

N-Version Assessment and Enhancement of Generative AI

Marcus Kessel, *University of Mannheim, Mannheim, 68159, Germany*

Colin Atkinson, *University of Mannheim, Mannheim, 68159, Germany*

Abstract—Generative AI (GAI) holds great potential to improve software engineering productivity, but its untrustworthy outputs, particularly in code synthesis, pose significant challenges. The need for extensive verification and validation (V&V) of GAI-generated artifacts may undermine the potential productivity gains. This paper proposes a way of mitigating these risks by exploiting GAI's ability to generate multiple versions of code and tests to facilitate comparative analysis across versions. Rather than relying on the quality of a single test or code module, this “differential GAI” (D-GAI) approach promotes more reliable quality evaluation through version diversity. We introduce the Large-Scale Software Observatorium (LASSO), a platform that supports D-GAI by executing and analyzing large sets of code versions and tests. We discuss how LASSO enables rigorous evaluation of GAI-generated artifacts and propose its application in both software development and GAI research.

By automatically generating software artifacts, including code, from small, natural language prompts, generative AI (GAI) has the potential to dramatically boost the productivity of software engineers. However, while the creativity of GAI systems is essentially boundless, the “quality” of the artifacts they generate is not, significantly limiting their trustworthiness in engineering domains where low quality is costly. Effectively judging, and if necessary enhancing, the quality of GAI-generated artifacts will therefore be critical to its success in practical software engineering (SE).

Software quality can be defined by various properties, but measuring some of the most critical quality attributes – specifically semantic (i.e., behavioral) properties – remains undecidable in most cases (cf. Rice's theorem¹). As a result, it is impossible, in general, to develop a tool that can automatically verify non-trivial semantic properties of arbitrary code modules, including their functional correctness relative to specifications or previous versions. This limitation is a key reason why modern software projects often allocate over 50% of their total effort to verification and validation (V&V) techniques such as testing.² Consequently, any potential productivity gains from using GAI could

be significantly diminished – or even negated – by the additional V&V work required to mitigate its inherent unreliability.

At first glance, it might seem feasible to address the V&V challenges posed by GAI by using it to automatically generate additional V&V artifacts, such as tests. However, these artifacts – like all GAI-generated outputs – are also of uncertain quality. This creates a dilemma: verifying code modules of uncertain quality with verification artifacts (e.g., tests) that are equally unreliable. The only practical way to break this cycle and achieve genuine confidence in the quality of GAI-generated code, without resorting to excessive manual effort, is to generate multiple, diverse versions of both code modules and tests³ and conduct extensive comparative analysis of their executions.

The concept of comparing multiple executions of similar versions of code modules has a long history across various branches of SE. It appears in approaches such as differential testing for software V&V,⁴ N-version programming for enhancing software reliability,⁵ mutation testing for assessing test quality, and the use of “cross-checking” oracles in automated test generation.⁶ While these methods differ in their objectives and the stages of a software artifact's lifecycle in which they are applied, they all share a common core: comparing the executions of multiple tests on multiple versions of code modules expected to deliver

specific functionality. This comparative approach is also valuable in experimentally evaluating tools used to create or modify these versions, including tools for code refactoring, program repair, code recommendation, and GAI for code generation.

The premise of this paper is that the future success of GAI in mainstream SE projects will be significantly enhanced by large-scale, “comparative” or “differential” testing of GAI-generated artifacts. However, current testing frameworks are not suited for this purpose, as they are designed to test individual code modules. Even existing differential testing tools offer limited user feedback on discrepancies. Moreover, these tools lack the decision-making capabilities and data-driven insights of code recommenders, which take into account users’ weighted preferences for functional and non-functional properties to select the best code modules or tests based on comprehensive criteria.⁷

In this paper, we explain how the “differential GAI” (D-GAI) approach to code and test synthesis could significantly accelerate GAI-driven SE in both development and research. To illustrate this, we introduce the Large-Scale Software Observatory (LASSO), a new software observation platform designed to support the D-GAI approach. LASSO provides domain-specific languages, data structures,⁸ and an execution “arena” that facilitates the execution of large numbers of tests across numerous code module versions. By analyzing the resulting observational data, functional and non-functional differences and patterns can be identified, leading to improved code quality and reliability – regardless of how the versions or tests were generated.

