MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

CODE GENERATION FOR MOBILE LANGUAGES USING LARGE LANGUAGE MODELS.

GENERARE DE COD PENTRU LIMBAJE MOBILE UTILIZÂND MODELE DE LIMBAJ.

Master's graduate: **Mircea-Serban Vasiliniuc**

Supervisor: **Dr. Prof. Adrian Groza**

July, 2023

Abstract

A recent research trend aims to discover statistical patterns in huge codebases that contain billions of lines of source code. These codebases reveal implicit knowledge on how to write clear and well-organized code, which can help create high-quality code that is both easy to read and debug. Together with the evolution of the capabilities of LLMs and the constant demand for software development, a focus was created to research how to use trained models as a tool to augment code writing and quality assurance in this field. Most of the research corpus revolves around Python, Java, and Javascript since these are the most used languages in academic circles and in software development communities. This study turns the focus on the programming languages used in native mobile application development, it explores the Code Intelligence capabilities, details the methodology of creating datasets for Swift and Kotlin, outlines the creation process of a GPT-2 model trained on the results datasets, presents a Case Study for applicability of these models in industry-specific tasks like technical onboarding and technical stack switch and finally discusses implications of these type of tools on the future of the industry.

CCS Concepts BigCode, Machine Learning (ML), Natural Language Processing (NLP), Large Language Models (LLM), Mobile Development, Swift, Kotlin, Software Development Industry, Code Generation, Text-to-Code, Code intelligence, Case Study.

Contents

Chapter 1

Introduction

The study is positioned at the intersection between BigCode, Large Language Models, and the requirements of a Mobile Software Development department. LLMs for Code, or Code Generation Models, are deep neural networks that are trained on large collections of code, mainly from open-source repositories. These types of models have achieved important results in several software engineering tasks like documentation generation, unit test generation, code generation, identification of issues and even generating full functional programs based solely on natural language descriptions.

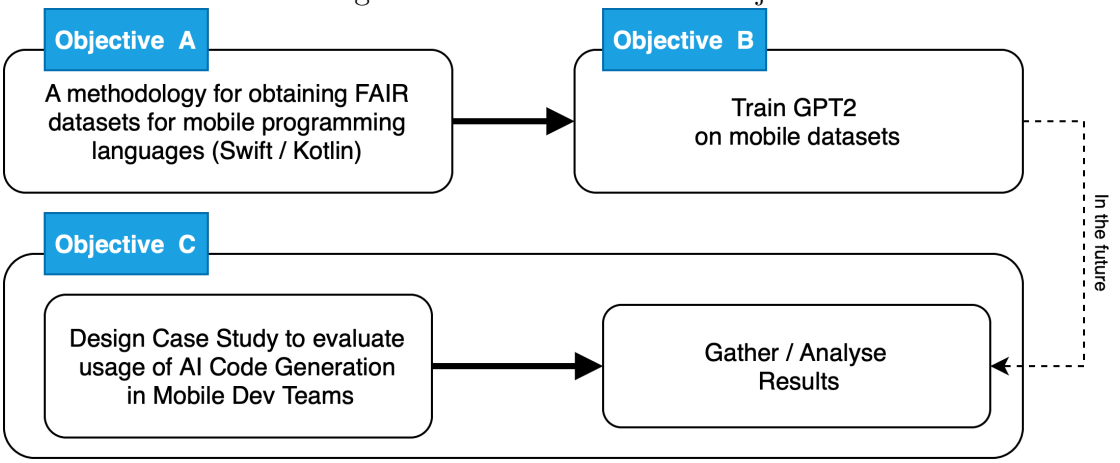
The paper will present the current state of AI-Assisted Software development, focusing on the Code Intelligence concept together with associated tasks, dedicated models, datasets, and benchmarks. It will exploit the lack of support for existing datasets for mobile languages and present the methodology we used to create public datasets for Swift and Kotlin languages that are published following FAIR principles. We go thru the process of training a GPT-2 model with the resulting datasets in an attempt to provide code generation specific to mobile development languages. In the final chapter, we present an extensive case study involving 16 participants from a software development department designed to understand the impact of using LLM trained for code generation in specific phases of the team, more specifically, technical onboarding and technical stack switch.

According to the Stack Overflow 2023 survey published in May of 2023, [1] the percentage of users interested in Swift programming language holds 4.65% and for Kotlin is 9.06%. The number of actual active mobile developers is 3.4% See Figure 1.2. At the time of writing the paper, there were 89,132 questions tagged with [Kotlin] 54,082 questions tagged with [Kotlin][Android] and 328,132 questions tagged with [Swift], and 190,923 questions tagged with [Swift][iOS]. By comparison, [Python] has 2,144,311 questions, [Java] has 1,900,023 questions and JavaScript has 2,500,013 questions.

On GitHub, there are 113,009 public repositories tagged with Android, 42,941 public repositories tagged with iOS, 38,690 public repositories tagged for Kotlin, and 35,211 public repositories tagged for Swift. By comparison, 249,084 public repositories are available for Python, 171,111 for Java, and 252,804 for Javascript. We can state that mobile development is not consumed as other programming languages of a wide range like Java,

Python, or JavaScript.

Figure 1.1: Overview of the Objectives



.

As an indicator of the growth of the research in this field, the Machine Learning on Source Code platform [2], which is dedicated to navigating the literature created in the area of *Machine Learning for Big Code and Naturalness*, identifies 13 papers published in the first 6 months of 2023.

This paper has its motivation based on the actual requirements of a mobile development department from Cluj-Napoca, Romania which is specialized in native development. At the core of the paper is the need to understand how these LLMs can be created and used in real production projects and their potential impact on the productivity of such departments. The paper seeks to generate meaningful academic discussions together with practical value for an actual development team, by contributing with the following objectives:

1. To present a methodology for creating datasets for a programming language that is not as exploited in the research community, like Swift or Kotlin. This includes data gathering, data curation, and dataset publication respecting FAIR principles and HuggingFace standards for open, reusable datasets. The result of Chapter 3, Methodology for creating a dataset 3, is a number of datasets, with DOI that can be used in training from scratch, or fine-tuning models dedicated to native Mobile Languages, Swift and Kotlin. These datasets will be used to train the model in the following chapter.

2. To understand a basic approach of specializing in an LLM for code-generation (auto-completion) of Swift or Kotlin language. It goes thru the phases of training a GPT-2 model for mobile languages, using modern tools like HuggingFace libraries for training with Pytorch or Weights+Biases to track experiments. The steps presented include the setup of a GPT-2 Tokenizer, GPT-2 training, inference, and publication on a curated dataset obtained in the previous chapter. We explore if an approach of

training from scratch can have actual business value in the context of a mobile team. The hardware support for the experiments was provided by the DGX machines of the Technical University of Cluj Napoca.

3. The last objective is to understand the usage of LLMs on code generation in two important phases of a mobile development department, the technical onboarding of a new member and the challenge to switch technical stacks between Android to iOS for an existing member. This study involves 18 people, 16 participants, and 2 reviewers and is split into the two phases mentioned earlier. It uses technical problems dedicated to each phase and requests solutions from the participants with and without using AI-Code generators. It measures time, correctness, and 'technical integration' using ReviewerScore, a metric specific to the paper and extracted from actual industry standards, the code reviewers of merge/pull requests. The output is converted and analyzed together with feedback from the participants in an attempt to determine if the impact of using LLMs for code generation is significant.

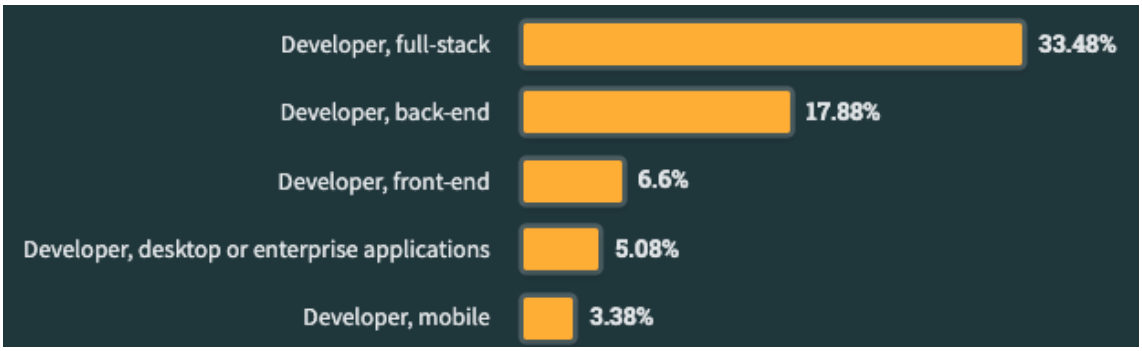


Figure 1.2: Mobile Developers as percentage in Stack Overflow Survey 2023. [1]

Chapter 2

Code Intelligence with LLM

Microsoft Research division groups under the term ***Code Intelligence*** all AI meant to help software developers improve the productivity of the development process [3]. It is expected to reach 28.7 million software developers in 2024 and the 'code intelligence' sector aims to help improve the productivity of the developers and the software development process by enabling AI solutions.

Developers can use search systems to find code written by others that does something similar to what they want and query them using natural language. Code completion systems can help developers when they are unsure about how to continue coding, by automatically suggesting suitable tokens based on the context of their edits. Code translation systems can also help developers when they want to translate code from one programming language (e.g. Python) to another (e.g. Java). We'll shortly describe how these types of programming behaviors are classified into tasks.

2.1 Tasks

1. **Clone detection**. A model is tasked with measuring the semantic similarity between codes. The task is to measure the semantic similarity between codes. This includes two subtasks: binary classification between a pair of codes and code retrieval, where the goal is to find semantically similar codes. Associated Datasets: BigCloneBench, POJ-104

2. **Defect detection**. A model is tasked with identifying whether a body of source code contains defects that may be used to attack software systems, such as resource leaks, use-after-free vulnerabilities, and DoS attacks. Associated Datasets: Devign.

3. **Cloze test**. A model is tasked with predicting the masked token from code, formulated as a multi-choice classification problem. This aims to predict the masked token of a code and includes two subtasks. The first one is to measure the accuracy of predicting the masked token from the whole vocabulary. The other is to test the

semantic reasoning ability by distinguishing between “max” and “min”. Associated Datasets: CT-all, CT-max/min

4. **Code completion**. A model is tasked with predicting the following tokens given a code context. Its subtasks are token-level completion and line-level completion. The former checks whether the next token has been predicted correctly, while the latter tests the goodness of the generated line. Associated Datasets: PY150, GitHub Java Corpus

5. **Code translation**. A model is tasked with translating the code in one programming language to the code in another one. Associated Datasets: CodeTrans

6. **Code search**. A model is given the task of measuring semantic similarity between text and code. Also named ”Code retrieval”. It measures the semantic relatedness between texts and codes and is composed of two subtasks. The first one is to find the most relevant code in a collection of codes according to a natural language query. The second subtask entails the analysis of a query-code pair to predict whether the code answers the query or not. Associated Datasets: CodeSearchNet, AdvTest; CodeSearchNet, WebQueryTest [4]

7. **Code repair**. A model is tasked with trying to automatically refine the code, which could be buggy or complex. Associated Datasets: Bugs2Fix

8. **Text-to-code generation**. A model is given the task to generate code given a natural language description. Associated Datasets: CONCODE, HumanEval, APPS

9. **Code summarization**. A model is given the task to generate natural language comments for a code. Associated Datasets: CodeSearchNet

10. **Documentation translation**. A model is given the task to translate code documentation between human languages. Associated Datasets: Microsoft Docs

2.2 Benchmarks and Evaluation Metrics

The following are benchmarks and evaluation metrics that are focused on computer intelligence tasks:

1. **CodeBLEU** In the area of code synthesis, the commonly used evaluation metric is BLEU or perfect accuracy, but they are not suitable enough to evaluate codes, because BLEU is originally designed to evaluate the natural language, neglecting important syntactic and semantic features of codes, and perfect accuracy is too strict thus it underestimates different outputs with the same semantic logic. CodeBLEU absorbs the strength of BLEU in the n-gram match, and further injects code syntax

via abstract syntax trees (AST) and code semantics via data-flow. This benchmark evaluates grammatical correctness by using the weighted ngram match and the syntactic AST match and uses the semantic data-flow match to calculate logic correctness. [5]

Useful in the following tasks: Text-to-code, Code Refinement and Code Translation Metrics: n-gram match, syntactic AST match, semantic data-flow match

2. **CodeXGLUE**

   Benchmarking mechanism tied to CodeXGLUE [6] set relies on results provided by the following models, depending on their tasks: CodeBERT [7], CodeGPT and an Encoder-Decoder model provided by the authors. Each task of the CodeXGLUE has its particular metrics usually tied to the datasets used. See Table 2.1.

Table 2.1: CodeXGLUE tasks and its particular metrics

Task	Metrics Info
`Clone detection`	For BigCloneBench data set uses F1 representing a binary classification to predict whether a given pair of codes has the same semantics. For POJ-104 it uses a MAP score (mean average precision).
Defect detection	The evaluation metric is an accuracy score that relies on a binary classification that predicts whether a function is vulnerable.
3. **Cloze test**	Macro-average accuracy scores for all languages is used as the overall evaluation metric.
Code completion	Metrics used: Exact match accuracy and Levenshtein edit similarity.
Code translation	The BLEU score, the CodeBLEU score and exact match accuracy are used.
Code search	Mean Reciprocal Rank (MRR) and F1 together accuracy scores are used as evaluation metrics, depending on dataset involved.
Code repair	Relies on exact match accuracy which involves the BLEU score and the CodeBLEU score.
Text-to-code generation	CodeBLEU and BLEU metrics are implicated.
Code summarization	BLEU score only.
Documentation translation	BLEU score only.

4. **MBPP + MathQA-Python** The Mostly Basic Programming Problems (MBPP) dataset contains 974 programming tasks, designed to be solvable by entry-level programmers. The MathQA-Python dataset, a Python version of the MathQA bench-

mark, contains 23,914 problems that evaluate the ability of the models to synthesize code from more complex text [8].

Associated datasets: BigCloneBench, POJ-104, Devign, CT-all, CT-max/min, PY150, GitHub Java Corpus, CodeTrans, CodeSearchNet, AdvTest, WebQueryTest, Bugs2Fix, CONCODE. More details in section 2.1 Tasks.

5. **APPS (Automated Programming Progress Standard)** The Automated Programming Progress Standard, abbreviated APPS, consists of 10,000 coding problems in total, with 131,836 test cases for checking solutions and 232,444 ground-truth solutions written by humans. Problems can be complicated, as the average length of a problem is 293.2 words. The data are split evenly into training and test sets, with 5,000 problems each. In the test set, every problem has multiple test cases, and the average number of test cases is 21.2. Each test case is specifically designed for the corresponding problem, enabling us to rigorously evaluate program functionality (Figure 3.2.

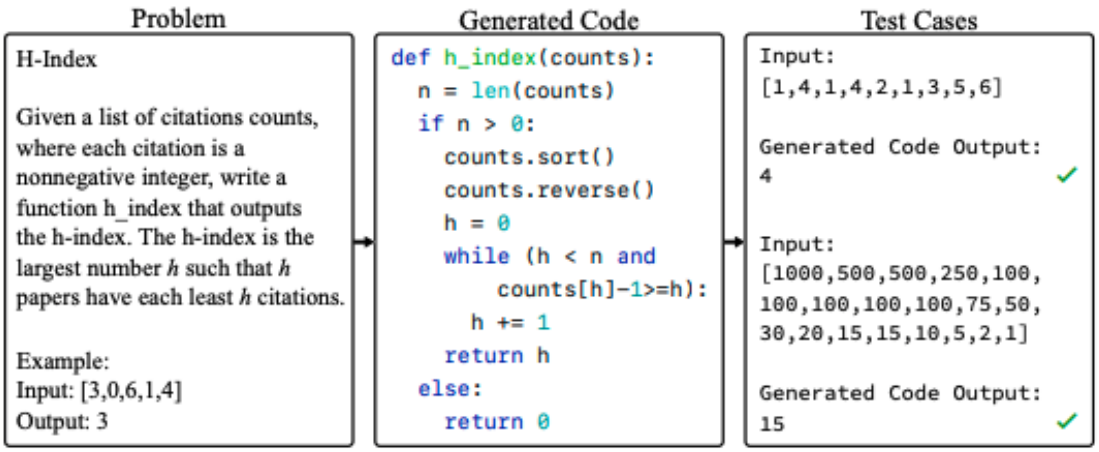


Figure 2.1: APPS Sample Containing problem, associated code, and test cases. [9]

APPS evaluates models not only on their ability to code syntactically correct programs but also on their ability to understand task descriptions and devise algorithms to solve these tasks. It contains problems at various levels of difficulty, covering simple introductory problems, interview-level problems, and coding competition challenges.

Metrics involved:

(a) **Test Case Average** which represents the average fraction of test cases passed.

(b) **Strict Accuracy**. The metric is computed by taking the number of solutions passing every test case divided by the total number of exercises.

Evaluating code generation on APPS is facilitated by a large bank of over 130,000 test cases and over 230,000 human-written solutions. The test cases are specifically chosen to probe correct functionality across the input space, including edge cases. By using test cases, the APPS [9] provides a gold-standard metric for model performance.

The dataset contains Python-only content. Most of the APPS test problems are not formulated as single-function synthesis tasks, but rather as full-program synthesis, reading input from stdin and printing output to stdout. APPS provides a precise and comprehensive view of code generation.

If a model were to perform well on APPS, this would indicate an ability to flexibly use data structures and programming techniques, as well as an ability to correctly interpret diverse task specifications, follow instructions, and understand human intent [9]

6. **HumanEval** Hand-written evaluation set containing 164 original programming problems with unit tests. These problems assess language comprehension, algorithms, and simple mathematics, with some comparable to simple software interview questions. Each problem includes a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem. Programming tasks in the HumanEval dataset assess language comprehension, reasoning, algorithms, and simple mathematics [10], [11].

   The metric used by HumanEval is **pass@k**. The evaluation metric introduced by SPoC approach [12] and modified by Codex team [13] to suit their model and evaluation dataset, HumanEval.

   pass@k is obtained by generating $n \geq k$ samples per task (in the Codex paper, $n = 200$ and $k \leq 100$), count the number of correct samples $c \leq n$ which pass unit tests, and calculate the unbiased estimator 2.1.

$$pass@k := \mathop{\mathbb{E}}_{Problems} 1 - \frac{\binom{n-k}{k}}{\binom{n}{k}} \tag{2.1}$$

A special mention is **MultiPL-E** which is a system for translating unit test-driven neural code generation benchmarks to new languages. It extends the HumanEval benchmark and MBPP benchmark to 18 languages and the authors state that is straightforward to extend with more. It uses the ‘pass@k‘ metric. [14]

2.3 Models For Text-to-Code

There are a growing number of language models that can analyze the text input provided by developers, obtain the context information, and generate code that fulfills the needs mentioned in the prompt. To solve this type of task, solutions must be tailored

for text understanding, code synthesis, code completion, and iterative refinement. We will describe some of these models that achieved high scores in corresponding benchmarks presented previously 2.2.

Open AI - Codex . OpenAI Codex is a descendant of GPT-3. Its training data contains both natural language and billions of lines of source code from publicly available sources, including code in public GitHub repositories. Is most capable in Python, but Open AI claims it works on JavaScript, Go, Perl, PHP, Ruby, Swift and TypeScript, and even Shell. As of March 2023, the Codex models are now deprecated and embedded into Chat models. Codes is behind the well-known Copilot, the "AI pair programmer" provided by Github, owned by Microsoft [11]. This tool is used in the Case Study presented in this paper as an AI Assistant option for the Android developers performing the task 5.

CodeBERT . Introduced in 2020 by Zhangyin Feng at. al, is a multi-programming-lingual model pre-trained on NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go). CodeBERT uses a Transformer-based neural architecture and is trained with a hybrid objective function that incorporates the pre-training task of replaced token detection, which is to detect plausible alternatives sampled from generators [7]. It is made and developed by Microsoft but made open-source.

GPTBigCode . The only non-corporate affiliation is on our list. It was proposed by Loubna Ben Allal et.al [15] in February 2023 an open-scientific collaboration working on the responsible development of large language models for code. It is trained using 1.1B parameter models on the Java, JavaScript, and Python subsets of The Stack and evaluated on the MultiPL-E text-to-code benchmark. The model is supported by the BigCode which is an organization empowering the machine-learning and open-source communities through open governance and invites AI researchers to collaborate on topics like constructing a representative evaluation suite for code LLMs, covering a diverse set of tasks and programming language, developing new methods for faster training and inference of LLMs and one crucial topic, the legal, ethics, and governance aspects of code LLMs [16].

AlphaCode . Presented by Yujia Li et. al. [17] in 2022, the approach involves large transformer language models to generate code, pre-trained on selected GitHub code, and fine-tuned on a curated set of competitive programming problems. Trained on 715.1 GB of code containing these languages: C++, C#, Go, Java, JavaScript, Lua, PHP, Python, Ruby, Rust, Scala, and TypeScript. Since it aims at solving complete competitive programming problems, not to provide code snippets like the other models, it presents a deeper lever of algorithmic reasoning.

We cannot close a section like this without mentioning generic LLM that presents results in code generation, models like ChatGPT from OpenAI which is powered by GPT-4, or Google's PALM-2 which was pre-trained on a large quantity of publicly available source

code datasets and has presented results in languages Python and JavaScript but can also generate specialized code in languages like Prolog, Fortran, and Verilog. We mentioned chatGTP because it is used as an AI-Assistant in generating Swift code in the Case Study chapter of this paper [5].

2.4 Available datasets

This section presents a few specialized datasets for Code Intelligence and a methodology to create a dataset for specific programming languages, using Swift/Kotlin as exemplification. In further sections, the research applies the dataset in the process of training an open-source GTP-2-based model with the purpose to obtain text-to-code capabilities for Swift/Kotlin programming language.

1. **Basic Semantic Parsing** Popular datasets explored in semantic parsing include Geo880, Jobs640, ATIS, and SQL but these are rarely used for code generation. The following is a subset of Semantic Parsing datasets used specifically for programming-related tasks:

   HEARTHSTONE (HS) dataset is a collection of Python classes that implement cards for the card game HearthStone. A similar dataset covers Magic: The Gathering which links the text that appears on cards to Java snippets code that precisely represents cards [18].

   IFTTT is a dataset that uses a specialized domain-specific language with short conditional commands used in IoT integration [19].

   RoboCup dataset pairs English rules with their representations in a domain-specific language that can be understood by virtual soccer-playing robots. [20]

2. **WikiSQL** consists of a corpus of 87,726 hand-annotated SQL query and natural language question pairs. These SQL queries are further split into training (61,297 examples), development (9,145 examples), and test sets (17,284 examples). It can be used for natural language inference tasks related to relational databases [21]

3. **DJANGO** is a collection of lines of code from the Django web framework, each with a manually annotated NL description. It pairs Python snippets with English and Japanese pseudocode describing them. The purpose of the data set is code generation. It comprising of 16,000 training, 1,000 development, and 1,805 test annotations. Each data point consists of a line of Python code together with a manually created natural language description [22].

4. **CodeSearchNet** Consists of 99 natural language queries with about 4k expert relevance annotations of likely results from CodeSearchNet Corpus. The corpus contains about 6 million functions from open-source code spanning six programming languages

(Go, Java, JavaScript, PHP, Python, and Ruby). The CodeSearchNet Corpus also contains automatically generated query-like natural language for 2 million functions, obtained from mechanically scraping and preprocessing associated function documentation [4].

5. **CodeNN** Consists of pairs of questions and answers automatically collected from StackOverflow. The final dataset retains 66,015 C# (title, query) pairs and 32,337 SQL pairs that are classified as clean (curated) [23].

6. **CoNaLa** Crawled from Stack Overflow, automatically filtered, then curated by annotators, split into 2,379 training and 500 test examples. Also provides a large automatically-mined dataset with 600k examples and links to other similar datasets. These data sets can be used for the CoNaLa challenge, or for any other research on the intersection of code and natural language [24], [25].

7. **150k Javascript Dataset** This dataset is released as a part of the Machine Learning for Programming project that aims to create new kinds of programming tools and techniques based on machine learning and statistical models learned over massive codebases. The dataset consists of 150,000 JavaScript files and their corresponding parsed ASTs The JavaScript programs are collected from GitHub repositories by removing duplicate files, removing project forks (copies of another existing repository), keeping only programs that parse and we aim to remove obfuscated files [26].

