *IEEE Transactions on Reliability* 71, 3 (2022), 1381–1397. https://doi.org/10.1109/TR.2021.3101318

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

[48] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to Design a Program Repair Bot? Insights from the Repairnator Project. In *40th International Conference on Software Engineering, Software Engineering in Practice track (ICSE 2018 SEIP track)*. 1–10.

[49] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. arXiv:2307.07221.

[50] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. *arXiv preprint arXiv:2308.04748* (2023).

[51] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[52] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207.

Received 2024-02-08; accepted 2024-04-18