Mutation-Guided LLM-based Test Generation at Meta

Christopher Foster
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Abhishek Gulati
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Mark Harman
Product Compliance and Privacy team, Meta Platforms
London, UK

Inna Harper
Developer Infrastructure team, Meta Platforms
London, UK

Ke Mao
WhatsApp team, Meta Platforms
London, UK

Jillian Ritchey
Messenger team, Meta Platforms
New York, USA

Hervé Robert
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Shubho Sengupta
FAIR, Meta Platforms
Menlo Park, USA

Abstract

This paper[1] describes Meta's Automated Compliance Hardening (ACH) system for mutation-guided LLM-based test generation. ACH generates relatively few mutants (aka simulated faults), compared to traditional mutation testing. Instead, it focuses on generating currently undetected faults that are specific to an issue of concern. From these currently uncaught faults, ACH generates tests that can catch them, thereby 'killing' the mutants and consequently hardening the platform against regressions. We use privacy concerns to illustrate our approach, but ACH can harden code against *any* type of regression. In total, ACH was applied to 10,795 Android Kotlin classes in 7 software platforms deployed by Meta, from which it generated 9,095 mutants and 571 privacy-hardening test cases. ACH also deploys an LLM-based equivalent mutant detection agent that achieves a precision of 0.79 and a recall of 0.47 (rising to 0.95 and 0.96 with simple pre-processing). ACH was used in Messenger and WhatsApp test-a-thons where engineers accepted 73% of its tests, judging 36% to privacy relevant. We conclude that ACH hardens code against specific concerns and that, even when its tests do not directly tackle the specific concern, engineers find them useful for their other benefits.

CCS Concepts

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

Keywords

Unit Testing, Automated Test Generation, LLMs.

[1]Author order is alphabetical. The corresponding author is Mark Harman, who is also a part time professor at University College London (UCL). Inna Harper and Shubho Sengupta were with Meta when this work was conducted, but have since changed affiliation.

ACM Reference Format:
Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3728544

1 Introduction

We report on Meta's deployment of ACH[2], an agentic LLM-based tool for generating tests to target specific classes of faults. The paper focuses on automated privacy hardening: the problem of automatically generating unit tests to reduce the risk of future regressions with respect to privacy issues. However, the approach can be applied to any issue and is not confined solely to tackling privacy issues. The paper reports results from the deployment of ACH at Meta between 28th October 2024 and 31st December 2024.

Although there has been a great deal of recent attention on LLM-based test generation [3, 18, 39, 52, 53], there has been little work on developing tests for specific classes of fault. Many companies have exposure to specific high impact faults related to important issues such as security, integrity, and privacy. The importance of such issues makes it equally important to have test generation techniques that can target these specific classes of faults.

Organizations typically collect data about bugs found during development. This provides a rich source of information with which to guide test generation. The challenge, therefore, is to find a way to generate tests that target specific issues on the basis of this information. We believe mutation testing holds the key: our key insight is to construct mutants that denote faults that are both relevant to the issue of concern and also currently not caught (unkilled) by any existing test case, and to use these as prompts to LLM-based test generation. This results in an overall agenetic LLM-based workflow, in which agents essentially generate problem-specific 'super bugs' and the tests that can catch them.

[2]ACH (Automated Compliance Hardener) is so-named because it is an automated test generation tool that 'hardens' compliance with respect to chosen issues of concern.

At Meta, we have developed and deployed just such an approach and tool, ACH, for automated unit test generation. ACH's workflow is based on the principle of Assured LLM-based Software Engineering [7]. In Assured LLMSE, the goal is not merely to use language models to tackle software engineering challenges, but to automatically generate software engineering artifacts that come with *assurances*. In the case of ACH, the artifacts are tests and they come with the following assurances:

(1) **Buildable**: The proposed new tests build, so are free from syntax errors and missing dependencies;
(2) **Valid Regression Tests**: The tests pass and do so consistently, so they are non-flaky [26] regression tests;
(3) **Hardening**: The new tests catch faults that *no* existing test can catch;
(4) **Relevant**: Many of the new tests are closely coupled to the issue of concern;
(5) **Fashion Following**: Most of the new tests respect the coding style used by the existing tests available for the class under test; they are 'fashion followers' [3].

The first three assurances are boolean in nature; they are unequivocal *guarantees* made by ACH about the tests it proposes to the engineer. The fourth and fifth are more aspirational and probabilistic in nature, reflecting the underlying (LLM) technology used to construct tests. The degree to which tests are relevant and 'fashion follow' the existing tests is an inherently non-boolean and subjective attribute. As such, while the boolean criteria are provided as verifiable guarantees that ACH promises to meet, the two more probabilistic assurances are best assessed through the standard code review process routinely undertaken by engineers. Finally, when it finds that a newly-generated test adds coverage, ACH also measures coverage achieved, including the result in the assurances provided to the engineer reviewing the test.

The newly generated tests need not extend coverage, since they may find faults simply by covering existing lines of code in some new way; the same path with different data, for example. It is well known [17] that mutation testing has this property, allowing mutation testing to claim superiority over structural coverage test adequacy criteria such as line coverage. As shown in Section 6, our findings further underscore the importance of mutation testing in 'going beyond' purely structural test adequacy criteria.

The paper focuses on privacy hardening, but we believe that our approach will find many applications to software testing, more generally. It combines existing well-known and widely-studied approaches to mutation testing, assured LLMSE, and test generation. However, this new combination allows it to tackle a wide range of other problems. The potential reach and impact of this approach derives from the fact that it answers a fundamental question for automated software test generation:

> *How can we automatically transform vague, incomplete and even contradictory textual narratives about software concerns, into unit tests that guard against bugs that, if introduced, may yield field failures that manifest these concerns?*

This fundamental nature makes the approach very widely applicable: we can use it in any situation where we have a way to capture, even in the most vague of textual descriptions, concerns we have about potential issues our systems will face. Furthermore, because ACH generates simulated faults as an intermediate stage in generating tests, we can use the distributions of simulated fault prevalence over the code base as a proxy for the risk exposure of individual system components to the issue under test. Finally, through simulation [2], we can further assess the likely impact (or severity) of such faults, were they to manifest as field failures.

There are four primary contributions of the paper:

Empirical results: We report results from the application of ACH to 7 real world software platforms deployed by Meta. In total, ACH generated 4,660 candidate mutants (simulated privacy faults) that it 'believed' to be potentially killable from 10,795 classes under test. From these candidates, ACH was able to generate an additional 571 unit tests. Of these 571 tests, 277 would have been discarded had we chosen to focus solely on the line coverage test adequacy criterion, underlining the importance of mutation testing over such coverage.

Deployment Experience: We report both qualitative and quantitative outcomes from the development, deployment and evaluation of ACH at Meta in 2024. Overall, the tests automatically generated by ACH achieved an acceptance rate of 73% from the engineers who reviewed them, with 36% being judged privacy relevant.

Equivalent Mutant Detection: We present an evaluation of the ACH equivalent mutant detection agent's ability to detect equivalent mutants, which yields precision and recall of 0.79 and 0.47 respectively. The figure for precision and recall rises to 0.95 and 0.96 respectively, when the agent is combined with simple preprocessing.

