Software Testing Research Challenges: An Industrial Perspective

Nadia Alshahwan
Instagram Product Foundation
Meta Platforms Inc.
London, UK

Mark Harman
Instagram Product Foundation
Meta Platforms Inc.
London, UK

Alexandru Marginean
Instagram Product Foundation
Meta Platforms Inc.
Miami, USA

***Abstract*—There have been rapid recent developments in automated software test design, repair and program improvement. Advances in artificial intelligence also have great potential impact to tackle software testing research problems. In this paper we highlight open research problems and challenges from an industrial perspective. This perspective draws on our experience at Meta Platforms, which has been actively involved in software testing research and development for approximately a decade. As we set out here, there are many exciting opportunities for software testing research to achieve the widest and deepest impact on software practice. With this overview of the research landscape from an industrial perspective, we aim to stimulate further interest in the deployment of software testing research. We hope to be able to collaborate with the scientific community on some of these research challenges.**

***Index Terms*—Automated Software Engineering, Software Testing, Automated Program Repair, Artificial Intelligence, Genetic Improvement, Automated Remediation.**

I. Introduction

We give an industrial perspective on research challenges, concerned with three broadly related areas of software testing and optimisation (improvement) research:

1) Software Test Generation (Section II)
2) Automated Repair and Improvement (Section III)
3) Automated Transplantation and Refactoring (Section IV)

In Section V we consider opportunities for the incorporation of artificial intelligence techniques in tackling these problems, both in their own right, and in combination with existing software engineering research. Finally, in Section VI, we collect together some of the lessons learned from previous work on the industrial deployment of software testing and improvement research.

II. Test Generation: Remaining Challenges

Automated software testing has been widely studied by the research community, leading to considerable industrial uptake, for example at Meta [1], [2], Microsoft [3] and Google [4]. Nevertheless, there remain many open challenges that represent opportunities for the research community.

The authors would like to thank the many Meta engineers, managers and leadership for their assistance, support and work on deploying automated software testing and improvement techniques. Author order is alphabetical. Mark Harman's scientific work is part supported by European Research Council (ERC), Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC) which is run out of University College London, where he is part time professor. He is a full time Research Scientist at Meta Platforms Inc.

A. Regression Testing

Traditionally, regression testing has focused on functional correctness and reliability (exceptions, crashes, etc). While functional correctness and reliability concerns are undoubtedly important, we need more work to tackle performance regressions such as those that affect execution time, memory usage, and disk space.

There is an important research challenge here at the intersection of statistics and software testing research. For functional correctness, a test observation such as a crash, tends to be unequivocal. Even a highly flaky test which leads to only an occasional crash provides existential proof that crashing is possible [1]. Sadly, this luxury does not extend to many forms of performance regressions. Test observations are made in a highly nondeterministic noisy space, in which execution times naturally vary.

There are multiple ways to tackle this problem. One intuitive approach, although not necessarily the best, is to set thresholds that determine whether the regression is deemed to have occurred. Setting thresholds makes performance more 'Boolean' (as it is for the reliability scenario): either a regression occurs, or it does not; there is no ambiguity.

Although the threshold-based approach is readily deployable it tends to suffer from the problem of test flakiness [5], [6]. Non-determinism means that observations may fluctuate either side of the threshold. We need more research on nuanced approaches that can identify relatively small performance regressions in noisy observation spaces. Also, we need research on techniques for giving confidence intervals on performance regressions. This is a particularly challenging research problem when also constrained by cost considerations. Taking account of cost is essential, because it is impractical to simply generate a large number of test observations in order to increase confidence intervals.

B. Unit Testing

Unit tests are typically considered to reside at the bottom of a testing 'pyramid' [7]. Above unit tests reside integration tests and, above them, end to end tests. The higher up the pyramid, the higher the level of test abstraction and the greater

the scope tested, but also the greater the effort required to design and compute test outcomes; at least, this is the *theory*. Specifically, in principle, unit tests should be simple to write, fast to execute, and easy to understand [8]. That is, simple, easy, and fast, by contrast to tests at higher levels of the testing pyramid. However, in practice, unit tests can prove to be far from simple to write because complex systems require mocking, which can become extremely involved [9].

Although generally fast to execute and easy to understand, unit tests can also be brittle; small changes in implementation details can cause unit tests to break (becoming inapplicable) or to fail (due to unimportant internal behavioural changes for which they should pass). Unit tests that break lead to additional maintenance costs. Unit tests that inappropriately fail impede developer velocity.

Automated unit test generation also faces the familiar oracle problem [10]; if we could *automatically* determine the correct output for a given input, then we would not need the system under test in the first place. Fortunately, for the important subproblem of regression testing, there is a readily available oracle: the previous version of the system under test.

For the regression testing scenario, the oracle problem is transformed into two (generally more tractable) problems of:

1) Determining when a behavioural change results from the code change under test.
2) Deciding whether the changed behaviour is sufficiently different to warrant a signal to the engineer.

In large complex nondeterministic systems, small behavioural changes may be insignificant. Reporting these insignificant behavioural changes (as warnings or bugs) to developers will quickly ensure that the test tool becomes regarded as a source of false positive nuisance. Technologies that significantly impede developer velocity are typically discarded [6]. This has important implications for the research community. In general, evaluations of software testing research prototypes do not consider the impact of test signals on developer velocity.

