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

by

Nikitha Rao

Thesis proposal submitted to the Software and Societal Systems Department
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Software Engineering

Carnegie Mellon University

Committee in Charge:

Vincent J. Hellendoorn, Carnegie Mellon University (Co-Chair)
Claire Le Goues, Carnegie Mellon University (Co-Chair)
Andrew Begel, Carnegie Mellon University
Daniel Fried, Carnegie Mellon University
Thomas Zimmermann, Microsoft Research

Spring 2024

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

by

Nikitha Rao

Submitted to the Software and Societal Systems Department
on March 25, 2024 in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Software Engineering

Committee in Charge:

Vincent J. Hellendoorn, Carnegie Mellon University (Co-Chair)
Claire Le Goues, Carnegie Mellon University (Co-Chair)
Andrew Begel, Carnegie Mellon University
Daniel Fried, Carnegie Mellon University
Thomas Zimmermann, Microsoft Research

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. Currently, 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.

My thesis work investigates the integration of 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 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 propose 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 propose 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. I am currently working on enhancing the reliability of LLM outputs, ensuring that the generated code is not only syntactically correct, but also executes successfully. I first built a benchmark of executable data science notebooks to highlight the importance of code execution as a key validation step in evaluating AI generated code. I then leverage the (often failing) execution of code generated by these LLMs as a signal which the model uses to refine the generated code before presenting them to the developer. Therefore, enhancing the over reliability of the generated code.

The goal of my thesis is to demonstrate the significance of integrating software-specific insights when training models to make code generation more reliable and useful for developers. My thesis work will result in 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. On the contrary, 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.

Contents

Chapter 1

Introduction

The recent development of Large Language Models (LLMs) has led to the widespread use of AI-powered programming tools such as ChatGPT [1] and GitHub Copilot [2] 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.

Currently, the primary approach to enhancing model performance has revolved around training larger models with increased parameters or incorporating more training data [3], [4]. 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) [5], [6], the presence of subtle bugs that are hard to find [6], [7] and lack of verification of the generated code [7], [8]. 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 [2] and ChatGPT [1] 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 [6]. 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 [5]. 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 [5].

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 [6]. 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 [6].

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, 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 [9], [10] and is often either partially or entirely neglected. This has led to extensive work on automated test generation, including both classical [11]–[14] and neural-based methods [11], [15], [16].

Classical test generation tools like EvoSuite [12] directly optimize to generate high-coverage tests. However, the generated tests are often hard to read and may be unrealistic or even wrong [17]. This requires time and effort from developers to verify generated test correctness [13]. Meanwhile, LLMs trained on code have made major strides in generating human-like, high-quality functions based on their file-level context [18]–[21]. AI-powered tools like Copilot excel at code generation, and can significantly improve the productivity of its users [2]. 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, 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 [22] and, the CodeGen [19] model family, which includes mono-lingual models trained on a much larger budget than ours. CAT-LM is also compared against TeCo [23], 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 practise 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

Even when utilizing high-quality code-generating tools like Copilot, there is a tendency for the generated code to contain subtle bugs that are challenging to detect. Previous studies have found that nearly 40% of participants indicated that the time required to debug or modify code generated by AI tools constitutes a significant reason for abstaining from using these tools [7]. This highlights a noteworthy obstacle in the adoption of such tools, emphasizing the need for enhancing the reliability and ease of debugging in AI-driven code generation processes.

Automating the generation of data science notebooks is inherently challenging due to the unique blend of natural language, code, graphics, and execution results that are commonly found in notebooks [24]. Additionally, the interactive nature of notebooks introduces multiple turns of interconnected natural language to code challenges [25]. Notably, tasks like

data exploration, visualization, and manipulation in data science workflows require a deep understanding of the dataset, execution states of various cells, as well as the non-trivial long range dependencies between code cells, beyond the mere syntactic correctness of the generated code [26]. The absence of this nuanced context presents a notable hurdle in automating the generation of data science notebooks, and generating reliable notebooks poses an even greater challenge.

Studies have shown that participants who frequently incorporate AI tools into their programming workflows assess the correctness of the generated code by executing it [7], [8]. Currently, LLMs are primarily trained to generate code based on prior context and lack the ability to utilize execution outputs for refining the generated code. However, prior context alone is insufficient to ensure the generation of bug-free code. This underlines the practical importance of code execution as a key validation step in evaluating the reliability of AI-generated code.

Key Insight: Even code that appears to be syntactically correct may contain bugs, and executing the code can help uncover these issues.

Solution: I seek to overcome this challenge in ongoing work by ensuring that the generated code is not only syntactically correct, but also executes successfully. I first built a benchmark of executable data science notebooks to highlight the importance of code execution as a key validation step in evaluating AI generated code. I then propose DEBUG-LM, a model that leverages the (often failing) execution of code generated by LLMs as a signal. This approach involves the collaboration of various models operating in unison: one model that uses the notebook prefix to generate the next code cell, which is then executed. Another model observes the failing execution to repair the generated code before presenting it to the developer. By observing this interaction, these models can be taught to debug the code they, or regular developers, generate.

Takeaway: LLMs often generate code that appears syntactically correct but contain subtle bugs that are usually uncovered during execution. I first built a benchmark of executable data science notebooks to highlight the importance of code execution as a key validation step in evaluating AI generated code. I then leverage the (often failing) execution of code generated by these LLMs as a signal. DEBUG-LM learns to utilize this execution signal in order to refine the generations before presenting them to the developers. Through this work, we can enhance the overall reliability of the code generated by LLMs, thereby promoting a more robust development environment.

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 ensuring that the generated code is not only syntactically correct, but also executes successfully. I do so by leveraging the (often failing) execution of code generated by these LLMs as a signal which the model uses to refine the generated code before presenting them to the developer. Thereby enhancing the reliability of the generated code.

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 the proposal. 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 in Chapter 5, I elaborate on how I improve the reliability of LLMs by training a model to use execution signal to generate code that is syntactically correct and executable. Lastly, in Chapter 6 I enumerate few stretch goals that are extensions of my work.

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

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	Execution signal	Debug-LM

The contributions presented in this proposal 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 and Related Work

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 [27]. Early computational language models, such as n-gram models [28], 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 [29]. 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. [30], 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) [31] 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 [32], [33]. Including code from open source software has now become common practise when training LLMs such as PaLM [34], Chinchilla [3], GPT-4 [35], and Llama [36].

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 [37]. LLMs have proven useful at assisting developers across various stages of the software development life cycle from extracting requirements [38] to repairing bugs [39]–[41]. They can be used for several tasks including: generating code from natural language [42], generating tests to verify code correctness [11], [15], [16], [23], finding and fixing bugs [40], [41], [43], summarizing code [44], generating documentation [45], automatic refactoring of code [46], clone detection [47] and so on. Beyond this, they can be used to automate code review [48], and suggest improvements [49]. LLMs can also be used as educational tools as they are uniquely placed to support developers by providing expert knowledge the form of conversations, answering questions about the code to aid understanding by providing explanations and examples [37], [50].

2.3 Training LLMs for Code

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 [18], and CodeGen [19] 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 [22], CodeLLama [51], SantaCoder [52], CodeGen2 [53], Incoder [20], GPT-NeoX [54], as well as closed source models like GPT4 [35], AlphaCode [55], CodeT5+ [56]. 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 [18], GPT3 [57], PolyCoder [58], CodeGen [19], StarCoder [22] 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 [59] and CuBERT [60], 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 [21], InCoder [20], CodeGen2 [53], StarCoder [22], SantaCoder [52], CodeLlama [51], 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 [61], PLBART [62] 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+ [56] 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 [31]. Models like CodeLlama [51], WizardCoder [63] and InstructCodeT5+ [56], OctoCoder [64] 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 [3]. 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 [6], [7], [65]–[67]. There have also been feasibility studies conducted for using LLMs in development tools [6], [8], [68]–[70] as well as using LLMs in education [5], [71], [72].

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 [6], [73] and code quality [68] , contrasting findings suggest that these tools have a positive association with developers' self-perceived productivity [74].

Ziegler et al. [74] 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. [6] which compared the user experience of traditional autocomplete 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. [8] 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. [7] 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. [5] 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 thesis, 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

Challenge 1: Syntactic Complexity

AI has demonstrated significant potential in empowering individuals with limited or no programming experience to write code. Copilot [2] and ChatGPT [1] 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 [5]. 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 [6].

