Software Engineering Research in a World with Generative Artificial Intelligence

Martin Rinard
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
rinard@csail.mit.edu

ABSTRACT

Generative artificial intelligence systems such as large language models (LLMs) exhibit powerful capabilities that many see as the kind of flexible and adaptive intelligence that previously only humans could exhibit. I address directions and implications of LLMs for software engineering research.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**;

KEYWORDS

Software Engineering, Generative Artificial Intelligence, Large Language Models

ACM Reference Format:
Martin Rinard. 2024. Software Engineering Research in a World with Generative Artificial Intelligence. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3597503.3649399

1 INTRODUCTION

Software engineering and artificial intelligence have largely comprised disjoint subfields of computer science since the inception of the field in the middle of the 20th century. The focus of software engineering research has been, and continues to be, on issues relevant to software systems that are deployed (or intended to be deployed). The focus of artificial intelligence research has been on understanding how to obtain systems that exhibit aspects of human intelligence, a goal that has been so far from delivering recognizably intelligent systems that deployability concerns have traditionally had little or no relevance to the field.[1]

[1]I acknowledge here that the field of artificial intelligence has produced many useful deployed systems and techniques. But these systems have traditionally been seen as spinoffs that do not themselves meaninfully exhibit any aspect of human intelligence. For much of the history of the field, as soon as a system developed to the point where it could be deployed, people largely understood the system and its behavior and the system was no longer considered to exhibit artificial intelligence.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3649399

With the recent advent of generative artificial intelligence in the form of large language models (LLMs), things have changed. These systems exhibit capabilities that many recognize as the kind of flexible and adaptive intelligence that previously only humans could exhibit. These capabilities include the ability to operate flexibly at different levels of abstraction and formalism in ambiguous situations with varying amounts of information about both the situation and the overall goal. This development surprised essentially every expert in the field. Moreover, these capabilities have proved to be immediately useful across a range of human endeavors. Artificial intelligence systems are now widely used and promise to play a central role in many human endeavors moving forward. This situation places artificial intelligence systems, and more broadly systems that leverage artificial intelligence within a broader system context, squarely within the purview of software engineering. I'll consider some recent research that involves modern artificial intelligence, using these systems as a springboard for a broader discussion of how the combination of artificial intelligence and software engineering research may play out over time.

2 LARGE LANGUAGE MODELS AND TRADITIONAL SOFTWARE ENGINEERING PROBLEMS

Traditional software systems have been the focus of software engineering for the last several decades. These systems store, retrieve, transmit, copy, manipulate, and analyze digital data at scale. While development processes vary, the basic idea is to obtain a system whose goals and behavior are understood and engineered by human developers. It is worth noting that this philosophy and approach has been enormously successful and the resulting systems have had a transformational effect on our society. Over the last several decades the field of software engineering has become familiar with the problems that arise in this context and has delivered (and continues to deliver) a large body of research motivated by these problems, with our ability to successfully design and implement this class of systems proceeding apace. I consider several systems that bring generative artificial intelligence into this basic framework.

2.1 Program Inference and Regeneration

Program inference and regeneration uses active learning to build a model of a software system or component, then uses the model to regenerate a new version of the system or component [Rinard et al.(2018), Vasilakis et al.(2021), Shen and Rinard(2021)]. The technique has

enormous promise for a range of software engineering tasks including software maintenance and retargeting, functionality extraction from legacy systems, and the elimination of software supply chain security vulnerabilities. Our first approach to this problem involved the deployment of fairly heavyweight programming language technology — paired combinations of domain specific languages and corresponding active learning algorithms. While this approach delivers guarantees and benefits unavailable with any other currently available approach, its practical scope has been limited by the fact that each new domain requires the development of a new domain specific language and learning algorithm.

LLMs provide an immediate, lightweight alternative approach. The basic idea is straightforward:

- **Component:** Give the LLM a component, for example from a popular software ecosystem such as npm.
- **Input Generation:** Ask the LLM to generate a set of inputs for the component.
- **Output Generation:** Run the component on the inputs to get corresponding outputs.
- **Regeneration:** Ask the LLM to generate a program that implements the input/output examples. Ideally the LLM will produce a better implementation, for example because any vulnerabilities in the original component are discarded in the regenerated version [Vasilakis et al.(2021)] or because the regenerated version is written in a more desirable language or can execute in a new environment [Rinard et al.(2018), Shen and Rinard(2021)].