In the specific case of regression testing, more research is required on equivalence classes. Techniques for constructing such equivalence classes would help determine suitable oracles for regression testing. Equivalence-based oracles can be used to balance the need to identify the true positives (high recall), while tensioning this requirement against the need to avoid unnecessarily impeding developer velocity with false negatives (high precision).

The outlook for automated unit test generation is optimistic, especially for regression testing. The research challenges are well understood, and widely studied, and the solution technologies have been demonstrated to be deployable. We can expect rapid uptake of automated test generation as a result. This uptake will transform the software testing landscape. Suppose instead of requiring developers to write unit tests, we had an entirely automated and scalable unit test generation platform, including the generation of suitable test oracles, and the generation of appropriate mocks. In this world, the remaining problem of brittleness would become much reduced: automated tests are cheap to generate and can be discarded and replaced with newly generated tests, as needed.

Recently, a number of commercial unit test generation frameworks have appeared, offering various solutions to tackle the automatic unit test generation problem. A brief search revealed more than ten such tools currently offered to market. More scientific research is needed to empirically evaluate the strengths and weaknesses of these different approaches and tools. This is an opportunity for the testing research community. Clearly it would be unrealistic to expect tool providers to conduct and report on unbiased scientific evaluations. On the other hand, test tool consumers typically lack the resources and training for a full empirical scientific evaluation. The scientific community is ideally placed to fill this knowledge gap.

There is also a need for more research on the most challenging aspects of test generation, including automatically determining whether to mock an object [11], how best to construct suitable mocks [12], [13], and how to determine the test oracles [14]–[16]. More work is required to develop, evaluate, and deploy these techniques on large scale production systems.

Unit tests also need to be realistic. In particular, actual parameters passed to the method under test need to be reflective of those that are witnessed in production. End to end tests can be used as a source of such realistic method parameters (See section II-D). We also need techniques for constructing realistic values for complex data types, without drawing in large quantities of boiler plate code (in the form of test builders) nor relying on arbitrary mocking (which may become fragile when the code changes).

C. Integration Testing

In theory, a unit test checks the specific functionality of the unit under test, independent of the rest of the system. This typically requires a high degree of mocking. As soon as we fail to mock a single external call from the unit under test, we have made the transition from unit testing towards integration testing.

The boundary between unit and integration testing is blurred. It is usually unrealistic to mock every call. As a result, many tests are regarded by engineers as unit tests when they are, strictly speaking, truly integration tests. Typically, a test of two or more components is regarded as a unit test when:

1) The components tested are sufficiently small.
2) The test oracle focuses on the behaviour of a single component, rather than interactions between two or more.

Determining the component granularity at which an integration-based approach to testing is required is, itself, an interesting and currently less-well-studied research problem.

D. Balancing e2e testing with more local testing

End to end (e2e) testing typically resides at the top of the test pyramid; salient to overall system behaviour, but expensive to execute. Despite their cost, e2e tests remain an essential part of the practical industrial test landscape. They are the ultimate

arbiter of the behaviour of the system under test. More research is needed on the engineering trade off between test efficiency and effectiveness.

In particular, research techniques that offer the test effectiveness of e2e testing with the cost of unit testing would be immediately actionable and highly impactful. Test carving [17] is one important idea that may help bridge this gap between e2e and unit level testing. More work is needed on the use of system level testing techniques, such as Sapienz [18], as a source of realistic, localised observations, from which unit and integration tests can be carved.

E. Practical mutation testing

Mutation testing [19], [20] has been studied for over five decades [21]. Recent results have demonstrated its superiority as a coverage criterion compared to traditional structural coverage goals, such as branch and statement coverage [22]. There have also been recent initial deployments of mutation testing in industry [23], [24]. These industrial deployments have demonstrated that the computational cost of mutation testing can be tamed [23]. Furthermore, developer surveys have indicated that engineers would be willing to adopt mutation testing if it could be made more practical [24]. In this section, we outline three research challenges which, when fully tackled, will make mutation testing highly practical, and potentially impactful in industry.

Change relevance: Traditionally, the research literature has considered the mutation generation problem as one in which mutants have been constructed *en masse*, for a single code base [19], [20]. This one-time deployment of mutation testing is poorly suited to modern continuous integration. Few authors have considered the generation of mutants incrementally, at the time of maximal relevance to engineers, and guided by the changes the engineers propose to land into the code base.

We need more work on 'Delta' forms of mutation [25]. Delta mutation is highly change aware. Delta mutants are constructed for a specific software change in order to drive, both the assessment of test effectiveness for covering the change, and for guiding test generation to better reveal faults introduced by the change.

Consistent test efficiency assessment: The most widely studied application of mutation testing concerns its ability to assess the fault-revealing power of existing test suites [19], [20]. Traditionally, the deployment model has been regarded as, once again, mutant construction and assessment *en masse*.

This '*en masse*' approach can assess the current effectiveness of a test suite, but is poorly suited to measuring relative improvements in test effectiveness over time. As the code base changes, the originally constructed *en masse* mutant suite becomes increasingly inapplicable. Therefore, improvements in test effectiveness cannot be *consistently* compared against previous iterations using the *en masse* mutant assessment approach.