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 [75] 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 [76]. Most low-code offerings for building AI pipelines currently favor visual programming [77]–[79]. 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 [80].

At the intersection of these two paradigms, we propose LowCoder, 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 the challenge of syntactic complexity in LLM generated code.

Most AI development today involves Python programming with popular libraries such as scikit-learn (sklearn) [81]. Unfortunately, writing code, even in a language as high-level as Python, is hard for *citizen developers* [82]—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" [82], where it is difficult to even conceptualize a solution. In contrast to other popular low-code domains such as traditional software [83], 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 [84]. We chose to target sklearn [81] 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 [80] to provide a Scratch-like [83] 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 [85] 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 [86]. 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

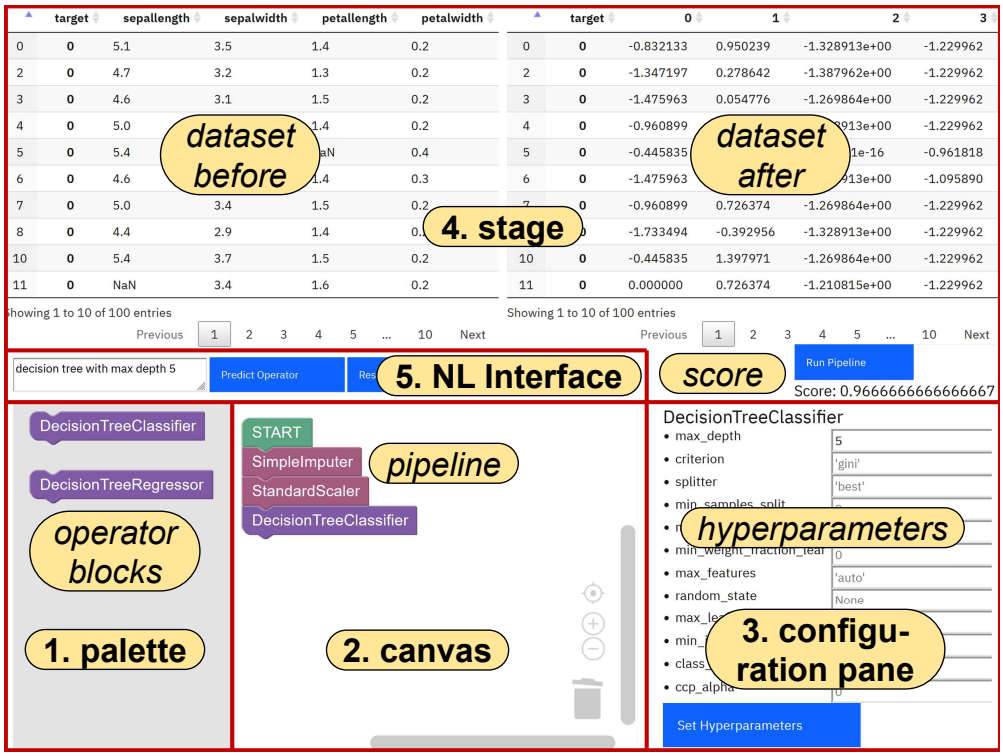


Figure 3.1: LowCoder interface with labeled components, described in the text.

for building ML pipelines with sklearn operators for tabular data that includes both visual programming (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 [87] 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 [75]. 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 [81], 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 [88] satisfies these requirements, and in addition, describes hyperparameters in JSON schema format [89], which our tool also uses. The current version of our tool supports 143 sklearn operators. LowCoder$_{\text{VP}}$ uses a client-server architecture with a Python Flask back-end server and front-end based on the Blockly [80] 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$_{VP}$ 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 [83] 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$_{VP}$ also encourages liveness [90] 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 [83].

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 [82]. For instance, the palette of LowCoder$_{VP}$ contains more than a hundred operator blocks. Rather than requiring users to know the exact name of the operator or scroll through so many operators, we provide LowCoder$_{NL}$, 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 [91]. In comparison, visual inputs and representations are unambiguous [75], requiring no probabilistic interpretation, so users can easily understand and manipulate the results returned by LowCoder$_{NL}$.

To ground our evaluation of LowCoder$_{NL}$, 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 [83].

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 [82]. For instance, the palette of LowCoder$_{VP}$ contains more than a hundred operator blocks. Rather than requiring users to know the exact name of the operator or scroll through so many operators, we provide LowCoder$_{NL}$, 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 [91]. In comparison, visual inputs and representations are unambiguous [75], 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 [59]. 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.[1] 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.

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

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 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 [92]. 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 [93] 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.

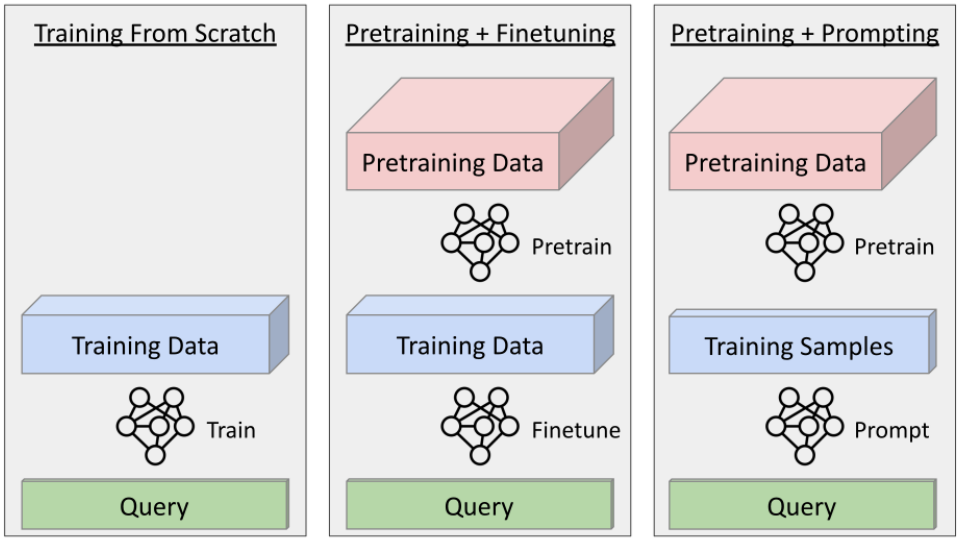


Figure 3.2: Overview of the “trifecta" of training approaches used in contemporary deep learning: smaller models are directly trained from scratch on downstream task data; medium sized models (100M-1B parameters) are pretrained with a generic training signal and then fine-tuned on task data; large models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

Fine-tuning CodeT5

CodeT5 is a pretrained encoder-decoder transformer model [85] that has shown strong results when fine-tuned on various code understanding and generation tasks [94]. 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[94] 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 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 [96]. More specifically, we look at CodeGen, a family of LLMs that are based on standard transformer-based autoregressive language modeling [86]. 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 [97] on a Python benchmark [86].

Models at this scale are expensive to fine-tune and are instead commonly used for in-

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.

ference by means of "few-shot prompting" [92]. LLMs are remarkably capable of providing high-quality completions given an expanded prompt containing examples demonstrating the task [96]. 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.