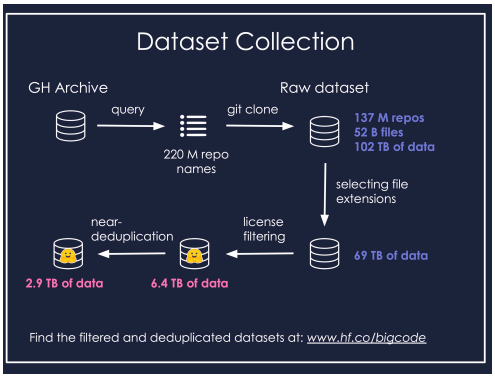


Figure 2.2: The Stack - Flow [27]

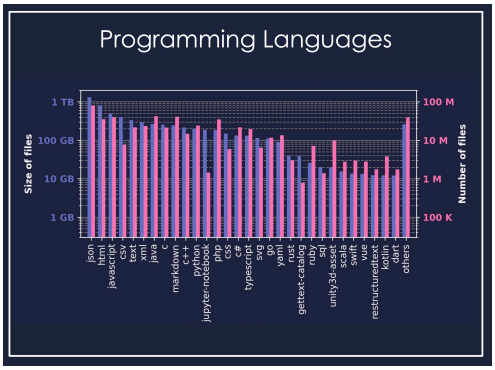


Figure 2.3: The Stack - Code Distribution [27]

8. **CONCODE** A new large dataset with over 100,000 examples consisting of Java classes from online code repositories, and develop a new encoder-decoder architecture that models the interaction between the method documentation and the class environment. Dataset comprising over 100,000 (class environment, NL, code) tuples by gathering Java files, from 33,000 repositories, containing method documentation from public Github repositories [28]

9. **CodeComplex** Consists of 4,200 Java codes submitted to programming competitions by human programmers and their complexity labels annotated by a group of algorithm experts. Their dataset takes inspiration from the recently disclosed AlphaCode. They assembled a dataset comprising codes from the coding competition platform Codeforces. The dataset encompasses 4,817 codes distributed across 7 classes, with approximately 500 codes per class. [29]

10. **TheStack** The BigCode project aims [16] to empower people by giving them control over their source code. The project has created and maintained The Stack, a huge dataset with 6 TB of source code from more than 300 programming languages [27]. The source code in The Stack is licensed permissively and can be used by anyone. However, the project respects the choice of developers who may not want their code to be used for building and testing machine learning models. The project has also developed a tool that lets users check if a repository with a certain username is part of The Stack dataset. Users can opt out of having their data in future versions of The Stack if they wish. See Figure 2.2 and Figure 2.3).

    This Stack dataset is the closest thing that can help us in our purpose to facilitate text-to-code in niche programming languages like Swift or Kotlin if we don't pursue building a dedicated dataset.

2.5 Related Work

Since the paper has three areas of research, we will split the related work into three paragraphs.

Case Study AI-Assistents for coding tasks, powered by Large Language Models, are a relatively new set of tools. Codex, the general-purpose programming model was released by Open AI in August 2021 and its public API was closed in March 2023. Github Copilot, the tool based on this model had its initial release in October 2021 and general release in June 2022. CodeBert or AlphaCode, models without open API or web interfaces, were released in 2020 and 2022 respectively.

Studies related to the impact of these tools are mainly focused on Python-specific challenges. The study in this paper draws from several of these research papers.

Vaithilingam et al. [30] the researchers explored how programmers use and view Github Copilot, with 24 participants in a user study. The study's goal was to understand the different modes and perceptions of the tool's usage. The findings showed that even though Github Copilot did not improve the time or success rate of completing tasks, most participants preferred to have the tool as part of their regular programming activities. This preference was due to Github Copilot often giving a useful initial point and reducing the need for online searches. However, participants faced difficulties in understanding,

changing, and fixing the code snippets that Copilot produced, which greatly affected their efficiency in solving tasks. Therefore, the researchers suggested several possible ways to improve the design of Copilot based on their observations and feedback from the participants.

One of the other few studies that focus on the usage of these tools is "Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming" by Majeed Kazemitabaar et al. [31] argues that OpenAI Codex is an AI code generator that can help novice programmers by creating code from natural language descriptions. Still, it is also important to take into account the possible negative effects of depending too much on such tools for learning and remembering. To study the impacts of AI code generators on beginner programming, a controlled experiment was done with 69 novices aged 10-17. The participants had to do 45 Python code-writing tasks, with half of them using Codex. After each code-writing task, a code-changing task was given. The results show that Codex helped a lot with code-writing performance, with a 1.15x higher completion rate and 1.8x better scores, without hurting performance on manual code-changing tasks. Moreover, learners who used Codex during the training phase did slightly better than their peers on evaluation post-tests done one week later, but this difference was not statistically significant.

Our study aims to replicate industry-related procedures in order to extract the actual practicality of using such tools and to evaluate what can be improved to adopt or make use of AI-Assisted code generation tools.

Methodology for Datasets Jalaj Pachouly et al. [32] introduced a tool to streamline the data collection from a repository, the pre-processing, and the validation steps of a dataset in a dynamic fashion. The paper introduces a quality process valuable for solutions requiring similar goals, still, our methodology aims to provide a programming-language-centric approach instead of a repository-targeted one, stil, we can draw from the paper an approach to automating and streamlining the process for any language by injecting the prepossessing rules.

Regarding related efforts, Pachouly et al. [33] have proposed an architecture to create a balanced dataset from the Open-Source Repository (GitHub) with features for Multi-Label classification and additional information about predicting and fixing software defects. The proposed method has focused on the dataset validation methods aiming at better-quality datasets for Software Defect Predictions. (Figura 18 din [33]) The need to design a custom dataset for specialized prediction tasks: Defect Estimates, References to the code for fixing the defect, Defect Severity, Resource Allocation, and Defect types.

Creating Models Text2app is a framework for creating Android apps from text descriptions, proposed by Masum et. al. [34]. It aims for Simplified App Representation (SAR) using an intermediate language between NL and Android code. They achieve this by transforming natural language into an abstract intermediate formal language representing

an application with a substantially smaller number of tokens. They attempt to generate mobile-specific code but the model used by them is based on a Seq2seq Neural Network.

One of the earliest proposed models for the approach of using Transformers for text-to-code tasks, was Colin B. Clement et. al [35] which introduced PyMT5 capable of performing method generation using combinations of signatures and docstrings for Python language alone.

One of the future goals of the model trained in this paper is to take the context of the repository. Building on top of the idea of Disha Shrivastava et. al [36] in which the prompt proposals take context from the entire repository, thereby incorporating both the structure of the repository and the context from other relevant files (e.g. imports, parent class files).

Chapter 3

Creating a dataset, for mobile code, using the FAIR principles

This research aims to bring the capabilities of Large Language Models closer to the needs of an actual mobile development team which mainly revolves around improving its output. As an alternative to using the models presented in Chapter 2, or generic LLM like ChatGPT, Microsoft Bing based on GPT-4, Google's PaLM 2, or other paid models dedicated to text-to-code tasks, a mobile team can train an open-source model to generate Swift or Kotlin code.

The first major setback in this challenge is the lack of datasets specialized in Swift or Kotlin programming language. As you will see in the following section, the current dataset corpora focus on languages with academic purposes like Python, or with a large spread like Java or JavaScript.

The amount of datasets dedicated to Swift or Kotlin code is limited. For example, at a quick search in the Papers With Code platform [37], the section dedicated to Datasets, the search for Python results in 90 datasets, Java results in 26, and JavaScript in 4 but both Swift and Kotlin have no results. In our research, we found that Dehaerne et. al [38] researched Code Generation using Machine Learning, and the paper mentions the existence of a Java repository for training and evaluation with more than 1.5M pairs of method bodies and corresponding class contexts called Android Drawer. The reference to this source is a Facebook page that cannot be accessed at the time of this study. One important mention here is a data set of mobile apps and code examples to evaluate clone detection algorithms across languages (Kotlin, Swift, and Dart) [39].

3.1 Code Gathering Sources

The two major sources for code-related tasks are Gihub and Stack Overflow. We will expand some information regarding these two data sources and we will present an overview of how this type of 'big data' can be used in machine learning tasks useful for

code intelligence.

GitHub GitHub is the largest source code host in the world which contains more than 200 million repositories.

To construct a dataset from a valuable resource such as Github, one potential first step is to use a tool or website able to aggregate existing packages or repositories. One example is Libraries.io [40] which monitors 5,264,226 open-source packages across 32 different package managers. For example, for MacOS or iOS-related projects, it can inspect more than 85.000 packages distributed via the Cocoapods dependency management system, over 4.000 distributed via SPM, and around 4000 distributed via Carthage.

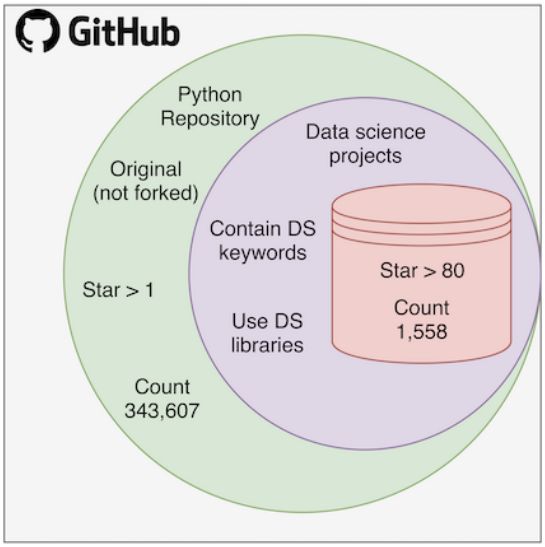


Figure 3.1: Filtering GitHub Repositories.

Curating methods are required on such large datasets and some involve usage or 'popularity' which are usually indicated by stars or forks in Github. Another parameter that can be used here is license. In the case of Text-to-Code tasks, it might be beneficial to filter out functions without documentation at least in training sets.

Usually, datasets created from this type of corpus will require more filtering steps, like avoiding code duplication, avoiding unit testing code, or auto-generated code from templates. Note that most of the resulting datasets will be used in the tokenization process. Our approach in the following sector relies on an existing GPT-2 tokenizer but other approaches can choose to use tools like Tree-Sitter [41] to create a concrete syntax tree for a source file which can later be used in different methodologies to achieve datasets for text-to-code tasks.

StackOverflow StackOverflow (SO) is the most popular website that features questions and answers on a wide range of topics in computer programming.

CaNoLa approach for dataset creation based on StackOverflow is to *learn semantic correspondence features between the natural language and code using neural network models for machine translation which can calculate bidirectional conditional probabilities of the code given the natural language and vice-versa of the natural language given the code.* [42].

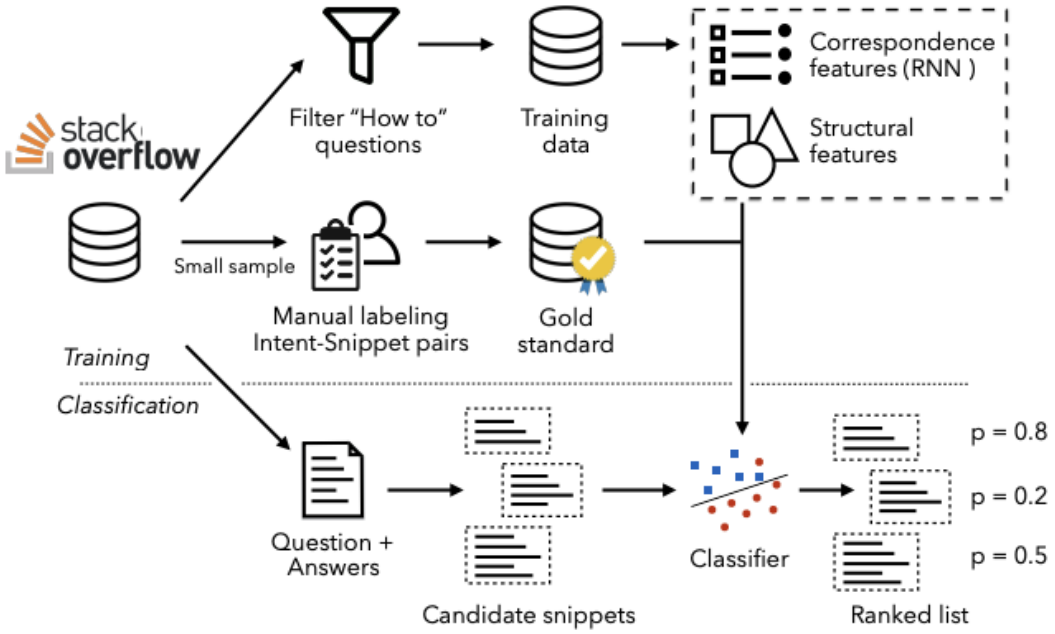


Figure 3.2: CaNoLa approach to dataset creation using StackOverflow

The technique for extracting aligned NL-code pairs from SO posts is based on a classifier that combines snippet structural features, with bidirectional conditional probabilities, estimated using a neural network model for machine translation.

Access to raw Stack Overflow data is obtained via the archive.org [43] website which provides regular data dumps of its content. This dump can be curated, and filtered based on programming language, trustworthiness, popularity, and so on.

CoNaLa mechanism focuses on extracting triples of three specific elements of the content included in SO posts:

1. Intent: A description in English of what the questioner wants to do;

2. Context: A piece of code that does not implement the intent, but is necessary setup, e.g., import statements, and variable definitions.

3. Snippet: A piece of code that actually implements the intent.

One last step to improve the performance of the mining solution is involving human feedback in an iterative manner. Professional programmers are asked to annotate the top-ranked predictions given by the mining model. These annotated results are used for training the model.

3.2 Methodology

In this paper, we use an unlabeled part of a dataset to train GPT-2, a large language model to adapt it for code generation in mobile-specific languages like Swift or Kotlin. For this task, a large corpus with Swift and Kotlin code can be useful without being accompanied by labels. In the following section, we will present a methodology to create datasets for programming languages of less wide circulation, using Kotlin and Swift as examples.

The result of the process will be 12 datasets, 6 for each programming language respecting the industry standard publication criteria.

3.2.1 Data Gathering

This methodology uses the terabytes of code repositories that are open for usage and download on GitHub. GitHub repositories can be accessed in two main ways:

1. **GitHub REST API**. It is intended for relatively minimal usage, per group of repositories or per group of users. It provides a limited rate for queries so in general it's not suited for large data gathering.

2. Public dataset inventories like **Google BigQuery**. This contains all files that are available in the open source standards.

The second option, Google BigQuery, is the one used in this methodology. As initial setup steps for this approach, one must create a Google Cloud account, create a Big Query project, and a Google Cloud Storage Bucket. The GitHub Activity Data can be consulted online [44].

For each mobile language, Swift and Kotlin the following steps were pursued for data gathering:

1. Creation of a dataset and a table in Google Big Query Project. See Figure 3.3

2. Creation of a bucket in Google Cloud Storage. See Figure 3.5

3. Creation of a query in Google Big Query Project. See Figure 3.4.

4. Running the query with the setting to output the results in the dataset and table created at step one. See Listing 3.1

5. Exporting the resulting dataset into the bucket created at step 2. The methodology in this paper used the export format of *JSON* with *gzip* compression (other options are available).

The result of these steps leads to the following results:

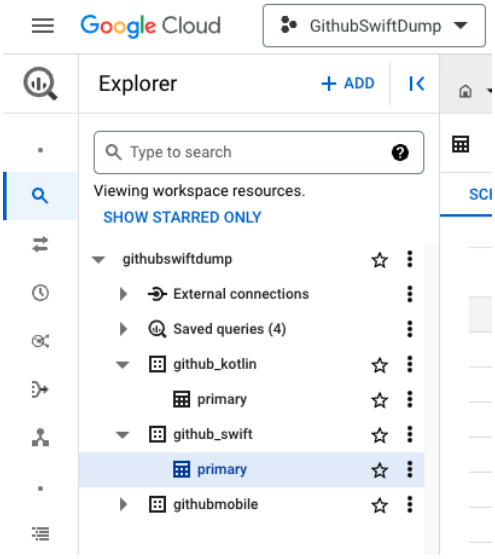


Figure 3.3: Google Dataset Creation

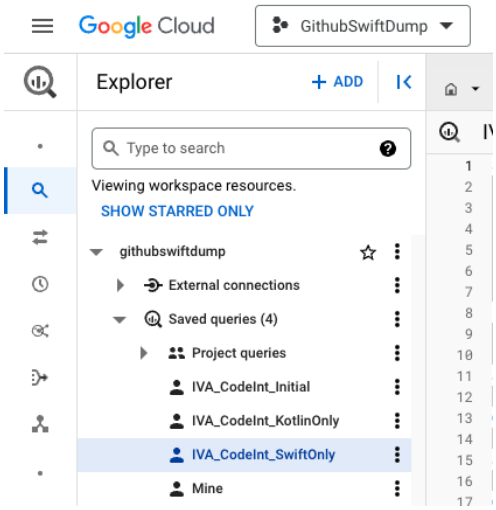


Figure 3.4: Google Query Overview

1. Extraction of Kotlin files resulted in
   - 2.7 TB Processed,
   - number of extracted rows/files was 464,215
   - total logical bytes 1.46 GB.
   - The result amounts to 7 json.gz files in a total of 361 MB.
2. Extraction of Swift files resulted in
   - 2.7 TB Processed,
   - number of extracted rows/files was 753,693
   - total logical bytes 3.05 GB
   - The result amounts to 3 json.gz files in a total of 712 MB.
3. Unarchived size for both Kotlin and Swift is 4.6 GB.

The queries used for each of the two programming languages can be found in Listing 3.1. We can consult relevant statistics related to the gathering process that might provide insight into the information stored in Google Big Query GitHub "dump". This includes the structure of the dataset 3.1, an instance of a field A.1, and the licenses involved A.3 A.2. Information related to the Instance of a field A.1, the Complete list of licenses of the Swift code A.2, Complete list of licenses of the Kotlin code A.3, can be found in the Appendix section.

Listing 3.1: SQL query used to extract the raw Swift source files from GitHub dump using Google Big Query

```
SELECT
  f.repo_name, f.path, c.copies, c.size, c.content,
   l.license
FROM
 (select f.*, row_number() over (partition by id order by
   path desc) as seqnum
      from `bigquery-public-data.github_repos.files` AS f) f
JOIN
  `bigquery-public-data.github_repos.contents` AS c
ON
  f.id = c.id AND seqnum=1
JOIN
  `bigquery-public-data.github_repos.licenses` AS l
ON
  f.repo_name = l.repo_name
WHERE
  NOT c.binary
  AND ((f.path LIKE '%.swift')
    AND (c.size BETWEEN 0 AND 1048575))
```

Table 3.1: Structure of the constructed dataset.

Field	Type	Description
repository	string	name of the GitHub repository
path	string	path of the file in GitHub repository
copies	string	number of occurrences in dataset
code	string	content of source file
size	string	size of the source file in bytes
license	string	license of GitHub repository

The structure of the dataset gathered via GitHub 'dump' using Google Big Query.

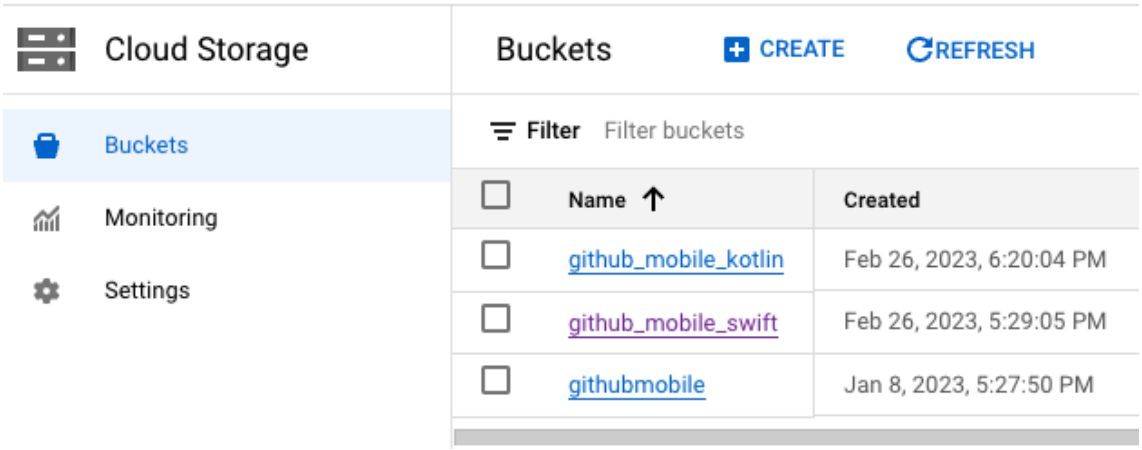


Figure 3.5: Google Cloud Bucket Overview

3.2.2 Dataset Curating

The performance of a model is directly tied to the quality of the data used for training. Training large models necessitates the use of extensive datasets of superior quality. However, when working with massive amounts of text data, ensuring the inclusion of top-notch text and handling preprocessing tasks become more complex.

GitHub repositories have varying quality levels because anyone can make them. To achieve good results in real-world situations, some choices need to be made carefully. Adding some noise to the training data can help the system deal with noisy inputs better during inference, but it can also cause more unpredictable outcomes. This paper exposes some general rules to curate the dataset, some extracted from predecessors methodology mentioned used in OpenAI's Codex Paper [11] or in the Codeparrot project that builds a 'clean' Python dataset [45].

The first phase in the curating process of this methodology is moving from Google Store to a Hugging Face dataset in order to easily perform operations on this. This involves the following steps:

1. Install gsutil, a Python application that lets you access Cloud Storage from the command line. E.g. 'pip install gsutil'
2. Configure gsutil to work with Google account via 'gsutil config.
3. Downloading the bucket.

```
gsutil -m -o "GSUtil:parallel_process_count=1" cp -r
   gs://github-mobile-swift
```

4. Install Hugging Face command line interface using 'pip install huggingface-cli'

5. Authenticate on Hugging Face: ‘huggingface-cli login‘
6. Create a Hugging Face dataset repository (via Web interface or cli). An example used in the research: ‘huggingface-cli repo create –type dataset –organization mvasilinuc iva-swift-codeint‘
7. Copy json.gz files into the Hugging Face repository location and push the changes. Note that for large files like these, one must use Git Large File Storage (LFS) support. ‘git lfs install‘.

At this point, we can easily manipulate the datasets using Hugging Face storage and the ‘datasets‘ Python library.

Before we move into code instructions used, we will focus on major items of the curating policy used in this methodology with customization related to Kotlin and Swift programming languages. The following rules were applied:

1. Removal of duplication files based on file hash.
2. Removal of file templates. Files containing the following:
   - For Kolin: `${PACKAGE_NAME}`, `${NAME}`, `${VIEWHOLDER_CLASS}`, `${ITEM_CLASS}`
   - For Swift: `___FILENAME___`, `___PACKAGENAME___`, `___FILEBASENAME___`, `___FILEHEADER___`, `___VARIABLE___`
3. Removal of the files containing the following words in the first 10 lines. Keywords used: `"generated, auto-generated", "autogenerated", "automatically generated"`.
4. Removal of the files containing the following words in the first 10 lines with a probability of 0.7. Keywords used: `"test", "unit test", "config", "XCTest", "JUnit"`
5. Removal of file with the rate of alphanumeric characters below 0.3 of the file.
6. Removal of near duplication based MinHash and Jaccard similarity.
7. Removal of files with mean line length above 100.
8. Removal of files without mention of keywords with a probability of 0.7.
   - For Swift: `"struct"`, `"class"`, `"for"`, `"while"`, `"enum"`, `"func"`, `"typealias"`, `"var"`, `"let"`, `"protocol"`, `"public"`, `"private"`, `"internal"`, `"import"`
   - Kotlin: `"fun"`, `"val"`, `"var"`, `"if"`, `"else"`, `"while"`, `"for"`, `"return"`, `"class"`, `"data"`, `"struct"`, `"interface"`, `"when"`, `"catch"`
9. Removal of files that use the assignment operator ‘=‘ less than 3 times.

10. Removal of files with the ratio between the number of characters and number of tokens after tokenization lower than 1.5.

For an example of file elimination, the file named 'Package.swift' A.4 therefor it will be gathered according to the query but it will be filtered due to multiple curating rules, for example, it contains no assignment operators since this file is constructed with the purpose of configuration and not code instructions.

The meaningful results of a curating process are seen in what the model that uses the dataset reflects. A model is as good as the dataset used for training and validation. Some indicators of the curation can be extracted by the reduction in files but an increase in file content as shown in Table 3.2. After curating, we will split the datasets into two different datasets for easier access to the training process presented in the next chapter. An example of the split of the resulted in Swift-based dataset can be found in table 3.3, the 'clean' dataset structure 3.4 and an instance of a field in the curated dataset A.5.

Table 3.2: Size Differences: Raw vs. Curated

Metric	Swift Raw	Swift Curated	Kotlin Raw	Kotlin Curated
Number of total files	753693	383380	464215	201843
Number of files below 500 Bytes	129827	3680	99845	3697
Avg. Content Size (Bytes)	4245	5942	3252	5205

A simplistic comparison of the elimination process looking into different quantitative characteristics.