LASSO can also be used to conduct large-scale, reproducible studies on the effectiveness of GAI models, with the observational data serving as valuable training input to further enhance GAI systems. After presenting a D-GAI scenario based on a typical code generation task, we outline how a D-GAI engine could be realized using a platform like LASSO. Finally, we explore how D-GAI can improve the utility of GAI in practical SE projects while also providing a means to assess and enhance GAI systems themselves.

EXAMPLE D-GAI SCENARIO

As an illustration of a D-GAI scenario, suppose a team of software developers wants to use one of the latest GAI code generation models to help accelerate their project. Some of the most well-known code models currently available include GPT-3.5/-4 Turbo, Codex (GitHub CoPilot), CodeGen and InCoder (see⁹), which are all based on the Transformer-driven LLM technology popularized by ChatGPT. These can perform a variety of SE tasks, but one of the most

canonical is synthesizing code modules from natural language descriptions of the desired functionality. In the GAI community this is usually referred to as a code completion “problem”,¹⁰ but essentially provides a service called “code recommendation” in the traditional software reuse community.⁷

To get such a code generation model (i.e., code model for short) to recommend a code module, developers have to provide it with a “prompt” that describes the desired software functionality in some way. For example, the following prompt is for a method that calculates the greatest common divisor (GCD) of two integers. After identifying the signature of the method (here in Python syntax), the prompt provides a short natural language description of the desired functionality as well as two tests a correct implementation is expected to pass.

```
def gcd(a: int, b: int) -> int:
    """ Return a greatest common divisor
    of two integers a and b
    >>> gcd(3, 7)
    1
    >>> gcd(10, 15)
    5 """
    (body to be generated)
```

On receiving such a prompt, a code model returns a recommended implementation of the desired functionality. State-of-the-art code models are able to provide correct recommendations of this size a surprisingly high proportion of times. Recent experiments (e.g., MultiPL-E,¹⁰ EvalPlus⁹) and leaderboards¹ have achieved significant success rates on popular benchmarks, with some exceeding 80%. However, while these rates are high from a GAI research point of view, they are not particularly high for practical software development (cf.¹¹). Thus, in order to be able to actually use a recommended code module in a real project, developers would probably need to test it extensively.

In the D-GAI version of this scenario, developers would receive the same basic service, driven by the same prompt, but the “quality” of the recommended code module would usually be much higher. This is achieved by means of a special tool or platform (i.e., a D-GAI engine) that automatically performs the code recommendation request multiple times, possibly using multiple code models, to generate N versions of the desired code module, and then automatically compares them using multiple, automatically generated tests. The D-GAI engine then returns the best version based on

¹ <https://evalplus.github.io/leaderboard.html>

specific ranking criteria. Applying such an N-version approach has numerous important benefits —

- › *Results Aggregation* — the D-GAI engine is able to aggregate the results of many GAI code recommendation requests, from one or more code models. Even if one code model is used, this is advantageous because they are inherently non-deterministic in their behavior, and thus can give different results for the same prompt. When more code models are available, obviously the best results can be selected. Since it aggregates results, a D-GAI engine will always deliver a result that is at least as good as a single-use realization of the code recommendation service.
- › *Semantic Awareness* — A major benefit of the D-GAI approach is that by actually executing all multiple candidates (i.e., N versions), a result can be returned that genuinely passes the required tests. The users of the recommendation service can then have confidence that the returned result, if there is one, truly has the behavior demanded by the tests. This not only increases confidence in the results, but obviates the need for further testing with the prompt tests. Code models themselves cannot give a guarantee that the recommended code passes the prompt tests, because they are unaware of the true behavior of software code.¹² This idea corresponds to the notion of test-driven search in the software reuse community.¹³
- › *Observational Measurement* — Another major benefit of executing the candidates is that their comparison can include dynamic code metrics, such as performance and coverage, as well as static code metrics (e.g., measurements obtained by applying GQM¹⁴). Code models are unable to consider the former because they are unaware of the true execution properties of the code modules they generate. They also do not apply classic static metrics in the traditional sense, which a D-GAI engine could easily do.