3.3 Evaluation

This section describes the evaluations for the language modeling that enables LowCoder$_{NL}$ 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 [85] and CodeGen [86] 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 [98] 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

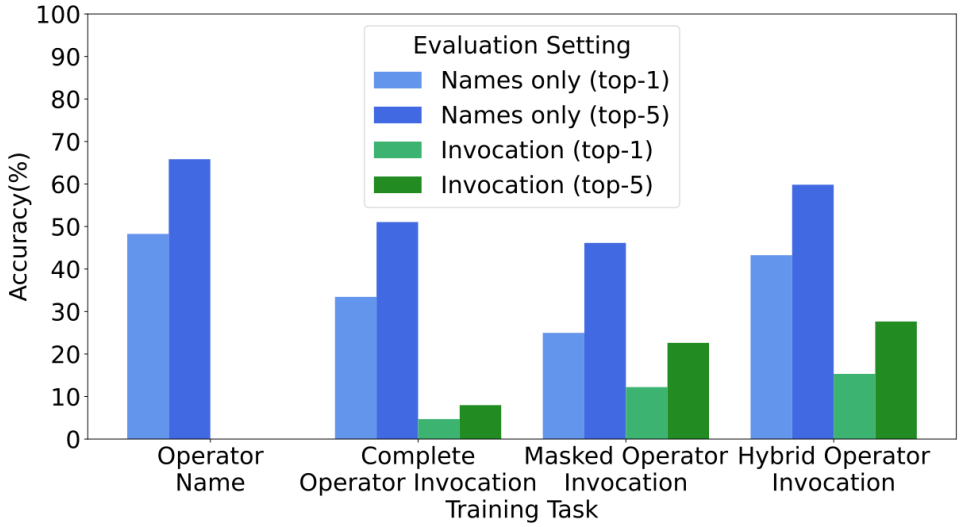


Figure 3.4: Accuracy of Transformer models trained from scratch on various task formulations. 'Invocation' test results refer to the specific invocation formulation of the training task, while 'Names only' just considers whether the generated code starts with the correct operator name. Only the Hybrid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

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 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

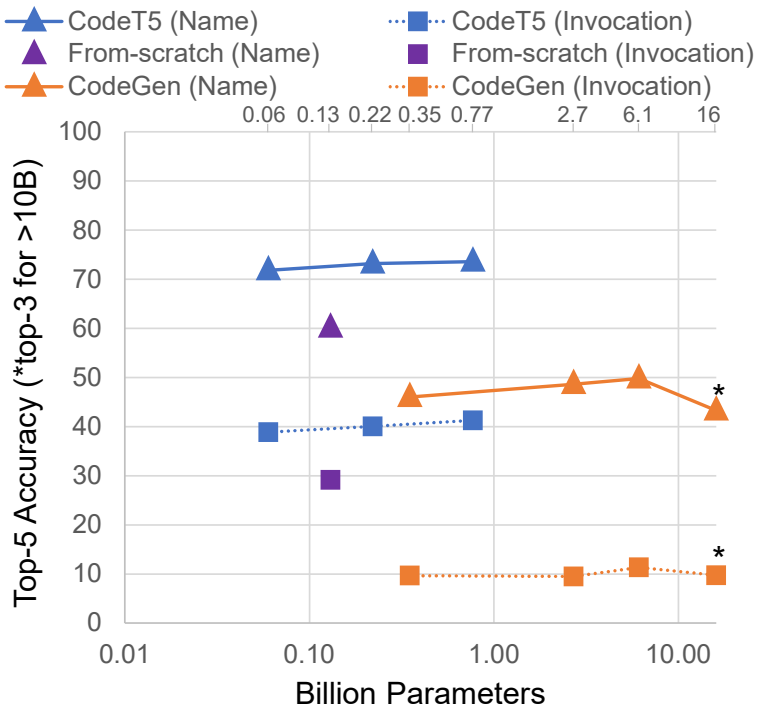


Figure 3.5: Accuracy vs. model size based on top-5 sampling. (*The 16B CodeGen uses top-3 due to memory constraints.) We compare the three modeling paradigms, namely training transformer from scratch, finetuning CodeT5, and fewshot prompting CodeGen, on both Operator Name generation and Hybrid Operator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

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:

RQ1: How do LowCoder$_{\text{NL}}$ 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 infor-

matics, 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 [99] where each participant was exposed to two conditions: using LowCoder with (*NL condition*) and without (*keyword condition*) the natural language (NL) interface powered by $\text{LowCoder}_{\text{NL}}$. 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 [100]. 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 [100]. 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 ex-

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$_{NL}$	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%)

act 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$_{NL}$, generic web search engine (Google), and scrolling through the palette. Participants may discover multiple unknown 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 hyper-parameters 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 [99] 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$_{NL}$ and other features help participants discover previously-unknown operators?

A known limitation of visual programming is discoverability [80]. 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$_{NL}$ 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$_{NL}$, 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$_{NL}$ 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$_{NL}$ 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$_{NL}$ 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$_{NL}$ 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 [84] 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 expe-

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%)

rience, 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 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$_{NL}$ 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 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$_{NL}$, 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$_{NL}$.

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*

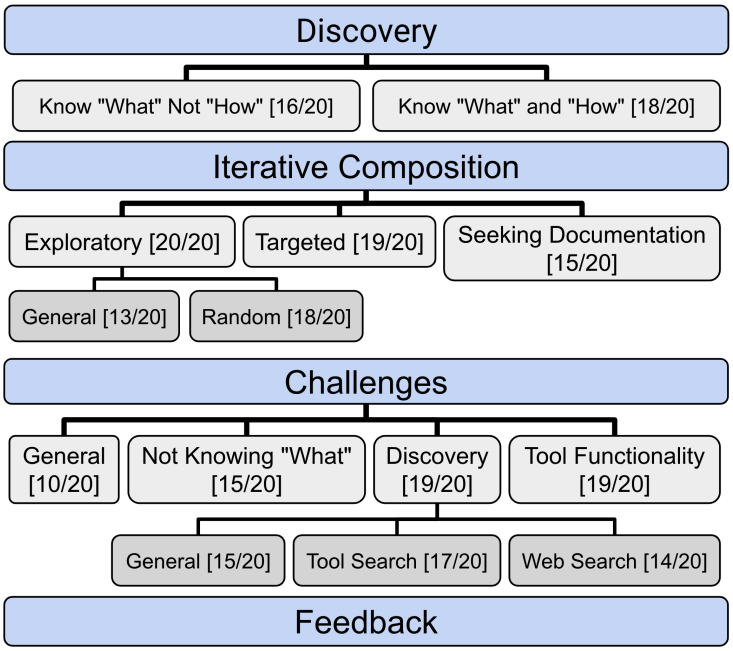


Figure 3.6: Axial codes from our qualitative analysis.

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 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.

3.5 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

Challenge 2: 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 [9], [10] and is often overlooked. To address this, there has been extensive work done in automating test generation, which includes both classical [11]–[14] and neural-based methods [11], [15], [16].

Classical test generation tools like EvoSuite [12] are optimized to generate tests with high-coverage. However, the generated tests are often hard to read and may be unrealistic or even wrong [17]. As a result, developers need to invest time and effort to verify the correctness of generated tests [13]. 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 [18]–[21]. Tools like Copilot excel at code generation, thereby significantly enhancing user productivity [2]. 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 [11], [15], [23]. 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, 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 [22] and, the CodeGen [19] model family, which includes mono-lingual models trained on a much larger budget than ours. We also compare against TeCo [23], 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 4.1 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 [101] standards (including deduplication), resulting in ∼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 [102]. We then determine the appropriate model size, 2.7B parameters based on our training budget and the Chinchilla scaling laws [3]. We use the GPT-NeoX toolkit [103] enhanced with Flash Attention [104] 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 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

