Large Language Models for Build System Maintenance

An Empirical Study of CodeGen's Next-Line Prediction

by

Akinbowale Akin-Taylor

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Akinbowale Akin-Taylor 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Build systems play a crucial role in software development and are responsible for compiling source code into executable programs. Despite their importance, build systems often receive limited attention because their impact is not directly visible to end users. This oversight can lead to inadequate maintenance, frequent build failures, and disruptions that require additional resources. Recognising and addressing the maintenance needs of build systems is essential to preventing costly disruptions and ensuring efficient software production.

In this thesis, I explore whether applying a Large Language Model (LLM) can reduce the burden of maintaining build systems. I aim to determine whether the prior content in build specifications provides sufficient context for an LLM to generate subsequent lines accurately. I conduct an empirical study on CodeGen, a state-of-the-art Large Language Model (LLM), using a dataset of 13,343 Maven build files. The dataset consists of the Expert dataset from the Apache Software Foundation (ASF) for fine-tuning (9,426 build files) and the Generalised dataset from GitHub for testing (3,917 build files). I observe that (i) fine-tuning on a small portion of data (i.e., 11% of fine-tuning datasets) provides the largest improvement in performance by 13.93% (ii) When applied to the Generalised dataset, the fine-tuned model retains 83.86% of its performance, indicating that it is not overfitted. Upon further investigation, I classify build-code content into functional and metadata subgroups based on enclosing tags. The fine-tuned model performs substantially better in suggesting functional than metadata build-code. The findings highlight the potential of leveraging LLMs like CodeGen to relieve the maintenance challenges associated with build systems, particularly in functional content. My thesis highlights the limitations of large language models in suggesting the metadata components of build code. Future research should focus on developing approaches to enhance the accuracy and effectiveness of metadata generation.

Acknowledgements

I want to thank my supervisors, Professor Shane McIntosh and Professor Mei Nagappan, for their steadfast support. They have consistently believed in me and motivated me throughout the process of working on my thesis. Their combined expertise and encouragement have been instrumental in shaping my thesis. I am eternally grateful to them.

I thank Dr. Rungroj Maipradit (Ice) for his insightful feedback, valuable ideas, and generous time. I also thank Professor Mahmoud Alfadel for his constant and timely advice throughout my master's program.

I am grateful to Professor Robin Cohen and Professor Mike Godfrey for their contributions as my thesis readers. Their considerate comments and constructive feedback have significantly enhanced the quality of this thesis.

I am grateful to my colleagues and friends at The Software REBELs (Software Repository Excavation and Build Engineering Labs) and The SWAG (Software Architecture Group) Labs.

To my big family—the Akin-Taylors, the Adepejus, the Fatimilehins, and the Ezimokhais—being a part of these families is truly a blessing. Your love and support have been the foundation of my journey. Your encouragement, wisdom, and countless moments of joy have shaped me into who I am today. Thank you for standing by me in difficult times and cheering me on in moments of triumph. I carry your love with me every step of the way and am endlessly grateful for our bond.

I thank my day ones, Stephen, Mayowa, Chijioke, Dubem, Richy, and Efe, for their love and support, and my newfound friends, to mention a few—Kelechi and Cally, thanks for the camaraderie.

I am grateful to my mentor, Akinjide, for always giving timely advice, supporting me, keeping me on my toes, and helping me execute, forever making me see the bigger picture—cheers to more collaborations.

Dedication

To my mother, Folasade, with love and gratitude.
I thank our Father, The Elohiym, for He is a righteous El.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables xi

1 Introduction 1

2 Related Work 5

2.1 Build Systems 5

2.2 Application of Large Language Models 6

3 Study Design 8

3.1 Data Curation 8

3.1.1 Collect data 8

3.1.2 Filter projects 11

3.1.3 Validate Maven build files 13

3.2 Data Preprocessing . . . 14
3.2.1 Mitigate noise . . . 14
3.2.2 Engineer prompts . . . 14
3.3 Build Code Generation . . . 17
3.4 Data Normalisation . . . 18
3.5 Analysis . . . 18

4 Study Results **20**
4.1 RQ1. How much fine-tuning is required by CodeGen to effectively infer build specifications? . . . 20
4.1.1 Approach . . . 20
4.1.2 Results . . . 22
4.2 RQ2. How well does the fine-tuned CodeGen perform on an unseen dataset? 22
4.2.1 Approach . . . 22
4.2.2 Results . . . 24

5 Follow-Up Analysis **26**

6 Threats to Validity **32**
6.1 Construct Validity . . . 32
6.2 Internal Validity . . . 32
6.3 External Validity . . . 33

7 Conclusion **34**

References **35**

APPENDICES **46**

A **47**

A.1 Large Language Models 47

A.2 Prompt Engineering 47

A.3 User-study Evaluation 49

List of Figures

1.1 Excerpt of a Maven build log showing a dependency resolution error from Macho et al. [46]. 2

1.2 Sample of prompts and code generations by CodeGen by Nijkamp et al. [57]. 3

3.1 Example of Expert dataset (top) and Generalised dataset (bottom). 9

3.2 An overview of our study design approach. 10

3.3 The number of Java and Maven non-comment lines in each TLP. I study the 39 TLPs with at least 10,000 lines of Maven code (above the horizontal dashed line). 12

3.4 Examples of Maven build code prompts. 16

4.1 BLEU score distributions for our base and fine-tuned models on the test sample of the Expert dataset, varying by the proportion of fine-tuning data. Interquartile ranges (Q1, Q2, and Q3) and confidence intervals (2.5th percentile—LCL and 97.5th percentile—UCL) are highlighted. The Q2 line represents the median BLEU scores, indicating that the most significant performance gain comes from the first 11% of the fine-tuning dataset. Additional data yields diminishing improvements 21

4.2 Comparative performance of our fine-tuned model on both datasets. The dotted lines represent the median BLEU score, and the line between them represents the optimism of our fine-tuned model on both datasets, highlighting an optimism score of 9.31 BLEU score points. 23

4.3 The BLEU score distribution of generated build code from our base model and fine-tuned model on the Generalised dataset. The main difference between both models occurs at the two extremes of the plot. 24

5.1 Compares tag percentages of build files between the test sample of our Expert dataset and the Generalised dataset. The violin plots illustrate the distribution of the percentage of content each tag occupies across build files. Tags are arranged from top to bottom based on decreasing median percentages, with the highest median at the top. 27

5.2 The BLEU score distribution of our fine-tuned model across both datasets, categorised by tag type. The fine-tuned model exhibits a higher average BLEU in generating functional tags compared to metadata tags. 28

5.3 Snippet of Maven build code showing examples of functional tags and metadata tags. 29

A.1 The transformer model architecture, from "Attention Is All You Need" by Vaswani et al. [71]. 48

A.2 Example of prompts — simple prompt (left) and complex prompt (right). . 49

List of Tables

3.1 The size of the studied datasets. 18

5.1 A summary of the Maven functional tags within our datasets. 30

5.2 A summary of the Maven metadata tags within our datasets. 31

Chapter 1

Introduction

Build systems automate the conversion of source code into executable programs [55]. By interpreting a set of build files, build tools orchestrate the compilation, linking, and packaging of code into a runnable form [75]. These build files detail the necessary steps, such as downloading dependencies, compiling sources, linking libraries, and preparing other prerequisites to create the final product. In modern software development, build systems manage more than the production of deliverables. They also integrate testing [50] and assist in deployment tasks to streamline the development life cycle [20]. This comprehensive approach allows developers to focus on writing source code while the build system manages the intricacies of producing, testing, and deploying updated versions of the software system.

Prior work has proposed approaches to ease the process of build maintenance. Tools like MAKAO helped software engineers better visualise build systems [5]. Another tool, SYMake, assisted engineers in detecting code smells in build files [69]. Although these studies help improve build maintenance, software engineers require significant involvement. More recently, work has been done on automating fixes to build failures. For example, in one study, HireBuild was created to fix build failures by generating patches based on historical fixes [34]. Another study introduces BUILDMEDIC, which generates candidate repair plans for build failures [46]. While these approaches have shown promising results, they only aid after a build fails. Figure 1.1 shows the example of a Maven exception log.

To reduce the effort required by software engineers in maintaining build systems, I explore the application of an LLM—CodeGen, on build code generation (see Figure 1.2). Recent studies indicate that LLMs have shown promise in generating source code. One notable example is PYMT5, an LLM specifically designed to generate Python methods

```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------
[INFO] Total time: 10.362 s
[INFO] Finished at: 2017-08-04T08:52:44+02:00
[INFO] Final Memory: 25M/264M
[INFO] ------------------------------------------------------------
[ERROR] Failed to execute goal on project flux-core: Could
not resolve dependencies for project com.github.ptgoetz:
flux-core:jar:0.2.3-SNAPSHOT: Could not find artifact org.
apache.storm:storm-hbase:jar:0.11.0-SNAPSHOT in sonatypenexus-
snapshots (https://oss.sonatype.org/content/
repositories/snapshots) -> [Help 1]
```

Figure 1.1: Excerpt of a Maven build log showing a dependency resolution error from Macho et al. [46].

from docstrings [14]. Similarly, the Codex model was fine-tuned for code generation [13]. Another study introduced CodeBERT, an LLM used to generate code documentation [24]. Lastly, another study highlights that CodeGen, an open-source LLM, is competitive with the Codex model in generating source code [57].

Since CodeGen is primarily trained on natural language and source code, I fine-tune the CodeGen model to specialise in the task of generating build code. Fine-tuning improves the performance of a pre-trained model by continuing the training process on a specific dataset, thereby adapting the model to the characteristic of that data [19].

My thesis aims to understand how much LLMs can support developers in maintaining build systems. I apply CodeGen to an Expert dataset and a Generalised dataset.

(RQ1) How much fine-tuning is required by CodeGen to effectively infer build specifications?

Motivation: Previous studies demonstrate the proficiency of LLMs in understanding and creating code-related content [78]; however, their application to generating build code remains largely unexplored. To address this gap, I conducted an experiment to determine whether fine-tuning the CodeGen model is necessary for generating build code and, if so, to what extent fine-tuning would be beneficial.