Lessons from Industrial Application: We detail the lessons learned, and open problems and research challenges raised by this application of Assured LLMSE to mutation-based test generation.

2 The ACH System

Figure 1 presents the overall architectural pipeline of the ACH System. ACH starts with free from text about an issue of concern. This textual input could come from one or more of a variety of sources, including (but not limited to):

(1) Previous faults found in development;
(2) User requirements;
(3) System technical constraints;
(4) Concerns raised by engineers, managers or organizational leadership about undesirable system behaviours;
(5) Regulatory requirements set by legislative bodies and compliance enforcement organizations.

This is the sense in which ACH is a 'compliance hardener': it improves ('hardens') the ability of the deployed regression test infrastructure to detect regressions that might lead to non-compliance with respect to the issue of concern.

The results presented in this paper were obtained by using privacy hardening concerns from previous faults found in development. The single language model Llama 3.1 70Bn [42] was used in all the agents reported on. The prompts used by the three LLM agents from Figure 1 can be found in Table 1.

Our focus is the development, deployment and evaluation of mutation-guided agenetic workflows. We have not yet felt the need to extend to more sophisticated prompting, nor to use fine-tuning,

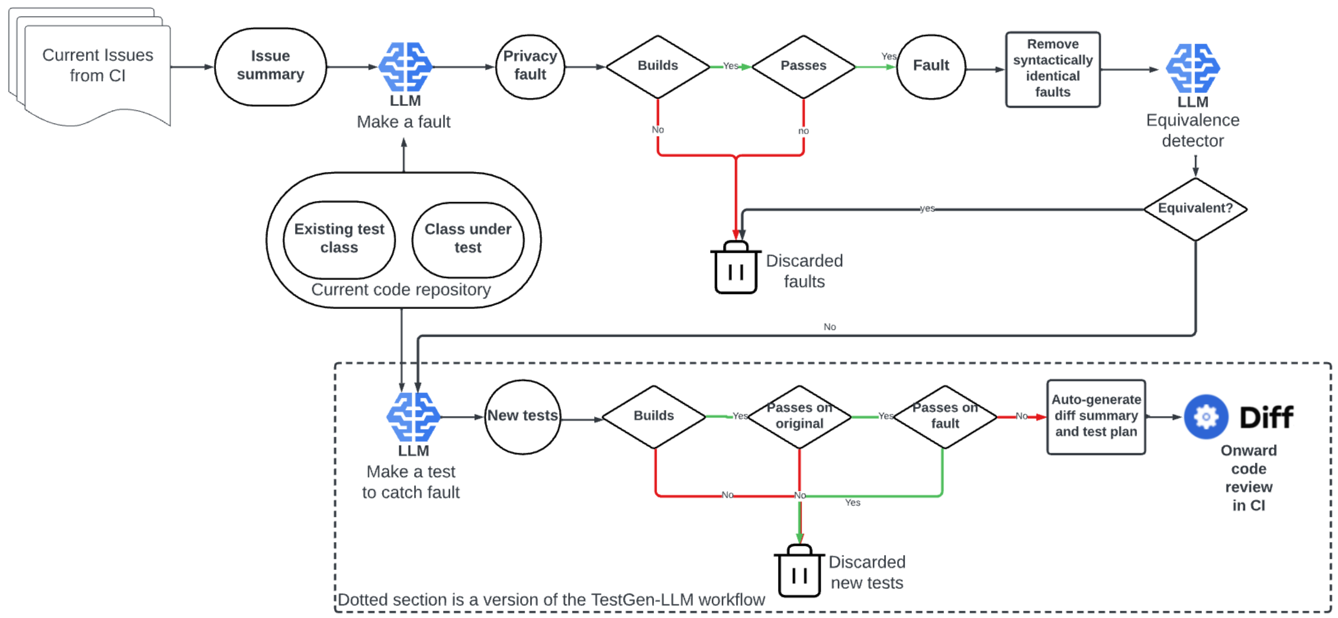


Figure 1: Top level architecture of the principal ACH components. The dotted section denotes a (slightly modified) version of the TestGen-LLM tool, on which we previously reported [3]. The workflow preceding this, shown above in the figure, is the additional agenetic workflow for generating candidate faults to drive the generation of tests. Solid rectangles denote components that are fully automated but entirely rule-based (and therefore do not use LLMs).

Table 1: The three simple prompts used in the architecture diagram in Figure 1

Agent name	Prompt Template, in which { · · · } denotes a parameter to the template
Make a fault	`CONTEXT: {context_about_concern} INSTRUCTION: Here is a Kotlin class and a test class with some unit tests for the class under test ‘‘‘{class_under_test}‘‘‘. ‘‘‘{existing_test_class}‘‘‘. Write a new version of the class under test in which each method is replaced by a new version of that method that contains a typical bug that introduces a privacy violation similar to {diff}. Delimit the mutated part using the comment-pair ‘// MUTANT <START>‘ and ‘// MUTANT <END>‘`
Equivalence detector	`I’m going to show you two slightly different versions of a Kotlin class. Here is the first version of the Kotlin class:‘‘‘class_version1‘‘‘. Here is the second version of the Kotlin class:‘‘‘class_version2‘‘‘. INSTRUCTION: If the first version of the class will always do exactly the same thing as the second version of the class, just respond with ‘{yes}‘. However, if the two versions of the class are not equivalent, respond with ‘{no}‘, and give an explanation of how execution of the first version can produce a different behaviour to execution of the second version.`
Make a test to catch fault	`What follows is two versions of a Kotlin class under test. An original correct class and a mutated version of that class that contains one mutant per method, each of which represents a bug. Each bug is delimited by the comment-pair ‘// MUTANT <START>‘ and ‘// MUTANT <END>‘. The original class and its mutant are followed by a test class that contains unit tests for the original correct class under test. This is the original version of the class under test:‘‘‘{original_class}‘‘‘. This is the mutated version of the class under test:‘‘‘{mutated_class}‘‘‘. Here is the existing test class:‘‘‘{existing_test_class}‘‘‘. Write an extended version of the test class that contains extra test cases that will fail on the mutant version of the class, but would pass on the correct version.`

nor to exploit language model ensembles [3], all of which would improve on the results we report. Therefore, our results denote only a baseline against which to measure future developments. Nevertheless, despite these limitations, we believe the results clearly highlight the potential for significant advances in test generation; its applicability and effectiveness. We would be interested and excited to collaborate with the wider research community, and hope this paper stimulates further work in this area.

In the ACH workflow depicted in Figure 1, the issue summary generates prompts that stimulate another LLM-based agent to generate faults; walking the code base, using existing tests, classes under test, and the summary as prompt ingredients. The prompts instruct the LLM to generate simulated faults (aka mutants [35]). There is no guarantee that these mutants will change the code under test, because it may not be relevant to the issue being hardened.

However, even when the fault generator does change the syntax, it may not change the semantics; the mutants could be equivalent mutants [27]. To tackle this equivalent mutant problem, ACH uses a further agent, the Equivalence Detector agent. This Equivalence Detector agent uses the LLM-as-judge approach [24, 63]. The judge is instructed to determine whether the mutant is equivalent to the original class under test. Although the underlying problem of mutant equivalence is undecidable [35], we found that this approach was surprisingly effective due to the properties of the mutants so-generated (Section 5 contains details).