One approach to tackle this mutant consistency problem is to focus on long-standing mutants [26]. A long-standing mutant is one that is relatively unaffected by code changes. There may be other approaches which can reduce, or remove, the detrimental effects of the mutant consistency problem. More work is required to develop mutation approaches that consistently assess relative improvements in test effectiveness in the presence of high degrees of test suite and code churn.

Mutation based test generation: Although test suite assessment has been the most widely-studied application of mutation testing, the most impactful is likely to be mutation-guided test generation. Since we know that strong mutation is one of the most powerful coverage criteria in terms of fault revelation [22], it makes sense to use mutants to guide the automated generation of tests. Mutation-guided test generation is a relatively less explored topic [27]–[29] that needs more work. In particular, generating test cases from particularly pernicious, otherwise hard-to-detect faults, such as the subtle faults generated from higher order mutants [30] would be particularly impactful.

F. Test Flakiness

Test flakiness is one of the most pressing problems for industrial software testing. It reduces test effectiveness and increases the cost of testing [6]. As systems become more complex, and their interactions and underlying technologies less deterministic, the flakiness problem is set to increase. More research is needed on techniques to identify and reduce flakiness [18], [31], and also to reformulate testing techniques to tolerate flakiness, while still giving useful signal [6].

More work is also required to understand the impact of test flakiness on the many downstream applications of software testing. These may be differentially affected and/or particular forms of flakiness may be more or less pernicious for different use cases. Laboratory controlled environments for flakiness research will help the research community to investigate this differential test flakiness impact [31].

G. Test effectiveness

Software testing consumes considerable effort in both human and machine time. Indeed, testing is often regarded as consuming as much as half of the overall development cost. Therefore, it is important to be able to assess the return on investment. When testing catches a fault before it reaches production, there is no ground truth evaluation of the exact impact of the fault, were it not to have been trapped by testing. However, in order to measure test effectiveness and account for the return on investment of test effort, techniques need to be found to approximate this ground truth.

In order to approximate, we need ways of evaluating the counterfactual scenario: what would have happened in production were this test not to have been caught sooner? Furthermore, when shifting left to catch bugs earlier in the development life-cycle, we need ways to measure the counterfactual scenario: what would have been the cost of fixing this bug had it been found later in the development life-cycle? This requires research on simulation of the impact of faults, fault severity, and the correlation of these surrogate metrics with production observations.

III. Automated Repair and Improvement

Since the inception of programming itself, software improvement has been a topic of interest. Ada Lovelace was the first to highlight the importance of software optimisation, writing the following paragraph, which still resonates today:

> "One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation." Extract from Ada Lovelace's 'Note D' to her translation of Menabrae's manuscript in the Analytical Engine [32].

The past decade has witnessed a considerable body of research developing the idea that, not just programs, but also large-scale software systems, can be automatically repaired and improved. Search Based Software Engineering [33] techniques have been particularly widely studied to tackle both problems. A great deal of work on automated repair has focused on functional correctness and reliability, typically caused by logic bugs [34]–[44]. The results are promising.

However, functional correctness and reliability are only two of many concerns, important though they are. In industrial settings, other non-functional criteria are also highly important, including memory consumption, power consumption, code footprint size, scroll performance[1], image rendering speed/quality, server to client latency, and CPU cycle consumption. Initial results from synthesis [45] and genetic improvement [46] are highly encouraging, but more work is required to extend, evaluate and deploy automated techniques for repair with respect to these non-functional system properties.

A. Tackling the Build Time Problem

Many approaches to automating software improvement, such as automated repair [47] and genetic improvement [48] require repeated experimental executions of candidate versions of the system under improvement. For example, search based approaches to repair and improvement may involve many tens or hundreds of thousands of system executions in order to compute the fitness values that guide the optimisation process.

This need for repeated execution raises the issue of the Build Time Problem [49]: software build times for large systems, consisting of tens to hundreds of millions of lines of code can run into minutes or even hours [2], [50], [51]. In order for search based optimisation techniques to be applied, the research community needs to find better ways to compute fitness without requiring overall system rebuild. Approaches using machine learning and simulation are possible candidate solutions. More research is required on these and other approaches if automated repair and improvement is to become more scalable and impactful.

[1]Scroll performance refers to the 'smoothness' with which the user interface responds to scroll requests. If the scroll behaviour is choppy (for example, due to dropped frames or slow user interface response times), this creates a poor user experience.

B. Effective Surrogates for Production Observations

The most realistic signal from testing comes from A/B testing [52], [53], because this executes candidate system improvements in production. However, it is clearly unrealistic to test every change using A/B testing because it would overburden production systems. In an ideal world, it would be possible to experiment, at system level, with large numbers of candidate changes using an efficient and safe surrogate for likely production performance. Such a highly scalable A/B testing surrogate would facilitate greater automation and exploration.

Research is needed on surrogate approaches that have high correlation to production observations. These highly correlated surrogates can be used to scale light touch testing that shares the advantages of A/B testing while reducing its cost. Simulation based testing approaches [54] are a natural avenue to follow here, but more research is required to understand the engineering trade-offs between simulation fidelity and computational cost.

