Navigating Challenges with LLM-based Code Generation using Software-specific Insights

Nikitha Rao

CMU-S3D-25-101
April 2025

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This work was supported by the Hima and Jive Fellowship, IBM Gift (IC-5010), Google Collab PhD Fellowship (Google INC– PO4100118057), and Vincent Hellendoorn's Start-Up Funds.

Keywords: Large Language Models for Code, Generative AI, Verification, Reliability

For Flynn, whose wagging tail kept me going

Abstract

The software development process is rapidly evolving with the advancement of Large Language Models (LLMs). LLMs are not only transforming the way code is written but are also increasingly integrated into AI programming tools, such as ChatGPT and GitHub Copilot, to enhance developer productivity by generating programs from natural language instructions, identifying and fixing bugs, generating documentation and so on.

These LLMs are pretrained on large volumes of natural language and code data. They are trained using cross-entropy and preference losses that have no coefficient for correctness and only optimize for matching the ground truth. Therefore, despite their proficiency in learning code syntax, they fall short in capturing semantic signals. To date, the main focus of efforts to improve these models has been training larger models and collecting more human preference data. However, user studies have found notable issues with the usability of these larger models, including difficulty in understanding the generated code, the presence of subtle bugs that are hard to find, and a lack of verification of the generated code.

This dissertation demonstrates that integrating domain insights from software engineering into AI-based code generation can enhance reliability and utility for developers. This is done by empowering the model to take on a more active role in building valid and usable code, instilling greater trust among users in the capabilities of the model. I focus on three main challenges identified by prior work and propose solutions using software-specific insights.

(1) The generated code can be difficult to understand and manipulate, especially for non-expert programmers. To address this, I contribute LOWCODER, a tool that abstracts away the syntactic complexity associated with traditional code and provides a more user-friendly interface using drag-and-drop functionality. As a result, LOWCODER provides a trusted environment where users can leverage the capabilities of AI without the need for extensive coding knowledge.

(2) Verifying the correctness of the generated code is hard. While LLMs excel at generating code, they are lacking when it comes to generating tests. This is largely because current models are trained on individual files and therefore can not consider the code under test context. To overcome this, I contribute CAT-LM, a LLM trained to explicitly consider the mapping between code and test files. CAT-LM can therefore help users with verifying code that they or other models generate, by generating tests that align more coherently with the underlying code.

(3) The generated code often has subtle bugs that are hard to find. To address this, I contribute DIFFSPEC, a framework for generating differential tests with LLMs using prompt chaining to verify code correctness. DIFFSPEC makes use of various software artifacts like natural language specification documents, source code, existing tests, and previous bug reports to generate tests to not only verify code correctness, but also checks for conformance against the specification. By highlighting meaningful behavioral differences between implementations, DIFFSPEC can enhance the overall reliability of even extensively tested software systems.

The goal of my dissertation is to demonstrate the significance of integrating software-specific insights when training models to make code generation more reliable and useful for developers. My dissertation work contributes several artifacts including datasets, evaluation frameworks and models that are trained by integrating software-specific insights to improve the quality of generated code. Importantly, these models are all quite small relative to cutting-edge general purpose models like GPT-4. While large, general models can also be very useful for these tasks, they have their own limitations: few companies can afford the immense resources required to train such large models, and most of these models are closed-source and provide limited (free) access to the community which can be unreliable. In contrast, my work produces smaller open-source models that are specialized to perform various programming related tasks, resulting in tools that make code generation more reliable and useful for developers.

Acknowledgments

These past four years have been nothing short of incredible. This PhD has not only helped me transform into a better researcher but has also helped me grow so much as a person and taught me skills I will carry for life. I cannot even begin to express the gratitude I feel for everyone who made this experience so memorable, but I will try.

First and foremost, I'd like to thank my advisor Vincent Hellendoorn for always being willing to take the time to give me advice and feedback on anything and everything. But most importantly, thank you for believing in me even when I didn't. I also really appreciate all the times you've given me a list of reasons why I shouldn't worry about something when I worry. I'd also like to thank my co-advisor Claire Le Goues for adopting me into SquaresLab. Thank you for always cheering me on and encouraging me to have a better balance. Thank you both for moulding me into the researcher that I am today. I know I parent trapped you at times, and I'm sorry but it always worked!

I'd also like to thank my thesis committee members, Andrew Begel, Daniel Fried and Tom Zimmermann for all the mentorship and advice throughout the years. Thank you for all the questions and comments that helped refine this dissertation into what it is. I'd also like to thank Bogdan Vasilescu, Rohan Padhye, Christian Kästner, and Joshua Sunshine for all the little pieces of advice you've given me over the years and checking in with me from time to time. I'd also like to extend a big thank you to Connie Herold, Alisha Roudebush, Dabney Schlea, Jennifer Cooper, Cole Jester and Tom Pope for helping me with all the admin and IT related concerns during my time here. I'd also like to thank other collaborators and mentors I've had throughout the years, namely, Martin Hirzel, Jason Tsay, Kiran Kate, Chetan Bansal, Sunayana Sitaram, B. Ashok, Aditya Kanade, Gopal Srinivasa, Karthik Ramachandra, Kalika Bali, Sonu Mehta, Chandra Maddila, and Sreangsu Acharyya.

This PhD has been more than just research and deadlines. I have so many fond memories that I made with so many wonderful people that I get to call friends, and I cannot thank them enough. Thank you Sanjith Athlur for your unconditional support with every little thing. Thank you Daniel Ramos for always being there, despite being on a different continent. Thank you Sam Estep for all the baking, brunches, and little rant sessions. You and Lee Brunco are the most incredible hosts, and always made me feel at home. Thank you Peter Carragher for all the fun adventures, and the pet dragon in Dungeons and Dragons.

Thank you to all the dog moms, Jane Hsieh, Jenny Liang, Courtney Miller, and Anna Kawakami, for helping me care for Flynn when I couldn't. Thank you Vasu Vikram for all the witty banters. (Let the record say that climbing is superior to pickleball). Thank you to Sagar Bharadwaj, Samvid Dharanikota, and Aditi Kabra for being so supportive and making me feel at home. Thank you Harrison Green and Andrew Haberlandt for climbing and getting pancakes at midnight with me. Thank you Daye Nam for being the best senior and for all the advice and conversations, especially when I had doubts (or an existential crisis) about my work. Thank you

Luis Gomes for all the walks around Schenley park.

Thank you to Catherine King for being the best office mate, and all the Taylor Swift song recommendations. Thank you Shrey Tiwari for all our rant sessions. Thank you Samantha Phillips for always matching Flynn's energy. Thank you Eli Claggett for all the conversations over tea. Thank you Sherry Sarkar, for always cheering me on and pushing me to do better at the climbing gym. Thank you Travis Evans and Evan Williams (and Cat, Sam and Peter) for fighting land sharks and skeletons with me in Dungeons and Dragons. Thank you Elisabeth Heremans for taking me to the secret climbing gym inside a church in Belgium and getting waffles with me. Thank you Vaibhav Bhosale for all the hiking and national park trips. Thank you Mik Zlatin for encouraging me to celebrate small wins.

I'd also like to extend a big thank you to all my lab mates from both IPSUM and Squareslab - Aidan Yang, Andrea Miller, Cláudia Mamede, Jeremy Lacomis, Kaia Newman, Kush Jain, Luke Dramko, Manisha Mukherjee, Paulo Santos, Tobi Dürschmid, Trent Tabor - for all the feedback you've given me during lab meetings, and all the fun lab socials we shared. I also want to thank all the other friends I made along the way, both at CMU, as well as at internships - Elizabeth Gilbert, Parv Kapoor, Paul Liang, Soham Pardeshi, Ao Li, Catarina Gamboa, Nadia Nahar, Kyle Liang, Simon Chu, Ian McCormack, Chenyang Yang, Leo Chen, Hugo Simao, Maria da Loura Casimiro, Abhishek Vijay, Manish Shetty, Sahil Bhatia, Sachin Goyal, Tarun Raheja, Nilay Pochi, Sai Surya, Aditya Kusupati, Zoe McIntyre, Ginnae Ayenew, Priyan Vaithilingam, Tessa Han, Christos Zarkos, Mingwei Zheng. Thank you for all the conversations, and fun memories we shared.

Lastly, I'd like to thank my family for their unwavering support and for always reminding me that they were proud of me. Thank you to my dad, Sathyanarayan Rao, for always being a phone call away and listening to me rant about anything and everything, no matter how late at night it was. Thank you to my mom, Kalpana Narayan Rao, for always reminding me that she loved me and missed me. Thank you to my brother, Bharath Bhambore, for encouraging me to do other things besides work, and celebrating small wins in my climbing journey. Thank you to my extended family for checking in on me and supporting me through this time. I cannot end this without extending a huge thank you to my dog, Flynn. He has spent endless hours cuddling up to me as I worked on this dissertation and other deadlines. He is the happiest dog, and his silly shenanigans never fail to make me laugh. I could not have done this without you, Flynn!

As I begin the next chapter of my life, I hope that this is a "see you soon" and not a "goodbye"!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of Large Language Models (LLMs) has led to the widespread use of AI-powered programming tools such as ChatGPT [7] and GitHub Copilot [6] in the software engineering community. These tools have fast become key resources for developers, playing a crucial role across various stages of the software engineering lifecycle. They facilitate tasks ranging from generating code to identifying and fixing bugs, generating tests for verification, and even generating comprehensive documentation.

LLMs are generally trained in a multi-stage process. First, the model is pretrained on large volumes of natural language and code data. This allows the LLM to learn the patterns in code including syntax, grammar, and contextual relationships within the given data. This is followed by a finetuning and/or instruction-tuning process to better align the model with human preferences. For code, this typically involves using code or pairs of natural language and code that help align the model to perform specific tasks and better understand the code semantics. Predominant training techniques often make use of cross-entropy and preference losses that just optimize for matching the ground truth and not code correctness. Thus, when applied to code, the models are only learning the syntax explicitly, not the semantic signals that the developers consider.

To date, the primary approach to enhancing model performance has revolved around training larger models with increased parameters or incorporating more training data [91, 106]. However, user studies have revealed notable issues with the usability of these larger models, including difficulty in understanding the code generated (especially for non-expert programmers) [171, 202], the presence of subtle bugs that are hard to find [125, 202] and lack of verification of the generated code [26, 125]. While scaling has proven to improve model performance on various benchmarks, it does not address the issues developers face when using these models.

My work explores an alternative to the scaling-centric approach by incorporating domain insights from software engineering. The goal is to address these challenges by proposing techniques that draw insights from software engineering. By integrating software-specific insights during training and evaluation of LLMs of code, we can produce more effective models that make code generation more reliable and useful for developers.

1.1 Challenges with LLM-based Code Generation

Despite the constant efforts being made to improve LLMs for various code understanding and generation tasks, several challenges remain that make the generated code less useful for developers in practice. Here, I elaborate on three main challenges identified by prior work and propose techniques for addressing them, while highlighting the software-specific insights employed in each method.

Challenge 1: Syntactic Complexity

AI programming tools like Copilot [6] and ChatGPT [7] can generate code from natural language instructions, which is especially helpful in ecosystems with large APIs. However, a key problem with these tools is that they generate (potentially complex) code, which can be difficult to understand and manipulate [202]. This is especially true for novices or non-expert programmers, where the complexity of textual code often makes it difficult for them to reason about the generated code. Studies have found that individuals who are not very proficient in coding were less inclined to use AI tools for code generation [171]. This reluctance arises from the background knowledge and skill necessary to comprehend the correctness and quality of AI-generated code. Additionally, intervening effectively requires programming experience in order to manipulate the generated code into a usable format [171].

This challenge also persists with experienced programmers. Studies found that users often discarded the code generated by Copilot when it did not behave as expected [202]. This was largely because they did not understand several parts of the generated code and therefore did not know how to debug the code. Others felt that it was more efficient to rewrite the whole code from scratch rather than to spend time reading and understanding the generated code to make the necessary changes [202].

Key Insight: The use of *abstraction*, in this case through low-code or visual programming, can help overcome the challenge of syntactic complexity in LLM generated code.

Solution: I developed LowCoder [170], a tool that abstracts away the syntactic complexity associated with traditional code and provides a more user-friendly interface using drag-and-drop functionality. LowCoder is the first low-code tool for developing AI pipelines that supports both a visual programming interface and offers an AI-powered natural language interface. The hypothesis is that the respective strengths of these two low-code techniques can compensate for each other's shortcomings. Programming by natural language (PBNL) uses AI to help users retrieve and use programming constructs based on natural language queries. This does not always result in correct programs, necessitating a way to help users understand and fix generated programs. Visual programming complements PBNL by providing a clear, unambiguous representation of the program that users can directly manipulate to experiment with alternatives.

LowCoder is leveraged to offer some of the first insights into both how and when low-code programming and PBNL assists developers with various degrees of expertise. This is done by conducting a user study with 20 participants with varying levels of AI expertise using LowCoder to complete four tasks, half of which with the help of the AI-powered search component. Overall, the combination of visual programming along with the natural language interface helped both novice and experienced users to successfully compose pipelines (85% of tasks) and then further refine their pipelines (72.5% of tasks) when using AI-powered search interface. Additionally, the AI-powered natural language interface helped users discover previously-unknown operators in 75% of the tasks compared to just 32.5% using other methods like web search. This work highlights the benefits of combining the power of AI with low-code programming and overcomes the challenge of syntactic complexity by abstracting away textual code.

Takeaway: AI has shown a lot of potential in empowering individuals with limited or no programming experience to write code, but this comes with its own set of challenges. The most important one being difficulty in understanding and reasoning about the generated code. We can address this challenge by abstracting away textual code and replacing it with more intuitive interfaces like drag-and-drop user interfaces. LOWCODER facilitates the integration of AI into a trusted environment tailored to the needs of non-expert programmers. Through LOWCODER, we show that AI can still be just as useful at this level of abstraction. Specifically, the natural language model supports users in the visual space despite being trained on textual code. Consequently, LOWCODER provides a user-friendly space where individuals can leverage the capabilities of AI without the need for extensive coding knowledge, thereby enhancing accessibility and usability.

Challenge 2: Verification

Verifying the correctness of automatically generated code is yet another challenge that comes with using LLMs. In software development, developers use tests to verify the correctness of the code they write. In well-tested projects, most code files are paired with at least one corresponding test file that implements unit and/or integration test functions that evaluate the functionality of the code. However, writing high quality tests can be time-consuming [30, 31] and is often either partially or entirely neglected. This has led to extensive work on automated test generation, including both classical [25, 40, 62, 72] and neural-based methods [62, 205, 219].

Classical test generation tools like EvoSuite [72] directly optimize to generate high-coverage tests. However, the generated tests are often hard to read and may be unrealistic or even wrong [157]. This requires time and effort from developers to verify generated test correctness [40]. Meanwhile, LLMs trained on code have made major strides in generating human-like, high-quality functions based on their file-level context [28, 47, 73, 151]. AI-powered tools like Copilot excel at code generation, and can significantly improve the productivity of its users [6]. Currently, these models are less well-suited for test generation, because they tend to be trained to generate the code in each file separately, standard practice in natural language processing and therefore can not consider the code under test context when generating the tests.

Key Insight: Generating meaningful tests critically requires considering the alignment between the tests and the corresponding code under test.

Solution: To overcome this challenge, I developed CAT-LM [168], a language model trained on aligned **C**ode **A**nd **T**ests. CAT-LM is a bi-lingual GPT-style LLM with 2.7B parameters. It is trained on a large corpus of Python and Java projects using a novel pretraining signal that explicitly considers the mapping between code and test files, when available, while also leveraging the (much larger) volume of untested code. Modeling the code file along with the test leads to additional challenges regarding a model's context length, which is overcome by training CAT-LM with a context window of 8,192 tokens.

CAT-LM is evaluated against several strong baselines across two realistic applications: test method generation and test method completion. For test method generation, CAT-LM is compared with both human written tests as well as the tests generated by StarCoder [119] and, the CodeGen [151] model family, which includes mono-lingual models trained on a much larger budget than ours. CAT-LM is also compared against TeCo [150], a recent test-specific model, for test completion.

The results show that CAT-LM effectively leverages the code file context to generate more syntactically valid tests that achieve higher coverage on average than StarCoder and all CodeGen models, and substantially outperforms TeCo at test completion. CAT-LM provides a strong prior for generating plausible tests. When combined with basic filters for compilability and coverage, it frequently generates tests with coverage close to those written by human developers. This highlights the merit of combining the power of large neural methods with a pretraining signal based on software engineering expertise—in this case, the importance of the relation between code and test files.

Takeaway: While LLMs excel at code generation, they are limited in their ability to generate tests because of the way they are trained to generate individual code files independently, a standard practice in natural language processing. As a result, they *can not* consider the code under test context when generating the tests. CAT-LM addresses this challenge by explicitly considering the mapping between code and test files during training. This enables users to generate tests that align more coherently with the underlying code, thereby enhancing the quality of tests produced. Moreover, CAT-LM supports users in verifying both the code they write and that which is generated by other LLMs, ensuring a more comprehensive and reliable testing process.

Challenge 3: Reliability

Several AI powered applications now rely on querying an LLM through an API call by using detailed natural language prompts. In fact, as of January 2024, there have been over 3 million custom versions of ChatGPT for specific tasks that are based on meticulously designed prompts [9]. These prompts include a detailed descriptions of all the requirements or specifications that the model needs to conform to, in order to provide the desired output. Requirements play a critical role in the development of these applications, and need to be explicitly stated. These could include both functional and non-functional requirements such as usability, reliability, latency, costs, privacy and safety [126]. However, nearly 54% of participants indicated that the code generation tools often fail to meet the specified requirements (both functional and non-functional) [125], therefore leading to unreliable code being generated.

Ensuring that code is reliable and conforms to the given set of requirements or specifications is not easy. Currently, this is a predominantly manual process, that involves developers reviewing the specifications and code to ensure it's implemented correctly, or a handwritten set of conformance tests [141]. This can be very tedious and expensive, and is prone to errors. Differential testing has shown significant success especially in testing language implementations, such as uncovering bugs in C compilers [113, 229] or browser engines, for example revealing inconsistencies in JavaScript interpreters and JIT compilers [34]. It also can be useful for testing cross-platform consistency (i.e., the same system across different configurations or operating systems) [67] or versions (as in regression testing) [76]. The key idea is to test two or more different systems (or two different versions of the same system) that should behave the same way under the same conditions on the same inputs. If their output behavior differs, it is likely that at least one of the implementations is incorrect.

Generating tests that specifically target differences between two versions of a program is especially challenging, as it involves simultaneously searching the vast input space of two programs to find rare inputs that trigger often subtle discrepancies [139]. Existing approaches to find such tests limit the possible search space by borrowing techniques from symbolic execution [179], guided semantic aware program generation [107], type aware mutations [99], and code coverage optimizations [50]. While some approaches leverage semantic and syntactic properties of the code or use information from static analysis tools, they are significantly limited in their ability to harness the wealth of information available from natural language artifacts.

Key Insight: Natural language requirements and specifications are critical for checking code conformance and ensuring that the code does what it is supposed to.

Solution: DIFFSPEC [169] is a framework for generating differential tests with LLMs using prompt chaining. DIFFSPEC takes into consideration software artifacts like natural language specification documents, the entire source code, existing tests, and previous bug reports. It can generate targeted tests that align more coherently with the specification and can therefore check the conformance of the code. These tests can also highlight meaningful behavioral differences between implementations, that points to potential bugs.

DIFFSPEC is evaluated on multiple implementations of two extensively tested and widely adopted frameworks: Wasm validators and eBPF runtimes. Using DIFFSPEC, we generated 1901 differentiating tests, uncovering at least four distinct and confirmed bugs in eBPF, including a kernel memory leak, inconsistent behavior in jump instructions, undefined behavior when using the stack pointer, and tests with infinite loops that hang the verifier in ebpf-for-windows. We also found 299 differentiating tests in Wasm validators pointing to two confirmed and fixed bugs. With DIFFSPEC, we show that considering software artifacts beyond just the code under test, such as specification documents, bug reports and so on, can help generate meaningful tests that both verifies code correctness and checks for conformance, therefore enhancing the reliability of the software systems.

Takeaway: With DIFFSPEC, we show that considering software artifacts beyond just the code under test, such as specification documents, bug reports and so on, can help generate meaningful tests that both verifies code correctness and checks for conformance. DIFFSPEC is a framework for generating differential tests with LLMs using prompt chaining. It can generate targeted tests that align more coherently with the specification and can therefore check the conformance of the code. We demonstrate that these tests can highlight meaningful behavioral differences between implementations, that point to bugs in two extensively tested systems, namely, eBPF runtimes and Wasm validators, therefore improving the overall reliability of these systems.

1.2 Thesis

Thesis Statement

By incorporating software engineering domain insights during the training and evaluation of Large Language Models of code, we can enhance the quality of the code they generate, thereby making them more reliable and useful for developers.

To evaluate the claim, I focus on three main challenges identified by prior work and propose solutions that make use of a key insight from software engineering domain.

1. I address the challenge of *syntactic complexity*, which makes it hard especially for non-expert programmers to reason about the generated code, with LOWCODER. LOWCODER abstracts away the syntactic complexity associated with traditional code and provides a more user-friendly interface using drag-and-drop functionality. As a result, LOWCODER provides a trusted environment where users can leverage the capabilities of AI without the need for extensive coding knowledge.
2. I overcome the challenge of *verification* using AI models by proposing CAT-LM. Unlike current LLM based models, CAT-LM is trained to explicitly consider the mapping between code and test files, something standard models cannot do. CAT-LM can therefore help users with verifying code that they or other models generate, by generating tests that align more coherently with the underlying code.
3. I overcome the challenge of *reliability*, by generating targeted tests that checks both code conformance against specifications, as well as code correctness. DIFFSPEC is a framework for generating differential tests with LLMs using prompt chaining that generates targeted tests that highlights meaningful behavioral differences between implementations by making use of context extracted from various software artifacts including natural language specifications, code implementations, bug reports and so on.

Empowering models to take on a more active role in building valid and usable code enables us to enhance the reliability of the generated code and increase trust in their outputs.

1.3 Outline

In Chapter 2, I provide background on LLMs, how they are trained, along with details of how they are used for code generation, and outline the challenges discovered with using AI-powered tools for code generation through human studies. In the following chapters, I will discuss each challenge along with my proposed solution that addresses the respective challenge. Table 1.1 provides an overview of the models and tools I developed that are included in this dissertation. Chapter 3 discusses the challenge of syntactic complexity, which I overcome by abstracting the textual code with LowCoder, a low-code tool for developing AI pipelines that supports both a visual programming interface as well as an AI driven natural language interface. I then discuss the challenge of verifying code correctness in Chapter 4 and how it's overcome by generating tests with CAT-LM, a LLM trained to explicitly consider the mapping between code and test files to improve the quality of tests generated. Then, I look at the challenge of reliability in Chapter 5 and propose DiffSpec, a framework to generate tests to verify code correctness and conformance of real world systems. DiffSpec is a prompt chain framework for differential testing that generates tests using natural language specifications as well as code artifacts. Lastly, I provide a summary of my contributions in Chapter 6, followed by discussion and future work in Chapter 7 and Chapter 8, and conclude with Chapter 9.

Table 1.1: Overview of all the tools and models in this dissertation.

Chapter	**Challenge**	**Software Engineering Insight**	**Tool or Model**
3	Syntactic Complexity	Abstraction of textual code	LowCoder
4	Verification	Code-test dependency	CAT-LM
5	Reliability	Specifications and code artifacts	DiffSpec

The contributions presented in this dissertation were carried out collaboratively with others. In acknowledgment of these collaborations, the use of "we" is employed in the subsequent chapters instead of the singular first person.

Chapter 2

Background

2.1 A brief history of Large Language Models

Language models are predictive models of text. They learn to estimate the probability of a token (or word) within a sequence of tokens. Accordingly, they can be used for a variety of tasks including text generation, machine translation, question-answering, and so on [2]. Early computational language models, such as n-gram models [87], were statistical in nature. While simple, they were remarkably effective at predicting the likelihood of words based on patterns from large corpora of text. More recent developments in neural networks have since led to more advanced models such as RNNs and LSTMs [186]. While these models are effective at capturing relationships with a sequence, they struggle to capture long-range dependencies.

The Transformer architecture, introduced by Vaswani et al. [203], was conceptualized around the idea of attention. This made it possible to model arbitrary dependencies between tokens in long sequences. More specifically, for each input token, attention estimates the relevance of every other token. This proved to be highly effective in capturing contextual information in language.

The development of Transformers and self-attention is pivotal in the advancements made in language modeling and in the development of Large Language Models (LLMs). LLMs amplify the scale of these models, enabling them to handle a wide array of natural language processing tasks. In contrast to earlier language models that could only reliably predict the next few words in a sequence, LLMs can generate long sequences of texts, including entire documents.

LLMs undergo extensive training on large volumes of data, typically mined from the Internet, books and other sources. Correspondingly, they require substantial amounts of compute resources, with associated costs reaching millions of US dollars. Training usually involves multiple stages. First, they undergo a self-supervised pretraining phase, wherein an LLM learns the statistical relationships between tokens in textual documents. This pretraining phase is commonly followed by additional stages, including finetuning and instruction tuning based on reinforcement learning from human feedback (RLHF) [155] to refine and enhance the performance by incorporating feedback.

Besides text generation, LLMs exhibit strong performance in NLP tasks including summarization, text classification and question-answering. Their extensive training also enables them to reason about code and math problems. An extension to enhance the capabilities of LLMs involves training with code extracted from public respositories on GitHub as well as other coding platforms like StackOverflow and documentation pages. This makes it possible to generate programs from natural language instructions [36, 212]. Including code from open source software has now become common practise when training LLMs such as PaLM [52], Chinchilla [91], GPT-4 [154], and Llama [200].

2.2 LLMs for SE

LLMs have not just revolutionized the field of natural language processing; their impact also extends to software engineering. The development of LLMs has led to the widespread use of AI-programming tools like ChatGPT and GitHub Copilot, transforming traditional SE into an AI augmented software development process [32]. LLMs have proven useful at assisting developers across various stages of the software development life cycle from extracting requirements [23] to repairing bugs [17, 102, 220]. They can be used for several tasks including: generating code from natural language [214], generating tests to verify code correctness [62, 150, 205, 219], finding and fixing bugs [17, 86, 102], summarizing code [16], generating documentation [135], automatic refactoring of code [164], clone detection [63] and so on. Beyond this, they can be used to automate code review [133], and suggest improvements [209]. LLMs can also be used as educational tools as they are uniquely placed to support developers by providing expert knowledge in the form of conversations, answering questions about the code to aid understanding by providing explanations and examples [32, 146].

2.3 Training LLMs for SE

LLMs are typically pretrained on vast datasets containing both natural language and code, and then finetuned for various code understanding and generation tasks. Program synthesis using natural language prompts first emerged with LLMs such as Codex [47], and CodeGen [151] models. Since then there have been several other models that consistently demonstrate better performance on various code benchmarks, which include open source models like StarCoder [119], CodeLLama [178], SantaCoder [18], CodeGen2 [152], Incoder [73], GPT-NeoX [37], as well as closed source models like GPT4 [154], AlphaCode [124], CodeT5+ [216]. These models have additionally been trained with different types of pretraining methods, these include:

- **Autoregressive Language Modeling**: Autoregressive or causal LM involves left to right generation where the goal is to generate the next token based on the previous tokens. These include models like Codex [47], GPT3 [42], PolyCoder [227], CodeGen [151], StarCoder [119] and so on, where the left-to-right nature of these models make them extremely use for generation tasks such as code completion.
- **Masked Language Modeling**: Masked language modeling is a popular bidirectional objective function that aims to predict the masked tokens based on surrounding context. Models like CodeBERT [68] and CuBERT [105], trained using this objective, are able to generate useful representations of a sequence of code which can be used for downstream tasks such as code classification, defect detection, and clone detection.

- **Infilling**: Also known as causal masking objective, which allows the model to fill in the missing lines of code based on the prefix and suffix context. Models like FiM [28], InCoder [73], CodeGen2 [152], StarCoder [119], SantaCoder [18], CodeLlama [178], make use of bidirectional context to fill in masked out regions of code, allowing them to perform various tasks including inserting missing lines of code, predicting return types of functions, generating docstrings, renaming variables, and inserting missing code tokens.
- **Encoder-Decoder Language Modeling**: Encoder-decoder models such as CodeT5 [215], PLBART [15] first encode an input sequence, and then use a left-to-right LM to decode an output sequence that is conditioned on the input sequence. They are pretrained using seq-to-seq denoising sequence reconstruction (where goal is to generate the original sequence given the corrupted sequence) or masked span prediction objectives (where the goal is to generate the missing content for masked spans in the input sequence) and are often finetuned on various downstream tasks including code summarization, refinement, translation, fixing bugs.
- **Hybrid Models**: CodeT5+ [216] is a sequence-to-sequence model that has been trained with a progression of objectives and pretrained initializations (including span denoising, causal LM, contrastive loss and matching loss) that operates in different modes (encoder only, decoder only and encoder-decoder) to perform various code generation and understanding tasks, including retrieval augmented generation.

These pretrained models are typically finetuned for specific tasks and, more recently, instruction tuned using RLHF to make the generations better align with the feedback provided [155]. Models like CodeLlama [178], WizardCoder [136] and InstructCodeT5+ [216], OctoCoder [143] have been instruction tuned to improve the generalization ability of the models to a wide variety of tasks.

On the other hand, when it comes to improving the performance of a model for a given task, most of the advancements have focused on a scaling-centric approach — training larger models and using more data [91]. My work proposes an alternative approach that incorporates software specific insights to build more effective models are more reliable and useful for developers.

2.4 Studies on LLM-based AI Programming Tools

New LLMs for code are constantly being developed and continue to demonstrate better performance on various code benchmarks. At the same time, researchers have been actively studying the potential of these LLM based AI programming tools.

Several studies have been conducted to evaluate the quality of code generated by LLMs [70, 125, 129, 161, 202]. There have also been feasibility studies conducted for using LLMs in development tools [26, 96, 149, 196, 202] as well as using LLMs in education [83, 109, 171].

Another class of studies focus on the usefulness of the LLM based programming tools. While certain recent studies show no significant difference in using AI programming assistants concerning task completion [202, 228] and code quality [96] , contrasting findings suggest that these tools have a positive association with developers' self-perceived productivity [237].

Ziegler et al. [237] analyzed telemetry data and survey responses to understand developers' perceived productivity with GitHub Copilot which showed that users only accepted close to one-fifth of the suggestions provided.

A study by Vaithilingam et al. [202] which compared the user experience of traditional auto-complete with GitHub Copilot found no significant effect on task completion time. It was also found that users often discarded the code generated by Copilot when it did not behave as expected and others often felt that it was more efficient to rewrite the whole code from scratch rather than trying to spend time reading and understanding the generated code to make the necessary changes.

Barke et al. [26] used grounded theory analysis to understand how programmers interact with models that generate code using Github Copilot. They found that developers often interacted with the tool with one of two modes, namely, acceleration and exploration. In the acceleration mode, the programmer employed Copilot to expedite tasks with a clear understanding of the next steps. In exploration mode, when uncertain about the next steps, the programmer used Copilot to explore various options.

Liang et. al. [125] performed a large scale survey and found that developers are often motivated to use AI programming assistants as they can aid them with code completion and recalling syntax which in turn helps reducing the number of key-strokes and helps them finish programming tasks more quickly. They also found that developers refrain from using AI programming tools primarily due to the following reasons: (1) tools not producing code that fulfills specific functional (e.g., security, performance) or non-functional requirements, (2) developers encountering difficulties in controlling the tool to generate the desired output and (3) developers spending too much time debugging or modifying the generated code.

Rasnayaka et. al. [171] conducted a user study with students to analyse the usefulness of LLMs for an academic software engineering project.They found that LLMs were most effective during the early stages of software development, especially with generating foundational code structures. LLMs also proved useful for helping with syntax and enhanced productivity when debugging errors. They also found interesting correlations between coding skills and prior experience with AI playing a crucial role in the adoption of AI tools.

In my dissertation, I aim to address some of the identified usability issues, namely syntactic complexity, verification and reliability of generated code, by incorporating domain insights from software engineering into the training and evaluation process to produce effective models that make code generation more reliable and useful for developers.

Chapter 3

LowCoder for Syntactic Complexity

AI has demonstrated significant potential in empowering individuals with limited or no programming experience to write code. Copilot [6] and ChatGPT [7] are popular examples of tools that support *programming by natural language* (PBNL), where users can generate code from natural language instructions. However, a key problem with these tools is that they generate (potentially complex) code, which can be difficult to understand and manipulate. This is largely attributed to the syntactic complexity that is associated with traditional text based code.

This challenge is prominent among non expert programmers, as studies show those with limited coding proficiency are hesitant to use AI tools for code generation. This reluctance arises from the lack of necessary background knowledge and skills required to comprehend the correctness of AI-generated code and to modify it into a usable format [171]. Experienced programmers also face difficulties with AI-generated code, often discarding it when unexpected behavior occurs due to a lack of understanding. Some even find it more efficient to rewrite the code from scratch than invest time in comprehending and debugging the generated code [202].

One viable solution to overcome this is to abstract away the syntactic complexity associated with textual code, replacing it with more intuitive interfaces. Low-code programming [89] overcomes this by reducing the amount of textual code developers write by offering alternative programming interfaces. This has been embraced by software vendors to both democratize software development and increase productivity [182]. Most low-code offerings for building AI pipelines currently favor visual programming [35, 59, 80]. While visual programming helps users navigate complex pipelines, it poorly supports *discoverability* of API components (basic building blocks of code) in large APIs due to the large range of options and limited screen space [158].