Of course, the extra testing and analysis involved in D-GAI comes at the cost of significantly lower response times. However, considering the widespread adoption of test-driven development and continuous integration practices in modern SE, we envision GAI code recommendations being primarily used in an asynchronous, offline mode that will not disrupt developers' workflows. As such, D-GAI will likely become a natural extension of the offline capabilities offered by powerful continuous integration platforms, further streamlining

the development process. Moreover, even for direct use, with a powerful D-GAI engine, the code recommendation response times are reasonable, depending on the number of module versions explored. In a recent experiment,¹³ average response times for 1,000 code modules harvested from code repositories were ~4 minutes on modern hardware.

While employing code models may introduce costs due to additional token generation from multiple models or repeated iterations on a single model to obtain N versions of the code modules and tests, the D-GAI approach offers a rapid feedback loop reminiscent of agile methodologies. This helps developers to making informed, data-driven decisions, avoiding the pitfalls of blindly integrating generated code. By minimizing technical debt and potential risks, this approach can lead to significant long-term cost savings through reduced rework, debugging, and maintenance.

TOWARDS A D-GAI ENGINE

In this section, we delve deeper into the architecture of a D-GAI engine and outline key components necessary for its implementation. To facilitate the discussion, we employ our LASSO platform², which was designed specifically to support the N-version comparison layer required to realize D-GAI on top of standard code models. While LASSO is not exclusively focused on D-GAI, it provides a suitable foundation for implementing the D-GAI approach. As well as a powerful testing platform and large code repository, LASSO offers several dedicated languages and data structures for N-Version comparison.

Figure 1 illustrates how the GCD code recommendation scenario described earlier can be efficiently implemented using LASSO. The top left section of the diagram introduces LASSO's dedicated "sequence sheet" notation, used to store stimulus-response pairs for sequences of method invocations. A sequence sheet functions similarly to a method with a corresponding signature, but instead of using traditional code for its body, it uses a table format. Each row in this table represents an individual method invocation, while the columns represent different components: column *B* designates the called method, columns *C* to *E* hold the input parameters, and column *A* captures the output. Users can either represent their test components directly as sequence sheets or have them generated automatically from the traditional prompt format shown earlier.