Results: Fine-tuning indeed improves the performance of our model when generating build code in the Expert dataset. To achieve the biggest increase in performance, our model requires 11% of the fine-tuning dataset. I also observe that all fine-tuned

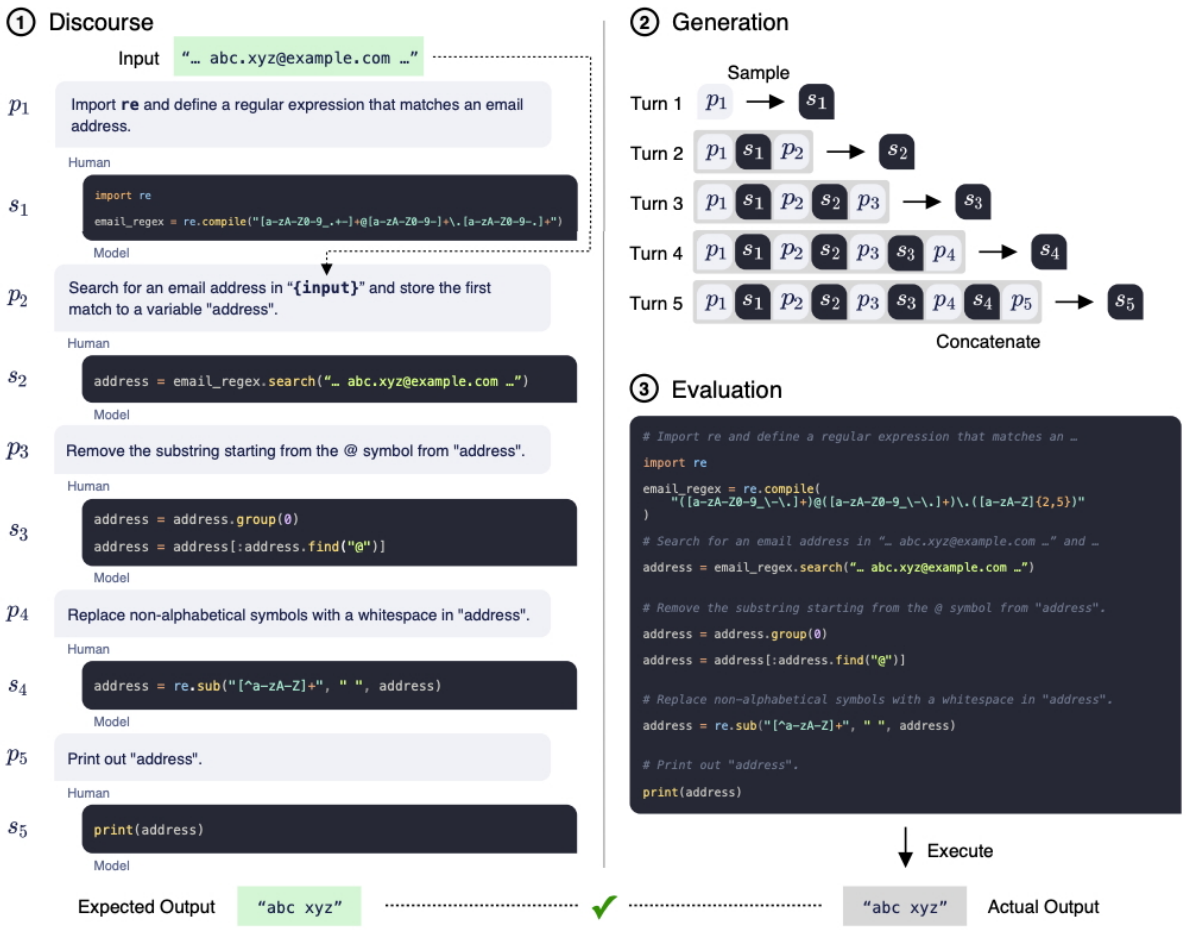


Figure 1.2: Sample of prompts and code generations by CodeGen by Nijkamp et al. [57].

models improve predictability. Their median Interquartile Range (IQR) of BLEU scores is 11.71 (representing a 29.24% decrease), and the median Confidence Interval (CI) is 36.00 (representing a 27.44% decrease) compared to our base model.

(RQ2) How well does the fine-tuned CodeGen perform on an unseen dataset?

Motivation: The primary objective is to assess the performance (i.e., accuracy of the generations) of the fine-tuned model on data it has not been trained on. This evaluation is crucial as it helps gauge the potential of the model in real-world applications by testing its ability to generalise across various and diverse contexts. Understanding how well the model adapts to unfamiliar data can significantly influence its deployment strategies and further development.

Results: When applied to the Generalised dataset, the performance of our fine-tuned model decreases by 14.6%. On the other hand, our fine-tuned model improves in terms of completely inaccurate recommendations by 24.18% and in completely accurate recommendations by 15.02%.

Inspired by the results of our RQs, we conduct a follow-up analysis to investigate why we observe an overall degradation in the performance of our model when applied to our Generalised dataset. Our findings reveal that the lack of two metadata tags (i.e., tags that do not affect the functionality of the build process), `contributors` and `mailingLists`, contribute a disproportionate amount to the decrease in performance. Specifically, the `contributors` tag never appears in the Generalised dataset, while the `mailingLists` tag only exists in one project.

Chapter 2

Related Work

This section positions our research with respect to the existing literature on build systems, applications of LLMs, CodeGen, and fine-tuning.

2.1 Build Systems

Build systems help software engineering practitioners automate the software delivery process; however, studies have shown that build systems require maintenance, which can be seen as overhead. Build maintenance refers to changes made to the build system of a software project as it evolves [32]. Kumfert and Epperly [41] showed that the maintenance of build systems can account for up to 35% of overhead in the software development process. McIntosh [48] found that at Mozilla, build maintenance creates up to 27% overhead on source code development and a 44% overhead on test development. Seo et al. [66] found that improper maintenance of build systems can increase this overhead, as they can lead to build breakages.

Build specifications define the dependencies and configurations necessary to compile and execute source code [60]. Software engineers have various options of build tools available to help build their source code. These build tools are written (or specified) in diverse formats, including low-level (e.g., Makefiles for the make tool), abstraction-based, framework-driven approaches (e.g., Maven pom.xml files) [52].

Previous work has focused on reducing the overhead acquired from maintaining build systems. For example, Adams [4] created the MAKAO tool to visually represent and analyse build dependencies. In another study, Hardt and Munson [32] created Formiga,

a tool designed to perform basic Ant maintenance tasks. Hassan and Wang [34] created Hirebuild, a tool that automatically uses historical data on build specification patches to fix build failures. They do so by extracting 175 build failures and their corresponding fixes from the TravisTorrent dataset and then using the first 135 fixes to generate fix patterns and the remaining 40 failures as test samples. Their results found that their tool could fix 45% of the reproducible build failures used in their study.

It is also important to mention that build maintenance can introduce overhead during execution. Companies like Google [82] and Microsoft [22] have deployed cloud-based build systems that leverage distributed caching and dependency parallelisation to improve build efficiency. In another study, Vakilian et al. [70] presented tools to decompose and refine unused build segments by not re-invoking unnecessary build commands.

These previous works have explored build systems comprehensively and the efforts to reduce overhead during maintenance or execution. My work continues in the same direction, representing the first attempt to understand build systems through the lens of LLMs. In the subsequent subsection, I highlight the capability of LLMs when applied to various source code tasks.

2.2 Application of Large Language Models

Large Language Models (LLMs) are computational models capable of understanding and generating human language and code [17, 28, 40]. These advanced language models feature massive parameter sizes and exceptional learning capabilities [38, 83].

Prior work has explored the applicability of LLMs to source code. Chen et al. [13] demonstrated that the Codex model was able to solve 70.2% of their problem set of generating correct Python code using repeated sampling (i.e., 100 samples per problem). Their study showed that Codex outperformed other LLMs on this dataset, such as GPT-3 and GPT-J. Similarly, Xu et al. [78] found that open-source LLMs (e.g., GPT-J, GPT-Neo, and CodeParrot) achieved comparable results to the state-of-the-art Codex model in several programming languages (e.g. C, Python, Java). Their study introduced PolyCoder, a model based on GPT-2 architecture with approximately 2.7 billion parameters. It outperformed all other models, including Codex, specifically in the C programming language.

Recent studies have shown the ability to use LLMs to improve code maintainability. For example, Dillmann et al. [18] evaluated how code generated by open-source LLMs performed on a Likert scale for readability, understandability, complexity, modularity, and overall maintainability. These evaluations were informed by domain experts, and their

results indicated a correlation between the Logical Lines of Code (LLOC) and the cross-entropy of the results generated by LLMs, with higher cross-entropy values correlating with lower maintainability. Bouzenia et al. [9] introduced RepairAgent, which uses an LLM agent concept for automated bug detection and code repair. Their approach solved 164 bugs, including 39 ones that prior techniques could not fix.

Despite these promising advances in applying LLMs to source code analysis, management, and repair, a significant gap remains in their application on build code. Build systems have complex dependencies, changing environments, and platform-specific needs, presenting challenges beyond the usual tasks, such as generating code and fixing bugs.

Chapter 3

Study Design

This chapter consists of five main phases. First, I describe my *Data Curation* and rationale for selecting the studied datasets (Section 3.1). Next, I describe my approaches to *Data Preprocessing* (Section 3.2), *Build Code Generation* (Section 3.3), and *Data Normalisation* (Section 3.4). Finally, I describe my *Analysis* and the rationale for conducting it in that manner (Section 3.5).

3.1 Data Curation

This thesis aims to develop a framework to evaluate the effectiveness of CodeGen in generating build code. To do so, I use an Expert dataset, which is leveraged to fine-tune our CodeGen model, and a Generalised dataset, which is used to test the performance of the fine-tuned model. Figure 3.1 shows a sample of build code from expert and generalised datasets. Figure 3.2 provides an overview of the three steps in the data curation process. I describe each step below:

3.1.1 Collect data

In this step, I collect the datasets that I use in our study.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven</artifactId>
  <version>4.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>Apache Maven</name>
  <properties>
    <mavenVersion>4.0.0</mavenVersion>
    <encoding>UTF-8</encoding>
  </properties>
    ...
  <contributors>
    ...
  </contributors>
  ...
</project>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>short-sample-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
      ...
    </properties>