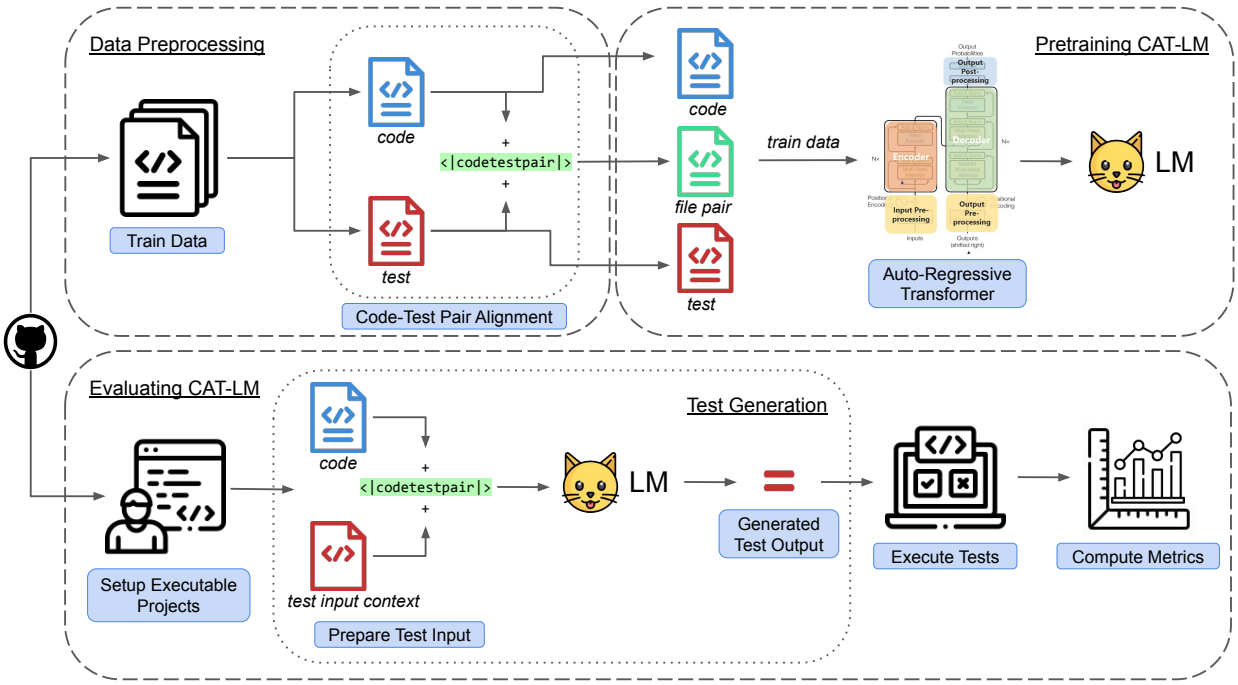


Figure 4.1: Approach overview. We extract Java and Python projects with tests from GitHub and heuristically align code and test files (top), which, along with unaligned files, train CAT-LM, a large, auto-regressive language model. We evaluate CAT-LM's generated tests on a suite of executable projects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

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.

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

Test generation with 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.

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 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).

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

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 [105] to mine Python and Java repositories that have at least 10 stars and have new commits after January 1st, 2020. Following [106] and [107], 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 [101] 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 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

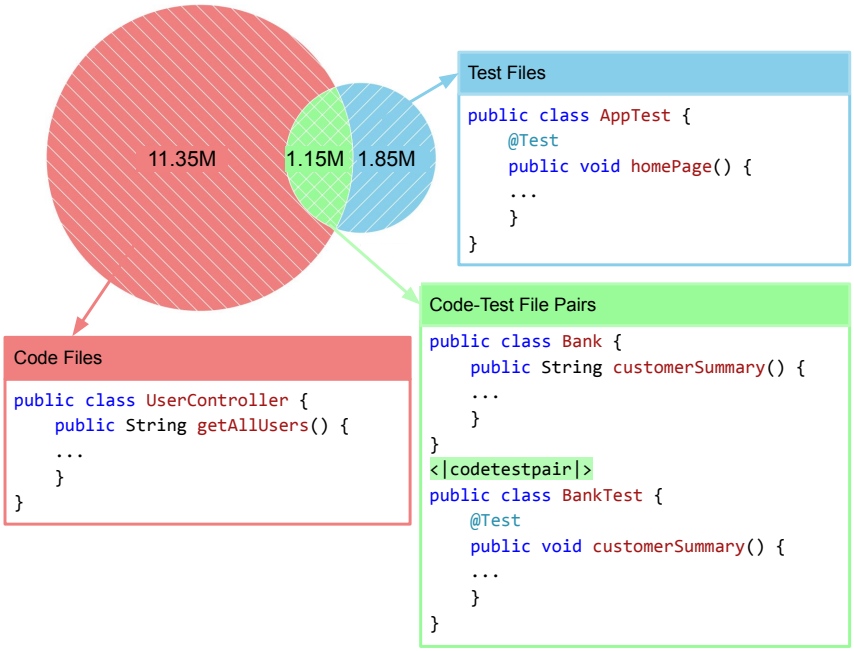


Figure 4.3: Distribution of files with sample code snippets

have the pattern `test_<CFN>`, `<CFN>_test`, `<CFN>Test` or `Test<CFN>`. If no matches are found, we perform a fuzzy string match [108] 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 9.7 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 [109], [110]. Note that we normalize the plot to help illustrate trends by aggregating 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.

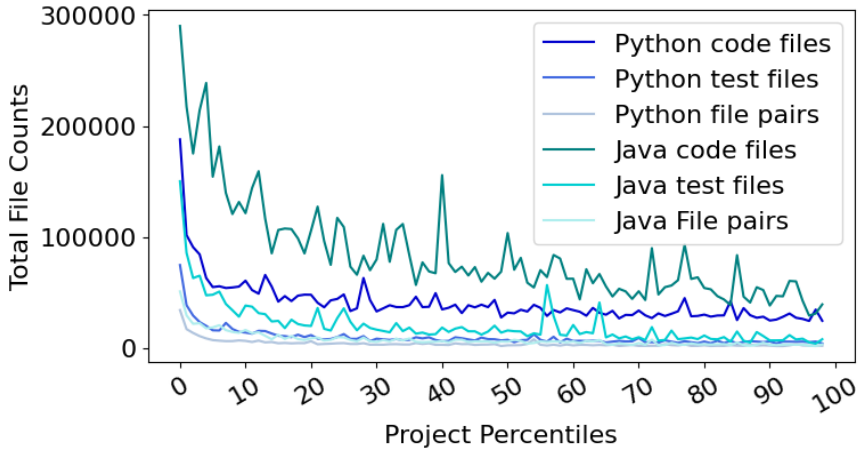


Figure 4.4: Distribution of files in projects sorted by GitHub stars, normalized by percentiles

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).

4.4 CAT-LM

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

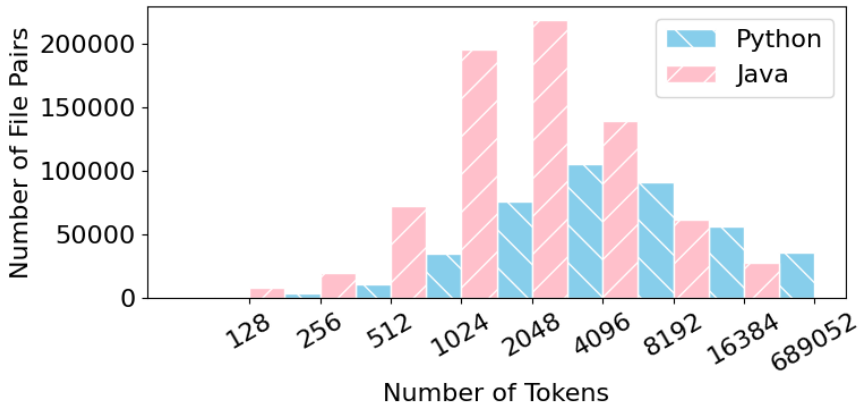


Figure 4.5: Distribution of file pair tokens

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 [111] using the SentencePiece [102] 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 [19], [58].[1] 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.

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 [3], 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)

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

parameters, a common size for medium to large language models [19], [58], 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 ∼2M tokens. We use the GPT-NeoX toolkit [103] 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[2] We note that this training duration is much shorter than many popular models [19], [36];[3] 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.

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.

[2]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).

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

Baseline Models

CodeGen is a family of Transformer-based LLMs trained auto-regressively (left-to-right) [19]. 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 [18] on a Python benchmark [19].

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 [22], 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 [21] 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 [112] (longest overlapping subsequence of tokens) and CodeBLEU [113] 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 [23], [61], [114], [115].

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 encourage more widespread adoption of runtime metrics (which are an important part of test utility), as prior work primarily focuses on lexical similarity [11], [15], [23].