C. Constructing Actionable Software Measurement Hierarchies

When tackling the improvement of complex deployed software systems, whether by human ingenuity or using automated techniques, there is the problem of how to combine the various software measurements involved. There is a large body of literature on software measurement (often called software 'metrics'). This body of literature includes many surveys covering different approaches to measurement and different use cases for measurement [55]–[57]. Despite this large body of literature, there are relatively few results on how best to combine multiple different software measurements. In the remainder of this section we outline four pressing challenges for software measurement research: composition, evolution, verification and validation.

Measurement Composition: There are many different software engineering stakeholders, each of whom represents diverse competing (and sometimes conflicting) engineering requirements. For example, release engineers working to release an app may wish to optimise the app's memory footprint, while production engineers may naturally be more concerned with the response times experienced by the user. These two objectives are not necessarily always in direct conflict, but they each reside in a software engineering trade off space. Ideally, the composition of metrics will accurately capture and thereby assess the choices within this trade of space.

Even when focusing specifically on the user experience itself, there are a number of different pertinent metrics such as scroll performance, video quality, and client-server latency. We need software measurement frameworks that allow us to compose multiple different component software measurements in meaningful ways. Such compositional frameworks need to support several technical goals, including:

1) Hierarchical composition and decomposition of component metrics.

2) Meaningful comparison between sets of component metrics.
3) Determination of suitable weightings to be given to components that manage the potential conflict between different metric components.

Measurement Evolution: Over time, new operational characteristics emerge, and new product features require additional user-facing metrics. However, there has been little research on how to best evolve an overall suite of metrics. New component metrics need to be incorporated, while maintaining consistency with previous observations.

Measurement Validation: The study of software metrics was focused initially on internal product metrics that measured properties of the code itself [58]. This meant that initial work on measurement validation tended to focus on the validation problem of checking that observable internal code-based metrics were correlated with more elusive properties, such as software quality [59].

With the widespread adoption of A/B testing [60], a new area of operational metrics has grown up, in which the measurements taken concern properties of direct or indirect concern to users. Some properties that matter to users are difficult to assess automatically and directly from production execution, leading to the study of indirect operational metrics. For indirect operational user-facing metrics, there is also an important validation question:

> How do engineers ensure that the metrics on which they drive their goals correlate sufficiently strongly to the system attributes their users care about?

Even for direct measures of user experience, such as scroll performance, there is a validation question: do these performance metrics correspond to improvements in the user experience? Clearly, very poor scroll performance would have a negative impact on user experience, but relatively small regressions may not be noticed. Therefore, it is unclear what the exact correlation is without proper statistical investigation, raising additional measurement validation questions. More research is needed at the interface between data science and software engineering to define techniques to validate the relationship between operational performance metrics and end user experience.

Measurement Verification: Many of the metrics computed in previous research work have been relatively simple. That is, they have been applied at the program level for which verification has not become a significant challenge. As a result, previous work on software metrics has also tended to ignore the metric verification problem. However, many user-facing metrics cater for large deployed systems involving complex logging, sampling, and inference, all of which raise important verification challenges. The metric verification question is:

> How do engineers ensure that automated measurements collected from software execution, correctly capture the properties they seek to measure?

More research is needed on testing and verification specifically targeted at complex measurements, computed from sampled production systems' logs.

Verification and validation of dynamically collected automated software measurement is simply the counterpart of the two key verification and validation concerns for software testing more generally. The twin problems can be thus summarised in the language of software testing as follows:

measurement validation: are we measuring the right thing?
measurement verification: are we measuring the thing right?

IV. Automated Transplantation and Pattern-Based Refactoring

Approximately one third of software development activity is spent on changes that should, in theory, affect neither the functional nor the non-functional characteristics of the system undergoing the change. These changes are typically performed to enhance ongoing maintenance, to pay off technical debt, or to update the underlying technologies used. This specific class of activities is important from a software testing research perspective because of the Oracle Problem. In these more structural change scenarios, the Oracle Problem reduces to the problem of regression testing, for which there is a readily available oracle. In the section, we outline open challenges for automated refactoring and transplantation.

A. Refactoring

There has been a great deal of previous research on software refactoring [61], which has introduced automated techniques that apply well-known low-level refactoring operations [62], [63]. For simple code level refactoring techniques [64], there is an underlying assumption that there is no test obligation to be discharged; the refactored code should be correct by construction.

More work is needed on generic frameworks for refactoring at higher levels of abstraction. Ideally, we need automated support for large-scale architectural changes to systems. For example, migrating from one API to another, or implementing a new design pattern. At this architectural level of abstraction, it can no longer be assumed that there is no test obligation to be discharged.

More research is required on testing techniques that are specifically tailored to the change in hand. Prototype test generation research tools typically require a great deal of engineering effort to deploy in practice. This is often a barrier to impact. By focusing on the specifics of such classes of architectural refactoring task, the research community has the potential for significant impact on industrial problems.

There is a balance to be struck between research and practice. If the research problems tackled are too general then there is a high barrier to uptake. At the other end of the spectrum, researchers would clearly prefer not to tackle problems so specific that they apply only to a narrow industrial subsector. Current research on automated test generation has tended to favour the more general end of this spectrum:

prototype research tools typically make *no* assumptions about the nature of the change under test.

By focusing on specific classes of architectural change, researchers would not be over constrained. The examples mentioned here (such as API migration implementation of new design patterns) are problems faced by *all* software practitioners. They also offer exciting opportunities for research. For example, knowing that the change affects a specific API gives a natural characterisation of the software testing search space: the function signatures of the API. Focusing on specific change scenarios such as API migration also simplifies the improvement opportunity space [65].