It is instructive to consider how little effort is required to perform this set of steps to obtain a regenerated version of a broad range of components. It is also instructive to apply this idea to range of components and observe how much more effort can be invested to maximize the benefits available via this approach. One takeaway is that current LLMs can produce large amounts of software very quickly, but getting software you actually want may require more time and effort. One theme is that the exercise does not resemble traditional engineering but instead is more like interacting with a cheerful bureaucracy where you have to know what to ask and say to get what you want, typically by exploring alternatives until you succeed (or give up and try other options). And it is never fully clear what you can hope to get until you get it. Nevertheless, the ability of LLMs to automatically generate software at scale given relatively little explicit guidance may fundamentally change the technical/economic landscape by dramatically reducing the cost of probable software (especially if the anticipated rapid increases in the capabilties of LLMs materialize). But improbable (i.e., truly new) software may be a very different story.

2.2 Acceptable Survivability

One perceived drawback of code generated by LLMs is that it comes with no guarantees. Here I focus on one important but often underrated and underexplored property, specifically survivability (in the form of continued meaningful execution in the face of errors or anomalies) and advocate a survivable by construction approach. One key insight behind this approach is that most survivable software conforms to a reactive model in which (either explicitly or implicitly) the execution can be decomposed into a sequence of computation units, each of which consumes an input unit, executes a computation that processes the input, then returns back to process the next input unit. Such software often exhibits *short error propagation distances* — as long as the computation preserves basic consistency properties, errors in one computation unit typically do not propagate to successive units as long as the errors are not fatal [Rinard et al.(2004), Long et al.(2014), Shen and Rinard(2017)]. This property endows large software systems with surprising resilience when measures are taken to replace language implementation mechanisms (such as throwing exceptions for null dereferences or out of bounds accesses) that preemptively terminate computations in the face of execution integrity errors with failure oblivious mechanisms that preserve basic consistency while enabling continued execution [Rinard et al.(2004)].

Filtered iterators, which atomically discard computation units that encounter errors (or consistency violations), so that the computation executes as if the corresponding input unit never existed, provide a model of computation that ensures continued execution in the face of otherwise fatal errors [Shen and Rinard(2017)]. In combination with language implementation mechanisms (cyclic memory allocation [Nguyen and Rinard(2007)] and infinite loop termination [Kling et al.(2012)]) that finitize each computation unit (by eliminating all sources of unbounded resource consumption), filtered iterators can guarantee survival via continued execution. Embedding LLM generated code inside a filtered iterator or forcing the LLM to generate filtered iterators can ensure survivability. Filtered iterators are therefore an example of a general class of *behavioral regulation* mechanisms that integrate unreliable or unpredictable components into a larger predictable, reliable system.

It is worth noting that LLMs themselves (like many other machine learning models) implement an inherently survivable model of computation. Because the LLM model of computation is so simple, LLMs are not susceptible to many of the errors that traditional software may encounter. Mechanisms that bound the computation (for example, by bounding the number of generated tokens) can guarantee survivability. In part because of this desirable property, we may see systems that use behavioral regulation mechanisms to acceptably embed machine learning models such as LLMs within larger software contexts, with LLMs implementing core processing computations [Rinard(2003)].

An intriguing property is that LLM outputs are essentially samples from a probabilistic model of the data on which they were trained. A query can be seen as eliciting one or more samples from the probabilistic model conditioned on the query prompt. One notable and useful aspect that LLMs share with other probabilisitic models is the ability to generate multiple samples. This fact enables strategies that repeatedly sample to find a response that has desirable properties (for example, automatically generated code that produces corrent outputs on a given input set or an automatically generated image that elicits positive emotions in a human observer). Current LLMs produce samples one token at a time, enabling strategies that efficiently steer the response by resampling tokens that do not conform to some property that the response must satisfy (for example, conforming to a syntactic requirement specified by a grammar).

Here we can see the field transitioning away from an engineering approach into a hybrid approach that includes components

whose behavior may be poorly understood or unpredictable. One major theme is applying techniques that place enough context around these components to recover guarantees and/or predictability for the system as a whole. Here past research on integrating unreliable components into acceptable software systems may be particularly relevant [Rinard(2003)]. Keys to success include 1) *asymmetrical engineering:* identifying and directing engineering resources to parts of the system that must execute predictably and reliably [Carbin and Rinard(2010)] and 2) *behavioral regulation:* surrounding components with mechanisms (such as filtered iterators or acceptability-oriented computing [Rinard(2003)]) that regulate their interaction with the rest of the system to deliver predictable and reliable behavior (guardrails is often another name for this concept). These techniques, appropriately deployed, can deliver a full range of acceptability properties and not just continued execution.