    <dependencies>
      ...
    </dependencies>

    <build>
      ...
    </build>
  ...
</project>
```

Figure 3.1: Example of Expert dataset (top) and Generalised dataset (bottom).

Expert dataset

The primary objective of the Expert dataset is to fine-tune our CodeGen model and enhance its build code generation capabilities. Hence, to control for confounding factors (e.g., differing release practices), I focus on a software community that meets two inclusion

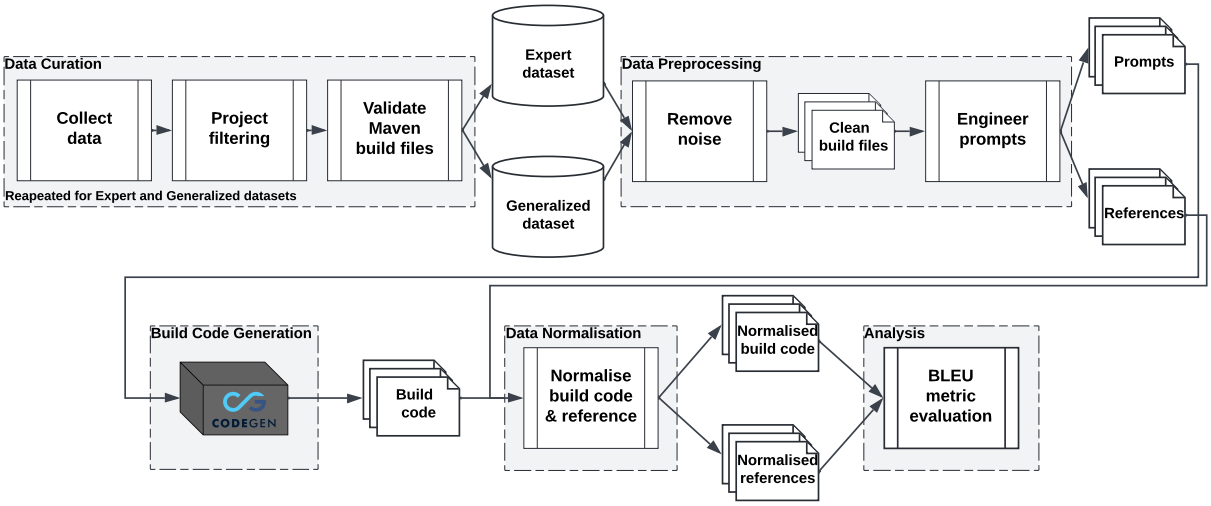


Figure 3.2: An overview of our study design approach.

criteria:

Criterion 1: Homogeneity of build tool adoption. Build tools use a variety of representations (e.g., XML, plain text) and subscribe to different paradigms [52] (e.g., abstraction-based, framework-driven). Effort is required to analyse each tool correctly. As an initial study of CodeGen for build systems, I focus on a deep analysis of a single tool. Should the results prove promising, future work can explore generalizability to other build tools.

Criterion 2: Varying project characteristics. To ensure that the results are not overly specific to a small set of projects, I study a community that develops a large number of projects of varying sizes and domains.

I choose to study projects from the Apache Software Foundation (ASF), which satisfies Criterion 1 because a large proportion (67%) of ASF projects are built using Maven. The ASF also satisfies Criterion 2, with projects of varying sizes spanning 24 domains, such as big data analysis and web frameworks.[1]

[1] http://apache.org/index.html#projects-list

Generalised dataset

I curate a second dataset of projects. Since the Expert dataset adopts Maven as its build tool, I collect a comparable set of projects using Maven. Unlike the Expert dataset, this dataset includes public projects on GitHub that are not part of the ASF organisation. This broader scope helps validate my observations on build specifications in a non-ASF setting.

To collect the Generalised dataset, I begin by querying the GitHub Search tool[2] for Java projects that were updated in 2023. The query returns a list of 1,563 projects. To extract build files, I begin by cloning these repositories. I successfully clone 1,548 projects. The 15 other projects could not be accessed because they are no longer publicly available (e.g., due to deletion or being made private).

3.1.2 Filter projects

I describe the procedure and rationale for filtering projects from the Expert and Generalised datasets.

Expert dataset

Select top-level projects. Projects fully endorsed and actively governed by the ASF are called Top-Level Projects (TLPs). To provide a path for other incoming projects to become TLPs (e.g., projects in early development phases or external projects becoming familiar with ASF governance), the ASF provides "podling" status in the Apache Incubator.[3] I aim to study mature projects that have build systems of substance. Therefore, I begin by selecting the 204 TLPs as candidates.

Select projects built with the most prevalent technology. Build tools follow different paradigms [52] that have implications for what must be expressed in build code. For example, low-level technologies like `make` explicitly define dependencies between input and output files (and recipes to be executed when inputs are updated), whereas framework-driven build technologies like Maven assume conventions are being respected (e.g., input and output files are stored in default locations) unless otherwise configured.

These different paradigms impact the shape that build specifications take. While the CodeGen model can theoretically be applied to any technology, this thesis focuses on

[2] https://seart-ghs.si.usi.ch/
[3] https://incubator.apache.org/

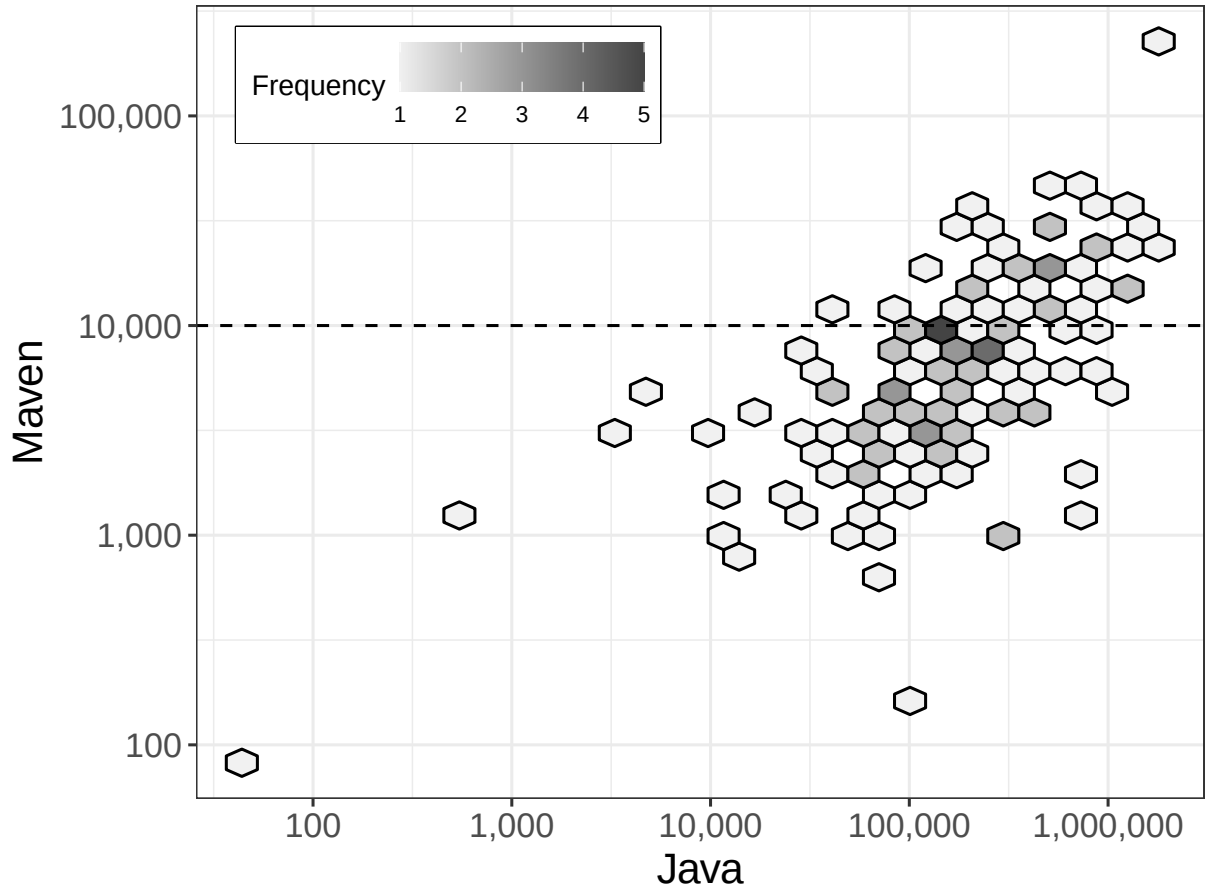


Figure 3.3: The number of Java and Maven non-comment lines in each TLP. I study the 39 TLPs with at least 10,000 lines of Maven code (above the horizontal dashed line).

analysing a single technology to control for the confounding impact that technology will have on our results.

To select the build tool for analysis, I study the prevalence of build tools among the TLPs. I identify the build tools used within each TLP using file name conventions. I identify that 136 of the TLPs (67%) adopt Maven, which is more than double the adoption rate of the next most popular technology. Therefore, I focus our analysis on TLPs that adopt Maven.

Select projects with substantial build code. Not every project requires complex build code. For some projects, the default build behaviour is sufficient. In such cases, Maven's

"convention over configuration" philosophy reduces the build code that must be written and maintained. Modelling simple build systems is not the goal of this thesis. If simple build systems are included in the sample, they may skew the results. Thus, I set out to select sufficiently complex build systems for further analysis.

Figure 3.3 shows the number of non-comment lines of code in the Java source code (X-axis) and Maven build systems (Y-axis) of the 136 candidate TLPs using a hexbin plot. Hexbin plots are scatterplots that control overplotting by grouping data points into hexagon-shaped bins. The darker the shade, the more data points (i.e., projects) within the bin. The plot is logarithmically scaled to lessen the influence of outliers. The plot shows that projects with more source code also tend to have more build code; however, the correlation is only moderate (Spearman's $\rho = 0.65$, $p < 2.2 \times 10^{-16}$).

The dashed line in Figure 3.3 shows that by selecting a threshold of 10,000 lines of Maven code, I not only filter out the projects that have too few build lines to be of interest, but also, the projects that have little Java code as well. After applying this filter, 39 TLPs have been selected for analysis.

Generalised dataset

The filtering process for our Generalised dataset has fewer requirements than the Expert dataset, as it is primarily used for testing our model. This approach better represents the characteristics of real-world projects, which I consider crucial for evaluating the performance of our model.

To do so, I exclude projects that do not use Maven by removing projects that do not include any `pom.xml` file, leaving us with 717 projects. Additionally, I remove projects with fewer than two `pom.xml` files, as I consider these projects lacking substantial build code. After this process, 316 projects are selected.

3.1.3 Validate Maven build files

To filter out syntactically incorrect build files in both datasets, I validate the `pom.xml` files of these projects using `lxml`,[4] a Python library for processing XML files. I parse each `pom.xml` file and validate its syntax using the current schema of Maven.[5] After this validation process, I exclude no `pom.xml` file from the Expert dataset and five syntactically

[4] https://lxml.de/

[5] https://maven.apache.org/xsd/maven-4.0.0.xsd

incorrect `pom.xml` files from the Generalised dataset. Consequently, the Expert dataset includes 9,426 `pom.xml` files that span 39 projects, while the Generalised dataset includes 3,197 `pom.xml` files that span 315 projects.

3.2 Data Preprocessing

I describe the process of mitigating noise from our datasets and engineer prompts alongside their references.

3.2.1 Mitigate noise

I set out to analyse the content of Maven files that specify the software build process. Therefore, I filter out comments and whitespace characters before performing our analyses. More specifically, I apply the following filters:

Remove comments. Fang et al. [23] showed that comments in code can introduce biases in code generation. To counteract this bias, I remove comments from build files before analysis. Comments in Maven files are written using standard markup language style.

Remove empty lines. A key use case I explore is next-line prediction. I create prompts based on each line within the studied build files. Since empty lines do not affect the behaviour of the Maven build process, I filter them out.

Remove extraneous whitespace. I further reduce noise by stripping whitespace from the beginning and end of each line. Removing unnecessary tokens retains only the content that specifies build behaviour.

3.2.2 Engineer prompts

The last step in the Data Processing phase involves parsing the studied `pom.xml` files to create prompts and corresponding references. In the context of this thesis, a prompt is a line or a concatenation of lines of build code from the studied build files. A prompt concatenates the current line and all preceding lines, while the reference is the subsequent line. The

Listing 3.1 depicts the relationship between a prompt and a reference of a sample `pom.xml` file. Given a build file with L lines of build code, where L represents a set

Listing 3.1: Sample `pom.xml` file from the Apache Maven Project. Lines 1–2 depicts a prompt and Line 3 highlights the corresponding reference.