²<https://softwareobservatorium.github.io/>

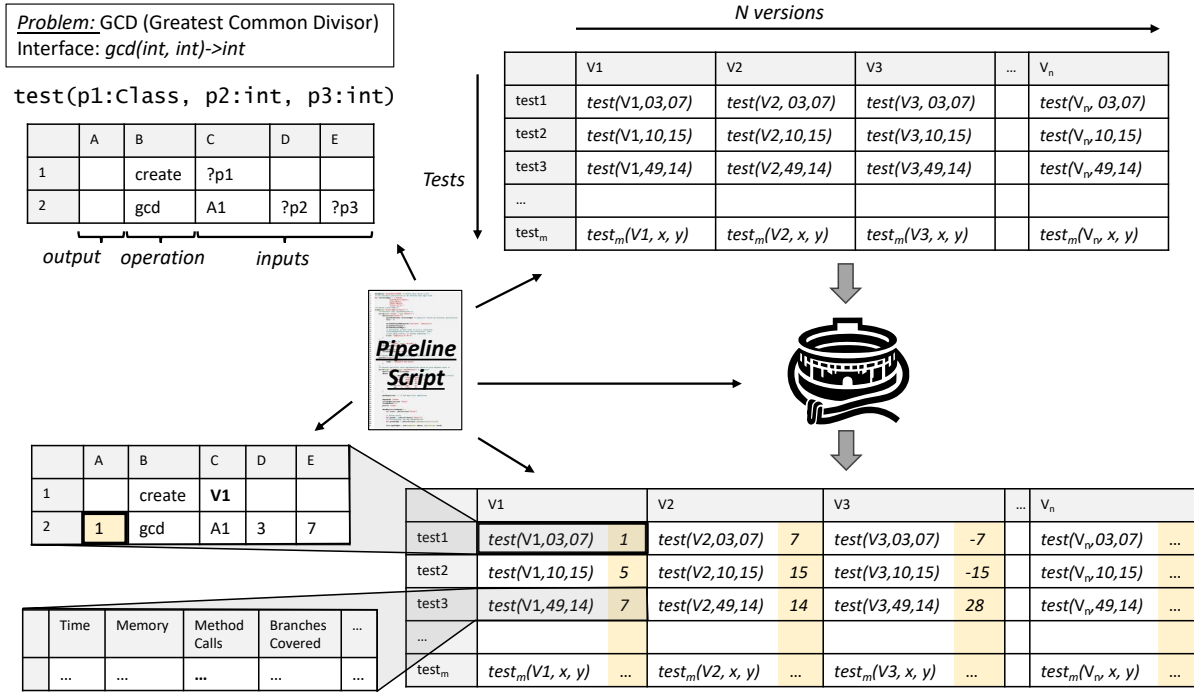


FIGURE 1: Towards a D-GAI Engine – Data Structures and Execution Arena for N-version Analysis

On receiving a code recommendation prompt, LASSO can generate a so-called stimulus matrix (SM) of the form shown on the top right-hand side of Figure 1 and populate it with multiple implementations of the desired functionality (as columns) and multiple tests of the desired functionality (as rows). The N versions can be obtained from a variety of sources, including from LASSO's own code corpus, but in the D-GAI scenario they would be generated by multiple code generation requests to one or more code models. If only one code model is available, N versions can be obtained either (a) by invoking it many times since the code synthesis behavior is non-deterministic, or (b) by varying the various model parameter choices like the temperature parameter to fine-tune the level of generation behavior. To populate the SM with tests, any tests provided in the prompt are converted into sequence sheets and invocations, as shown by the first two rows in the example. More tests can also be added automatically through code models using generated prompts, or by applying standard unit test generation tools like EvoSuite¹⁵ or Randoop.

Once the SM has been populated it can be input to LASSO's dedicated test engine, the so-called "arena", where all the tests are executed on all N versions of the desired functionality. To cope with the high volume of executions that can be involved, the arena is implemented as a parallel, distributed architecture

with load balancing and sandboxed (controllable and secure) code execution environments based on containerization. Once all the testing has been performed, the results are output as a stimulus response matrix (SRM), which has the same shape as the input SM, but contains all the runtime data observed during the execution process (i.e., outputs, execution time, methods called, branches covered etc.). As shown in the bottom right-hand side of Figure 1, the core advantage of using sequence sheets to define tests is that it allows SRMs to store the responses of the different module versions within the sequence sheets that provoked them. For example, the sequence sheet at the bottom left of the diagram shows that the invocation of GCD module version V1 with the values 3 and 7 (the first test in the prompt) returned the value 1. SRMs therefore provide a persistent, analyzable record of the behavior of each version in response to each test.

One other important LASSO feature represented in Figure 1 is the pipeline script used to drive the work flow steps involved in a D-GAI code recommendation request. This allows engineers to write new D-GAI services in a dedicated DSL.

Finally, the LASSO platform has a large underlying repository of executable software code, harvested from large open source code repositories, that can be accessed using classic code search technologies.¹³ Although the use of this repository is not essential

for D-GAI, including human-written modules and tests harvested from the repository can enhance diversity, and thus improve the quality and trustworthiness of the overall results.