At the intersection of these two paradigms, we propose LowCoder[1], the first low-code tool to combine visual programming with PBNL. We conjecture that the respective strengths of these two low-code techniques can compensate for each other's weaknesses. PBNL uses AI to help users retrieve and use programming constructs based on natural language queries. This does not always return correct programs, necessitating a way to help users understand and fix generated programs. Visual programming complements PBNL by providing a clear, unambiguous representation of the program that users can directly manipulate to experiment with alternatives. In other words, the use of *abstraction*, in this case through visual programming, can help overcome

[1]LowCoder [170] was published at the ACM Conference on Intelligent User Interfaces (IUI) 2024

the challenge of syntactic complexity in LLM generated code.

Most AI development today involves Python programming with popular libraries such as scikit-learn (sklearn) [159]. Unfortunately, writing code, even in a language as high-level as Python, is hard for *citizen developers* [110]—people who lack formal training in programming but nevertheless write programs as part of their everyday work. This is a fairly common situation for data scientists, among others. AI programming libraries also tend to be large and change regularly. Needing to remember hundreds of AI operators and their arguments slows down even professional developers.

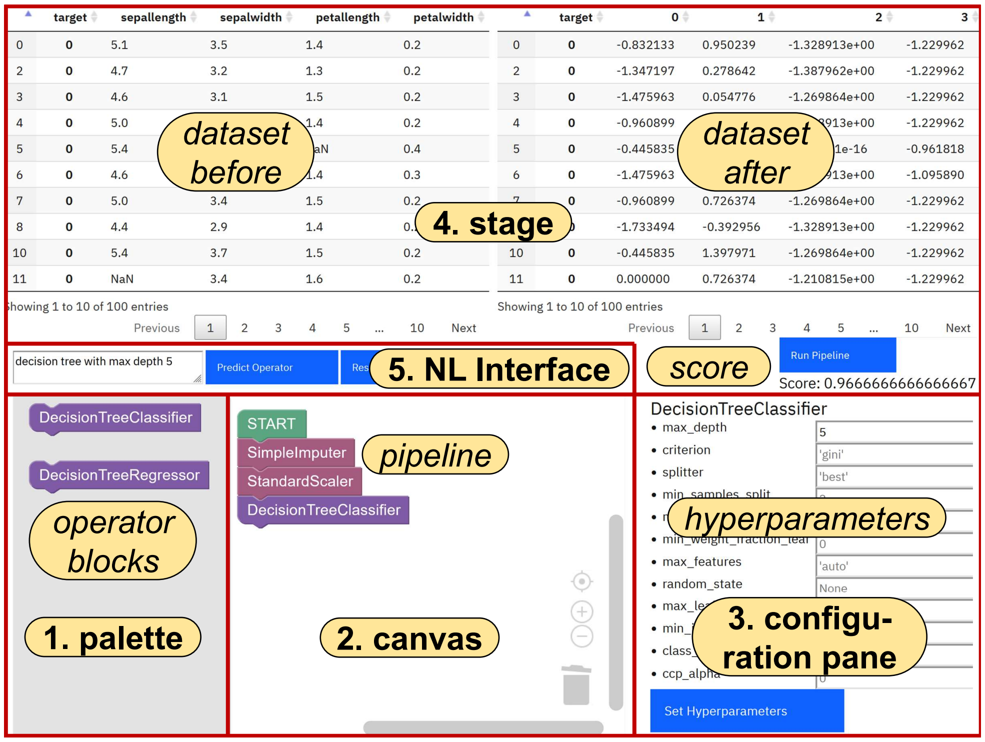
Our goal is to help people who know *what* they want to accomplish (e.g., build an AI pipeline) but face syntactic barriers from the programming language and library (the *how* part), perhaps due to a lack of formal programming training. End-users writing software face similar "design barriers" [110], where it is difficult to even conceptualize a solution. In contrast to other popular low-code domains such as traditional software [173], the domain of developing AI pipelines is particularly difficult in this regard due to its experimental nature where progress has a high degree of uncertainty [211]. We chose to target sklearn [159] because of its pervasive use, because visual programming naturally fits the pipeline structure of sklearn, and because PBNL is particularly useful in aiding recall of operators from the relatively large API of sklearn.

LOWCODER's visual programming component, LOWCODER$_{\text{VP}}$, lets users snap together visual blocks for AI operators into well-structured AI pipelines. It uses Blockly [158] to provide a Scratch-like [173] look-and-feel. The PBNL component, LOWCODER$_{\text{NL}}$, lets users enter natural language queries and predicts relevant operators, optionally configured with hyper-parameters. It uses a fine-tuned variant of the CodeT5 model [215] that we developed through experiments with a variety of neural models for program generation, ranging from training models from scratch to few-shot prompting large language models [151]. We further noticed that queries usually mention at most a subset of hyper-parameters for each pipeline step, so we developed a novel task formulation tailored to this use case that improved learning outcomes.

We leverage LOWCODER to provide some of the first insights into both how and when low-code programming and PBNL help developers with various degrees of expertise. We conduct a user study with 20 participants with varying levels of AI expertise using LOWCODER to complete four tasks, half of which with the help of the AI-powered component LOWCODER$_{\text{NL}}$. Overall, the combination of visual programming along with the natural language interface helped both novice and non-novice users to successfully compose pipelines (85% of tasks) and then further refine their pipelines (72.5% of tasks) when using LOWCODER$_{\text{NL}}$. Additionally, LOWCODER$_{\text{NL}}$ helped users discover previously-unknown operators in 75% of the tasks compared to just 32.5% using other methods like web search when LOWCODER$_{\text{NL}}$ was not available. In addition, despite being trained on a different dataset, LOWCODER$_{\text{NL}}$ accurately answered real user queries.

3.1 LOWCODER Tool Design

This work explores the intersection of visual programming and language models in an effort to understand the benefits and limitations of using the combination in low-code programming. We accomplish this by implementing and studying LOWCODER, a prototype low-code tool for building ML pipelines with sklearn operators for tabular data that includes both visual program-



	target	sepallength	sepalwidth	petallength	petalwidth
0	0	5.1	3.5	1.4	0.2
2	0	4.7	3.2	1.3	0.2
3	0	4.6	3.1	1.5	0.2
4	0	5.0	[illegible]	1.4	0.2
5	0	5.4	[illegible]	aN	0.4
6	0	4.6	[illegible]	1.4	0.3
7	0	5.0	3.4	1.5	0.2
8	0	4.4	2.9	1.4	0.
10	0	5.4	3.7	1.5	0.2
11	0	NaN	3.4	1.6	0.2

howing 1 to 10 of 100 entries

	target	0	1	2	3
0	0	-0.832133	0.950239	-1.328913e+00	-1.229962
2	0	-1.347197	0.278642	-1.387962e+00	-1.229962
3	0	-1.475963	0.054776	-1.269864e+00	-1.229962
4	0	-0.960899	[illegible]	913e+00	-1.229962
5	0	-0.445835	[illegible]	1e-16	-0.961818
6	0	-1.475963	[illegible]	913e+00	-1.095890
7	0	-0.960899	0.726374	-1.269864e+00	-1.229962
[illegible]	[illegible]	-1.733494	-0.392956	-1.328913e+00	-1.229962
10	0	-0.445835	1.397971	-1.269864e+00	-1.229962
11	0	0.000000	0.726374	-1.210815e+00	-1.229962

Showing 1 to 10 of 100 entries

decision tree with max depth 5

Score: 0.9666666666666667

DecisionTreeClassifier

• max_depth	5
• criterion	'gini'
• splitter	'best'
• min_samples_split	[illegible]
• [illegible]	[illegible]
• min_weight_fraction_leaf	0
• max_features	'auto'
• random_state	None
• max_le	[illegible]
• min_	[illegible]
• class_	[illegible]
• ccp_alpha	0

Figure 3.1: LowCoder interface with labeled components, described in the text.

ming (VP) and natural language (NL) modalities, which complement each other by mitigating the limitations of either modality separately. Building this tool provided us with the opportunity to examine the impact of both modalities on users. Figure 3.1 highlights the main features and inputs of LowCoder.

To support multiple low-code modalities, we follow the lead of projectional code editors [207] by adopting the model-view-controller pattern. Specifically, we treat visual programming as a read-write view, PBNL as a write-only view, and let users inspect data in a read-only view [89]. The tool keeps these three views in sync by representing the program in a domain-specific language (DSL). The domain for the DSL is AI pipelines. A corresponding, practical desideratum is that the DSL is compatible with sklearn [159], the most popular library for building AI pipelines, and is a subset of the Python language, in which sklearn is implemented, which also enables us to use AI models pretrained on Python code. The open-source Lale library [27] satisfies these requirements, and in addition, describes hyper-parameters in JSON schema format [162], which our tool also uses. The current version of our tool supports 143 sklearn operators. LowCoder$_{VP}$ uses a client-server architecture with a Python Flask back-end server and front-end based on the Blockly [158] meta-tool for creating block-based visual programming tools. The front-end converts the block-based representation to Lale which is then sent to the back-end. The back-end validates the given Lale pipeline using internal schemas, then evaluates the pipeline against a given dataset. The results of this evaluation (including any error messages) are returned to the front-end and presented to the user.

3.1.1 Visual Programming Interface

LowCoder$_{\text{VP}}$ is our block-based visual programming interface for composing and modifying AI pipelines. One goal that this tool shares with other block-based visual tools such as Scratch [173] is to encourage a highly interactive experience. The block visual metaphor allows for blocks that correspond to sklearn operators to be snapped together to form an AI pipeline. The shape of the blocks suggest how operators can connect. Their color indicates how they affect data: red for operators that transform data (with a *transform()* method) and purple for other operators that make predictions, such as classifiers and regressors (with a *predict()* method).

Figure 3.1 illustrates the interface. A *palette (1)* on the left side of the interface contains all of the available operator blocks. Blocks can be dragged-and-dropped from the palette to the *canvas (2)*. For ease of execution, our tool only allows for one valid pipeline at a time, so blocks must be attached downstream of the pre-defined *Start* block to be considered part of the active pipeline. Figure 3.1 shows an example of blocks defining a pipeline where the `SimpleImputer`, `StandardScaler`, and `DecisionTreeClassifier` blocks are connected to the *Start* block and each other. Input data are transformed by the first two operators (`SimpleImputer` and `StandardScaler`) and then sent to `DecisionTreeClassifier` for training and then scoring. Blocks not attached to the *Start* block are disabled but can be left on the canvas without affecting the execution of the active pipeline. Selected operator blocks also display a *hyper-parameter configuration pane (3)* on the right. The pane lists each hyper-parameter for an operator along with a description (when hovering over the hyper-parameter name) and default values along with input boxes to modify each hyper-parameter.

Our tool provides a *stage (4)* with *Before* and *After* tables to give immediate feedback with every input on how the current pipeline affects the given dataset. When a tabular dataset is loaded, the *Before* table displays its target column on the left and feature columns on the right. When a pipeline that transforms input data is executed, the *After* table shows the results of the transformations. At any time, a pipeline can be executed on the given dataset by pressing the “Run Pipeline” button. Executing a pipeline will attempt to train the given pipeline on the training portion of the given dataset and then return a preview of all data transformations on the training data in a second table. For instance, in the example shown in Figure 3.1, executing the pipeline with `SimpleImputer` and `StandardScaler` transforms data from the *Before* table by imputing missing values and standardizing all feature values in the *After* table. If training is successful, then the trained pipeline is scored against the test set and the score (usually accuracy) is displayed. LowCoder$_{\text{VP}}$ also encourages liveness [193] by executing the pipeline when either the active pipeline is modified or hyper-parameters are configured. For example, adding a `PCA` operator and setting the `n_components` hyper-parameter to 2 for the prior example will reduce the feature columns in the *After* table to 2. Hence, users receive immediate feedback on the effect of pipeline changes on the dataset without requiring separate training or scoring steps. This liveness encourages a high degree of interactivity [173].

3.1.2 Natural Language Interface

A potential weakness of visual low code tools is that users have trouble discovering the right components to use [110]. For instance, the palette of LowCoder$_{\text{VP}}$ contains more than a hun-dred operator blocks. Rather than requiring users to know the exact name of the operator or scroll through so many operators, we provide LowCoder$_{\text{NL}}$, which allows users to describe a desired operation in the *NL interface (labeled component 5 in Figure 3.1)* text box and press the "Predict Pipeline" button. The tool then infers relevant operator(s) and any applicable hyper-parameters using an underlying natural-language-to-code translation model and automatically adds the most relevant operator to the end of the pipeline. The palette is also filtered to only display any relevant operator(s) such as in Figure 3.1. Pressing the "Reset Palette" button will undo filtering (so the palette shows all available operators again) without clearing the active pipeline or canvas. Depending on the NL search, the automatically added operator may either have hyper-parameters explicitly defined or potentially relevant hyper-parameters highlighted. As an example, the NL search *"PCA with 2 components"* will automatically add the `PCA` operator where the `n_components` hyper-parameter is set to 2 and may highlight other hyper-parameters such as `random_state` for the user to consider setting. Section 3.2 describes the design and implementation of this model in detail. A potential weakness of natural language low-code tools is that the generated programs can be incorrect, due to a lack of clarity, or ambiguity, in the query, or a lack of context for the model providing inferences [21]. In comparison, visual inputs and representations are unambiguous [89], requiring no probabilistic interpretation, so users can easily understand and manipulate the results returned by LowCoder$_{\text{NL}}$.

To ground our evaluation of LowCoder$_{\text{NL}}$, we also provide a version of the tool without a trained language model to users in our study (described in Section 4.5). In this setting, the *NL interface (5)* text box becomes a simple substring keyword search that matches the query against operator names. For example, inputting *"classifier"* filters the palette to only display sklearn operators that contain *'classifier'* in the name such as `RandomForestClassifier` (but notably not all classifiers such as `SVC`). Hence, users receive immediate feedback on the effect of pipeline changes on the dataset without requiring separate training or scoring steps. This liveness encourages a high degree of interactivity [173].

3.1.3 Natural Language Interface

A potential weakness of visual low code tools is that users have trouble discovering the right components to use [110]. For instance, the palette of LowCoder$_{\text{VP}}$ contains more than a hun-dred operator blocks. Rather than requiring users to know the exact name of the operator or scroll through so many operators, we provide LowCoder$_{\text{NL}}$, which allows users to describe a desired operation in the *NL interface (labeled component 5 in Figure 3.1)* text box and press the "Predict Pipeline" button. The tool then infers relevant operator(s) and any applicable hyper-parameters using an underlying natural-language-to-code translation model and automatically adds the most relevant operator to the end of the pipeline. The palette is also filtered to only display any relevant operator(s) such as in Figure 3.1. Pressing the "Reset Palette" button will undo filtering (so the palette shows all available operators again) without clearing the active pipeline or canvas. Depending on the NL search, the automatically added operator may either have hyper-parameters explicitly defined or potentially relevant hyper-parameters highlighted. As an example, the NL search *"PCA with 2 components"* will automatically add the `PCA` operator where the `n_components` hyper-parameter is set to 2 and may highlight other hyper-parameters such as `random_state` for the user to consider setting. Section 3.2 describes the design and imple-

mentation of this model in detail. A potential weakness of natural language low-code tools is that the generated programs can be incorrect, due to a lack of clarity, or ambiguity, in the query, or a lack of context for the model providing inferences [21]. In comparison, visual inputs and representations are unambiguous [89], requiring no probabilistic interpretation, so users can easily understand and manipulate the results returned by LowCoder$_{\text{NL}}$.

To ground our evaluation of LowCoder$_{\text{NL}}$, we also provide a version of the tool without a trained language model to users in our study (described in Section 4.5). In this setting, the *NL interface (5)* text box becomes a simple substring keyword search that matches the query against operator names. For example, inputting *"classifier"* filters the palette to only display sklearn operators that contain *'classifier'* in the name such as `RandomForestClassifier` (but notably not all classifiers such as `SVC`).

3.2 Using Language Models for Low-Code

This section discusses the language modeling for LowCoder$_{\text{NL}}$.

3.2.1 Data Collection

Our goal is to make a large API accessible through a low-code tool by allowing users to describe *what* they want to do when they do not know *how*. More specifically, we want to enable users to build sklearn pipelines in a low-code setting, using a natural language interface that can be used as an *intelligent search* tool. This problem can be solved using language models that can be trained to translate a natural language query into the corresponding line of code [68]. However, such models heavily rely on data to learn such behaviour and would need to be trained on an aligned dataset of natural language queries and the corresponding sklearn line(s) of code demonstrating how a user would want to use such an intelligent search tool. Naturally, we cannot collect such a dataset without this tool, creating a circular dependency. To overcome this challenge, we curate a *proxy dataset* using 140K Python Kaggle notebooks that were collected as part of the Google AI4Code challenge.[2] From these notebooks, we extracted aligned Natural Language (NL) & Code cells related to machine learning and data science tasks. While the distribution of the NL in the markdown cells is not completely representative of the NL queries that users would enter in the low-code setting, they provide the model with a broad range of such examples. Results in Section 3.3.1 show that this is indeed effective.

3.2.2 Data Preprocessing

We first filter out notebooks that do not contain any sklearn code. This leaves 84,783 notebooks – evidently, many notebooks involve sklearn. We further filter out notebooks with non-English descriptions in all of the markdown cells, resulting in 59,569 notebooks. We then create a proxy dataset by extracting all code cells containing sklearn code and pairing these with their preceding NL cell to get a total of 211,916 aligned NL-code pairs. We remove any duplicate NL-code

[2]https://www.kaggle.com/competitions/AI4Code

pairs, leaving 102,750 unique pairs. For each code cell, we then extract the line(s) of code corresponding to an sklearn operation invocation statement.

We discard any code cells that do not include sklearn operation invocation statements but include other sklearn code, leaving a final total of 79,372 NL-Code pairs. We separate these into train/validation/test splits resulting in 64,779 train samples, 7,242 validation samples, and 7,351 test samples.

3.2.3 Tasks

Table 3.1: Task formulations highlighting the code components: mask, operator name, hyper-parameter name, hyper-parameter value. The Hybrid Operator Invocation setting does not mask 'balanced' as it appears in the query.

Task Formulation	**Code for the NL query:** ***Random forest with balanced class weight***
Operator Name	`RandomForestClassifier`
Complete Operator Invocation	`RandomForestClassifier(n_estimators=100, class_weight='balanced')`
Masked Operator Invocation	`RandomForestClassifier(n_estimators=MASK, class_weight=MASK)`
Hybrid Operator Invocation	`RandomForestClassifier(n_estimators=MASK, class_weight='balanced')`

Given the NL query, our model aims to generate a line of sklearn code corresponding to an operation invocation that can be used to build the next step of the pipeline. We consider a range of formulations of the task with different levels of details, as illustrated in Table 3.1.

Operator Name Generation

The simplest task is generating only the operator name from the NL query. This alone can significantly help a developer with navigating the extensive sklearn API. We process the aligned dataset to map the query to the name(s) of operator(s) invoked in the code cell, discarding any other information such as hyper-parameters.

Complete Operator Invocation Generation

At the other extreme, we task the model with synthesizing the complete operation invocation statement, including all the hyper-parameter names and values. Preliminary results (discussed in Section 3.3.1) show that the model often makes up arbitrary hyper-parameter values, resulting in lines of code that can rarely be used directly by developers.

Masked Operator Invocation Generation

In this scenario, we mask out all the hyper-parameter values from the invocation statement, keeping only their names. The goal of this formulation is to ensure that the model learns to predict the specific invocation signature, even if it is unaware of the values to provide for the hyper-parameters.

Hybrid Operator Invocation Generation (HOI)

Manual inspection of the NL-code pairs revealed that the queries sometimes explicitly describe a subset of the hyper-parameter names and values to be used in the code. When this is the case, the model has the necessary context to predict at least those hyper-parameter values. Supporting this form of querying enables users to express the most salient hyper-parameters up-front. Therefore, we formulated a new hybrid task, where we keep the hyper-parameter values if they are explicitly stated in the NL query and mask them otherwise. This gives the model an opportunity to learn the hyper-parameter names and values if they are explicitly stated in the description, and unburdens it from making up values that it lacks the context to predict by allowing it to generate placeholders (masks) for them.
Evaluation: To evaluate the feasibility of predicting code using the different task formulations, we train a simple sequence-to-sequence model (detailed in Section 3.2.4) and compare the results for the various training tasks in Section 3.3.1. We find HOI to be the most accurate/reliable formulation for our setting. We therefore proceed to use this task formulation for training the models.

3.2.4 Modeling

All tasks from Section 4.2 are sequence-to-sequence tasks. We compare and contrast three different deep learning paradigms for this type of task, illustrated in Figure 3.2: 1) train a standard sequence-to-sequence transformer *from scratch*, 2) fine-tune (calibrate) a pretrained "medium" sized model, 3) query a Large Language Model (LLM) by means of few-shot prompting [165]. We elaborate on these models below. Note that we use top-k sampling for our top-5 results.

Transformer (from scratch)

We train a sequence-to-sequence Transformer model [203] with randomly initialized parameters on the training data. Our relatively small dataset of ca. 70K training samples limits the size of a model that can be trained in this manner. We use a standard model size, with 6 encoder and decoder layers and 512-dimensional attention across 8 attention heads and a batch size of 32 sequences with up to 512 tokens each. We use a sentence piece tokenizer (trained on Python code) with a vocabulary size of 50K tokens. The model uses an encoder-decoder architecture that jointly learns to encode (extract a representation of) the natural language sequence and decode (generate) the corresponding sklearn operator sequences.

Fine-tuning CodeT5

CodeT5 is a pretrained encoder-decoder transformer model [215] that has shown strong results when fine-tuned on various code understanding and generation tasks [134]. CodeT5 was pretrained on a corpus of six programming languages from the CodeSearchNet dataset [95] and fine-tuned on several tasks from the CodeXGLUE benchmark[134] in a multi-task learning setting, where the task type is prepended to the input string to inform the model of the task. We fine-tune CodeT5 on the HOI generation task by adding the 'Generate Python' prefix to all NL

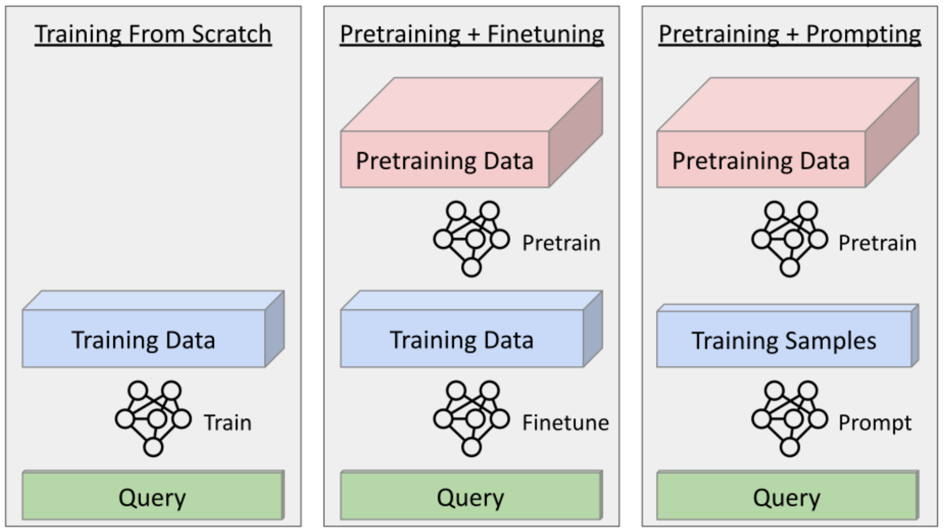


Figure 3.2: Overview of the “trifecta” of training approaches used in contemporary deep learning: smaller models are directly trained from scratch on downstream task data; medium sized models (100M-1B parameters) are pretrained with a generic training signal and then fine-tuned on task data; large models (>1B parameters) are only pretrained on very large datasets and are prompted with examples from the training data as demonstration followed by the query.

queries. We experiment with different size CodeT5 models: codet5-small (60M parameters), base (220M), and large (770M).

Few-Shot Learning With CodeGen

Lastly, we explore large language models (LLMs) that are known to perform well in a task-agnostic few-shot setting [42]. More specifically, we look at CodeGen, a family of LLMs that are based on standard transformer-based autoregressive language modeling [151]. Pretrained CodeGen models are available in a broad range of sizes, including 350M, 2.7B, 6.1B and 16.1B parameters. These were all trained on three different datasets, starting with a large, predominantly English corpus, followed by a multi-lingual programming language corpus, and concluding with fine-tuning on just Python data, which we use in this work. The largest model trained this way was shown to be competitive with Codex [47] on a Python benchmark [151].

Models at this scale are expensive to fine-tune and are instead commonly used for inference by means of “few-shot prompting” [165]. LLMs are remarkably capable of providing high-quality completions given an expanded prompt containing examples demonstrating the task [42]. We prompt our model with 5 such NL-code examples. Figure 3.3 illustrates an example prompt with 3 such pairs. The model does in-context learning on the examples in the prompt and completes the sequence task, which results in generating the HOI code.

```
NL: Build a simple linear support vector classification
Code: SVC(kernel='linear', random_state=MASK)

NL: PCA with 2 components
Code: PCA(n_components=2)

NL: Put the column median instead of missing values
Code: SimpleImputer(missing_values=MASK, strategy='median')

NL: [Enter query here]
Code:
```

Figure 3.3: Example of a few (3) shot prompting template for querying a large language model in our study.

3.3 Evaluation

This section describes the evaluations for the language modeling that enables LowCoder$_{\text{NL}}$ along with the user studies that we conducted to analyze the benefits and challenges of using low-code for developing AI pipelines using LowCoder.

3.3.1 Modeling

Experimental Setup

All of our models are implemented using PyTorch transformers and the HuggingFace interface. We use the latest checkpoints of the CodeT5 [215] and CodeGen [**?**] models. Our models were trained on a single machine with multiple 48 GB NVIDIA Quadro RTX 8000 GPUs until they reached convergence on the validation loss. We clip input and output sequence lengths to 512 tokens, but reduce the latter to 64 when using the model in LowCoder to reduce inference time. We find in additional experiments that since few predictions are longer than this threshold, this incurs no significant decrease in accuracy, but speeds up inference by 34%. We use a batch size of 32 for training and fine-tuning all of our Transformer and CodeT5 models, except for CodeT5-large, for which we used a batch size of 64 to improve stability during training.

Test Datasets

To ensure a well-rounded evaluation, we look at two different test datasets.
(i) Test data (from notebooks) - We use the NL-code pairs from the Kaggle notebooks we created in Section 3.2.2 containing 7,351 samples. These are noisy – some samples contain vague and underspecified Natural Language (NL) queries, such as - *"Data preprocessing"*, *"Build a model"*, *"Using a clustering model"*. Others contain multiple operator invocation statements corresponding to a single NL query, even though the NL description only mentions one of them, e.g., *"Model # 2 - Decision Trees"* corresponds to `DecisionTreeClassifier()` and `confusion_matrix(y_true, y_pred)`. Furthermore, these samples were collected from Kaggle notebooks, so the distribution of the NL queries collected from the markdown cells are not necessarily representative of NL queries that real users may enter into LowCoder$_{\text{NL}}$.

(ii) Real user data - We log all the NL queries that users searched for in LowCoder during the user studies along with the list of operators that the model returned. This gives us a more accurate distribution of NL queries that developers use to search for operators in LowCoder$_{\text{NL}}$. We obtained a total of 218 samples in this way, which we then manually annotated to check whether (i) the predictions were accurate, that is, if the operators in any of the predictions matches the inferred intent in the query and (ii) the NL query was clear, with an inter-rater agreement of 97.7% and a negotiated agreement [74] of 100%.

Test Metrics

We use both greedy (top-1) and top-K (top-5) decoding when generating the operator invocation sequences for each NL query. We evaluate the models' ability to generate just the operator name as well as the entire operator invocation (including all the hyper-parameter names and values) based on the hybrid formulation.

Task Comparison

We first train a series of randomly initialized 6-layer Transformer models from scratch on each task formulation from Section 4.2. We compare the model's ability to correctly generate the operator name and the operator invocation based on the formulation corresponding to the training task using top-1 and top-5 accuracy as shown in Figure 3.4. We find that the hybrid formulation of the operation invocation task, while challenging, is indeed feasible and allowed the model to achieve reasonably strong performance when generating the entire operation invocation statement. Contrary to the other task formulations, a model trained with the HOI signal also achieved comparable performance to the model trained solely on operator names when evaluated purely on operator name prediction (ignoring the generated hyper-parameter string). These results highlight that the hybrid representation helps the model learn by unburdening it from inferring values that it lacks the context to predict.

Model Comparison

We next evaluate the performance of the trifecta of modeling strategies from Section 3.2.4 on the task of Hybrid Operation Invocation (HOI) generation. We benchmark across different model sizes and compare the performance for both operator name and operator invocation generation using top-5 accuracy in Figure 3.5. The results show that the 0.77B parameter fine-tuned CodeT5 is the best performing model with an accuracy of 73.57% and 41.27% on the test data for the operation name and operation invocation generation respectively. The 0.22B parameter fine-tuned CodeT5 model has comparable performance, but its inference time is approximately 2–3 seconds faster than the 0.77B fine-tuned CodeT5 model, making it more desirable for integration with the tool.

Performance in Practice

Up to this point, all our evaluations have been based on the proxy dataset from Kaggle. To get a better idea of the model's performance in the real world, we further evaluate the performance of

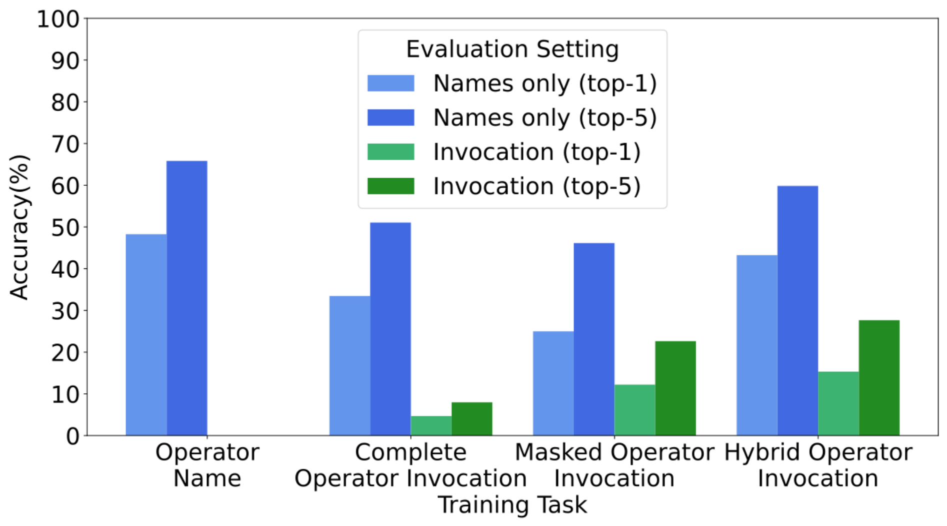


Figure 3.4: Accuracy of Transformer models trained from scratch on various task formulations. 'Invocation' test results refer to the specific invocation formulation of the training task, while 'Names only' just considers whether the generated code starts with the correct operator name. Only the Hybrid Operator Invocation setting yields useful quality on both tasks.

the fine-tuned 0.22B parameter CodeT5-base from the tool on real user data that was collected during the user studies. The distribution of NL queries collected from the user studies represents the "true" distribution of queries that can be expected from users in a low-code setting. Out of the 218 samples that were collected, we found only one sample in which a user explicitly specified a hyper-parameter value in their query. We therefore only compute the accuracy of the operation name generated rather than the entire operation invocation (as they would use default values anyway and so the scores remain the same except for that one sample).

Out of 218 query requests, the fine-tuned CodeT5-base model that was used in our tool answered 150 queries correctly, which would suggest an overall accuracy of 68.8%. However, 33 of these requests targeted actions that are not supported by the sklearn API, such as dropping a column (commonly the territory of the Pandas library). Disregarding such unsupported usage, LowCoder$_{\text{NL}}$ answered 141 out of 185 queries correctly for an overall accuracy of **76.2%**. For 33 additional samples, neither annotator could infer a reasonable ground truth since the prompt was unclear (e.g.: "empty"). Leaving these out, i.e., when the prompt is both clear *and* the operator is supported by the tool, LowCoder$_{\text{NL}}$ was accurate in over **90%** (137/152) of completions

3.3.2 User Study

We conducted a user study with 20 participants with varying levels of AI expertise to create AI pipelines using LowCoder across four tasks, replacing LowCoder$_{\text{NL}}$ with a simple keyword search in half the tasks. We collect and analyze data to investigate the following research questions:

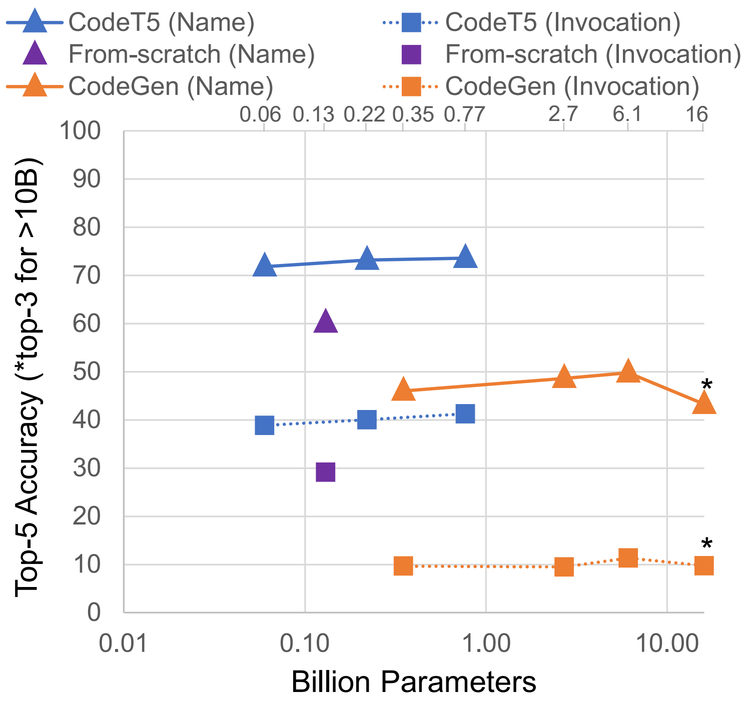


Figure 3.5: Accuracy vs. model size based on top-5 sampling. (*The 16B CodeGen uses top-3 due to memory constraints.) We compare the three modeling paradigms, namely training transformer from scratch, finetuning CodeT5, and fewshot prompting CodeGen, on both Operator Name generation and Hybrid Operator Invocation generation.