B. Transplantation

Automated software transplantation [66]–[68] is an important special case of refactoring: a feature is transplanted from one system (the donor) to another (the host). As with refactoring more generally, there is a test obligation: to check that no regression has occurred. In the case of transplantation, there is an additional test obligation: to check that the new feature operates as expected in the host. This additional test obligation can also be supported by automated test data generation [66].

Automated transplantation offers great promise for software productivity. Building small features involves engineers in very low level code concerns. It is a highly time-consuming activity. It involves many tedious details that are irrelevant to the overall architectural goals that the engineer has in mind.

Operating at the feature level, engineers could achieve an order of magnitude increase in their productivity. Automated feature transplantation would also free engineers up from these tedious low-level details to consider higher level architectural concerns. Transplantation reflects a natural balance between human and machine: the machine should be concerned with tedious specific details, while the engineer has the domain knowledge and linkage to business and customer requirements required to facilitate overall system design choices.

Although transplant testing is more challenging than pure regression testing, as with other refactoring problems, there is a wealth of additional information available. Transplantation is not a general problem of test generation for new features. Rather, it constitutes a specific case, where the feature has been transplanted from an existing donor system. Feature execution, *in situ*, in the donor system can thus be used as a test oracle. Since oracle automation is the last remaining barrier to fully automated test generation, software transplantation is fully automatable.

V. Incorporating Artificial Intelligence

Programmers have long been interested in the potential of artificial intelligence, discussing AI's potential (or otherwise) since the 19th century [32]. The past five decades have witnessed and increasing intensity of work on AI-based optimisation techniques for improving software testing [69]–[71] and, since at least 2001 [33], software engineering more generally. Over the past three decades, software engineers have also become increasingly excited by the potential of machine learning to tackle automation challenges in software engineering [72]–[74].

Recent advances in generative AI have the potential to perfectly complement existing research directions in Software Engineering automation. In particular, automated test data generation is a natural complement to generative AI. While the generative AI approach may have powerful capabilities to generate highly human-readable, domain and context aware solutions, its tendency to hallucinate renders it relatively unreliable on its own. However, automated test data generation has the ability to add the necessary assurances, and to weed out such hallucinatory aspects of AI-based solutions.

Generative AI models have been recently shown to exhibit powerful emergent behaviours [75], which has far-reaching implications. This makes their behaviour not only powerful, but also inherently poorly understood. In applications where there is no ground truth, such as general queries concerning arbitrary facts about reality, the models' emergent behaviour may be problematic since it cannot be cross checked against a ground truth. However, for software engineering tasks, such as testing and code improvement, we have a very reliable ground truth: the execution of the proposed test or improved version of the code.

Finding a suitable query for a generative AI model is also a non trivial undertaking due to the model's emergent behaviour. The query engineering problem is to find the optimal query that will tend to obtain the most effective response. Such query engineering is a topic that is likely to witness rapidly growing interest. For software engineering applications, Search Based Software Engineering (SBSE) [33] is well placed to tackle the search for optimal (or near optimal) queries. SBSE is well-suited to query engineering for software engineering because it can use fitness, based on concrete ground truth, to search the space of queries. SBSE is also known to be good at situations characterised by noisy, partial and contradictory fitness signal [76], all of which are prevalent for the query engineering optimisation problem.

A. Augmenting existing test suites

Generative AI is likely to find impactful applications in augmenting and extending existing test suites. For example, given an existing set of test cases, it could be used to generate additional test cases that cover corner cases or other likely fault models, guided by a suitable fault history. Generative AI may be able to additionally maintain the coding style of the existing given test suite. Readability and maintainability have long been concerns for automated test generation and improvement [77]. AI techniques that mimic existing coding styles will help facilitate this onward human maintenance and may outperform existing purely rule-based approaches.

AI may also be useful in augmenting existing test cases with additional oracle information. For example, adding `assert` statements to unit tests that capture pertinent cases that the developer may have overlooked. In both of these testing applications, Large Language Models (LLMs), trained on suitable code corpuses, are likely to be effective. More research is

needed to explore these possibilities and, in particular, to better understand how they should inter-operate with existing rule-based approaches to test generation, oracle improvement, and coverage maximisation.

Coverage reported from the first studies of unit test generation are highly encouraging. For example JavaScript unit testing based on the Codex LLM was able to achieve median statement coverage of 68.2% on non trivial systems [78]. Early results from hybrid approaches to test generation, incorporating traditional techniques with Machine learning have also shown the promise of this agenda. For example, the ATLAS approach [79] is able to add meaningful assertions to existing test cases thereby tackling the oracle problem which has, hitherto, proved a barrier to fully automated test data generation. CODAMOSA [80] has been used to unblock Search Based Software Test data generation, thereby elevating coverage, also using a Large Language Model.

B. Replacing execution with prediction

Testing can always be more efficient, if we can find ways to squeeze the attainment of more valuable test signal into the same amount of available system resources. Much research has focused on test case selection and prioritization to tackle these challenges [81]. Predictive modelling (of likely test outcomes, including those related to performance) provides further opportunities to increase the amount of signal that can be attained from given resources available for testing.