Table 3.3: Example of curated dataset files split. Swift Language.

File	Size	Split
file-000000000001.json.gz	114 MB	Training
file-000000000002.json.gz	113 MB	Training
file-000000000003.json.gz	113 MB	Training
file-000000000004.json.gz	113 MB	Training
file-000000000005.json.gz	89.9 MB	Validation

After this process, we obtained 8 datasets for each coding language, all available publicly on HuggingFace [46], with the main 4 of them published publicly in an industry-standard manner. Datasets are enumerated in Table 3.5. It's paramount to state that the Swift and Kotlin programming languages are open-sourced on GitHub. A number of 875 files from 'apple/swift' repository inside the curated set (compared with 2786 from the non-curated dataset) and 567 files from JetBrains (The company that hosts Kotlin programming language), compared with 6835 found in the not-curated dataset.

Table 3.4: Structure of the curated dataset

Field	Type	Description
repo_name	string	name of the GitHub repository
path	string	path of the file in GitHub repository
copies	string	number of occurrences in dataset
content	string	content of source file
size	string	size of the source file in bytes
license	string	license of GitHub repository
hash	string	Hash of content field.
line_mean	number	Mean line length of the content.
line_max	number	Max line length of the content.
alpha_frac	number	Fraction between mean and max line length of content.
ratio	number	Character/token ratio of the file with tokenizer.
autogenerated	boolean	True if the content is autogenerated by looking for keywords in the first few lines of the file.
config_or_test	boolean	True if the content is a configuration file or a unit test.
has_no_keywords	boolean	True if a file has none of the keywords for Swift Programming Language.
has_few_assignments	boolean	True if file uses symbol '=' less than `minimum` times.

The structure of the 'clean' dataset after applying the curating rules.

3.2.3 Dataset Publication: FAIR and Datasets Cards

In the paper Datasheets for Datasets by Gebru et. al [47] the authors propose that every dataset is accompanied by a sheet that documents its motivation, composition, collection process, and recommended uses, arguing that this type of information will facilitate better communication between dataset creators and dataset consumers, and encourage the machine learning community to prioritize transparency and accountability. In this section, we will present how four of the datasets introduced in this paper, adhere to these characteristics.

One goal of the HuggingFace platform is to provide a platform where people can share and access various research datasets. By inviting users to upload their own datasets, the hope is to support growth in the machine-learning community and enable progress for everyone involved. The platform recommends that a dataset card should be provided with each dataset to promote responsible usage and inform users about potential biases. This concept draws inspiration from Mitchell et. al. [48] proposal of Model Cards. Dataset cards aim to educate users about the dataset's content, provide context for its usage, describe its creation process, and emphasize any other relevant factors that users should be aware of. The datasets created with this study respect the repository structure and dataset card instructions of the HuggingFace platform. See Figure 3.6.

All datasets have been publicized on the HuggingFace platform and all contain

Table 3.5: Resulted Datasets.

Dataset Name	Description
iva-swift-codeint	Contains uncurated Swift files gathered with the purpose to train a code generation model. Contains Dataset Card and adheres to FAIR principles.
iva-swift-codeint-clean	Curated dataset of Swift files. Contains Dataset Card and adheres FAIR principles.
miva-kotlin-codeint	Contains uncurated Kotlin files gathered with the purpose to train a code generation model. Contains Dataset Card and adheres to FAIR principles.
iva-kotlin-codeint-clean	Curated dataset of Kotlin files. Contains Dataset Card and adheres to FAIR principles.
iva-swift-codeint-clean-train	Curated dataset of Swift files, the train split. Contains Dataset Card.
iva-swift-codeint-clean-valid	Curated dataset of Swift files, the valid split. Contains Dataset Card.
iva-kotlin-codeint-clean-train	Curated dataset of Swift files, the train split. Contains Dataset Card.
iva-kotlin-codeint-clean-valid	Curated dataset of Kotlin files, the valid split. Contains Dataset Card.

All datasets are publicly available at `https://huggingface.co/mvasiliniuc`.

Figure 3.6: Snapshot of publicly available datasets created as a result of this research.

DatasetCards. We will present a summarisation of dataset card information using the curated dataset for Swift programming language.

Dataset Description. This includes Dataset Summary, Supported Tasks and Leaderboards, and Languages. For our case, we have a summary containing a description, content statistics, how to use it, and the following tags to provide findable information about the tasks and languages:

1. Tasks: Text Generation
2. Sub-tasks: language-modeling
3. Languages: code
4. Size Categories: 100K<n<1M
5. Language Creators: crowdsourced
6. Tags: code, swift, native iOS development
7. License: other

Dataset Structure. This includes Data Instances. Data Fields. Data Splits, all included in the Dataset card with the same information found in this chapter. See Data Structure Table 3.4, Instance Field Table A.1, and Split Table 3.3.

Dataset Creation. Should contain Curation Rationale and Source Data. In this section, we elaborated on the reason for the dataset creation, this being code generation, we provided information about gathering sources and curating logic, as presented in this chapter. See Listing 3.1

Considerations for Using the Data. In this section, we draw attention that the dataset comprises source code from various repositories, potentially containing harmful or biased code, along with sensitive information such as passwords or usernames.

Additional Information. The section is filled with creator details and research identifiers.

For the datasets that are in 'final' form, DOI (Digital object identifier) was set up 3.2.

FAIR is a set of core principles for effective data management and stewardship that was proposed in 2016 by E. A. Huerta et. al. [49]. They aim to make data findable, accessible, interoperable, and reusable (FAIR) so that they can be used for scholarly purposes. These principles have evolved to cover different types of digital assets, such as software, tools, algorithms, and workflows that are involved in data generation. Nowadays,

Listing 3.2: DOI citation for dataset containing Raw Swift language

```
@misc {mircea_vasiliniuc_2023,
author     = { {Mircea Vasiliniuc} },
title      = { iva-swift-codeint (Revision c09ebf8) },
year       = 2023,
url = {
   https://hf.co/datasets/mvasiliniuc/iva-swift-codeint },
doi        = { 10.57967/hf/0778 },
publisher = { Hugging Face }
}
```

the FAIR principles are being adapted and implemented in the context of AI models and datasets. For the main datasets in our research we used two self-assessment FAIR tools, one provided by the Dutch national center of Expertise and Repository for research data (DANS) [50] and one provided by the Australian Research Data Commons [51]. Examples of assessment criteria are in table A.1 where answers are given from a predefined set.

The four main datasets published with this paper achieved a score of 87% FAIR score in DANS evaluation (77% at Findable, 100% at Accessibility, 100% at Interoperability, 72% at Reusability). Similar results were obtained using the Australian Research Data Commons tool.

Chapter 4

Building a Model for Mobile Languages

To achieve results in text-to-code tasks, OpenAI's Codex used a causal language modeling approach with 100 billion tokens involved. This study doesn't attempt to compete with such models, instead, it pursues to discover the process of adapting an existing large model and tokenizer for the same task but a different and limited dataset consisting of programming languages used in native mobile development.

With the availability of a vast pretraining corpus on Swift/Kotlin mobile programming languages and a tokenizer, we can approach the training of a GPT-2 (small version) transformer model. One inherent task when dealing with textual data is to present the model with the initial segment of a code sample and prompt it to generate potential completions. The task can be interpreted as code autocompletion and a decoder-only architecture such as the GPT family of models is usually best suited for this task. This is an example of a self-supervised training objective that allows us to leverage the dataset without any annotations.

The approach in this paper aims to train a large model, GPT-2 from scratch to obtain results in the text-to-code task of Code Intelligence presented 2 tailored for niche programming languages used in native mobile development. More precisely, we build one model for each programming language, Swift and Kotlin. In our case the datasets are large and unlabeled, meaning that they can usually only be labeled through the use of heuristics. As the quantity of training data you have access to approaches the amount of data used for pretraining, it becomes an argument in favor of considering training the model and the tokenizer from scratch.

Since models like GPT-2 and GPTBigCode are available for anyone to fine-tune, one option would be to avoid the training loop and settle with fine-tuning it. A general consensus at this point is stat it makes sense to train a model from scratch for programming languages, molecular sequences, or music notes. For the purpose of this research, our aim was to start from scratch.

4.1 Training a GPT-2 model

Alec Radford et. al introduced GPT-2 (Generative Pre-trained Transformer 2) a well-known open-source artificial intelligence large language model created by OpenAI in February 2019. The auto-regressive model is built using transformer decoder blocks.

Having a limited, unlabeled dataset, as we presented in section 3 can present a limitation in performance. For these situations, a causal language model, such as GPT-2, utilizes predictive abilities to determine the subsequent token in a token sequence, with the model's attention limited to the tokens preceding it. Consequently, the model lacks visibility of forthcoming tokens.

The steps in our approach involve the following:

1. Creating datasets for training tokenizer and model for Swift and Kotlin programming language. Details about this procedure can be found in the previous section.

2. Training the standard open-source GTP-2 Tokenizer for Swift and Kotlin programming languages.

3. Prepare dataset for processing.

4. Initializing and Training the standard open-source GPT-2 Model (small version).

Tokenizers For Swift/Kotlin

This approach uses a GPT-2 tokenizer. Based on byte-level Byte-Pair-Encoding. The process of Byte Pair Encoding (BPE) commences with a collection of individual characters and proceeds to construct a vocabulary by systematically merging the most commonly appearing character pairs. These merged pairs are then incorporated into the vocabulary. This iterative procedure continues until a predetermined vocabulary size is achieved.

The available tokenizer utilizes an efficient conversion process from strings to integers by utilizing a vocabulary composed of atomic strings. Alongside this vocabulary, there is a corresponding method that performs operations such as conversion, normalization, cutting, or mapping on a text string, resulting in a list of indices. This list of indices serves as the input for our neural network.

For our situation, we need a custom tokenizer that is suited to tokenizing Swift or Kotlin programming code. Since code has a large body of tab characters, newlines, escapes and other nonprintable characters, the GPT-2 tokenizer maps all the 256 input bytes to Unicode strings that can be consumed by the standard BPE algorithms

The process of training a tokenizer does not involve backpropagation or weights. Instead, it focuses on establishing an optimal mapping from a text string to a list of integers that can be effectively processed by the model. The concept involves gradually building a vocabulary of a predetermined size by systematically merging the most commonly occurring pair of tokens within the vocabulary.

Listing 4.1: Initializing GPT tokenizer with raw dataset

```
# Define iterator
def batch_iterator(batch_size):
    for _ in tqdm(range(0, 16384, batch_size)):
        yield [next(iter_dataset)["content"] for _ in
   range(batch_size)]
# Base GPT-2 tokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")
base_vocab = list(bytes_to_unicode().values())
# Load raw dataset
dataset = load_dataset("mvasiliniuc/iva-swift-codeint",
   split="train", streaming=True)
iter_dataset = iter(dataset)
# Training
new_tokenizer = tokenizer.train_new_from_iterator(
    batch_iterator(12), vocab_size=200_000,
   initial_alphabet=base_vocab
)
```

Mainly our steps involve, initializing the GPT2-Tokenizer, training it on Swift/Kotlin code, and pushing it to HuggingFace for further usage. The approach relies on PyTorch and Huggingface APIs.

Our GPT-2 tokenizer encompasses a vocabulary of 50,257 words, which includes the following components:

- Base vocabulary consisting of the 256 values for the bytes.
- Additional set of 50,000 tokens generated by iteratively merging the most frequently co-occurring tokens.
- Special character in the vocabulary to signify document boundaries.

In our process, the first step is to initialize the GPT-2 tokenizer on the non-curated dataset. See Listing 4.1.

This initial tokenizer is used to apply extra curating rules into the processing phase of the dataset. The tokenizer is re-initialized afterward with the curated/clean training split as seen in Listing 4.2.

In the end, we push the tokenizer in the HuggingFace model repository to be further used in the training of the model.

```
new_tokenizer.save_pretrained(args.tokenizer_name,
   push_to_hub=args.push_to_hub)
```

Listing 4.2: Re-Initializing GPT tokenizer with curated dataset train split

```
# Load curated dataset
dataset =
   load_dataset("mvasiliniuc/iva-swift-codeint-clean-train",
   split="train", streaming=True)
iter_dataset = iter(dataset)
# Training
new_tokenizer = tokenizer.train_new_from_iterator(
    batch_iterator(12), vocab_size=200_000,
   initial_alphabet=base_vocab
)
```

Training Process

GPT-2, a transformer model, introduced by Radford et al. [52], was pre-trained on a vast corpus of English data using a self-supervised approach. This involved training on raw texts without human labeling, allowing it to leverage publicly available data. The training process involved predicting the next word in sentences.

```
GPT-2 (xl) size: 1797.2M parameters
GPT-2 size: 239.4M parameters
```

The first step is to initialize the open-source model and upload it to the destination of our final model. Later we will upload checkpoints and final versions of this model to the same repository.

```
# Load tokenizer trained for Swift/Kotlin code tokenization
tokenizer =
AutoTokenizer.from_pretrained(mvasiliniuc/iva-codeint-swift-small)
config_kwargs = {
    "vocab_size": len(tokenizer),
    "scale_attn_by_inverse_layer_idx": True,
    "reorder_and_upcast_attn": True,
}
# Load model config (GPT-2 small in this case)
config = AutoConfig.from_pretrained("gpt2", **config_kwargs)
# Initialize new model with config
model = AutoModelForCausalLM.from_config(config)
# Save the model to the hub
model.save_pretrained(iva-codeint-swift-small,
   push_to_hub=args.push_to_hub)
```

The second milestone is to configure the data loading support. It is essential to provide the model with sequences that fill its context. GPT-2's context length is set at 1,024 tokens, it is necessary to consistently supply training sequences of 1,024 tokens.

However, it is possible that some of our code examples may be shorter or longer than that length. What is needed is to create Custom Lenght Dataset programmatically, by creating a custom IterableDataset, which is a helper class provided by PyTorch) for preparing constant-length inputs for the model. This will help us yield the elements in sequences that provide a reliable stream for the training process. DataLoader class from Torch will be used to load the iterable datasets.

An overview of this process involves the following details: (a) Construction of an IterableDataset (b) 1,024-token sequence length during training (c) 1,024-token sequences to keep in the buffer (d) Estimation of average character length per token. 4.85 characters per token in Swift language. 4.00 for Kotlin language.

The approach was derived from Hugginface's 'Training a causal language model from scratch' article [53] published in December 2022 and O'Reilly Natural Language Processing with Transformers book, released in May 2022 [54].

The Training Process is implemented around the Accelerate library from Huggingface. Accelerate is a library that simplifies and enhances the process of training and inference at scale in PyTorch. By adding only a few lines of code, it enables the execution of the same PyTorch code across any distributed configuration. Accelerate enabled the study to create a straightforward, efficient, and adaptable solution. There is an even higher level API from HuggingFace that can be used to make the training process simple, called Trainer API but for the scope of this study, we choose Accelerate. An overview of the training process and entities involved:

Preparation Phase. Involves calling Accelerate *prepare()* function which prepares all objects for distributed training and mixed precision. The objects involved are the model, the optimizer, the train data loader, and the evaluation data loader. Optimizer used is AdamW with a cosine learning rate schedule of 2e-4. **Configuration Phase.** Arguments used can be found in 4.1, the hyperparameters used are extracted from similar solutions related to Python Code generation from HuggingFace [53] and Natural Language Processing with Transformers [54].

Logging and Versioning. The logging mechanism relies on Weights and Biases to keep track of the experiments and manage the runs so we can easily inspect logs and data using a web interface. For versioning, we rely again on HuggingFace to store the versions of the model.

Evaluation. Since this is a causal language model, it is evaluated by calculating the cross-entropy Loss and perplexity. The mechanism contains a function to evaluate these two values after a certain number of steps (1000). The aim is for a low perplexity in order to increase performance. We use perplexity to measure how well the model predicts the correct tokens with the corresponding probability distributions. We compute perplexity by raising the cross-entropy loss from the model's output to the power of e. In the early stages of training, we may get numerical overflow when calculating perplexity because the

Table 4.1: Training arguments. For a complete list, please see Table E.1

Parameter	Value
seed	1
seq_length	1024
weight_decay	0.1
learning_rate	0.0005
max_eval_steps	-1
shuffle_buffer	10000
max_train_steps	150000
_mixed_precision	fp16
num_warmup_steps	2000
train_batch_size	5
valid_batch_size	5
lr_scheduler_type	cosine
save_checkpoint_steps	15000
gradient_checkpointing	false
gradient_accumulation_steps	1

Configuration used in the training process.

loss is high. We assign infinity as the perplexity value when this occurs.

Training. The algorithm mainly involves the following steps:

1. Iteration over the DataLoader and passing batches to the model.
2. Reevaluate loss function using gradient accumulation steps (we choose 1 for this configuration).
3. Gradient Norm Clipping [55] is applied (using Accelerate API).
4. Model evaluation at a certain number of steps.

Note: The code of the training process can be found in the associated repository [56].

To ensure a standard level of reproductivity, we must note that the same datasets were involved in the training process of the two models, one for Kotlin language and one for Swift. Datasets are the ones created in Chapter 3 with a high degree of documentation and availability. Preprocessing steps were followed as described in this chapter. All the code involved in the training process and preprocessing is available for review in the GitHub public repository associated with the paper [56].

The code relies on open-source libraries like Datasets, Transformers, and Accelerate from Huggingface and the PyTorch library.

Hardware Support. The computation environment used for the study was provided by the Technical University of Cluj Napoca. All training was done in a Docker container

with an associated Python Environment. The approach uses libraries and standard steps for optimizing the training process and data parallelism. Considering the model deals with a large model like GPT-2 (239.4M parameters), a solution that can scale the training process is the most appropriate. For example, Accelerate uses DDP (Data Distrubted Parallelism), which implements data parallelism at the module level which can run across multiple machines.

All operations outside of training were done on a Macbook Apple M1 Pro, Memory: 16 GB, 16 Cores GPU which PyTorch could not activate CUDA on. The training itself was done on the DGX machine provided by UTCN. The machine has the following characteristics in the table. Maximum 2 GPUs at the time were used.

Table 4.2: DGX Hardware Capabilities.

CPU count	40
GPU count	8
GPU type	Tesla V100-SXM2-32GB

The system information of the machine the model was trained on. This can be seen also in the Logs from Weights and Biases.

4.2 Results

The purpose of creating this model was to understand if an openly available LLM like GPT-2 is capable of providing solutions specific to mobile development if trained on datasets containing programming languages dedicated to this type of work. Since at the moment of writing the paper, the access to benchmarks that can evaluate models dedicated to Swift or Kotlin language, this version of the paper will showcase results without the reliance on the usual Benchmarks as the ones that can be used for Python, Java, or Javascript. We will present results only for the Swift model, the *IVA-CodeInt-Swift-Small* [57], but note that IVA-*CodeInt-Kotlin-Small* [58] is publicly available for usage. Both models have Model Cards with information about the purpose, usage, reproductivity, and when benchmarks permit, evaluation details. Training configurations can be also consulted in 4.1 in the Appendix section.

For inference, we used HuggingFace API [59] and Web Interface [60] to generate predictions, complete documentation, and functions in the two language models that were targeted by the paper. The model is able to generate code that has a decreased level of industry usage but it can help the developer with a structure or skeleton for a specific problem. It is capable of providing correct solutions, especially for prompts related to the usage of Platform API which can be of increased usage with a developer adapting to a new platform, as the ones presented in Case Study 5.2.2 especially Technical Stack Switch 5.2.2. We will showcase a snapshot from the Web Interface in this section and for the ones accessed via API, please consult B.0.2.

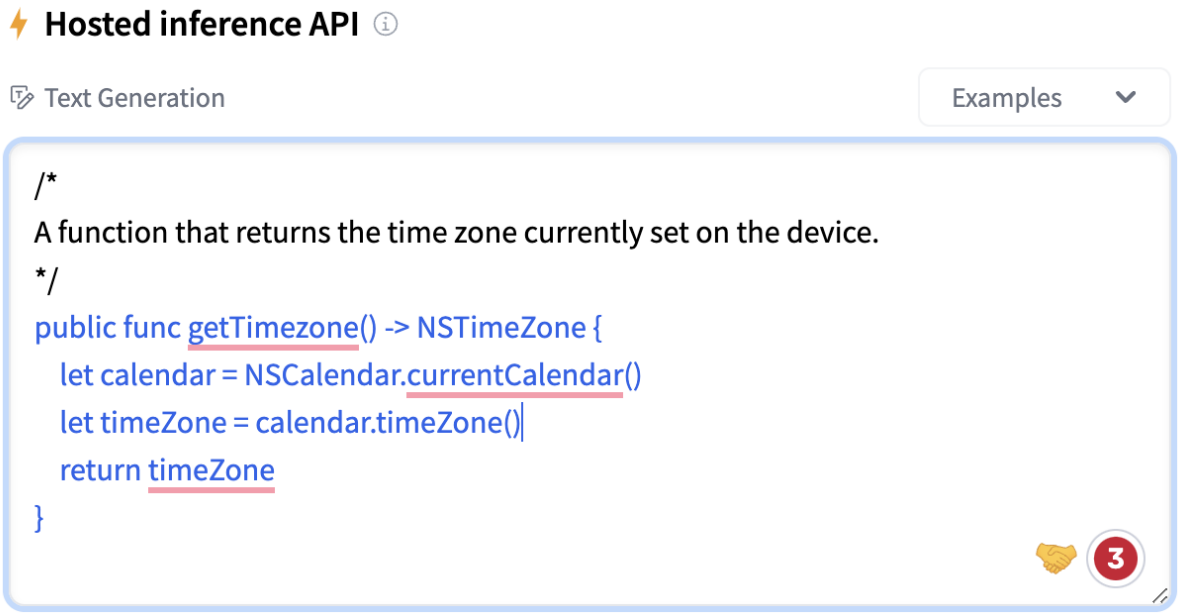


Figure 4.1: Current Timezone prompt in Swift by IVA-CodeInt-Swift-Small. The code in the black color is the prompt, and the code in blue is the generated code.

Some examples of mobile-specific code, in Figure 4.1 we see the output of the Swift-trained model performing code generation for a task that is clearly platform-specific, retrieving the current time zone of the device. We can see functional code, correct usage of API, static call of current calendar on NSCalendar type, and correct function call. In Figure 4.2 in which we see the model correctly generating the code for generating a local notification in Swift language with the correct API associated with it, usage of the default notification center and correctly generating the function and associated arguments still it generated a callback argument which is out of context. Another example is the usage of a high-level persistence library, specific for the iOS SDK, the User Preference as seen in 4.3 where the output is functionally correct but would need adaptations in a real project.

One last example resolves around the prompt to create a struct that can be encoded/decoded and represents a widely known domain model. Generated code is close to the domain and generated parts respect the lines given as instruction, It can be used as a starting point to resolve an issue. 4.4.

We must specify that generated code by this model will have a relatively low score for technical integration mainly because it is generating non-modern syntax and API usage since the corpus of the training contains more old code than new, modern ones. For more examples, please consult the Appendix section for Training a Model B.

To assess how well a training process or model works, performance metrics are essential. These metrics 4.3 reveal useful information about how fast and accurately the model learns, how well it can handle new data, and how much progress it makes during

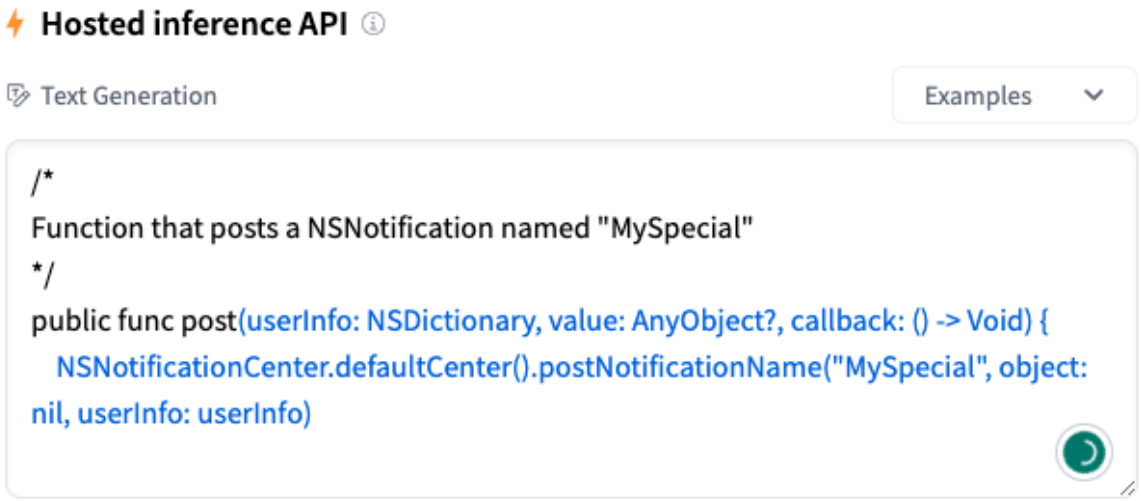


Figure 4.2: Push Notification Code Generated by the Model by IVA-CodeInt-Swift-Small [57]. The code in the black color is the prompt, and the code in blue is the generated code.