RQ1: How do LowCoder$_{\mathrm{NL}}$ and other features help participants discover previously-unknown operators?

RQ2: Are participants able to compose and then iteratively refine AI pipelines in our tool?

RQ3: What are the benefits and challenges of integrating language models with visual programming for low-code?

Study Methodology

We recruited 20 participants within the same large technology company via internal messaging channels. We expect that citizen developers without formal programming training may also have varying levels of AI expertise and intentionally solicited participants of all backgrounds. Potential participants filled out a short pre-study survey to self-report experience in the following: machine learning, data preprocessing, and sklearn using a 1 (no experience) to 5 (expert) scale. Participants include a mix of roles including developers, data scientists, and product managers working in a variety of domains such as AI, business informatics, quantum computing, and software services. 25% of the participants are female and the remaining 75% are male. 40% of the participants self-reported being novices in machine learning by indicating a 1 or 2 in the pre-study survey.

The study design is within-subjects [55] where each participant was exposed to two conditions: using LowCoder with (*NL condition*) and without (*keyword condition*) the natural language (NL) interface powered by LowCoder$_{\text{NL}}$. The keyword condition used a simple substring filter for operator names. Each participant performed four tasks total across the two conditions. For each task, participants were instructed to create AI pipelines with data preprocessing and classifier steps on a sample dataset with as high a score (accuracy on the test set) as possible during a time period of five to ten minutes. Each sample dataset was split beforehand into separate train and test sets. Tasks were open-ended with no guidance on what preprocessing steps or classifiers should be used.

There were four sample datasets in total and each participant was exposed to all four. The sample datasets are public tabular datasets from the UCI Machine Learning Repository [64]. Two of the tasks (A and D) require a specific data preprocessing step in order to successfully create a pipeline while two (B and D) technically do not require preprocessing to proceed. For each participant, the order of the conditions and the order of the tasks were shuffled such that there is a uniform distribution of the order of conditions and tasks.

As our study included machine learning novices, we gave each participant a short overview of the basics of machine learning with tabular datasets and data preprocessing. We avoided using specific terms or names of operators in favor of more general descriptions of data-related problems.

We then gave each participant an overview of LowCoder. To mitigate potential biasing or priming, the tool overview used a fifth dataset from the UCI repository [64]. To avoid operators that were potentially useful in user tasks, the overview used both a non-sklearn operator that was not available in the study versions of the tool as well as sklearn's DummyClassifier that generates predictions without considering input features. Participants were allowed to use external resources such as web search engines or documentation pages. Nudges were given by the study administrators after five minutes if necessary to help participants progress in a task. Nudges were in the form of reminders to use tool features such as the NL interface, external resources, or to include missing steps such as data preprocessing or classifiers. Nudges did not mention specific operator names nor guidance on specific actions to take.

For each version of the tool, study administrators would describe the unique features of the particular version and then have participants perform tasks using two out of four sample datasets. After performing tasks using both versions of the tool and all four sample datasets, participants were asked to provide open-ended feedback and/or reactions for both LowCoder and the comparison between the NL and keyword conditions.

Data Collection and Analysis

To answer our research questions, for each participant, we collect and analyze both quantitative and qualitative data. For quantitative data, we report on the incidence of participants discovering a previously-unknown operator (RQ1) and the incidence of completing the task and iterating or improving the pipeline (RQ2). We consider an operator 'previously-unknown' if the participant found and used the operator without using the exact or similar name. For example, using an NL query such as *"deal with missing values"* to find the `SimpleImputer` operator is considered discovering a previously-unknown operator while a query such as *"simpleimpute"* is not. We report discovery using the following methods: through LowCoder$_{\text{NL}}$, generic web search engine (Google), and scrolling through the palette. Participants may discover multiple unknown

Table 3.2: Incidence of tasks where participants find previously-unknown operators per condition (40 tasks for all, 16 tasks by novices, and 24 by non-novices). Note that rows may not sum to 100% as participants can use multiple methods to discover operators for a given task or not discover operators at all.

Condition	**Participant**	**Method of Discovery**		
		LowCoder$_{\text{NL}}$	Web search	Palette
NL	All	30 (75.0%)	5 (12.5%)	5 (12.5%)
	Novice	8 (50.0%)	2 (12.5%)	4 (25.0%)
	Non-Novice	22 (91.7%)	3 (12.5%)	1 (4.2%)
Keyword	All	*Not available in this condition.*	13 (32.5%)	11 (27.5%)
	Novice		3 (18.8%)	5 (31.3%)
	Non-Novice		10 (41.7%)	6 (25.0%)

operators during the same task, possibly using different methods. For each participant's task, we consider it 'complete' if the composed pipeline successfully trains against the dataset's training set and returns a score against the test set. We consider the pipeline iterated if a participant modifies an already-complete pipeline. More specifically, we consider the following forms of iteration: a preprocessing operator block is added or swapped, a classifier block is swapped, or hyperparameters are tuned. We report each of these as separate types of pipeline iteration. Participants may perform multiple types of iteration during the same task. Both sets of quantitative metrics are counted per task (80 tasks total for 20 participants, 40 tasks per condition).

We use qualitative data to answer RQ3. This data focuses on the participants' actions in LowCoder, commentary while using the tool and performing tasks, and answers to open-ended questions after the study. Specifically, the same two authors that administered the user study analyzed the notes generated by the study along with the audio and screen recordings when the notes were insufficient, using discrete actions and/or quotations as the unit of analysis. The first round of analysis performed open coding [55] on data from 16 studies to elicit an initial set of 73 themes. The two authors then iteratively refined the initial themes through discussion along with identifying 13 axial codes which are summarized in Figure 3.6. The same authors then performed the same coding process on a hold-out set of 4 studies. No additional themes were derived from the hold-out set of studies, suggesting saturation.

Study Results

We answer RQ1 and RQ2 using quantitative data collected from observing participant actions per task and answer RQ3 through open coding of qualitative data.

RQ1: How do LowCoder$_{\text{NL}}$ and other features help participants discover previously-unknown operators?

A known limitation of visual programming is discoverability [158]. Table 3.2 reports how often participants discovered previously-unknown operators during their tasks. 80% of the participants discovered an unknown operator across 63.8% of all 80 tasks in the study. Participants discovered unknown operators in 82.5% of the 40 NL condition tasks compared to 45% of the 40 keyword condition tasks. The odds of discovering an unknown operator are significantly greater in the NL condition than keyword ($p \ll 0.001$) using Barnard's exact test. We examine the methods of discovery in more detail, noting that LowCoder$_{\text{NL}}$ is only available in the NL condition whereas web search and scrolling through the operator palette are available in both conditions. Participants were not able to use the keyword search to discover unknown operators due to needing at least part of the exact name. Using LowCoder$_{\text{NL}}$, participants discovered unknown operators in 75% of tasks in the NL condition as opposed to an average of 22.5% using web search engines (12.5% in the NL condition and 32.5% in the keyword condition) and an average of 20% by scrolling through the operator palette (12.5% in the NL condition and 27.5% in the keyword condition). Within the NL condition, the odds of an unknown operator being discovered are significantly greater using LowCoder$_{\text{NL}}$ as opposed to both web search ($p \ll 0.001$) and scrolling ($p \ll 0.001$). When splitting on the experience of the participant, we find statistically greater chances of novices discovering operators in the NL condition using LowCoder$_{\text{NL}}$ as opposed to web search (p=0.013) but not scrolling (p=0.086). Non-novices were significantly more likely to discover operators using LowCoder$_{\text{NL}}$ compared to web search or scrolling ($p \ll 0.001$, $p \ll 0.001$). Results do not change if considering web searches or scrolling across all 80 tasks. These results suggest that LowCoder$_{\text{NL}}$ is particularly helpful in discovering previously-unknown operators, especially compared to web search, but novices still face some challenges. We discuss these challenges in RQ3.

RQ2: Are participants able to compose and then iteratively refine AI pipelines in our tool?

Machine learning development is intensely iterative [211] and tools should support this. Table 3.3 reports how often participants iterated on pipelines. Participants completed 82.5% of the 80 tasks in the study and further iterated their pipelines in 72.5% of the tasks. Splitting on condition, the NL condition has 85% task completion and 72.5% further iteration while the keyword condition has 80% task completion and 72.5% iteration rate. Swapping classifiers was the most common form of iteration at 48.8%, followed by adding or swapping preprocessors at 43.8% and setting hyper-parameters at 30%. Comparing novices to non-novices, both types of participants are mostly successful in iterating pipelines with no significant differences in iteration rate using Barnard's exact test (p=0.109). This result holds when iterating preprocessors (p=0.664) but not classifiers (p=0.038) nor hyper-parameters (p=0.005). Non-novices are more likely to complete the task than novices (p=0.002). Regardless of experience, both novices and non-novices are able to iteratively refine their pipelines, but novices face some challenges compared to non-novices regarding actually completing the task. These challenges are discussed in the next research question.

RQ3: What are the benefits and challenges of integrating language models with visual programming for low-code?

Figure 3.6 shows our 13 axial codes for answering RQ3. These codes broadly represent three overarching themes regarding combining visual programming and language models for low-code:

1) *Discovery* of machine learning operators relevant for the task at hand, 2) *Iterative Composition* of the operators in the tool, and 3) *Challenges* that participants, particularly novices, face regarding working with machine learning and/or using low-code tools. We also collect *Feedback* from participants to inform future development of LowCoder. Due to space limitations, we

Table 3.3: Incidence of tasks where participants complete and iterate on preprocessors, classifiers, and hyper-parameters.

Iteration Type	***Total Tasks* (80)**	***Novice* (32)**	***Non-Novice* (48)**
Task Completion	66 (82.5%)	21 (65.6%)	45 (93.8%)
Swap Classifier	39 (48.8%)	11 (34.4%)	28 (58.3%)
Add/Swap Preprocessors	35 (43.8%)	15 (46.9%)	20 (41.7%)
Set Hyper-parameters	24 (30.0%)	4 (19.0%)	20 (41.7%)
All Iterations	58 (72.5%)	20 (62.5%)	38 (79.2%)

only report on a selection of the 13 axial codes and 73 codes derived from open coding.

For the first category of **Discovery**, our analysis derived two axial codes related to the participants' goal while attempting to discover operators: 1) *Know "What" Not "How"* where participants have a desired action in mind but do not know the exact operator that performs that action (19 out of 20 participants experienced this axial code) and 2) *Know "What" And "How"* where participants have a particular action and operator in mind (18/20). We dive deeper into *Know "What" Not "How"* which includes the code where participants *Discover a previously-unknown operator using NL* (16/20). We found in RQ1 that LOWCODER$_{\text{NL}}$ was helpful in finding unknown operators compared to other methods. The qualitative data suggests that participants were able to find unknown operators using LOWCODER$_{\text{NL}}$ during cases where they have an idea of the action to perform but do not know the exact operator name for a variety of reasons. For example, when discovering `SimpleImputer` with LOWCODER$_{\text{NL}}$, P11 noted that they *"never used SimpleImputer but had an idea of what I wanted to do, even though I generally remove NaNs in Pandas."* Another example is P16 who *"preferred the [NL version of* LOWCODER *], even when I was doing Google searches, they... didn't give me options, your tool at least returns some options that I can try out and swap out."* As a novice, P16 had difficulties finding the names of useful operators from web search results as opposed to the LOWCODER$_{\text{NL}}$ which directly returned actionable operators. Challenges regarding general web search is also an axial code.

For the second category of **Iterative Composition**, we derived four axial codes related to participant behaviors while attempting to compose and iterate on pipelines: 1) *General Exploratory* (13/20) iteration, 2) Exploratory iteration but where participants will select operators or hyper-parameters seemingly at *Random* (18/20), 3) *Targeted* (19/20) iteration where participants select operators or hyper-parameters with a particular intent, and 4) *Seeking Documentation* (15/20) where participants search for documentation to inform iteration decisions. For both forms of Exploratory iteration and Targeted iteration, we find examples of participants iterating classifiers, preprocessors, and hyper-parameters. For the axial code of seemingly *Random* iteration, participants, especially (but not exclusively) novices, when unsure of how to proceed, tended to try out arbitrary preprocessors or classifiers. This was more common for more difficult tasks that required particular data preprocessing to proceed. For example, non-novice P9 remarked *"I'm not familiar enough with it, so do I Google it or brute force it? [...] I don't even know what to Google to figure this out... I guess I'll do some light brute-forcing"* and proceeded to swap in and out preprocessors from the palette. In contrast, the axial code of *Targeted* (19/20) iteration has codes that reflect particular intentions that participants derived from

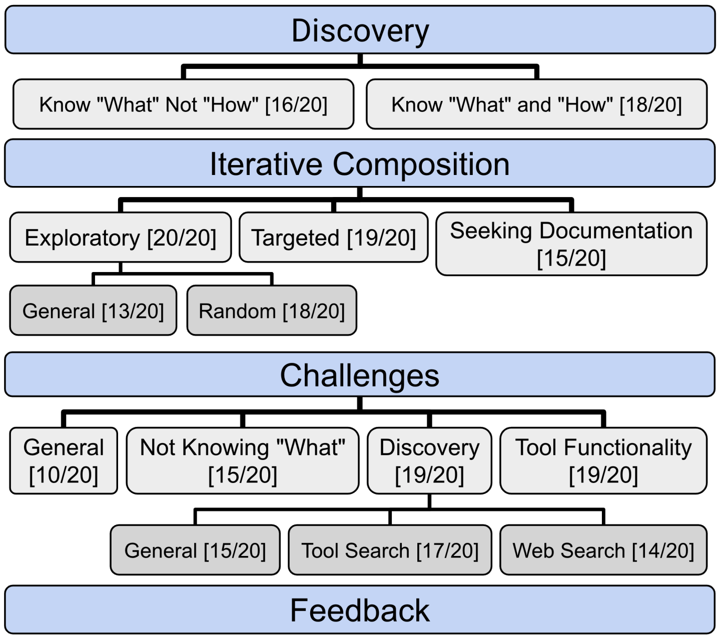


Figure 3.6: Axial codes from our qualitative analysis.

observations within the tool, such as *Noticing error messages* (10/20) or *Making use of data tables in task* (14/20). As an example of the data tables case, P11 realized through the *Before* data table that the given dataset had *"too many columns"* and added the `IncrementalPCA` operator along with setting its `n_components` hyper-parameter to 5. Upon seeing the change in data in the *After* data table, they remarked, *"Wow... I really like that I can see all the hyper-parameters that I can play with"* and proceeded to tune various hyper-parameters.

The third category is the variety of **Challenges** that participants faced while using LowCoder and performing the machine learning tasks, where we derive six axial codes: 1) *General* challenges (10/20) faced by participants that are not particular to our tool or tasks, 2) *Not Knowing "What"* (15/20) where participants experienced difficulties due to knowing neither "what" nor "how" to begin, 3) *General Discovery* challenges (15/20), 4) Discovery challenges around using *Web search* (14/20), 5) Discovery challenges when using *Tool search* (17/20) or specifically using LowCoder$_{\text{NL}}$, and 6) *Tool Functionality* (19/20) which describes challenges participants faced using (or not using) LowCoder features. We dive deeper into the axial code of *Not Knowing "What"* and note its contrast to the *Know "What" Not "How"* axial code where participants may have intentions but not know how to execute them or the *Exploratory* iteration axial code where participants may not have specific intentions but know how to iterate. All novices (8/8) and most non-novices (7/12) experienced this challenge. The primary code is that participants *Did not know "what" they wanted to do* (11/20). One possible cause of this lack of progression is choice paralysis, for example on P17's first task, *"first things first, I don't even know where to begin... right now it's super overwhelming, I guess I'll start throwing stuff in there."* We also describe the axial code of *Tool search* (17/20) where participants had difficulties forming search queries for LowCoder$_{\text{NL}}$.

Participants noted that despite the interface being intended for general natural language, the interface still *Needed a specific vocabulary* (8/20). As P19, a novice, described it, *"I get the idea of how it's supposed to work but it's hit and miss... even if I use very layman's terms... it expects a non-naive explanation of what needs to be done."* Part of this challenge may be due to a mismatch in the natural language in Kaggle notebooks used to train LowCoder$_{\text{NL}}$ and the language used by novices.

3.4 Reflection of Practical and Societal Impact

Our results show that the integration of LowCoder$_{\text{VP}}$ with LowCoder$_{\text{NL}}$ was helpful with aspects like operator discovery (RQ1) or iteratively composing pipelines (RQ2), even for novice participants. Through our work, we hope to help with the democratization of AI by supporting users with varying levels of AI expertise. LowCoder is especially useful for citizen developers who have an idea of *what* they would like to do but do not fully know *how* to accomplish that, perhaps due to a lack of formal programming training. In fact, our qualitative analysis (RQ3) reveals that a number of our participants (including all novices who participated) struggled with knowing *what* to do. End-users writing software face similar "design barriers" [110], where it is difficult for a non-programmer to even conceptualize a solution. In contrast to other popular low-code domains such as traditional software [173], the domain of developing machine learning pipelines is particularly difficult in this regard due to its experimental nature, where progress has a high degree of uncertainty [211]. This uncertainty then requires an abundance of judgment calls that rely heavily on prior machine learning experience [88] that novices lack. Some participants in our studies echo this, identifying that some ML knowledge is necessary to use our tool. That suggests that our low-code approach may be best-suited for citizen developers who have some domain knowledge but lack programming training, such as statisticians for the low-code domain of machine learning. A further improved low-code machine learning tool could thus be made more suitable towards novice citizen developers by guiding them to discover the *what* along with the *how*, i.e., by helping developers acquire the necessary ML knowledge.

Assisting novices without domain knowledge may then require low-code approaches that are orthogonal to both visual programming and language models. One such approach, suggested by a study participant, is to provide suggestions in the form of templates or recipes for pipelines. These suggestions could also be contextual to the given dataset or active pipeline, for example automatically suggesting encoders when detecting categorical features. Ko et al. [110] also suggest templates as a possible solution for design barriers. A related suggestion made by a number of our study participants is data visualization and summarization for the given dataset, such as plots, charts, confusion matrices, etc. These visualizations could themselves inform contextual suggestions – a histogram detecting a non-standard distribution may suggest the need for a `StandardScaler`. These contextual suggestions may also help in guiding developers in *what* to do, making for a more generally useful low-code tool for both citizen and experienced developers alike. Additionally, some visual programming languages risk vendor lock-in; we avoid that problem by backing LowCoder$_{\text{VP}}$ with a pre-existing, open-source DSL with the Lale library.

Threats to Validity: The user study for LowCoder has several limitations. The study focused on relatively small, public tabular datasets and sklearn operators and may not be indicative of other machine learning tasks such as deep learning on large datasets. Participants also all come from the same large technology company and may not be representative of general users. However, we did intentionally elicit participation from a variety of groups and experience levels to mitigate this. As our user study has a within-subjects design, there may be potential learning effects between tasks and conditions. In fact, we observed some cases of this (8/20), with some participants explicitly mentioning selecting particular operators due to the previous task. We mitigated this learning effect by randomizing the order of tasks and conditions, as well as by having two tasks (A and D) require the use of preprocessing operators that were not applicable to other tasks.

3.5 Related Work

Low-code: In adopting a visual programming approach to low-code, we follow a long tradition [38]. We were particularly inspired by Scratch, a popular visual programming environment for children that uses lego-like connected blocks [173]. Our other inspiration came from projectional editors, where the visual programming interface is a projection, or *view*, over an internal domain-specific language (DSL) [207]. Our implementation uses Blockly, a meta-tool for creating block-based visual programming tools [158], and Lale, a DSL for machine-learning pipelines [27].

Visual programming for AI: Most low-code interfaces for programming AI pipelines use visual programming. Examples include WEKA [80], Orange [59], and KNIME [35]. Each has a palette of operators that can be dragged onto a canvas, where they can be connected into a boxes-and-arrows style diagram. Commercial low-code visual interfaces follow the same approach, such as Vertex AI, Sagemaker, AzureML, and Watson Studio. A related approach for low-code ML pipeline development is automated machine learning (AutoML) [195], which is also used by many of the same commercial AI interfaces mentioned earlier. These tools tend to have a black-box approach where the user has little control over the AutoML search and may not even see the resulting pipeline. AutoML libraries such as auto-sklearn [69], TPOT [153], and hyperopt [33] provide a Python interface, which is intended for textual code development. There are also natural-language interfaces for professional developers based on large language models such as GitHub Copilot which uses Codex [47] and ChatGPT. Since these support APIs for which there is sufficient publicly available code to use as training data, they cover popular AI libraries such as sklearn. The main difference between these low-code tools for AI and our paper is that we combine the ease-of-use of visual programming with a natural language interface to help users discover and configure operators and, inspired by Scratch [173], our tool encourages liveness [193] through immediate user feedback for each user input into the system. This contrasts with most tools that require explicit training and scoring steps for feedback. Figure 3.7 summarizes the relationship between LowCoder and other low-code for AI tools.

Using AI for low-code development: The most prominent AI technique for low-code is programming by natural language (PBNL). When Androutsopoulos et al. surveyed natural language interfaces to databases in 1995, it was already a well-established field [21]. Desai et al. treat PBNL as a program synthesis problem targeting a DSL designed for the purpose [61]. The

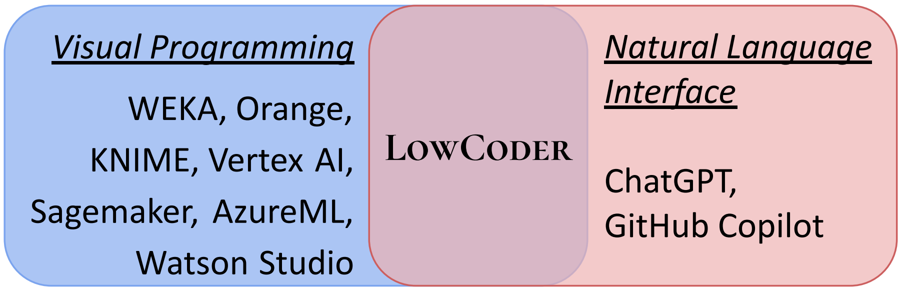

Visual Programming
WEKA, Orange, KNIME, Vertex AI, Sagemaker, AzureML, Watson Studio
LowCoder
Natural Language Interface
ChatGPT, GitHub Copilot

Figure 3.7: Relationship between LowCoder and other low-code for AI tools. LowCoder is the only low-code tool that supports both visual programming and a natural language interface.

Overnight paper addresses the problem of missing training data for PBNL interfaces by crowdsourcing [217]. And SwaggerBot lets users extend and customize a chatbot from within the chatbot itself [204]. Unlike these works, our paper uses language models for PBNL, uses PBNL for creating AI pipelines, and integrates with a visual programming interface.

Combining low-code techniques: Our work combines visual programming with PBNL. In a similar vein, Rousillon combines visual programming with programming by demonstration [45] and Pumice combines programming by demonstration with PBNL [121]. Like Rousillon and Pumice, our goal in combining techniques is to use strengths of each technique to mitigate weaknesses in the other. However, unlike Rousillon and Pumice, we choose different techniques to combine and target a different domain, namely AI pipelines.

User studies on AI tools: There are a few studies that aim to evaluate whether developers perform better on programming tasks when working with AI tools. Vaithilingam et al. had developers use GitHub Copilot on three programming tasks and found that while neither task success rate nor completion time improved while using Copilot, developers preferred using it compared to the standard code completion [202]. Similarly, Xu et al. had developers perform several programming tasks with and without the use of a natural language to code generation model and found no significant differences with regards to code quality, task completion time and program correctness [228]. Wang et al. interviewed several data scientists to better understand their perceptions of automated AI and found that they had mixed feelings [213]. However, nearly all of them felt that the future of data science involved collaboration between humans and AI systems. Unlike other work which tends to focus on how AI supports software development by experienced developers, our paper focuses on AI tools in the context of low-code systems where developers have varying expertise levels in both building software and AI.

3.6 Summary

One of the challenges with AI programming tools is that the generated code can be difficult to understand due to its syntactic complexity. Low-code techniques like visual programming overcome this by allowing users to create programs without any textual code. However, tools that rely only on visual development poorly support discoverability of API components (basic building blocks of code). To solve this problem, we developed LowCoder, the first low-code tool for developing AI pipelines that supports both a visual programming interface and offers an AI-powered natural language interface. By evaluating LowCoder with developers, we show both that the visual programming interface supports rapid prototyping and that the AI driven natural language interface helps developers discover code blocks when they know what they want to do but not how to implement their designs.

Data Availability: The implementation of LowCoder, datasets for training and evaluating LowCoder$_{\text{NL}}$, results of additional experiments, as well as the material from the user study, including the full set of (axial) codes and anonymized quantitative and qualitative data, are available at: DOI 10.5281/zenodo.7601206

3.7 Takeaway

AI has shown a lot of potential in empowering individuals with limited or no programming experience to write code, but this code is often difficult to understand and manipulate. We address this challenge by abstracting away textual code and replacing it with a more intuitive drag-and-drop based visual interfaces. LowCoder facilitates the integration of AI into a trusted environment tailored to the needs of non-expert programmers. Through LowCoder, we show that AI can still be just as useful at this level of abstraction. Specifically, the natural language model supports users in the visual space despite being trained on textual code. Consequently, LowCoder provides a user-friendly space where individuals can leverage the capabilities of AI without the need for extensive coding knowledge, thereby enhancing accessibility and usability.

Chapter 4

CAT-LM for Verification

Generating tests for verification is another prominent challenge that arises with using Large Language Models (LLMs) for code generation. Despite excelling in code generation, LLMs face limitations when it comes to generating tests. This is attributed to their training approach, which focuses on generating individual code files independently, following the standard practice in natural language processing. As a result, they can not consider the code under test context when generating the tests.

During software development, developers often write tests to verify the correctness of their code. In projects that are well tested, most code files have at least one corresponding test file that implements unit and/or integration test functions that evaluate the functionality of the code. However, writing high quality tests can be time-consuming [30, 31] and is often overlooked. To address this, there has been extensive work done in automating test generation, which includes both classical [25, 40, 62, 72] and neural-based methods [62, 205, 219].

Classical test generation tools like EvoSuite [72] are optimized to generate tests with high-coverage. However, the generated tests are often hard to read and may be unrealistic or even wrong [157]. As a result, developers need to invest time and effort to verify the correctness of generated tests [40]. Meanwhile, LLMs trained on code have made significant advancements in generating functions that are human-like and of high quality, leveraging their file-level context [28, 47, 73, 151]. Tools like Copilot excel at code generation, thereby significantly enhancing user productivity [6]. However, these models are currently less adept at generating tests, because they are trained to generate the code in each file separately - a standard practice in natural language processing.

Generating meaningful tests, of course, critically requires considering the alignment between the tests and the corresponding code under test. Some prior work on neural-based test generation methods has focused on modeling this alignment [62, 150, 219]. However, this work typically focuses on the relatively narrow task of generating individual assertions in otherwise complete tests, based on a single method under test. Unlocking the more impactful ability to generate entire tests requires leveraging both the entire code file and existing tests as context, which in turn requires substantially larger models.

In this work, we make a significant step towards accurate whole-test generation via CAT-

LM,[1] a language model trained on aligned **C**ode **A**nd **T**ests. CAT-LM is a bi-lingual GPT-style LLM with 2.7B parameters. It is trained on a large corpus of Python and Java projects using a novel pretraining signal that explicitly considers the mapping between code and test files, when available, while also leveraging the (much larger) volume of untested code. Modeling the code file along with the test leads to additional challenges regarding a model's context length. Most code generation models support a context window of up to 2,048 tokens. However, our data analysis indicates that many code-test file *pairs* comprise more than 8K tokens. We thus increase the maximum sequence input length, training CAT-LM with a context window of 8,192 tokens.

Our results show that the model effectively leverages the code file context to generate more syntactically valid tests that achieve higher coverage. The model provides a strong prior for generating plausible tests: combined with basic filters for compilability and coverage, CAT-LM frequently generates tests with coverage close to those written by human developers.

We evaluate CAT-LM against several strong baselines across two realistic applications: test method generation and test method completion. For test method generation, we compare CAT-LM to both human written tests as well as the tests generated by StarCoder [119] and, the CodeGen [151] model family, which includes mono-lingual models trained on a much larger budget than ours. We also compare against TeCo [150], a recent test-specific model, for test completion. CAT-LM generates more valid tests on average than StarCoder and all CodeGen models, and substantially outperforms TeCo at test completion. Our results highlight the merit of combining the power of large neural methods with a pretraining signal based on software engineering expertise—in this case, the importance of the relation between code and test files.

4.1 Overview

CAT-LM is a GPT-style model that can generate tests given code context. Figure 8.1 shows an overview of our entire system, which includes data collection and preprocessing (detailed in Section 4.3.1), pretraining CAT-LM (Section 4.4), and evaluation (Section 4.5).

We first collect a corpus of ca. 200K Python and Java GitHub repositories, focusing on those with at least 10 stars. We split these at the project level into a train and test set (Section 4.3.1). We filter our training set following CodeParrot [221] standards (including deduplication), resulting in $\sim$15M code and test files. We align code and test files using a fuzzy string match heuristic (Section 4.3.2).

We then prepare the training data, comprising of the code-test file pairs, paired with a unique token (`<|codetestpair|>`), as well as unpaired code and test files. We tokenize the files using a custom-trained sentencepiece tokenizer [4]. We then determine the appropriate model size, 2.7B parameters based on our training budget and the Chinchilla scaling laws [91]. We use the GPT-NeoX toolkit [3] enhanced with Flash Attention [58] to pretrain CAT-LM using an auto-regressive (standard left-to-right) pretraining objective that captures the mapping between code and test files, while learning general code and test structure.

Finally, we evaluate CAT-LM on the held-out test data. We manually set up all projects with executable test suites from the test set to form our testing framework. We prepare our test inputs

[1]CAT-LM [168] was published at the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2023

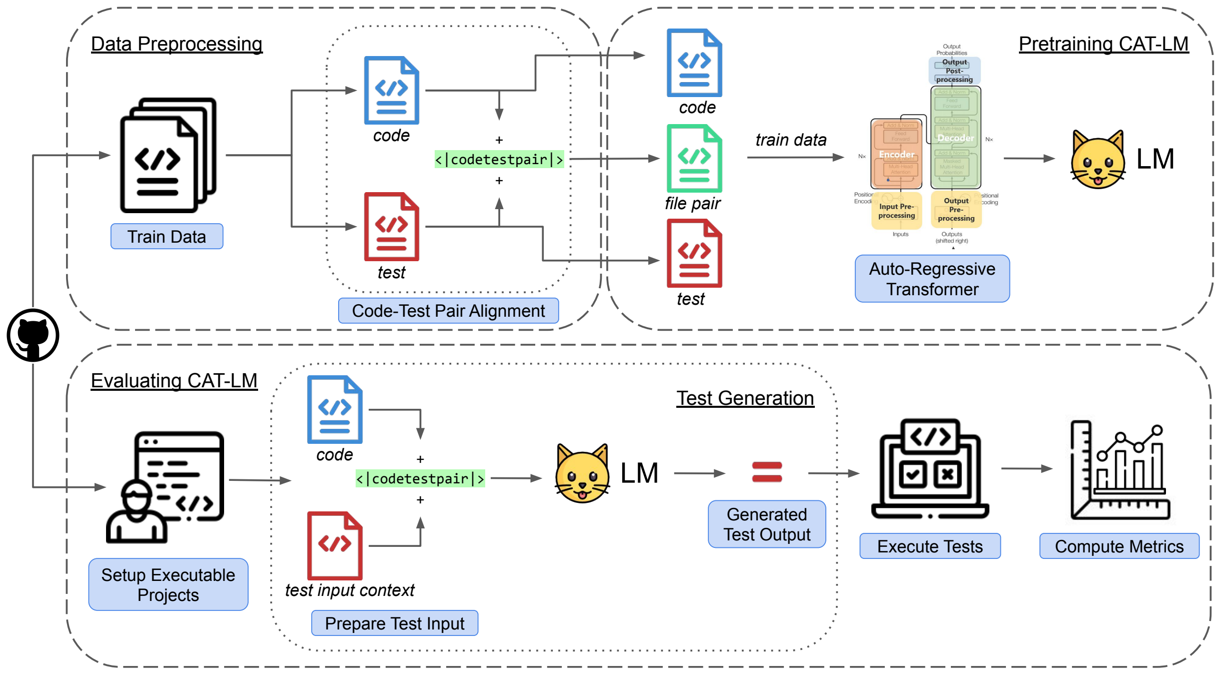
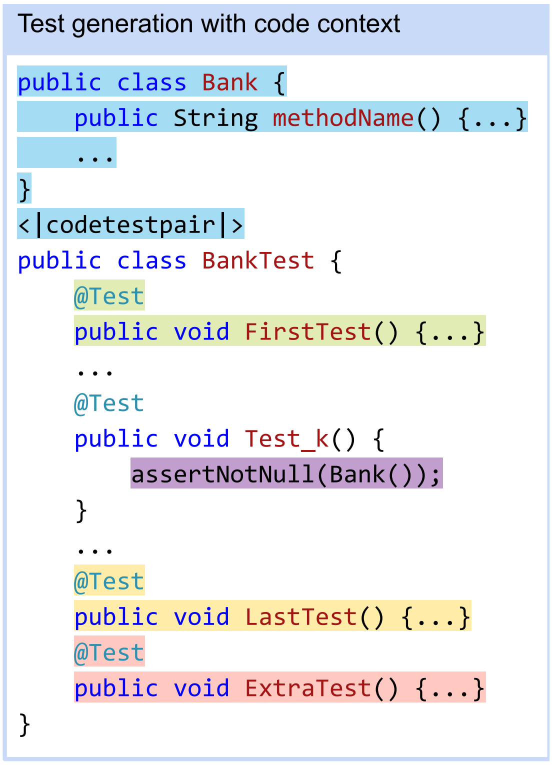


Figure 4.1: Approach overview. We extract Java and Python projects with tests from GitHub and heuristically align code and test files (top), which, along with unaligned files, train CAT-LM, a large, auto-regressive language model. We evaluate CAT-LM's generated tests on a suite of executable projects (bottom), measuring its ability to generate syntactically valid tests that yield coverage comparable to those written by developers.

for CAT-LM by concatenating the code context to the respective test context for test generation. The test context varies based on the task. We asses our model's ability to generate (1) the first test method, (2) the last test method, add (3) an additional, new test to an already complete test suite. We also evaluate completing a statement within a test function. We tokenize prepared input and task CAT-LM with sampling multiple (typically 10) test outputs, each consisting of a single method. We then attempt to execute the generated tests with our testing framework and compute metrics like number of generated tests that compile and pass, along with the coverage they provide, to evaluate test quality.