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,

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

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

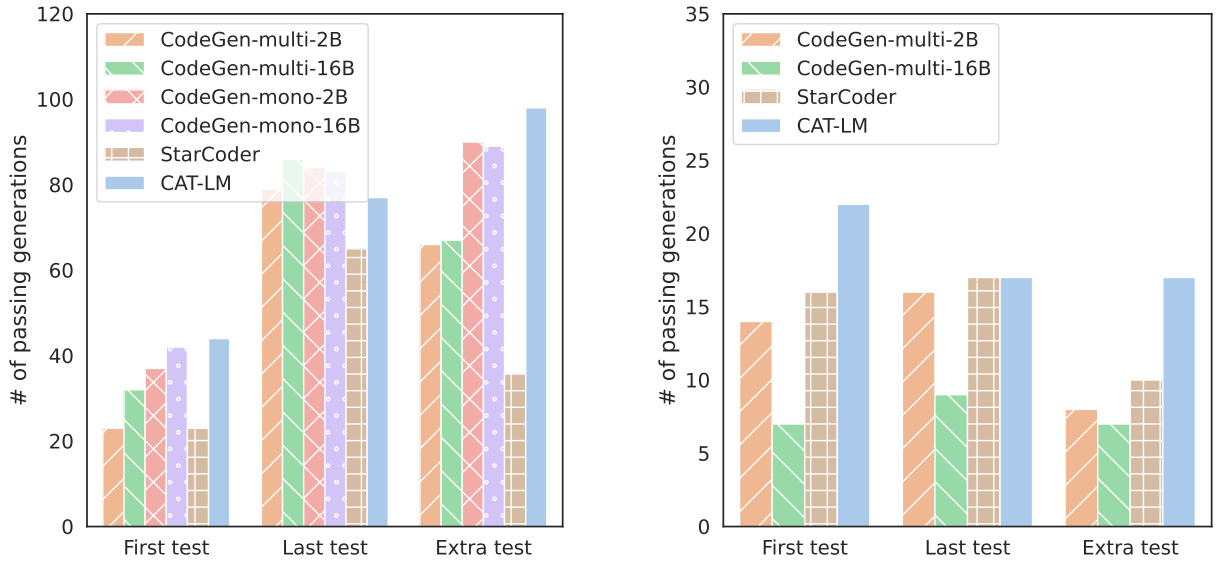
We evaluate the quality of the generated tests using the containers that we setup to execute projects in Section 4.3.3. We insert the generated test into the original test file, execute the respective project's setup commands and check for errors, recording the number of generated tests that compile 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.

Baseline Model

We compare against TeCo [23], a state of the art baseline on test statement completion that has outperformed many existing models, including CodeT5 [61], CodeGPT [116] and TOGA [11]. TeCo [23] is a encoder-decoder transformer model based on the CodeT5 archi-



(a) Python.

(b) Java.

Figure 4.6: Passing tests by model for Python and Java.

tecture [61]. 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

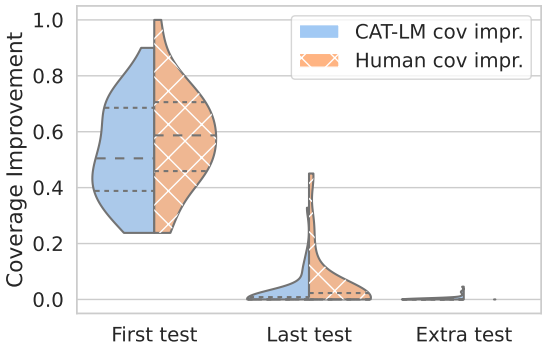
We evaluate CAT-LM's ability to generate valid tests that achieve coverage, comparing against state of the art baselines for both code generation and test completion.

4.6.1 Test Method Generation

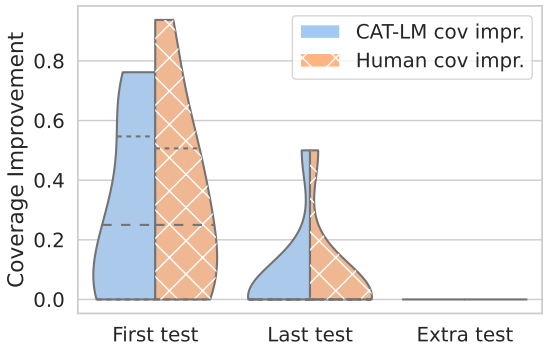
Pass Rate

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.[4]

[4]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.



(a) Coverage improvement of our model vs humans for Python.



(b) Coverage improvement of our model vs humans for Java.

Figure 4.7: Coverage improvement of our model vs humans for different languages.

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 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).

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

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 [23] 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 [19], GPT-4 [35] and EvoSuite [12]. 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.[5] 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.

[5] 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 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.

4.8 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 practise 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

Challenge 3: Reliability (Proposed Work)

The developments in Large Language Models (LLMs) has led to the widespread adoption of AI powered programming tools like Copilot and ChatGPT. However, even these high-quality code-generating tools are not always reliable and they often generate code that contains subtle bugs that are challenging to detect. So much so that studies have found that nearly 40% of participants abstain from using these tools altogether because of the time required to debug or modify the generated code [7]. The lack of reliability poses a significant hurdle in the widespread adoption of such tools. Therefore, emphasizing the need for enhancing the reliability and ease of debugging in AI-driven code generation processes.

Data science notebooks constitute a unique blend of natural language, code, graphics, and execution results [24]. The interactive nature of notebooks introduces multiple turns of interconnected natural language to code challenges [25], including the need for a more nuanced context. This includes a deeper understanding of the dataset to perform tasks like data exploration, visualization, and manipulation, as well as the execution states of various cells and the non-trivial long range dependencies between code cells [26]. The absence of this nuanced context presents a notable hurdle in automating the generation of data science notebooks, and generating reliable notebooks that extend beyond syntactic correctness poses an even greater challenge.

The lack of reliable generations does not hinder everyone from incorporating AI tools into their programming workflows. Studies found that 70% of participants who frequently do so overcome this challenge by evaluating the correctness of the generated code by executing it. Presently, LLMs are mainly trained to generate code based on prior context, and they lack the capability to use execution outputs for refining the generated code. However, relying solely on prior context is insufficient to guarantee that the generated code is bug-free. Even code that appears to be syntactically correct code may contain bugs, and executing the code can uncover these issues. This underlines the significance of code execution as a key validation step in evaluating the reliability of AI-generated code. Notebooks inherently offer the advantage of providing the execution states for each cell, which can aid with evaluating the reliability of the generated code cell.

We aim to overcome the challenge of reliability by leveraging the (often failing) execution of code generated by these LLMs as a signal. The proposed work consists of two main stages. First, we create a new evaluation framework for testing the code generation capabilities of LLMs in data science notebooks. Unlike prior benchmarks, our work will perform this

evaluation on a large scale dataset of data science notebooks that can be executed. This will allow us to evaluate the models performance on both lexical metrics as well as run time metrics by making use of the execution signal. We additionally evaluate the models inherent ability to refine the generated code by using the execution signal and use these results to establish baselines on our dataset. Next, we build a system to teach LLMs to debug generated code by leveraging the execution signal. We employ several models working in tandem: first, a model uses the notebook prefix to generate the next code cell, which is then executed. Another model observes the failing execution, which is then used to repair the generated code before presenting it to the developer. By observing this interaction we can then teach these models to debug the code they, or regular developers, generate.

5.1 Evaluation Framework

Our goal is to create an evaluation framework for the task of code completion on data science notebooks, which includes a large scale dataset of notebooks that are executable and benchmarks with baseline performance if existing open source models. Being able to execute the notebook will allow us to better evaluate generations by going beyond syntactical similarity and lexical metrics by comparing the code semantics using execution output of the code generated without the need for test cases.

Data Collection and Preprocessing

We start with 140K Python Kaggle notebooks that were collected as part of the Google AI4Code challenge[1]. We then filter out notebooks where the content is not in English, resulting in 133K notebooks. Each notebook in this dataset is originally in the form of a json file detailing the code cell contents, markdown cell contents and the order of the cells. We transform them into executable interactive python notebooks using `nbformat` python library. Next, we upgrade the leading comments in code cells to a markdown cell preceding the code cell. The leading comments are usually descriptions of what the code does, that should have ideally appeared in a markdown cell, and ensures that the models have more context on what to generate when generating the next code cell.

Execution pipeline

We first build an execution harness for each notebook using the `nbformat` and `nbconvert` Python libraries. This allows us to execute the cells in each notebook and extract the execution including error messages, and traceback when the execution fails. We then scale this up using Docker containers to safely execute each notebook in a separate container and saving all the no error notebooks and track the meta data including number of code and markdown cells, the types of errors and so on to better understand the distribution of the entire dataset. The Docker containers have been iteratively updated by analysing the most common errors and resolving them. We find that `FileNotFoundError` and `ModuleNotFoundError` were the most common errors, spanning over 75% of the notebooks on a sample of 4000 notebooks,

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

with `FileNotFoundError` appearing over 50% of the time. To resolve these, we manually download the most common missing datasets and install the most common missing modules after extracting the necessary information from the error messages and update the Docker image with this information. We then rerun the script launching a new container, attempting to execute the notebook, and storing the necessary information. We repeat this process several times, resulting in a total of 5000 notebooks.