D-GAI USE CASES & SERVICES

The *LASSO* data structures described above are not essential for building a D-GAI engine. However, using them offers several advantages. Firstly, treating tests as “first-class” citizens in the differential comparison process allows them to be assessed and enhanced using D-GAI alongside the *N* module versions. Secondly, storing and analyzing execution observations offline with analysis tools like Python/Pandas, enables researchers and practitioners to assess, benchmark, and enhance the underlying code models across multiple coding problems.

Figure 2(a) presents a comprehensive overview of the range of use cases and services that could be supported by a D-GAI engine of the kind outlined previously. It categorizes the services into three main tiers. The root-level distinction is between services aimed at assisting developers in building concrete software applications and those intended to aid researchers in evaluating or enhancing code models. Application Engineering services are further divided into Development-time and Runtime services. The latter focus on improving the quality of deployed applications during execution time as in classic *N*-version programming.⁵ Research services comprise GAI code model evaluation services for benchmarking and comparing different code models, as well as code model enhancement services for training and refining GAI models themselves. Within the Development-time application engineering category, subservices help developers to evaluate, create and enhance code and tests. In the remainder of this section, we will delve deeper into these leaf cases.

Code Recommendation

Code recommendation is the use case exemplified in the second section, and is one of the core use cases of D-GAI for software development. The basic idea resembles the test-driven search technology provided by some code search engines,¹³ and offers the same important benefit of improving the user's confidence in the correctness of the recommended module (or modules), since it has been verified to pass the tests supplied in the prompt. Code models themselves do not give a guarantee that the recommended code passes the prompt tests, because they are unaware of the true behavior of software code.¹² The basic difference is that whereas code search engines retrieve

human-written code modules from code repositories, a D-GAI code recommendation service selects the most suitable code module from a set of GAI-synthesized modules. Executing the candidates also allows the final ranking of the candidates to include both dynamic metrics and static metrics. Code models can enhance code recommendations services not only by synthesizing large numbers of alternative module versions, but also by synthesizing additional tests.

Differential Testing

Differential testing is a widely practiced testing technique in modern SE projects which aims to improve the quality of a particular code module under development (i.e., “base version”) by comparing its behavior to that of one or more alternative versions to identify discrepancies that may reveal faults or functional differences.⁴ It is most commonly applied in the form of regression testing where the alternative versions are older versions of the code module. This is because obtaining a diverse set of alternative versions of the base version (i.e., that are not simply clones) is a challenging task.

Test Quality Evaluation & Enhancement

Test quality evaluation is perhaps the simplest of the test-oriented D-GAI services in which the prompt includes a set of tests and a base code module of the desired functionality, and the output is one or more test quality metrics. Code models are used only to generate new versions of the base version, which play the same role as the mutations in mutation testing.² After executing all the tests on all the versions, the D-GAI engine can calculate a version kill score, analogous to a mutation score to capture the quality of the supplied test sets.

Test enhancement services take this one step further by not only determining the version kill score but also leveraging this information to produce a minimal subset of input tests (i.e., test set minimization) that achieves the same (or a comparable) level of test effectiveness.

Test Recommendation

This service resembles the code recommendation service, but recommends new tests, or more precisely test sets. Here the prompt would include the base module the developer wishes to have tests for, rather than the tests the developer wishes to have a code module for. As in the previous cases, the code model would be asked to synthesize new tests as well as new versions of the supplied code whose execution outputs would then be compared for differences. The new versions would be used in the same way as in the previous two services to identify effective sets and return the