4.2 Tasks

We describe two tasks for which CAT-LM can be used, namely test method generation (with three settings) and test completion. Figure 4.2 demonstrates the setup for all tasks including code context.



```
public class Bank {
    public String methodName() {...}
    ...
}
<|codetestpair|>
public class BankTest {
    @Test
    public void FirstTest() {...}
    ...
    @Test
    public void Test_k() {
        assertNotNull(Bank());
    }
    ...
    @Test
    public void LastTest() {...}
    @Test
    public void ExtraTest() {...}
}
```

Figure 4.2: Evaluation tasks, with code context shown for completeness: test generation for the first test method, last test method, and extra test method, along with test completion for Java.

4.2.1 Test Method Generation

Given a partially complete test file and its corresponding code file, the goal of *test method generation* is to generate the next test method. Developers can use test generation to produce an entire test suite, or add tests to an existing test suite to test new functionality. We evaluate three different settings, corresponding to different phases in the testing process, namely generating (1) the *first test* in the file, representing the beginning of a developer's testing efforts. In this setting, we assume that basic imports and high-level scaffolding are in place, but no test cases have been written, (2) the *final test* in a file, assessing a model's ability to infer what is missing from a near-complete test suite. We evaluate this ability only on test files that have two or more (human-written) tests to avoid cases where only a single test is appropriate, and (3) an *extra* or additional test, which investigates whether a model can generate new tests for a largely complete test suite. Note that this may often be unnecessary in practice.

4.2.2 Test Completion

The goal of *test completion* is to generate the next statement in a given incomplete test method. Test completion aims to help developers write tests more quickly. Although test completion shares similarities with general code completion, it differs in two ways: (1) the method under

Table 4.1: Summary statistics of the overall dataset.

Attribute		Python	Java	Total
Project	Total	148,605	49,125	197,730
	Deduplicated	147,970	48,882	196,852
	W/o Tests	84,186	15,128	99,314
	W/o File pairs	108,042	23,933	131,975
Size (GB)	Raw	123	157	280
	Deduplicated	53	94	147
Files	Total	8,101,457	14,894,317	22,995,774
	Filtered	7,375,317	14,698,938	22,074,255
	Deduplicated	5,101,457	10,418,609	15,520,066
	Code	4,128,813	8,380,496	12,509,309
	Test	972,644	2,038,113	3,010,757
	File pairs	412,881	743,882	1,156,763
	Training	4,688,576	9,674,727	14,363,303

test offers more context about what is being tested, and (2) source code and test code often have distinct programming styles, with test code typically comprising setup, invocation of the method under test, and assertions about the output (the test oracle).

4.3 Dataset

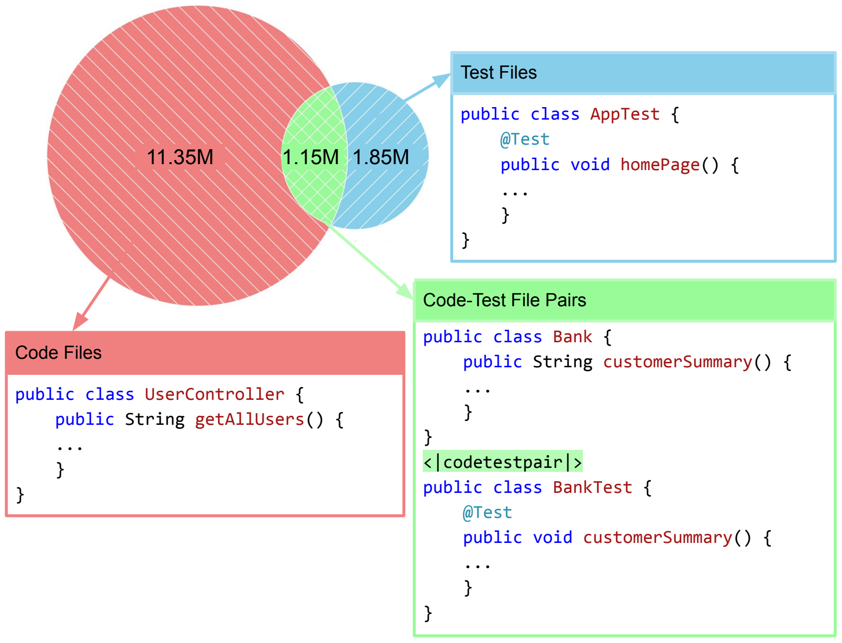
This section describes dataset preparation for both training and evaluating CAT-LM. Table 4.1 provides high-level statistics pertaining to data collection and filtering.

4.3.1 Data Collection

We use the GitHub API [1] to mine Python and Java repositories that have at least 10 stars and have new commits after January 1st, 2020. Following [19] and [132], we also remove forks, to prevent data duplication. This results in a total of 148,605 Python and 49,125 Java repositories with a total of ~23M files (about 280 GB). We randomly split this into train and test set, ensuring that the test set includes 500 repositories for Python and Java each.

4.3.2 Training Data Preparation

We first remove all non-source code files (e.g., configuration and README files) to ensure that the model is trained on source code only. We then apply a series of filters in accordance with CodeParrot's standards [221] to minimize noise from our training signal. This includes removing files that are larger than 1MB, as well as files with any lines longer than 1000 characters; an



```
public class AppTest {
    @Test
    public void homePage() {
    ...
    }
}
```

```
public class Bank {
    public String customerSummary() {
    ...
    }
}
<|codetestpair|>
public class BankTest {
    @Test
    public void customerSummary() {
    ...
    }
}
```

```
public class UserController {
    public String getAllUsers() {
    ...
    }
}
```

Figure 4.3: Distribution of files with sample code snippets

average line length of >100 characters; more than 25% non-alphanumeric characters, and indicators of being automatically generated. This removes 9% of both Python and Java files. We deduplicate the files by checking each file's md5 hash against all other files in our corpus. This removes approximately 30% of both Python and Java files.

We extract code-test file pairs from this data using a combination of exact and fuzzy match heuristics. Given a code file with the name `<CFN>`, we first search for test files that have the pattern `test_<CFN>`, `<CFN>_test`, `<CFN>Test` or `Test<CFN>`. If no matches are found, we perform a fuzzy string match [5] between code and test file names, and group them as a pair if they achieve a similarity score greater than 0.85. If multiple matches are found, we keep the pair with the highest score.

Following file pair extraction, we prepare our training data by replacing the code and test files with a new file that concatenates the contents of the code file and the test file, separating them with a unique `<|codetestpair|>` token. This ensures that the model learns the mapping between code and test files from the pretraining signal. Note that we always combine these files starting with the code, so the model (which operates left-to-right) only benefits from this pairing information when generating the test. We additionally include all the other code and test files for which we did not find pairs in our training data, which results in 4.7M Python files and 9.7M Java files. We include these unmatched files to maximize the amount of data the model can learn from. Figure 4.3 summarizes the distribution of files in the training data along with sample code snippets for each type of file.

Distribution of files and file pairs: Figure 4.4 summarizes the distribution of files in projects with respect to their star count. We observe a decreasing trend in not just the number of code files and test files, but also the file pairs. Upon manual inspection of a few randomly selected projects, we find that popular projects with a high star count tend to be better-tested, in line with prior literature [111, 189]. Note that we normalize the plot to help illustrate trends by aggregating

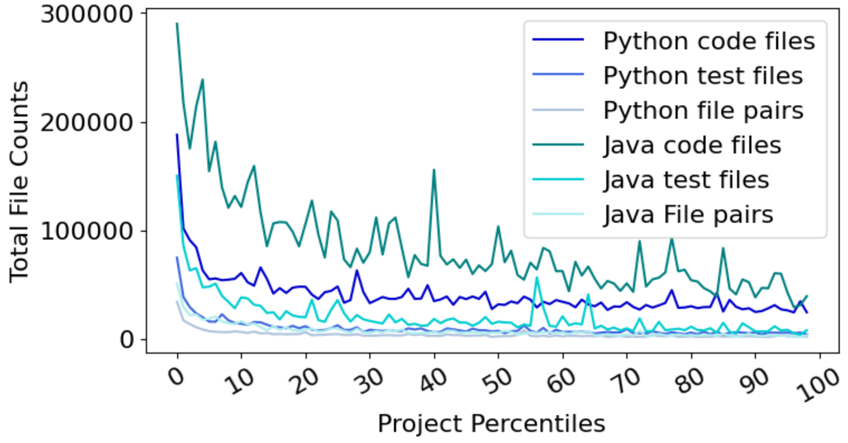


Figure 4.4: Distribution of files in projects sorted by GitHub stars, normalized by percentiles

projects in buckets based on percentiles, after sorting them based on stars. The data distribution varies between Python and Java: Python has approximately 3x more projects than Java, but Java has roughly twice as many code-test file pairs.

4.3.3 Test Data Preparation and Execution Setup

To prepare our test data, we first excluded all projects without code-test file pairs. This resulted in a total of 97 Java and 152 Python projects. We then attempted to set up all projects for automated test execution.

Execution Setup for Java: Projects may use different Java versions (which include Java 8, 11, 14, and 17) and build systems (mostly Maven and Gradle). We manually set up Docker images for each combination. We then attempted to execute the build commands for each project in a container from each image. We successfully built 54 out of the 97 Java projects, containing 61 code-test file pairs.

Execution Setup for Python: We manually set up Docker containers for Python 3.8 and 3.10 with the `pytest` framework and attempted to run the build commands for each project until the build was successful. We successfully built 41 of the 152 Python projects, containing 1080 code-test file-pairs.

We further discarded all *pairs* within these projects with only a single code method or a single test method to ensure that code-test file-pairs in our test set correspond to nontrivial test suites. We additionally require the Java and Python projects to be compatible with the `Jacoco` and `coverage` libraries respectively. This leaves a total of 27 code-test file pairs across 26 unique Java projects and 517 code-test file pairs across 26 unique Python projects. In Python, we randomly sampled up to 10 file pairs per project to reduce the bias towards large projects (the top two projects account for 346 tests) leading to a final set of 123 file pairs across 26 unique Python projects. Note that we reuse these Docker containers in our testing framework (See Section 4.5.1).

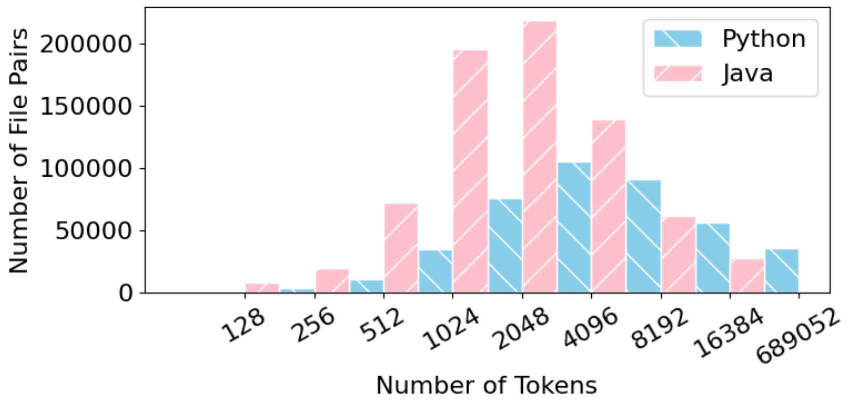


Figure 4.5: Distribution of file pair tokens

4.4 CAT-LM

This section describes the details for preparing the input, pretraining CAT-LM and generating the outputs.

4.4.1 Input Representation for Pretraining CAT-LM

We use the corpus of 14M Java and Python files that we prepared for the pretraining of our model (see Section 4.3.1). We first train a subword tokenizer [112] using the SentencePiece [4] toolkit with a vocabulary size of 64K tokens. The tokenizer is trained over 3 GB of data using ten random lines sampled from each file. We then tokenize our input files into a binary format used to efficiently stream data during training.

Analysing the distribution of tokens: Language models are typically constrained in the amount of text they fit in their context window. Most current code generation models use a context window of up to 2,048 tokens [151, 227].[2] Our analysis on the distribution of tokens, visualized in Figure 4.5, showed that this only covers 35% of the total number of file pairs. As such, while it may be appropriate for a (slight) majority of individual files, it would not allow our model to leverage the code file's context while predicting text in the test file. This is a significant limitation since we want to train the model to use the context from the code file when generating tests.

Further analysis showed that approximately 82% of all file pairs for Java and Python have fewer than 8,192 tokens. Since the cost of the attention operation increases quadratically with the context length, we choose this cutoff to balance training cost and benefit. Therefore, we chose to train a model with a longer context window of 8192 tokens to accommodate an additional ∼550K file pairs. Note that this does not lead to any samples being discarded; pairs with more tokens will simply be (randomly) chunked by the training toolkit.

[2]The average length of a token depends on the vocabulary and dataset, but can typically be assumed to be around 3 characters.

4.4.2 Model and Training Details

We determined the model size based on our cloud compute budget of \$20,000 and the amount of available training data, based on the Chinchilla scaling laws [91], which suggest that the training loss for a fixed compute budget can be minimized (lower is better) by training a model with ca. (and no fewer than) 20 times as many tokens as it has parameters. Based on preliminary runs, we determined the appropriate model size to be 2.7 (non-embedding) parameters, a common size for medium to large language models [151, 227], which we therefore aimed to train with at least 54B tokens. This model architecture consists of a 2,560-dimensional, 32 layer Transformer model with a context window of 8,192 tokens. We trained the model with a batch size of 256 sequences, which corresponds to $\sim$2M tokens. We use the GPT-NeoX toolkit [3] to train the model efficiently with 8 Nvidia A100 80GB GPUs on a single machine on the Google Cloud Platform. We trained the model for 28.5K steps, for a total of nearly 60B tokens, across 18 days, thus averaging roughly 1,583 steps per day[3] We note that this training duration is much shorter than many popular models [151, 200];[4] the model could thus be improved substantially with further training. The final model is named CAT-LM as it is trained on aligned **C**ode **A**nd **T**ests.

4.4.3 Prompting CAT-LM to generate outputs:

Since CAT-LM has been trained using a left-to-right autogressive pretraining signal, it can be prompted to generate some code based on the preceding context. In our case, we task it to either generate an entire test method given the preceding test (and usually, code) file context, or generating a line to complete the test method (given the same). We prompt CAT-LM with the inputs for each task, both with and without code context, and sample 10 outputs from CAT-LM with a “temperature” of 0.2, which encourages generating different, but highly plausible (to the model) outputs. Sampling multiple outputs is relatively inexpensive given the size of a method compared to the context size, and allows the model to efficiently generate multiple methods from an encoded context. We can then filter out tests that do not compile, lack asserts, or fail (since we are generating behavioral tests), by executing them in the test framework. We prepare the outputs for execution by adding the generated test method to its respective position in the baseline test files, without making any changes to the other tests in the file.

4.5 Experimental Setup

We describe the setup for evaluating CAT-LM across both tasks outlined in Section 4.2, namely test method generation, and test completion.

[3]We further trained the model to 35.3K steps, thanks to an additional grant received for \$5000, after the paper was published. This latest checkpoint is now available on HuggingFace. Please see https://github.com/RaoNikitha/CAT-LM for more details on usage. Note that the numbers reported in this paper make use of the older checkpoint (28.5K steps), and may not match the numbers from the newer public checkpoint (35.3K steps).

[4]The “Chinchilla” optimum does not focus on maximizing the performance for a given model size, only for a total compute budget.

4.5.1 Test Method Generation

The test method generation task involves three different cases: generating the first test, the final test, and an extra test in a test suite (see Section 4.2). We evaluate CAT-LM on test method generation both with code context and, as an ablation, without code context.

Baseline Models

CodeGen is a family of Transformer-based LLMs trained auto-regressively (left-to-right) [151]. Pretrained CodeGen models are available in a wide range of sizes, including 350M, 2.7B, 6.1B and 16.1B parameters. These models were trained on three different datasets, starting with a large, predominantly English corpus, followed by a multi-lingual programming language corpus (incl. Java and Python), and concluding with fine-tuning on Python data only. The largest model trained this way is competitive with Codex [47] on a Python benchmark [151].

For our evaluation, we compare with CodeGen-2.7B-multi, which is comparable in size to our model and trained on multiple programming languages, like our own. We also consider CodeGen-16B-multi (with 16B parameters, ca. 6 times larger than CAT-LM) which is the largest available model trained on multiple programming languages. For all Python tasks, we also compare against CodeGen-2.7B-mono and CodeGen-16B-mono, variants of the aforementioned models fine-tuned on only Python code for an additional 150k training steps.

We also compare the performance of CAT-LM with StarCoder [119], which is a 15.5B parameter model trained on over 80 programming languages, including Java and Python, from The Stack (v1.2). StarCoder has a context window of $8,192$ tokens. It was trained using the Fill-in-the-Middle objective [28] on 1 trillion tokens of code, using the sample approach of randomizing the document order as CodeGen.

Lexical Metrics

Although our goal is not to exactly replicate the human-written tests, we provide measures of the *lexical* similarity between the generated tests and their real-world counterparts as indicators of their realism. Generated tests that frequently overlap in their phrasing with ground-truth tests are likely to be similar in structure and thus relatively easy to read for developers. Specifically, we report both the rate of exact matches and several measures of approximate similarity, including ROUGE [128] (longest overlapping subsequence of tokens) and CodeBLEU [172] score (n-gram overlap that takes into account code AST and dataflow graph). We only report lexical metrics for our first test and last test settings, as there is no ground truth to compare against in our extra test setting. These metrics have been used extensively in prior work on code generation and test completion [93, 114, 150, 215].

Runtime Metrics

We also report runtime metrics that better gauge test utility than the lexical metrics. This includes the number of generated tests that compile, and generated tests that pass the test suite. We also measure coverage of the generated tests. For first and last tests, we compare this with the coverage realized by the corresponding human-written tests. We hope that this work will

Table 4.2: Baseline coverage for human written tests over the given number of file pairs.

PL	Case	Cov Imp %	# File Pairs
Python	First test	59.3%	112
	Last test	5.0%	93
	Extra test	0.0%	123
Java	First test	50.5%	27
	Last test	5.3%	18
	Extra test	0.0%	27

encourage more widespread adoption of runtime metrics (which are an important part of test utility), as prior work primarily focuses on lexical similarity [62, 150, 219].

Preparing Input Context and Baseline Test Files

We use an AST parser on the ground-truth test files to prepare partial tests with which to prompt CAT-LM. For first test generation, we remove all test cases (but not the imports, nor any other setup code that precedes the first test); for last test generation, we leave all but the final test method, and for final test generation we only remove code after the last test. We then concatenate the code context to the test context using our delimiter token for the 'with code context' condition.

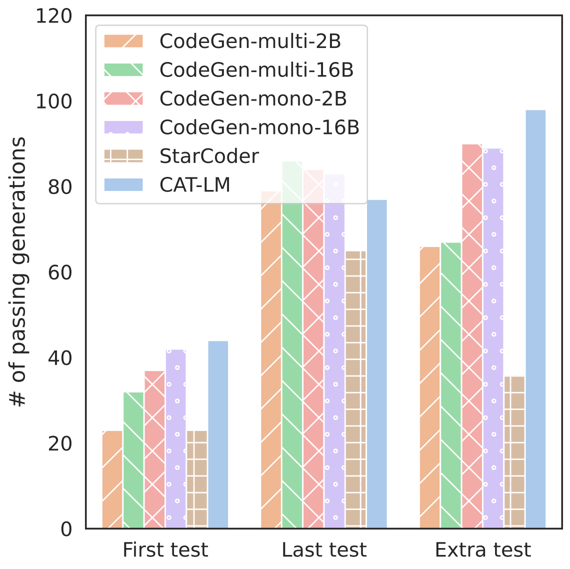
We additionally obtain coverage with the original, human-written test files under the same conditions, keeping only the first or all tests as baselines for first and last test prediction respectively. Note that there is no baseline for the extra test generation task.

Testing Framework

We evaluate the quality of the generated tests using the containers that we setup to execute projects in Section 4.3.3. We insert the generated test into the original test file, execute the respective project's setup commands and check for errors, recording the number of generated tests that compile and pass the test suite (see Section 4.5.1). If the generated test compiles successfully (or, for Python, is free of import or syntax errors), we run the test suite and record whether the generated test passed or failed. We compute code coverage for all passing tests, contrasting this with the coverage achieved by the human-written test cases (when available) as baselines.

4.5.2 Test Completion

Recall the test completion task involves generating a single line in a given test method, given the test's previous lines. We perform our evaluation for test completion under two conditions, with code context and without code context.



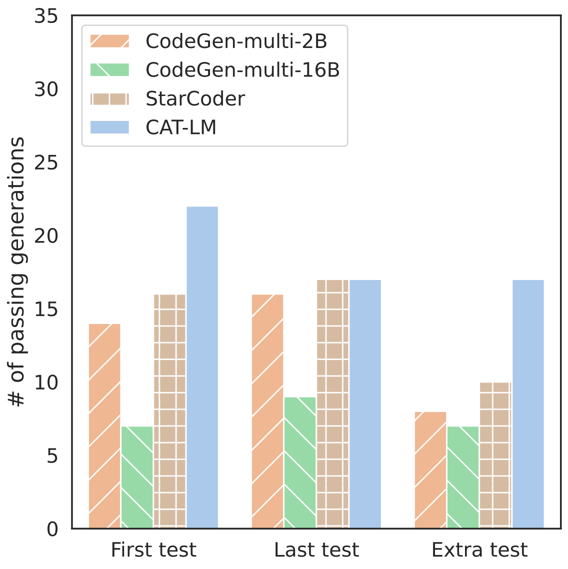


Figure 4.6: Passing tests by model for Python (left) and Java (right).

Baseline Model

We compare against TeCo [150], a state of the art baseline on test statement completion that has outperformed many existing models, including CodeT5 [215], CodeGPT [134] and TOGA [62]. TeCo [150] is a encoder-decoder transformer model based on the CodeT5 architecture [215]. TeCo takes the test method signature, prior statements in the test, the method under test, the variable types, absent types and method setup and teardown as input.

Initially, we intended to compare CAT-LM against TeCo on our test set. However, TeCo performs extensive filtering including requiring JUnit, Maven, well-named tests, a one-to-one mapping between test and method under test, and no if statements or non-sequential control flow in the test method. We thus compared CAT-LM against TeCo for 1000 randomly sampled statements from their test set.

Metrics

We compare CAT-LM against TeCo across all lexical metrics (outlined in Section 4.5.1).

4.6 Evaluation

We evaluate CAT-LM's ability to generate valid tests that achieve coverage, comparing against state of the art baselines for both code generation and test completion.

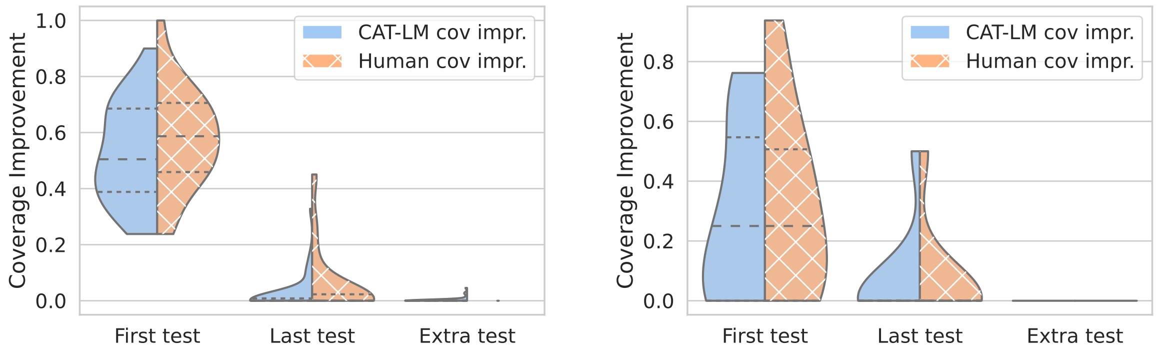


Figure 4.7: Coverage improvement of our model vs humans for Python (left) and Java (right).

4.6.1 Test Method Generation

Pass Rate

Figure 4.6 shows the number of passing tests generated by each model for Python and Java. Note that these are absolute numbers, out of a different total for each setting.[5]

CAT-LM outperforms StarCoder and all CodeGen models, including ones that are much larger and language-specific in most settings. For Python, all models perform worst in the first test setting, where they have the least context to build on. Nonetheless, equipped with the context of the corresponding code file, our model generates substantially more passing tests than StarCoder (with 15.5B parameters) and the multilingual CodeGen baselines (trained with far more tokens) in both first and extra test setting. Only in the last-test settings do some of the models compete with ours, though we note that their performance may be inflated as the models may have seen the files in our test set during training (the test set explicitly omits files seen by CAT-LM during training). For Java, we find that CAT-LM generates more passing tests than StarCoder and the two multilingual CodeGen models (no Java-only model exists). The difference is most pronounced in the extra test setting, where CAT-LM generates nearly twice as many passing tests compared to StarCoder and the CodeGen baseline models. Overall, despite being undertrained, CAT-LM generates more number of passing tests on average across all settings. Both StarCoder and the CodeGen models don't show significant gains with more parameters or longer contexts (StarCoder can use $8,192$ tokens), highlighting that training with code context is important.

Coverage

Figure 4.7 shows the coverage distribution of CAT-LM, contrasted with that of the human-written tests. For both the first test and last test settings, our model performs mostly comparably to humans, with both distributions having approximately the same median and quartile ranges. The extra test task is clearly especially hard: while our model was able to generate many tests in this setting (Figure 4.6), these rarely translate into *additional* coverage, beyond what is provided

[5]The denominator for each group is the number of file pairs shown in Table 4.2 multiplied by 10, the number of samples per context.

Table 4.3: Lexical and runtime metrics performance comparison of the models on the held-out test set for Java and Python. CodeGen refers to CodeGen-multi for Java and CodeGen-mono for Python results. We only report lexical metrics for our first test and last test settings, as there is no gold test to compare against in our extra test setting.

	Java					**Python**				
	Lexical Metrics			**Runtime Metrics**		**Lexical Metrics**			**Runtime Metrics**	
Model	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**
First Test (Total: Java = 270, Python = 1120)										
CAT-LM w Context	41.4%	**15.4%**	60.9%	**50**	**22**	21.0%	0.3%	**39.4%**	**384**	**44**
CAT-LM w/o Context	37.5%	**15.4%**	56.5%	9	9	17.7%	0.4%	30.2%	236	31
Codegen-2B	35.5%	7.7%	56.8%	24	14	18.2%	0.0%	30.9%	259	37
Codegen-16B	42.2%	7.7%	61.8%	25	7	20.8%	0.3%	35.1%	361	42
StarCoder	**44.6%**	10.9%	**62.2%**	28	16	**24.0%**	**1.8%**	38.8%	269	23
Last Test (Total: Java = 180, Python = 930)										
CAT-LM w Context	55.4%	20.8%	70.8%	**54**	**17**	**38.3%**	**4.8%**	**54.9%**	335	77
CAT-LM w/o Context	53.6%	20.8%	68.9%	33	14	33.2%	1.4%	51.9%	**350**	79
Codegen-2B	51.7%	13.0%	69.2%	43	16	36.3%	2.2%	53.2%	326	**84**
Codegen-16B	56.5%	14.3%	**70.9%**	24	9	37.9%	3.4%	54.0%	349	83
StarCoder	**56.9%**	**21.0%**	69.9%	34	**17**	37.6%	4.2%	54.5%	227	65
Extra Test (Total: Java = 270, Python = 1230)										
CAT-LM w Context	–	–	–	**41**	17	–	–	–	380	98
CAT-LM w/o Context	–	–	–	29	**20**	–	–	–	**425**	**104**
Codegen-2B	–	–	–	17	8	–	–	–	376	90
Codegen-16B	–	–	–	15	7	–	–	–	384	89
StarCoder	–	–	–	17	10	–	–	–	269	36

by the rest of the test suite, in part because most of the developer-written test suites in our dataset already have high code coverage (average coverage of 78.6% for Java and 81.6% for Python), and may have no need for additional tests. Table 4.2 shows the average human coverage improvement for the first and last test added to a test suite. Note that the average is significantly lower for last test, as baseline coverage is already high for this mode (74.7% for Java and 76.1% for Python).

We note that we could not compute coverage for all the file pairs in each setting. We excluded file pairs with only one test from our last test setting to differentiate it from our first test setting. For the first test setting, some baseline files were missing helper methods between the first test and last test in the file, preventing us from computing coverage.

Lexical Similarity

Table 4.3 shows the lexical similarity metrics results relative to the human-written tests for CAT-LM, both with and without context, along with StarCoder and CodeGen baselines. CAT-LM reports high lexical similarity scores when leveraging code context, typically at or above the level of the other best model, StarCoder (with 15B parameters). This effect is consistent across first and last test generation.

Impact of Code Context

As is expected, CAT-LM heavily benefits from the presence of code context. When it is queried without this context, its performance on lexical metrics tends to drop to below the level of CodeGen-2B, which matches it in size but was trained with more tokens. The differences in lexical metric performance are sometimes quite pronounced, with up to a 9.2% increase in Rouge score and up to a 5.1% increase in CodeBLEU score.

In terms of runtime metrics, code context mainly helps on the first and last test prediction task, with especially large gains on the former. Context does not seem to help generate more passing tests in the extra test setting. This may be in part because the test suite is already comprehensive, so the model can infer most of the information it needs about the code under test from the tests. It may also be due to the test suites often being (nearly) complete in this setting, so that generating additional tests that pass (but yield no meaningful coverage) is relatively straightforward (e.g., by copying an existing test Section 4.6.3). Overall, these results support our core hypothesis that models of code should consider the relationship between code and test files to generate meaningful tests.

Other Runtime Metrics

Table 4.3 also shows a comparison between CAT-LM and StarCoder and CodeGen baselines for all runtime metrics. CAT-LM outperforms both StarCoder and the CodeGen baselines in both Python in Java across compiling and passing generations, with CAT-LM typically generating the most samples that compile and pass. The one setting where the CodeGen baselines perform slightly better is in generating more last tests that pass for Python. However, the compile rate of these CodeGen generated tests is significantly lower than those generated by CAT-LM. We note that CodeGen's performance may be inflated in the last test setting, as it may have seen the files from the test set during training.

CAT-LM outperforms StarCoder and CodeGen for both Python and Java, **generating more passing tests** on average across all settings. We find that **code context improves performance** across most settings in terms of both lexical and runtime metrics.

Table 4.4: Comparison of CAT-LM and TeCo on 1000 randomly sampled statements in their test set.

Model	CodeBLEU	XMatch	Rouge
CAT-LM w/ Context	**67.1%**	**50.4%**	**82.8%**
CAT-LM w/o Context	65.9%	48.9%	82.2%
TeCo	26.7%	13.8%	60.2%

4.6.2 Test Completion

For test completion (see Section 4.2.2 for task definition), we compare CAT-LM against TeCo [150] on the lexical metrics outlined in Section 4.5.1. Specifically, we sample 1000 statements at random from across the test set released by the authors of TeCo, on which we obtain similar performance with TeCo to those reported in the original paper. Table 4.4 shows the results. CAT-LM outperforms TeCo across all lexical metrics, with a 36.6% increase in exact match, 22.6% increase in ROUGE and 40.4% increase in CodeBLEU score. Even prompting CAT-LM with just the test context (i.e., without the code context) yields substantially better results than TeCo. This underscores that providing the entire test file prior to the statement being completed as context, rather than just the setup methods, is helpful for models to reason about what is being tested.

In contrast to the test generation task, code context only slightly helps CAT-LM in this setting, with an increase in CodeBLEU score of 1.2% and increase in exact match accuracy of 1.5%. Apparently, many individual statements in test cases can be completed relatively easily based on patterns found in the test file, without considering the code under tests. This suggests that statement completion is significantly less context-intensive than whole-test case generation. We therefore argue that entire test generation is a more appropriate task for assessing models trained for test generation.

CAT-LM outperforms TeCo across all lexical metrics, with a 40.4% improvement in CodeBLEU score and 36.6% improvement in exact match accuracy. We find that **context only slightly helps** with test statement prediction, indicating that test completion can largely be done without the code under test, in contrast to entire test generation.