2.3 Security and Trust

Because software generated by LLMs comes with no guarantees, security and trust are likely to play a prominent role. There are a variety of techniques for addressing potential issues, including techniques (such as those described in the previous subsection) that place generated code in a context that enforces acceptability properties such as security or trust.

Here I highlight a potential threat model, specifically using compromised training data to subvert the LLM to generate code containing security vulnerabilities. It is important to realize that the most likely attack scenario would not involve compromising the enormous amount of pretraining data, but instead the relatively much smaller amount of fine tuning data (which is a much more feasible proposition). Trust is an important but much less precisely defined property. Obtaining trust is an instance of the much broader large language model alignment problem, which is still in its very early stages of exploration and development.

2.4 Probable vs. Improbable Software

LLMs can be seen as powerful probabilistic models of their training data that generate outputs by sampling from the trained distribution conditioned on the prompt. As such, they can be deployed to execute prioritized searches of complex search spaces to (ideally) quickly target desirable points in the space. Many problems in traditional software engineering, as in computer science more generally, can be framed as search problems [Harman and Jones(2001)]. Prominent examples include automatic software repair [Monperrus(2018), Zhang et al.(2023), Perkins et al.(2009), Le Goues et al.(2012)] [Nguyen et al.(2013), Long and Rinard(2016)], software fault localization [Wong et al.(2016)], and test input generation [Wang et al.(2024)]. Given their ability to model data, match patterns at scale, and generate probable responses to complex queries, the field has already delivered an enormous amount of research that deploys LLMs to more precisely and efficiently target desirable points in the search spaces that underly many traditional software engineering problems. This is a classic case of deploying new technology to better solve existing problems and, given the remarkable flexibility and scale of LLMs, I expect this trend to only accelerate in the near future.

There is the widespread expectation that we are still in the very early stages of LLM development and, given the progress to date, may see very rapid improvements in the capabilities of large language models to generate probable software. If so, the cost and difficulty of obtaining probable software will only decrease. I also note that any such development may not necessarily always produce positive outcomes — for example, the ability to cheaply and efficiently generate enormous amounts of probable software may increase the ability of attackers to much more rapidly, cheaply, and efficiently explore the space of potential security exploits.

The implications of LLMs for the creation of truly new (and therefore presumably improbable) software are unclear, if only because the field currently has no good understanding of how improbable truly new software really is. I anticipate a spectrum of possibilities, from software that mostly combines existing patterns and components in new and creative ways (and is therefore mostly probable) to software that is improbable from the ground up. To my knowledge the field currently has no good understanding of where along this spectrum systems are likely to fall, how probable existing software systems are, how different existing software systems are from each other, or where any probability/improbability is concentrated in existing or envisioned software systems. Developing meaningful metrics and methodologies to understand these distinctions is a potentially very interesting direction for the field.

3 INVERTING THE SOFTWARE ENGINEERING PROCESS

Perhaps the most surprising aspect of LLMs is their ability to absorb information at scale, then engage with that information at all levels of abstraction to convert the information into flexibly accessible knowledge. This ability hints at the promise of inverting the software engineering process by enabling LLMs to perform (or play major roles in) many activities previously understood to be the domain of humans alone. At the extreme end of the spectrum, instead of developers telling the computer what to do, the computer tells the developers what system to build (and perhaps even builds the system itself). I have been exploring this idea in the context of using LLMs in an undergraduate compiler development class. Compilers have been extensively studied for decades now, with much of the resulting knowledge available in papers and textbooks that one would expect to see in the training data for modern LLMs. Perhaps because of this reason, our very early experience is promising (although it is far from clear what the final result will be) [tra([n. d.]a), tra([n. d.]b), tra([n. d.]c)].

3.1 Requirements Gathering

Requirements gathering has traditionally been seen as a human activity focusing on interacting with stakeholders to understand their needs. LLMs may transform this activity in at least the following ways: 1) guiding the requirements gathering process to ensure all relevant stakeholders are identified, 2) ensuring that all important aspects of the envisioned system are addressed, 3) suggesting functionality that the system should include, 3) for well understood classes of systems, automatically generating requirements (as opposed to gathering requirements from human stakeholders), and 4) writing portions or even all of the requirements documents.