.

training. Based on these metrics, one can make better choices for enhancing and fine-tuning the mode. In these metrics "loss/train" refers to the training loss, "time_per_iteration" is the time taken per iteration, "_step" indicates the step number, "_runtime" is the total runtime, "tflops" represents computational performance, "samples" refers to the number of training samples, "loss/eval" is the evaluation loss, "_timestamp" indicates the time the metrics were recorded, "perplexity" reflects the model's uncertainty, "loss_per_step/train" is the average loss per training step, "lr" represents the learning rate, and "steps" indicates the total number of training steps.

Charts can be highly useful for visualizing and interpreting the values associated with these metrics and therefore help us understand the training process better. We choose to present only perplexity in Figure 4.5 and the loss/step-train in Figure 4.6 and the rest of the charts can be consulted in Appendix C

Related to the connection to Case Study in 5, the aim was to provide a model that can be ready for usage by real developers. The model was excluded from the study due to reduced output when it comes to specific programming problems, particularly the ones involved in timed tasks. The model was evaluated by the reviewers of the Study and the following were observed:

1. The model in this phase can assist a developer in finding platform-specific libraries for a specific problem.

2. With the corresponding prompts, it can provide minimal solutions that can be used directly in mobile-related projects.

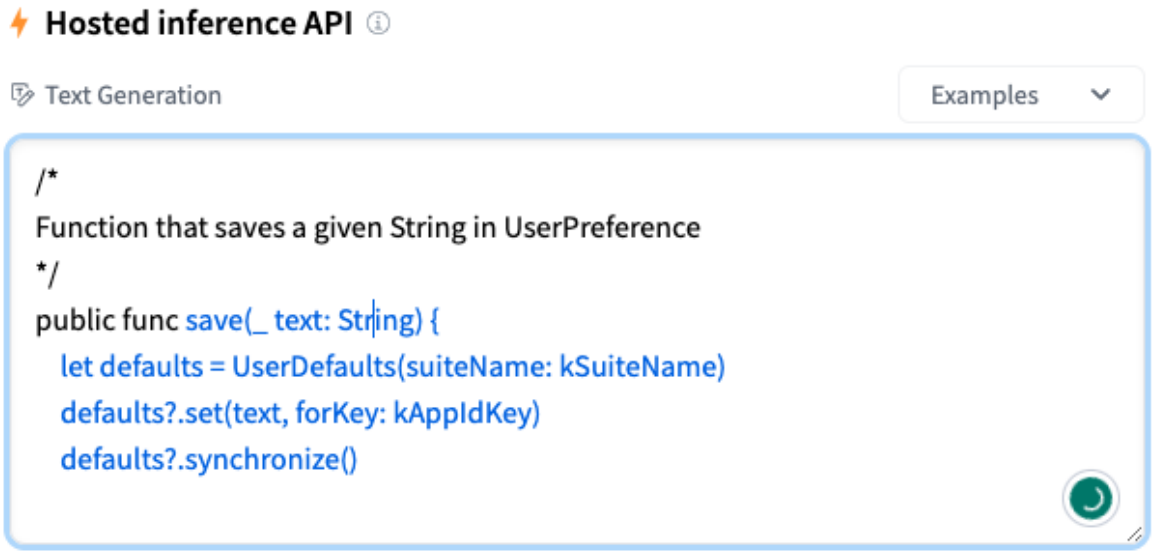


Figure 4.3: User Preference Save Generated in Swift by IVA-CodeInt-Swift-Small [57]. The code in the black color is the prompt, and the code in blue is the generated code.

3. The models are able to correlate common pair of concepts in Swift and Kotlin languages, like Push Notification with Tokens, or User Preferences with Persistence.

Nevertheless, it was decided to exclude it from the review since the interactions with the model in this phase would not be in line with the purpose of the Study.

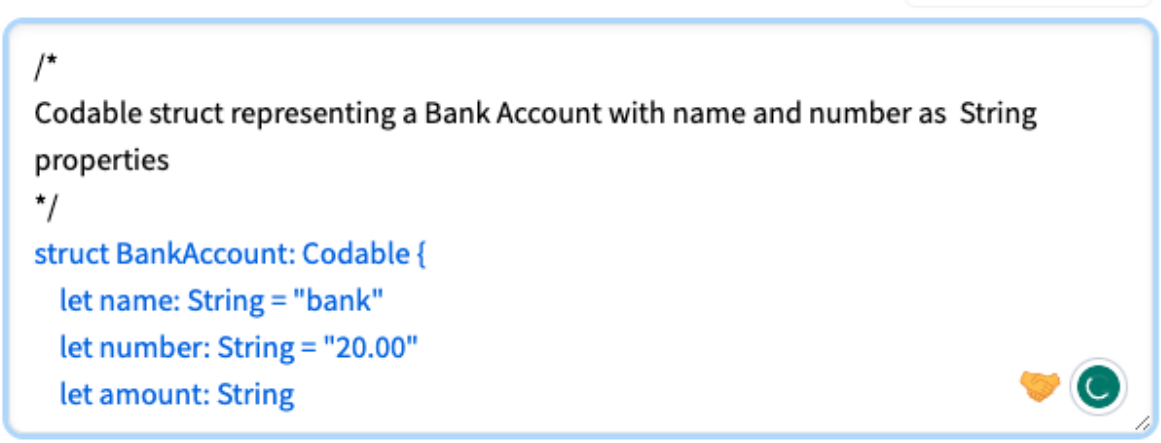


Figure 4.4: A Bank Account struct generated in Swift by IVA-CodeInt-Swift-Small [57]. The code in the black color is the prompt, and the code in blue is the generated code.

Listing 4.3: Metrics of the training process result


```
{
    "loss/train": 0.9347587823867798,
    "time_per_iteration": 0.3986995220184326,
    "_step": 149999,
    "_runtime": 78899.45855522156,
    "tflops": 16.874756165293206,
    "samples": 749995,
    "loss/eval": 1.0912784337997437,
    "_timestamp": 1687694268.8288612,
    "perplexity": 2.978078842163086,
    "loss_per_step/train": 0.934758722782135,
    "lr": 2.2529228482781605e-13,
    "steps": 149998
}
```

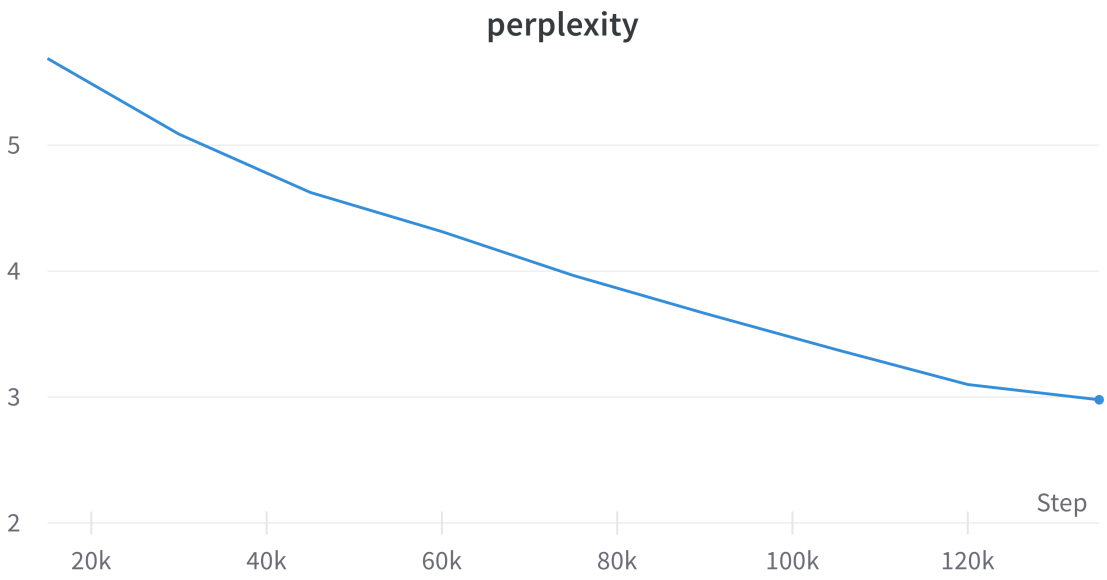


Figure 4.5: Perplexity of IVA-CodeInt-Swift-Small [57].

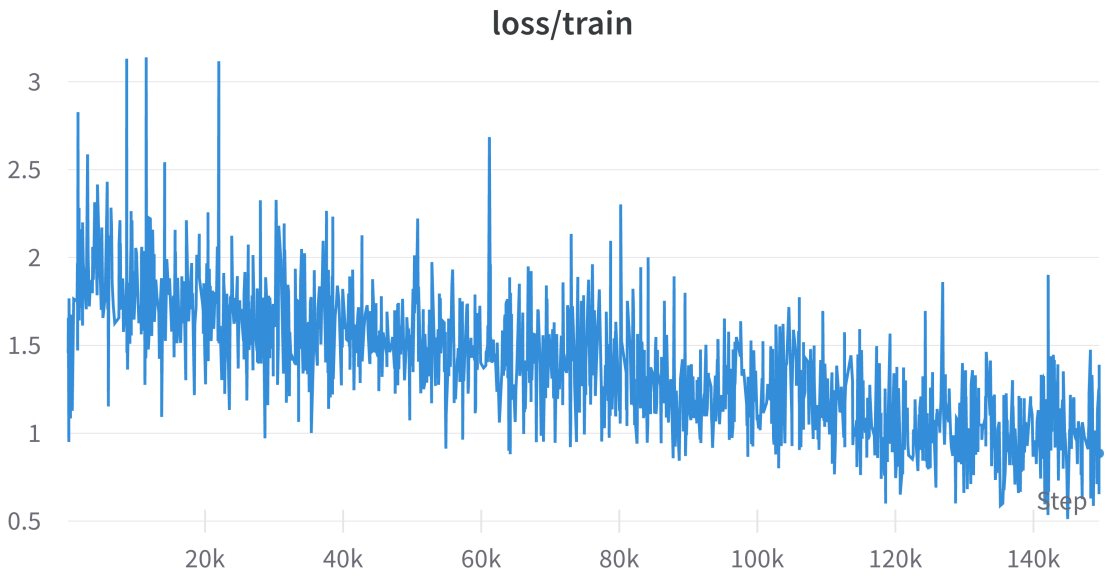


Figure 4.6: Training Loss of IVA-CodeInt-Swift-Small [57].

Chapter 5

Case Study: Using AI-Assisted Code Generation

One of the main purposes of this research paper is to evaluate the performance of AI-assisted programming in real mobile software departments that are focused on native mobile development. The study was performed between May and June 2023 with members of the mobile department of a software development company based in Cluj-Napoca, with Romanian ownership and management.

The characteristics of native mobile development involve programming in two main languages Swift for iOS apps destined for iOS operating systems and Kotlin for apps dedicated to the Android operating system. The study focuses on native development only and avoids implications with non-native technological stacks.

Two of the requirements of a mobile department lead are to create a technical onboarding procedure and to provide means for a developer to be proficient in the sibling platform (iOS to Android or vice-versa). The study is split following these two requirements, with more details in the Study Design section.

Having good procedures for these two phases ensures three major objective results for the team: (a) individual software engineering efficiency, (b) team collaboration regardless of platform, and (c) consistency in output between the two platforms. The latter is crucial for long-term projects.

When working on a native mobile project, in a sensitive sector like Banking and Financial, a technical lead must prepare a team for projects with the following characteristics, the list not being comprehensive:

- The lifetime of the applications is long. Between 3 and 10 years.
- The codebase must respect a quality gateway and technical and security audits.
- The codebase must be verbose and highlight business-specific requirements, rules, and limitations as much as possible.
- Knowledge transfer between team members must occur with minimum friction.

- A high level of technical dependency on other services (environments, middlewares, backends, etc.)
- Constant changing or requirements due to legal or user-driven demands. Code must quickly respond to changes and extensions.
- Maintaining two different codebases for Android and iOS applications.

We must make the note that the Study can be considered a generic one, for any mobile team, with the observation that in Banking and Financial projects, there is a prioritization for having a standardized code base and a strong technical alignment within the team.

In order to be more efficient in achieving some of the characteristics above, the study aims to understand if using AI-assisted programming tools will have an impact on getting developers onboard in a project, helping them understand the sibling platform code better or even help them with a smooth transition between the two native development environment. The AI-assisted programming tools involved are GitHub Copilot and chatGPT. Mainly the study aims to answer the following research questions:

RQ1. How can an AI-based code generator affect the experience when onboarding a new team member or switching technical stacks of an existing colleague?

RQ2. Can AI-based code generators affect the performance (completion time, correctness) of technical onboarding or technical stack switch tasks?

RQ3. Can AI-based code generators affect the technical integration efforts of a mobile development team?

5.1 Mobile Development Requirements

To be proficient in a native mobile development team, for any of the two platforms, a developer must have:

1. a good knowledge of the specifics of the programming language syntax and fundamentals. This includes knowledge of variables, constants, data types, operators, control flow structures (such as if statements, and loops), functions, closures, and error handling. Also includes a standard library of data types (String, Date, Numeric, etc.)
2. a good knowledge of collections and data structures: both languages provide a variety of collection types, including arrays, dictionaries, and sets. Developers should understand how to create, modify, and access elements in these collections
3. a good knowledge of the platform-specific frameworks, examples are UIKit, Combine, Foundation (iOS) and Compose, Coroutines (Android).

4. a good knowledge of OOP concepts. Both languages support OOP principles such as classes, objects, inheritance, polymorphism, and encapsulation. One should be able to define and use classes, create and manipulate objects, and understand the concepts of inheritance and overriding.

5. a good knowledge of functional programming: Swift and Kotlin have functional programming capabilities, and programmers should be familiar with higher-order functions, immutability, closures, and functional operators (map, reduce, filter, etc.)

6. knowledge of asynchronous programming: Swift and Kotlin offer similar mechanisms for handling tasks concurrently, and handling background tasks.

7. capacity to switch platforms easily. This is a characteristic that is rarely encountered in native mobile developers.

The tasks for both challenges involved in the study are designed to measure these capabilities.

A quick comparison between Kotlin and Swift is required at this point hence these two modern programming languages have a lot in common and the process of encouraging technical switches must take advantage of these similarities. Even if used on two different platforms, these two share the following elements:

- Modern, expressive, and concise syntax, enhancing developer productivity and code readability.
- Type inference and focus on compile time safety.
- Nullability/Optional Types providing support for similar representations in domain layers, highly significant in sensitive areas like Banking and Financial projects.
- OOP support including abstract class / protocol-oriented programming.
- Functional programming support including dedicated frameworks.

These shared concepts facilitate quick knowledge transfer between the two code bases, making it easier for developers to transition between iOS and Android native development after being proficient in one of the languages. The concordance between Swift and Kotlin helps technical leads streamline the development process, and encourages similar practices across the two platforms. Of course, developers must pick the best solutions for their specific platform.

5.2 Designing the Study

In order to analyze the impact of AI-assisted programming in a native mobile development environment, the study was designed to compare the output of development

problems with and without the assistance of AI systems for each phase: technical onboarding, and technical stack switch. The study is separated into these two critical phases in the existence of a mobile team/department as reflected in Figure 5.1. Information about tasks, participants, measurements, and results will take into account each of the phases.

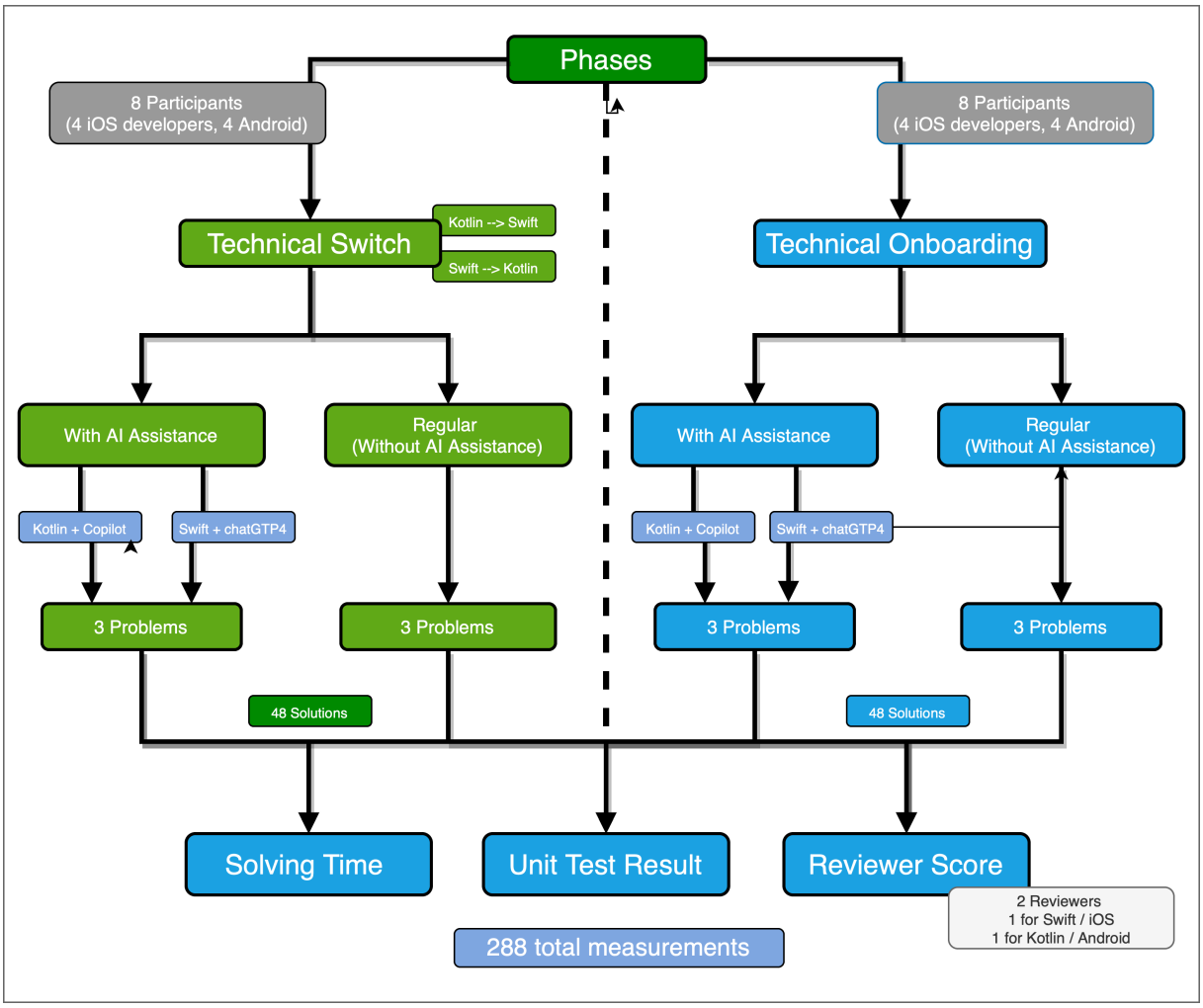


Figure 5.1: Overview of the Study Design.

5.2.1 Procedure

Each participant must resolve two sets of problems. The first set of problems contains one problem of each difficulty level without using any AI-assisted tools with access to classic online resources (documentation, StackOverflow, search engines). An example of what the participants received can be found in Figure D.1. This ensures the control conditions and the baseline required for the study. The second set of problems also contains a

problem of each difficulty but none of these problems are found in the first set. Participants must solve this second set using GitHub Copilot or ChatGPT without access to anything else outside of documentation. This ensures the main experiment conditions. Details like the three different levels of difficulty: easy, moderate, and increased, are explained in the following sections. See Figure 5.2 for a graphical representation.

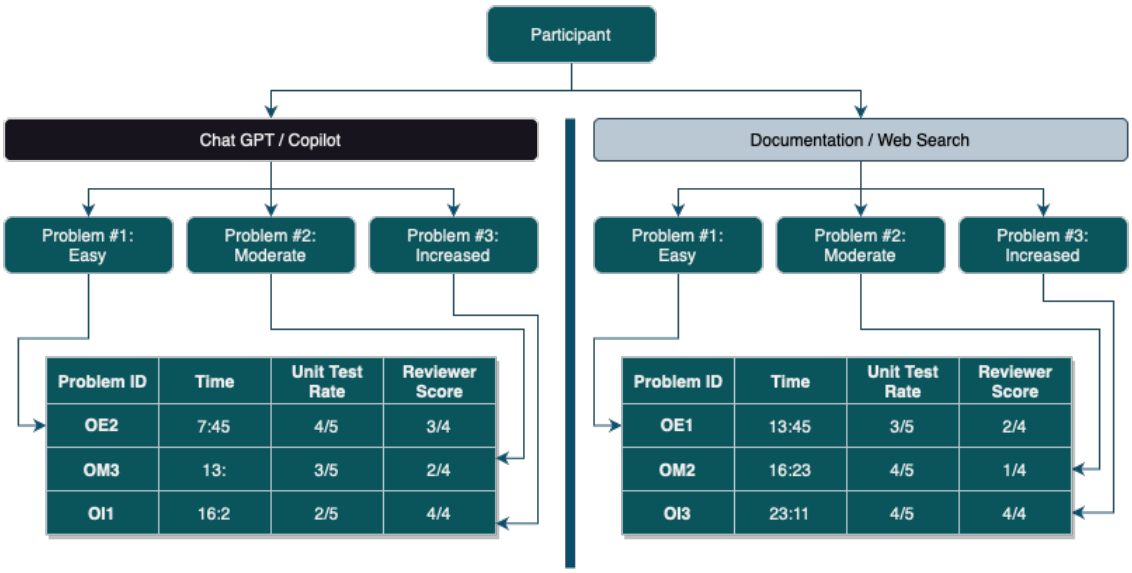


Figure 5.2: Breakdown of a participant output in any of the phases (Onboarding or Technical Stack Switch).

The case study involved two main phases: Technical Onboarding and Technical Stack Switch. In the Technical Onboarding phase, the AI tool used for Kotlin was GitHub Copilot, while for Swift, it was ChatGPT. There were a total of 9 distinct problems, 8 participants, and each participant worked on 6 problems. Out of the 6 problems, 3 were solved using the respective AI tool. In total, 48 solutions were generated, and each solution had 3 resulting metrics. The same AI tools were used in the Technical Stack Switch phase, and the metrics remained consistent. See Table 5.1

Table 5.1: Numbers involved in the Case Study.

	Technical Onboarding	**Technical Stack Switch**
AI Tool Used for Kotlin	GitHub Copilot	GitHub Copilot
AI Tool Used for Swift	ChatGPT	ChatGPT
Total number of distinct problems	9	9
Number of participants	8	8
Number of problems per participant	6	6
Number of problems using AI Tool	3	3
Total Number of solutions generated	48	48
Number of metrics resulted per solution	3	3

Phase Type	Participant ID	Problem ID	Difficulty Class	Language	Time	Unit Test Rate	Reviewer Score	AI Tool
TO	P012	OE2	E	Swift	7:45	4/5	3/4	None
TO	P015	OM3	M	Android	13:20	3/5	2/4	Copilot
TS	P017	OI1	I	Swift	16:30	2/5	4/4	ChatGPT4

Figure 5.3: Exemplification of the data accumulated as a result of the case study. Possible Phase Types: TO = Technical Onboarding, TS = Technical Tech Stach Switch. Difficulty Classes are E = Easy, M = Moderate and I = Increased.

The investigation is conducted by utilizing an environment that possesses the subsequent characteristics and attributes:

- Tasks are resolved in a mobile-specific environment, Android Studio IDE for Android or Xcode for iOS.
- For tasks that do not imply the usage of AI-assisted tools, the developer is free to use anything outside of these tools.
- For tasks that do imply the usage of AI-assisted tools, the developer is not allowed to use anything outside of the tools and documentation provided by the IDE or mobile platform. Only the usage of GitHub Copilot and ChatGPT is allowed.
- Each task is done under the supervision of a reviewer for data collection purposes which includes the timing of each task and developer code writing behavior.

- Output is collected by the reviewer in the form of source files, one file per task. These files are used in the construction of results.
- Participants are requested to fill out a form for gathering qualitative results.
- The time spent by each participant should not be more than 4 hours in total.

The metrics involved and the technical information associated that are integrated into the Results are presented in Table 5.2 and a simplification of the results format can be seen in Figure 5.3.