4.6.3 Qualitative Comparisons

Finally, we conduct a small-scale qualitative case-study of tests generated by CAT-LM, CodeGen-2B-multi [151], GPT-4 [154] and EvoSuite [72]. GPT-4 is a vastly larger language model than ours, trained with an undisclosed budget by OpenAI. EvoSuite is a popular test generation tool for Java based on evolutionary algorithms.

We analyze a randomly sampled passing generation from CAT-LM in contrast to the tests generated by the other tools in the same context across each our three settings (first test, last test and extra test). The tests here are generated for a `Bank` class, which includes methods to add a customer, open an account and print a summary of all accounts and customers. Our goal is to better understand the benefits and drawbacks of each tool's generated tests. Specifically, we look for characteristics of high quality tests, such as meaningful method and variable names, proper invocation of the method under test and high quality assertions.

CAT-LM

Listing 4.1 shows the first test generation by CAT-LM. The name of the test is informative, along with its variables. It also follows unit testing conventions of testing one specific method in the

`Bank` class. This is consistent across the examples for last test and extra test. However, for our extra test example, CAT-LM copied the previous test and changed the name of the test method, not testing new functionality.

CodeGen

In Listing 4.2, the test generated by CodeGen is quite readable, semantically correct, and natural looking. However, it uses multiple non-existent methods from the code under test—a phenomenon popularly dubbed "hallucinating"—since it lacks awareness of `Bank`'s implementation. StarCoder performs similarly, generating tests that are readable, semantically correct, and natural looking but suffer from hallucinations.

GPT-4

GPT-4 consistently performs the best of all three tools, generating tests that either are identical to the ground truth or test new functionality that none of the existing tests do. Listing 4.3 shows GPT-4's generation for the first test case. Similar to CAT-LM, the GPT-4 generated test has meaningful identifier names and assertions. GPT-4 had similarly good tests for our last test and extra test settings. However, these results come with several caveats. First, GPT-4 was trained on a very large volume of data, including public code, so it is quite likely that it was trained on our test data and has thus seen the original tests.[6] Second, GPT-4 is a much larger, model, with a training budget orders of magnitude higher than ours. Given our strong performance compared to the (already much more expensive) CodeGen models, we expect that modestly scaling up our training approach could well yield similar or better results.

EvoSuite

EvoSuite performs the worst in all three settings. Listing 4.4 shows the EvoSuite completion for the bank class. The generated test uses very poor naming conventions, such as naming the method `test0`, and each of the variables `bank0`, `customer0`, and `account0`. The deposit amounts do not make logical sense, as they are not rounded to the nearest cent. There is also a timeout of 4000 milliseconds. Such timeouts are highly likely to lead to flaky tests, where this test might pass in one environment and timeout in a different environment. The other generations by EvoSuite, suffer similar problems, including lacking asserts and using spurious exception handling. Due to this lack of proper naming conventions and the use of trivial asserts, it is very difficult to understand what is being tested in EvoSuite's generation.

[6]In fact, a similar caveat applies to CodeGen, which we do outperform.

Both GPT-4 and CAT-LM generate **high quality tests**, checking for realistic situations with readable asserts. However CAT-LM struggles to generate meaningfully distinct tests in the extra test setting. CodeGen and StarCoder produces highly readable, but incorrect tests. EvoSuite struggles to generate meaningful tests; it uses **poor naming conventions** and **spurious exception handling**.

4.7 Related Work

Classical Test Generation: Classical test generation techniques employ both black-box and white-box techniques to generate test inputs and test code. Random/fuzzing techniques such as Randoop [156], aflplusplus [71] and honggfuzz use coverage to guide generation of test prefixes. Property testing tools such as Korat [39], QuickCheck [53] and Hypothesis [137] allow a developer to specify a set of properties and subsequently generates a suite of tests that test the specified properties. PeX [197] and Eclipser [51] use dynamic symbolic execution to reason about multiple program paths and generate interesting inputs. The core issue with fuzzing and classical test generation techniques is their reliance on program crashing or exceptional behavior in driving test generation [62], which limits the level of testing they provide. EvoSuite [72] addresses these challenges by using mutation testing to make the generated test suite compact, without losing coverage. However, EvoSuite generates tests that look “unnatural”, and significantly different from human tests, suffering from both stylistic and readability problems [40, 57, 174].

Neural Test Generation: More recently, neural test generation methods have been developed to generate more natural and human understandable tests. ConTest[205] makes use of a generic transformer model, using the tree representation of code to generate assert statements. ATLAS [219], ReAssert [224], AthenaTest [201] and TOGA [62] extend this work by leveraging the transformer architecture for this task. They show that their generated asserts are more natural and preferred by developers when comparing against existing tools such as EvoSuite. TeCo [150] expands the scope of test completion by completing statements in a test, one statement at a time. They leverage execution context and execution information to inform their prediction of the next statement, outperforming TOGA and ATLAS on a range of lexical metrics. While these neural approaches solve many of the readability issues of classical test generation approaches, they focus on generating individual statements in a test, which offers significantly less time saving benefits than generating entire tests.

Large Language Models of Code: Large language models (LLMs) can perform well across many tasks when prompted with instructions and examples [42, 200]. Codex [47] is an autoregressive (left to right generation) LLM with 12B parameters, fine-tuned from GPT-3 on 54 million GitHub Python repositories. CodeGen-16B, with which we compare, outperforms this model [151]. Later, unpublished, iterations of Codex have also been applied to commercial settings, powering GitHub’s Copilot [6]. TestPilot [184] uses Codex to generate unit tests. However, it requires significant volumes of documentation as input, which is often not available for open-source projects. While all of these models perform well at generating code, they are relatively poor (for their size) at generating *tests* for the code. These models are typically trained on a randomly shuffled corpus of entire files, and thus do not learn the alignment of tests to the code under test. We pretrained a comparatively small language model on a much more modest budget that explicitly learns to align code and the corresponding test files, which yields substantially better performance than modestly larger classically trained models.

4.8 Summary

Even automatically generated code requires verification. Developers use tests to verify the correctness of the code they write, however current AI-powered tools struggle to generate good tests for a code file because they are typically not trained to consider the corresponding code file. To overcome this challenge, we develop CAT-LM, a GPT-style language model with 2.7 Billion parameters that was pretrained using a novel signal that explicitly considers the mapping between code and test files when available. We elect to use a larger context window of 8,192 tokens, 4x more than typical code generation models, to ensure that code context is available when generating tests. We evaluate CAT-LM on both test method generation and test completion, with CAT-LM outperforming CodeGen, StarCoder, and TeCo state-of-the-art baselines, even with CodeGen and StarCoder baselines significantly larger training budgets and model sizes. We show that adding the additional context helps CAT-LM, with code context significantly improving both lexical and runtime metric performance. Overall, we highlight how incorporating domain knowledge, namely the relationship between code and test files, can be used to create more powerful models for automated test generation.

Data Availability: The model weights for CAT-LM, code and datasets for training and evaluating CAT-LM, results of additional experiments and comparison with TeCo and CodeGen are available at: DOI 10.5281/zenodo.7909299

4.9 Takeaway

While LLMs excel at code generation, they are limited in their ability to generate tests because of the way they are trained to generate individual code files independently, a standard practice in natural language processing. As a result, they *can not* consider the code under test context when generating the tests. CAT-LM addresses this challenge by explicitly considering the mapping between code and test files during training. This enables users to generate tests that align more coherently with the underlying code, thereby enhancing the quality of tests produced. Moreover, CAT-LM supports users in verifying both the code they write and that which is generated by other LLMs, ensuring a more comprehensive and reliable testing process.

```java
@Test
public void testCustomerSummary() {
    Bank bank = new Bank();
    assertEquals("Customer Summary", "Customer Summary",
        bank.customerSummary());
}
```

Listing 4.1: CAT-LM first test generation

```java
@Test
public void testBank() {
    Bank bank = new Bank();
    assertEquals(0, bank.getBalance(), DOUBLE_DELTA);
    bank.deposit(100);
    assertEquals(100, bank.getBalance(), DOUBLE_DELTA);
    bank.withdraw(50);
    assertEquals(50, bank.getBalance(), DOUBLE_DELTA);
}
```

Listing 4.2: CodeGen first test generation

```java
@Before
public void setUp() {
    bank = new Bank();
}

@Test
public void testAddCustomerAndCustomerSummary() {
    Customer customer1 = new Customer("John Doe");
    bank.addCustomer(customer1);

    String expectedSummary = "Customer Summary\n - John Doe (0 accounts)";
    String actualSummary = bank.customerSummary();

    assertEquals(expectedSummary, actualSummary);
}
```

Listing 4.3: GPT-4 first test generation

```java
@Test(timeout = 4000)
public void test0() throws Throwable  {
    Bank bank0 = new Bank();
    Customer customer0 = new Customer("v\"PD");
    bank0.addCustomer(customer0);
    Account account0 = new Account(0);
    account0.deposit(148.3628547);
    customer0.openAccount(account0);
    double double0 = bank0.totalInterestPaid();
    assertEquals(0.14836285470000002, double0, 0.01);
}
```

Listing 4.4: EvoSuite first test generation

Figure 4.8: Example first tests generated by CAT-LM, CodeGen, GPT-4, and EvoSuite. CAT-LM and GPT-4 both generate realistic and readable tests; EvoSuite struggles with poor naming conventions and unrealistic tests. CodeGen generates readable test cases, but hallucinates methods in the code under test.

Chapter 5

DiffSpec for Reliability

In Chapter 4, we showed that explicitly considering the mapping between code and test files during training helps generate tests that align more coherently with the underlying code, thereby enhancing the quality of tests produced with CAT-LM. With DiffSpec, we take that one step further and consider other software artifacts beyond just code under test, such as natural language specification documents, source code, existing tests, previous bug reports and so on. We show that using these artifacts can aid in generating targeted tests that can highlight meaningful behavioral differences in systems, that points to bugs in some cases, thereby improving the overall reliability of the software system.

Large Language Models (LLMs) excel at extracting and understanding information from large amounts of natural language text, enabling a wide variety of tasks such as program comprehension, bug localization, and software testing. Recent research has shown significant promise in leveraging LLMs to enhance various testing techniques. For instance, LLMs have been used to generate more effective mutations in mutation testing [60], to create higher-quality unit tests [20, 122, 130, 168], and to improve fuzzing methods by producing diverse and targeted inputs [226].

Given the natural language specification document, we can not only generate tests to check for code correctness but can also check for conformance. We aim to do so with differential testing, an approach for automatically generating potentially bug-finding tests for applications that correspond to multiple implementations of the same functionality [139]. The key idea is to test two or more different systems (or two different versions of the same system) that should behave the same way under the same conditions on the same inputs. If their output behavior differs, it is likely that at least one of the implementations is incorrect.

Differential testing has shown significant success especially in testing language implementations, such as uncovering bugs in C compilers [113, 229] or browser engines, for example revealing inconsistencies in JavaScript interpreters and JIT compilers [34]. It also can be useful for testing cross-platform consistency (i.e., the same system across different configurations or operating systems) [67] or versions (as in regression testing) [76].

Generating tests that specifically target differences between two versions of a program is especially challenging, as it involves simultaneously searching the vast input space of two programs to find rare inputs that trigger often subtle discrepancies [139]. Existing approaches to find such tests limit the possible search space by borrowing techniques from symbolic execution [179], guided semantic aware program generation [107], type aware mutations [99], and code coverage optimizations [50]. While some approaches leverage semantic and syntactic properties of the code or use information from static analysis tools, they are significantly limited in their ability to harness the wealth of information available from natural language artifacts.

In this work, we propose a differential testing technique that leverages natural language and code artifacts describing the software systems under test to inform and prompt an LLM to produce effective, and targeted, differential tests. The extracted information describes the system specification, its source code implementation, and historical bug information.

We realize this intuition in DIFFSPEC, a general approach to differential testing of multiple systems implemented with respect to a documented specification, and with functionality that can be decomposed into testable units. DIFFSPEC is well-suited for testing systems that correspond to, or integrally include, language compilers, runtimes, and verification systems, like network protocol parsers or JVM or EVM or web browsers. This is the predominant domain for differential testing applications in research [46, 120, 188] and practice [85]. Such systems are typically associated with comprehensive language specification documentation, like the Instruction Set Architecture (ISA) specification associated with eBPF [56], or the WebAssembly (Wasm) language specification [77]. Moreover, testing these types of systems can be naturally decomposed into testing language instructions or subsets thereof.

We demonstrate DIFFSPEC on various implementations of Wasm and eBPF runtimes, which both have rich evolving natural language artifacts that DIFFSPEC can leverage, and are widely used in practice. Both runtimes vary in domain, the type of contextual information that our approach must extract from the natural language artifacts, and the format and language of the tests to be generated (see Section 5.3), demonstrating the generalizability of DIFFSPEC.

Using DIFFSPEC, we found 299 differentiating tests across four different implementations of Wasm validators. Upon manual analysis, we found that these point to at least two bugs which includes a type mismatch and cast of out-of-bounds. These bugs were reported to the maintainers of Wasm and have now been fixed. We also generated 1901 differentiating tests, that helped discover at least four distinct bugs across three different implementations of eBPF runtimes. These include a kernel memory leak, inconsistent behavior in jump instructions, undefined behavior when using the stack pointer, and tests with infinite loops that hang the verifier in ebpf-for-windows. These bugs were confirmed by the contributors of eBPF and issues have been filed for them.

5.1 Illustrative Example

This section presents an example illustrating how DIFFSPEC uses natural language specifications and code artifacts to generate tests for eBPF. First, given a natural language specification document [], DIFFSPEC extracts a list of instructions in the underlying language, along with the corresponding constraints for each instruction. For example, constraints extracted for the `RSH` instruction include:

1. The RSH instruction performs a right shift operation. The destination register (dst) is shifted right by the number of bits specified in the source operand (src or imm).
2. The source operand can come from either the src register (if the source bit in the opcode is set to X) or the immediate value (if the source bit in the opcode is set to K).

3. For ALU: {RSH, K, ALU} means $dst = (u32)(dst >> imm)$ and {RSH, X, ALU} means $dst = (u32)(dst >> src)$.

Next, for each considered instruction DIFFSPEC extracts the implementation of that instruction from each of the two codebases corresponding to the systems under test. For example, DIFFSPEC extracts an implementation of `RSH` from the source code for eBPF in linux ARM 32, a subset of which includes:

```
/* ... */
/* dst = dst >> src */
case BPF_ALU | BPF_RSH | BPF_X:
  case BPF_ALU64 | BPF_RSH | BPF_X:
    switch (BPF_SRC(code)) {
      case BPF_X: /* Shift right by variable */
        emit_a32_alu_r64(is64,dst,src,ctx,BPF_OP(code));
        break;
      case BPF_K: /* Shift right by immediate value */
      if (unlikely(imm > 31))
        return -EINVAL;
/* ...continues, elided... */
```

Listing 5.1: RSH source code extracted from the bpf implementation from linux arm 32 implementation

Given two code snippets, DIFFSPEC then reasons about the implementation differences, such as “Checking of Immediate Value: The first implementation checks if the immediate value is greater than 31 or 63 for 32-bit and 64-bit operations respectively. The second implementation does not perform this check.”

DIFFSPEC additionally looks at historical bugs to generate a set of bug classes to guide test generation. For example, “Shift Operation Bug: These bugs occur when the JIT compiler incorrectly handles shift operations, especially when the shift amount is zero. Incorrect shift operations can lead to unexpected results, or in worst cases, hang the kernel.”

Using all extracted context, DIFFSPEC first generates a set of test descriptions that detail what the test should check for. This along with a set of hand written guidelines that provides instructions on what makes a valid test, is then used to generate the test code. Here is an example of the test description along with the corresponding test code generated by DIFFSPEC.

```
// Test with a zero-shift count.
// Check for edge cases where the shift count is zero.
// The source value should remain unchanged.
-- asm
mov %r0, 0x12345678
rsh %r0, 0
exit
-- result
0x12345678
```

Listing 5.2: Generated test code

Interestingly, there was a historical bug in the linux implementation of BPF.[1] The issue titled

[1]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bb9562cf5c67

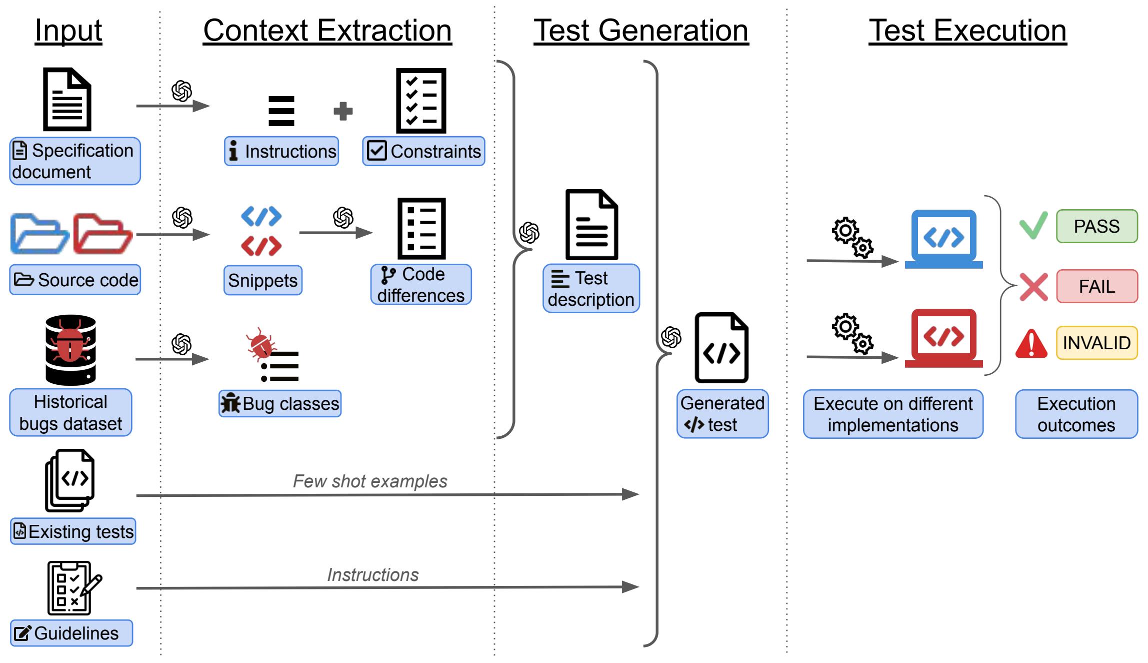


Figure 5.1: Approach overview. DIFFSPEC extracts relevant context from natural language and code artifacts by prompting an LLM, covering: instructions and constraints, source code and tests, and historical bug information. DIFFSPEC then follows a two-step process: (1) it generates test descriptions using the extracted context, and then (2) uses the test description along with few-shot examples of human written tests along with a set of human written guidelines to generate actual test code. Generated tests are executed on different implementations of the specification, seeking those that are potentially differentiating.

"arm, bpf: Fix bugs with ALU64 RSH, ARSH BPF_K shift by 0" describes how "The current arm BPF JIT does not correctly compile RSH or ARSH when the immediate shift amount is 0..." The test generated by DIFFSPEC would have caught this issue, preemptively.

5.2 Approach

Figure 8.1 provides an overview of how DIFFSPEC uses LLMs to generate tests using natural language specifications and code artifacts. The core components of DIFFSPEC are:

- **Extracting relevant context** from software artifacts (Section 5.2.1) by prompting an LLM. The information includes but is not limited to, for each instruction: relevant constraints; source code snippets from the tested implementations; and bug information.
- **Generating tests** (Section 5.2.2) using the extracted information as context to prompt an LLM to generate natural language descriptions of test cases. This description, along with relevant human-written tests (used as few-shot examples) and a set of curated instructions guide the model to generate actual test inputs.
- **Evaluating the systems** under test. Finally, we execute generated tests on the implementations under test with an evaluation harness. A generated test is *differentiating* if different im-

plementations produce different outputs for the same test input. Not all such tests correspond to defects, but they serve as a key starting point to identify potential problems in either the implementation(s), or the specification.

The rest of this section describes the first two components in detail. Evaluating the systems under test is straightforward conceptually; we provide implementation specifics with respect to our evaluation systems in Section 5.3. We elide certain prompts to reduce redundancy and improve flow; full prompts are available in the artifact. DOI 10.5281/zenodo.13836070

5.2.1 Extraction of relevant context from artifacts

Software systems are associated with many artifacts that encode desired and expected behavior as well as potential incorrect behavior. DIFFSPEC leverages these artifacts to guide the LLM to generate targeted tests that highlight informative behavioral differences between different implementations. The considered sources of information include (1) a system *specification*, which are typically semi-structured natural language documents describing what the multiple systems under test *should* do, (2) system *implementations* of the functionality under test, (3) system *tests* provided by developers, and (4) *historical bug information*, drawn from a bug report database or expert knowledge, that can provide clues about common failure modes. This results in the following types of information:

Instructions (ℹ) and Constraints (☑) Given a natural language specification document for a given system specification, DIFFSPEC first prompts the LLM to extract all instructions in the implemented language (ℹ).

Prompt to extract language instructions

```
Here is the official {language} specification document that details
the exact standards that need to be followed for {language}:
{manual}.  This document is used as a reference to implement the
{language} framework for different environments.  Extract all language
instructions from the document, following this format...
```

Specification documents describe the specified behavior of each instruction in a language, including constraints (☑) on correct implementation. DIFFSPEC therefore prompts an LLM to extract this information for each instruction, again from the specification document. Language specifications can be dozens to hundreds of pages long; extracting only the text that describes each instruction allows subsequent prompts to be specific, and overcomes LLM context window limitations.

Prompt to extract relevant specifications for each instruction

```
You are an expert in {language}.  Here is the official {language}
document that details the exact standards that need to be followed
for {language}:  {manual} This document is used as a reference to
implement the {language} framework for different environments.  The
goal is to find differential tests that returns different outputs
in different implementations.  Extract the key points relevant to
```

```
the {instruction} instruction from the documentation.  Make sure
you are extremely specific, and extract all the constraints and
conditions that strictly need to be followed when implementing the
{instruction} instruction..include details that can be easily missed or
misinterpreted...Do not include example tests.
```

Implementation code (📂) Code-based artifacts can provide useful guidance to differential test generation. First, for each instruction, DIFFSPEC uses its extracted constraints as a guide to identify and extract relevant source code snippets from each implementation. Although implementations of a given specification vary, they are expected by construction to be semantically equivalent. We hypothesize that implementation differences can be used to guide the LLM to generate tests that may result in differential behavior. In order to get these differences we first extract the relevant code pertaining to the given instruction by prompting the LLM with the source code files and the list of constraints. DIFFSPEC then uses the extracted code snippets from each implementation to prompt the LLM to reason about differences, producing a list in natural language (⑂).

Prompt to extract relevant code for given instruction from source code file

```
You are an expert in {language}.  Here is the source code of the
{language} for {implementation} architecture.  {codefile}.  Here are
the key points relevant to the {instruction} instruction, along with
a set of constraints that must be strictly followed when implementing
it.  {instruction_constraints}.  Using these points as reference,
extract the relevant code for {instruction} instruction from the code
implementation.
```

Prompt to extract code differences

```
You are an expert in {language}.  The goal is to find differential
tests that returns different outputs in different implementations
of {language}.  You are provided with the key points relevant to the
{instruction} instruction, along with a set of constraints that must
be strictly followed when implementing it.  {instruction_constraints}.
You are provided with the relevant code implementing the {instruction}
instruction in two different implementations:  {code, associated with
implementations} Identify differences in the two code implementations.
```

LLMs have been shown effective at generating summaries, including code summaries [192]. Inspired by the self-debug [49] work, which uses an LLM to explain the code and compares the natural language summary to the problem description to find bugs, we explore an alternative approach to extract information from the code snippets by having the LLM describe what the code is doing in natural language. We then use the code descriptions ((⧉) of both the implementations to guide the test generation process. We take this one step further and also have the LLM reason about the differences in the two descriptions (◫). We compare the efficacy of these approaches in Table 5.2.

Prompt to extract code description

```
You are an expert in {language}.  The goal is to find differential
tests that returns different outputs in different implementations
of {language}.  You are provided with the key points relevant to the
{instruction} instruction, along with a set of constraints that must
be strictly followed when implementing it.  {instruction_constraints}.
You are provided with the relevant code implementing the {instruction}
instruction in implementation implementations:  {code}.  Using the
natural language constraints as reference, can you provide a line by
line explanation of the code implementing the instruction instruction.
```

Prompt to extract differences in code descriptions

```
You are an expert in {language}.  The goal is to find differential
tests that returns different outputs in different implementations
of {language}.  You are provided with the key points relevant to the
{instruction} instruction, along with a set of constraints that must
be strictly followed when implementing it.  {instruction_constraints}.
You are provided with a detailed line by line description
of the code implementing the instruction instruction in two
different implementations:  {code description, associated with
implementations}Identify differences in the two implementations based on
the natural language descriptions of the code and return the output as
a list of differences between the two implementations.
```

Test code () Mature software systems include human-written test suites that exercise functionality that DIFFSPEC also targets, like specific instruction implementations. DIFFSPEC generates a mapping between each instruction and any human written tests for it by providing the LLM lists of instructions, and test file names. The mapping provides two main benefits. First, it helps identify instructions that don't have a corresponding test, highlighting gaps. Second, the mapping provides examples for test generation.

Prompt to map the human-written tests to instructions

```
You are given a list of instructions, along with a list of tests.
Create a mapping between the list of instructions and their
corresponding and output the results as a table with the following
format:  {format} List all tests that you did not find a mapping for
under UNKNOWN instruction.  Here is an example:  {example} Here are
the list of instructions:  {instructions_list}.  Here are the tests:
{human_test_files}
```

Bug categories () Finally, historical bug information can help provide useful context on common edge cases or failure modes. DIFFSPEC can make use of multiple sources of such information including bug reports, prior empirical studies, and expert knowledge; it queries an LLM to distill the information accordingly.

Prompt to generate bug categories

```
You are an expert in {language}.  Here is a list of commit
messages corresponding to previously identified bugs in {system}
.  {bug_commits}.  Provide a list of descriptive categories for
the different kinds of bugs that can occur in {system}.  Include a
description of what each category means.
```

5.2.2 Test Generation Framework

DIFFSPEC uses the context extracted from the natural language and code artifacts to generate differential tests. It does this in two phases:

Generating test descriptions (☰) DIFFSPEC incorporates the specifications and constraints for each tested instruction, along with all the additional extracted context, to prompt the LLM to generate a configurable number (we use 10 in our experiments) of natural language descriptions for what a test should do. This step is repeated for every combination of extracted code difference and bug category.

Prompt to generate test descriptions

```
You are an expert in {language}.  The goal is to find differential
tests that returns different outputs in different implementations
of {language}.  You are provided with the key points relevant to the
{instruction} instruction, along with a set of constraints ...  You are
provided with a key difference in the code implementations ...  Your
goal is to generate {number} unique differential tests so we can test
for this difference in the implementation in {implementations}.  Focus
on {bug_class} when generating the tests.  {bug_description}.  Generate
a natural language description of tests that can result in differential
behaviour...  Make sure to reason about the specific constraint that
the test is checking for and include information on how the test
relates to the {bug_class}.
```

Generating tests (</>) Given a natural language test description, DIFFSPEC next prompts the LLM to convert these into executable tests. The generated tests include expected output. We also include an optional set of guidelines (✎) for valid tests for the system, provided by a human expert; we evaluate its contribution to test validity in Section 5.5.2. In our analysis, the two-phase approach is generally more effective for generating syntactically valid code than a single-phase approach (See Table 5.2).

Prompt to generate test

```
You are an expert in {language}.  The goal is to find differential
tests that returns different outputs in different implementations
of {language}.  You are provided with the key points relevant to
the {instruction} instruction, along with a set of constraints...
```

Table 5.1: Selection of bug categories and descriptions for eBPF and Wasm

Target	*Category* - Description
eBPF	***Instruction Encoding*** - Incorrect assembly code generation by the eBPF JIT compiler, involving incorrect opcodes, registers, or misinterpreting eBPF instructions. ***Stack Layout*** - Incorrect stack frame setup or teardown, causing memory corruption. ***Shift Operation*** - Incorrect handling of shift operations, e.g. shift by 0 errors, leading to unexpected results, or hanging the kernel. ***Register Handling*** - Incorrect usage, saving, or restoration of CPU registers. ***Endianness Conversion*** - Incorrect conversions from big and little endian representations.
Wasm	***Branch Target Resolution*** - Branching instructions rely on specifying labels to determine the branch target. A bug in resolving these labels can cause the control flow to jump to the wrong location, leading to unexpected behavior. ***Block Nesting*** - Failure to correctly manage block nested structures, leading to incorrect flow of control. For instance, a `br_table` instruction that incorrectly interprets the depth of nested blocks can cause the control flow to exit the wrong block or loop. ***Type Mismatch in Control Flow*** - Control flow instructions must adhere to specific type constraints. otherwise leading to runtime type errors. ***Control Flow Across Module Boundaries*** - Wasm modules can import and export functions, and bugs can occur if control flow instructions don't correctly handle calls or returns across module boundaries.

```
{instruction_constraints}. Your goal is to generate differential
tests so we can test for nuances in the two implementations... Here
are some examples of existing tests. {example_tests}. Here is
a description of a test that can result in differential behavior
in the two implementations for the {instruction} instruction.
{nl_test_description}. Generate the code for the test in the same
format as the example. You are required to strictly follow these
instructions when generating the test. {optional_human_instructions}
```

5.3 Systems Under test

DIFFSPEC is a framework that can apply to many different systems for which differential testing is appropriate. For the purposes of evaluation, we apply it to two complex real-world problem specifications and several associated implementations: the extended Berkeley Packet Filter (eBPF) (Section 5.3.1), a kernel-extension framework that safely runs custom bytecode programs; and WebAssembly (Wasm) (Section 5.3.2), a portable bytecode language and compilation target originally designed for browser-based applications but with broad application beyond. This Section describes these systems, with particular focus on details relevant to our evaluation.

5.3.1 extended Berkeley Packet Filter (eBPF)

Background. The extended Berkeley Packet Filter, or eBPF, is a kernel-extension framework originally integrated into the Linux kernel at version 3.18 [54]. It allows developers to safely run custom bytecode programs inside the kernel, without inserting risky modules or modifying the kernel itself. A key component of eBPF ecosystem is the verifier, which ensures safety properties (like memory safety, crash-freedom, or termination) of user-defined extensions. Support for eBPF framework has been implemented for multiple runtime environments and architectures (including but not limited to x86_64, ARM, Risc-V). There furthermore exists several user-space eBPF runtime implementations [65] that extend its reach beyond kernel-level interactions.

Tested Implementations and Code artifacts. We test three eBPF runtimes in our experiments: (1) Linux Kernel via Libbpf [127], a user-space library that simplifies the use of eBPF programs in the Linux kernel. (2) Userspace BPF (uBPF) [98], a lightweight, user-space implementation of eBPF. (2) eBPF for Windows via bpf2c [140], which Microsoft has introduced into the Windows ecosystem, again providing a user-space eBPF runtime.

We take the 206 human-written tests for `bpf_conformance` for the test artifacts.

Natural language Artifacts. All runtime implementations of eBPF must conform to the eBPF Instruction Set Architecture [56], standardized and documented through the IETF, and the authoritative source for the specification standard. We use the BPF ISA as the specification document for testing; DIFFSPEC extracts a total of 34 instructions from the ISA, including arithmetic, jump, load, and store instructions. For historical bug information, we collect 55 historical bug reports from prior work [148]. We use the commit title and description of the bug fixes in the linux implementation of eBPF as input and prompt the LLM to group the bugs into high level categories and include a description for each category. Table 5.1 shows a subset of bug categories generated for eBPF.

Evaluation Harness. We run generated tests using the `bpf_conformance` plugin. The BPF Conformance project [141] aims to measure the conformance of BPF runtimes to the ISA by providing a unified testing interface. The possible test execution outcomes are:

- **`PASS`**: "Test succeeded". The test is valid and the execution produced the expected return value.
- **`FAIL`**: "Plugin returned incorrect return value x expected y." The test is valid but does not pass. This can happen either because there is a bug in the tested implementation, or the LLM generated the incorrect expected output. The differential testing context means that DIFFSPEC does not rely solely on the LLM to adjudicate expected behavior, instead comparing the output of multiple implementations.
- **`SKIP`**: "Test file contains unsupported instructions/has no BPF instructions." The test is not a valid BPF program.
- **`ERROR`**: "Plugin returned error code 1 and output <msg>." The test is again an invalid program producing an error that the conformance plugin can handle, like referencing an invalid register ID in a program instruction.
- **`CRASH`**: "Unhandled Exception reached the top of main: <msg>." The test is invalid in a way that causes the conformance plugin to crash (such as an instruction referencing an invalid label).

5.3.2 WebAssembly (Wasm)

Background. WebAssembly [78], or Wasm, is a portable bytecode that was originally built to run in the browser, but has since become popular in more domains such as cloud and edge computing [92, 167, 194], embedded systems [210], industrial systems [145], and more. Wasm provides a safe runtime for untrusted code in many languages (Rust, C/C++, OCaml, and others) as memory is sandboxed; it is also highly performant.

A Wasm application, consisting of one-to-many *modules*, is run on a Virtual Machine (VM), also referred to as engine or runtime. There are many available implementations including, Wizard Engine [198], for teaching and research; Wasmtime [14], owned by the Bytecode Alliance, for Edge Computing; and V8 [10], used in Google Chrome, especially noted for its JavaScript integraion.