The faults generated by the initial phase of the workflow are used to generate prompts for test generation. The test generation phase shown in a dotted box is a slightly modified version of the previously published TestGen-LLM tool [3].

3 Results from Meta's ACH Deployment

We applied ACH to the 7 Meta platforms Aloha, Facebook Feed, Instagram, Messenger, Oculus, Wearables, and WhatsApp. Facebook Feed and Instagram are social media platforms. WhatsApp and

Messenger are messaging platforms. Oculus is a set of virtual reality headsets. Wearables are platforms for augmented reality glasses and wrist controllers. We also included 'cross-app' software products that provide features affecting more than one of these platforms.

Table 2 presents top level summary statistics for the application of ACH to these platforms. Once a mutant is found that builds and passes, the workflow terminates[3] for the current class under test. The number of candidate mutants that successful build and pass existing tests reported in Table 2 is therefore bounded above by the number of classes under test.

By contrast with traditional rule-based mutation testing tools [19, 33, 37, 54], ACH generates relatively few, highly specific mutants, by design. Furthermore, the generated mutants have a higher probability of relevance to the issue of concern than can be achieved by rule-based approaches.

As can be seen from Table 2, ACH generates 9,095 mutants that build and pass from 10,795 classes under test. Although the language model approach generates fewer and more specific mutants than rule-based approaches, we found that it also generates more equivalent mutants. For example, as Table 2 reveals, 25% of the mutants generated are trivially syntactically equivalent. This contrasts with overall equivalent mutant generation rates that are typically around 10% to 15% for rule-based approaches [27, 45]. Fortunately, as revealed in Section 5, the equivalent mutant problem is relatively unimportant for our use case.

Of the 9,095 mutants that build and pass, 4,660 (51%) were deemed to be non-equivalent by the workflow, and thus become the subject of each of the prompts used for test generation. The test generation phase uses similar prompts to those reported previously [3], but based on covering the mutant rather than covering uncovered lines.

4 Engineers' Evaluation of ACH

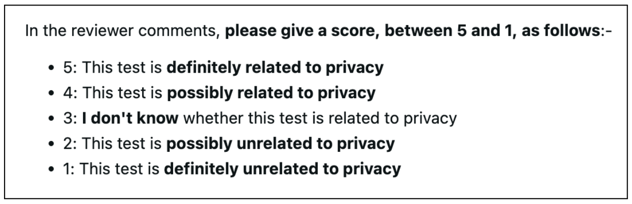
We followed the tried-and-tested formula [3] for the deployment of a new software testing technology at Meta, starting with initial trials and moving onto to test-a-thons led by engineers, thereby evaluating initial deployment.

4.1 Initial Trial

At Meta, a code change submitted to the Continuous Integration (CI) system is called a 'diff' (short for differential). In order to get initial feedback from engineers on the tests generated by the first version of ACH, we used it to generate 30 new test cases, submitting each as a separate diff for review. The generated tests were recommended to engineers in the same way as any other code modification, such as those created by human engineers. That is, ACH submits tests, as diffs, through the normal review process. The ACH diff summary explains the kind of fault caught by the test case giving, as a specific example, the mutant that ACH has already determined to be killed by the test case.

We wanted to obtain broad coverage of Meta's platforms, so included messaging apps like WhatsApp and Messenger, traditional social media platforms such as Facebook Feed, as well as hardware technologies such as Oculus and Wearable Computing, and also code that spanned over multiple apps.

[3]This decision was motivated by the use case: we first want to establish whether a test is relevant. When we find a test that is relevant, we can choose to focus in on the code it tests, hoping that adjacent code will also have a higher likelihood of relevance.


In the reviewer comments, **please give a score, between 5 and 1, as follows**:-

- 5: This test is **definitely related to privacy**
- 4: This test is **possibly related to privacy**
- 3: **I don't know** whether this test is related to privacy
- 2: This test is **possibly unrelated to privacy**
- 1: This test is **definitely unrelated to privacy**

Figure 2: Likert scale instructions given to code reviewers to score test cases according to their privacy relevance.

Table 3 presents details of these 30 diffs. As can be seen, the overall acceptance rate is 90% accepted of those submitted (27 of 30), and 93% of all diffs reviewed (27 of 29). Initial results for acceptance rate were deemed to be highly encouraging, so we proceeded to build out the Minimal Viable Product (MVP) and use it in the test-a-thons reported in Section 4.2.

The primary feedback from developers who reviewed these 30 diffs was as follows. Engineers reported that

(1) Many of the tests added coverage, which they computed themselves manually. They asked that coverage information be automatically computed and reported in the diff summaries, as well as the faults found.
(2) Some of the tests were relevant to privacy, and could be useful in hardening against future privacy regressions. They also felt that having the specific example faults was very helpful in understanding the behavior of the tests.
(3) Even for those which were not clearly related to privacy, the tests seemed to be adding value by tackling corner cases or adding coverage.

4.2 Experience from Privacy Test-a-thons

In the week of 9th December 2024, we conducted two test-a-thons, focusing on the application of ACH to Meta's two messaging platforms: WhatsApp and Messenger.

As with the initial trial, test cases were submitted as diffs into the normal continuous integration review process. However, the diff summary additionally claimed additional coverage, for those tests that did add coverage as well as finding currently uncatchable faults. The engineers participating in the test-a-thons reviewed the diffs for usefulness in the normal way they would any other diff, ultimately determining whether the diff is accepted, and thus lands into production.

In order to evaluate the privacy relevance of each test, we additionally asked the software engineers to give each a score on a Likert scale [32]. Figure 2 is a screen capture of the exact instructions given to the engineers regarding this privacy relevance scoring procedure.

Procedure for Messenger two-phase test-a-thon: The Messenger test-a-thon was conducted in two phases. The same 6 reviewers were used for both the first and the second phase. All Messenger reviewers had strong expertise in testing, and (at least a) basic experience with privacy engineering.

In the first phase, 50 generated tests were selected from a randomly selected pool of 60 tests. All 60 were prescreened manually

Table 2: Meta platforms used to evaluate ACH and the numbers of simulated privacy faults 'believed' non-equivalent by the equivalence detector agent from Figure 1. Percentages in the final four columns are distributions of ACH's equivalence 'belief' over mutants that build and pass, while those in the fourth column report the percentage of all mutants that build and pass.

Platform	Number of classes under test	Number of mutants generated in total	Number of mutants generated that build and pass	Number of candidate mutants that build and pass and also ... are syntactically identical	the agent believes to be equivalent	the agent gives no clear answer on equivalence	the agent believes to be non-equivalent
Facebook Feed	346	1,097	252 (23%)	50 (20%)	19 (7.5%)	19 (7.5%)	164 (65%)
Messenger	3,339	9,185	2,922 (32%)	742 (25%)	384 (13%)	401 (14%)	1,395 (48%)
Instagram	1,691	5,199	1,381 (27%)	277 (20%)	154 (11%)	208 (15%)	742 (54%)
Aloha	175	576	144 (25%)	32 (22%)	16 (11%)	6 (4%)	90 (63%)
Wearables	2,841	8,592	2,468 (29%)	621 (25%)	207 (8%)	326 (13%)	1,314 (53%)
Cross-app	805	1,819	562 (31%)	146 (26%)	64 (11%)	82 (15%)	270 (48%)
Oculus	325	997	279 (28%)	96 (34%)	17 (6%)	16 (6%)	150 (54%)
WhatsApp	1,273	4,212	1,087 (26%)	282 (26%)	155 (14%)	115 (11%)	535 (49%)
Totals	10,795	31,677	9,095 (29%)	2,246 (25%)	1,016 (11%)	1,173 (13%)	4,660 (51%)

Table 3: Results from initial trial ACH deployment on 6 platforms and cross-app products to gauge developers' reactions, gain feedback and guide development.