For many systems we may eventually see requirements gathering shifting from a human activity to an activity largely driven by interactions with an LLM. Relevant research problems here include understanding how to best interact with the LLM to elicit meaningful requirements, how to achieve the best balance between gathering requirements from humans and LLMs, and which domains are best suited for this approach.

3.2 Specification and Design

Much like requirements gathering, these activities have traditionally been seen as primarily human activities. Because LLMs can draw on a huge collection of software specification and design information, they hold out the promise of similarly automating aspects of the specification and design process for traditional software systems. Because LLMs can interact so flexibly at multiple levels of abstraction and precision, interacting with them can help developers conceptualize, explore, and evaluate ideas throughout the design process (to the extent that developers remain involved in these processes). Drawing on the enormous amount of design knowledge present in the training data, we anticipate that LLMs may prove to be particularly useful in identifying particularly productive or problematic aspects of human-generated designs.

4 UNDERSTANDING GENERATIVE ARTIFICIAL INTELLIGENCE SYSTEMS

The behavior of modern generative artificial intelligence systems arises because of poorly understood interactions between the structure, training data, and training process that created the system. The resulting systems exhibit surprising capabilities (well beyond what experts thought was possible even several years ago) and unpredictable, sometimes desirable, sometimes undesirable behavior. This fact motivates research into understanding the reasons behind these phenomena.

Here I focus on recent research investigating whether LLMs develop any meaningful understanding of the domain in which they operate or whether their behavior is due only to leveraging statistical correlations at scale [Jin and Rinard(2023)]. To make the investigation more concrete, the research focuses on the semantics of programs that control agents in a simulated world. An advantage of this approach is its foundation in a precise formalized context devoid of ambiguity.

The experiment trains an LLM to, given an input and output grid world with a robot, synthesize a robot program that transforms the input world to the output world. This program is written in a language that includes operations that move and rotate the robot and enable the robot to act on the world in various ways. The question is whether the internal state of the LLM works with any semantic representation of the robot or world state as it synthesizes the sequence of operations in the program — essentially, does the LLM understand any aspect of the robot or world as it generates the program?

One complication is that any such semantic state may be present but encoded so that it is not immediately apparent. One solution is to train a probe to extract the state if present. It turns out that 1) it is indeed possible to train a probe to extract semantically meaningful aspects of the robot state such as the facing direction of the robot, 2) it is also possible to probe for these aspects of the robot state after one or two (as yet ungenerated) operations into the future, raising the possibility that the LLM may be predicting future states to guide the synthesis, 3) during training the ability of the probe to extract these aspects of the robot state exhibits a phase transition from not being able to extract the state to being able to do so, and 4) the phase transition is correlated with a corresponding phase transition in the ability of the LLM to synthesize a robot program that correctly transforms the input grid world to the output grid world.

A final question is whether the probe is simply learning to process syntactic information about the program generated to date (for example, the probe may be learning to execute a record of the program generated to date) so that the semantics is in the probe and not in the LLM. An experiment that attempts to train a probe to map the same LLM states to different robot states based on an alternate semantics of the robot operations fails to produce a probe as accurate as the probe that works with the original semantics on which the LLM was trained, supporting the hypothesis that the LLM state includes some semantic state (and not just some surface syntactic representation without meaning).

Given the scale and complexity of modern generative artificial intelligence systems, and the prominent role that such systems are likely to play in our future society, I expect to see research that is designed to better understand these systems, including how they operate and the reasons for their behavior, to play a prominent role in the field moving forward.

5 NEW SOFTWARE WITH LARGE LANGUAGE MODELS

New technologies are often first deployed in existing contexts to better solve existing problems. But if the technology is sufficiently transformational, the ultimate role it plays is almost always very different from its initial motivation and deployment. Generative artificial intelligence appears to offer capabilities that are qualitatively different from previous software systems, most prominently the ability to operate flexibly and seamlessly at an abstract conceptual level, at a detailed formal level, and all levels in between. Particularly impressive is the ability to coherently integrate information from very different domains (try asking an LLM what the relationship is between software development and surfing in Portugal).

Current LLMs are trained on enormous amounts of data (by some accounts a substantial fraction of all of the human produced text currently extant) and feature an enormous number of training parameters. The resulting complexity of these models places them outside traditional approaches for understanding and predicting system behavior (here I refer to understanding the design, documentation, and source code of a software system). Instead of being implemented to satisfy a set of requirements or meet a specification, the behavior of the system emerges through the training process. A resulting new strength is the ability to operate in situations with ambiguous, vague goals pursued in the presence of incomplete and varying amounts of information. Presented with such situations, LLMs have the ability to internally generate a precise goal, fill in missing information, and produce a fully realized response.