Before a Wasm module is executed, it is *validated* by the wasm *validator*; this step protects the host system from security vulnerabilities, runtime traps, and undefined behavior. Generally, each Wasm VM has its own custom Validator, simplifying integration, and improving performance. This motivates standardized testing and robust tooling to ensure runtime conformance. An established format for testing Wasm runtimes is via `.wast` tests written in the human-readable WebAssembly Text (WAT) format.

Tested implementations and code artifacts. Given the importance of validation to runtime conformance, we focus our testing on the validator modules of four Wasm implementations: (1) Wasm spec [12] (the reference implementation, considered the oracle for Wasm behavior), (2) Wizard Engine [198], (3) Wasmtime [14], which uses `wasmparser`'s [13] validator under-the-hood, and (4) V8 [10]. We take tests from the official Wasm test suite [12]. Note that DIFFSPEC was only provided the source code context for the Wasm spec and the Wizard Engine. We test other implementations to deem if the generated tests are useful without further prompting or context.

Natural language artifacts. We use the Wasm language specification document [77] for testing. Focusing especially on control-flow instructions, known to be both tricky and error prone, we extract 11 instructions to test from this documentation. DIFFSPEC identifies 11 test files with 596 test cases (from the Wasm spec codebase) for these instructions. We get an initial list of bug categories by prompting ChatGPT (GPT 3.5). This list was then verified by a maintainer from Wasm. Table 5.1 shows a subset of bug categories inferred for Wasm.

Evaluation Harness. DIFFSPEC generated tests for Wasm targeting the `.wast` format (examples in Table 5.5). For the implementations expecting a different format (like Wizard engine, expecting `bin.wast`), we use the wasm-spec CLI to translate accordingly and automatically. Note that invalid or syntactically incorrect tests fail this transformation step. Additionally, the tested systems do not all report errors with equal precision or granularity. For consistency, we therefore check for a simple PASS/FAIL result, ignoring the error message produced by test execution. We also refactored multi-assertion test files to provide one assertion per test, for cleaner comparisons of testing results. We evaluated execution results by comparing to the Wasm spec reference implementation. The possible test execution outcomes are:

- **`PASS`**: A validator successfully labeled an invalid module as invalid (as expected by the `asserts` in the `.wast`).
- **`FAIL`**: The validator labeled an invalid module as valid (as this behavior does not satisfy the

`asserts` in the `.wast`).
- **`CRASH`**: An Exception was thrown during execution and printed to the console.
- **`INVALID`**: The tests fail to convert into desired format.

5.4 Experimental Design

This section describes our experimental setup for evaluating DiffSpec on our systems under test. We implement DiffSpec in Python. We make use of GPT-4-32k as the LLM for all the eBPF experiments. Specifically, we use the 0613 version of the GPT-4-32k checkpoint through the Azure OpenAI API. The Wasm experiments have been run using GPT-4o, since the GPT-4-32k model endpoints were deprecated. We use the default values for all the hyper-parameters for both GPT-4-32k and GPT-4o. Specifically, we investigate the following research questions:
- **RQ1** How effectively does DiffSpec produce meaningful differential tests? We evaluate this question initially on eBPF (with results for Wasm discussed in RQ3).
- **RQ2** To what extent does each component of DiffSpec contribute to its effectiveness?
- **RQ3** How well does DiffSpec generalize across systems?

Finally, we qualitatively examine the generated differential tests to determine the cause of differential behavior and possibility of bug attribution.

5.4.1 LLM-based Baselines

Both eBPF and Wasm have been extensively tested using traditional fuzzers. However, these do not check for conformance with a specification. We define two baselines that explicitly consider the specification document. First, we use a naive prompting technique that provides the entire specification document and three randomly-selected human tests as context, and prompt the LLM to generate tests for each instruction. Second, as a more targeted baseline, we extract the most relevant sections from the document for each instruction. We then use the mapping between the instructions and the test files to only use the tests corresponding to the given instruction as examples in the generation prompt.

We also consider Fuzz4All [226], a fuzzing technique that generates and mutates test inputs for projects written in different programming languages. Fuzz4All leverages a larger model, GPT4, to automatically generate prompts for semantically interesting and syntactically valid input, and a cheaper model, StarCoder to generate and mutate these inputs. To apply Fuzz4All's autoprompting and LLM-fuzzing loop to eBPF, we extend it by adding: (1) an interface to integrate the execution of eBPF bytecode through the Linux libBPF plugin, (2) custom functions to filter, clean, and transform LLM generated output into the required eBPF test format. To fuzz eBPF bytecode instructions, we provide Fuzz4All with different sections of the BPF ISA and 3 hand-selected examples of valid bytecode tests for each section. For each section of the ISA, we run experiments with 1000 fuzzing iterations using the best performing configuration reported in the paper. In total, we generate 4,000 tests, using the libBPF plugin to provide evaluation feedback to the auto-prompting and fuzzing loops. Despite having access to few shot examples and instruction semantics from the ISA, 61% of generated tests did not include valid BPF semantics (e.g., missing exit instructions, hallucinated instruction codes). Considering the remaining 39%

of tests, 38.5% did not follow the expected test format (e.g. not including –asm or –result), and 0.5% of tests were successfully parsed but all failed. Upon manual inspection, none of the generated tests passed because the expected result was either empty or invalid. We require valid tests to check for differential behavior, and therefore could not use these generated tests as a baseline. The code and tests generated with Fuzz4All have been included in the artifact.

5.4.2 Fuzzing Baselines

We implement two fuzzing baselines to evaluate how LLM-guided test case synthesis (with DIFFSPEC) compares to more traditional random test case generation. In order to provide a fair comparison, we ensure that both fuzzing baselines are *grammar-aware*, generating *syntactically-valid* eBPF and Wasm programs (but not necessarily semantically-valid).

For the Wasm fuzzer, we use GRAMMARINATOR [90] to generate syntactically-valid WAT syntax trees according to a reference ANTLR specification. For the eBPF fuzzer, we randomly sample valid eBPF opcodes for instructions and fill their arguments with random values. For a fair evaluation, we ran the fuzzer for the same amount of time it took DiffSpec to generate and run tests for both wasm and eBPF. This resulted in 300k tests for wasm run over ~ 6 hours and 200k tests for eBPF run over ~ 12 hours.

5.4.3 Evaluation Metrics (RQ1)

We use the following metrics in evaluating DIFFSPEC performance:

Validity. A generated test is *valid* if it results in `PASS`, `FAIL`, or `ERROR` across all implementations — that is, DIFFSPEC produced a syntactically valid test. Tests that are skipped or lead to crashes are considered invalid.

Differentiating tests. A *differentiating test* is one that produces a different outcome/return value for at least two different implementations. For example, a test resulting in `PASS` on one system and `FAIL` or `ERROR` on another, or a test resulting in `FAIL` on two systems but return different values that do not match the test's expected value. Note that it is still possible for the generated expected value to be incorrect. However, this risk is mitigated by the use of multiple systems in the differential testing context, where their outputs can be compared independently of the test's oracle value.

Test complexity. Automatically generated tests can be difficult for humans to understand or use [185]. This can be especially risky in the differential testing context [104, 229]. We use the number of lines in the tests as a simple proxy for test complexity.

Test diversity. To assess the ability of DIFFSPEC to generate a *diverse* set of tests covering many different behaviors, we examine the generated tests and compute the unique number of features across the set. For eBPF we consider: unique instructions (e.g. eBPF opcodes), unique registers (e.g. `%r0`, `%r1`, ...), unique memory addresses (e.g. `[%r3]` or `[%r1+4]`), and unique immediate values (e.g. `0x1`, `0x1337`, ...).

5.4.4 Configurations for ablation (RQ2)

We evaluated the performance of DiffSpec under multiple configurations (summarized in Table 5.2) to evaluate the contribution of the context extracted from the different artifacts, and the effect of the two-stage test generation procedure. These settings are:
3-shot-random: Our first baseline; uses the instruction (ℹ), the language specification (🗎), and three random test examples (🗋) to directly generate tests (</>).
target-section: Our second baseline; uses the instruction (ℹ), the manual section under which the instruction was listed (▤), and targeted examples (🗋) from the test suite to directly generate tests(</>).
prompt-chain: Asks an LLM to extract the key information about an instruction from the manual (☑), and uses a two-step test generation process, combined using prompt chaining. We include three random test examples (🗋) when generating the test (</>) from the test descriptions (≣).
prompt-chain-instruct: Provides a curated set of guidelines to ensure valid tests, such as "ensure that the output is in %r0" for eBPF. We provide these guidelines (✎) during test generation, on top of the **prompt-chain** approach.
bug-centric: Builds on prompt-chain-instruct setup by additionally providing the LLM with cues from historical bug data (🐞).
code-description: Builds on prompt-chain-instruct setup by additionally extracting and summarizing relevant code snippets (⧉).
code-description-diff: Builds on code-description by generating a list of implementation differences (◫) in the two code descriptions.
code-diff: We observed that using the differences in descriptions (◫) can help the LLM generate differentiating tests. In this setup, we have the LLM generate a list of differences directly from the relevant code snippets (⑂) extracted from the implementations. We then use each difference that was generated to guide the LLM when generating test descriptions (uses two step test generation).
bug-guided-code-diff, or DiffSpec: Combines all useful elements: prompt-chain-instruct (☑ ✎ 🗋) with bug categories (🐞) and code differences (⑂), with a two step generation process (≣ </>).

Note that we conducted these experiments first on the eBPF system; we then used the configuration that led to the most differential tests to subsequently test Wasm validators.

5.5 Results

This section presents experimental results, speaking to DiffSpec effectiveness (Section 5.5.1); the contributions of individual design decisions and types of information to that effectiveness (Section 5.5.2); and DiffSpec's generalizability to diverse domains (Section 5.5.3).

5.5.1 RQ1: Overall effectiveness

Table 5.2 summarizes the results of running DiffSpec on the eBPF systems, as discussed in Section 5.3.1. The first three rows of the tables show baselines; the final row, the results for DiffSpec (the intermediate rows are discussed in Section 5.5.2).

Table 5.2: Performance of DiffSpec compared against baselines (first 3 rows) and various ablations on eBPF runtimes. Legend: instruction: ℹ, specification document: 📄, relevant section of specification document: ▤, constraints: ☑, example tests: ⟨/⟩, test descriptions: ≡, guidelines: ✎, bug classes: 🐞, code descriptions: ⧉, code descriptions diff: ◫, code diffs: ⑂, W: windows, L: linux, U: ubpf, †: fuzzer generates all possible instruction variants and millions of unique addresses/values

Ablation type	Context used	Validity (%)	*Test Diversity (Unique #)*				*Differential Tests Found*			
			Instr.	Reg.	Addr.	Values	W-L	L-U	W-U	Total
fuzzing	ℹ	100.0	†	†	†	†	1	1	2	**4**
3shot-random	ℹ 📄 ⟨/⟩	68.3	25	2	0	79	14	13	4	**14**
target-section	ℹ ▤ ⟨/⟩	76.3	52	8	14	91	39	37	3	**39**
prompt-chain	ℹ ☑ ⟨/⟩ ≡	24.8	34	12	2	118	23	21	6	**24**
prompt-chain-instruct	ℹ ☑ ⟨/⟩ ≡ ✎	66.3	46	13	5	181	38	34	9	**38**
bug-centric	ℹ ☑ ⟨/⟩ ≡ ✎ 🐞	64.8	64	15	26	500	266	251	53	**271**
code-description	ℹ ☑ ⟨/⟩ ≡ ✎ ⧉	63.4	45	13	8	189	29	29	3	**29**
code-description-diff	ℹ ☑ ⟨/⟩ ≡ ✎ ◫	65.3	63	13	25	446	158	137	32	**159**
code-diff	ℹ ☑ ⟨/⟩ ≡ ✎ ⑂	66.5	61	16	23	404	198	139	79	**200**
DiffSpec: bug-guided-code-diff	ℹ ☑ ⟨/⟩ ≡ ✎ 🐞 ⑂	69.4	87	16	106	1850	1886	1790	226	**1901**

The fuzzing baseline is designed to generate *syntactically valid* tests and therefore has a 100% validity. The two LLM-based baselines also generate tests with high validity (68% and 76%) since they primarily test for simple cases. DiffSpec's integration of bug category and code differences improves test validity (69%) over other ablations, while identifying many more behavioral differences.

Additionally, looking at number of differential tests found, we see that despite both fuzzing and LLM-based baseline approaches having high validity rates, they generate very few differential tests (4, 27, and 75). In contrast, DiffSpec identifies 1901 differential tests, while having comparable validity. We manually analyze a random sample of 200 differential tests to identify potential bugs in eBPF. Note that there can be multiple differentiating tests pointing to the same underlying bug. Using this analysis, we identify 4 concrete bugs for eBPF, and have filed reports with the maintainers of the associated implementations. All of them have been confirmed as real bugs. For eBPF, we observe the following classes of tests that demonstrate meaningful behavioral differences among different implementations; Table 5.3. shows examples of tests corresponding to these categories, which are:

- Uses uninitialized registers, includes not storing output in r0.
- Undefined behaviour when using stack pointers (%r10).[2].
- Tests resulting in potential memory leaks.[3].
- Inconsistencies in how jump instructions are handled.[4].
- Tests containing call instructions to helper functions (differential behavior is expected).

[2]https://github.com/Alan-Jowett/bpf_conformance/issues/293
[3]https://github.com/Alan-Jowett/bpf_conformance/issues/294
[4]https://github.com/Alan-Jowett/bpf_conformance/issues/292

Table 5.3: Examples of differentiating tests generated by DiffSpec for eBPF runtimes that identified bugs

Category	Generated Test	Execution Outcomes

Kernel Memory Leak	`-- asm`
	`ldxw %r0, [%r1]`
	`exit`
	`-- mem`
	`00 00 00 00`
	`-- result`
	`0x0`
		Windows: PASS: Test succeeded
		uBPF: PASS: Test succeeded
		Linux: FAIL: Plugin returned incorrect return value ffff8b09dc604100 expected 0

Inconsistencies in jump	`-- asm`
	`mov %r1, 5`
	`jset %r1, %r1,`
	`lbl1`
	`mov %r0, 0`
	`exit`
	`lbl1:  mov %r0, 1`
	`exit`
	`-- result`
	`0x1`
		Windows: FAIL: Plugin returned incorrect return value 0 expected 1
		uBPF: FAIL: Plugin returned incorrect return value 7ffff338a820 expected 1
		Linux: ERROR: Plugin returned error code 1

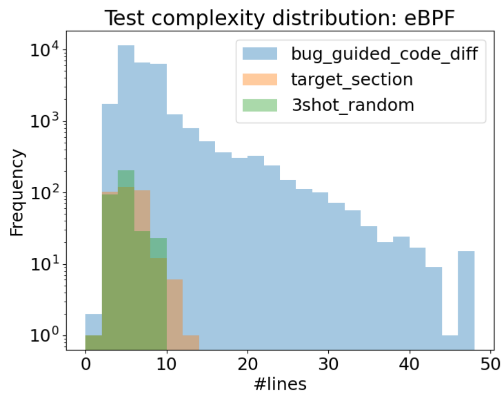
- Tests with infinite loops that causes the ebpf-for-windows verifier to hang. [5]

Using DiffSpec, we were able to generate differential tests that uncovered four different bugs in the different implementations of eBPF, despite many of these being extensively tested by fuzzers and other techniques. These bugs have been confirmed by the maintainers of the bpf conformance project and are currently in the process of being fixed.

Test complexity. Figure 5.2 shows the distribution of test complexity of the tests generated by DiffSpec for eBPF. We observe similar trends for Wasm. For eBPF, we additionally compare the complexity of the tests generated by DiffSpec with the two baseline approaches. We find that overall, DiffSpec generates short tests that average fewer than 20 lines. Additionally, we observe that the tests generated by the baseline approaches are much shorter than the ones that DiffSpec generates, which highlights that DiffSpec generates more complex tests that can find differentiating behavior in different implementations. Despite the baselines having a higher validity rate, these valid tests are not effective at finding interesting differential behavior.

Test diversity. Our test diversity results (Table 5.2) show an interesting story. Both the LLM baselines have lower unique numbers of program features such as types of instructions and registers. As we include more context, the resulting test cases cover more of the possible eBPF instruction variants and test more modes of operation (such as using different memory addresses),

[5] https://github.com/vbpf/ebpf-verifier/issues/783



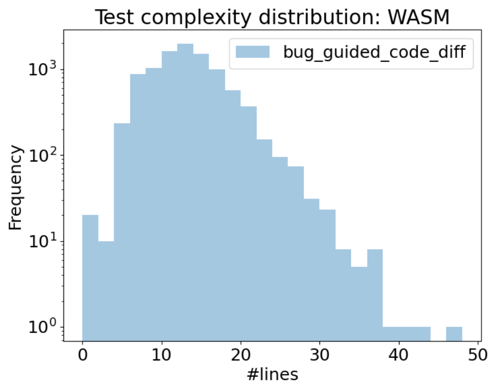


Figure 5.2: The distribution of generated test complexity (measured by test length) for eBPF (left) and Wasm (right). The bug-guided-code-diff (DIFFSPEC) generates a more complex distribution of tests.

correlated with an increase in the number of differentiating test cases. Interestingly however, the fuzzing baseline covers far more possible program variants (by orders of magnitude), yet finds the *fewest* number of differentiating tests. These results suggest that syntax-adherence and diversity alone is insufficient to find interesting tests, and *guidance*, for example though the use of specification, source code, guidelines, etc., is *critical* to exercise interesting behavior.

Instruction-level performance Figure 5.3 shows the distribution of the test status or execution outcome of all the generated tests for each instruction using the bug-guided-code-diff approach on the windows implementation of eBPF. Upon closer analysis, we find that the DIFFSPEC generates mostly valid tests (that PASS or FAIL) for most arithmetic and logic instructions. The percentage of valid tests declines for more complex instructions like the jump instructions. On the other hand, load/store instructions, namely, `LD`, `ST`, `LDX`, `STX`, along with `END` and `MOVSX`, have the highest invalid test rate (tests either CRASH or throw an ERROR). This gives us insights into the types of instructions that LLMs can and cannot reason about. We observe similar trends when the generated tests are run on other implementations of eBPF using the different ablations. Additional plots for different ablations and different implementations can be found in the supplementary material.

Note on specification and code coverage We manually confirmed the model successfully extracted (and generated tests for) all instructions in each language. We also manually confirmed all specifications related to a given instruction were correctly extracted for a small sample. These judgments informally suggest good specification coverage. Doing this completely is likely infeasible since the specifications are long. Given the differential testing goal (to find bugs by comparing implementations), we consider observed differential behavior a more instructive metric than code coverage. We don't expect much coverage improvement, for such well-tested projects, with a technique that doesn't target it.

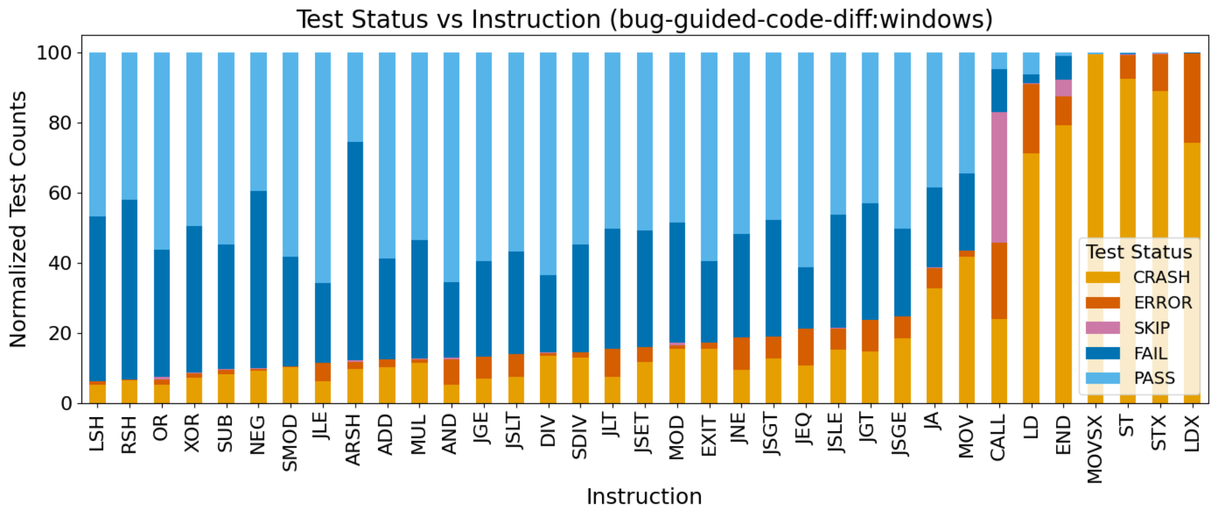


Figure 5.3: Visualization of test status distribution per instruction for eBPF using DIFFSPEC on Windows.

5.5.2 RQ2: Ablations

Table 5.2 also summarizes the results of the ablations for eBPF. As above, targeted prompting has the highest validity rate, but the generated tests are often simple. Meanwhile, comparing prompt-chain with prompt-chain-instruct, we find that the human written instructions/ guidelines helps improve validity by a huge margin of 42%. Bug context alone generates complex tests, and having code context (in all forms) helps improve the validity of the tests. On the other hand, using the bug categories and differences in code are extremely useful for generating differentiating tests, with the number of differentiating tests going up to 271 and 200 respectively.

Overall, most ablations that only look at the natural language specifications document and existing tests generate fewer than 40 differentiating tests. Interestingly, while the differences in code descriptions are seemingly useful, with 159 differentiating tests generated, the code descriptions on their own are not very useful, resulting in only 29 differentiating tests.

The bug and code diff context helps generate more complex tests that identify differential behavior in different implementations. Guiding the model with bug categories, and differences in code are especially useful. A two-step test generation process with the human written instructions is more effective.

5.5.3 RQ3: Generalizability to Wasm

We demonstrate DIFFSPEC's ability to generalize beyond eBPF by evaluating its performance on Wasm validators. We use the best combination of features, substantiated by the ablation study, for these experiments. With respect to validity, using the Wasm spec's translation facilities as a proxy, 85% of the generated tests were valid (could be translated).

Table 5.4 summarizes the remaining results. We find a total of 74 differentiating tests with

Table 5.4: DiffSpec vs. fuzzing baseline performance on Wasm validators, compared to the Wasm spec reference implementation.

Comparison	***Differential Tests Found***	
	DiffSpec	**Fuzzing baseline**
Wizard Engine vs. Wasm spec	6	74
Wasmtime vs. Wasm spec	256	0
V8 vs. Wasm spec	37	0
Total	299	74

the fuzzing baseline. However, upon manual inspection, we found that the Wizard engine imposes a hard limit on memory and table size (capped at 10000000) during validation, whereas the spec interpreter does not. Therefore, differential behavior is expected, finding no underlying bug. On the other hand, DiffSpec produced a total of 299 differentiating tests across the four different implementations of Wasm validators. Upon manual analysis, we observe the following classes of tests that demonstrate behavioral differences among different implementations for Wasm validators, and therefore highlight potential bugs:

- Tests with type mismatch (expected vs. actual type).
- Tests with invalid type (extremely large numbers).
- Implementations had different semantics for `.wast` assertions. The `assert_malformed` and `assert_invalid` cases are treated differently on the Wasm spec, but the same on all other implementations (differential behavior is expected).
- Implementations had different semantics for handling unavailable imports, error vs. validate what is available (differential behavior is expected).

The two bugs we identified, (i) *Type Mismatch*: Validate that `[return_]call_indirect` operates on a table with `funcref`,[6], and (ii) *Unknown Type*: Fix cast of out-of-bounds values, [7], have been reported to project maintainers, and have since been fixed (as of September 2024). Table 5.5 shows the tests, which have also been added to test suite of both Wizard and wasm-spec.[8]

The differential tests generated by DiffSpec uncovered 2 different bugs in the Wizard Engine validator, despite being extensively tested. These bugs were reported to the maintainers and have now been fixed. This speaks to DiffSpec's ability to differentially test a variety of systems.

5.6 Related Work

Testing eBPF: There exists a large body of work on improving eBPF verifiers and JIT compilers through fuzzing [94, 97, 123, 208], state embeddings [190], or rewriting the verifier with proof-

[6]https://github.com/titzer/wizard-engine/commit/33b58749895dc5afdd2c032a31d1848475f04ce2
[7]https://github.com/titzer/wizard-engine/commit/4ad7eca1b98146fa4c8884b32c7e99c41f5d5a81
[8]https://github.com/WebAssembly/spec/pull/1822

Table 5.5: Differentiating tests generated by DIFFSPEC for Wasm validators that identified bugs.

Category	Generated Test	Execution Outcomes
Unknown type	`(assert_invalid (module(type` `(func (param i32)))` `(table 1 funcref)` `(func $conditional-dangling-type` `(if (i32.const 1)` `(then (call_indirect (type` `0xffffffff) (i32.const 0))))))` `"unknown type")`	wasm-spec: PASS wizard-engine: CRASH
Type mismatch	`(assert_invalid (module (type` `(func))` `(table 10 externref)` `(func $call-indirect` `(call_indirect` `(type 0) (i32.const 0))))` `"type mismatch")`	wasm-spec: PASS wizard-engine: FAIL

carrying code [101, 148, 206], including using abstract interpretation to prove various functional and safety properties [75]. However, as existing eBPF ecosystems continue to grow in complexity and novel runtimes are added [140, 148], concerns regarding correctness remain. Despite extensive testing and verification efforts, kernel bugs introduced by the verifier and JIT, as well as exploits leveraging unsafe extensions that pass the verifier but violate other safety properties, are constantly reported [100].

Kernel fuzzing techniques like LKL-fuzzer [142], BRF [94], or BVF [191], generate eBPF programs that passthe verifier to find correctness bugs, using structured program generation to enforce the eBPF ISA grammar. DIFFSPEC instead relies on LLMs to infer the grammar and generate valid bytecode from the specification document and example tests, overcoming common limitations of generation-based fuzzing techniques, namely that they do not evolve with the program semantics, and they have restricted generation ability. Closer to our work is Kgent [235], which uses LLM agents to generate valid eBPF programs grounded in formal specifications generated from natural language documentation. While we also seek to leverage informal natural language specifications, we focus on generating executable tests with the purpose of identifying differential behavior.

Testing Wasm: There has been work to further the correctness guaranteed by the Wasm spec through mechanization via a custom DSL [230]. Naturally, bugs still exist due to gaps in testing, diversity in use cases, variation in implementation, etc., and have been studied [176, 218]. Tools to find such bugs use static and dynamic analysis [41, 82, 115, 117, 175, 199], compiler fuzzing [11, 79, 233] and binary fuzzing [81, 116, 234]. Some fuzzers have been developed to target the behavior in a specific domain such as smart contracts [48].

Efforts to use differential testing for Wasm have taken several forms (such as using a stackdirected binary generator [160]), none of which (as far as we are aware) use LLMs or leverage natural language. DITWO [131] leveraged differential testing to uncover missed Wasm optimization opportunities. WADIFF [236], the first differential testing framework for Wasm, generated test cases for each operator and then fuzzed them. Wasmaker [43] performs similarly, but can generate more complex binaries.

Testing with LLMs: Neural test generation techniques have been developed to address limitations and challenges of traditional testing techniques, for example, improving coverage of the input space and readability of the generated tests. Several works rely on LLMs to improve unit test generation techniques, by leveraging build [20] and code coverage information [118, 163], using multi-step prompting with AST-based context retrieval [180], and training models on aligned code and tests to improve generated test validity [168].

Fuzzing techniques, like TitanFuzz [60] which uses LLMs to generate and mutate human-like code to test deep learning library APIs, have also been used to improve the coverage and quality of fuzzing inputs. Fuzz4All [226] aims to address the limitations of traditional compiler fuzzers by leveraging LLMs as an input generation and mutation engine. Fuzz4All relies on a set of user provided documentation, example code, or formal specifications for each component under test. It then uses autoprompting techniques to summarize these artifacts and iteratively mutate generated inputs. Our approach leverages similar inputs, however DIFFSPEC does not require users to manually extract relevant sections of documentation and other artifacts for the component under test. Instead, DIFFSPEC automatically extracts relevant specifications for each instruction from the given document. This is to ensure higher level specifications, which can be at scattered across a document, are not missed, improving the validity of the generated inputs. While TitanFuzz and Fuzz4All generate input programs for a single system, DIFFSPEC generates test inputs that differentiate two given programs, does not require a user defined oracle, and uses prompt chaining to incorporate evolving differential information, such as difference in code implementations and historic bugs, to guide test generation. The target-section baseline we use closely resembles the Fuzz4All approach. Closest to our work is Mokav [66], an LLM-guided differential testing technique. Mokav uses execution based feedback to prompt models to generate difference exposing tests between two versions of a python program. Similar to our work, Mokav generates natural language descriptions of each versions of the program, however, unlike our approach it does so independently, without prompting the model to explicitly look for differences. While Mokav targets differential testing, it does not consider other sources of natural language artifacts to guide test generation. Other approaches, such as AID [130] and a Differential Prompting framework introduced by Li et al. [122], leverage buggy versions of code to generate fault-localizing tests that expose differences between a buggy and fixed version. In contrast, DIFFSPEC directly generates differentiating tests without access to a known buggy version of the code.

5.7 Summary

DIFFSPEC helps improve the overall reliability of software systems by generating tests using context from various software artifacts like natural language specification documents, the entire source code, existing tests, and previous bug reports. DIFFSPEC, a novel approach to differential testing with LLMs using prompt chaining that is driven by software artifacts. DIFFSPEC harnesses large language models to learn from vast amounts of natural language specifications, source code, and historical bug data, enabling it to generate targeted tests that reveal meaningful differences between the systems under test. We evaluate our approach on multiple implementations of two extensively tested and widely adopted frameworks: Wasm and eBPF runtimes. Using DIFFSPEC, we generated 1901 differentiating tests, uncovering at least four distinct and confirmed bugs in eBPF, including a kernel memory leak, inconsistent behavior in jump instructions, undefined behavior when using the stack pointer, and tests with infinite loops that hang the verifier in ebpf-for-windows. We also found 299 differentiating tests in Wasm validators pointing to two confirmed and fixed bugs. Our findings again reinforce the importance of incorporating domain knowledge to generate more meaningful tests that can improve the reliability of even well tested software systems. In this case the additional context provided by various artifacts that exist with large software systems, such as specifications, bug reports, etc, can be used to generate more targeted tests that can check verify correctness and check for conformance, and find bugs in even extensively tested systems.

Data Availability: All the code for DIFFSPEC along with the prompts to generate the tests, the scripts to run the tests on the evaluation harness, the generated differential tests, results from additional experiments and manual analysis are available at: DOI 10.5281/zenodo.13836070

5.8 Takeaway

With DIFFSPEC, we show that considering software artifacts beyond just the code under test, such as specification documents, bug reports and so on, can help generate meaningful tests that both verifies code correctness and checks for conformance. DIFFSPEC is a framework for generating differential tests with LLMs using prompt chaining. It can generate targeted tests that align more coherently with the specification and can therefore check the conformance of the code. We demonstrate that these tests can highlight meaningful behavioral differences between implementations, that point to bugs in two extensively tested systems, namely, eBPF runtimes and Wasm validators, therefore improving the overall reliability of these systems.

Chapter 6

Summary of Contributions

My dissertation makes a number of contributions to improve the reliability and usability of code generated by LLMs using domain insights from software engineering, including:

- LowCoder, a low-code tool which supports both visual programming interface and natural language interface to help build AI pipelines by abstracting away textual code.
- We analyze the trade-offs between the two modalities (visual programming interface and natural language interface) and provide the first insight into the effects of using language models for low-code programming through a user study involving 20 participants with varying levels of AI expertise.
- CAT-LM, a specialized model trained to generate tests from code context to verify code correctness.
- The largest corpus of code-test pairs with 1.1M code-test file pairs along with 14.4M Java and Python files across 196K open-source projects to aid future testing related research.
- A testing framework for evaluating tests generated by CAT-LM and other language models that uses both lexical and runtime metrics.
- DiffSpec, the first differential test generation framework that uses natural language specifications and code artifacts to generate tests.
- With DiffSpec, we extend the existing test suites for both eBPF and Wasm, and contribute to open source.
- The tests generated by DiffSpec identified several bugs in the implementations of both eBPF and Wasm runtimes, which have been reported to the maintainers of the projects. The Wasm bugs have been fixed since we reported them.

Chapter 7

Discussion

The advancements in LLMs are rapidly transforming the way developers build software, with significant portions of code now being written by AI in big tech companies like Google.[1] However, as demonstrated throughout this dissertation, solely scaling up model size and training data is insufficient and presents limitations in usability and reliability that can be addressed by incorporating domain insights from software engineering.

Scale vs. Domain Insights. While the industry trend has largely focused on building increasingly larger models trained on large volumes of data, this dissertation has demonstrated that incorporating software engineering domain insights can yield significant improvements in model performance and reliability despite being smaller in size. Each chapter in this dissertation — LowCoder, CAT-LM, and DiffSpec — highlights the benefits of incorporating domain insights from software engineering to improve the model performance by targeting fundamental gaps in existing models. By addressing these challenges that exist with using LLMs for code generation, we developed tools that are more reliable and useful for developers. The success of these specialized approaches suggests that the future of AI-assisted programming may not solely lie in developing ever-larger general-purpose models, but rather in creating an ecosystem of smaller specialized models that perform specific programming related tasks by leveraging domain knowledge. This is especially important given that most organizations cannot afford the immense resources required to train cutting-edge general-purpose models like GPT-4, and many commercial models remain closed-source with limited (free) access to the community which can be unreliable.