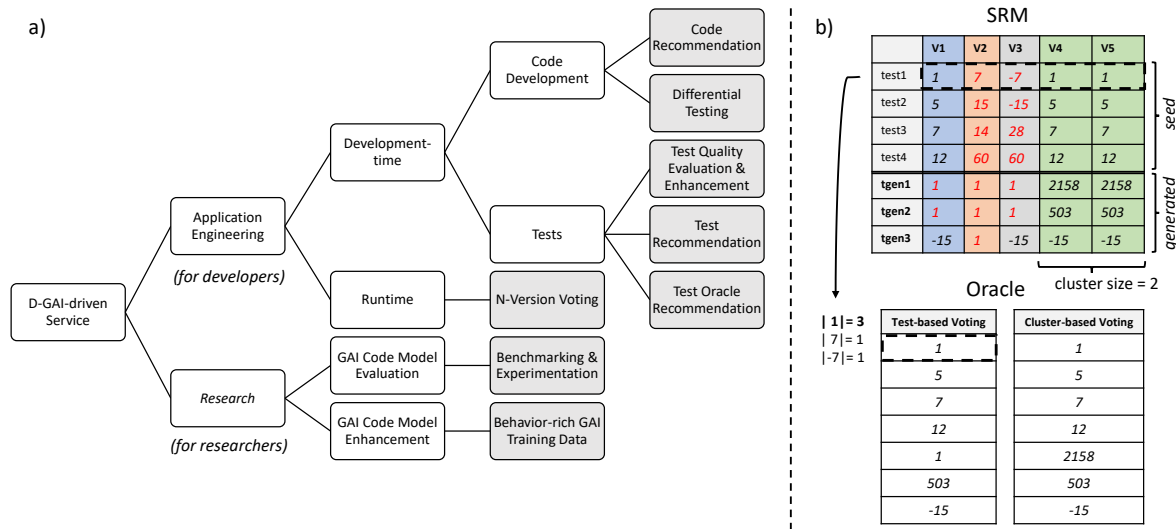


FIGURE 2: (a) Differential GAI Use Cases and Services, and, (b) GCD Oracle Example

minimum number needed to kill the largest possible number of versions.

Test Oracle Recommendation

Obtaining oracles to define correct behavior has traditionally been a big obstacle to test automation, and is still typically performed by hand.⁶ D-GAI can help by facilitating the creation of multiple “verdicts” on correct output values from N versions of the code modules. There are two basic strategies for doing this, illustrated at the bottom of Figure 2(b) for the GCD example. The first approach, *test-based voting*, identifies the oracle values by selecting the most common result for each test case across all N versions, while the second approach, *cluster-based voting*, identifies the oracle values based on behavioral clustering over common outputs over all the tests. The former is essentially the approach taken in the original N-version programming approach.⁵ However, in the example in Figure 2(b), test-based clustering is inconclusive, since none of the N versions match the oracle for all outputs, thus no variant is functionally correct and deemed to be of the highest quality according to these criteria. In contrast, the clustering-based approach leads to two versions being judged to as functionally correct, and thus of the highest quality – namely V4 and V5 which are in the large (green) cluster. In the figure, outputs that deviate from the cluster-based voting oracle are shown in red font.

Research

To evaluate GAI-synthesized code artifact quality, researchers conduct experiments comparing generated

artifacts in various coding problems. D-GAI facilitates this by aggregating SRMs for multiple problems and offering oracle-based quality assessments (cf.⁸). As the number of code modules and tests grows, results generalizability and statistical power improve, allowing more precise reliability estimates. Additionally, combining D-GAI services mirrors existing benchmark enhancement approaches like EVALPLUS.⁹ Furthermore, SRMs offer valuable observational data that can be used to train and improve code models, leading to better performance of GAI-synthesized code modules and tests.

CONCLUSION

This paper has introduced Differential GAI (D-GAI), a practical approach to improving the quality of generative AI (GAI)-produced software artifacts through the comparative analysis of multiple versions. By incorporating “true” semantic information derived from the execution of multiple versions, D-GAI enables more reliable quality assessments without depending on the trustworthiness of any single artifact.

Our key contribution is the introduction of the Large-Scale Software Observatory (LASSO), a platform designed to support large-scale execution and analysis of code versions and tests. LASSO offers developers and researchers a powerful tool for assessing GAI-generated artifacts. By comparing different executions, LASSO facilitates the identification of behavioral patterns, functional discrepancies, and non-functional variations across versions, thereby enhancing the reliability and trustworthiness of GAI outputs.