```
1 <project>
2 <modelVersion>4.0.0</modelVersion>
3 <groupId>com.mycompany.app</groupId>
4   <artifactId>my-app</artefactId>
5   <version>1</version>
6 </project>
```

of lines $L = \{l_1, l_2, l_3, \ldots, l_n\}$. At any line, l_i I represent a prompt and reference as; prompt(l_i)=$\{l_1, \ldots, l_{i-1}\}$ and reference(l_i)=l_i. Figure 3.4 shows examples of the prompts used in this study.

Choosing A Model

Below, I explain the criteria for selecting CodeGen, then introduce CodeGen itself, and describe how I apply it to generate build code. The primary reasons for selecting CodeGen are:

Criterion 1: Open-source model. Open-sourced LLMs provide access to their source code, enabling fine-tuning, which is crucial in this thesis, as it allows to optimise the model specifically for generating build code.

Criterion 2: Optimised for code generation. Although most LLMs can generate code and natural language, I prioritise those that specialise in code generation due to the similarities between source code and build code.

Criterion 3: Runnable on commodity hardware. Given that the primary task involves fine-tuning and generating build code for developers working on projects of various sizes, I want to choose an LLM of modest size that can be deployed on commodity developer machines.

I choose CodeGen as our LLM because it satisfies all three criteria. CodeGen meets Criterion 1, as it is an open-source LLM created by Salesforce [57]. As part of an open-source project, it provides access to the source code of the models. CodeGen satisfies Criterion 2 by matching or outperforming the Codex model by OpenAI in code generation

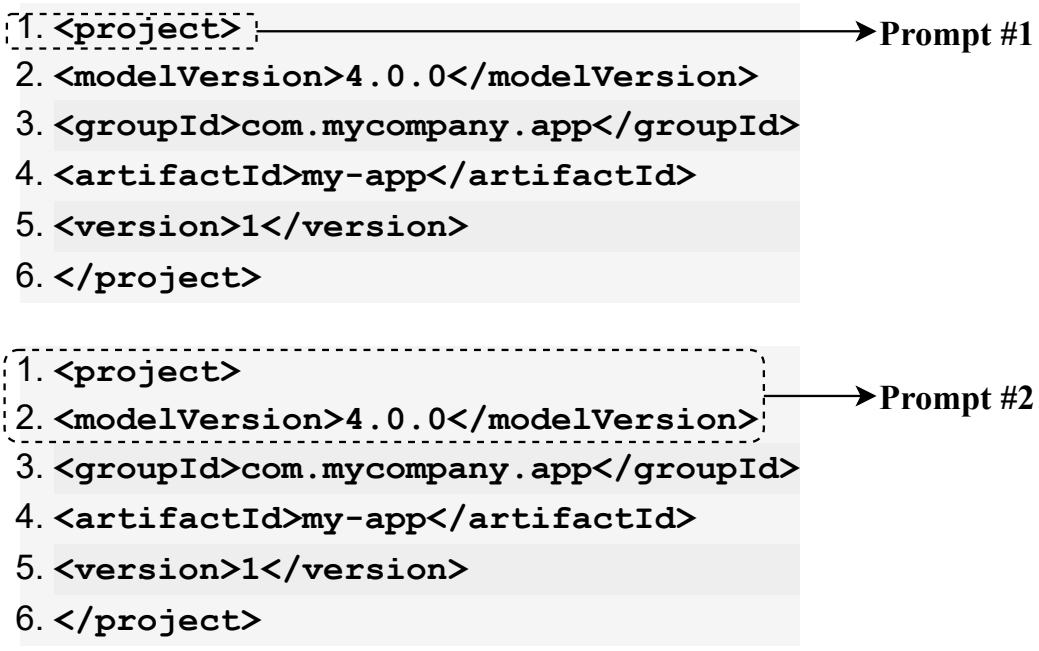


Figure 3.4: Examples of Maven build code prompts.

tasks [57], underscoring its potential in this domain. CodeGen also satisfies Criterion 3, as its smallest model is 797 MB in size; however, even with the smallest model, the experimental runs still consume 1183.89 hours of compute time spent on 3 V100 GPUs, each with 64 GB of available VRAM.

I also consider the Code Llama model from Meta AI; however, the smallest version of their model is 18 times larger than CodeGen, and Code Llama also has 20 fold more parameters than the selected CodeGen model. This would impose even larger computational demands on our equipment and may not yet be practical to fine-tune and generate build code on commodity development machines (Criterion 3).

CodeGen is a transformer-based LLM designed for code synthesis, which is the process of generating code to satisfy the intention of a user [31]. As a transformer-based model, CodeGen leverages transformer capabilities to handle sequential data efficiently, allowing for parallelisation and capturing long-range dependencies in text [74].

The CodeGen model has three variants (a.k.a., pre-training checkpoints). I select the mono variant because it is the pre-trained model that was exposed to the most data, i.e., the Pile dataset [29], GitHub activity data within the BigQuery Dataset,[6] and the BigPython dataset [57]. To minimise the cost of fine-tuning and generating build code, I select the **codegen-350M-mono**[7] model for our study. The model is the smallest one being offered; however, it still has 350 million trainable parameters.

3.3 Build Code Generation

To generate build code, I load our model for causal prediction of the next-line of the prompt and tokenise our prompts, using features that are provided by the transformer module of the Hugging Face API.[8] Given that the model has a maximum token size of 2048, I left-truncate prompts exceeding this limit. After generating the build code, I extract the line by excluding the prompt and selecting the first generated line.

[6] https://console.cloud.google.com/marketplace/product/github/github-repos?project=gold-chassis-225408

[7] https://huggingface.co/Salesforce/codegen-350M-mono

[8] https://huggingface.co/docs/transformers/en/index

Table 3.1: The size of the studied datasets.

Dataset	# projects	# pom.xml files	# prompts
Expert dataset (fine-tune sample)	27	7,211	N/A
Expert dataset (test sample)	12	2,215	282,408
Generalised dataset	315	3,917	287,711

3.4 Data Normalisation

I normalised the generated build code and references to remove expressions that might vary in different build files. Below, I describe the two steps in this normalisation process.

Normalise paths & version numbers. Maven specifications frequently include precise file system paths and version numbers. Since generating these precise details requires plenty of project-specific contexts that LLMs are unlikely to have, I replace each instance with generic placeholders to exclude their impact from our model performance estimates.

Remove angle brackets. I position this thesis to focus on the content within the angle brackets (i.e., <, >) to evaluate the performance of our models. Hence, I remove angle brackets from our build files.

3.5 Analysis

In this phase, I measure the performance of our models using the "Bilingual Evaluation Understudy" (BLEU) metric. This metric is known for assessing the quality of machine-generated text, with a primary focus on machine translation. BLEU has gained widespread recognition in Natural Language Processing (NLP) and machine translation research for evaluating translation system performance [58]. It has also been used as a metric in other studies on code generation [35].

In this thesis, I use the sacreBLEU tool[9] on our generated normalised build code and normalised references to calculate the BLEU score. The BLEU score value ranges between 0–100, with a higher score indicating greater similarity between the generated build code and its references.

Table 3.1 shows an overview of the studied datasets. The Expert dataset is divided into two segments, one for fine-tuning (27 projects with 7,211 `pom.xml` files) and the other

[9] https://github.com/mjpost/sacrebleu

for testing (12 projects with 2,215 `pom.xml` files and 282,408 prompts). The Generalised dataset includes 315 projects with 3,917 `pom.xml` files and 287,711 prompts. The Expert dataset for fine-tuning does not include the number of prompts because I only use them to fine-tune our base model.

Chapter 4

Study Results

This section presents the approach and findings for each research question.

4.1 RQ1. How much fine-tuning is required by CodeGen to effectively infer build specifications?

4.1.1 Approach

To address RQ1, I use the Expert dataset to explore whether and to what extent CodeGen requires fine-tuning to generate build code effectively. From the fine-tuning sample of the Expert dataset, I start by selecting three projects (11% projects) to fine-tune our first model. For each subsequent model, I increase the number of projects used for fine-tuning by three. This process is repeated until our largest fine-tuned model, which uses 24 projects. I randomly select the remaining projects for the validation process, such that the cardinality of the validation set is 10% of the cardinality of the fine-tuning set (rounded up).

In this thesis, we use the `run_clm.py` module from the Hugging Face transformers API library—[1] a widely adopted template for fine-tuning pre-trained models. The module uses an autoregressive training method where the model learns to generate the next token(s) in a sequence based on the preceding tokens [7]. This approach, which focuses solely on the left context of the sequence, is particularly suited for our build code generation tasks because we model our problem on next-line generation. A limitation of this will be using our model to predict build code between lines.

[1] https://huggingface.co/docs/transformers/index

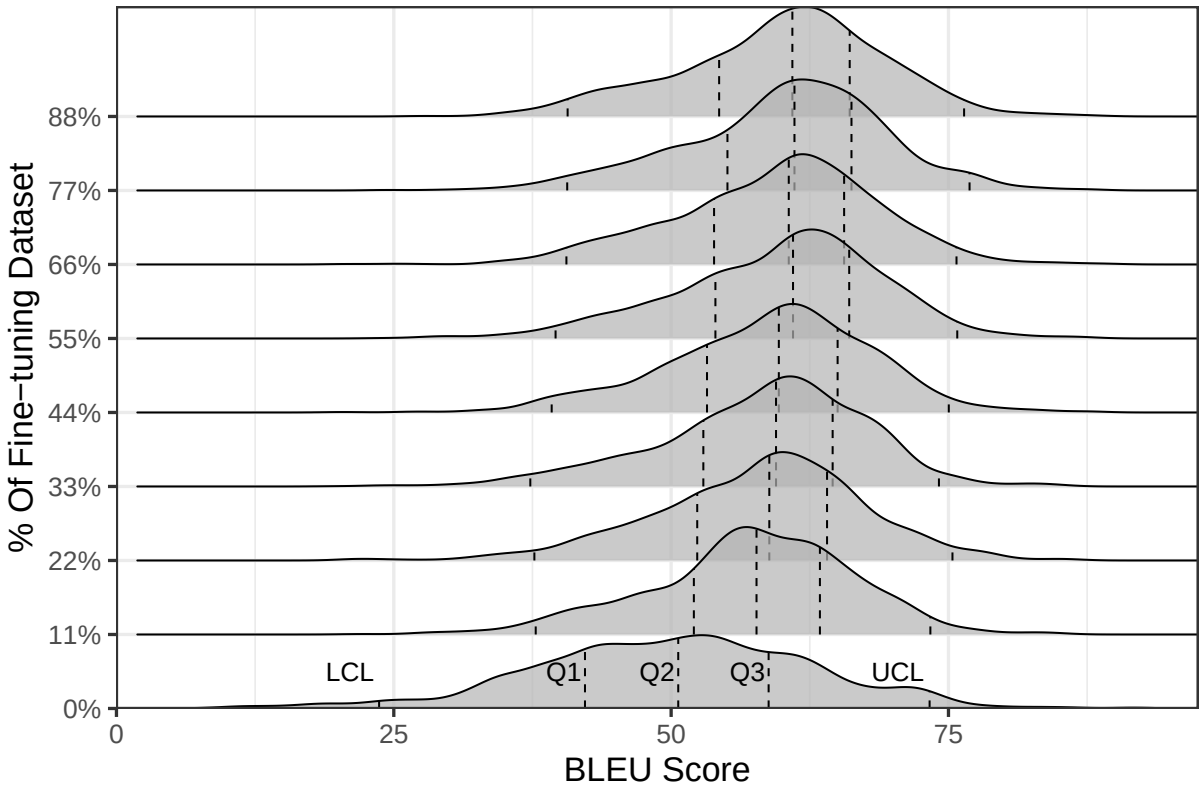


Figure 4.1: BLEU score distributions for our base and fine-tuned models on the test sample of the Expert dataset, varying by the proportion of fine-tuning data. Interquartile ranges (Q1, Q2, and Q3) and confidence intervals (2.5th percentile—LCL and 97.5th percentile—UCL) are highlighted. The Q2 line represents the median BLEU scores, indicating that the most significant performance gain comes from the first 11% of the fine-tuning dataset. Additional data yields diminishing improvements