Shifting developer focus from writing code to verifying code correctness. As LLMs increasingly automate code generation tasks, the role of software developers is evolving from primarily writing code from scratch to reviewing the correctness of AI generated code. This transition highlights the importance of tools like CAT-LM and DiffSpec, that can be used for test generation to support verification of both developer written and AI generated code. By generating tests that align more coherently with the underlying code implementation, and additionally leveraging other software artifacts such as natural language specifications, or bug reports, we can gen-

[1]https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/

erate meaningful tests that can be used to check code correctness as well as conformance. With DIFFSPEC, we further demonstrate how execution outcomes can be leveraged to improve the reliability of existing software systems. By executing the tests across different implementations that should all conform to the same specification document, DIFFSPEC is able to highlight behavioral differences that points to bugs. This shift towards a verification-centric software development suggests that future programming education and tools should place greater emphasis the skills required to assess the correctness of code, rather than solely focusing on writing code.

Democratizing access to code generation capabilities. LOWCODER addresses the accessibility gap in AI-assisted programming by providing an interface that abstracts away syntactic complexity that exists with traditional code. This enables non-expert programmers to benefit from LLMs in a visual programming space without extensive coding knowledge, therefore democratizing access to code generation capabilities. Our user studies showed that LOWCODER is especially useful for citizen developers who have an idea of *what* they would like to do but do not fully know *how* to accomplish that, perhaps due to a lack of formal programming training. As AI tools become increasingly integrated into software development workflows, making them accessible to programmers across all skill levels is essential to avoid creating a technical divide that might otherwise advantage only those with specialized AI expertise.

Assisting in requirements engineering. The performance of LLMs on various tasks critically depends on the prompt used to query the model, making it extremely important to ensure that the requirements are not under-specified or conflicting [126]. LOWCODER highlights a persistent challenge: the ability to clearly specify what needs to be done. While LLMs excel at generating code, they cannot replace the domain knowledge that is required to define goals and evaluation criteria for a given task. As evidenced in our user study, participants — particularly novices — struggled not with implementation details but with conceptualizing what they wanted to achieve in the first place, encountering what Ko et al. [110] refer to as "design barriers". Despite the existence of powerful LLMs, it is critical for developers to learn the ability to specify requirements by articulating clear objectives to effectively direct these models. Therefore, future work should focus on guiding users to discover and articulate the "what" through contextual suggestions, and solution templates to help build fundamental domain knowledge alongside technical skills required to implement these solutions.

High-quality datasets for advancing future research. A significant contribution of this dissertation is the creation of high quality specialized datasets that enable further research in LLM-based code generation. As the field increasingly adopts post-training techniques like finetuning, reinforcement learning from human feedback, direct preference optimization and integration of external tools, high-quality domain-specific datasets become essential resources. The dataset developed for training CAT-LM is a significant resource for testing-related research, containing the largest corpus of code-test pairs to date. It comprises of 1.1M code and test file pairs, supplemented by 14.4 million Java and Python files gathered from 196,000 open-source projects.

This dissertation has demonstrated that by targeting fundamental gaps in existing LLM-based code generation models through software engineering domain insights, we can create more reliable and useful tools for developers. As AI continues to transform software development, this work suggests that the most effective approaches will be those that combine the generative power of large models with specialized knowledge and capabilities tailored to the unique challenges of software engineering. By continuing to explore exploit insights from software engineering, and incorporating them into the training and evaluation of LLMs, we can build AI programming tools that truly enhance developer productivity while maintaining code quality and reliability.

Chapter 8

Future Work

Building on the research done in previous chapters, we identify two promising directions for future work that extends our contributions to AI-assisted programming.

First, we envision an LLM-powered conversational agent specifically designed to enhance code review processes. With DIFFSPEC (Chapter 5), we showed that extracting relevant context from various software artifacts can be extremely useful for generating targeted tests can highlight bugs in real world systems. LOWCODER (Chapter 3) demonstrated the effectiveness of using LLMs to improve code search in a low-code setting. Both these insights showcase significant improvements in performance for the task at hand by incorporating the right context and tools.

The software engineering (SE) research community has developed numerous tools to search and extract actionable insights from software artifacts, ranging from static analysis tools to testing frameworks and continuous integration pipelines (hereafter just "search tools"). Despite their potential, many of these search tools remain underutilized during code review, a critical process for ensuring software quality. Key challenges include the overwhelming volume of information generated by automated tools, high false-positive rates, and the need for manual configuration or interpretation, which disrupts the flow of review. These barriers, coupled with tight review timelines, often result in code review practices focusing on manual examination and collaborative discussion, rather than leveraging comprehensive tool-based analyses. The emergence of LLMs, is both accelerating the pace of code production and increasing the importance of robust review processes to assess AI-generated and human-written changes. In Section 8.1, we propose a vision for an LLM-powered conversational agent designed to assist code reviewers by bridging the gap between human reviewers and search tools. This agent would summarize relevant insights, tailor them to the specific code change under review, and facilitate context-aware interactions. By enhancing the human-in-the-loop nature of code review, such a tool has the potential to amplify reviewer effectiveness, streamline the review process, and ultimately improve software quality.

Second, we aim to address code-test coevolution. With CAT-LM (discussed in Chapter 4) we demonstrated that LLMs can be effective at generating syntactically valid tests with high coverage that are comparable to human-written tests for a given code file. We extend this with DIFFSPEC (in Chapter 5) and show that we can use other software artifacts such as specifications and bug reports, to generate tests that highlight differential behavior and point to bugs even in extensively tested systems. In both cases, we only look at the given state of the code file or the software system, however, software development is dynamic. Changes to code often necessitate modifications to tests to maintain relevance and effectiveness, and conversely, updates to tests may require adjustments to code for consistency and functionality. Our goal is to develop models that understand this bidirectional relationship, ensuring code and tests evolve harmoniously to support reliable and maintainable software. We elaborate on the research questions and methodological details of this approach in Section 8.2.

8.1 Bridging Code Search and Code Review with LLMs

From previous chapters, we derive several key insights. LowCoder (Chapter 3) established the efficacy of leveraging LLMs to enhance code search capabilities in low-code environments. CAT-LM (Chapter 4) demonstrated the effectiveness of using code context to generate tests that are more aligned with the code. With DiffSpec (Chapter 5), we demonstrated that extracting relevant context from various software artifacts can be extremely valuable for generating targeted tests that effectively highlight bugs in real-world systems. Collectively, these findings underscore a significant improvement in task performance achieved through the strategic utilization of appropriate context and tools.

Over the past several decades, the software engineering (SE) research community has made significant strides in developing tools and techniques to extract useful information from software artifacts (e.g., [84]). These tools span a wide range of categories, from static analysis tools that detect potential bugs, vulnerabilities, or code smells, to testing frameworks and continuous integration (CI) infrastructure that monitor the behavior of software during development [22, 108, 181]. Together, these tools can be thought of as a broad class of "search" tools, designed to mine and interpret various forms of information hidden within software systems, in source code, the history of changes to the repository, documentation, etc.

Unfortunately, many of these tools go underutilized [103], especially during code review [44], one commonly-used checkpoint for ensuring that changes made to software systems do not degrade its quality [24, 138]. Two main obstacles hinder the effective use of search tools during code review. First, in contexts with a high degree of automation, many such tools are already invoked as part of CI pipelines [108, 181, 222]. The sheer volume of information generated by them, only a small fraction of which may be relevant to the particular code change under review, can be overwhelming for reviewers [225]. Moreover, many of these tools suffer from high false positive rates, causing alert fatigue [187, 223]. Second, if one needs to invoke tools on demand, the diversity of available tools places a heavy burden on the reviewer, who both has to know that they should invoke the search tool, along with the knowledge of how to use the search tool to use it to gather information pertinent to their current review task. Many search tools are not designed with the specific needs of code reviewers in mind, requiring manual configuration or interpretation that can disrupt the flow of the review. These two challenges combine to impede the velocity of code reviews, which are often constrained by tight timelines [29]. This has led modern code review practice to primarily focus on careful manual examination of compact diff-style changes and enabling collaborative discussion, instead of incorporating deeper consideration of more comprehensive tool-based analyses.

The emergence of AI-supported programming tools, such as Microsoft's Copilot, is poised to further increase the importance of code review in the software development process. This is due to both the increased cadence with which code can be produced using AI [8], and the importance of having a quality gate to assess the correctness of both AI-generated and human-written changes.

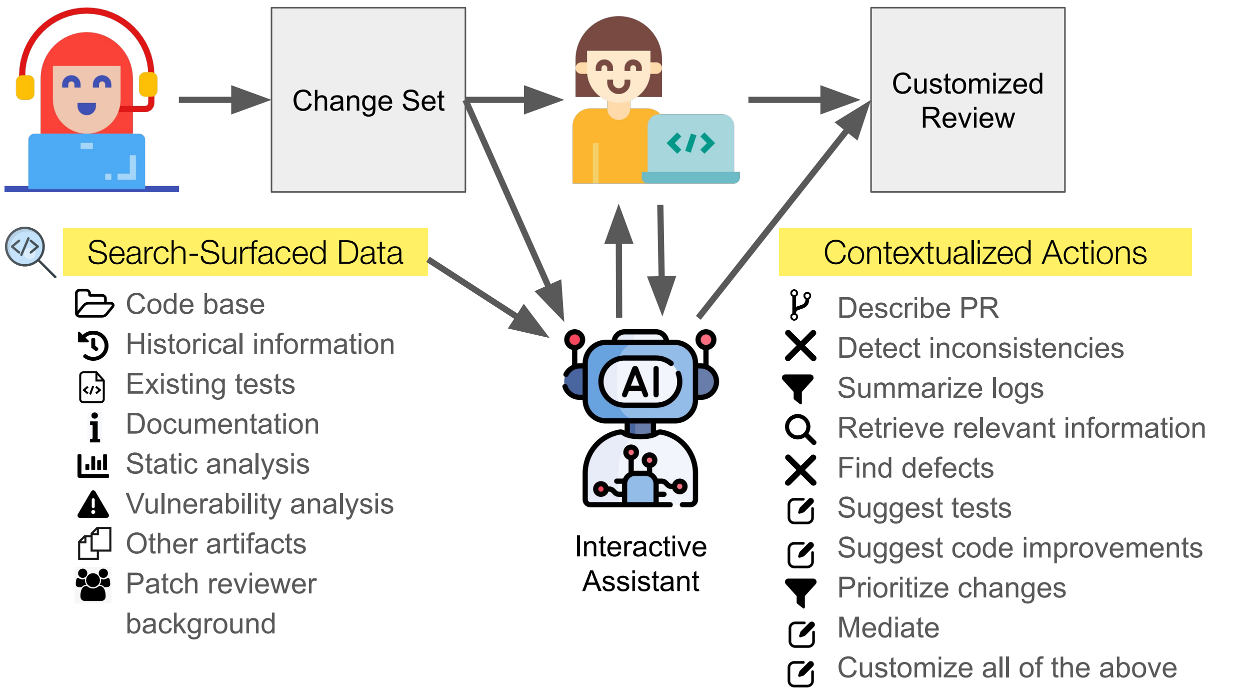


Figure 8.1: An LLM-based interactive assistant could sift through the large volume of information produced by code search tools, and create a customized code review experience.

However, while these AI-based approaches, often using large language models (LLMs), are inducing new pressure on the code review process, they can also improve how engineers perform code reviews. Our vision is that instead of expecting reviewers to manually sift through logs, warnings, and tool output, an LLM-powered conversational agent acts as a bridge between code reviewers and the plethora of available search tools, summarizing relevant insights, presenting them in a way that is tailored to the specific context of the code change under review, and allowing back-and-forth discussion (Figure 8.1). Central to this vision is the recognition that code review is fundamentally a human-in-the-loop process – our goal is not to replace human reviewers but to amplify their effectiveness through improved tooling. This LLM-powered assistance should enable reviewers to more completely, effectively, and quickly assess a given change. Concurrently, these advances will also enable developers to improve their changes before they are submitted for review, enabling the code review process to be more efficient and ultimately improving software quality.

Commercial LLM-based code review systems are starting to emerge.[1] We next outline the main opportunities and challenges to building even more capable systems bridging code search and code review with LLMs. Our work fits in the broader context of conversational assistants for

[1]https://github.blog/changelog/2025-04-04-copilot-code-review-now-generally-available/

software engineering [147, 177].

8.1.1 Opportunity for AI-Enhanced Code Review

The current patch-centric approach to code review constrains reviewers to a narrow, change-focused view that often lacks broader context and project-wide implications of a change. Patches further focus reviewer effort on the exact change itself, without the additional tool-managed metadata and analyses commonly present during modern software development (e.g., code coverage information, static analysis feedback, security analyses). This format is not particularly amenable to rich human interaction or analyses that extend beyond the immediate contents of the patch itself. Reviewers are frequently left to manually piece together the wider impact of changes, cross-reference related parts of the code base, and consider other broad cross-cutting questions such as security and privacy. Each of these are time-consuming tasks that increase cognitive load and reduce reviewer efficiency.

However, we argue that code review offers one of the most interesting touch points where AI-based tools and humans could collaborate on cognitively demanding, highly technical software engineering tasks. Such collaboration can be both natural and potentially highly effective. *Natural* in that code review is already an interactive process, in which participants (historically, humans) seek consensus on the quality of a patch and the fate of a merge request through natural language dialogue. An LLM-based conversational agent would fit naturally in this process, providing a natural analysis interface through which developers can ask questions about the code change, and interact with analysis tools using their own domain understanding. One major downside of many analysis tools is that developers must know that a tool exists, how to get the data from it, and when to apply the tool. Being able to use natural language can free developers to focus on their intention and allow intelligent agents to manage the complexity of marshaling these tools for them [183].

And potentially *highly effective* in that LLMs have shown remarkable abilities at summarizing and synthesizing structured text, in addition to customizing responses [232]. By summarizing and synthesizing data from various analysis tools, the AI could intelligently augment the patch with minimal sets of important information to help the reviewer better understand the impact and context behind a change. In addition, the customization capability could enable the AI to personalize the information to the human actors involved in a way that makes it more useful. Taken together, AI augmentation could reduce the limitations of traditional patch-based code review by providing a more holistic, context-rich environment that extends far beyond the constraints of traditional patch-based workflows.

8.1.2 Realizing the Vision

There are an abundance of information sources that can help reviewers during the code review process. Furthermore, there are a number of underlying actions that an LLM-based conversational agent can perform on these information sources to provide meaningful insights and answer specific questions that a reviewer might have about the code change. Crucially, this is a reviewer-driven search process that is only enabled by the LLM: all of these data are not rele-

Table 8.1: A number of ways in which an LLM-based conversational agent could enhance the code review experience with (information from) search tools. *Information sources*: 🗁 Code Base, ↺ Historical Information, 🗎 Existing Tests, ℹ Documentation, 📊 Static Analysis, ⚠ Vulnerability Analysis, 🗐 Artifacts, 👥 Users. *Actions*: ⑂ Describe PR, ✖ Detect Inconsistencies, ▼ Summarize Logs, 🔍 Retrieve Information, ✎ Provide Suggestions.

Goals / Tasks	**Sources of Information**	**Actions**	**Proposed Ideas**
Finding defects	↺ 🗎 ℹ 📊 🗐	⑂ ✖ ▼ 🔍	The agent retrieves and learns from previous bug fixes to detect and suggest repairs for any defects in the code, plus summarizes CI logs and build logs to identify breaking changes. It also learns from historical changes, to identify files that are often changed together, and suggests changes to dependencies if they are not updated.
Software testing	🗁 ↺ 🗎 📊	⑂ ✖ ▼ ✎	The agent detects inconsistencies between the code change and the corresponding tests, generates new tests when needed, and suggests changes to the existing ones. It also summarizes findings from executing the tests, and answers questions about them. The reviewer can ask the agent to generate a test that exercises specific lines of code by highlighting the code. Based on reviewer background, the agent can also answer questions about the testing framework and provide suggestions on the style/format of the test suite.
Code improvements - functional	🗁 ↺ ℹ 📊 ⚠	▼ 🔍 ✎	The agent calls static analysis tools to detect null pointer exceptions or changes to data flow/control flow graphs, and vulnerability detection tools to detect security issues. It summarizes the findings from these tools. It retrieves information about alternative APIs and frameworks that can be used by querying web search tools, links to similar changes in the past, and suggests alternative implementations for the code change. Based on the reviewer's background (rather the lack thereof), the agent takes a more active role in identifying and alerting the reviewer of various issues that may be present in the code.
Code improvements - non-functional	🗁 ℹ 📊 🗐	⑂ 🔍 ✎	The agent reads the code and documentation to learn the general style of the project, and applies these rules to the code change to ensure consistency. It also suggests comments and documentation updates given a patch. Moreover, the agent invokes program analysis tools to detect dead code, and removes it.
Updating other software artifacts	🗁 ↺ 🗐	⑂ ✖ 🔍 ✎	The agent detects inconsistencies between the code and other artifacts using historical repository information. Similarly, it retrieves relevant company policies to ensure none of them are being violated by the patch. The agent then alerts the reviewer if any inconsistencies are detected and suggests changes to address them.
Knowledge transfer	🗁 ↺ ℹ 🗐 👥	⑂ ▼ 🔍	The agent retrieves and summarizes relevant API documentations for APIs present in the patch, personalized to the reviewer's background and expertise. It also has information about previous related change sets and other relevant parts of the code base, such that the reviewer has more context when reviewing the code change. It can help junior engineers to better understand the code base when reviewing the code change.
Prioritization	🗁 ↺ 👥	✎	The agent identifies groups of similar changes (e.g., refactorings) and related changes (e.g., function definition and call sites), and presents the groups to the reviewer in a personalized way, ordered by familiarity with the change.
Mediation	↺ 🗐 👥	✎	The agent monitors the communication between the reviewer and the author, and suggests edits to language that can be perceived as toxic, pushback, etc.

vant for every patch (nor for every reviewer); providing a lightweight and intuitive mechanism for surfacing this information is the core idea underlying this work.

In the following, we expand on the sources of information and types of actions. In addition, we list a number of concrete ideas on how the LLM assistant could enhance the code review experience with information from search tools in Table 8.1. The list is not intended to be exhaustive, although we cover the main goals of the code review process reported in the literature [24], from *finding defects* to *knowledge transfer*; we also include two tasks part of the code review process, where we expect AI augmentation to be fruitful — deciding how to present the change set to the reviewer (*prioritization*) [166] and managing possible interpersonal conflicts between the submitter and reviewers (*mediation*) [144]. Rather, we seek to illustrate the potential for substantial advances in this area, and inspire future research. Please see the supplementary material DOI 10.5281/zenodo.15265736 for concrete examples of prompts and responses that demonstrate personalized support during the code review process based on reviewer background.

Sources of Information

Code Base. Often a code change alone may not provide enough context for an effective review, and having access to other files in the code base can be helpful, as the patch exists within this larger context. This extra context must be added judiciously, though, to prevent the context from overwhelming the change itself.

Historical Information. Code bases are constantly evolving, and all of these changes are recorded. These recordings, through version control histories, issue trackers, continuous integration logs, etc. provide access to historical information which can be used to suggest changes or provide hints based on common observed patterns. For example, these patterns could include previous code changes made by the same author, previous comments from the same reviewer (providing hints on the reviewer's commonly-held concerns), previous code changes similar to the one being reviewed (providing hints on concerns other reviewers have had for the changed code), and common code files that are often edited together (providing hints on whether a change is incomplete).

Existing Tests. The existing test suite can help ensure that changes made to the code do not break existing functionality. Test execution traces can also help guide the patch reviewer and patch writer towards new test cases that need to be added, or existing test cases that need to be updated. Exposing the dynamic outcome of these tests, specifically showing the tests relevant to a change and whether they continue to pass after the change, can further help the reviewer understand the risk associated with a code change.

Documentation. Documentation related to APIs being used in the code change can be useful if the reviewer is unfamiliar with them. Inspecting API documentation can also help validate whether they are being used appropriately.

Static Analysis. Warnings from static analysis tools can provide useful information and help identify issues with the code, such as null pointer exceptions, unexpected changes to data and control flow, and changes in project-relevant code quality metrics.

Vulnerability Analysis. Access to vulnerability datasets can be a useful for evaluating a change for commonalities with known vulnerabilities. Changes can also be evaluated against known vulnerability solutions to further reinforce reviewer feedback. Both aspects can provide insight for the reviewer into the security risks associated with a change.

Other Artifacts. Software often has other supporting artifacts beyond code, including natural language specification documents, design documents, diagrams for various use cases, requirements documents, and policy documents related to privacy, the company goals, and other concerns. Source code often is intended to conform to these requirements, but the informal / unstructured nature of many of these kinds of documentation makes ensuring both adherence and consistency challenging. Having a system evaluate areas of support and contravention between a change and these documentation can help ensure the overall coherence of the change.

Patch Reviewer & Patch Writer. The patch reviewer's and patch writer's experience and preferences can be learned from the past interactions. This information can be useful for fetching relevant information that suits the needs of both stakeholders before and during code review. For example, a reviewer looking at code changes on a file that they have never interacted with requires more support than a reviewer who has authored or contributed to the code file. Correspondingly, recommendations can also be made to the patch writer in advance of the reviewer actually looking at the change, giving them a chance to preemptively improve their submission.

Actions

Describe PR. A pull request (PR) often contains many different snippets of information, including code changes, the corresponding changes to the tests and documentation, commit messages, etc. Having the agent consume all this information to briefly summarize the changes in the PR provides a starting point for the reviewer.

✖ *Detect Inconsistencies.* Code is rarely treated as an independent entity and often has dependencies to other code files, test files, documentation, specifications, etc. These dependencies can be learned from historical information. The agent should then be able to consume this information to determine if there are inconsistencies between a given code change and a commit message, a code change and the code comments, other documentation, and other artifacts.

Summarize Logs. Many tools can be used to perform various checks for code, such as static analysis or program analysis tools, logs from CI, security checks, performance measures, test metrics, and so on. The agent should consume these and extract the most relevant parts. For example, identifying new tests that do not add any new coverage, or alerting the reviewer of a security threat.

Retrieve Relevant Information. The supporting artifacts for a code change can be lengthy and difficult to consume in their raw format. For example, if the reviewer is not familiar with the library being used, going through the entire documentation can be tedious. However, having the agent summarize the most relevant parts of the documentation based on the reviewer's background can save time and effort. Additionally, having the agent summarizing other artifacts such as the design or requirements documents, company policies or relevant code and tests can help provide relevant context without overwhelming the reviewer.

Provide Suggestions. Having a checklist of things to look for during code review is beneficial; one might even automatically infer this checklist based on historical information (for example, style guidelines, or updating test suite), and other guidelines specific to the project. The agent can then provide suggestions based on the derived checklist to both the patch writer (to ensure that the PR is complete) and the reviewer (to ensure they do not miss anything).

8.1.3 The Road Ahead

While Section 8.1.1 may make it seem like we are proposing an overwhelmingly disparate set of information sources to be brought to bear on this problem, there are important commonalities across all of these information sources that enables progress to be made in a stepwise fashion.

Several primary challenges face this work, although each can be tackled independently.

Natural Interaction. The disparate set of search tools we propose to augment code review with each have their own unique interaction mechanisms. One research avenue for this work is to investigate whether a consistent and natural interaction layer can be applied on top of these search tools to reduce developer friction for invoking and interacting with these tools. Fortunately, the output from most search tools is itself text, upon which LLMs have demonstrated strengths. This is augmented by the context of the change, the patch, also being fully text-based, further easing input to the LLM.

Summarization and Distillation. Each of the search tools will return results in their own formats that must be filtered and tailored to both the specific patch under review, as well as the individual needs of the patch reviewer. Once again, the importance of this task further leans on the strength of LLMs to perform these kinds of tasks on structured text. The key challenge in this space is not in the summarization itself, but in finding useful facts that can help augment code review without overwhelming the reviewer with facts that do not improve the quality or speed of their review.

Trust & Explainability. A fundamental challenge facing this approach is one of trust. This can be thought most simply in terms of precision and recall. In terms of precision, the wealth of information search tools can surface about a change can easily overwhelm the developer. This suggests that considerable effort will be made to elide data that are not useful (e.g., the results should have high precision with few false positives). But this puts the approach in tension with recall. If augmenting code reviews with external information proves useful, developers will want to be able to trust that the tool will not hide information that could have improved their code review (e.g., the results should have high recall with few false negatives). Managing the trust of the patch-writer and patch-reviewer is likely to be more challenging than the technical aspects of interacting with and summarizing the results from the search tools

8.1.4 Summary

Code review plays a longstanding multi-faceted role in modern software development. By its very nature, code review is a time-intensive human process that takes place in a complicated technical domain. In this work we propose deploying LLMs to augment the code review process without specific developer-provided training and tuning in a way that can surface this additional information, enabling reviewer time to be more effectively spent. Naturally, many challenges remain for this vision: can existing public models effectively fulfill this role, or do teams need to train their own? Can locally-hosted models be directly augmented with the necessary context, or will software development stacks gain yet another expensive service they need to pay for? How will the continually-evolving software development ecosystem hinder the kinds of feedback LLMs will propose within the code review? All of these important questions remain to be answered, but the promise itself is clear: enabling better code review decisions by extending discussions far beyond the patch itself.

8.2 Code-Test Coevolution

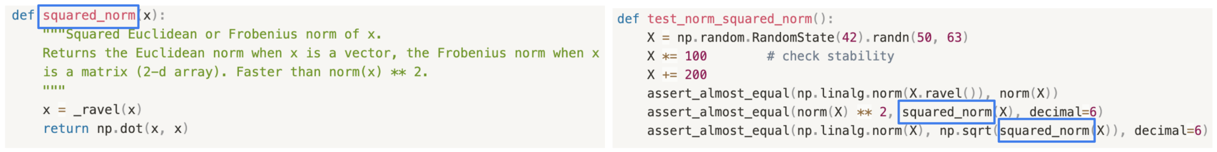
In Chapter 4, we looked at the challenge of verification of generated code and how we overcome it with CAT-LM. Our results showed that the model effectively leverages the code file context to generate more syntactically valid tests that achieve higher coverage. The model provides a strong prior for generating plausible tests: combined with basic filters for compilability and coverage, CAT-LM frequently generated tests with coverage close to those written by human developers. As a natural extension, we aim to model co-adaptations. Changes made to the code may necessitate modifications to the tests to ensure they remain relevant and effective, and conversely, updates to the tests may prompt adjustments to the code to maintain consistency and functionality. The goal is to ensure that code and tests evolve harmoniously, supporting the development of reliable and maintainable software. This is the problem of code-test coevolution. We aim to tackle the former problem, specifically, cases where code functionality is changed in a way that should lead to updates to the tests as well, but the latter are often forgotten [231]. We aim to answer the following research questions related to the problem of code-test coevolution:

RQ1 Can we successfully model co-adaptations and predict whether the given code and test methods are in the same state (consistent) or not?

RQ2 Can we automatically generate the changes to be made to the test method, given the changes to the code method?

The code-test method pairs are identified by checking if the test method is testing the behavior of the given code method. We verify this by examining if the test method includes a call to the corresponding code method. Figure 8.2 shows an example of a test method named `test_norm_squared_norm` that tests the behaviour of `squared_norm`.

As software evolves, it is ideal for any modifications to the behavior of the code method to prompt corresponding changes in the test method. This ensures 'alignment' between both the code and test method, keeping them consistent and in sync with each other. An example of an aligned code-test method pair can be found in Figure 8.3. Here, the code method was updated with a type check and a warning. The corresponding test method is then updated with an assert to test the new warning.

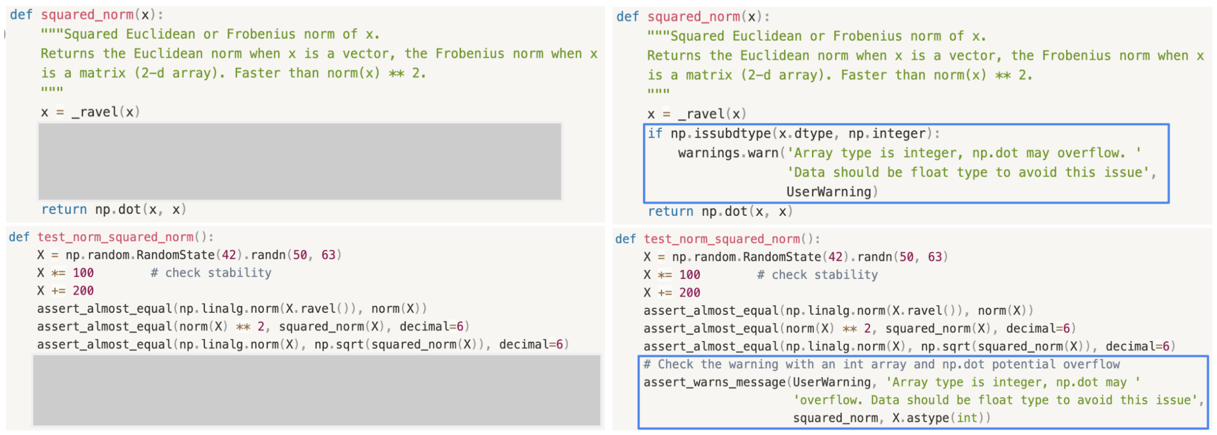


```
def squared_norm(x):
    """Squared Euclidean or Frobenius norm of x.
    Returns the Euclidean norm when x is a vector, the Frobenius norm when x
    is a matrix (2-d array). Faster than norm(x) ** 2.
    """
    x = _ravel(x)
    return np.dot(x, x)
```

```
def test_norm_squared_norm():
    X = np.random.RandomState(42).randn(50, 63)
    X *= 100        # check stability
    X += 200
    assert_almost_equal(np.linalg.norm(X.ravel()), norm(X))
    assert_almost_equal(norm(X) ** 2, squared_norm(X), decimal=6)
    assert_almost_equal(np.linalg.norm(X), np.sqrt(squared_norm(X)), decimal=6)
```

Figure 8.2: An example of a code-test method pair. Code on the left and the corresponding test on the right.

To better understand the problem of code-test co-evolution, we first aim to build a dataset of aligned code-test method pairs having both the *before* and *after* state by mining changes made to code and test methods from GitHub. We then use this dataset to build a model to verify if a given pair of code-test methods are aligned and if not, we generate the changes to be made to the test method, given the change made to a code method.



```python
def squared_norm(x):
    """Squared Euclidean or Frobenius norm of x.
    Returns the Euclidean norm when x is a vector, the Frobenius norm when x
    is a matrix (2-d array). Faster than norm(x) ** 2.
    """
    x = _ravel(x)

    return np.dot(x, x)
```

```python
def test_norm_squared_norm():
    X = np.random.RandomState(42).randn(50, 63)
    X *= 100        # check stability
    X += 200
    assert_almost_equal(np.linalg.norm(X.ravel()), norm(X))
    assert_almost_equal(norm(X) ** 2, squared_norm(X), decimal=6)
    assert_almost_equal(np.linalg.norm(X), np.sqrt(squared_norm(X)), decimal=6)
```

```python
def squared_norm(x):
    """Squared Euclidean or Frobenius norm of x.
    Returns the Euclidean norm when x is a vector, the Frobenius norm when x
    is a matrix (2-d array). Faster than norm(x) ** 2.
    """
    x = _ravel(x)
    if np.issubdtype(x.dtype, np.integer):
        warnings.warn('Array type is integer, np.dot may overflow. '
                      'Data should be float type to avoid this issue',
                      UserWarning)
    return np.dot(x, x)
```

```python
def test_norm_squared_norm():
    X = np.random.RandomState(42).randn(50, 63)
    X *= 100        # check stability
    X += 200
    assert_almost_equal(np.linalg.norm(X.ravel()), norm(X))
    assert_almost_equal(norm(X) ** 2, squared_norm(X), decimal=6)
    assert_almost_equal(np.linalg.norm(X), np.sqrt(squared_norm(X)), decimal=6)
    # Check the warning with an int array and np.dot potential overflow
    assert_warns_message(UserWarning, 'Array type is integer, np.dot may '
                         'overflow. Data should be float type to avoid this issue',
                         squared_norm, X.astype(int))
```

Figure 8.3: An example of code-test aligned pairs, before (left) and after (right) a change was made.

Data Collection

We use GitHub Archive [2] to get the top 1000 python projects and clone these while keeping the revision history. This results in 1.6M commits with modifications made to 6.8M files across all projects. Since we are interested in studying coevolution of code and test methods, we filter out the commits that only make changes to code or test files, resulting in 230K commits with a total of 2.8M files where changes were made to both code and test files. Finally, we use fuzzy string matching to map the test files to the corresponding code files, resulting in a total of 241,098 code-test file pairs.

Next, for each file pair, we extract all the aligned code and test method pairs using static analysis. This is done by building ASTs using the python-graphs[3] framework to find which method calls were made, and dynamic dispatch to identify the class a method call belongs to. This results in a total 53,095 code-test method pairs. Not only does the static analysis capture regular method calls (20,035 samples), it also handles other cases such as: implicit calls to init during object creation, class method invocation and one to many test-code mappings. We can use the same method to further scale up this dataset to several thousand projects across GitHub to expand this dataset if required (based on modeling results).

Modeling Coevolution

Using the 53K code-test method pairs that we collected, we plan to finetune CAT-LM to model co-evolutions, thereby teaching the model how the code and corresponding test changes from the *before* to *after* state as the code/software evolves. Given that CAT-LM has a strong prior for generating plausible tests, our hope is that it's ability can be extended to the problem of co-evolution and effectively model changes made to the code method and adapt them to generate the necessary changes to the test method. We introduce a new signal here to finetune the model, where we use several examples of the *before* code and test method followed by the *after* state.