Platform	Number of tests	Status after human review is ... Accepted 'as is' by the engineer	Accepted with simple changes	Rejected by the engineer	Not reviewed
FB Feed	4	4	0	0	0
Messenger	12	8	3	1	0
Aloha	2	2	0	0	0
Wearables	3	1	0	1	1
Cross-app	2	2	0	0	0
Oculus	3	3	0	0	0
WhatsApp	4	3	1	0	0
Totals	30	23	4	2	1

and, as a result, 10 were excluded because they concerned only faults relating to Null Pointer Exceptions, two of which improved coverage, and eight of which did not. These were excluded because we initially (wrongly, as it turned out) thought that such exception-catching tests would be rejected by engineers due to being irrelevant to privacy. However, the pre-screening approach was abandoned based on experience from this first phase: Engineers proved ready to accept tests, even when they were not directly relevant, because they perceived other benefits. Therefore, in the second phase, a further 50 tests were selected at random, without pre-screening.

Procedure for the WhatsApp test-a-thon: In the WhatsApp test-a-thon, 72 tests[4] were selected randomly from a pool of 120 available and allocated to 6 engineers to review. No pre-screening was performed. All 6 engineers had a background in both privacy engineering and software testing and, as such, were well-placed with relevant expertise and highly calibrated in their expectations about privacy concerns.

[4]The decision to use 72 tests was based on the initial request from the 6 engineers that they could reasonably review 12 tests; this number being deemed large enough for calibration, yet small enough to avoid reviewer overload.

Having completed the reviews of the initial 72 tests allocated, several of the engineers requested additional tests to review, having found the experience sufficiently rewarding. Additional tests were therefore allocated at random from the remaining pool of 48 available. Therefore, in total, 91 WhatsApp tests were reviewed for usefulness, and 90 were reviewed for relevance.

Results from the test-a-thons: Table 4 presents summary statistics for the two test-a-thons. The upper table gives the number of tests reviewed for usefulness and for privacy relevance and, of these, the number that were accepted and rejected for usefulness, and scored for relevance. The lower table gives the proportions of tests reviewed for usefulness that were accepted and rejected, and the proportion of those reviewed for relevance that fall into each of the five categories of the Likert scale on which relevance was assessed by the software engineers. Percentages are rounded to the nearest whole number percentage point, and so may not quite total 100% due to rounding.

As Table 4 reveals, the overall acceptance rate was 73%. This is very similar to acceptance rates from previous test-a-thons that focused exclusively on elevating coverage [3]. Overall, about one third of the tests deemed definitely *irrelevant* to privacy. This makes it all the more interesting that the rejection rate is much lower than this, at 27%. Indeed, only approximately 36% were deemed to be either possibly or definitely related to privacy. We therefore observe that the engineers are very willing to accept tests which may not be relevant to their current concern, when they are found useful for other reasons.

In looking at the individual comments left by the engineers on each of the 191 tests that were reviewed for usefulness, we observe, more anecdotally, that the engineers were likely to accept tests for two primary reasons:

(1) They add line or branch coverage of non-trivial code
(2) They covered a tricky corner case, such as handling special values, even when this failed to add coverage

Tests that were rejected tended not to add coverage, or to add coverage only of trivial code, such as one-line behavioral functions. Tests were also rejected if they were written in a style that was

Table 4: Results from WhatsApp and Messenger Test-a-thons, where ACH was deployed and evaluated in December 2024.

Test-a-thon	total number of tests reviewed for **usefulness**	total number of tests that were ... accepted	rejected	total number of tests reviewed for **relevance**	privacy relevance scored at level ... 5	4	3	2	1
WhatsApp	91	50	41	90	6	29	8	24	23
Messenger Phase 1 (pre-screened)	50	47	3	44	5	8	5	4	22
Messenger Phase 2 (not pre-screened)	50	43	7	41	4	11	0	7	19
Overall Total	191	140	51	175	15	48	13	35	64

Proportions of tests in each category for usefulness and relevance :-

		accepted	rejected		5	4	3	2	1
Over all test-a-thons		73%	27%		9%	27%	7%	20%	37%
WhatsApp		56%	44%		7%	32%	9%	26%	26%
Messenger Phases 1 and 2		90%	10%		11%	22%	6%	13%	48%
Messenger Phase 1 alone		94%	6%		11%	18%	11%	9%	50%
Messenger Phase 2 alone		86%	14%		10%	27%	0%	17%	46%

deemed to be unsuitable, such as using a deprecated API, something that was also observed in previous work [3].

In terms of the relevance of the tests, we believe that the finding that 36% are deemed relevant to privacy is a positive overall outcome. This is because we devoted so little of the overall process to weeding out tests that could be automatically determined to be irrelevant. As such, the result for relevance can be considered a baseline for comparison with future work. We believe that with additional static analysis, and further LLM-as-judge agents in the overall workflow, it could be considerably improved.

We also noticed that the comments about relevance for tests scored 4 and 2 were often quite similar, indicating uncertainty, and the belief that the test may be relevant to privacy, but the engineer was not certain. Therefore, an upper bound on those potentially relevant to privacy is approximately two thirds of those tests assessed. Given the 73% acceptance rate, we conclude that ACH adds privacy hardening in about one third of the cases, and does not cause unnecessary developer friction in considering the remaining cases.

There is an interesting difference in the relevance score between the two phases of the Messenger test-a-thon results reported in Table 4. In the first phase, the engineers stated that they were unsure in 10% of the cases, but this percentage dropped to zero in the second phase. This apparent growing scoring 'confidence' indicates a potential 'learning effect', which has been seen in similar empirical studies of software engineers during multiphase trials [21].

Finally, looking at the differences between the acceptance rates and relevance assessments for WhatsApp and Messenger, we also see interesting differences. WhatsApp engineers rated the tests to be at least as relevant to privacy (39% for WhatsApp vs. 33% for Messenger), yet accepted fewer tests (56% WhatsApp vs. 89% for messenger). All tests that were acceptable for usefulness were landed into production. We believe that the different acceptance rates may simply denote different cultures between different teams; deciding to land a test into production is an inherently subjective judgment and may be influenced by team culture.

5 The Equivalent Mutant Problem

All approaches to mutation testing need to tackle the problem of equivalent mutants [35]: the mutant may be *syntactically* different to the original program, but we cannot guarantee it will be *semantically* different, because the underlying program equivalence problem is undecidable [27, 59].