C. Suggesting code repairs and remediations

Recent research [82] indicated the effectiveness of Generative AI techniques, such as ChatGPT, based on LLMs, for suggesting code repairs. These techniques are likely to prove important for tackling existing software engineering automation problems such as automated program repair. However, they also open up new potential deployment routes for repair-based technologies.

In particular, the interactive discursive nature of LLMs lends itself to supporting engineers in the full software engineering process, from the detection of a bug through to its ultimate repair. Such models might initially recommend a remediation step that will tackle the immediate consequences of a bug. This remediation immediacy is increasingly important in DevOps scenarios where downtime is expensive [83].

After helping engineers to find an initial remediation, the next step is to attempt to root cause the problem. LLMs also have a role to play here. They will likely perform better when provided with data from more traditional root cause identification techniques, such as Spectrum-Based Fault Localisation [84]. This root causing activity currently consumes a great deal of engineers' time. It can also be a deeply frustrating experience for engineers [85].

One concern for this research agenda will be the potential for engineers to end up relying on unsound AI-suggested solutions. LLMs are prone to hallucination [86] just like other sources of intelligence, such as human intelligence [87]. Fortunately, the interactive and discursive nature of the investigation also facilitates iterative verification. That is, the engineer can check AI suggestions using, for example, targeted automated test generation at each stage of the process. We need more research to define techniques that combine the exploratory power of AI, with the comprehensive assurance of traditional test generation.

D. Suggesting performance improvements

While researchers have started to consider Generative AI's potential for automated repair [82], more work is needed to understand its potential application to identify candidate performance improvements. Here, the potential of a combination with traditional techniques is exciting. With repair, the engineer is left with the oracle problem. They cannot be sure that the candidate repair has fixed the problem, and may need to generate additional test cases to gain such assurance.

However, when seeking performance optimisations, the engineer can rely on regression testing in order to protect against functional regressions. Given the wealth of available data from previous executions, the regression testing problem may prove more amenable than checking functional correctness. Furthermore, the engineer seeking to use AI for performance optimisation also has a natural way to measure the improvement: the set of metrics against which they seek to measure improvement, whether by human hand or by machine.

VI. Overview of Lessons From Deployment

Achieving impact from software testing research is not simply a case of transplanting prototype tools into production environments. The point in the life-cycle at which the technology interposes, and the manner in which it communicates its signal, are highly important. We have witnessed situations where a technology deployed too late in the software development process had close to zero uptake from software engineers, yet the exact same technology, deployed earlier (at the time when its signals were maximally actionable) had profound impact [6].

In this section we draw out lessons learnt, sometimes through bitter experience, from the deployment of software testing and improvement technology into scaled-out industrial development environments. While our experience is naturally limited to a single company, we focus on those lessons which our colleagues from other companies tell us apply equally well in their software engineering experience.

Space constraints permit only an overview in this section, so we provide references from which the reader can obtain more details in each case.

Deploy at the point of maximum relevance: Software testing research tends to evaluate number of bugs found for a given corpus of software. While this is perfect for scientific evaluation, it represents a poor deployment strategy. The best deployment of software testing research ensures that the signal provided by the test tool is available at exactly the time at which the developer is working on the change. Therefore, in order to achieve impact, it is essential to deploy into the

continuous integration system at a time of maximum relevance [6].

Determine the role of the human in the automation loop: It is important to consider where the developer should interpose in the automated improvement loop. For example, developers typically want to take and own the code change to be applied, even when an automated recommendation is 100% correct [43]. Ultimately, human engineers will be responsible for the changes that land into production. Maximum automation is achieved when human decision time is minimised. Maximum automation therefore requires careful thought about how and when signals and recommendations from automation are presented to the engineer.

Automated fix detection: It is important to be able to measure the impact of signals provided by automated testing improvement technologies. This impact assessment also needs to be automated, to support continuous evaluation and improvement of the deployment. Typically, a bug can only truly be said to be fixed when there is certainty that there remains no production manifestation. Automatically determining this at scale is a surprisingly challenging problem. For example, automated fix detection from production observations of crash hashes remains an open research problem [1].

Piggybacking increases testing research deployment velocity: Fully automated software testing is most readily achieved in regression testing scenarios where the oracle problem is relatively automatable. Fortunately, regression testing deployment also offers many opportunities for other software testing research to become deployed. For example, regression testing and metamorphic testing [88], [89] are so closely related that a single system can be deployed to achieve both [2]. Similarly, fuzzing technology [90] has achieved widespread deployment, and is technically almost indistinguishable from Search Based Software Testing techniques [91], suggesting hybridisation opportunities [92]. In this way, those hitherto less-well-deployed software testing research techniques can piggyback on existing high impact deployments.

The build-time problem must be tackled for improvement deployment: The build-time problem [49] for automated repair and genetic improvement can be a barrier to deployment of automated improvement techniques that require repeated execution of the system to be improved. We were able to ameliorate using simulation based techniques [54], but other approaches may also remove this barrier, such as predictive modelling and deep parameter optimisation [93].