Benchmarking and Evaluation

We benchmark the performance of various open source models on the task of code completion within notebooks. We do so by randomly removing a code cell from the notebook and task the model with generating the missing code, given the prefix context of the notebook. This process allows us to evaluate the model's ability to accurately generate code sequences without the explicit need for test cases to evaluate the functionality of the generated code. The evaluation extends beyond just syntactic correctness; we thoroughly examine the generated code by analysing its execution output and its impact on the overall functionality of the entire notebook.

We aim to replicate this evaluation process across the entire dataset, systematically comparing the performance of various open-source models on both lexical similarity metrics and run-time metrics that validate the functionality of the generated code. We will also studying the distribution of types of errors encountered during code generation. We further aim to analyze these metrics against the position of the generated cell, and the models' efficacy in debugging the code with the aid of execution signals. This multifaceted evaluation framework aims to provide a nuanced understanding of the strengths and weaknesses of different models under diverse conditions, contributing valuable insights for the task of code generation in notebook environments.

5.2 Debug-LM

In the second part of the proposed work, the goal is to build a system to teach LLMs to automatically debug the code they generate by leveraging the failing execution signal. Training any model for a specific task necessitates a substantial amount of data. To address this requirement, we initiate the data collection process by instructing the CodeLlama-34B model to generate instances of buggy code cells for the code completion task. Subsequently, we augment this dataset by prompting the same model to fix the buggy code, providing additional context such as the notebook prefix and execution information. These pairs of buggy-fixed code, coupled with the corresponding execution output, become the foundation for fine-tuning a much smaller specialized model specifically trained for the task of debugging code that might have been generated by other models or written by developers.

Data Collection (bug-fix pairs)

We start the data collection process by randomly removing a code cell from the notebook and then prompting the model to generate the missing code, given the notebook's prefix context. We then execute the generated cell, and if the resulting execution encounters an

error, we save both the buggy code cell snippet and its corresponding ground truth code cell snippet as a bug-fix pair. We repeat this process across the entire dataset to gather close to 10K buggy-fixed pairs.

Data Augmentation

To enhance the diversity and richness of our dataset, we engage in a data augmentation phase. We leverage the initial seed dataset of buggy-fixed pairs, and further prompt a model to debug the buggy code cell by providing it with the notebook prefix context and execution output. We evaluate the generated fix by executing the notebook and verifying that the overall functionality and expected outputs remain unchanged. We additionally apply filters on the generated samples by applying some threshold constraints over the lexical similarity of the ground truth code, this will eliminate generations of passing cells that are actually wrong but cannot be caught by running the notebook. The augmented dataset is expanded with new bug-fix pairs only if the generated fix is different from the original ground truth cell. The augmentation allows us to capture multiple potential fixes for a given buggy code snippet, thereby contributing to the dataset's robustness and diversity.

Debugging Model training

The data collected will then be used to fine-tune a much smaller model, DEBUG-LM, tailored specifically for the task of debugging code. The objective is to teach the model to automatically debug code by providing it with the notebook prefix, the buggy code, and the execution output. For the fine-tuning process, we will leverage LoRA tuning [117], a technique widely recognized for its optimization capabilities without needing excessive computational resources.

The expected outcome is the creation of a specialized model DEBUG-LM that is not only smaller in size, but optimized for debugging code generated by both AI models and developers, thus culminating in a powerful tool that enhances the reliability and utility of code generation for developers. Lastly, both the benchmark dataset as well as the DEBUG-LM model will open-sourced, contributing to the broader developer community.

5.3 Takeaway

LLMs often generate code that appears syntactically correct but contain subtle bugs that are usually uncovered during execution. I first built a benchmark of executable data science notebooks to highlight the importance of code execution as a key validation step in evaluating AI generated code. I then leverage the (often failing) execution of code generated by these LLMs as a signal. DEBUG-LM learns to utilize this execution signal in order to refine the generations before presenting them to the developers. Through this work, we can enhance the overall reliability of the code generated by LLMs, therefore promoting a more robust development environment.

Chapter 6

Stretch Goal: 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 **empty citation** 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 6.1 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 6.2. 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.

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

```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)
```

(a) Code method.

```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)
```

(b) Test method.

Figure 6.1: An example of a code-test method pair.

```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)
```

(a) Code-test pair before change.

```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))
```

(b) Code - test pair after change.

Figure 6.2: An example of code-test aligned pairs, before and after a change was made.

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.

Data Collection

We use GitHub Archive [1] 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[2] framework to find which

[1] https://www.gharchive.org/

[2] https://github.com/google-research/python-graphs

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. 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 the trained model to facilitate further research in this space.

Chapter 7

Proposed Contributions

My thesis is expected to make a number of contributions to improve the reliability and usability of code generated by LLMs using domain insights from software engineering that go beyond mere syntactical considerations, 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.
- CAT-LM, a specialized model trained to generate tests from code context to verify code correctness.
- A testing framework for evaluating tests generated by language models that uses both lexical and runtime metrics.
- A new benchmark dataset of executable notebooks for evaluating the code generation abilities of language models.
- Debug-LM, a specialized model trained to debug code using execution signal in data science notebooks.

Overall, my thesis work will demonstrate the significance of integrating software-specific insights when training models to make code generation more reliable and useful 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. Additionally, my work will result in several artifacts which includes 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. On the contrary, 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.

Chapter 8

Proposed Timeline

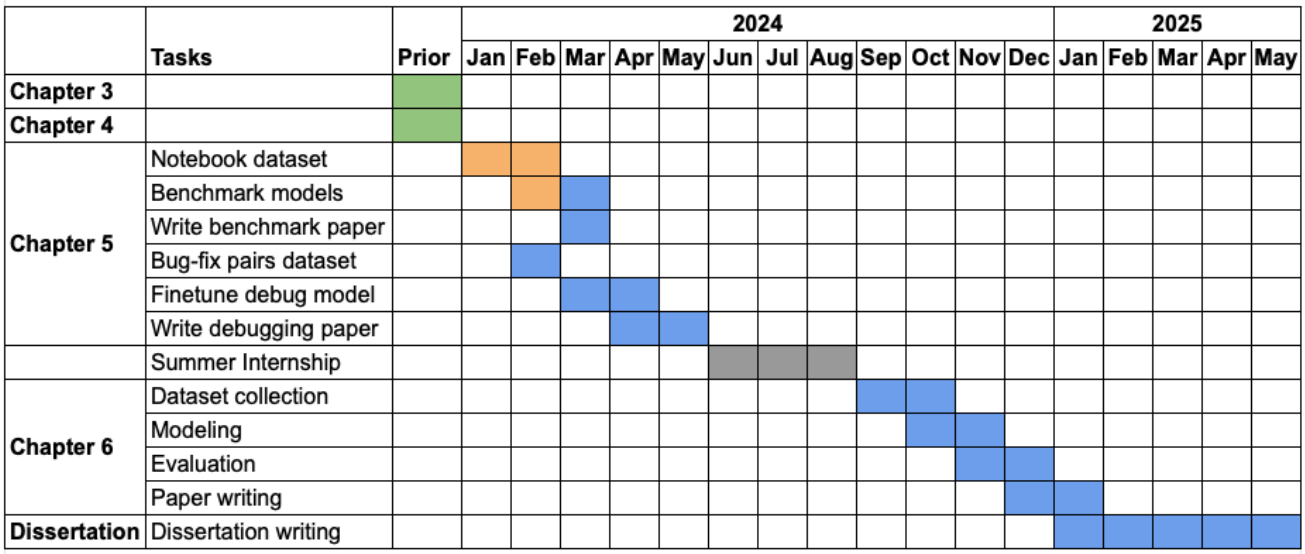


Figure 8.1: Proposed timeline. Legend - Green: Completed tasks, Orange: In-progress tasks, Blue: Planned tasks, Gray: Summer internship - no planned tasks.

Figure 8.1 outlines the proposed timeline for my thesis. I am currently working on enhancing the reliability of generated code by training LLMs to collaboratively debug code using execution signals (Chapter 5). I have been working on building the dataset of executable notebooks, using which I plan to benchmark the performance of various open-source models. Simultaneously, I plan to prepare the dataset of bug-fix code pairs, which I will use to finetune a smaller model, and write papers with the findings by end of the semester.