Table 5.2: The technical information that is integrated into the Results

Duration	Time interval for solving the task. If time expires, the participant can no longer contribute and the solution is provided as it is.
Correctness - Unit Test Results	Unit tests created by a reviewer based on problem-specific test cases. Aims to assess the correctness of the output and fitness of the function. For normalization factors, each problem has 5 test cases defined. These test cases aim for maximum correctness coverage. For each solution, actual unit tests are created from use cases and we retain the number of unit tests passed. Mainly, the score for each solution will be a figure from 0 to 5 representing the number of unit tests passed.
Integration - ReviewerScore	This consists in a qualitative marker that encapsulates the usage of platform APIs, the usage of most modern features of the language, and the verbosity of the code compared with the industry standards of the team which follows the Clean Coding principles (ref to Clean Coding book). Labels are Poor, Fair, Satisfactory, and Aligned. See Table 5.3 for details. An example of a poor ReviewerScore can be seen in Example F.0.1

Table 5.3: The 4 levels or ReviewerScore

Poor	Meaning the code violated all 3 characteristics of the ReviewerScore.
Fair	Meaning the code avoided at least one type of violation. This can be corrected using relatively low effort by a mentor
Satisfactory	Meaning the code only violated one type of criteria. This can be even without the need for supervision.
Aligned	No violations of the ReviewerScore characteristics were found.

One extra type of information collected after the technical tasks are completed involves the completion of a form with a set of queries designed to extract user perception of the tasks. We will call this form PostTaskSurvey 5.4. In this survey, participants were

invited to add their own observations for each item and these can be found in the appendix section G.

Table 5.4: The items of the Post Task Survey.

PTS1	Rate the helpfulness of the AI-assisted tools (GitHub Copilot/ChatGTP) on a scale from 1 to 5. 1 representing it didn't help in any manner, 5 meaning it solved the problem from the first interaction with it.
PTS2	Rate the level of understanding of the AI-assisted code on a scale from 1 to 5. 1 representing the participant didn't understand any instructions provided by the tool, and 5 meaning all instructions were clear.
PTS3	Rate the level of confidence you have in the final code on a scale from 1 to 5. 1 represents you aren't confident, and 5 means full confidence that the code is functionally correct and respects the coding guidelines of the platform.
PTS4	Rate the level of expected future usage of the AI-assisted code tools on a scale from 1 to 5. 1 being the participants don't see any future usage of the tools, 5 meaning that all future tasks will involve usage of AI-assistants.

5.2.2 Tasks

For a business area as demanding as Banking and Financing, mobile applications involve different levels of complexity and a variety of tasks. It is notable that developers are technically onboarded on a team, with a set of exercises that highlight these types of requirements. It is a means to communicate to them the expectations and to motivate them to be focused on constant improvements. The same aspects apply when assisting a developer to switch technical stacks, from iOS to Android and vice-versa.

We must consider that this study involves a simplification of what occurs in a standard technical onboarding process. In a real process, this procedure takes from 2 weeks to 4 weeks, involves many more tasks, and implies providing solutions for small-scale projects or even full feature implementations. The same applies to a technical stack switch, which usually involves gradual transition and procedures that can take from 1 to 3 months. The study attempts to simulate tasks given to developers involved in these situations, meant to help them achieve technical alignment with the team. We are proposing three levels of difficulties in this study that are described in Table 5.5

Since one of our goals is to specialize developers in the specifics of native mobile development, we require the usage of 'platform-specific representations' and APIs. This implies using the specialized type that the platform has provided for that representation. Examples: "abstract class Number" for Kotlin and "Numeric Protocol" in Swift for numbers, or "Date Structure" in Swift and "Date Class" in Kotlin.

Table 5.5: Difficulty levels for tasks involved in the study.

Level	Description	Time
`Easy`	Implying problems that do not assume expertise in the specific mobile platform.	10 minutes.
`Moderate`	Implying problems that require some minimal research effort to be solved.	20 minutes.
`Increased`	Implying problems that require a level of solution preparation or design.	30 minutes.

Tasks are of three types of difficulty for each phase of the study. The Time column indicates the time constraint for each type of level

Technical On-boarding In a software development department, mentorship starts with a proper technical onboarding procedure. This procedure usually involves setting up a set of problems or tasks that help the new team member calibrate from a technical perspective, aligning him/her with expectations related to correctness and technical integration. Aspects like code structure, style, testability, and verbosity are verified and discussed in this process. Depending on the experience of the person who is being onboarded, this procedure can require a high effort of effort from an existing team member. In departments that are fully emerged in project-related tasks, it would be useful if some of the effort is 'delegated' to the AI assistants. This type of approach can also enhance the concept of a 'self-sufficient' developer which relies less on the team for tasks of medium to low complexity.

This type of study and applicability is significant since the goal of any team is to achieve cost-effectiveness in developer onboarding. AI-Assistants have the potential to guide the subject thru the process with reduced feedback from mentors increasing independence and self-reliance and ideally knowledge. One example of reduced feedback from a mentor is "Try to make the code more language idiomatic.". This seems like an abstract request but with the help of an AI-Assitantant that can modify or regenerate a piece of code to respect this, the results would ideally come faster.

One other challenge in the industry is the friction between approaches and communication between mentor and person onboarded. Not all developers react the same to a standard process of technical onboarding. The type of solutions presented in this paper can accommodate different ways of learning and provide some flexibility in the process. The aim is to reduce team efforts on the integration of a new member into the regular tasks of a team or department.

The last major criterion is the scalability of such a process. Can a Coding-Inteligence assistant help a team grow fast respecting a high bar related to technical onboarding? Therefore it is important for a mobile department, to have efficient technical onboarding and in this area, AI-assisted programming can contribute.

We must specify that usually, technical onboarding tasks should focus on the usage of platform-specific APIs to encourage making the code resilient to SDK updates. Also, the

technical onboarding tasks should revolve around the business sector that is the main focus of the department or team, specifically, the requirements should be similar in terminology and scope to the tasks usually occurring in a project. This would minimize the friction of the onboarded developer during the transition to an ongoing project.

For the technical onboarding section, the tasks are as follows, grouped by levels of difficulty can be found in the Appendix E.

Technical Stack Switch It is estimated that as of 2020 there were 5,9 million Android developers and 2,8 million iOS developers in the world. If a mobile department is not capable of facilitating the transition from Android development to iOS (or vice versa) it will lose ground compared to other departments. This doesn't necessarily apply only to the mobile sector. This concept can extend between the ReactNative technical stack and the native one (and vice versa) and also to web/frontend technical stack, like Angular, to native mobile stack, and of course, the other way around. This research focuses only on the impact of AI assistance in the facilitation between the two major native mobile technical stacks.

There are a small number of mobile developers that can be proficient in both iOS and Android. The ones who do are of high importance to a mobile department focused on native applications in a complex domain like banking. This is due to their knowledge to facilitate a fundamental principle mentioned previously in the paper: having consistency between approaches in the two code bases that require maintenance. Therefore a goal of a mobile department is to have as many developers proficient in both platforms. Using AI-assisted programming in facilitating the switch can be highly productive.

Android and iOS SDKs, Kotlin and Swift, have strengths and weaknesses. By focusing on a good technical stack switch culture, a mobile team encourages a developer to be more versatile, and adaptable. In this type of switch, a participant can adopt broader concepts in Software Engineering and increase his/her skills in the generic capability of such a profession: problem-solving.

This phase assumes some knowledge of the mobile programming requirements. More information regarding this can be found in the Participants section. The aim is to reduce team efforts on the integration of a new member in another technical stack.

For the technical stack switch section, the tasks can be found in Appendix section E, grouped by levels of difficulty.

5.2.3 Participants

For each of the two areas of the study, we have 8 participants, having a total of 16 participants, assisted by 2 reviewers. Participants are members of the mobile department part of the Software Development company, situated in Cluj-Napoca, Romania. The department is specialized in providing native applications and solutions, like frameworks or consultancy in sensitive areas like Banking and Financing. For both phases, 8 developers

participated in the study, and for data collection, the procedure implied the usage of 2 reviewers.

Technical On-boarding. The participants are selected from the members that joined the mobile department in the last year. Only members with less than 3 years of experience participated, in order to avoid large impacts related to previous relative experience in the field. All participants finished a Computer Science faculty and have basic knowledge in Software Development and have previous interactions with mobile development SDKs.

The two reviewers are Software Engineers that also share Technical Leadership responsibilities and they participate in the construction of technical onboarding procedures in the mobile department and provide consultancy in other software departments.

It's important to note that the participants were subject to the department's technical onboarding procedure when they joined the company but they didn't encounter the type of tasks used in the study.

Technical Stack Switch The participants are selected from the members that are active in the mobile department. Only members with more than 3 years of experience on their respective technical stack (Android or iOS) participated in the study. All participants finished a Computer Science faculty, already accumulated industry experience, and are familiar with current industry and department standards. The same reviewers as in the Technical Onboarding were involved. All participants have minimal understanding of the sibling platform and never had official responsibilities to resolve tasks outside of their technology stack.

	A	B	C	D	E	F	G	H	I
1	PhaseType	Participant ID	Problem ID	Difficulty Class	Language	Time	Unit Test Rate	Reviewer Score	AI Tool
71	TS	P012	SE3	Easy	Swift	167	3	2	Chat …
72	TS	P012	SM3	Moder…	Swift	237	4	2	Chat …
73	TS	P012	SI1	Increa…	Swift	403	4	2	Chat …
74	TS	P013	SE3	Easy	Kotlin	900	4	2	None
75	TS	P013	SM1	Moder…	Kotlin	1800	1	2	None
76	TS	P013	SI1	Increa…	Kotlin	1800	0	2	None
77	TS	P013	SE1	Easy	Kotlin	420	3	3	Github…
78	TS	P013	SM3	Moder…	Kotlin	600	4	4	Github…
79	TS	P013	SI2	Increa…	Kotlin	540	2	3	Github…
80	TS	P014	SE1	Easy	Kotlin	450	3	3	None
81	TS	P014	SM3	Moder…	Kotlin	810	4	3	None
82	TS	P014	SI2	Increa…	Kotlin	1380	4	2	None
83	TS	P014	SE2	Easy	Kotlin	160	3	3	Github…
84	TS	P014	SM2	Moder…	Kotlin	190	3	3	Github…

Figure 5.4: Preview of the raw results file (.xls).

5.3 Results

In this section, we examine the metrics gathered in the two phases of the study. We will use the association between the three metrics, duration, correctness (unit test), and integration (ReviewerScore 5.3) in association with the usage of lack of AI Tools, programming language, and problem complexity. We will attempt to draw conclusions related to the utility of the tools in a software development team specialized in mobile applications. The raw format (.xls) of the gathered data can be found in the public repository associated with this paper [56]. A preview of how this looks can be seen in Figure 5.4

5.3.1 Quantitative

Time Participants using AI Tools for Technical Onboarding Tasks had a 35% decrease in the duration of the task and a 45% decrease for Technical Switch tasks in the case study environment that was designed to mimic problems that are usually involved in mobile departments, as shown in Table 5.6. For the Technical Switch phase, which contains problems that require knowledge in mobile development, the average time spent on a task was approximately 16 minutes, and with the usage of AI Tool, the average is lowered to 9 minutes as can be observed in 5.5.

Table 5.6: Total and average measurements in seconds.

	Tech Onboarding		Tech Switch	
	Sum	Avg.	Sum	Avg.
Without AI	13.431	560	22.964	957
With AI	8.657	361	12.550	523
Accumulated	22.088	460	35.514	740

For easy tasks, we observed little difference between AI-Tools used and when not, but this changed for moderate where the time spent without an AI-Assistant was doubled on average, and for increased tasks, approx. 40% more time was spent when participants didn't use AI-Tools and relied on regular sources like Google Search or StackOverflow besides the platform documentation. These differences can be seen in Figure 5.6 which combines data from Onboarding and Technical Switch phases.

There is a slight difference between the two platforms, iOS and Android, and their corresponding programming languages and AI Tools, ChatGPT and GitHub Copilot. The candidates onboarding or switching to iOS solved the tasks faster without assistance, one reason for this difference can be explained by the fact that, in the Technical Switch phase, the Android developers can adapt to iOS development faster than iOS can to the Android echo-åsystem and the Kotlin language. No major difference was seen in the average time of the AI-Assistance usage with respect to the programming language and the corresponding AI-Tool, as seen in Figure 5.7.

Finally, we noticed that without an AI Tool, the chances that a solution is not provided or provided after time expiration increased. As visible in 5.8 only in one case, a candidate crossed the 20 minutes threshold using AI Tools and without tolls, 8 solutions exceeded the threshold

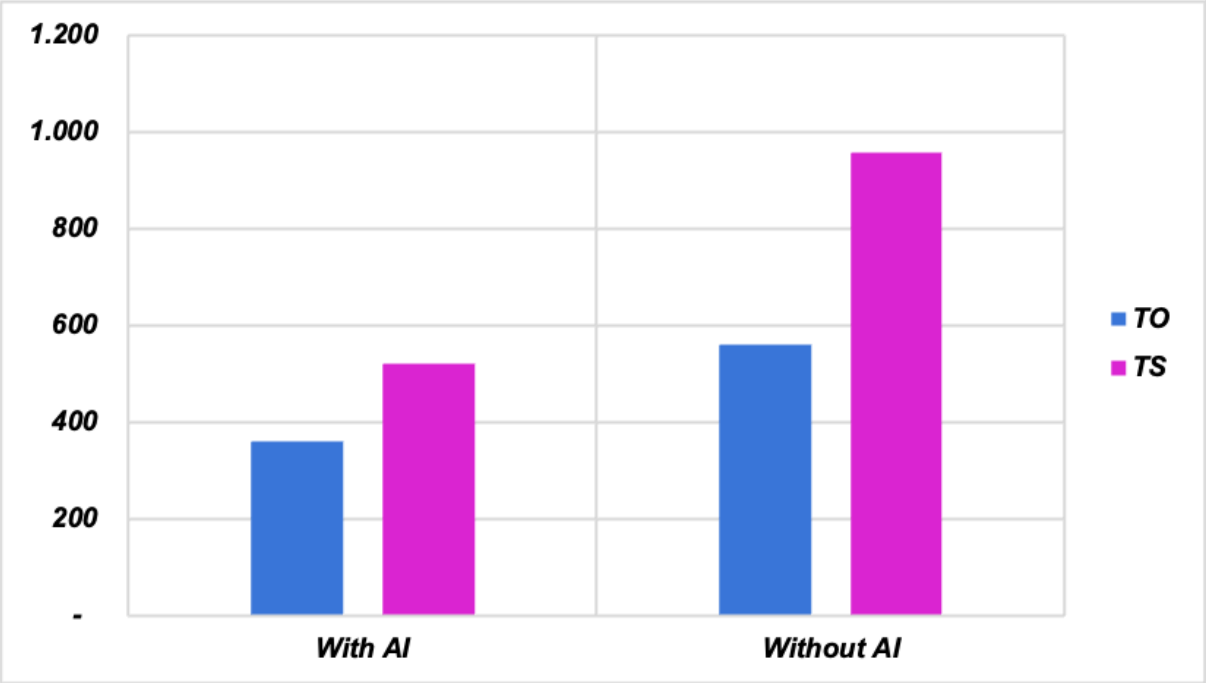


Figure 5.5: Average time spent on tasks, with indications related to the usage of AI tools. Time (y) is represented in seconds. TO = Technical Onboarding, TS = Technical Tech Stach Switch

Correctness and Technical Integration We measure correctness via unit tests and technical integration using the ReviewerScore 5.3. On average unit tests for the task, onboarding had a higher score for the technical onboarding task, 3.83/5, than with AI-Tools 3.54/5, on which we can asses that tasks that are usually low-medium in complexity and the majority of developers can asses the correctness risks and provide proper handling for it without the assistance of an AI tool as seen in Figure 5.9. A lower grade on AI-Tool can be explained by the decrease in ownership and understanding on the side of the candidate, in the solution generated using such a tool. Figures can be consulted in Table 5.7.

On the other hand, in the technical switch phase, unit test scores were higher for AI-Assited solutions and this can be explained by the nature of the models which are already trained in the required programming language and can easier assess its particularities like nullability or error handling which can lead to lower correctness score. Developers that are new to a programming language might require more time to achieve this understanding.

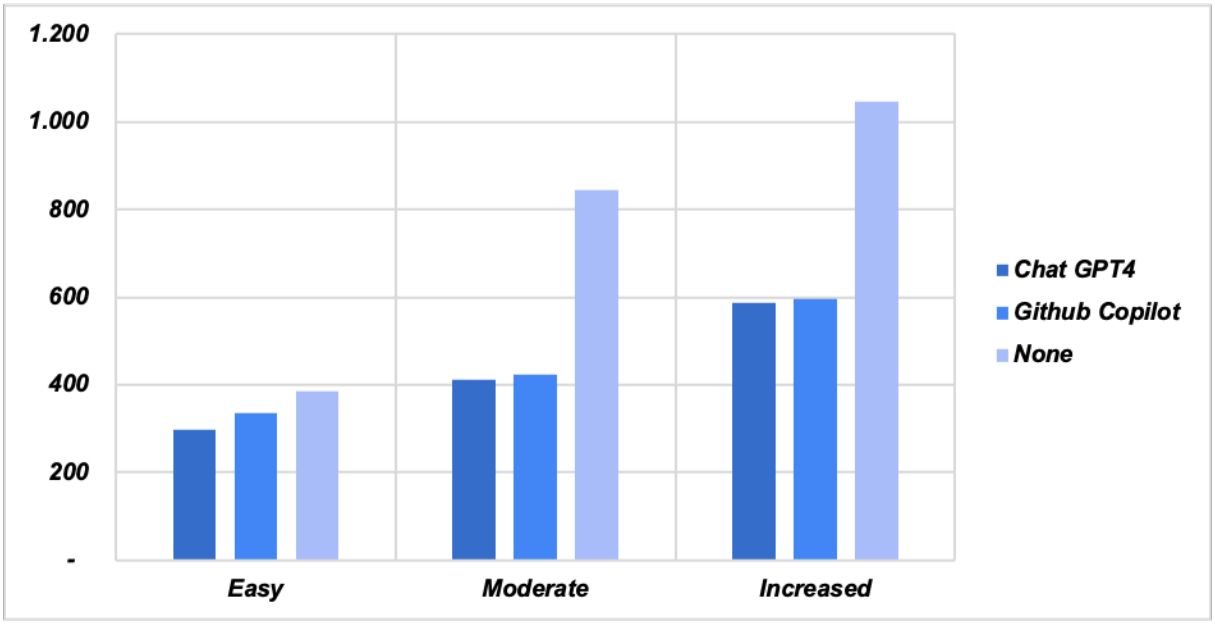


Figure 5.6: Average time spent on tasks, for each type of difficulty and with bars for each AI Tool used, or None for solutions without AI-Tools. Time (y) is represented in seconds.

Table 5.7: Averages of Unit test scores and ReviewerScore.

	Tech Onboarding		Tech Switch	
	Unit Test	ReviewerScore	Unit Test	ReviewerScore
Without AI	3,83	2,75	3,13	2,29
With AI	3,54	2,21	3,25	2,46

With respect to ReviewerScore, we have noticed a similar pattern as seen in Figure 5.10. On Technical Onboarding, a phase in which developers are already accustomed to the platform and programming language, solutions without AI-Assistance received better scores. On the other hand, the ones in Technical Switch favor AI-Tools usage, and this can be explained by the same reasoning, developers transitioning from one technology stack to another tend to be biased to the known stack and programming language and replicate approaches from that which are not in line with the standards of the new tech stack.

We gathered all quantitative metrics into one Figure 5.11 regardless of the type of problem, and based on the data supporting it, available in [56], we can extract that time is significantly reduced in both mobile programming languages if AI-Assistance is used for code generation, unit tests score have relatively minor differences, both Swift and Kotlin delta is approx. 1% and ReviewerScore show different variations based on the programming language, Swift showing a better score for solutions without AI Tools, and Kotlin solutions without AI-Tools having an 8% increase.

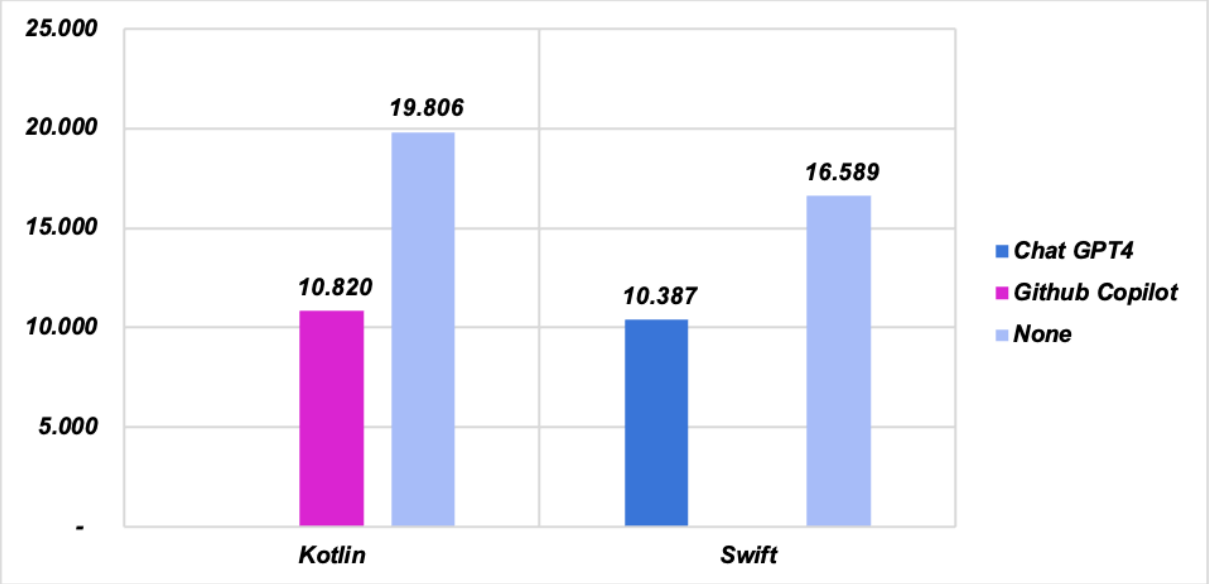


Figure 5.7: Average time spent on tasks, for each type of language involved and with bars for each AI Tool used, or None for solutions without AI-Tools. Time (y) is represented in seconds.

5.3.2 Qualitative

Adopting the AI Tools Candidates for technical stack switch had difficulties finding the correct sources to provide information about how to start, the APIs to be used, and language features due to a lack of experience in this type of data gathering. The sessions that implied AI had a quicker starting point since the role of these assistants is to aggregate data. It's essential to consider that this was a time-limited study and in a real technical switch or onboarding procedure, time is usually not a factor of pressure and developers do not need to quickly run after information, on the other hand, the tasks are usually more complex.

In the same area, the candidates that had increased difficulties with the technical switch, represented by 0 unit test scores, or long times for solutions, had better results time-wise and correctness-wise with AI. It was observed that interactions with the ChatGPT tool were in general faster and provided more satisfaction than with the GitHub Copilot.

Quality Prompts When working with a language model, providing appropriate prompts will increase the chances of a good result. An appropriate prompt will provide context and domain adaptation regarding the task at hand. in our context, we need to make sure the context is set to the corresponding programming language and the type of problem we need to solve.