Testability transformation increases applicability: Redesigning an entire software testing platform to cater for the specific features of a system under test can be prohibitively expensive. Sometimes the root cause for non-deployment of software testing techniques lies simply in irritating detailed incompatibilities. There are often implicit assumptions made by an otherwise highly deployable tool. In these situations, it can be easier to apply Testability Transformation [94]–[99] to the system under test to render it amenable to previously inapplicable technologies. Transforming a system, specifically with testability in mind, may also have additional benefits by supporting human design of test cases.

REFERENCES

[1] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "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, September 8th-10th 2018, pp. 3–45, springer LNCS 11036.

[2] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer, S. Sapora, and J. 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.

[3] N. Tillmann, J. de Halleux, and T. Xie, "Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger," in *29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 385–396.

[4] A. F. Donaldson, "Metamorphic testing of android graphics drivers," in *Proceedings of the 4th International Workshop on Metamorphic Testing, MET@ICSE 2019, Montreal, QC, Canada, May 26, 2019*, X. Xie, P.-L. Poon, and L. L. Pullum, Eds. IEEE / ACM, 2019, p. 1.

[5] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in 22^{nd} *International Symposium on Foundations of Software Engineering (FSE 2014)*, S.-C. Cheung, A. Orso, and M.-A. Storey, Eds. Hong Kong, China: ACM, November 16 - 22 2014, pp. 643–653.

[6] M. Harman and P. 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)*, Madrid, Spain, September 23rd-24th 2018, pp. 1–23.

[7] G. J. Myers, *The Art of Software Testing*. New York: Wiley - Interscience, 1979.

[8] R. Osherove, *The Art of Unit Testing: with examples in C*. Simon and Schuster, 2013.

[9] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[11] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 402–412.

[12] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, "Automock: automated synthesis of a mock environment for test case generation," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.

[13] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 365–368.

[14] A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2021, pp. 30–39.

[15] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *International Symposium on Software Testing and Analysis (ISSTA 2010)*. Trento, Italy: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[16] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1708–1728, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2934409

[17] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009.

[18] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: models, tools, and controlling flakiness," in 35^{th} *International Conference on Software Engineering (ICSE 2013)*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. San Francisco, CA, USA: IEEE Computer Society, May 18-26 2013, pp. 1479–1480.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 – 678, September–October 2011.

[20] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practical programmer," *IEEE Computer*, vol. 11, pp. 31–41, 1978.

[22] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. 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*, 2017, pp. 597–608.

[23] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 163–171.

[24] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. 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)*. IEEE, 2021, pp. 268–277.

[25] W. Ma, T. T. Chekam, M. Papadakis, and M. Harman, "Mudelta: Delta-oriented mutation testing at commit time," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 897–909. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00086

[26] M. Ojdanic, M. Papadakis, and M. Harman, "Keeping mutation test suites consistent and relevant with long-standing mutants," 2022.

[27] M. Harman, Y. Jia, and W. 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)*. New York, NY, USA: ACM, September 5th - 9th 2011, pp. 212–222. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025144

[28] M. Papadakis and N. Malevris, "Searching and generating test inputs for mutation testing," *SpringerPlus*, vol. 2, no. 1, pp. 1–12, 2013.

[29] F. Carlos, M. Papadakis, V. Durelli, and E. M. Delamaro, "Test data generation techniques for mutation testing: A systematic mapping," in *Workshop on Experimental Software Engineering (ESELAW'14)*, 2014.

[30] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in 5^{th} *International Workshop on Mutation Analysis (Mutation 2010)*, Paris, France, April 2010.

[31] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman, "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, 2022, pp. 982–994. [Online]. Available: https://doi.org/10.1145/3510003.3510194

[32] A. A. Lovelace, "Sketch of the analytical engine invented by Charles Babbage by L. F. Menabrea of Turin, officer of the military engineers, with notes by the translator," 1843, translation with notes on article in italian in Bibliothèque Universelle de Genève, October, 1842, Number 82.

[33] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[34] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[35] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 448–458.

[36] S. Mechtaev, M. D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in 40^{th} *ACM/IEEE International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, May 27 - 3 June 2018.

[37] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *35th International Conference on Software Engineering (ICSE 2013)*, B. H. C. Cheng and K. Pohl, Eds. San Francisco, USA: IEEE, May 18-26 2013, pp. 772–781.

[38] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015, pp. 166–178.

[39] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in 40^{th} *International Conference on Software Engineering (ICSE 2018)*, 2018.

[40] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[41] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus, "NPEFix: Automatic runtime repair of null pointer exceptions in java," 2015, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01251960

[42] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating automatic program repair capabilities to repair API misuses," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2658–2679, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3067156

[43] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*, Montreal, Canada, 2019.

[44] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[45] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 317–330, Jan. 2011.

[46] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation (TEVC)*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[47] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[48] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, Jun. 2018.

[49] M. Harman, "Scaling genetic improvement and automated program repair (keynote paper)," in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3524459.3527353

[50] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 770–781.

[51] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 197–207.

[52] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 959–967.

[53] R. Kohavi and R. Longbotham, "Online controlled experiments and A/B testing," *Encyclopedia of machine learning and data mining*, vol. 7, no. 8, pp. 922–929, 2017.

[54] J. Ahlgren, K. Bojarczuk, S. Drossopoulou, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, S. Lucas, E. Meijer, S. Omohundro, R. Rojas, S. Sapora, J. M. Zhang, and N. 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, keynote talk given jointly by Inna Dvortsova and Mark Harman.