[2]https://www.gharchive.org/
[3]https://github.com/google-research/python-graphs

Here is an example of the training signal:
```
<before>
def squared_norm():
    ...
<codetestpair>
def test_norm_squared_norm():
    ...
<after>
def squared_norm():
    ...
<codetestpair>
def test_norm_squared_norm():
    ...
```
At inference time, we can provide the *before* context along with the *after* code method to generate the updated test method. We plan to use LoRA tuning or some other form of parameter efficient training to finetune CAT-LM to optimize training for the available compute resources. A potential challenge here are the cases when changes made to the code does not lead to any changes to the test, and therefore teaching the model to predict that the test doesn't change. We aim to address this by splitting the inference as a two step process, first by predicting if the the given code and test methods are in the same state (consistent) or not? And only generating the updated test if they are not consistent. Additionally, when training the model to generate the updated tests, we could include examples of cases where changes made to code does not lead to any changes in the test, for example when the code is refactored, or if there was a bug fix. This would require mining files from commits where changes are only made to code files and not the test. Lastly, we plan to benchmark various models on this task and release both the dataset as well as the trained model to facilitate further research in this space.

Chapter 9

Conclusion

Software development has undergone a significant transformation with the emergence of Large Language Models (LLMs). These powerful AI systems have revolutionized the coding process, becoming essential components of modern programming tools such as ChatGPT and GitHub Copilot. By enabling capabilities like code generation from natural language instructions, automated bug detection and resolution, and documentation generation, LLMs have substantially enhanced developer productivity and efficiency.

Though these models are pretrained on large volumes of natural language and code data, their fundamental training approach — using cross-entropy and preference losses that optimize for matching ground truth without explicit coefficients for correctness — creates inherent constraints on their effectiveness. While this enables LLMs to achieve remarkable proficiency in learning code syntax, they fall short in capturing crucial semantic signals. For a long time, the main focus of efforts to improve these models has been to train larger models and collect more human preference data. However, user studies have uncovered significant usability challenges with these larger models, including difficulty in understanding the generated code, the presence of subtle bugs that are hard to find, and a lack of verification of the generated code. These findings suggest that simply scaling model size and expanding training data may not address the fundamental limitations with training LLMs for code generation.

My research proposed solutions to these challenges by developing techniques that integrates domain insights from software engineering into AI-based code generation with the goal of enhancing reliability and utility for developers. This is done by empowering the model to take on a more active role in building valid and usable code, instilling greater trust among users in the capabilities of the model. I focused on three main challenges identified by prior work and proposed solutions using software-specific insights. (1) The generated code can be difficult to understand and manipulate, especially for non-expert programmers. To address this, I proposed LOWCODER, a tool that abstracts away the syntactic complexity associated with traditional code and provides a more user-friendly interface using drag-and-drop functionality. As a result, LOWCODER provides a trusted environment where users can leverage the capabilities of AI without the need for extensive coding knowledge. (2) Verifying the correctness of the generated code is hard. While LLMs excel at generating code, they are lacking when it comes to generating tests. This is largely because current models are trained on individual files and therefore can not consider the code under test context. To overcome this, I proposed CAT-LM, a LLM trained to explicitly consider the mapping between code and test files. CAT-LM can therefore help users with verifying code that they or other models generate, by generating tests that align more coherently with the underlying code. (3) The generated code often has subtle bugs that are hard to find. To address this, I proposed DIFFSPEC, a framework for generating differential tests with LLMs using prompt chaining to verify code correctness. DIFFSPEC makes use of various software artifacts like natural language specification documents, source code, existing tests, and previous bug reports to generate tests to not only verify code correctness, but also checks for conformance against the specification. By highlighting meaningful behavioral differences between implementations, DIFFSPEC enhances the overall reliability of even extensively tested software systems.

Overall, my dissertation demonstrated the significance of integrating software-specific insights when training models to make code generation more reliable and useful for developers. This was accomplished by empowering models to take on a more active role in building valid and usable code, instilling greater trust among users in the capabilities of these models. Additionally, my work contributes several artifacts which include datasets for various tasks, models that are trained using software-specific insights, and evaluation frameworks. Note that these models are all quite small relative to cutting-edge general-purpose models like GPT-4. While large, general models can also be very useful for these tasks, they have their own limitations: few companies can afford the immense resources required to train such large models, and most of these models are closed-source and provide limited (free) access to the community, which can be unreliable. In contrast, my work produces open-source models, which are often smaller, that are specialized to perform various programming-related tasks, resulting in tools that make code generation more reliable and useful for developers.

Bibliography

[1] GitHub REST API. URL https://docs.github.com/en/rest. 4.3.1

[2] Introduction to LLMs. URL https://developers.google.com/machine-learning/resources/intro-llms. 2.1

[3] GPT-neox Toolkit. URL https://github.com/EleutherAI/gpt-neox. 4.1, 4.4.2

[4] SentencePiece. URL https://github.com/google/sentencepiece. 4.1, 4.4.1

[5] TheFuzz: Fuzzy String Matching in Python. URL https://github.com/seatgeek/thefuzz. 4.3.2

[6] GitHub Copilot, 2021. URL https://github.com/features/copilot. 1, 1.1, 1.1, 3, 4, 4.7

[7] ChatGPT, November 2022. URL https://openai.com/blog/chatgpt/. 1, 1.1, 3

[8] Google Q3 earnings call: CEO's remarks. https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024, 2024. 8.1

[9] Introducing the gpt store — openai, 2024. Retrieved August 29, 2024 from https://openai.com/index/introducing-the-gpt-store/. 1.1

[10] V8, 2024. URL https://chromium.googlesource.com/v8/v8.git. (Accessed 2024-09-12). 5.3.2

[11] wasm-smith, 2024. URL https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith. (Accessed 2024-09-12). 5.6

[12] spec, 2024. URL https://github.com/WebAssembly/spec. (Accessed 2024-09-12). 5.3.2

[13] wasm-tools, 2024. URL https://github.com/bytecodealliance/wasm-tools. (Accessed 2024-09-12). 5.3.2

[14] Wasmtime, 2024. URL https://github.com/bytecodealliance/wasmtime. (Accessed 2024-09-12). 5.3.2

[15] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021. 2.3

[16] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3559555. URL https://doi.org/10.1145/3551349.3559555. 2.2

[17] Toufique Ahmed and Premkumar Devanbu. Better patching using llm prompting, via self-

consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1742–1746, 2023. doi: 10.1109/ASE56229.2023.00065. 2.2

[18] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars!, 2023. 2.3

[19] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, SPLASH '19, pages 143–153, 2019. 4.3.1

[20] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024. 5, 5.6

[21] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995. URL https://doi.org/10.1017/S135132490000005X. 3.1.2, 3.1.3, 3.5

[22] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2016. 8.1

[23] Chetan Arora, John Grundy, and Mohamed Abdelrazek. Advancing requirements engineering through generative ai: Assessing the role of llms, 2023. 2.2

[24] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013. 8.1, 8.1.2

[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Survey*, 51(3): 50–88, 2018. 1.1, 4

[26] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models, 2022. 1, 2.4

[27] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, Avraham Shinnar, and Jason Tsay. Pipeline combinators for gradual AutoML. In *Advances in Neural Information Processing Systems (NeurIPS)*, December 2021. URL https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf. 3.1, 3.5

[28] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey,

Jerry Tworek, and Mark Chen. Efficient Training of Language Models to Fill in the Middle. *CoRR*, abs/2207.14255, 2022. 1.1, 2.3, 4, 4.5.1

[29] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21:932–959, 2016. 8.1

[30] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, page 179–190, 2015. 1.1, 4

[31] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *International Conference on Software Engineering*, ICSE '15, page 559–562, 2015. 1.1, 4

[32] Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: Prospects, challenges, and a case study. In Bernhard Steffen, editor, *Bridging the Gap Between AI and Reality*, pages 355–374, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-46002-9. 2.2

[33] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning (ICML)*, pages I–115–I–123, 2013. 3.5

[34] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022. 1.1, 5

[35] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME - the Konstanz information miner: Version 2.0 and beyond. *ACM SIGKDD Explorations Newsletter*, 11 (1):26–31, November 2009. URL https://doi.org/10.1145/1656274.1656280. 3, 3.5

[36] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata. 2.1

[37] Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Usvsn Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. In Angela Fan, Suzana Ilic, Thomas Wolf, and Matthias Gallé, editors, *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, virtual+Dublin, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.bigscience-1.9. URL https://aclanthology.org/2022.bigscience-1.9. 2.3

[38] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. Technical Report UCB/CSD-04-1368, University of California, Berkeley, 2004. URL

https://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-04-1368.pdf. 3.5

[39] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, 2002. 4.7

[40] Carolin Brandt and Andy Zaidman. Developer-centric test amplification: The interplay between automatic generation human exploration. *Empirical Software Engineering*, 27 (4), 2022. ISSN 1382-3256. 1.1, 4, 4.7

[41] Florian Breitfelder, Tobias Roth, Lars Baumgärtner, and Mira Mezini. Wasma: A static webassembly analysis framework for everyone. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 753–757, 2023. doi: 10.1109/SANER56733.2023.00085. 5.6

[42] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 1877–1901, 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. 2.3, 3.2.4, 4.7

[43] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1262–1273, 2024. ISBN 9798400706127. doi: 10.1145/3650212.3680358. URL https://doi.org/10.1145/3650212.3680358. 5.6

[44] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. The silent helper: the impact of continuous integration on code reviews. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 423–434. IEEE, 2020. 8.1

[45] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping distributed hierarchical web data. In *Symposium on User Interface Software and Technology (UIST)*, pages 963–975, 2018. URL https://doi.org/10.1145/3242587.3242661. 3.5

[46] Chu Chen, Pinghong Ren, Zhenhua Duan, Cong Tian, Xu Lu, and Bin Yu. Sbdt: Search-based differential testing of certificate parsers in ssl/tls implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 967–979, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598110. URL https://doi.org/10.1145/3597926.3598110. 5

[47] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser,

Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374, 2021. 1.1, 2.3, 3.2.4, 3.5, 4, 4.5.1, 4.7

[48] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. Wasai: uncovering vulnerabilities in wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 703–715, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534218. URL https://doi.org/10.1145/3533767.3534218. 5.6

[49] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KuPixIqPiq. 5.2.1

[50] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016. 1.1, 5

[51] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering*, ICSE '19, pages 736–747, 2019. 4.7

[52] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. 2.1

[53] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, ICFP '00, page 268–279, 2000. 4.7

[54] Jonathan Corbet. The bpf system call api, version 14. URL https://lwn.net/Articles/

612878/. (Accessed 2024-09-12). 5.3.1

[55] John W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE publications, 4th edition, 2013. 3.3.2, 3.3.2

[56] Ed. D. Thaler. Bpf instruction set architecture (isa), 2024. URL https://www.ietf.org/archive/id/draft-ietf-bpf-isa-02.html. (Accessed 2024-09-12). 5, 5.3.1

[57] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children Thing1 and Thing2? In *International Symposium on Software Testing and Analysis*, ISSTA '17, pages 57–67, 2017. 4.7

[58] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022. 4.1

[59] Janez Demsar, Blaz Zupan, Gregor Leban, and Tomaz Curk. Orange: From experimental machine learning to interactive data mining. In *European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 537–539, 2004. URL https://doi.org/10.1007/978-3-540-30116-5_58. 3, 3.5

[60] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the international symposium on software testing and analysis*, pages 423–435, 2023. 5, 5.6

[61] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *International Conference on Software Engineering (ICSE)*, pages 345–356, 2016. URL https://doi.org/10.1145/2884781.2884786. 3.5

[62] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. Toga: A neural method for test oracle generation. In *International Conference on Software Engineering*, ICSE '22, page 2130–2141, 2022. 1.1, 2.2, 4, 4.5.1, 4.5.2, 4.7

[63] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. Towards understanding the capability of large language models on code clone detection: A survey, 2023. 2.2

[64] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml. 3.3.2

[65] eBPF.io. ebp.io. URL https://ebpf.io/infrastructure/. (Accessed 2024-09-12). 5.3.1

[66] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. Mokav: Execution-driven differential testing with llms. *arXiv preprint arXiv:2406.10375*, 2024. 5.6

[67] Mattia Fazzini and Alessandro Orso. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 308–318, 2017. doi: 10.1109/ASE.2017.8115644. 1.1, 5

[68] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained

model for programming and natural languages. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1536–1547, November 2020. URL https://aclanthology.org/2020.findings-emnlp.139/. 2.3, 3.2.1

[69] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Conference on Neural Information Processing Systems (NIPS)*, pages 2962–2970, December 2015. URL http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning. 3.5

[70] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396431. doi: 10.1145/3511861.3511863. URL https://doi.org/10.1145/3511861.3511863. 2.4

[71] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Conference on Offensive Technologies*, WOOT '20, pages 10–10, 2020. 4.7

[72] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, 2011. 1.1, 4, 4.6.3, 4.7

[73] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR*, abs/2204.05999, 2022. 1.1, 2.3, 4

[74] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, 2006. 3.3.1

[75] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019. 5.6

[76] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential regression testing for rest apis. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 312–323, 2020. ISBN 9781450380089. doi: 10.1145/3395363.3397374. URL https://doi.org/10.1145/3395363.3397374. 1.1, 5

[77] WebAssembly Community Group. Webassembly specification, 2024. URL https://webassembly.github.io/spec/core/. (Accessed 2024-09-12). 5, 5.3.2

[78] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi:

10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363. 5.3.2

[79] Zhao Hai, Zhichen Wang, Mengchen Yu, and Lei Li. Is webassembly really safe? - wasmvmescape andrcevulnerabilities have been found in new way. Technical report, 07 2022. URL https://i.blackhat.com/USA-22/Wednesday/US-22-Hai-Is-WebAssembly-Really-Safe-wp.pdf. 5.6

[80] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009. URL http://doi.acm.org/10.1145/1656274.1656278. 3, 3.5

[81] Keno Haßler and Dominik Maier. Wafl: Binary-only webassembly fuzzing with fast snapshots. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'21, page 23–30, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396028. doi: 10.1145/3503921.3503924. URL https://doi.org/10.1145/3503921.3503924. 5.6

[82] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 385–397, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598064. URL https://doi.org/10.1145/3597926.3598064. 5.6

[83] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 93–105, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600139. URL https://doi.org/10.1145/3568813.3600139. 2.4

[84] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR cookbook: Mining a decade of research. In *International Conference on Mining Software Repositories (MSR)*, pages 343–352, 2013. 8.1

[85] Mike Hicks. How we built cedar with automated reasoning and differential testing. URL https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing. (Accessed 2024-09-12). 5

[86] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. Cigar: Cost-efficient program repair with llms, 2024. 2.2

[87] Djoerd Hiemstra. *N-Gram Models*, pages 1910–1910. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_935. URL https://doi.org/10.1007/978-0-387-39940-9_935. 2.1

[88] C Hill, R Bellamy, T Erickson, and M Burnett. Trials and tribulations of developers of intelligent systems: A field study. In *Symposium on Visual Languages and Human-Centric*

Computing (VL/HCC), pages 162–170, sep 2016. 3.4

[89] Martin Hirzel. Low-code programming models. *Communications of the ACM (CACM)*, 66(10):76–85, October 2023. URL https://doi.org/10.1145/3587691. 3, 3.1, 3.1.2, 3.1.3

[90] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, pages 45–48, 2018. 5.4.2

[91] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *CoRR*, arXiv:2203.15556, 2022. 1, 2.1, 2.3, 4.1, 4.4.2

[92] Dotan Horovits. Webassembly: The next frontier in cloud-native evolution. URL https://wasmcloud.com/blog/webassembly-the-next-frontier-in-cloud-native-evolution. (Accessed 2024-09-12). 5.3.2

[93] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with Hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020. 4.5.1

[94] Hsin-Wei Hung and Ardalan Amiri Sani. Brf: Fuzzing the ebpf runtime. *Proceedings of the ACM on Software Engineering*, 1(FSE):1152–1171, 2024. 5.6

[95] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. 3.2.4

[96] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 319–321, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392235. doi: 10.1145/3510454.3522684. URL https://doi.org/10.1145/3510454.3522684. 2.4

[97] iovisor. bpf-fuzzer: fuzzing framework based on libfuzzer and clang sanitizer., 2015. URL https://github.com/iovisor/bpf-fuzzer. (Accessed 2024-09-12). 5.6

[98] iovisor. Userspace ebpf vm, 2015. URL https://github.com/iovisor/ubpf. (Accessed 2024-09-12). 5.3.1

[99] Nathan Jay and Barton P Miller. Structured random differential testing of instruction decoders. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 84–94. IEEE, 2018. 1.1, 5

[100] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 150–157, 2023. 5.6

[101] Guang Jin, Jason Li, and Greg Briskin. Enhanced ebpf verification and ebpf-based runtime safety protection. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 224–230. IEEE, 2024. 5.6

[102] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and

Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms, 2023. 2.2

[103] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013. 8.1

[104] Rajeswari Hita Kambhamettu, John Billos, Tomi Oluwaseun-Apo, Benjamin Gafford, Rohan Padhye, and Vincent J. Hellendoorn. On the naturalness of fuzzer-generated code. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 506–510, 2022. ISBN 9781450393034. doi: 10.1145/3524842.3527972. URL https://doi.org/10.1145/3524842.3527972. 5.4.3

[105] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020. 2.3

[106] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. 1

[107] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–600. IEEE, 2017. 1.1, 5

[108] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *International Conference on Software Engineering (ICSE)*, pages 476–487. IEEE, 2019. 8.1

[109] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli Calling '23, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400716539. doi: 10.1145/3631802.3631806. URL https://doi.org/10.1145/3631802.3631806. 2.4

[110] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Symposium on Visual Languages – Human Centric Computing (VL/HCC)*, December 2004. URL https://doi.org/10.1109/VLHCC.2004.47. 3, 3.1.2, 3.1.3, 3.4, 7

[111] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *International Conference on Quality Software*, ICQS '13, pages 103–112, 2013. 4.3.2

[112] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Annual Meeting of the Association for Computational Linguistics*, ACL '18, pages 66–75, 2018. 4.4.1

[113] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014. 1.1, 5

[114] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A Neural model for generating

natural language summaries of program subroutines. In *International Conference on Software Engineering*, ICSE '19, pages 795–806, 2019. 4.5.1

[115] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304068. URL https://doi.org/10.1145/3297858.3304068. 5.6

[116] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly, 2021. URL https://arxiv.org/abs/2110.15433. 5.6

[117] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. That's a tough call: Studying the challenges of call graph construction for webassembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 892–903, 2023. ISBN 9798400702211. doi: 10.1145/3597926.3598104. URL https://doi.org/10.1145/3597926.3598104. 5.6

[118] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 919–931. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00085. URL https://doi.org/10.1109/ICSE48619.2023.00085. 5.6

[119] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023. 1.1, 2.3, 4, 4.5.1

[120] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 238–251, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582053. URL https://doi.org/10.1145/3582016.3582053. 5

[121] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and

Brad A. Myers. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Symposium on User Interface Software and Technology (UIST)*, pages 577–589, 2019. URL https://doi.org/10.1145/3332165.3347899. 3.5

[122] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 14–26. IEEE, 2023. 5, 5.6

[123] Youlin Li, Weina Niu, Yukun Zhu, Jiacheng Gong, Beibei Li, and Xiaosong Zhang. Fuzzing logical bugs in ebpf verifier with bound-violation indicator. In *ICC 2023-IEEE International Conference on Communications*, pages 753–758. IEEE, 2023. 5.6

[124] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158. 2.3

[125] J. T. Liang, C. Yang, and B. A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 605–617, Los Alamitos, CA, USA, apr 2024. IEEE Computer Society. URL https://doi.ieeecomputersociety.org/. 1, 1.1, 2.4

[126] Jenny T. Liang, Melissa Lin, Nikitha Rao, and Brad A. Myers. Prompts are programs too! understanding how developers build software containing prompts, 2024. URL https://arxiv.org/abs/2409.12447. 1.1, 7

[127] libbpf. libbpf. URL https://github.com/libbpf/libbpf. (Accessed 2024-09-12). 5.3.1

[128] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Conference on Text Summarization Branches Out*, pages 74–81, 2004. 4.5.1

[129] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf. 2.4

[130] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. Llm-powered test case generation for detecting tricky bugs. *arXiv preprint arXiv:2404.10304*, 2024. 5, 5.6

[131] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 436–448, New

York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598068. URL https://doi.org/10.1145/3597926.3598068. 5.6

[132] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on GitHub. In *Proceedings of the ACM on Programming Languages*, volume 1 of *OOPSLA '17*, pages 1–28, 2017. 4.3.1

[133] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658, 2023. doi: 10.1109/ISSRE59848.2023.00026. 2.2

[134] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664, 2021. 3.2.4, 4.5.2

[135] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Repoagent: An llm-powered open-source framework for repository-level code documentation generation, 2024. 2.2

[136] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023. 2.3

[137] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A New Approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019. 4.7

[138] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016. 8.1

[139] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10 (1):100–107, 1998. 1.1, 5

[140] Microsoft. ebpf for windows, 2021. URL https://github.com/microsoft/ebpf-for-windows. (Accessed 2024-09-12). 5.3.1, 5.6

[141] Microsoft. Bpf conformance, 2022. URL https://github.com/Alan-Jowett/bpf_conformance. (Accessed 2024-09-12). 1.1, 5.3.1

[142] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of linux ebpf subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 87–92, 2023. 5.6

[143] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2024. 2.3

[144] Emerson Murphy-Hill, Ciera Jaspan, Carolyn Egelman, and Lan Cheng. The pushback effects of race, ethnicity, gender, and age in code review. *Communications of the ACM*, 65

(3):52–57, 2022. 8.1.2

[145] Otoya Nakakaze, István Koren, Florian Brillowski, and Ralf Klamma. Retrofitting industrial machines with WebAssembly on the edge. In Richard Chbeir, Helen Huang, Fabrizio Silvestri, Yannis Manolopoulos, and Yanchun Zhang, editors, *Web Information Systems Engineering – WISE 2022*, pages 241–256, Cham, 2022. Springer International Publishing. ISBN 978-3-031-20891-1. 5.3.2

[146] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding, 2024. 2.2

[147] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *International Conference on Software Engineering (ICSE)*, 2024. 8.1

[148] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020. 5.3.1, 5.6

[149] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2022. doi: 10.1145/3524842.3528470. 2.4

[150] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *International Conference on Software Engineering*, ICSE '23, page 2111–2123, 2023. 1.1, 2.2, 4, 4.5.1, 4.5.1, 4.5.2, 4.6.2, 4.7

[151] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A Conversational Paradigm for Program Synthesis. *CoRR*, abs/2203.13474, 2022. 1.1, 2.3, 3, 3.2.4, 4, 4.4.1, 4.4.2, 4.5.1, 4.6.3, 4.7

[152] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023. 2.3

[153] Randal S. Olson and Jason H. Moore. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning (AutoML)*, pages 66–74, June 2016. URL https://proceedings.mlr.press/v64/olson_tpot_2016.html. 3.5

[154] OpenAI. Gpt-4 technical report, 2023. 2.1, 2.3, 4.6.3

[155] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. 2.1, 2.3

[156] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, 2007. 4.7

[157] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and

Vincent J Hellendoorn. Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities. In *International Conference on Software Maintenance and Evolution*, pages 523–533, 2020. 1.1, 4

[158] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. Tips for creating a block language with Blockly. In *Blocks and Beyond Workshop (B&B)*, October 2017. URL https://doi.org/10.1109/BLOCKS.2017.8120404. 3, 3.1, 3.3.2, 3.5

[159] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 3, 3.1

[160] Árpád Perényi and Jan Midtgaard. Stack-driven program generation of webassembly. In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 – December 2, 2020, Proceedings*, page 209–230, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-64436-9. doi: 10.1007/978-3-030-64437-6_11. URL https://doi.org/10.1007/978-3-030-64437-6_11. 5.6

[161] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23. ACM, November 2023. doi: 10.1145/3576915.3623157. URL http://dx.doi.org/10.1145/3576915.3623157. 2.4

[162] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *International Conference on World Wide Web (WWW)*, pages 263–273, 2016. URL https://doi.org/10.1145/2872427.2883029. 3.1

[163] Juan Altmayer Pizzorno and Emery D Berger. Coverup: Coverage-guided llm-based test generation. *arXiv preprint arXiv:2403.16218*, 2024. 5.6

[164] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. Together we go further: Llms and ide static analysis for extract method refactoring, 2024. 2.2

[165] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018. URL https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf. 3.2.4, 3.2.4

[166] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In *International Conference on the Foundations of Software Engineering (FSE)*, pages 201–212, 2018. 8.1.2

[167] Liam Randall. How webassembly will transform edge computing. URL https://cosmonic.com/blog/industry/webassembly-at-the-edge. (Accessed 2024-09-12). 5.3.2

[168] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. Cat-lm: Training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 409–420. IEEE, 2023. 1.1, 1, 5, 5.6

[169] Nikitha Rao, Elizabeth Gilbert, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. Diffspec: Differential testing with llms using natural language specifications and code artifacts, 2024. URL https://arxiv.org/abs/2410.04249. 1.1

[170] Nikitha Rao, Jason Tsay, Kiran Kate, Vincent Hellendoorn, and Martin Hirzel. Ai for low-code for ai. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*, IUI '24, page 837–852, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705083. doi: 10.1145/3640543.3645203. URL https://doi.org/10.1145/3640543.3645203. 1.1, 1

[171] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project, 2024. 1, 1.1, 2.4, 3

[172] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR*, abs/2009.10297, 2020. 4.5.1

[173] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communications of the ACM (CACM)*, 52 (11):60–67, November 2009. URL https://doi.org/10.1145/1592761.1592779. 3, 3.1.1, 3.1.2, 3.4, 3.5

[174] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ASE '11, pages 23–32, 2011. 4.7

[175] João Rodrigues and Jorge Barreiros. Aspect-oriented webassembly transformation. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2022. doi: 10.23919/CISTI54924.2022.9820136. 5.6

[176] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. An empirical study of bugs in webassembly compilers. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 42–54. IEEE Press, 2022. ISBN 9781665403375. doi: 10.1109/ASE51524.2021.9678776. URL https://doi.org/10.1109/ASE51524.2021.9678776. 5.6

[177] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *International Conference on Intelligent User Interfaces (IUI)*, page 491–514, 2023. 8.1

[178] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. 2.3

[179] Richard Rutledge and Alessandro Orso. Automating differential testing with overapproximate symbolic execution. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 256–266. IEEE, 2022. 1.1, 5

[180] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024. 5.6

[181] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, volume 1, pages 598–608. IEEE, 2015. 8.1

[182] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020. URL https://doi.org/10.1109/SEAA51224.2020.00036. 3

[183] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf. 8.1.1

[184] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive Test Generation Using a Large Language Model. *CoRR*, abs/2302.06527, 2023. 4.7

[185] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018. doi: 10.1109/ICST.2018.00033. 5.4.3

[186] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, March 2020. ISSN 0167-2789. doi: 10.1016/j.physd.2019.132306. URL http://dx.doi.org/10.1016/j.physd.2019.132306. 2.1

[187] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. Why can't Johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Symposium on Usable Privacy and Security (SOUPS)*, pages 221–238, 2020. 8.1

[188] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2z2: detecting rendering regressions in web browsers through differential fuzz testing. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1818–1829, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510044. URL https://doi.org/10.1145/3510003.3510044. 5

[189] Hugo Henrique Fumero de Souza, Igor Wiese, Igor Steinmacher, and Reginaldo Ré.

A characterization study of testing contributors and their contributions in open source projects. In *Brazilian Symposium on Software Engineering*, SBES '22, pages 95–105, 2022. 4.3.2

[190] Hao Sun and Zhendong Su. Validating the ebpf verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 615–628, 2024. 5.6

[191] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 689–703, 2024. 5.6

[192] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024. 5.2.1

[193] Steven L. Tanimoto. A perspective on the evolution of live programming. In *International Workshop on Live Programming (LIVE)*, pages 31–34, 2013. URL https://doi.org/10.1109/LIVE.2013.6617346. 3.1.1, 3.5

[194] The DFINITY Team. The internet computer for geeks. Technical report, DFINITY, 04 2022. URL https://internetcomputer.org/whitepaper.pdf. 5.3.2

[195] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 847–855, August 2013. URL https://doi.org/10.1145/2487575.2487629. 3.5

[196] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023. 2.4

[197] Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Tests and Proofs*, volume 4966 of *TAP '08*, pages 134–153, April 2008. 4.7

[198] Ben L. Titzer. Wizard, An advanced Webassembly Engine for Research, 2024. URL https://github.com/titzer/wizard-engine. (Accessed 2024-09-12). 5.3.2

[199] Ben L. Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. Flexible non-intrusive dynamic instrumentation for webassembly. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 398–415, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651338. URL https://doi.org/10.1145/3620666.3651338. 5.6

[200] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *CoRR*, abs/2302.13971, 2023. 2.1, 4.4.2, 4.7

[201] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundare-

san. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020. 4.7

[202] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Conference on Human Factors in Computing Systems (CHI)*, 2022. URL https://doi.org/10.1145/3491101.3519665. 1, 1.1, 2.4, 3, 3.5

[203] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 2.1, 3.2.4

[204] Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. Generating chat bots from web api specifications. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 44–57, 2017. URL http://doi.acm.org/10.1145/3133850.3133864. 3.5

[205] Johannes Villmow, Jonas Depoix, and Adrian Ulges. ConTest: A Unit Test Completion Benchmark featuring Context. In *Workshop on Natural Language Processing for Programming*, pages 17–25, August 2021. 1.1, 2.2, 4, 4.7

[206] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023. 5.6

[207] Markus Voelter and Sascha Lisson. Supporting diverse notations in MPS' projectional editor. In *Workshop on The Globalization of Modeling Languages (GEMOC)*, pages 7–16, 2014. URL https://hal.inria.fr/hal-01074602/file/GEMOC2014-complete.pdf#page=13. 3.1, 3.5

[208] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised, coverage-guided kernel fuzzer, 2015. URL https://github.com/google/syzkaller. 5.6

[209] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. Frustrated with code quality issues? llms can help!, 2023. 2.2

[210] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. Potential of webassembly for embedded systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2022. doi: 10.1109/MECO55406.2022.9797106. 5.3.2

[211] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. How does machine learning change software development practices? *Transactions on Software Engineering (TSE)*, 2019. doi: 10.1109/TSE.2019.2937083. 3, 3.3.2, 3.4

[212] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021. 2.1

[213] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. Human-AI collaboration in data science: Exploring data scientists' perceptions of automated AI. *Proc. ACM Hum.-*

Comput. Interact., 3(CSCW), nov 2019. doi: 10.1145/3359313. URL https://doi.org/10.1145/3359313. 3.5

[214] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289, 2023. doi: 10.1109/MedAI59581.2023.00044. 2.2

[215] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8696–8708, 2021. URL https://aclanthology.org/2021.emnlp-main.685/. 2.3, 3, 3.2.4, 3.3.1, 4.5.1, 4.5.2

[216] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023. 2.3

[217] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1332–1342, 2015. URL https://www.aclweb.org/anthology/P15-1129.pdf. 3.5

[218] Muhammad Waseem, Teerath Das, Aakash Ahmad, Peng Liang, and Tommi Mikkonen. Issues and their causes in webassembly applications: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 170–180, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717017. doi: 10.1145/3661167.3661227. URL https://doi.org/10.1145/3661167.3661227. 5.6

[219] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. *CoRR*, abs/2002.05800, 2020. 1.1, 2.2, 4, 4.5.1, 4.7

[220] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 172–184, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616271. URL https://doi.org/10.1145/3611643.3616271. 2.2

[221] Leandro von Werra. Codeparrot. https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot. 4.1, 4.3.2

[222] Mairieli Wessel, Bruno Mendes De Souza, Igor Steinmacher, Igor S Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A Gerosa. The power of bots: Characterizing and understanding bots in OSS projects. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–19, 2018. 8.1

[223] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. Don't disturb me: Challenges of interacting with software bots on open source software projects. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW2):1–21, 2021. 8.1

[224] Robert White and Jens Krinke. Reassert: Deep learning for assert generation. *CoRR*, abs/2011.09784, 2020. 4.7

[225] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of Travis CI. In *International Conference on the Foundations of Software Engineering (FSE)*, pages 647–658, 2019. 8.1

[226] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. 5, 5.4.1, 5.6

[227] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A Systematic Evaluation of Large Language Models of Code. *CoRR*, abs/2202.13169, 2022. 2.3, 4.4.1, 4.4.2

[228] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2), mar 2022. URL https://doi.org/10.1145/3487569. 2.4, 3.5

[229] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993532. URL https://doi.org/10.1145/1993498.1993532. 1.1, 5, 5.4.3

[230] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the webassembly standard up to speed with spectec. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi: 10.1145/3656440. URL https://doi.org/10.1145/3656440. 5.6

[231] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229, 2008. URL https://api.semanticscholar.org/CorpusID:15074269. 8.2

[232] Haopeng Zhang, Philip S. Yu, and Jiawei Zhang. A systematic survey of text summarization: From statistical methods to large language models, 2024. URL https://arxiv.org/abs/2406.11289. 8.1.1

[233] Xiangwei Zhang, Junjie Wang, Xiaoning Du, and Shuang Liu. Wasmcfuzz: Structure-aware fuzzing for wasm compilers. In *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, EnCyCriS/SVM '24, page 1–5, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705656. doi: 10.1145/3643662.3643959. URL https://doi.org/10.1145/3643662.3643959. 5.6

[234] Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1035–1047, New York, NY, USA, 2024. Association for Computing Machinery. ISBN

9798400706127. doi: 10.1145/3650212.3680340. URL https://doi.org/10.1145/3650212.3680340. 5.6

[235] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. Kgent: Kernel extensions large language model agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 30–36, 2024. 5.6

[236] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. Wadiff: A differential testing framework for webassembly runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 939–950, 2023. doi: 10.1109/ASE56229.2023.00188. 5.6

[237] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 21–29, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534864. URL https://doi.org/10.1145/3520312.3534864. 2.4