There are a number of techniques in the literature that can weed out *some* of the equivalent mutants [40]. However, the undecidability of the problem means that some equivalent mutants will inevitably remain, so engineers and automated test generators may waste time trying to kill (unkillable) equivalent mutants.

5.1 Equivalent Mutants Have no Impact on ACH

For the application of mutation-guided test generation, the equivalent mutant problem has no direct impact on engineers. Our workflow requires only that engineers review test cases, not mutants. The only scenario in which an engineer *might* consider looking at a mutant, would be to see an example of the kind of faults that can be caught by the test case they are reviewing. By construction, such mutants are *non-equivalent*, so the engineer will *never* see an equivalent mutant. This relegates the equivalent mutant problem to a relatively subordinate position in our overall use case for mutation testing.

Nevertheless, it would be inefficient to generate many equivalent mutants, because ACH would waste computational resources trying to kill the unkillable. We therefore incorporate, into the agentic workflow, an LLM-based agent that checks for mutant equivalence. The remainder of this section reports on the evaluation of the effectiveness of this agent.

5.2 Detecting Equivalent Mutants

To evaluate the performance of the equivalence detector agent from Figure 1, we performed a manual study on a random selection of mutants drawn from the four platforms with the most mutants available. The purpose of this manual analysis is to answer the research question:

How good is the Equivalence Detector Agent?

We must manually analyze mutants to answer this research question, because it requires a ground truth for equivalence. However, it is important to underline that it is *never* necessary for a *practicing software engineer* to manually evaluate *any* mutant.

For each of the four platforms, we attempted to sample 100 mutants that would still build and pass on the original code. The actual number manually analyzed for each sample was slightly different to the intended 100, because some code may have changed since the mutants were constructed. This is the Mutant Relevance Problem [43]: previously generated mutants may no longer be relevant when the code changes after they were constructed.

Mutant relevance does not impact the deployment of ACH, since ACH generates mutants on the fly, discarding them once tests have been generated to kill them. However, it did have a minor impact on our mutant sampling to evaluate the research question: Since we cannot be sure mutants in changed code remain valid at manual inspection time, we discarded them from the human analysis sample.

In our evaluation of equivalence detection, we are concerned only with so-called 'weak' mutation testing, not 'strong' mutation testing [35]. That is, because ACH is a unit test generation technology, we do not need to consider failed error propagation [9], which would be required for strong mutation testing. Failed error propagation is the situation in which mutant execution changes the local computation state, yet this change is always masked along every path to an observable output. In weak mutation testing, the mutant is non-equivalent if the local state is changed, irrespective of the whether the change propagates. Manually detecting failed error propagation is a highly labor-intensive process, so we are fortunate that it is not required.

Table 5 describes the types of mutant, based on our manual analysis. In this analysis, the terms 'deletion' and 'injection' each refer to semantic changes where the sole change is, respectively, the addition of values or the removal of values. The term 'Injection and deletion' refers to cases where the semantics are changed by an independent combination of addition and removal. The category 'other' covers cases where the change is more nuanced. For example replacing one variable name with another, which could be counted as a deletion and an injection, but where the two are not independent.

In Table 5, the column 'equivalence is not obvious' counts the number of cases where the human assessor had to think noticeably longer about whether the mutant was equivalent. For all other cases, the decision was obvious. As Table 5 reveals, the decision is 'obvious' for a surprisingly large number of mutants.

The category 'Misleading comment' is perhaps the most interesting category. These are cases where the mutant is clearly equivalent, because all the mutant generation agent has done is to insert comments; no executable code has been added. An example of such a misleading injected comment is: "`// Introduce a bug by not checking if the user is inactivated before resetting their custom reactions`". Since language models are text-based predictive technologies that currently do not specifically distinguish executable code from comments, such misleading comments are clearly problematic. They can easily mislead the equivalence detector agent into predicting that a mutant is *not* equivalent, when it clearly *is* equivalent.

The reason that this category is so interesting, is that ACH could simply strip out any comments added when constructing a mutant before submitting it to the equivalence detector agent. This simple pre-processing step would render all cases of misleading comments syntactically identical. They would then be automatically discarded without even needing to consult the Equivalence Detector Agent. Based on the evaluation reported here, we decided to incorporate this pre-processor into ACH's workflow.

5.3 Equivalence Detector Precision and Recall

Table 6 reports the overall precision and recall of the equivalence detector agent. The detailed results for each platform are shown in Table 7. As Table 6 reveals, when we consider 'unsure' as the same as 'equivalent', the precision is good, at 0.79 over all platforms studied. However, recall is relatively low, at 0.47. If we wanted even greater precision we could treat 'unsure' as 'non-equivalent', since this gives precision of 0.97 (and a recall of 0.44).

Based on this precision and recall, we can have high confidence that, when the detector determines a mutant to be equivalent, it is very likely to be correct. However, it weeds out only approximately half of the equivalent mutants. High precision means ACH will not discard many non-equivalent mutants. Not losing mutants means not losing the tests that ACH might generate from them. By contrast, higher recall would merely save some computational resource.

As such, high precision is generally more valuable than high recall for our use case, because we are prepared to spend computational resources to automatically generate good unit tests. However, the lower the recall, the more often ACH will inefficiently seek to 'kill the unkillable', so we still want the highest recall achievable without reducing precision.

As Table 2 shows, approximately 25% of all mutants are trivially equivalent, being syntactically identical to the original. Furthermore, Table 5 reveals that 61% of all equivalent mutants are almost trivially equivalent, because they contain only a single 'misleading' comment. Syntactically identical mutants are removed by a simple lexical comparison, while a simple pre-processing transformation to remove comments would remove the 'misleading comments' category. Therefore, we can combine the LLM-based detector with a simple rule based pre-processor to yield an overall precision and recall of 0.95 and 0.96 respectively (See Table 6).

These figures for precision and recall are surprisingly high, given that the underlying problem is undecidable. However, we cannot claim that these surprisingly good results arise because the mutant equivalence detector agent is excellent at determining program equivalence. Rather, they are more reflection of the kind of mutants the detector needs to judge for equivalence. That is, the mutant generation agent tends to make changes that either obviously introduce semantic changes, or are obviously equivalent (such as those that only change comments).

6 The Importance of Mutation Testing

It is well known in the literature on testing, both theoretically and empirically, that mutation adequacy criteria outperform traditional structural criteria, such as line coverage and branch coverage [17, 46]. In this section, we present results that further underline

Table 5: Number of mutants of different semantic types generated. We used human analysis to determine ground truth equivalence, and the different types of semantic transformation used to generate a mutants.

		mutant equivalence			semantic type of mutation transformation				
Platform	Total mutants analyzed	Ground truth equivalent	LLM agent claims equivalence	Equivalence is not obvious	Deletion	Injection	Deletion and injection	Misleading comment	Other
Messenger	91	38 (42%)	27 (30%)	2 (2%)	23 (25%)	32 (35%)	19 (21%)	11 (12%)	6 (7%)
Wearables	101	32 (32%)	11 (12%)	3 (3%)	26 (26%)	24 (24%)	20 (20%)	24 (24%)	7 (7%)
WhatsApp	99	30 (30%)	16 (16%)	3 (3%)	29 (29%)	26 (26%)	16 (16%)	26 (26%)	2 (2%)
Instagram	90	37 (41%)	12 (13%)	2 (2%)	27 (30%)	14 (16%)	11 (12%)	32 (36%)	6 (7%)
Overall Totals	381	137 (36%)	66 (17%)	10 (2.6%)	105 (28%)	96 (25%)	66 (17%)	93 (24%)	21 (6%)

Table 6: Precision & recall for the equivalence detector agent

when unsure is simply not counted					
TP	FP	TN	FN	Precision	Recall
56	2	205	72	0.97	0.44

with unsure counted as equivalent					
TP	FP	TN	FN	Precision	Recall
65	17	205	72	0.79	0.47

with identical code counted as equivalent					
TP	FP	TN	FN	Precision	Recall
161	17	205	72	0.90	0.69

with the stripping of all added comments					
TP	FP	TN	FN	Precision	Recall
183	9	251	8	0.95	0.96

these previous research findings with direct experience from the deployment of mutation testing on large scale industrial systems.