[55] F. Riguzzi, "A survey of software metrics," *Università degli Studi di Bologna*, 1996.

[56] J. K. Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of computer science and technology*, vol. 25, pp. 1016–1029, 2010.

[57] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[58] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[59] N. F. Schneidewind, "Methodology for validating software metrics," *IEEE Transactions on software engineering*, vol. 18, no. 5, pp. 410–422, 1992.

[60] D. Siroker and P. Koomen, *A/B testing: The most powerful way to turn clicks into customers*. John Wiley & Sons, 2013.

[61] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[62] M. O'Keeffe and M. Ó Cinnédie, "Search-based refactoring: an empirical study," *Journal of Software Maintenance*, vol. 20, no. 5, pp. 345–364, 2008.

[63] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Genetic and evolutionary computation conference (GECCO 2006)*, vol. 2. Seattle, Washington, USA: ACM Press, 8-12 Jul. 2006, pp. 1909–1916. [Online]. Available: http://www.cs.bham.ac.uk/ wbl/biblio/gecco2006/docs/p1909.pdf

[64] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[65] W. B. Langdon, D. R. White, M. Harman, Y. Jia, and J. Petke, "API-constrained genetic improvement," in *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, 2016, pp. 224–230.

[66] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015, pp. 257–269.

[67] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 665–676.

[68] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated patch transplantation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–36, 2020.

[69] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[70] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[71] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing (keynote paper)," in 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015.

[72] S. C. Bailin, R. H. Gattis, and W. Truszkowski, "A learning-based software engineering environment," in *Proceedings of the 6th International Conference on Knowledge-Based Software Engineering*, 1991, pp. 198–206.

[73] T. Menzies, "Practical machine learning for software engineering and knowledge engineering," in *Handbook of Software Engineering and Knowledge Engineering*. World-Scientific, December 2001, available from http://menzies.us/pdf/00ml.pdf.

[74] M. Harman, "The role of artificial intelligence in software engineering (keynote paper)," in 1^{st} *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2012)*, Zurich, Switzerland, 2012.

[75] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[76] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–11:61, November 2012.

[77] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *International Symposium on Software Testing and Analysis (ISSTA'12)*, Minneapolis, Minnesota, USA, July 2012, pp. 177–187.

[78] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," *arXiv preprint arXiv:2302.06527*, 2023.

[79] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[80] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023, to appear.

[81] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[82] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT," *arXiv preprint arXiv:2301.08653*, 2023.

[83] S. Elliot, "Devops and the cost of downtime: Fortune 1000 best practice metrics quantified," *International Data Corporation (IDC)*, 2014.

[84] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng*, vol. 42, no. 8, pp. 707–740, 2016.

[85] A. Zeller, "Automated debugging: Are we close?" *Computer*, vol. 34, no. 11, pp. 26–31, 2001.

[86] N. Dziri, S. Milton, M. Yu, O. Zaiane, and S. Reddy, "On the origin of hallucinations in conversational models: Is it the datasets or the models?" *arXiv preprint arXiv:2204.07931*, 2022.

[87] H. Merckelbach and V. van de Ven, "Another white christmas: fantasy proneness and reports of 'hallucinatory experiences' in undergraduate students," *Journal of Behavior Therapy and Experimental Psychiatry*, vol. 32, no. 3, pp. 137–144, 2001.

[88] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[89] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: a review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 4:1–4:27, January 2018.

[90] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *CoRR*, vol. abs/1812.00140, 2018. [Online]. Available: http://arxiv.org/abs/1812.00140

[91] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[92] N. Alshahwan, A. Ciancone, M. Harman, Y. Jia, K. Mao, A. Marginean, A. Mols, H. Peleg, F. Sarro, and I. Zorin, "Some challenges for software testing research (keynote paper)," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019), Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 1–3.

[93] F. Wu, M. Harman, Y. Jia, J. Krinke, and W. Weimer, "Deep parameter optimisation," in *Genetic and evolutionary computation conference (GECCO 2015)*, Madrid, Spain, July 2015, pp. 1375–1382.

[94] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, Jan. 2004.

[95] M. Harman, "We need a formal semantics for testability transformation (keynote paper)," in 16^{th} *International Conference on Software Engineering and Formal Methods (SEFM 2018)*, Toulouse, France, 2018, pp. 3–17.

[96] D. Gong and X. Yao, "Testability transformation based on equivalence of target statements," *Neural Computing and Applications*, vol. 21, no. 8, pp. 1871–1882, 2012.

[97] A. Kalaji, R. M. Hierons, and S. Swift, "A testability transformation approach for state-based programs," in ^1st *International Symposium on Search Based Software Engineering (SSBSE 2009)*. Windsor, UK: IEEE, May 2009, pp. 85–88.

[98] Y. Li and G. Fraser, "Bytecode testability transformation," in *Search Based Software Engineering - Third International Symposium (SSBSE 2011)*, ser. Lecture Notes in Computer Science, vol. 6956. Springer, 2011, pp. 237–251.

[99] P. McMinn, "Search-based failure discovery using testability transformations to generate pseudo-oracles," in *Genetic and Evolutionary Computation Conference (GECCO 2009)*, F. Rothlauf, Ed. Montreal, Québec, Canada: ACM, 2009, pp. 1689–1696.