Looking ahead to the Fall 2024 semester, I have outlined a stretch goal centered around addressing the challenge of code-test coevolution (See Chapter 6). This entails a systematic approach involving data collection, preprocessing, modeling, and evaluation. I plan to document our findings in a research paper by the end of the Fall semester. To ensure a well-structured thesis, I have dedicated Spring 2025 primarily to writing my dissertation and preparing the final defense. This timeline also accommodates any additional time required for paper submissions or revisions. Lastly, I aim to defend my thesis in May 2025.

References

[1] *ChatGPT*, Nov. 2022. [Online]. Available: https://openai.com/blog/chatgpt/.

[2] *GitHub Copilot*, 2021. [Online]. Available: https://github.com/features/copilot.

[3] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models," *CoRR*, vol. arXiv:2203.15556, 2022.

[4] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, *Scaling laws for neural language models*, 2020. arXiv: 2001.08361 [cs.LG].

[5] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, *An empirical study on usage and perceptions of llms in a software engineering project*, 2024. arXiv: 2401.16186 [cs.SE].

[6] P. Vaithilingam, T. Zhang, and E. 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. [Online]. Available: https://doi.org/10.1145/3491101.3519665.

[7] 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)*, Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 605–617. [Online]. Available: https://doi.ieeecomputersociety.org/.

[8] S. Barke, M. B. James, and N. Polikarpova, *Grounded copilot: How programmers interact with code-generating models*, 2022. arXiv: 2206.15000 [cs.HC].

[9] M. Beller, G. Gousios, A. Panichella, and A. 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*, ser. ESEC/FSE '15, Bergamo, Italy, 2015, pp. 179–190.

[10] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" In *International Conference on Software Engineering*, ser. ICSE '15, Florence, Italy, 2015, pp. 559–562.

[11] E. Dinella, G. Ryan, T. Mytkowicz, and S. Lahiri, "Toga: A neural method for test oracle generation," in *International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 2130–2141.

[12] G. Fraser and A. 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*, ser. ESEC/FSE '11, Szeged, Hungary, 2011, pp. 416–419.

[13] C. Brandt and A. Zaidman, "Developer-centric test amplification: The interplay between automatic generation human exploration," *Empirical Software Engineering*, vol. 27, no. 4, 2022, ISSN: 1382-3256.

[14] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Survey*, vol. 51, no. 3, pp. 50–88, 2018.

[15] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," *CoRR*, vol. abs/2002.05800, 2020.

[16] J. Villmow, J. Depoix, and A. Ulges, "ConTest: A Unit Test Completion Benchmark featuring Context," in *Workshop on Natural Language Processing for Programming*, Aug. 2021, pp. 17–25.

[17] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities," in *International Conference on Software Maintenance and Evolution*, 2020, pp. 523–533.

[18] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021.

[19] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A Conversational Paradigm for Program Synthesis," *CoRR*, vol. abs/2203.13474, 2022.

[20] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," *CoRR*, vol. abs/2204.05999, 2022.

[21] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient Training of Language Models to Fill in the Middle," *CoRR*, vol. abs/2207.14255, 2022.

[22] R. Li, L. B. Allal, Y. Zi, *et al.*, "Starcoder: May the source be with you!," 2023. arXiv: 2305.06161 [cs.CL].

[23] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *International Conference on Software Engineering*, ser. ICSE '23, 2023, pp. 2111–2123.

[24] J. M. Perkel, "Reactive, reproducible, collaborative: computational notebooks evolve," *Nature*, vol. 593, no. 7857, pp. 156–157, May 2021. DOI: 10.1038/d41586-021-01174-. [Online]. Available: https://ideas.repec.org/a/nat/nature/v593y2021i7857d10.1038_d41586-021-01174-w.html.

[25] G. Heyman, R. Huysegems, P. Justen, and T. Van Cutsem, "Natural language-guided programming," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 39–55, ISBN: 9781450391108. DOI: 10.1145/3486607.3486749. [Online]. Available: https://doi.org/10.1145/3486607.3486749.

[26] P. Yin, W.-D. Li, K. Xiao, *et al.*, *Natural language to code generation in interactive data science notebooks*, 2022. arXiv: 2212.09248 [cs.CL].

[27] *Introduction to LLMs*. [Online]. Available: https://developers.google.com/machine-learning/resources/intro-llms.

[28] D. Hiemstra, "N-gram models," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 1910–1910, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_935. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_935.

[29] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132 306, Mar. 2020, ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306. [Online]. Available: http://dx.doi.org/10.1016/j.physd.2019.132306.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[31] L. Ouyang, J. Wu, X. Jiang, *et al.*, *Training language models to follow instructions with human feedback*, 2022. arXiv: 2203.02155 [cs.CL].

[32] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*, version 1.0, If you use this software, please cite it using these metadata., Mar. 2021. DOI: 10.5281/zenodo.5297715. [Online]. Available: https://doi.org/10.5281/zenodo.5297715.

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

[34] A. Chowdhery, S. Narang, J. Devlin, *et al.*, *Palm: Scaling language modeling with pathways*, 2022. arXiv: 2204.02311 [cs.CL].

[35] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[36] H. Touvron, T. Lavril, G. Izacard, *et al.*, "LLaMA: Open and Efficient Foundation Language Models," *CoRR*, vol. abs/2302.13971, 2023.

[37] L. Belzner, T. Gabor, and M. Wirsing, "Large language model assisted software engineering: Prospects, challenges, and a case study," in *Bridging the Gap Between AI and Reality*, B. Steffen, Ed., Cham: Springer Nature Switzerland, 2024, pp. 355–374, ISBN: 978-3-031-46002-9.

[38] C. Arora, J. Grundy, and M. Abdelrazek, *Advancing requirements engineering through generative ai: Assessing the role of llms*, 2023. arXiv: 2310.13976 [cs.SE].

[39] Y. Wei, C. S. Xia, and L. 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*, ser. ESEC/FSE 2023, <conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 172–184, ISBN: 9798400703270. DOI: 10.1145/3611643.3616271. [Online]. Available: https://doi.org/10.1145/3611643.3616271.

[40] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, *Inferfix: End-to-end program repair with llms*, 2023. arXiv: 2303.07263 [cs.SE].

[41] T. Ahmed and P. Devanbu, "Better patching using llm prompting, via self-consistency," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1742–1746. DOI: 10.1109/ASE56229.2023.00065.

[42] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 2023, pp. 284–289. DOI: 10.1109/MedAI59581.2023.00044.

[43] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, *Cigar: Cost-efficient program repair with llms*, 2024. arXiv: 2402.06598 [cs.SE].

[44] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, <conf-loc>, <city>Rochester</city>, <state>MI</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3559555. [Online]. Available: https://doi.org/10.1145/3551349.3559555.

[45] Q. Luo, Y. Ye, S. Liang, *et al.*, *Repoagent: An llm-powered open-source framework for repository-level code documentation generation*, 2024. arXiv: 2402.16667 [cs.CL].

[46] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, *Together we go further: Llms and ide static analysis for extract method refactoring*, 2024. arXiv: 2401.15298 [cs.SE].

[47] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang, *Towards understanding the capability of large language models on code clone detection: A survey*, 2023. arXiv: 2308.01191 [cs.SE].

[48] J. Lu, L. Yu, X. Li, L. Yang, and C. 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)*, 2023, pp. 647–658. DOI: 10.1109/ISSRE59848.2023.00026.

[49] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani, *Frustrated with code quality issues? llms can help!* 2023. arXiv: 2309.12938 [cs.AI].

[50] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, *Using an llm to help with code understanding*, 2024. arXiv: 2307.08177 [cs.SE].

[51] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL].

[52] L. B. Allal, R. Li, D. Kocetkov, *et al.*, *Santacoder: Don't reach for the stars!* 2023. arXiv: 2301.03988 [cs.SE].

[53] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, *Codegen2: Lessons for training llms on programming and natural languages*, 2023. arXiv: 2305.02309 [cs.LG].

[54] S. Black, S. Biderman, E. Hallahan, *et al.*, "GPT-NeoX-20B: An open-source autoregressive language model," in *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, A. Fan, S. Ilic, T. Wolf, and M. Gallé, Eds., virtual+Dublin: Association for Computational Linguistics, May 2022, pp. 95–136. DOI: 10.18653/v1/2022.bigscience-1.9. [Online]. Available: https://aclanthology.org/2022.bigscience-1.9.