But many generative artificial intelligence uses (and arguably the most potentially transformative uses) involve situations where an unambiguous specification is not a goal and it is not realistically possible to ever state a complete range of correctness or acceptability properties. Such use cases often involve subjective human interactions, with desirable/acceptable behavior varying across people, organizations, and cultures. There is also the hope that generative artificial intelligence systems will eventually become creative entities that generate new knowledge, new problems and solutions, and ultimately new culture. Ensuring that the resulting behavior is consistent with the goals of the people, organizations, and societies using the system is the artificial intelligence alignment problem. This is an emerging area of research that will only grow in importance over time.

Judging by the published software engineering literature to date, the field remains largely focused on applying this new technology to traditional software engineering problems and has yet to start exploring the truly revolutionary potential that this technology may offer. To fully realize this opportunity, the software engineering community will need to pivot to embrace more ambiguous and subjective system goals than have been the traditional focus of the field (while also developing techniques to ensure conformance to objective requirements and specifications when available and appropriate).

ACKNOWLEDGMENTS

I thank Daniel Jackson, Logan Engstrom, and Sam Park for useful feedback on earlier versions of this paper. Tarushii Goel provided the lexer and parser interaction sessions. I acknowledge support from DARPA and Boeing.

REFERENCES

[tra([n. d.]a)] [n. d.]a. https://chat.openai.com/share/c35c90c5-e6b8-4410-981f-f1b9c0d3b622.

[tra([n. d.]b)] [n. d.]b. https://chat.openai.com/share/25ba04f9-81aa-41c3-9ae2-5a58821823fe.

[tra([n. d.]c)] [n. d.]c. https://chat.openai.com/share/0615381d-bef8-454c-ad3b-1e981f9f3800.

[Carbin and Rinard(2010)] Michael Carbin and Martin C. Rinard. 2010. Automatically identifying critical input regions and code in applications. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 37–48. https://doi.org/10.1145/1831708.1831713

[Harman and Jones(2001)] Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Inf. Softw. Technol.* 43, 14 (2001), 833–839. https://doi.org/10.1016/S0950-5849(01)00189-6

[Jin and Rinard(2023)] Charles Jin and Martin Rinard. 2023. Evidence of Meaning in Language Models Trained on Programs. arXiv:2305.11169 [cs.LG]

[Kling et al.(2012)] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin C. Rinard. 2012. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 431–450. https://doi.org/10.1145/2384616.2384648

[Le Goues et al.(2012)] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[Long and Rinard(2016)] Fan Long and Martin C. Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 298–312. https://doi.org/10.1145/2837614.2837617

[Long et al.(2014)] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 227–238. https://doi.org/10.1145/2594291.2594337

[Monperrus(2018)] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[Nguyen et al.(2013)] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[Nguyen and Rinard(2007)] Huu Hai Nguyen and Martin C. Rinard. 2007. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, Greg Morrisett and Mooly Sagiv (Eds.). ACM, 15–30. https://doi.org/10.1145/1296907.1296912

[Perkins et al.(2009)] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 87–102. https://doi.org/10.1145/1629575.1629585

[Rinard(2003)] Martin C. Rinard. 2003. Acceptability-oriented computing. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 221–239. https://doi.org/10.1145/949344.949402

[Rinard et al.(2004)] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 303–316. http://www.usenix.org/events/osdi04/tech/rinard.html

[Rinard et al.(2018)] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active learning for inference and regeneration of computer programs that store and retrieve data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*, Elisa Gonzalez Boix and Richard P. Gabriel (Eds.). ACM, 12–28. https://doi.org/10.1145/3276954.3276959

[Shen and Rinard(2017)] Jiasi Shen and Martin C. Rinard. 2017. Robust programs with filtered iterators. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 244–255. https://doi.org/10.1145/3136014.3136030

[Shen and Rinard(2021)] Jiasi Shen and Martin C. Rinard. 2021. Active Learning for Inference and Regeneration of Applications that Access Databases. *ACM Trans. Program. Lang. Syst.* 42, 4 (2021), 18:1–18:119. https://doi.org/10.1145/3430952

[Vasilakis et al.(2021)] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1755–1770. https://doi.org/10.1145/3460120.3484736

[Wang et al.(2024)] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]

[Wong et al.(2016)] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[Zhang et al.(2023)] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 55 (dec 2023), 69 pages. https://doi.org/10.1145/3631974