To evaluate the performance of each model, I generate build code for each prompt (Subsection 3.2.2) from the projects in the testing sample of the Expert dataset. These generated lines of build code are used to create build files, which I compare to their corresponding reference files using the BLEU metric. Given the presence of outliers in our results, I use the median BLEU score as a measure of the central tendency of these scores. The median BLEU scores across our eight fine-tuned models range from 57.69 to 61.11, all representing improvements over our base model, which has a median BLEU score of 50.64.

4.1.2 Results

Observation 1: **Fine-tuning on a small portion of the fine-tuning dataset provides the largest improvement in performance for build code generation.** Figure 4.1 shows the distribution of the BLEU score for each build file within the testing sample of the Expert dataset. The figure highlights a 13.93% performance increase from the base model to the model fine-tuned with 11% of the fine-tuning sample of the Expert dataset, with subsequent iterations showing diminishing returns. This quick plateau in performance was not expected, as we expected subsequent fine-tuning to match or exceed the performance increase of the model trained with 11% of the fine-tuning dataset.

Observation 2: **Across all models, fine-tuned models yield more predictable outcomes.** As shown in Figure 4.1, there is a noticeable reduction in the breadth of BLEU scores for the fine-tuned models compared to the base model. To show this, I use an interquartile range ($IQR = Q3 - Q1$) and 95% confidence interval ($CI = UCL - LCL$). The fine-tuned models show improvements, with reductions in the interquartile range of 32.49% to 32.82% (11.18 to 11.77 BLEU score points) and reductions in the confidence interval of 24.18% to 29.04% (35.16 to 37.58 BLEU score points). These reductions indicate greater stability in model performance for the fine-tuned models compared to the base model. This improvement reduces variability in output quality, ensuring that the model produces more reliable and predictable results.

> Fine-tuning with a small proportion of the available data provides most of the benefits, but further fine-tuning yields diminishing returns. Nonetheless, fine-tuning enhances performance and reduces uncertainty, leading to more predictable and reliable results when generating build code.

4.2 RQ2. How well does the fine-tuned CodeGen perform on an unseen dataset?

4.2.1 Approach

To address RQ2, I first select the model fine-tuned with 11% of the fine-tuning sample of the Expert dataset, which demonstrates the largest performance improvement, and use both datasets to validate the generalisability of our model on unseen data using the concept of optimism [33]. Optimism is the difference between training and testing accuracy. In this

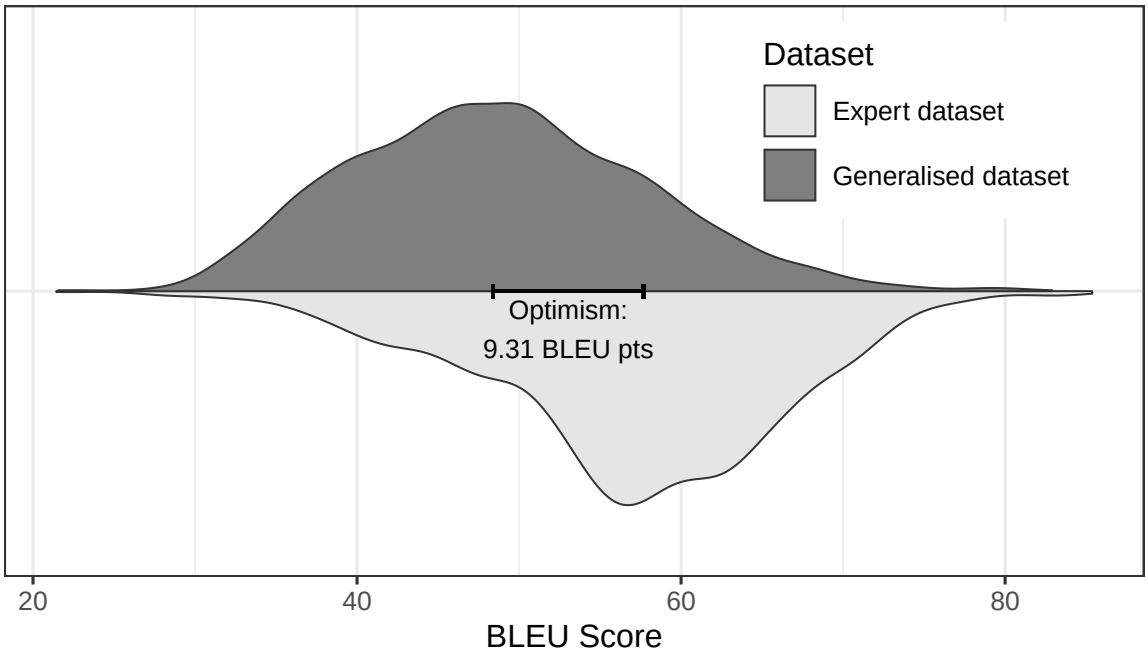


Figure 4.2: Comparative performance of our fine-tuned model on both datasets. The dotted lines represent the median BLEU score, and the line between them represents the optimism of our fine-tuned model on both datasets, highlighting an optimism score of 9.31 BLEU score points.

thesis, I evaluate the accuracy using the BLEU metric on both datasets. A lower optimism score suggests better generalisation and indicates that the model is less prone to overfitting [81]. I follow a similar approach as in RQ1 to evaluate the performance of each model. I calculate the BLEU score for each file in both the test sample of the Expert dataset and the Generalised dataset. Then, I compute optimism as the difference in the median BLEU scores of both datasets:

$$Optimism = BLEU_{Expert\ dataset} - BLEU_{Generalised\ dataset}$$

Additionally, to understand how fine-tuning impacts the performance on unseen data, I study the improvement at the granularity of lines of build code, assuming that the performance might not improve uniformly. I compare the build code generated by our base and fine-tuned models in the Generalised dataset, allowing me to draw comparisons between the distributions of BLEU scores of the base and fine-tuned models.

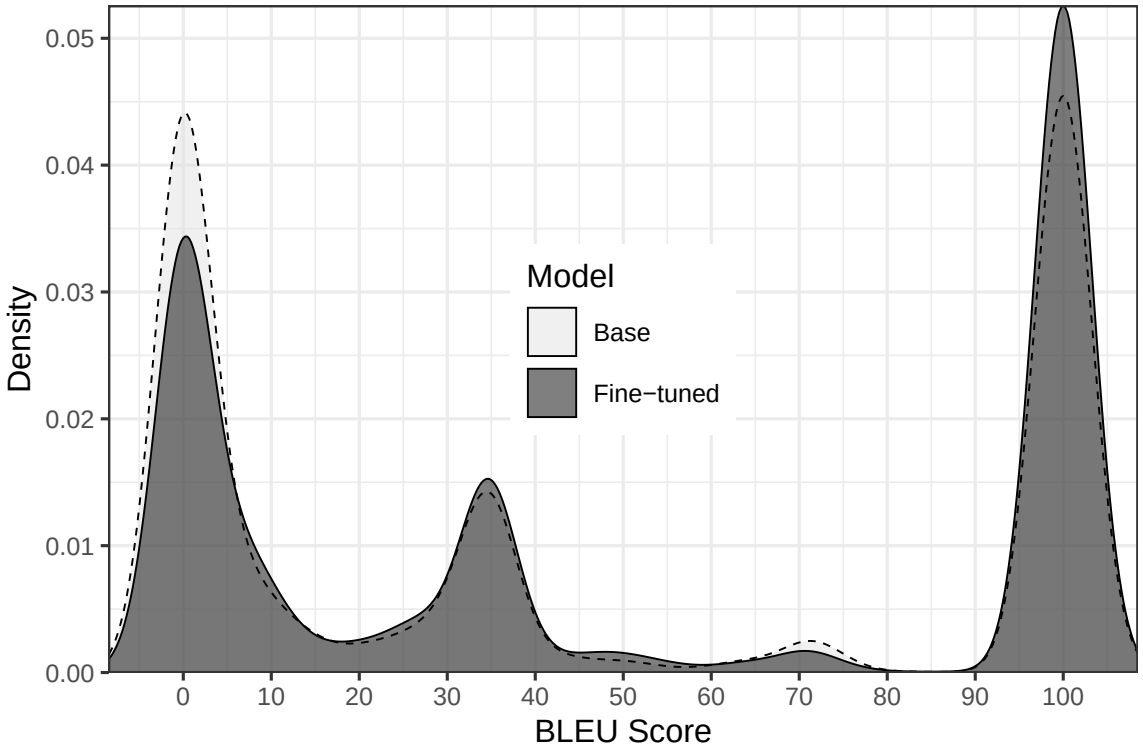


Figure 4.3: The BLEU score distribution of generated build code from our base model and fine-tuned model on the Generalised dataset. The main difference between both models occurs at the two extremes of the plot.

4.2.2 Results

Observation 3: **Model performance decreases by 16.13% when it is applied to the Generalised dataset.** Figure 4.2 illustrates the distribution of the BLEU scores, with median BLEU scores of 57.69 and 48.38 for Expert and Generalised datasets, respectively. The optimism score is 9.31, highlighting a 16.13% drop in performance on the Generalised dataset. This decrease underscores the difference between the two datasets, which I suspect is attributed to differences in the type of content that they store. For example, I observe that there are no contributor tags in the Generalised dataset.

Observation 4: **Fine-tuning decreases entirely inaccurate suggestions by 24.18% and increases completely accurate build suggestions by 15.02%.** Figure 4.3 shows a reduction as the curve approaches a BLEU score of 0 (i.e., completely incorrect

build code generation) in the fine-tuned model, indicating fewer inaccurate build suggestions are generated when the model is applied to our Generalised dataset. In contrast, the area increases as the curve approaches a BLEU score of 100, reflecting more accurate build suggestions. This change underscores the performance improvements achieved through the fine-tuning process. The results reveal a multi-modal distribution with three peaks.The central peak remains relatively consistent between the two models, while the extreme peaks, representing BLEU scores of 0 and 100, appear at minimum and maximum points of the spectrum. This pattern highlights that both models mostly suggest build code lines that are either completely accurate (i.e., a BLEU score of 100) or completely inaccurate (i.e., a BLEU score of 0). Comparing the fine-tuned and base models, we observe an increase in completely accurate build code generation by 5.56 percentage points and a decrease in completely inaccurate build code generation by 8.29 percentage points.

The fine-tuned model improves accuracy in generating build code, and it achieves sufficiently general performance to serve as a strong foundation for researchers to build upon.

Chapter 5

Follow-Up Analysis

In Observations 3 and 4, I note that overall performance declines when our fine-tuned model is applied to the Generalised dataset. Upon closer inspection, I observe that the models often generate lines that are completely accurate (i.e., BLEU score of 100) or completely inaccurate (i.e., BLEU score of 0).

I hypothesise that while characteristics are shared between both datasets, there are also characteristics that are distinct between the datasets. To understand the characteristics of the datasets, I analyse the frequency at which tag types appear in build files of projects, and compare those distributions between the two datasets.

I extract tags from build files using the `lxml`[1] library. Inside the build file, the root element is the `project` tag, which contains information related to the build system of that project. I then extract the first-level child tags within the `project` tag to obtain a list of child elements (i.e., tags) within each build file. Next, to analyse the size of each tag in the build file, I count the number of lines within each tag and then calculate its proportion relative to the total number of lines in the build file.

To understand which tags are associated with accurate or inaccurate predictions, I inspect samples of the results from RQ1 and RQ2. I stratify the results along the type of tag, which may be *functional*, i.e., tags that affect the outcome of the build, or and *metadata*, i.e., tags that do not affect the outcome of the build.

I reference the Maven project descriptor[2] to determine whether each tag type is functional or metadata. Tags that, when modified, impact the build process or outcome are

[1] https://lxml.de/

[2] https://maven.apache.org/ref/3.9.6/maven-model/maven.html

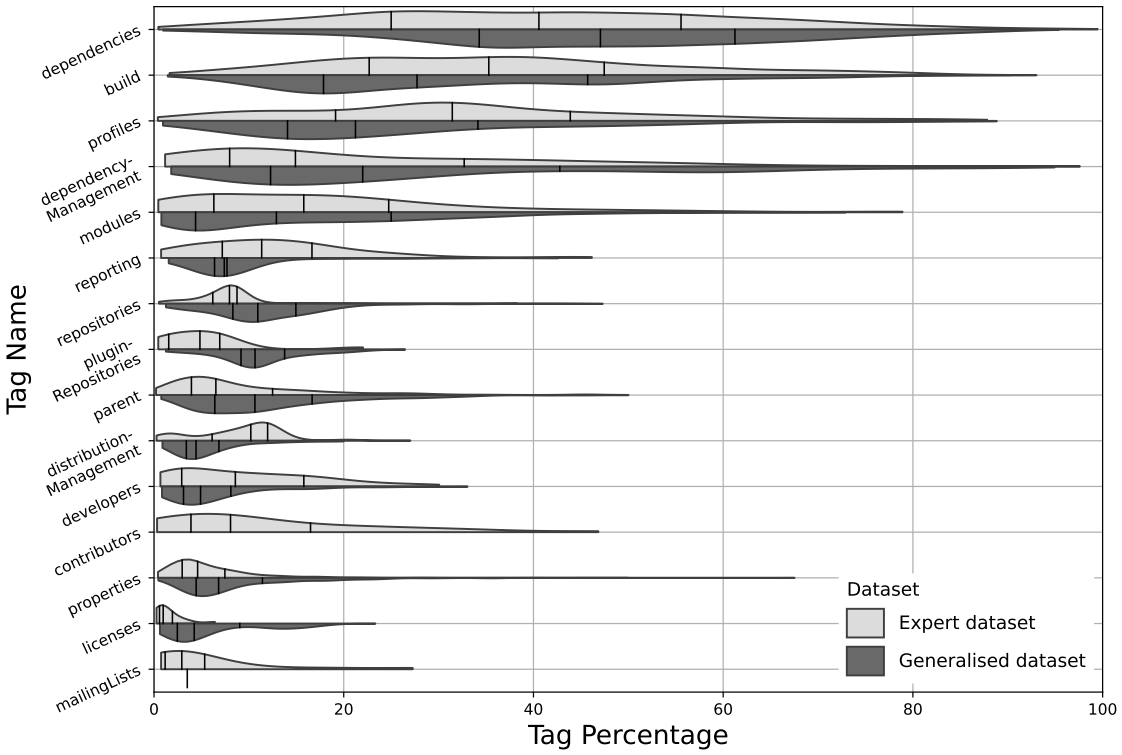


Figure 5.1: Compares tag percentages of build files between the test sample of our Expert dataset and the Generalised dataset. The violin plots illustrate the distribution of the percentage of content each tag occupies across build files. Tags are arranged from top to bottom based on decreasing median percentages, with the highest median at the top.