Another aspect is an expression of intent. For example, a quality prompt should contain elements related to the signature of a function and the expected result. Specificity

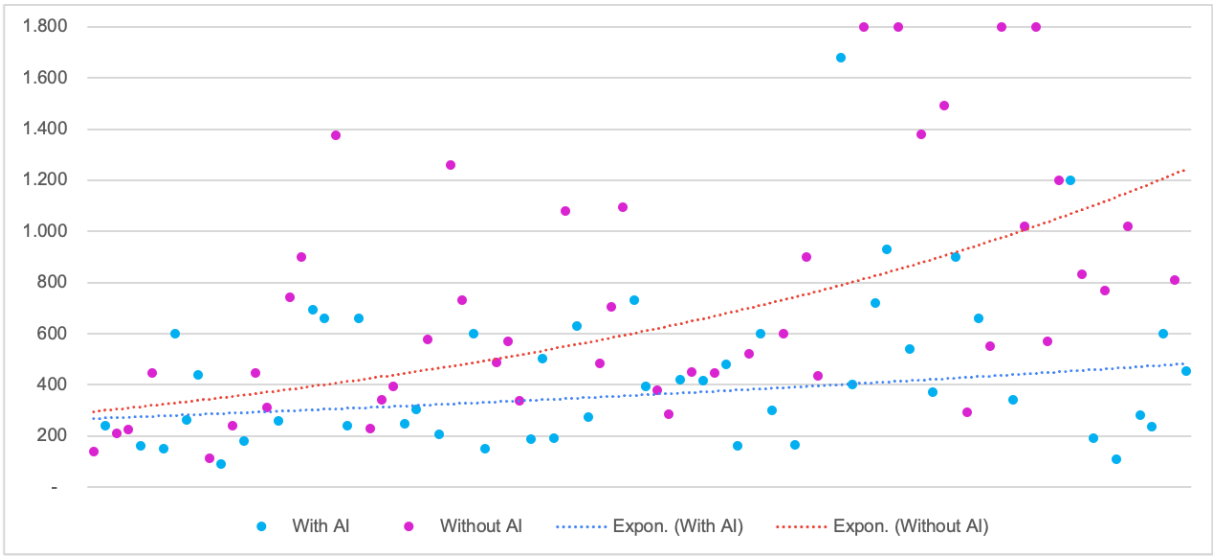


Figure 5.8: Exponential trend-line in time for issues resolved with or without AI Tools. Time (y) is represented in seconds. Time (y) is represented in seconds.

is also a fundamental factor, which can be expressed by providing clear language types to be used in libraries or even design patterns. The problems in our case study were engineered with a low level of specificity, for two reasons, to avoid relying solely on AI tools and to mimic the industry procedure which explicitly avoids concrete details to evaluate the capacity of a developer to fill in the gaps in requirements with his/her own analytic capabilities. Candidates that were more attentive at modifying the prompts used in communication with the model have higher scores in correctness and ReviewerScore. Examples related to 'prompt engineering' extracted from the case study observations can be found in the Appendix section, "Prompts without context" F.0.1 and "Prompts containing technical details" F.0.3

Incremental Prompts A rewarding behavior extracted by the technical reviewers of the interactions with AI-Assisted tools was the incremental improvement of the solution by breaking the solution into multiple prompts or altering the provided solution. Especially in the Technical Stack Switch phase, where candidates had already some experience on how to solve issues, solutions were improved by breaking the task given to the AI Tools into multiple smaller tasks or asking it to recreate the solution. Examples of incremental improvements using prompts, extracted from the candidates, can be found in the Appendix section, for Copilot F.0.4 and ChatGPT F.0.5.

Platform Association in Technical Stack Switch . We can assume the candidates of this phase know how to solve the received problems in the technical stack they have experience with. As reviewers, we observed if this knowledge benefits the overall result,

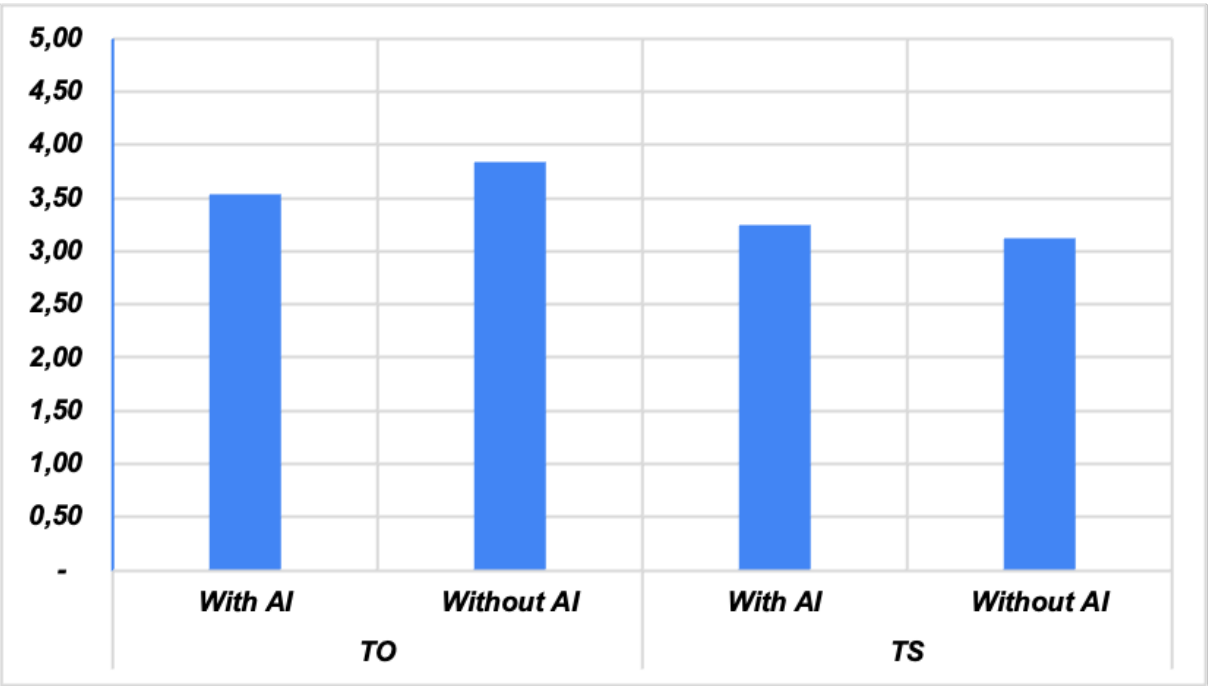


Figure 5.9: Unit test score for all 4 major categories of the study. Time (y) is represented in seconds. TO = Technical Onboarding, TS = Technical Tech Stach Switch

especially when using AI-Tools. Candidates that had to solve Technical Switch tasks in Swift using the iOS SDK, had the 'capability' to interact with ChatGPT which can also provide natural language solutions related to code. A few candidates used this support to learn the equivalent of what they already knew on the platform in which they had experience already. See Example F.0.6. For the iOS / Swift developers converted to Android / Kotlin ones, for this study, this was more difficult because Copilot has reduced capability in association with languages outside of one of the projects. The experience of the candidates helped in this case since documentation can easily provide the API for a given need and afterward, this can be used in a more accurate prompt.

Post Study Participants Survey From the survey that all the 18 participants were asked to fill out, 13 (72%) of the participants rated the helpfulness of the AI-Tools as high 5.12 and 14 of them (77%) stated that they understood the solutions provided by the AI-Tools 5.13. Regarding the confidence that the candidates had in the final output of the code, 12 (66%) gave high confidence votes with 11 choosing second the maximum 5.14. Of all the participants, 12 (66%) stated that will rely on AI-Assisted code generation tools in the expected future, with the rest mostly being uncertain at this point 5.15.

From the given feedback extracted from the participants, which can be consulted in Appendix G, we extracted some recurring ideas:

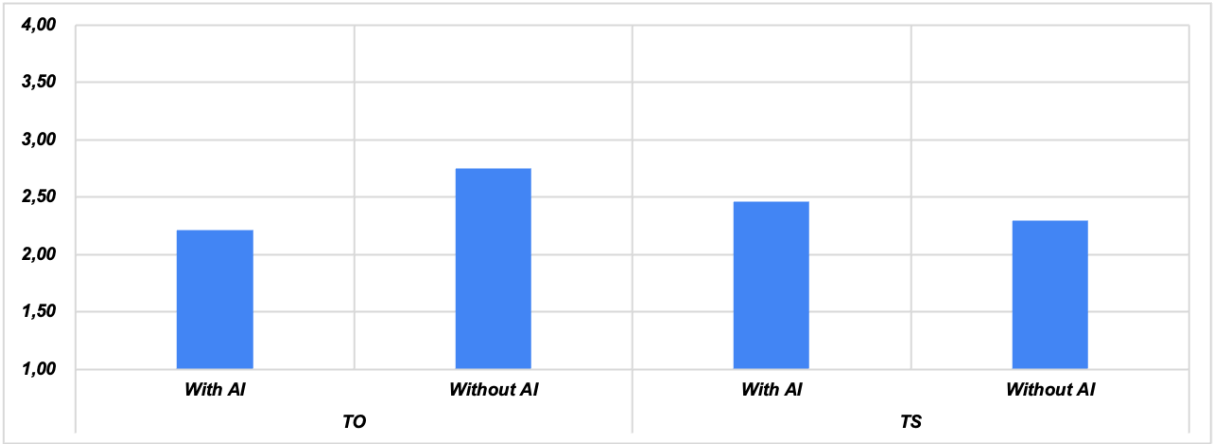


Figure 5.10: ReviewerScore for all 4 major categories of the study. Time (y) is represented in seconds. TO = Technical Onboarding, TS = Technical Tech Stach Switch

1. Related to helpfulness, most of the participants appreciated the quick access to solutions, the friendly interaction with this type of source of information, and the fact that it provided explanations for code in some cases, but also remarked that the solution provided needed a level of adjustments from the user.

2. Related to the level of understanding of the provided solutions, most participants mentioned that solutions were intuitive and clear and this can be improved with the experience of the user/developer, nevertheless some noticed that in particular cases, solutions were out of context.

3. Related to the level of confidence in the solution, most of the participants had a high level of confidence in functionality but a lower level related to the coding guidelines and efficient usage of language and platform features. Some participants reported errors even for simpler tasks and clear prompts. The consensus is that for small and clear tasks, solutions can be considered reliable.

4. Related to the usage of such tools in the future, participants see benefits in AI-Assisted programming and observe the opportunity to use them, particularly in small, repetitive tasks related to boilerplate code, some being skeptical about the advantages of using it on feature development that usually involve complexity.

5.4 Discussion

How can an AI-based code generator affect the experience when onboarding a new team member or switching technical stacks of an existing colleague? For these particular types of procedures in a mobile development team, meaning providing technical assistance

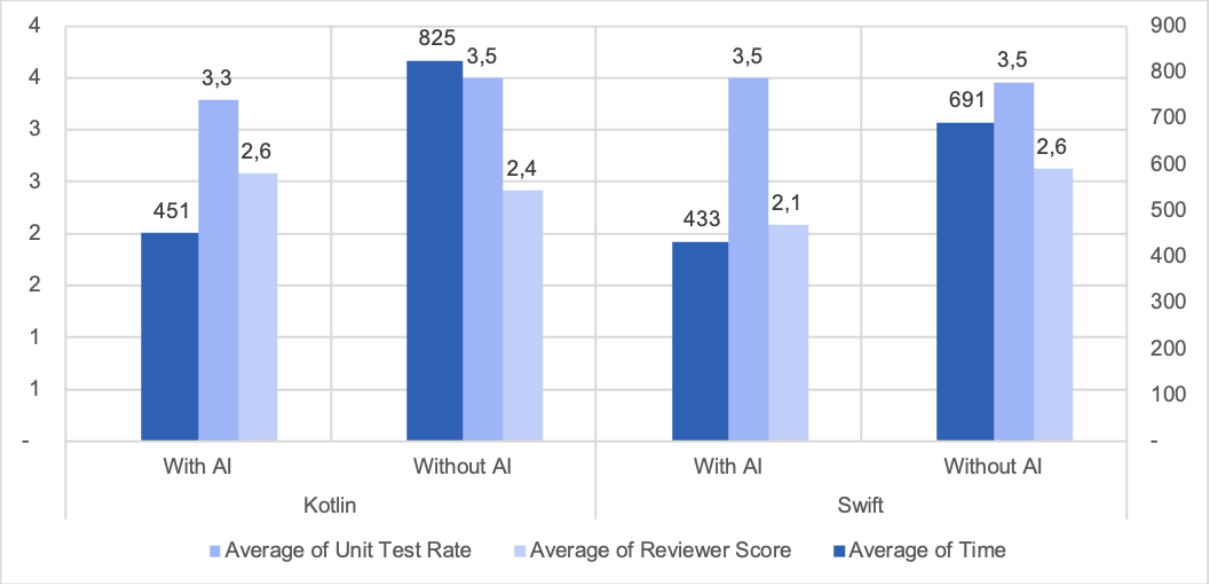


Figure 5.11: Average Time, Average Unit Test Score, and Average Reviewer Score associated with mobile programming languages.

for a new member or assisting a colleague to adopt the sibling technical stack, we can asses based on the results of the study and the participant's feedback, that AI-Based code generators can improve the experience by providing an additional source of information which if combined with developer experience and analytic thinking can lead to rewarding results for both the subject of the procedure and the mentors assisting the procedure.

Can AI-based code generators affect the performance (completion time, correctness) of technical onboarding or technical stack switch tasks? Results of the study reflect a clear improvement in the duration of achieving tasks, in a simulation of the two types of procedures, without major implications related to the correctness of the solutions provided. This can be seen as a positive effect on performance and it can be considered reasonable for mobile teams to adopt such solutions with the consideration that the long-term learning and information retention of the members must be measured before scaling the process to a standard one. The regular research, done before starting to write a solution, was reduced by the usage of AI tools, with some candidates doing API validation directly using the generated code, increasing the interaction with documentation.

Can AI-based code generators affect the technical integration efforts of a mobile development team? The design of the study considers the notable factor of technical alignment with the requirements of a team, by the addition of the ReviewerScore, in order to measure the impact of integration efforts. Results show that the impact of using such tools can lower the alignment due to reliance on a tool and less on the team's context and requirement, still, the risk can be taken into consideration when designing procedures for technical onboarding or technical task switch. A key piece of information to note is that

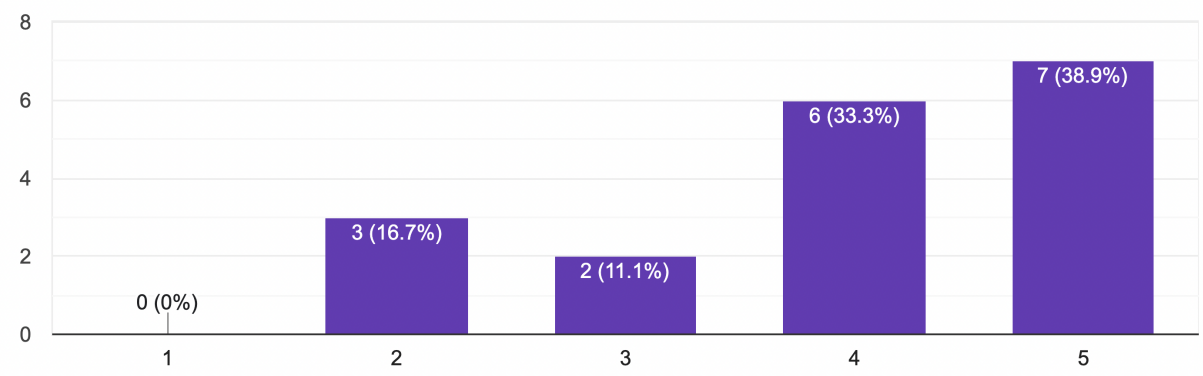


Figure 5.12: Result to PTS1: Rate the helpfulness of the AI-assisted tools (GitHub Copilot/Chat GTP) on a scale from 1 to 5. 1 representing it didn't help in any manner, 5 meaning it solved the problem from the first interaction with it.

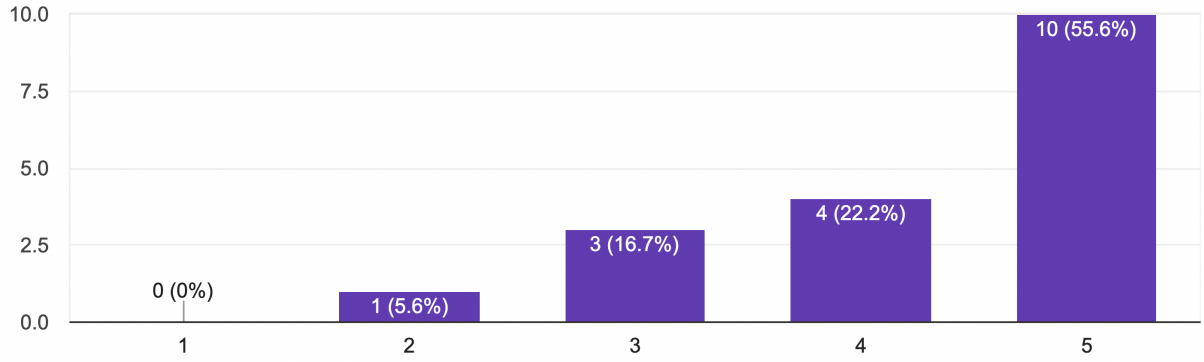


Figure 5.13: Result to PTS2: Rate the level of understanding of the AI-assisted code on a scale from 1 to 5. 1 representing the participant didn't understand any instructions provided by the tool, and 5 meaning all instructions were clear.

tools like GitHub Copilot can adapt to a specific project or team in time. Alternatives to increase chances of alignment are building a dedicated model, like IVA-CodeInt presented in Chapter 4 which can be tailed to the needs of the team or project. A team can also consider taking into solutions similar to the one presented by Disha Shrivastava et. al [36] in which the prompt proposals take context from the entire repository, thereby incorporating both the structure of the repository and the context from other relevant files (e.g. imports, parent class files).

Other aspects that were observed during the study. Effective prompt engineering is an attribute that was again observed to affect the results and understanding of the problem and its solution in a significant manner. We must note that candidates that were more attentive at modifying the prompts used in communication with the model have higher scores in correctness and ReviewerScore. As observed by Yongchao Zhou, Andrei Ioan

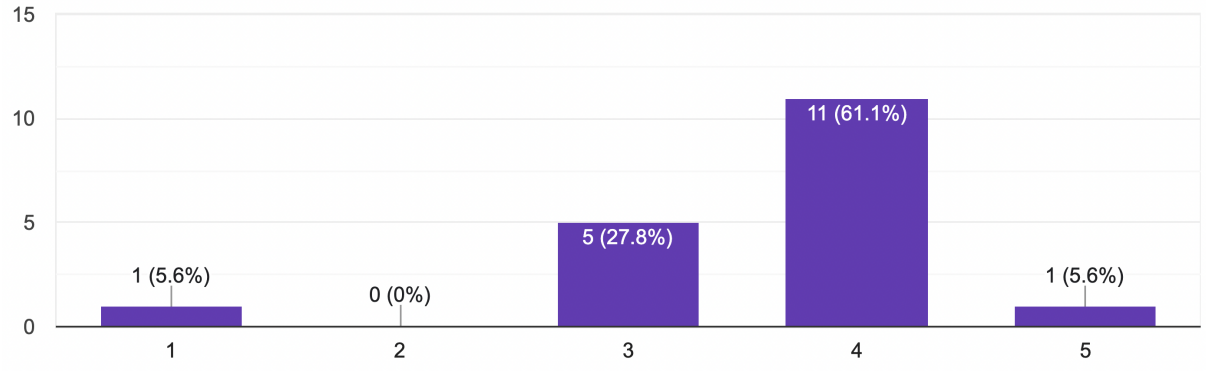


Figure 5.14: Result to PTS3: Rate the level of confidence you have in the final code on a scale from 1 to 5. 1 represents you aren't confident, and 5 means full confidence that the code is functionally correct and respects the coding guidelines of the platform.

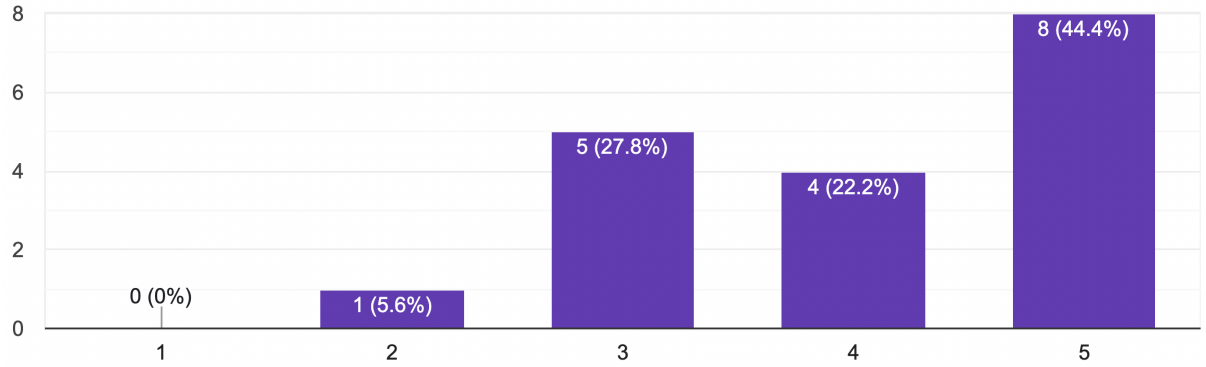


Figure 5.15: Result to PTS4: Rate the level of confidence you have in the final code on a scale from 1 to 5. 1 represents you aren't confident, and 5 means full confidence that the code is functionally correct and respects the coding guidelines of the platform.

Muresanu, Silviu Pitis et. al [61], task performance depends significantly on the quality of the prompt used to steer the model, and most effective prompts have been handcrafted by humans

The study reflected that transitioning from the iOS (Swift) technical stack to the Android (Kotlin) technical stack can require more effort with and without using the AI Code Generators. Candidates reported that GitHub Copilot has low support for detecting incorrectly generated code, and requires corrections from developers but it is expected that the behavior of the Tool increases with the understanding of the user and context.

Chapter 6

Conclusions

6.1 Contributions

In this work, we presented a methodology for creating a dataset of value to LLMs designed for tasks related to code generation for a specific programming language. Using the methodology, we created and published, adhering to FAIR principles, datasets for Kotlin and Swift, languages used mainly in mobile development. We used the resulting datasets to train from scratch a GPT-2 model with the purpose of generating Swift and Kotlin, platform-specific code, based on prompts. Finally, the paper elaborates on a Case Study of using models for code generation in actual requirements of important value for a mobile development team. The study evaluated the impact of AI-Code-Assistants in the process of onboarding a new developer inside a team, from a technical point of view, and in the process of a developer switching technical stacks, in this case, from Android to iOS development or vice-versa.

The paper originated in the actual needs of an actual market sector, such as a mobile development team, to use LLM's capability to generate code in order to improve the output of some of its development phases. The paper, together with its analysis, contains applicable elements to fulfill these types of needs. The paper also aims to contribute to the academic sector and open-source community by focusing on how these types of tools intersect with niche languages like Swift and by publishing the artifacts such as the datasets created and the models trained together with all the information associated with them.

Methodology for creating a dataset One of the results of this work is a methodology for creating and publishing a dataset that can be reused for training or fine-tuning models dedicated to Swift and Kotlin programming languages but the methodology can be adapted to other languages. The steps involved data gathering using GitHub public repositories, curating the raw data using language-specific rules and specialized libraries, and making the datasets available for reusability. The output is several datasets with train and validation splits with two of them having DOI and respecting FAIR principles of publication.

Building a Model for Mobile Languages Another result was achieved by the training of a GPT-2 model from scratch using tools like PyTorch, HuggingFace, and Weights & Biases. Using manual inference we showed that the models are capable of generating code from prompts especially when these revolve around platform-specific APIs. In a practical sense, the model can be used, by a developer, to obtain specific platform sample code when performing a task. Even if the resulting models generated useful code for platform-specific tasks, these did provide not perform as expected for generic programming tasks and were not introduced in the case study that aimed to compare results with and without AI-Assisted tools.

We concluded that training a small GPT-2 model from scratch can be useful to create tools for quick language adoption and in case the authors need to understand and control the entire training process. Still, other solutions are fine-tuning a specialized model for code generation, and using mobile-specific datasets should be explored to obtain better practical results for an objective that implies a business goal. This approach is at the core of Codex, the GPT language model fine-tuned on publicly available Python code from GitHub [11]. This methodology usually requires labeling, specialized datasets, and benchmarks that can be applied to mobile languages but this capability can be considered scarce at this point still. One can also consider applying prompt tunning on a large, generalized model, as recent studies in the natural language processing (NLP) field show that prompt tuning, a new paradigm for tuning, achieves promising results in various NLP tasks as shown in Wang et. all [62].

A business decision on which approaches mentioned above should be used to increase the productivity of a team on a specific market segment like mobile development, in particular phases of development, should be taken taking into consideration the properties of the datasets involved, the available models, paid or open-sourced, the hardware support at hand and the costs associated with training and improvement such a model. This paper argues that training a model from scratch is an accessible option.

Case Study: Using AI-Assisted Code Generation .

A mobile department would technically increase its business value and output if overall productivity is increased, even if only in areas like technical onboarding or technical stack switch, and therefore the incentive of using such a tool, from a business perspective is present at this point. Our study shows that applying AI-Assisted code generators, at least in non-sensitive areas at the beginning can have a positive impact on productivity, with proper adoption policies such as appropriate selection of LLMs models and tools.

Results of the study show a clear improvement in the duration of achieving tasks, in a simulation of industry standard procedures, without major implications related to the correctness of the solutions provided and technical alignment with the guidelines of the team. Effective prompt engineering was observed to significantly affect the results and understanding of the problem and its solution. We observed that candidates that were more attentive at modifying the prompts used in communication with the model have higher scores in correctness and technical alignment (ReviewerScore).

Code generators can assist the concept of a 'self-sufficient' developer that is required in an industry-standard software development team. It does this by reducing the friction between the approaches of different developers and improving communication between mentors and developers by focusing on complexity and solution design rather than coding style or API usage. These types of tools can reduce team efforts on the integration of a new member into the team or an existing one into a new technical stack.