[55] Y. Li, D. Choi, J. Chung, *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, ISSN: 1095-9203. DOI: 10.1126/science.abq1158. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158.

[56] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, *Codet5+: Open code large language models for code understanding and generation*, 2023. arXiv: 2305.07922 [cs.CL].

[57] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are Few-Shot learners," in *Advances in Neural Information Processing Systems*, 2020, pp. 1877–1901.

[58] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," *CoRR*, vol. abs/2202.13169, 2022.

[59] Z. Feng, D. Guo, D. Tang, *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/.

[60] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, *Learning and evaluating contextual embedding of source code*, 2020. arXiv: 2001.00059 [cs.SE].

[61] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[62] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, *Unified pre-training for program understanding and generation*, 2021. arXiv: 2103.06333 [cs.CL].

[63] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, *Wizardcoder: Empowering code large language models with evol-instruct*, 2023. arXiv: 2306.08568 [cs.CL].

[64] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, *Octopack: Instruction tuning code large language models*, 2024. arXiv: 2308.07124 [cs.CL].

[65] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 21 558–21 572. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

[66] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, "The robots are coming: Exploring the implications of openai codex on introductory programming," in *Proceedings of the 24th Australasian Computing Education Conference*, ser. ACE '22, <conf-loc>, <city>Virtual Event</city>, <country>Australia</country>, </conf-loc>: Association for Computing Machinery, 2022, pp. 10–19, ISBN: 9781450396431. DOI: 10.1145/3511861.3511863. [Online]. Available: https://doi.org/10.1145/3511861.3511863.

[67] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, ACM, Nov. 2023. DOI: 10.1145/3576915.3623157. [Online]. Available: http://dx.doi.org/10.1145/3576915.3623157.

[68] S. 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*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 319–321, ISBN: 9781450392235. DOI: 10.1145/3510454.3522684. [Online]. Available: https://doi.org/10.1145/3510454.3522684.

[69] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5. DOI: 10.1145/3524842.3528470.

[70] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, *Is chatgpt the ultimate programming assistant – how far is it?* 2023. arXiv: 2304.11938 [cs.SE].

[71] A. Hellas, J. Leinonen, S. Sarsa, C. Koutcheme, L. Kujanpää, and J. 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*, ser. ICER '23, <conf-loc>, <city>Chicago</city>, <state>IL</state>, <country>USA</country>, </conf-loc>: Association for

Computing Machinery, 2023, pp. 93–105, ISBN: 9781450399760. DOI: 10.1145/3568813.3600139. [Online]. Available: https://doi.org/10.1145/3568813.3600139.

[72] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. 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*, ser. Koli Calling '23, <conf-loc>, <city>Koli</city>, <country>Finland</country>, </conf-loc>: Association for Computing Machinery, 2024, ISBN: 9798400716539. DOI: 10.1145/3631802.3631806. [Online]. Available: https://doi.org/10.1145/3631802.3631806.

[73] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," vol. 31, no. 2, Mar. 2022, ISSN: 1049-331X. DOI: 10.1145/3487569. [Online]. Available: https://doi.org/10.1145/3487569.

[74] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 21–29, ISBN: 9781450392730. DOI: 10.1145/3520312.3534864. [Online]. Available: https://doi.org/10.1145/3520312.3534864.

[75] M. Hirzel, "Low-code programming models," *Communications of the ACM (CACM)*, vol. 66, no. 10, pp. 76–85, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3587691.

[76] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178. [Online]. Available: https://doi.org/10.1109/SEAA51224.2020.00036.

[77] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME - the Konstanz information miner: Version 2.0 and beyond," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 26–31, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1656274.1656280.

[78] J. Demsar, B. Zupan, G. Leban, and T. Curk, "Orange: From experimental machine learning to interactive data mining," in *European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, 2004, pp. 537–539. [Online]. Available: https://doi.org/10.1007/978-3-540-30116-5%5C_58.

[79] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278.

[80] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with Blockly," in *Blocks and Beyond Workshop (B&B)*, Oct. 2017. [Online]. Available: https://doi.org/10.1109/BLOCKS.2017.8120404.

[81] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[82] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Symposium on Visual Languages – Human Centric Computing (VL/HCC)*, Dec. 2004. [Online]. Available: https://doi.org/10.1109/VLHCC.2004.47.

[83] M. Resnick, J. Maloney, A. Monroy-Hernández, *et al.*, "Scratch: Programming for all," *Communications of the ACM (CACM)*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1592761.1592779.

[84] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *Transactions on Software Engineering (TSE)*, 2019. DOI: 10.1109/TSE.2019.2937083.

[85] Y. Wang, W. Wang, S. Joty, and S. C. 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)*, 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685/.

[86] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, *A conversational paradigm for program synthesis*, 2022. [Online]. Available: https://arxiv.org/abs/2203.13474.

[87] M. Voelter and S. Lisson, "Supporting diverse notations in MPS' projectional editor.," in *Workshop on The Globalization of Modeling Languages (GEMOC)*, 2014, pp. 7–16. [Online]. Available: https://hal.inria.fr/hal-01074602/file/GEMOC2014-complete.pdf#page=13.

[88] G. Baudart, M. Hirzel, K. Kate, P. Ram, A. Shinnar, and J. Tsay, "Pipeline combinators for gradual AutoML," in *Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf.

[89] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON schema," in *International Conference on World Wide Web (WWW)*, 2016, pp. 263–273. [Online]. Available: https://doi.org/10.1145/2872427.2883029.

[90] S. L. Tanimoto, "A perspective on the evolution of live programming," in *International Workshop on Live Programming (LIVE)*, 2013, pp. 31–34. [Online]. Available: https://doi.org/10.1109/LIVE.2013.6617346.

[91] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases – an introduction," *Natural Language Engineering*, vol. 1, no. 1, pp. 29–81, 1995. [Online]. Available: https://doi.org/10.1017/S135132490000005X.

[92] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2018. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[93] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[94] S. Lu, D. Guo, S. Ren, *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *ArXiv*, vol. abs/2102.04664, 2021.

[95] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[96] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[97] M. Chen, J. Tworek, H. Jun, *et al.*, *Evaluating large language models trained on code*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374.

[98] D. R. Garrison, M. Cleveland-Innes, M. Koole, and J. Kappelman, "Revisiting methodological issues in transcript analysis: Negotiated coding and reliability," *The Internet and Higher Education*, vol. 9, no. 1, pp. 1–8, 2006.

[99] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, 4th. SAGE publications, 2013.

[100] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[101] L. v. Werra, *Codeparrot*, https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot.

[102] *SentencePiece*. [Online]. Available: https://github.com/google/sentencepiece.

[103] *GPT-neox Toolkit*. [Online]. Available: https://github.com/EleutherAI/gpt-neox.

[104] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, 2022.

[105] *GitHub REST API*. [Online]. Available: https://docs.github.com/en/rest.

[106] M. 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*, ser. SPLASH '19, 2019, pp. 143–153.

[107] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on GitHub," in *Proceedings of the ACM on Programming Languages*, ser. OOPSLA '17, vol. 1, 2017, pp. 1–28.

[108] *TheFuzz: Fuzzy String Matching in Python*. [Online]. Available: https://github.com/seatgeek/thefuzz.

[109] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *International Conference on Quality Software*, ser. ICQS '13, 2013, pp. 103–112.

[110] H. H. F. d. Souza, I. Wiese, I. Steinmacher, and R. Ré, "A characterization study of testing contributors and their contributions in open source projects," in *Brazilian Symposium on Software Engineering*, ser. SBES '22, 2022, pp. 95–105.

[111] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *Annual Meeting of the Association for Computational Linguistics*, ser. ACL '18, 2018, pp. 66–75.

[112] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Conference on Text Summarization Branches Out*, 2004, pp. 74–81.

[113] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," *CoRR*, vol. abs/2009.10297, 2020.

[114] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with Hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[115] A. LeClair, S. Jiang, and C. McMillan, "A Neural model for generating natural language summaries of program subroutines," in *International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 795–806.

[116] S. Lu, D. Guo, S. Ren, *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[117] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9.