categorised as functional; otherwise, they are categorised as metadata. For example, the `modelVersion` tag defines how Maven parses build files and thus directly affects the build, making it a functional tag. In contrast, the `name` tag identifies the project name but does not alter the build outcome, even if it contains an incorrect setting, so I classify it as metadata. Tables 5.1 and 5.2 describe the encountered functional and metadata tags, respectively. Figure 5.3 shows examples of functional and metadata tags.

Observation 5: **Contributor information does not exist in the Generalised dataset.** Figure 5.1 shows the overall prevalence of different tags across both datasets.

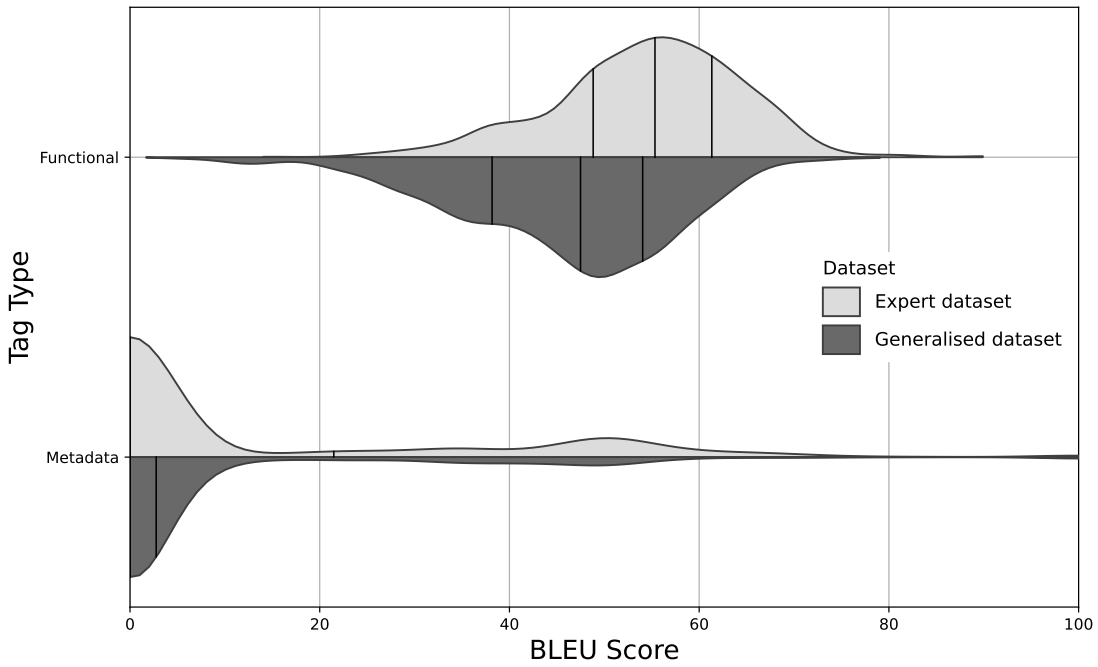


Figure 5.2: The BLEU score distribution of our fine-tuned model across both datasets, categorised by tag type. The fine-tuned model exhibits a higher average BLEU in generating functional tags compared to metadata tags.

There are 28 tags (96%) out of 29 tags that appear in both datasets, whereas `contributors` is exclusive to the test sample of the Expert dataset. Figure 5.1 shows that the `contributors` tag is absent, and the `mailingLists` tag appears only once in the Generalised dataset. These metadata tags provide details about contributors and communication methods. Consequently, their absence or limited occurrence does not influence the functionality of the build system. This discrepancy highlights different objectives between the Expert and Generalised datasets.

Observation 6: **The fine-tuned model underperforms in generating metadata content.** Figure 5.2 shows a concentration of BLEU scores at or near to zero for metadata tags in both datasets. In the Expert dataset (test sample) and Generalised dataset, our median BLEU score for metadata tags is Zero. Conversely, our model performs substantially better in suggesting functional tags, achieving median BLEU scores of 55.34 and

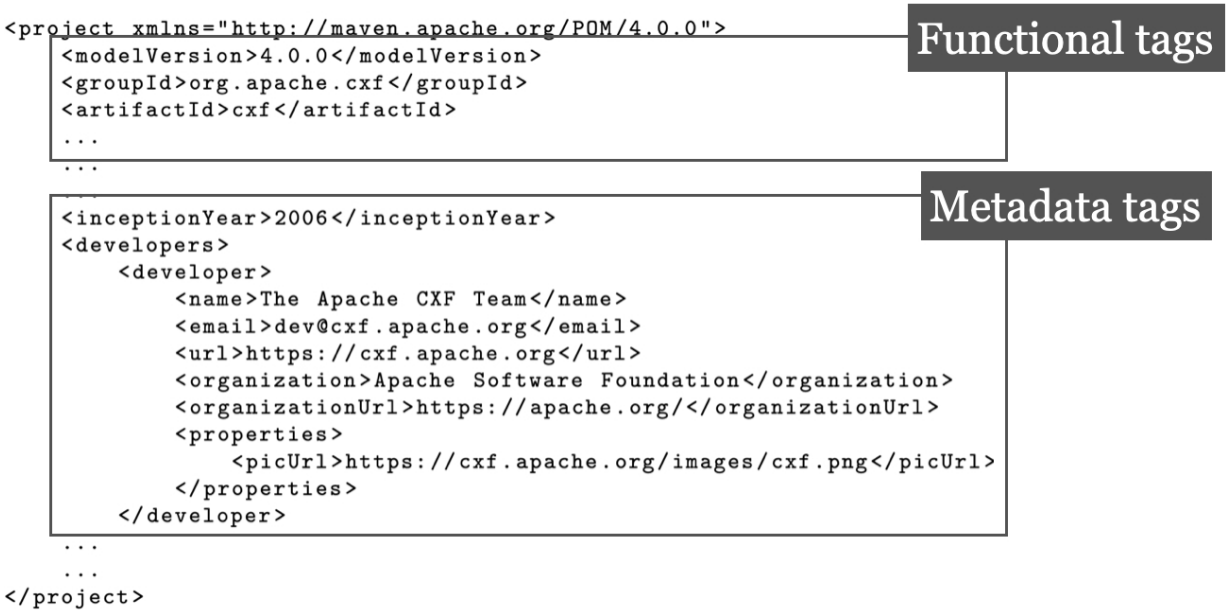


Figure 5.3: Snippet of Maven build code showing examples of functional tags and metadata tags.

47.49 in the Expert and Generalised datasets, respectively.

Our fine-tuned model performs substantially better in generating functional tags than metadata tags; furthermore, software developers can use the fine-tuned model to generate functional build code with higher confidence than metadata tags.

Table 5.1: A summary of the Maven functional tags within our datasets.

Tag Name	Description
`artifactId`	It specifies the unique identifier for an artefact within a group, directly influencing the name of the generated artefact. It determines the artefact's base name in the output.
`build`	It directly influences the build process and final artefact through plugin executions and settings.
`dependencies`	It specifies the project's direct dependencies, including external libraries in the build, affecting compilation and runtime behaviour.
`dependency-Management`	It provides dependency inheritance to child projects, controlling versions of dependencies and affecting the consistency and outcome of child builds.
`groupId`	It defines the unique identifier for the project's group, crucial for accurate dependency resolution and directly affecting the build process.
`modelVersion`	Specifies the Project Object Model (POM) version that the current POM follows, guiding Maven in interpreting its structure and content.
`modules`	It specifies the modules in a multi-module project, impacting the overall build structure and outcome.
`packaging`	It specifies the packaging type of the project (e.g., jar, war), defining the type of artefact produced and influencing the build output.
`parent`	It allows the project to inherit dependencies and plugins from a parent POM, potentially altering the build outcome.
`plugin-Repositories`	Defines additional repositories for plugins, affecting the build if required plugins are retrieved.
`prerequisites`	Specifies the minimum Maven version required to build the project, ensuring the build process utilizes appropriate Maven features.
`profiles`	Allows customization of build settings, altering the build outcome when profiles are activated and enabling different configurations.
`properties`	Defines custom properties used throughout the POM, influencing the build process if configurations use these properties.
`repositories`	Defines additional repositories for dependencies, impacting the build by providing access to dependencies not found in default repositories.
`version`	Determines an artefact's version, influencing dependency management and build reproducibility.

Table 5.2: A summary of the Maven metadata tags within our datasets.

Tag Name	Description
`ciManagement`	Information about the continuous integration system. It is provided for informational purposes.
`contributors`	Lists the contributors involved in the project. It serves as acknowledgement in generated sites.
`description`	It briefly describes the project and serves informational purposes in the documentation.
`developers`	Lists the developers involved in the project. It is used in documentation and metadata.
`distribution-Management`	It specifies where to deploy the project's artefacts. It does not affect the build output but determines deployment destinations post-build.
`inceptionYear`	Indicates the year the project was started. It is used in generated reports.
`issue-Management`	Information about the issue tracking system. It is included in the documentation for user reference.
`licenses`	Provides information about the project licenses. It is included in the metadata for legal compliance.
`mailingLists`	Information about mailing lists related to the project. It is used in generated documentation.
`name`	It specifies the name of the project outcome used for display in the generated documentation.
`organization`	Information about the organization behind the project. It is included for informational purposes.
`reporting`	Specifies the reporting settings, usually for site generation. It affects the generation of reports.
`scm`	Details of source control management for the project. It may affect the build if certain plugins use SCM information; otherwise, it is mainly for metadata.
`url`	It specifies the URL of the project's website and is included in the metadata for reference.

Chapter 6

Threats to Validity

6.1 Construct Validity

Construct validity refers to how accurately the measurements in our study represent the theoretical constructs they aim to assess. Given the similarities between source code and build code, I considered using CodeBLEU [65]; however, CodeBLEU does not yet support build specifications, and developing such support is beyond the scope of this research. Nevertheless, I use BLEU consistently as our evaluation metric, as it is an established method in this field, e.g., [42, 53, 61, 79]. Conducting human evaluations would be the next step to evaluate the models; we elaborate further on this in the section A.3.

6.2 Internal Validity

Internal validity refers to how our study accurately establishes cause-and-effect relationships between variables, minimizing the influence of alternative explanations. In this study, I collect Maven build files from GitHub to create the Generalised dataset; however, there is a risk that the build files might be syntactically incorrect, which could affect the BLEU evaluation. To mitigate this, I validate all Maven (see section 3.1.3) build specifications against the official schema of Maven `pom.xml` to verify their correctness.

6.3 External Validity

External validity refers to how well the results of our study generalise. In our research, I fine-tune and test our model using our Expert dataset; however, this approach risks creating a model that performs well on this dataset but may not generalise to other contexts. To mitigate potential bias, I calculate the optimism score of our model using a Generalised dataset to assess its performance on a different dataset. Like the Expert dataset, this dataset consists of real-world Maven projects. While our study focuses on the Maven build system, future research could expand this approach to other build systems like CMake, Bazel, etc.

Chapter 7

Conclusion

Maintaining build systems is an essential task in the software development process. With the emergence of LLMs being used in source-code maintenance tasks, I explore their potential to improve build system maintenance by applying an LLM to generate build code. I conduct an empirical study of the CodeGen model when it is applied to the Maven build system. The subjects include an Expert dataset hosted by ASF and a Generalised dataset sample from GitHub. The Expert dataset is used for fine-tuning and testing, and the Generalised dataset is used strictly for testing.