From the survey associated with the case study, we concluded that most of the participants appreciated the quick access to solutions and the interaction with this type of source for information, the majority mentioned that solutions provided by code generators were intuitive and clear but these can be improved by the experience and skills of the developer. A high number of participants reported an increased level of confidence in functionality but a lower level related to the coding guidelines and efficient usage of language and platform features, nevertheless, almost all participants see benefits in AI-Assisted programming and observe the opportunity to use them, particularly in small, repetitive tasks related to boilerplate code.

We can assess based on the results of the study and the participant's feedback, that AI-Based code generators can improve the experience by providing an additional source of information which if combined with developer experience and analytic thinking can lead to rewarding results for both the subject of the procedure and the mentors assisting the procedure.

6.2 Further Development

Methodology for creating a dataset Limitations of the resulting datasets can be tied to the 'staleness' factor of the GitHub dump. Further releases rely on more modern code and adopt the curating rules to the updates for language models will be added. The creation of datasets that can be used as benchmarks for mobile-specific problems with labeling concepts like associated unit tests will be created with developer support in measuring and assisting models with a focus on Swift and Kotlin solutions.

Building a Model for Mobile Languages . For our specific use case, the next gradual steps would be training a larger version of GPT-2, and training larger models like GPT-3 would ensure better results especially if complete validation sets for mobile languages are available. Fine-tuning code-specific models like GPTBigCode, CodeBert, or SantaCoder can also be pursued. Evaluation of the Prompt-tuning on generic LLMs is also a direction of research. The plan of the authors is to continue the research inside the work environment and in future academic phases.

Case Study . Study Design can be extended with larger problems more similar to industry-specific procedures and include follow-up Tests to measure the level of retained knowledge with the usage of LLMs for code generation. The plan of the authors is to

expand this study inside the work environment with the collaboration of the academic community.

Bibliography

[1] "Stackoverflow 2023 developer survey," Published on: https://survey.stackoverflow.co/2023.

[2] M. Allamanis, "Machine learning for big code and naturalness." [Online]. Available: https://ml4code.github.io

[3] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[4] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[5] 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," 2020.

[6] "Microsoft's codexglue project," Published on: https://microsoft.github.io/CodeXGLUE/.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[8] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[9] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," 2021.

[10] "Humaneval: Hand-written evaluation set, github project," Published on: https://github.com/openai/human-eval.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov,

H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[12] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, "Spoc: Search-based pseudocode to code," 2019.

[13] M. Chen, J. Tworek, H. Jun, and Q. Yuan, "Evaluating large language models trained on code," Published on: https://arxiv.org/abs/2107.03374, 2021.

[14] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," 2022.

[15] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. L. Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. D. Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" 2023.

[16] "Bigcode is an open scientific collaboration working on the responsible development and use of large language models for code (code llms)." Published on: https://www.bigcode-project.org/docs/about/mission/.

[17] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, dec 2022. [Online]. Available: https://doi.org/10.1126%2Fscience.abq1158

[18] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," 2016.

[19] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proceedings of the 53rd Annual Meeting*

of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 878–888. [Online]. Available: https://aclanthology.org/P15-1085

[20] O. Michael, O. Obst, F. Schmidsberger, and F. Stolzenburg, "Robocupsimdata: A robocup soccer research dataset," *CoRR*, vol. abs/1711.01703, 2017. [Online]. Available: http://arxiv.org/abs/1711.01703

[21] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, vol. abs/1709.00103, 2017.

[22] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Lincoln, Nebraska, USA: IEEE Computer Society, November 2015, pp. 574–584. [Online]. Available: https://doi.org/10.1109/ASE.2015.36

[23] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://aclanthology.org/P16-1195

[24] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *International Conference on Mining Software Repositories*, ser. MSR. ACM, 2018, pp. 476–486.

[25] "Conala: The code/natural language challenge, a joint project of the carnegie mellon university neulab and strudel lab," Published on: https://conala-corpus.github.io/.

[26] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[27] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.

[28] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," 2018.

[29] M. Jeon, S.-Y. Baik, J. Hahn, Y.-S. Han, and S.-K. Ko, "Deep Learning-based Code Complexity Prediction," 2022.

[30] E. L. G. Priyan Vaithilingam, Tianyi Zhang, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models." [Online]. Available: https://dl.acm.org/doi/10.1145/3491101.3519665

[31] M. Kazemitabaar, J. Chow, C. K. T. Ma, B. J. Ericson, D. Weintrop, and T. Grossman, "Studying the effect of AI code generators on supporting novice learners in introductory programming," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, apr 2023. [Online]. Available: https://doi.org/10.1145%2F3544548.3580919

[32] J. Pachouly, S. Ahirrao, and K. Kotecha, "Sdptool: A tool for creating datasets and software defect predictions," *SoftwareX*, vol. 18, p. 101036, 2022.

[33] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104773, 2022.

[34] M. Hasan, K. S. Mehrab, W. U. Ahmad, and R. Shahriyar, "Text2app: A framework for creating android apps from text descriptions," *arXiv preprint arXiv:2104.08301*, 2021.

[35] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: multi-mode translation of natural language and python code with transformers," 2020.

[36] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," 2023.

[37] "Papers with code," Published on: https://paperswithcode.com/datasets.

[38] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code generation using machine learning: A systematic review," *IEEE Access*, vol. 10, pp. 82 434–82 455, 2022.

[39] "Cross-language clone detection for mobile platforms," Published on: https://doi.org/10.5281/zenodo.5228822.

[40] J. Katz, "Libraries. io open source repository and dependency metadata," 2018.

[41] "Tree-sitter, a parser generator tool and an incremental parsing library," Published on: https://tree-sitter.github.io/tree-sitter/.

[42] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 2018, pp. 476–486.

[43] "The archive," Published on: archive.org.

[44] "The github activity data," Published on: https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset.

[45] "Codeparrot dataset cleaned," Published on: https://huggingface.co/datasets/codeparrot/codeparrot-clean.

[46] "Huggingface repository containing the datasets and models associated with this paper." Published on: https://huggingface.co/mvasiliniuc.

[47] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. I. au2, and K. Crawford, "Datasheets for datasets," 2021.

[48] M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru, "Model cards for model reporting," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*. ACM, jan 2019. [Online]. Available: https://doi.org/10.1145%2F3287560.3287596

[49] E. A. Huerta, B. Blaiszik, L. C. Brinson, K. E. Bouchard, D. Diaz, C. Doglioni, J. M. Duarte, M. Emani, I. Foster, G. Fox, P. Harris, L. Heinrich, S. Jha, D. S. Katz, V. Kindratenko, C. R. Kirkpatrick, K. Lassila-Perini, R. K. Madduri, M. S. Neubauer, F. E. Psomopoulos, A. Roy, O. Rübel, Z. Zhao, and R. Zhu, "Fair for ai: An interdisciplinary, international, inclusive, and diverse community building perspective," 2022.

[50] "Dutch national center of expertise and repository for research data." Published on: https://satifyd.dans.knaw.nl/.

[51] "Australian research data commons - fair self assessment," Published on: https://ardc.edu.au/resource/fair-data-self-assessment-tool/.

[52] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[53] "Training a causal language model from scratch," Published on: https://huggingface.co/learn/nlp-course/chapter7/6?fw=pt#training-a-causal-language-model-from-scratch.

[54] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, Incorporated, 2022. [Online]. Available: https://books.google.ch/books?id=7hhyzgEACAAJ

[55] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 2013.

[56] M. Vasiliniuc, "Github repository containing the project associated with this paper." [Online]. Available: https://github.com/mvasiliniuc/IVA_Main_Project

[57] "Iva-codeint-swift-small, model for generating swift code, part of the objectives of the paper." Published on: https://huggingface.co/mvasiliniuc/iva-codeint-swift-small.

[58] "Iva-codeint-kotlin-small, model for generating kotlin code, part of the objectives of the paper." Published on: https://huggingface.co/mvasiliniuc/iva-codeint-kotlin-small.

[59] "Huggingface api for inference," Published on: https://huggingface.co/docs/api-inference/index.

[60] "Huggingface web interface for hosted inference," Published on: https://huggingface.co/inference-api.

[61] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," 2023.

[62] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2022. [Online]. Available: https://doi.org/10.1145%2F3540250.3549113

Appendix A

Creating a dataset

Listing A.1: Instance of a field in Swift raw dataset

```
{
   "repo_name":"simpleandpretty/decider-ios",
   "path":"MessagesExtension/MediaResources.swift",
   "copies":"1",
   "size":"1232",
   "content":"import Foundation\nimport UIKit\n\nclass
   MediaResources {\n\n    static func
   mediaURL(forGameOption option:FightMove) -> URL {\n
     let bundle = Bundle.main\n        guard\n
   let mediaURL = bundle.url(forResource: option.rawValue,
   withExtension: \"mp4\")\n ...",
   "license":"gpl-3.0"
}
```

Listing A.2: Complete list of licenses of the Swift code extracted together with their occurrences

```
{
    "agpl-3.0": 2775,
    "apache-2.0": 180178,
    "artistic-2.0": 314,
    "bsd-2-clause": 5342,
    "bsd-3-clause": 11429,
    "cc0-1.0": 2718,
    "epl-1.0": 980,
    "gpl-2.0": 15751,
    "gpl-3.0": 33074,
    "isc": 1647,
    "lgpl-2.1": 1741,
    "lgpl-3.0": 6150,
    "mit": 476518,
    "mpl-2.0": 11799,
    "unlicense": 3277
}
```

Listing A.3: Complete list of licenses of the Kotlin code extracted together with their occurrences

```
{
    "agpl-3.0": 9146,
    "apache-2.0": 272388,
    "artistic-2.0": 219,
    "bsd-2-clause": 896,
    "bsd-3-clause": 12328,
    "cc0-1.0": 411,
    "epl-1.0": 2111,
    "gpl-2.0": 11080,
    "gpl-3.0": 48911,
    "isc": 997,
    "lgpl-2.1": 297,
    "lgpl-3.0": 7749,
    "mit": 92540,
    "mpl-2.0": 3386,
    "unlicense": 1756
}
```

Listing A.4: Exemple of a Swift file that we want to eliminate during dataset curating process

```
let package = Package(
    name: "Syft",
    products: [
        .library(name: "Syft", targets: ["Syft"])
    ],
    targets: [
        .target(
            name: "Sample",
            dependencies: ["Syft"]),
        .target(
            name: "Syft"),
        .testTarget(
            name: "SyftTests",
            dependencies: ["Syft"])
    ]
)
```

Listing A.5: Instance of a field in Swift clean dataset


```
{
    "repo_name":"...",
    "path":".../BorderedButton.swift",
    "copies":"2",
    "size":"2649",
    "content":"...",
    "license":"mit",
    "hash":"db1587fd117e9a835f58cf8203d8bf05",
    "line_mean":29.1136363636,
    "line_max":87,
    "alpha_frac":0.6700641752,
    "ratio":5.298,
    "autogenerated":false,
    "config_or_test":false,
    "has_no_keywords":false,
    "has_few_assignments":false
}
```

Table A.1: FAIRNess evaluation question and answers.

Question	Answer
Findable	
Did you provide sufficient metadata (information) about your data for others to find, understand and reuse your data?	Required metadata fields and some additional fields
Does the dataset have any identifiers assigned?	Globally Unique, citable and persistent (e.g. DOI, PURL, ARK or Handle)
What type of repository or registry is the metadata recorded in?	Domain-specific repository
Accessible	
How accessible is the data?	Publicly accessible
Is the data available online without requiring specialized protocols or tools once access has been approved?	Standard web service API (e.g. OGC)
Is the metadata publicly accessible even if the data is no longer available?	Yes
Interoperable	
What (file) format(s) is the data available in?	In a structured, open standard, machine-readable format
Did you provide contextual information about your dataset?	Persistent Identifier(s), Reference to other datasets, Reference to publications
What best describes the types of vocabularies/ontologies/tagging schemas used to define the data elements?	Standardised vocabularies/ontologies/schema without global identifiers
Reusable	
How much provenance information has been captured to facilitate data reuse?	Fully recorded in a machine-readable format (YAML)
What kind of information did you provide about the provenance of your data?	Origin of data, Workflow description for collecting data (machine-readable)
Which of the following best describes the license/usage rights attached to the data?	Non-standard text-based license

This is a set of questions gathered from both DANS and AU Research.

Appendix B

Training a Model

Example B.0.1 (Inference via API - Swift Enums)

Example B.0.2 (Inference via API - Swift Structure) *We have an example of inference via the HuggingFace Inference Python library. The example contains a nonspecific prompt, receiving code generation that is syntactically correct, and mostly in line with the domain specifications of the prompt. 'var city' is considered out of context.*

```
API_URL =
    "https://api-inference.huggingface.co/models/mvasiliniuc/
iva-codeint-swift-small"
headers = {"Authorization": "Bearer <APIKey>"}
def query(payload):
    response = requests.post(API_URL,
                headers=headers, json=payload)
    return response.json()
output = query({
"inputs": """
/*
A codable struct representing a bank account.
*/
"""
})
pprint.pprint(output, compact=True)
[{'generated_text':
'/*'
'A codable struct representing a bank account'
'*/'
'struct BankAccount: Codable {'
'    // The bank account'
'    var bankAccount: Int'
```

```
'     // The bank account name'
'     var name: String'
'     // The bank account city'
'     var city'
}]
```

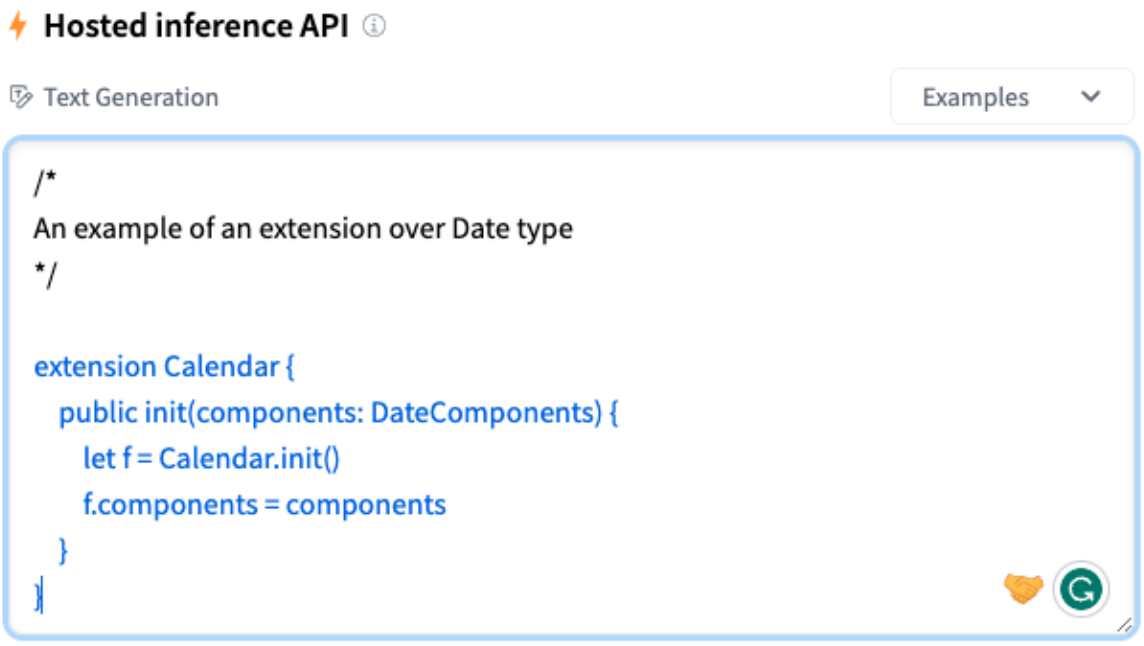


Figure B.1: Example of relatively good generation over an abstract and incomplete prompt by Swift by IVA-CodeInt-Swift-Small. The extension if not over the actual "Date" type but over a type that represents Dates and Calendar class is a common one. Also, the purpose of the extension is not provided but the model generated a constructor that has a potential utility. The code in the black color is the prompt, and the code in blue is the generated code.

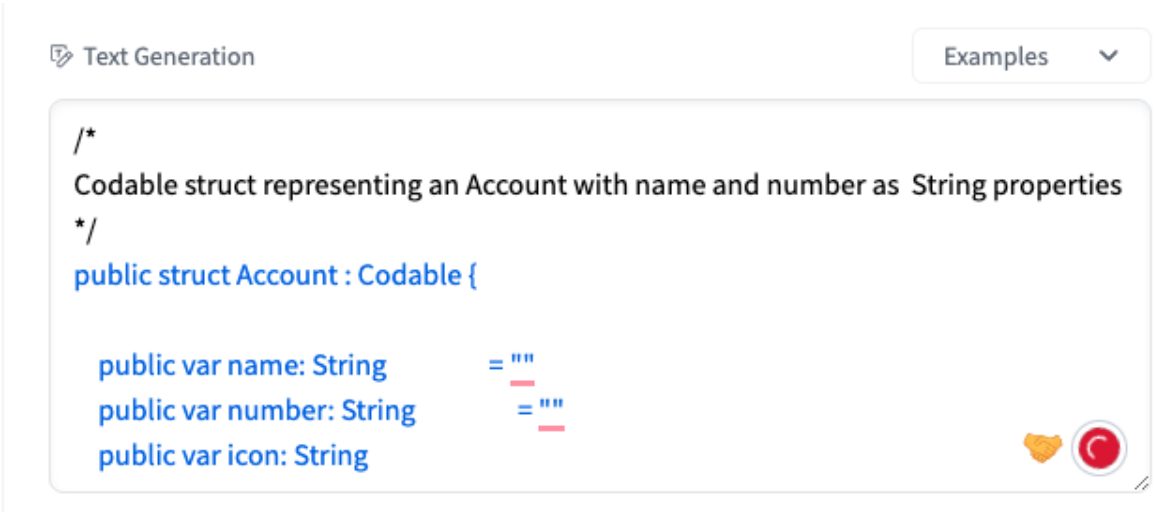


Figure B.2: Example of a structure Swift by IVA-CodeInt-Swift-Small [57]. The code in the black color is the prompt, and the code in blue is the generated code.

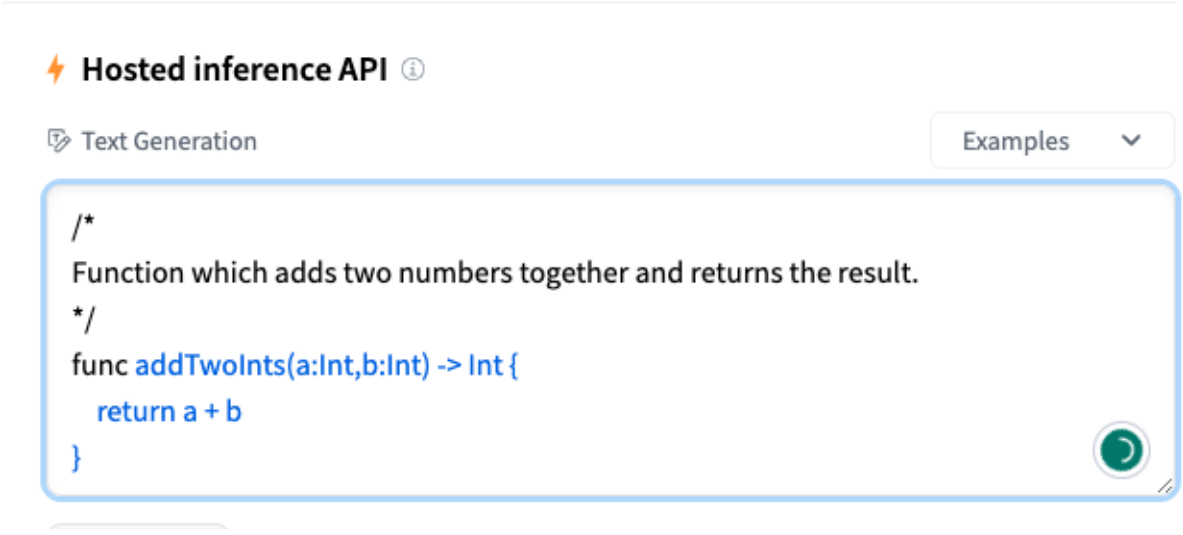


Figure B.3: Example of simple task to generate addition code provided Swift by IVA-CodeInt-Swift-Small [57]. The code in the black color is the prompt, and the code in blue is the generated code.

Appendix C

Training a Model - Charts

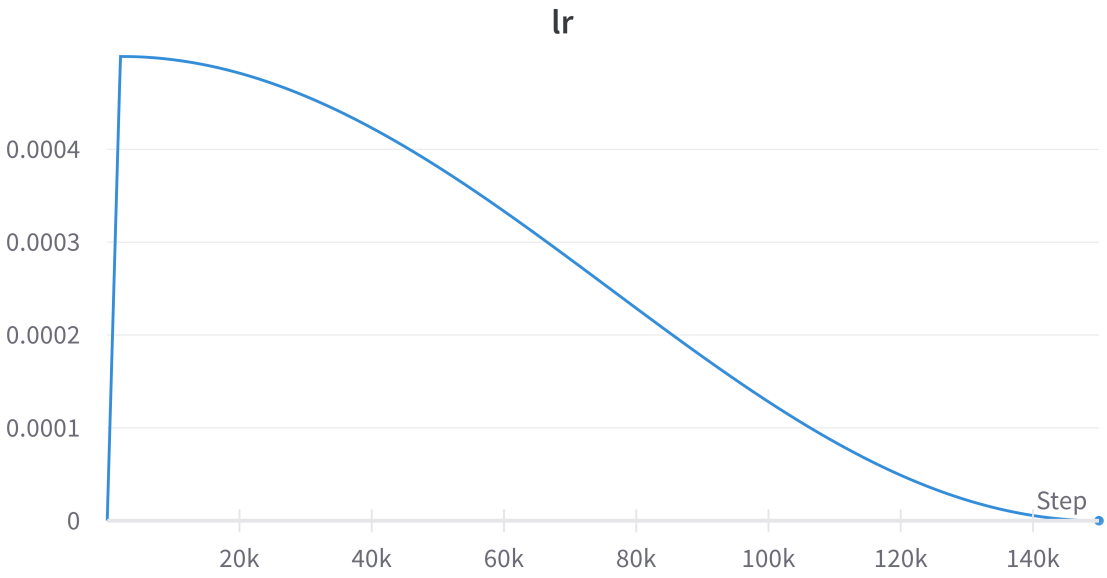


Figure C.1: Learning Rate of IVA-CodeInt-Swift-Small [57].

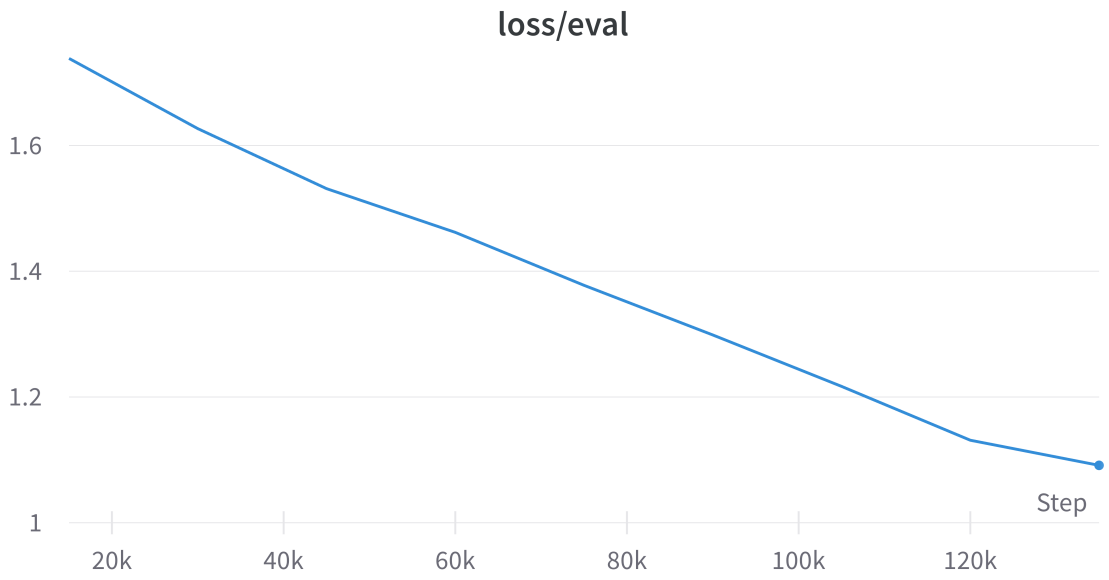


Figure C.2: Evaluation Loss of IVA-CodeInt-Swift-Small [57].