ACKNOWLEDGEMENT

This research is funded by the Ministry of Science, Research and Arts Baden-Württemberg, Research Seed Capital (RiSC)

REFERENCES

- ¹ H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: <http://www.jstor.org/stable/1990888>
- ² P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.
- ³ B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48, no. 1, sep 2015. doi: 10.1145/2807593. [Online]. Available: <https://doi.org/10.1145/2807593>
- ⁴ M. A. Gulzar, Y. Zhu, and X. Han, "Perception and practices of differential testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019. doi: 10.1109/ICSE-SEIP.2019.00016 pp. 71–80. [Online]. Available: <https://doi.org/10.1109/ICSE-SEIP.2019.00016>
- ⁵ A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, 1985. doi: 10.1109/TSE.1985.231893. [Online]. Available: <https://doi.org/10.1109/TSE.1985.231893>
- ⁶ E. T. Barr, M. Harman *et al.*, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015. doi: 10.1109/TSE.2014.2372785. [Online]. Available: <https://dl.acm.org/doi/10.1109/TSE.2014.2372785>
- ⁷ M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010. doi: 10.1109/MS.2009.161. [Online]. Available: <https://doi.org/10.1109/MS.2009.161>
- ⁸ M. Kessel and C. Atkinson, "Promoting open science in test-driven software experiments," *Journal of Systems and Software Special Issue - Open Science in Software Engineering research*, vol. 212, p. 111971, 2024. doi: <https://doi.org/10.1016/j.jss.2024.111971>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121224000141>
- ⁹ J. Liu, C. S. Xia *et al.*, "Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: <https://openreview.net/forum?id=1qv610Cu7>
- ¹⁰ F. Cassano, J. Gouwar *et al.*, "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, pp. 1–17, 2023. doi: 10.1109/TSE.2023.3267446. [Online]. Available: <https://doi.org/10.1109/TSE.2023.3267446>
- ¹¹ K. Jesse, T. Ahmed *et al.*, "Large language models and simple, stupid bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023. doi: 10.1109/MSR59073.2023.00082 pp. 563–575. [Online]. Available: <https://doi.org/10.1109/MSR59073.2023.00082>
- ¹² M. Kessel and C. Atkinson, "Morescient GAI for software engineering," 2024, (invited for extended version in TOSEM; accepted in SE2030 - Software Engineering in 2030 Workshop; Talk in 1st Alware conference, co-located with FSE 2024, Porto de Galinhas, Brazil). [Online]. Available: <https://doi.org/10.48550/arXiv.2406.04710>
- ¹³ —, "Code search engines for the next generation," *Journal of Systems and Software Special Issue - Software Reuse for the next Generation*, p. 112065, 2024. doi: <https://doi.org/10.1016/j.jss.2024.112065>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121224001109>
- ¹⁴ V. R. Basili *et al.*, "Goal question metric paradigm," *Encyclopedia of software engineering*, pp. 528–532, 1994.
- ¹⁵ G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, dec 2014. doi: 10.1145/2685612. [Online]. Available: <https://doi.org/10.1145/2685612>

Marcus Kessel is a (postdoc) researcher in software engineering at the University of Mannheim, Germany. His research focuses on developing scalable and behavior-aware methods for software code analysis, integrating insights from static and dynamic program analysis with data-driven approaches. The author received the Ph.D. degree in computer science from University of Mannheim. Contact him at marcus.kessel@uni-mannheim.de.

Colin Atkinson is the leader of the software engineering group at the University of Mannheim, Germany. His current research interests include multi-level deep modeling, view-driven software engineering and scalable software analysis and observation. The author received the Ph.D. degree in computer science from Imperial College, London, and is currently a profes-

Special Issue on Generative AI for Software Engineering

sor at the University of Mannheim. Contact him at
colin.atkinson@uni-mannheim.de.