Our results show that LLMs can effectively maintain Maven build system specifications. Fine-tuning the CodeGen model with a small subset of the Expert dataset (i.e., 11%) yields the largest increase in performance. Furthermore, when applied to the Generalised dataset, the fine-tuned model retains 83.86% of its performance, suggesting it has not been overfit. Lastly, I categorise the content of our build files into functional and metadata content based on their enclosing tags. Our model performs better at generating functional build content than metadata content. This suggests that while the model is proficient in suggesting functional build code, which is essential to run the build system, it exhibits limitations in suggesting non-functional (supplementary) build code. Therefore, when using the model to maintain Maven build specifications, developers can expect higher accuracy in generating functional content, while metadata content may require additional review. This may be because metadata content tends to be project-specific or intra-company-specific, often reflecting unique requirements or conventions not widely represented in functional build content. This makes it more challenging for LLMs to generate metadata content reliably. This highlights the need to fine-tune the model to project-specific or intra-company-specific data.

References

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering*, 47(12):2740–2754, 2021. doi: 10.1109/TSE.2020.2967380.

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 47(3): 448–463, 2021. doi: 10.1109/TSE.2019.2897300.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Bram Adams. Co-evolution of source code and the build system. In *2009 IEEE International Conference on Software Maintenance*, pages 461–464. IEEE, 2009.

[5] Bram Adams, Herman Tromp, Kris de Schutter, and Wolfgang de Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123, 2007. doi: 10.1109/ICSM.2007.4362624.

[6] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the EASST*, 8, 2008.

[7] Dang Hoang Anh, Dinh-Truong Do, Vu Tran, and Nguyen Le Minh. The impact of large language modeling on natural language processing in legal texts: A comprehensive survey. In *2023 15th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–7, 2023. doi: 10.1109/KSE59128.2023.10299488.

[8] Jelmer P. Borst, Niels A. Taatgen, and Hedderik van Rijn. What makes interruptions disruptive? a process-model account of the effects of the problem state bottleneck on

task interruption and resumption. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, page 2971–2980, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450331456. doi: 10.1145/2702123.2702156. URL https://doi.org/10.1145/2702123.2702156.

[9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair, 2024.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[11] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. Build system with lazy retrieval for java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 643–654, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950358. URL https://doi.org/10.1145/2950290.2950358.

[12] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024. ISSN 2157-6904. doi: 10.1145/3641289. URL https://doi.org/10.1145/3641289.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra,

Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[14] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. PyMT5: multi-mode translation of natural language and python code with transformers. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.728. URL https://aclanthology.org/2020.emnlp-main.728.

[15] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for msr studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564, 2021. doi: 10.1109/MSR52588.2021.00074.

[16] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[18] Marc Dillmann, Julien Siebert, and Adam Trendowicz. Evaluation of large language models for assessing code maintainability. *arXiv preprint arXiv:2401.12714*, 2024.

[19] Saket Dingliwal, Ashish Shenoy, Sravan Bodapati, Ankur Gandhe, Ravi Teja Gadde, and Katrin Kirchhoff. Prompt tuning gpt-2 language model for parameter-efficient domain adaptation of asr systems, 2022.

[20] Eelco Dolstra. Integrating software construction and software deployment. In *International Workshop on Software Configuration Management*, pages 102–117. Springer, 2001.

[21] Bradley Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983. ISSN 01621459. URL http://www.jstor.org/stable/2288636.

[22] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's

distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 11–20, 2016.

[23] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do llms really do their job?, 2024.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[25] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[26] Maik Frye, Johannes Mohren, and Robert H. Schmitt. Benchmarking of data preprocessing methods for machine learning-applications in production. *Procedia CIRP*, 104:50–55, 2021. ISSN 2212-8271. doi: https://doi.org/10.1016/j.procir.2021.11.009. URL https://www.sciencedirect.com/science/article/pii/S2212827121009070. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0.

[27] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering*, 48(6):2040–2052, 2022. doi: 10.1109/TSE.2020.3048335.

[28] Jianfeng Gao and Chin-Yew Lin. Introduction to the special issue on statistical language modeling, 2004.

[29] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

[30] Jindong Gu, Zhen Han, Shuo Chen, Ahmad Beirami, Bailan He, Gengyuan Zhang, Ruotong Liao, Yao Qin, Volker Tresp, and Philip Torr. A systematic survey of prompt engineering on vision-language foundation models, 2023.

[31] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[32] Ryan Hardt and Ethan V Munson. Ant build maintenance with formiga. In *2013 1st International Workshop on Release Engineering (RELENG)*, pages 13–16. IEEE, 2013.

[33] Frank E Harrell et al. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*, volume 608. Springer, 2001.

[34] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089, 2018. doi: 10.1145/3180155.3180181.

[35] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025*, 2018.

[36] Lorin Hochstein and Yang Jiao. The cost of the build tax in scientific software. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 384–387, 2011. doi: 10.1109/ESEM.2011.54.

[37] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 13–25, 2020.

[38] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

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

[40] Stefan Kombrink, Tomas Mikolov, Martin Karafiát, and Lukás Burget. Recurrent neural network based language modeling in meeting recognition. In *Interspeech*, volume 11, pages 2877–2880, 2011.

[41] G Kumfert and T Epperly. Software in the doe: The hidden overhead of"the build". 2 2002. doi: 10.2172/15005938. URL https://www.osti.gov/biblio/15005938.

[42] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1057. URL https://aclanthology.org/P16-1057.

[43] Mingxing Liu, Junfeng Wang, Tao Lin, Quan Ma, Zhiyang Fang, and Yanqun Wu. An empirical study of the code generation of safety-critical software using llms. *Applied Sciences*, 14(3):1046, 2024.

[44] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), January 2023. ISSN 0360-0300. doi: 10.1145/3560815. URL https://doi.org/10.1145/3560815.

[45] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting Build Co-Changes with Source Code Change and Commit Categories. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 541–551, 2016.

[46] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically Repairing Dependency-Related Build Breakage. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page 106–117, 2018.

[47] Vincent Massol and Timothy M O'Brien. *Maven: A Developer's Notebook: A Developer's Notebook*. " O'Reilly Media, Inc.", 2005.

[48] Shane McIntosh. Build system maintenance. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1167–1169, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1986031. URL https://doi.org/10.1145/1985793.1986031.

[49] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 141–150, 2011. doi: 10.1145/1985793.1985813.

[50] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.