Table 8 reports line coverage results for the tests that kill non-equivalent mutants. For Messenger, Instagram, Wearables and WhatsApp the 'likely actual' number is based on the ground truth equivalent proportion for these platforms in Table 5. For the other platforms, it is based on the overall average ground truth proportion over all four platforms. The final three columns show the results for TestGen-LLM [3] generated tests, as proportions for comparison with the corresponding proportions for ACH generated tests. TestGen-LLM does not target specific faults, but merely attempts to generate tests that will acquire extra coverage.

As Table 8 shows, a large proportion (49%) of test cases that uniquely additionally kill a mutant, do not also add line coverage. Clearly, these tests have value, since they catch faults that would otherwise go undetected. However, if we were to judge test cases solely on the basis of the coverage they add, then such valuable test cases would be wrongly discarded.

The results in Table 8 also reveal that, although TestGen-LLM generates tests for a higher proportion of classes compared to ACH (32% vs. 5.3%), it nevertheless, succeeds in killing a far smaller proportion of mutants (2.4% vs 15%). This is because ACH specifically targets the mutants, whereas TestGen-LLM does not.

From these results, we conclude that targeting mutants can also elevate coverage, but targeting coverage will be inadequate to kill mutants. Although this is already known from the research literature [17, 46], that literature is based on laboratory controlled experiments with non-industrial systems. Our findings thus augment the existing literature and add weight to claims for the importance of mutation testing, based on industrial practice and experience.

It is also worth noting that 70% of the mutants left unkilled by TestGen-LLM reside in classes that do not have *any* TestGen-LLM test, let alone one that kills the mutant. Combined with the high proportion (51%) of ACH tests that *do* also raise coverage, we conclude that Mutation-Guided LLM-based Test Generation has attractive side benefits on coverage.

Specifically, we may be able to adapt our ACH approach to target coverage using mutants: Suppose we generate many mutants, of all different kinds, and at all different levels of abstraction, for a given method under test. We can then use each as a prompt to an LLM. In this way, mutants play a role similar to Retrieval Augmented Generation (RAG). Given our results and the fact that RAG is known to improve LLM performance on other Software Engineering tasks [22], we believe mutation-as-RAG for coverage is likely to prove attractive for coverage as well as for targeting specific faults.

7 Related Work

Given the strong empirical evidence for the predictability of code [10, 23, 28], it is unsurprising that predictive language models have proved effective at generating usable code. As a result, LLMs now play a code-generation role in a significant number of applications across the spectrum of software engineering activity [22].

Software engineers' acceptance rates for LLM-generated code have been widely studied. For example, researchers at Google focussed on factors that may influence trust in AI-powered code completions [13], while it was recently reported that Copilot had an acceptance rate of approximately 30% [55].

However, such studies concern the code completion use case, which does not come with any assurances (the code completions may not even compile). By contrast, ACH uses Assured LLM-Based Software Engineering (Assured LLMSE) [7]. That is, ACH provides assurances about the semantics and usefulness of the tests it proposes. Its proposals are also *whole* compilable code units (not merely completion fragments), and it is deployed on-demand (to generate tests targeting classes of faults of particular interest), rather than opportunistically (to suggest code completions). The fact that ACH provides these assurances, and its different deployment route, mean that we can expect a higher overall acceptance rate from ACH, than we would for code completion technologies.

Table 7: Precision (Prec.) and Recall achieved by the equivalence detector agent for the four platforms with most mutants

Messenger Mutants						Wearables Mutants						WhatsApp Mutants						Instagram Mutants					
When the agent is ‘unsure’, this is simply not counted in determining precision and recall																							
TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall
17	2	50	19	0.89	0.47	11	0	55	18	1.0	0.38	16	0	51	11	1.0	0.59	12	0	49	24	1.0	0.33
When the agent is ‘unsure’, this is counted as if it had determined the mutant to be equivalent																							
TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall
19	3	50	19	0.86	0.50	14	1	55	18	0.93	0.50	19	9	51	11	0.68	0.63	13	4	49	24	0.76	0.35
When we include mutants that are syntactically identical in the numbers counted as determined to be equivalent																							
TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall
41	3	50	19	0.93	0.58	42	1	55	18	0.98	0.70	47	9	51	11	0.84	0.81	31	4	49	24	0.86	0.56
When we additionally include, as determined to be equivalent, those mutants that are syntactically identical after stripping any comments added by the generation agent																							
TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall	TP	FP	TN	FN	Prec.	Recall
41	3	67	2	0.93	0.95	42	1	69	4	0.93	0.91	47	1	66	0	0.99	1.0	53	4	49	2	0.93	0.96

Table 8: The numbers of tests that add coverage as well as detecting mutated faults. This gives an upper bound on the number of faults we might miss by merely targeting new coverage alone. Approximately half of all tests generated, although finding faults missed by all existing tests, do not add line coverage. Had we used coverage as our sole test adequacy criterion, the platforms would thus have continued to be vulnerable to regressions denoted by up to approximately half of the faults.

Platform	Number of (believed; likely actual) non-equivalent mutants	Number of mutant-killing tests	%age of classes with a mutant-killing test	%age mutant kill rate on (believed; likely actual)	Number (%age) of tests that do not add coverage	%age Classes with a TestGen-LLM test	%age TestGen-LLM kill rate on (believed; likely actual)	%age unkilled mutants with no TestGen-LLM test
Facebook Feed	164; 133	15	4.3%	9%; 11%	9 (60%)	33%	3.0%; 3.7%	71%
Messenger	1,395; 1,155	196	5.9%	14%; 17%	84 (43%)	31%	2.0%; 2.4%	71%
Instagram	742; 687	122	7.2%	16%; 18%	77 (55%)	27%	2.9%; 3.1%	74%
Aloha	90; 74	9	5.1%	10%; 12%	4 (44%)	27%	1.0%; 1.2%	76%
Wearables	1,314; 1,007	45	1.6%	3%; 4%	32 (71%)	31%	1.9%; 2.5%	70%
Cross-app	270; 275	35	4.3%	13%; 13%	20 (57%)	28%	2.8% 2.7%	75%
Oculus	150; 121	14	4.3%	9%; 12%	3 (21%)	35%	4.9%; 6.1%	70%
WhatsApp	535; 445	135	10.6%	25%; 30%	48 (36%)	44%	3.5%; 4.2%	55%
Totals	4,660; 3,897	571	5.3%	12%, 15%	277 (49%)	32%	2.0%; 2.4%	70%

While code completions are deployed directly into the IDE in real time (online), ACH is an offline technology that proposes code updates in exactly the same way that engineers would propose them: through Continuous Integration (CI) and standard code review. This has the advantage that the decision to accept machine-generated code has an audit trail, which is missing for LLM-based code completions. By contrast, LLM-based code completion suggestions accepted by engineers are attributed to the engineer by the CI audit trail; details of the LLM’s role in constructing production code through completion suggestions is thus unavailable.