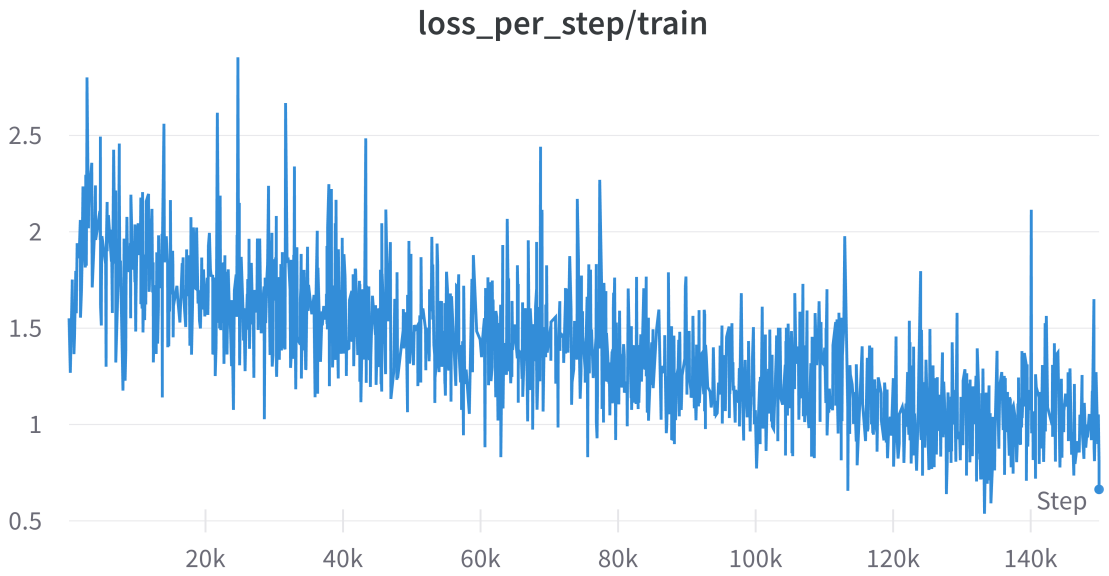


Figure C.3: Loss Per Step of IVA-CodeInt-Swift-Small [57].

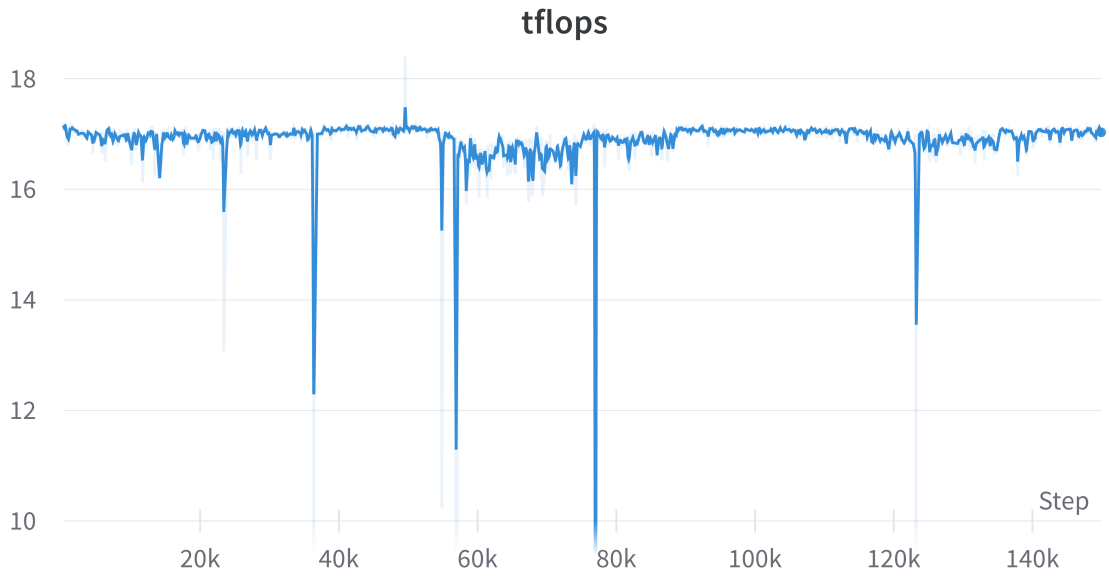


Figure C.4: TFLOPS of IVA-CodeInt-Swift-Small [57].

Appendix D

CaseStudy - Requirements Model

Session 2: AI Assistant Enabled

Rules:

1. You will receive 3 problems. Solve them in the order shown (top to bottom).
2. You can use only the following resources: Official platform documentation and Github Copilot
3. Write your code in an Android application environment.
    - See attached instructions for Intelij IDE and Github Copilot Plugin installation
    - See attached instructions for Project Setup
4. Each problem has a time limit. If you exceed the time limit please stop and move to the next task (if any remaining).
5. Maximum time for entire session: 1h.
6. After the technical tasks are completed. Please fill in the survey found in the https://docs.google.com/forms/d/e/1FAIpQLScyZRa7INxqipo0svssZQkB-OvkP7wPFlKp4XlJ7Uk8KWQ4YA/viewform?usp=sf_link

Task 1:
Time limit: 10 minutes

OL2: Write a function that takes a list of Strings as input and returns a String concatenating all elements in the provided input ignoring `whitespace` characters having all entries in Camel Case format.

Task 2:
Time limit: 20 minutes

OM2: Write a function that does a simplistic validation of a given String representing an IBAN. The validation requires the following: the first two characters are letters, the rest of them numbers, and the total number of characters is 14. The function should return an Enum having a case when the IBAN is valid and the other cases representing reasons for failure.

Task 3:
Time limit: 30 minutes

OI1: Write a class type called "Account" with properties for account number, account holder name, and balance. Implement functions to deposit and withdraw money from the account. Provide failure mechanism for withdraw operation in case balance is below 0. Implement a function that returns the current balance. Implement equality for the type "Account".

Figure D.1: Model of requirements received by a Candidate in Technical Stack Switch Phase.

Appendix E

CaseStudy - Problems

Tasks used in the Technical Onboarding Phase / Panel of the Case Study.

1. Low Difficulty:

   OE1 Write a function that receives a floating number, rounds up (ceiling) if the number is below 100, and rounds down if above 100. Returns the result as platform-specific number representation.

   OE2 Write a function that takes a list of Strings as input and returns a String concatenating all elements in the provided input ignoring ‘whitespace‘ characters having all entries in Camel Case format.

   OE3 Write a function that receives an array of arrays containing Strings and flattens it into one array with unique elements while preserving the order of the input arrays. The function must return the resulting array.

2. Medium Difficulty:

   OM1 Write a function that validates a card limit given as String and returns an Enum having a case when the limit is valid and the other cases representing reasons of failure. The card limit is considered invalid if it cannot be converted to an integer if it’s below 0 if it’s above 10.000 and if it is not divisible by 100.

   OM2 Write a function that does a simplistic validation of a given String representing an IBAN. The validation requires the following: the first two characters are letters, the rest of the numbers, and the total number of characters is 14. The function should return an Enum having a case when the IBAN is valid and the other cases representing reasons for failure.

   OM3 Write a function that receives an array of Strings representing floating numbers (simulating updated price values for a trading instrument). Return an array holding the comparison result between each consecutive number as an Enum value with 3 cases (equal, greater, smaller). The returned array of Enum values must be of the capacity of the given array minus one element.

3. Increased Difficulty:

   OI1 Write a class type called "Account" with properties for account number, account holder name, and balance. Implement functions to deposit and withdraw money from the account. Provide failure mechanism for withdraw operation in case balance is below 0. Implement a function that returns the current balance. Implement equality for the type "Account".

   OI2 Write a platform-specific value type that represents a Banking Card with the following properties 'Identity' of type UUID, 'FPAN' of type String, and 'Holder' as an optional String property. Make the type copyable, capable of returning a hash code based on its properties, and serializable.

   OI3 Write a platform-specific value type 'Utility' with two methods: (1) 'hasCapitalCase' that accepts a String as an argument and a function with a Boolean as an argument and no return type. Implementation of this method must call the function argument before the block of the function is finished with a Boolean indicating if the String given as argument is written in capital letters or not. The second function (2) named 'hasLowerCase' receives the same type of arguments but the function given will be called after a one-second delay with true if the given string has lower case, false otherwise.

Tasks used in the Technical Stack Switch Phase / Pannel of the Case Study.

1. Low Difficulty:

   SL1 Write an extension function to String type that returns true if the string given represents the current operating system version or false otherwise.

   SL2 Write an extension function to Date type (use a platform-specific representation), that returns true if the date is a leap day or false otherwise.

   SL3 Write an extension function to the platform-specific locale type that returns true if the system has the current language set to Romanian.

2. Medium Difficulty:

   SM1 Write a platform-specific asynchronous function that receives an array of integers and returns the minimum and maximum values in the array using a tuple (pair) type after a delay of one second.

   SM2 Extend native representation of Date with 3 instance functions. First returns the value of the calendar date in the German time zone with the following format: "yyyy-MM-dd'T'HH:mm:ss.SSS". The second function returns the value of the date using the US locale and the Romanian timezone. The 3rd function returns the value of the date in epoch time.

SM3 Write a platform-specific class type named ‘CallbacksStore‘ that has two methods: (1) ’addCallback’ which receives a function with one integer argument and no return type (closures/lamdas) and stores them and (1) ‘triggerCallbacks‘ which accepts an integer. Every time ‘triggerCallbacks‘ is called, all registered callbacks until that time must be called.

3. Increased Difficulty:

SI1 Write a class that wraps app state events and provides a data stream that publishes an event at each app state change. Use platform-specific types for data flows and event representation.

SI2 Write an abstract type ‘ErrorTranslatable‘ with two methods: 1. ‘userFacingReason‘ receiving a platform-specific error and returning a String to be shown to the user, specific to the error if the error is recognizable if not a generic message. 2. ‘userFacingCode‘ receives a platform-specific error and returns an optional Int to be shown to the user if the error is cognizable. Write a concrete type that implements the abstract type. Provide default behavior for the two methods if not overridden by a concrete type. Use type extensions to implement the behavior of the abstract type.

SI3 Write a class that acts as a private implementation for accessing User Preference store. Provide only two methods, one for writing a String for a given String key, and one for reading a String for a given key.

Table E.1: Training - Summary of Config Parameters

Parameter	**Value**
_cpu	False
seed	1
device	cuda
backend	None
save_dir	output
use_ipex	False
tokenized	true
model_ckpt	mvasiliniuc/iva-codeint-swift-small
seq_length	1024
weight_decay	0.1
fork_launched	False
learning_rate	0.0005
num_processes	1
process_index	0
max_eval_steps	-1
shuffle_buffer	10000
max_train_steps	150000
_mixed_precision	fp16
deepspeed_plugin	None
distributed_type	DistributedType.NO
num_warmup_steps	2000
train_batch_size	5
valid_batch_size	5
lr_scheduler_type	cosine
dataset_name_train	mvasiliniuc/iva-swift-codeint-clean-train
dataset_name_valid	mvasiliniuc/iva-swift-codeint-clean-valid
local_process_index	0
save_checkpoint_steps	15000
gradient_checkpointing	false
resume_from_checkpoint	null
gradient_accumulation_steps	1

Appendix F

CaseStudy - Examples

Example F.0.1 (Poor reviewer score) *As an example here is the result provided by ChatGPT for one of the tasks in Technical Onboarding pool, in a Swift language setup:*

```
func roundNumber(_ number: Double) -> NSNumber {
    if number < 100 {
        return NSNumber(value: ceil(number))
    } else {
        return NSNumber(value: floor(number))
    }
}
```

This solution uses an ObjC type NSNumber instead of a modern Swift-like version like Double or any Numeric. The signature is not in line with Swift API design which requires the following in such a situation:

```
func round(number: Double) -> any Numeric
```

The control flow is not ideal in this case and the policy of no inline constants is violated. We want to avoid if else construction in a modern language. Alternative:

```
enum RoundingThreshold {
  static let ceiling: Double = 100
}
func round(number: Double) -> any Numeric {
  let roundedResult: any Numeric
  switch number {
  case let value where value < RoundingThreshold.ceiling:
    roundedResult = ceil(value)
  default:
```

```
        roundedResult = floor(number)
      }
      return roundedResult
    }
```

Example F.0.2 (Prompts without context - ChatGPT4) *Initial Prompt from the candidate, copied directly from the case study: "Write a function that takes a list of Strings as input and returns a String concatenating all elements in the provided input ignoring 'whitespace' characters having all entries in Camel Case format."*

```
// This prompt generated a Python code since the 'preferred' programming
// language of an LLM is Python. We will avoid showing the code here since
// it's not relevant to the example.
```

The candidate changed the prompt to: "Write a function in swift that takes a list of Strings as input and returns a String concatenating all elements in the provided input ignoring 'whitespace' characters having all entries in Camel Case format."

```
func concatenateStringsIgnoringWhitespace(strings: [String]) -> String {
  var concatenatedString = ""

  for string in strings {
    let trimmedString = string.replacingOccurrences(of: " ", with: "")
    concatenatedString += trimmedString.capitalized
  }

  return concatenatedString
}
```

This generated a solution that passes 2/5 unit tests. This is due to the misunderstanding of the concept of the 'Camel Case" format which demands the output to be as such: "firstSecondThird"

To compare, other candidates added a short contextual instruction and an example of the expected format which improved the output. By doing this type of association, the model was able to provide a better understanding of the Camel Case concept.

A more successful prompt was: "Act as a Swift developer. Solve the following: Write a function that takes a list of Strings as input and returns a String concatenating all elements in the provided input ignoring 'whitespace' characters having all entries in Camel Case format. Example of output format 'largeLanguageModel'"

```
func concatenateToCamelCase(_ strings: [String]) -> String {
  var camelCaseString = ""
```

```
  for string in strings {
    let trimmedString = string.trimmingCharacters(in: .whitespaces)
    let components = trimmedString.components(separatedBy: .whitespaces)
    for component in components {
      camelCaseString += component.lowercased()
      camelCaseString.replaceSubrange(
        camelCaseString.startIndex...camelCaseString.startIndex,
        with: String(camelCaseString[camelCaseString.startIndex])
            .capitalized)
    }
  }
  return camelCaseString
}
```

Example F.0.3 (Prompts with technical details) *The initial prompt was: "Write a function that receives an array of arrays containing Strings and flattens it into one array with unique elements while preserving the order of the input arrays. The function must return the resulting array.". The output of this initial prompt:*

```
func flattenArray(_ arrays: [[String]]) -> [String] {
  var flattenedArray = [String]()
  var uniqueElementsSet = Set<String>()
  for array in arrays {
    for element in array {
      if !uniqueElementsSet.contains(element) {
        uniqueElementsSet.insert(element)
        flattenedArray.append(element)
      }
    }
  }
  return flattenedArray
}
```

A more successful prompt included technical guidance: "Act as Swift Developer. Write a function that receives an array of arrays containing Strings and flattens it into one array with unique elements while preserving the order of the input arrays. The function must return the resulting array. Solve it using functional programming via Combine framework and remove duplicates using an operator."

```
func flattenArray(_ arrays: [[String]]) -> [String] {
  let flattenedArray = arrays
      .publisher
      .flatMap { $0.publisher } // Flatten the array of arrays
```

```
      .removeDuplicates() // Remove duplicate elements
      .collect() // Collect the result into an array
  return flattenedArray
}
```

Example F.0.4 (Incremental Improvements via Step by step prompts in Copilot.) *A few candidates achieved high scores at corrected and ReviewerScore, in the more complex phase of the Technical Stack switch, used an approach that broke the problem into smaller ones. This is especially useful when the AI-Tool / Model is not yet fined tuned in the context. For this example, the candidate decided to put the text of the problem at the beginning of a file but helped the model with concrete steps. One important note regarding this approach is that if the model is fined tuned with step inline comments, it will provide these inline comments in future recommendations/code completions if the context is preserved.*

```
// Problem: Write a platform-specific asynchronous function
// that  receives an array of integers and returns the minimum and
// maximum values in  the array using a tuple (pair) type after a
// delay of one second.

// Step 1: Declare a suspend function that returns an optional
// pair consisting  of max and min from a given array.
suspend fun bounds(array: Array<Int>): Pair<Int, Int>? {
    // Step 2: Return null pair if the array is empty
    return if (array.isEmpty()) {
        null
    } else {
        // Step 3: Define var min and max each having the value of
        // the first element in the array.
        var min = array.first()
        var max = array.first()
        // Step 4: Iterate through the array and compare each element
        // with min and max.
        array.forEach {
            if (it < min) {  min = it  }
            if (it > max) {  max = it  }
        }
        // Step 5: Add a delay of 1 second
        kotlinx.coroutines.delay(1000)
        // Step 6: Return the pair of min and max
        Pair(min, max)
    }
}
```

Example F.0.5 (Incremental Improvements in ChatGPT4.) *The initial prompt: "Act as a Swift developer. Write a class that wraps app state events and provides a data stream that publishes an event at each app state change. Use platform-specific types for data flows and event representation.". Since this involves multiple entities, we will not post the entire code as the output of each iteration but describe some characteristics that were not ideal.*

For this initial prompt, the generated code was functional but would never pass a Merge Request in a modern mobile department. Flaws include the usage of deprecated types, representing the state as String, and flaws in solution design (tight coupling). To achieve the resulted results, the candidate provided the following prompts:

Second prompt: "Avoid using UIApplicationDelegate and use Notification Center. It is required that the events published to the eventStream be triggered from inside of the AppStateWrapper implementation."

Third prompt: "Convert the AppState from String to an enum type for all required cases."

Final prompt: "Provide an interface type for AppStateWrapper". Note that in Swift, the 'interface type' is called 'protocol' but since this is a LLM trained on multi-tasks, the association is done immediately.

The resulting output received a high ReviewerScore.

```
import Combine

// Define the app state enum
enum AppState {
  case foreground case background case inactive
}
// Define the app state event structure
struct AppStateEvent {
  let newState: AppState
}
// Define the AppStateWrapper protocol
protocol AppStateWrapperProtocol {
  var eventStream: AnyPublisher<AppStateEvent, Never> { get }
}
// Implement the AppStateWrapper protocol in a class
class AppStateWrapper: AppStateWrapperProtocol {
  // Define a data stream publisher for the app state events
  private let eventPublisher = PassthroughSubject<AppStateEvent, Never>()

  // Define a computed property to expose the data stream as a publisher
  var eventStream: AnyPublisher<AppStateEvent, Never> {
    return eventPublisher.eraseToAnyPublisher()
  }
```

```
  init() {
    registerForAppStateNotifications()
  }
  // Rest of the implementation...
}
```

Example F.0.6 (Platform Association in Technical Stack Switch) *The initial prompt was extracted from the task itself without altering for platform adaptations. This led to a solution considered deprecated.*

```
// A class that acts as a private implementation for accessing User
// Preference (PreferenceStorage) store.
// Provide only two methods, one for writing a String for a given
// String key, and one for reading a String for a given key.
class PreferenceStorage(private val sharedPreferences: SharedPreferences) {
    fun save(key: String, value: String) {
        sharedPreferences.edit().putString(key, value).apply()
    }
    fun read(key: String, defaultValue: String): String {
        return sharedPreferences.getString(key, defaultValue)!!
    }
}
```

The second prompt provides two names existing in the library and gives a more clear direction of what should be used.

```
// A class that acts as a wrapper on top of DataStore for Preferences.
class PreferencesDataStoreWrapper(
    private val dataStore: DataStore<Preferences>,
    context: Context
) {
    suspend fun store(key: String, value: String) {
        dataStore.edit { preferences ->
            preferences[stringPreferencesKey(key)] = value
        }
    }
    suspend fun read(key: String) : String? {
        val preferences = dataStore.data.first()
        return preferences[stringPreferencesKey(key)]
    }
}
```

Appendix G

CaseStudy - Complete Candidate Feedback

PTS1 Rate the helpfulness of the AI-assisted tools (Github Copilot/Chat GTP and IVA-CodeInt)

- I was very helpful and easy also because the tasks were well articulated and I had only to feed the task to Chat GPT. In a real-world scenario, I would probably use both AI and Google Search for the issues that I have.
- I can say that's very fast. I solved 2 tasks out of 3 in less than 4 minutes each.
- The official documentation of any platform contains more information, but requires more time to find specific information. The AI tools is helpful to find information without using platform-specific language and it also provides samples for the required information
- "Getting input from the AI-assisted tool saved time, which otherwise would have been spent on Google searches for API definition, language syntax, etc.

  In some cases it even implemented function logic which was not mentioned in the prompt, but could be deducted from the function naming, saving up additional time."
- It's helpful, but it's important to understand the code that generates and not trust in them 100
- The solution given by ChatGPT was good enough. I had to fix some things related to optional types. (default value)
- Only typing the name of a class (for example) is able to generate useful properties and methods
- The tool was able to provide very clear solutions which required only minimum levels of adaptations.

- AI provided an explanation for the code, making it easier to understand.
- The copilot gave almost complete solutions for all the problems.
- The AI-assisted tool (Github Copilot) helped me solve the problems, but needed to give more information to the Copilot to get the correct solutions.
- It was helpful by providing certain language-specific autocompletion. Ex: Pair for tuple, Build.Version etc.
- Getting used to the tool requires more preparation time than expected.

PTS2: Rate the level of understanding of the AI-assisted code

- It described well all the objects involved. For a beginner in iOS like me, I needed some extra questions asked for the third task. Probably with somebody with a little more experience with the platform, the initial explanation would have been enough.
- The code is easy to understand and most of the time the AI explains what it's doing.
- Results were clear and useful to find the solution
- Usually, concepts used within the provided answer for a given question are pretty well explained from the first iteration. Additionally, requesting a detailed description of some of the language features is an engaging way of getting an understanding of the new language.
- The information provided by ChatGPT was detailed and easy to understand.
- The solution was very easy to understand, Chat GPT not only provided the code but also explained every step of the solution.
- Github Copilot is very easy to use, it is pretty intuitive.
- A lot of the suggestions provided by Copilot were out of context.

PTS3: Rate the level of confidence you have in final code

- Functional, yes. Not sure about the coding guidelines as I didn't focus at all on that.
- The final code might not be perfect, but can be easily adjusted. I couldn't solve one of my tasks in time.
- I'm confident about the functionality of the code, but I would bring improvements in terms of coding guidelines. However, I don't think this is necessarily related to the AI Tool, but more to the knowledge and experience in writing for a specific platform and the time invested in solving a task.

- ”When talking about smaller code snippets, preferably inside a single method the confidence is pretty high. However, when it comes to the integration of the code, the AI tool provided answers lack some details, and for a beginner, things can get confusing cause a full picture is not provided. Some experience from the developer is required to bind all the provided code snippets in their right place in order to have a functional final code.”
- ChatGPT does not respect all existing rules of the guidelines, but the code is pretty well written.
- The solutions are explained in detail and in an easy-to-understand manner by the AI tool and are well structured, giving me confidence in its decision. Still, a minor level of checking is required not necessarily in the solution design but in code quality.
- From experience, it can make errors even on simpler tasks.
- The solutions provided are the correct ones.
- I am confident that the final code is functionally correct but I am not confident in it respects the coding guidelines. Additionally, the code output by the tool had some issues which I had to fix manually.

PTS4: Rate the level of expected future usage of the AI-assisted code tools on a scale from 1 to 5

- Overall I think it helped me with productivity and it is something I could use in some situations.
- I see the benefits of using it for computational tasks and for finding quick answers to specific tasks. However, in day-to-day work, the tasks are more complex, and finding a solution using an AI tool, in my opinion, requires a change in the approach of asking for information, in comparison to short and isolated.
- Can see an opportunity to use it when getting onboarded into new topics. Since in the beginning, entry-level information is enough to get started quickly, it can speed up the learning process significantly. However, once more advanced topics are on the table, careful double-checking of official documentation sources is necessary.
- The tool is very powerful and easy to use, there are not many reasons to not use it.
- I feel that Copilot will be used for autocompleting repetitive code (code that is shared between a lot of projects), and Chat GPT for the cases when you just want a complete solution at once.
- I expect such a tool to be used more for specific tasks like unit tests. I don’t see a big advantage to using it in feature development.

Appendix H

How was ChatGPT used for this paper

In a paper that involves the usage of generic LLMs for productivity aspects, we find it important to make transparent how such a tool was used in the process of writing. We used ChatGPT for the following:

- Latex Language. We found the tool highly efficient to create Latex structures such as tables, tabular, figures, examples, code, and itemization together with corresponding characteristics only based on prompts. Generated structures were used and filled in with original content from authors. We used also it for finding modules in Latex to achieve a specific feature and compare them visually in a few seconds.
- JSON formatting. Same as for latex, we used LLMs with prompts, to transform JSON content easier to consume (for example: alphabetically sorting pairs by keys).
- Python API search and exemplification.
- Provide Excel support for formulas and charts.
- Finding appropriate academic synonyms for repeating terms.