[51] Shane Mcintosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Mining co-change information to understand when build changes are necessary. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 241–250, 2014. doi: 10.1109/ICSME.2014.46.

[52] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering*, 20(6):1587–1633, 2015.

[53] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In Greg Kondrak and Taro Watanabe, editors, *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing. URL https://aclanthology.org/I17-2053.

[54] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 881–881. IEEE Computer Society, 2024.

[55] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. Build system issues in multilanguage software. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 140–149, 2012. doi: 10.1109/ICSM.2012.6405265.

[56] J Nguyen Xuan, B Copy, and M Dönszelmann. A C/C++ Build System Based On Maven for the LHC Controls System. *Conf. Proc.*, C111010:WEPKS026, 2011. URL https://cds.cern.ch/record/1392650.

[57] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[58] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://doi.org/10.3115/1073083.1073135.

[59] Matt Post. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Belgium, Brussels, October 2018. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W18-6319.

[60] Mridula Prakash. Software build automation tools a comparative study between maven, gradle, bazel and ant. *International Journal of Software Engineering & Applications. DOI: https://doi. org/10.5121/ijsea*, 2022.

[61] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://aclanthology.org/P17-1105.

[62] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[63] Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, and Mathieu Acher. On the benefits and limits of incremental build of software configurations: an exploratory study. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1584–1596, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510190. URL https://doi.org/10.1145/3510003.3510190.

[64] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.

[65] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

[66] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 724–734, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568255. URL https://doi.org/10.1145/2568225.2568255.

[67] Elior Sulem, Omri Abend, and Ari Rappoport. Bleu is not suitable for the evaluation of text simplification. *arXiv preprint arXiv:1810.05995*, 2018.

[68] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural cnn decoder for code generation. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. ISBN 978-1-57735-809-1. doi: 10.1609/aaai.v33i01.33017055. URL https://doi.org/10.1609/aaai.v33i01.33017055.

[69] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 366–369, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312042. doi: 10.1145/2351676.2351749. URL https://doi.org/10.1145/2351676.2351749.

[70] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. Automated decomposition of build targets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 123–133, 2015. doi: 10.1109/ICSE.2015.34.

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[72] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. *Code generation as a dual task of code summarization*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[73] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[74] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[75] Xin Xia, Xiaozhen Zhou, David Lo, Xiaoqiong Zhao, and Ye Wang. An empirical study of bugs in software build system. *IEICE TRANSACTIONS on Information and Systems*, 97(7):1769–1780, 2014.

[76] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E. Hassan. Cross-project build co-change prediction. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 311–320, 2015. doi: 10.1109/SANER.2015.7081841.

[77] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R. Cogo, and Zhen Ming (Jack) Jiang. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 297–306, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392266. doi: 10.1145/3510457.3513050. URL https://doi.org/10.1145/3510457.3513050.

[78] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

[79] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, Vancouver, Canada, July 2017. URL https://arxiv.org/abs/1704.01696.

[80] Pengcheng Yin and Graham Neubig. Reranking for neural semantic parsing. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1447. URL https://aclanthology.org/P19-1447.

[81] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2):022022, feb 2019. doi: 10.1088/1742-6596/1168/2/022022. URL https://dx.doi.org/10.1088/1742-6596/1168/2/022022.

[82] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 334–344. IEEE Press, 2017. ISBN 9781538615447. doi: 10.1109/MSR.2017.2. URL https://doi.org/10.1109/MSR.2017.2.

[83] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

APPENDICES

Appendix A

A.1 Large Language Models

Large Language Models (LLMs) are advanced computational systems designed to understand and generate human language, distinguished by their large parameter sizes and exceptional capabilities [12]. LLMs use the transformer architecture (as shown in Figure A.1) and the attention mechanism introduced by Vaswani et al. [71], which has significantly enhanced the learning capabilities of LLMs to manage long-range dependencies in natural language texts. This design represents a significant shift from earlier models, like Recurrent Neural Networks (RNN), allowing parallel processing and more direct interactions between different sequence parts.

LLMs are typically pre-trained on large amounts of data to acquire a huge knowledge base. During this phase, which is self-supervised, these models learn the intricacies of the training data structure without explicit human-labeled examples. After pre-training, LLMs can be fine-tuned to solve specialized tasks through a supervised training method, where smaller labelled datasets guide the model to generate context-specific or domain-specific outputs. This combination of self-supervised learning and targeted fine-tuning is key to the versatility and effectiveness of modern LLMs.

A.2 Prompt Engineering

A prompt is a set of instructions given to an LLM to program it, customize its behaviour, or enhance and refine its generation capabilities [44]. Such prompts range from simple questions to more detailed specifications (see Figure A.2). By carefully crafting the prompt, one can greatly influence the quality of the generation of the LLM.

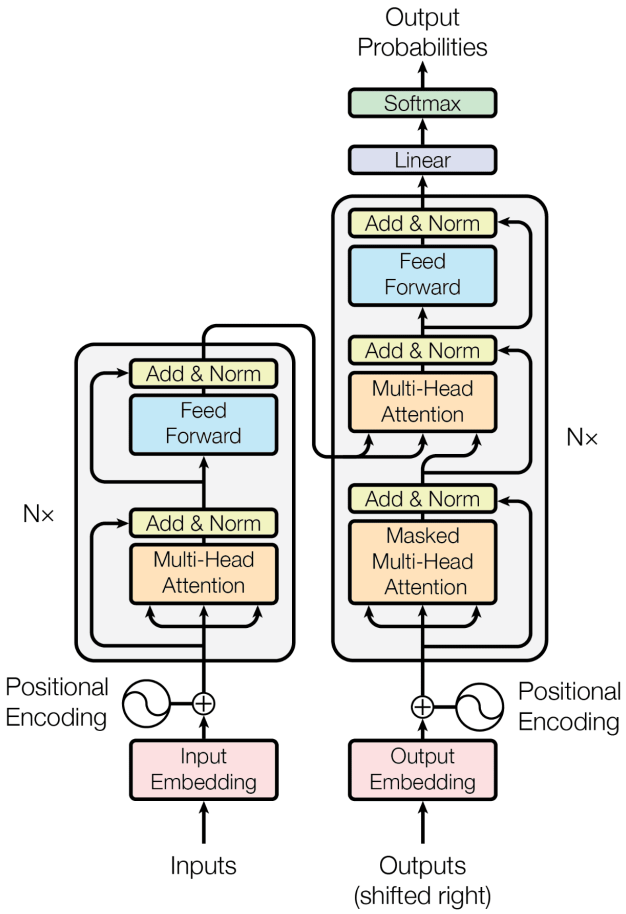


Figure A.1: The transformer model architecture, from "Attention Is All You Need" by Vaswani et al. [71].

Prompt engineering is the method of programming LLMs by crafting and using prompts [73]. Prompt engineering enables practitioners to guide the output of an LLM through systematic experimentation and iteration, often without retraining or modifying the LLM. This approach leverages the knowledge acquired during pre-training or fine-tuning while allowing on-the-fly adjustments to address diverse tasks such as text generation or summarisation.

```
"Explain photosynthesis
in simple terms."
```

```
"Write a 500-word essay describing the
process of photosynthesis at a molecular
level. Include details about the roles of
chlorophyll, the light-dependent and light
-independent (Calvin) cycles, and glucose
production."
```

Figure A.2: Example of prompts — simple prompt (left) and complex prompt (right).

A.3 User-study Evaluation

A user study could be conducted with software practitioners, software developers, DevOps engineers, and researchers who regularly write or maintain build code. The fine-tuned model can be viewed as a specialized version of GitHub Copilot, tailored specifically to generating build code. During the evaluation, participants would be asked to complete tasks such as creating a build script from scratch or finishing a partially written script, all with and without the help of this fine-tuned model.

The primary focus of the evaluation would be to measure how often the next-line generated by the fine-tuned model is accepted or rejected. Additionally, it would be valuable to capture instances where generations are accepted but subsequently modified to distinguish between minor and major edits. This approach would shed light on the accuracy of the generations of the fine-tuned model and the nature of user interactions with its generated build code.