There is already a considerable body of literature on LLMSE [22]. The subset of this literature devoted specifically to software testing is too large to survey in detail here. Instead, we refer readers to recent surveys [22, 61]. In this section, we focus specifically on the application of LLMs to mutation testing.

Traditionally [35], mutation testing techniques have used sets of pre-defined rule-based operators to construct mutants. However, it has been widely observed [36, 51] that such pre-defined rule-based mutation operators are ill-suited to the task of generating *realistic* faults. This has led researchers to consider ways to make more realistic mutants [34], a problem to which language models have recently been applied [57, 60]. ACH represents an extension of this direction that additionally seeks to generate a set of faults, relevant to a chosen issue of concern.

Mutation testing has also been used as a way to evaluate language models themselves [38] and to mutate compiler inputs to test compilers [31]. In this work, the desirable property of mutants is the way that they generate near neighborhoods of programs that are similar to the program that they mutate, both syntactically and semantically. This allows researchers to explore properties of language models when applied to code, such as their ability to ‘understand’ the code. This ‘near neighborhood’ property has also been used in combination with language models [14] to generate mutants to help improve other software engineering technologies, such as Genetic Improvement [47].

The equivalent mutant problem has been widely studied [35, 40], and we are not the first to consider using language models to tackle it. Tian et al. [56] recently studied the use of language models to detect equivalent mutants for traditional rule-based mutation testing operators on MutantBench. MutantBench [59] is a set of mutant-original-code pairs, written in C/C++ and Java, introduced in 2021 by van Hijfte and Oprescu. Tian et al. found that LLM-based equivalence detection outperformed traditional non-LLM-based equivalence detection by 35% for F1 score (the harmonic mean of precision and recall). In particular, they highlighted the high-performance of fine-tuned embeddings.

All the previous work on LLMs for mutation testing has concerned benchmark problems and empirical analysis under laboratory conditions. The present paper is the first to report the deployment of LLM-based mutation testing at scale in industry. Non-LLM-based mutation testing has previously been evaluated in industrial application settings, at Meta [12] and at Google [48–50]. However, in both these previous industrial deployments, there was no attempt to automatically generate tests to *kill* the mutants. The challenging task of constructing tests to kill mutants was left to engineers, who also therefore had to contend with the equivalent mutant problem. Unlike these previous industrial deployments, ACH is also the first to automatically generate tests to kill mutants; a problem which has been studied in the research literature [16, 25], but not previously deployed at scale in industry.

The work on ACH reported here, follows a succession of automated analysis and testing work deployed at Meta. Our initial focus was end-to-end test generation tools such as Sapienz [5, 41], and static analysis tools such as Infer [15], followed by simulation-based test generation for testing interacting user communities [1, 2, 58]. While much of this previous work has been concerned with system level testing, we have also worked on unit level test generation techniques using observations [8] and language models [3]. However, this previous unit test generation work was based on the sole objective of increasing coverage, which meant it could not target specific classes of faults as ACH does.

8 Open Problems

We outline directions for future work, hoping to stimulate the research community with new open research questions.

Mutant Equivalence: we found that the simulated privacy faults generated by language models tend to be bimodal, with the consequence that equivalent mutants are surprisingly easy to detect (See Section 5). It is possible that these results are merely specific to Kotlin, to Android, to Meta, or to simulated privacy faults.

However, there is nothing in our approach that suggests that the results would fail to generalize. If they *were* to generalize, this would be an important finding for the mutation testing research agenda, given the significant impediment to deployment hitherto posed by mutant equivalence.

Mutant Relevance: We have been able to generate specific mutants that capture similar faulty behaviors as those previously witnessed. However, in some cases, anecdotally, looking at the faults generated, it seemed that the mutant was related to the general class of fault simulated, but was not an example of the specific instance. One difficulty here is that we have no way to consistently and reliably measure problem similarity or relevance.

More research is needed to define what it means for one fault to be similar to another. We also need approaches that use such similarity metrics to guide agenetic LLM workflows, e.g., with fine-tuning, prompt engineering, re-prompting, and/or Chain-of-Thought, to ensure that the mutants generated are relevant to the original fault.

Detecting existing faults: our approach hardens against future regressions. This means that the current version of the system under the test is used as the regression oracle [6]: a test is deemed to pass after some change if the test behaves the same before and after the change. This allows us to protect against future regressions, but it cannot detect *existing* faults residing in the code base. To do this requires us to tackle the well known Oracle problem [11].

It would be very exciting if an approach can be found to infer oracles for Targeted LLM-based Mutation-Guided Test Generation. Such an advance would be highly impactful because it would allow us to search for existing classes of faults. Although there has been much previous work on oracle inference [30, 44, 58, 62], we need higher precision to avoid wasting engineers' time on false positives. Generating faults/tests for specific issues of concern, as ACH does, may help oracle generation.

9 Conclusions

Mutation testing has been the subject of research for five decades [20, 29, 35, 46], so has clearly retained intellectual appeal and researchers' curiosity. Despite this, it has proved challenging to deploy mutation testing in industry [4, 6, 12].

Traditionally, mutants have been constructed using simple *rule-based* approaches in order to *assess* test suites, largely *written by humans*. However, software engineers need automatically generated tests, that are relevant to a specific pressing concern; their time would be better spent articulating these pressing concerns, rather than trying to construct test cases that should, instead, be generated by a machine. Fortunately, two crucial recent advances, drawn together in ACH, make this possible: Automated test generation to kill generated mutants, coupled with language models' ability to generate highly-relevant mutants. In particular, we found that LLMs help us to overcome existing barriers [12] to deployment of mutation testing. Specifically, they

(1) Allow us to generate highly realistic faults, using their ability to convert text (the concern or issue) into code (simulated faults from which we generate tests);
(2) Provide an additional agent to weed out equivalent mutants, which is especially powerful when combined with simple static analyses as a pre-processing step;
(3) Provide a way to automatically generate unit tests to kill the mutants.

This paper presented results from deployment of ACH at Meta. Neither LLM-based test generation, nor LLM-based mutant generation is new, but this paper is the first to report on their combined deployment on large scale industrial systems. We believe this form of automated test generation is highly aligned with modern software development and deployment. It supports software engineers who must contend with many competing and conflicting concerns, often expressed in natural language in vague, incomplete and even contradictory ways. Our results also suggest Mutation-as-RAG will prove impactful in optimizing for structural coverage criteria.

ACKNOWLEDGMENTS

We would like to thank the Meta's Llama team and leadership of the Fundamental Artificial Intelligence Research (FAIR), Developer Infrastructure (DevInfra) and Product Compliance and Privacy teams for supporting this work and the Meta Software Engineers who so freely and kindly gave of their time, experience, and expertise in reviewing the tests automatically generated by ACH.

References

[1] 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. Testing web enabled simulation at scale using metamorphic testing. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*, Virtual, 2021.

[2] John Ahlgren, Kinga Bojarczuk, Sophia Drossopoulou, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Simon Lucas, Erik Meijer, Steve Omohundro, Rubmary Rojas, Silvia Sapora, Jie M. Zhang, and Norm Zhou. Facebook's cyber–cyber and cyber–physical digital twins (keynote paper). In *25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*, Virtual, June 2021.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using Large Language Models at Meta. In *ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, July 2024.

[4] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. Some challenges for software testing research (keynote paper). In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019), Beijing, China, July 15-19, 2019*, pages 1–3. ACM, 2019.

[5] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with Sapienz at Facebook (keynote paper). In 10^{th} *International Symposium on Search Based Software Engineering (SSBSE 2018)*, pages 3–45, Montpellier, France, September 8th-10th 2018. Springer LNCS 11036.

[6] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. Software testing research challenges: An industrial perspective (keynote paper). In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST 2023)*, pages 1–10. IEEE, 2023.

[7] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Assured llm-based software engineering (keynote paper). In $2^{nd.}$ *ICSE workshop on Interoperability and Robustness of Neural Software Engineering (InteNSE)*, April 2024.

[8] Nadia Alshahwan, Mark Harman, Alexandru Marginean, and Eddy Wang. Observation-based unit test generation at meta. In *Foundations of Software Engineering (FSE 2024)*, 2024.

[9] Kelly Androutsopoulos, David Clark, Haitao Dan, Mark Harman, and Robert Hierons. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In 36^{th} *International Conference on Software Engineering (ICSE 2014)*, pages 573–583, Hyderabad, India, June 2014.

[10] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In 22^{nd} *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, pages 306–317, Hong Kong, China, November 2014.

[11] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[12] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 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)*, pages 268–277. IEEE, 2021.

[13] Adam Brown, Sarah D'Angelo, Ambar Murillo, Ciera Jaspan, and Collin Green. Identifying the factors that influence trust in ai code completion. In *1st ACM International Conference on AI-powered Software (AIware 2024)*, July 2024.

[14] Alexander Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. Enhancing genetic improvement mutations using large language models. In *International Symposium on Search Based Software Engineering*, pages 153–159. Springer, 2023.

[15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium*, pages 3–11, 2015.

[16] Francisco Carlos, Mike Papadakis, Vinícius Durelli, and Eduardo Márcio Delamaro. Test data generation techniques for mutation testing: A systematic mapping. In *Workshop on Experimental Software Engineering (ESELAW'14)*, 2014.

[17] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 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*, pages 597–608, 2017.

[18] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for LLM-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[19] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for Java. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 449–452, 2016.

[20] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practical programmer. *IEEE Computer*, 11:31–41, 1978.

[21] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, 2003.

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

[23] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *FSE*, pages 147–156, 2010.

[24] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on LLM-as-a-judge. *arXiv preprint arXiv:2411.15594*, 2024.

[25] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 212–222, New York, NY, USA, September 5th - 9th 2011. ACM.

[26] Mark Harman and Peter O'Hearn. 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)*, pages 1–23, Madrid, Spain, September 23rd-24th 2018.

[27] Mark Harman, Xiangjuan Yao, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In 36^{th} *International Conference on Software Engineering (ICSE 2014)*, pages 919–930, Hyderabad, India, June 2014.

[28] Abram Hindle, Earl Barr, Zhendong Su, Prem Devanbu, and Mark Gabel. On the naturalness of software. In *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012.

[29] William E. Howden. Theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

[30] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 70–81, 2022.

[31] Davide Italiano and Chris Cummins. Finding missed code size optimizations in compilers using LLMs. In *Compiler Construction*, 2025. To appear.

[32] Andrew T Jebb, Vincent Ng, and Louis Tay. A review of key likert scale development advances: 1995–2019. *Frontiers in psychology*, 12(637547), 2021.

[33] Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In 3^{rd} *Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, August 2008.

[34] Yue Jia and Mark Harman. Higher order mutation testing. *Journal of Information and Software Technology*, 51(10):1379–1393, 2009.

[35] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

[36] Matthieu Jimenez, Thierry Titcheu Chekam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural?: a study on how "naturalness" helps mutant selection. In Markku Oivo, Daniel Méndez Fernández, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 3:1–3:10. ACM, 2018.

[37] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 433–436, 2014.

[38] Ziyu Li and Donghwan Shin. Mutation-based consistency testing for evaluating the code understanding capability of LLMs. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 150–159, 2024.

[39] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. LLM-Powered test case generation for detecting tricky bugs. *arXiv preprint arXiv:2404.10304*, 2024.

[40] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2013.

[41] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for Android applications. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 94–105, 2016.

[42] Meta. Introducing Llama 3.1: Our most capable models to date, July 2024.

[43] Milos Ojdanic, Mike Papadakis, and Mark Harman. Keeping mutation test suites consistent and relevant with long-standing mutants. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2067–2071, 2023.

[44] Rafael AP Oliveira, Upulee Kanewala, and Paulo A Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in computers*, 95:113–199, 2014.

[45] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In 37^{th} *International Conference on Software Engineering (ICSE 2015)*, pages 936–946, Florence, Italy, 2015.

[46] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[47] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, June 2018.

[48] Goran Petrović and Marko Ivanković. State of mutation testing at Google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 163–171, 2018.

[49] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from Google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2021.

[50] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53. IEEE, 2018.

[51] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173. IEEE, 2022.

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

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

[54] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for Java. In 7^{th} *joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2009)*, pages 297–298, 2009.

[55] Richard Speed. Github: 30% of copilot coding suggestions are accepted, June 2023.

[56] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1733–1745, 2024.

[57] Frank Tip, Jonathan Bell, and Max Schäfer. LLMorpheus: Mutation testing using large language models. *arXiv preprint arXiv:2404.09952*, 2024.

[58] Shreshth Tuli, Kinga Bojarczuk, Natalija Gucevska, Mark Harman, Xiao-Yu Wang, and Graham Wright. Simulation-driven automated end-to-end test and oracle inference. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 122–133. IEEE, 2023.

[59] Lars van Hijfte and Ana Oprescu. Mutantbench: an equivalent mutant problem comparison framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 7–12. IEEE, 2021.

[60] Bo Wang, Mingda Chen, Youfang Lin, Mike Papadakis, and Jie M Zhang. An exploratory study on using large language models xfor mutation testing. *arXiv preprint arXiv:2406.09843*, 2024.

[61] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language model: Survey, landscape, and vision, 2023. arXiv:2307.07221.

[62] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. Search-based inference of polynomial metamorphic relations. In Ivica Crnkovic, Marsha Chechik, and Paul Gruenbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 701–712, Vasteras, Sweden, September 15-19 2014.

[63] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging LLM-as-a-judge with MT-bench and chatbot arena. *Advances in Neural Information Processing Systems (NeurIPS 2023)*, 36:46595–46623, 2023.