Empowering Mobile-Only App Generation — Offline AI Code Generation with App Inventor

by

Joyce Yuan

B.S. Electrical Engineering and Computer Science, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTERS OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Joyce Yuan
Department of Electrical Engineering and Computer Science
May 9, 2025

Certified by: Harold Abelson
Class of 1922 Professor of Computer Science and Engineering, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee
Graduate Officer, Department of Research

Empowering Mobile-Only App Generation — Offline AI Code Generation with App Inventor

by

Joyce Yuan

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2025 in partial fulfillment of the requirements for the degree of

MASTERS OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

As digital tools become more accessible, creating software is becoming a powerful way for anyone to make real-world impact. Computational action—the idea that learners can build computing artifacts with authentic relevance to their lives and communities—reframes computing as a tool for empowerment. Low-code platforms like MIT App Inventor support this vision by fostering digital agency through purposeful creation. Recent advances in large language models (LLMs) expand these possibilities further by enabling code generation from natural language, offering a timely opportunity to lower the barrier to app creation.

MIT App Inventor has long championed accessibility, allowing even young learners in underserved regions to build meaningful mobile apps. Its natural language tool, Aptly, enables users to describe app ideas and generate functional code. However, Aptly's reliance on cloud-based LLMs limits access for users without stable internet—often those who could benefit most.

This thesis addresses that challenge by enabling AI-powered app creation to run entirely offline on mobile devices. We fine-tune and quantize LLaMA 3B using QLoRA and deploy it on iOS with MLC LLM, enabling on-device inference without internet. We also introduce a custom evaluation framework tailored to Aptly's grammar, combining a Tree-sitter parser and a modified CodeBLEU metric to assess both semantic and syntactic quality. Using curated evaluation datasets, we benchmark out-of-box and fine-tuned models across prompting strategies. In our evaluations, fine-tuned GPT-4.1 achieved the highest normalized CodeBLEU score (0.36 ± 0.12) and parsed over 81% of completions, outperforming its baseline by more than 5%. QLoRA-finetuned LLaMA improved parseability by 11.7% over its base model, showing progress in adapting smaller models to the Aptly domain, though semantic fidelity remains a challenge. Our results show that offline natural language–to–app generation is feasible, and that smaller models can be adapted to the Aptly domain. By lowering the technical and infrastructural barriers to app creation, this work lays the foundation to empower AI-assisted programming that is accessible, offline, and on the phone.

Thesis supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I want to thank my advisor, Professor Hal Abelson, for being the most patient, thoughtful, and inspiring mentor I could've asked for. Thank you for always taking the time to offer guidance, support, and encouragement—and for giving me the opportunity to work on such a cool project in the first place.

A huge thank you to my supervisors, Evan Patton and David Kim. Evan, thank you for your endless knowledge and for always being willing to dive into the depths of debugging and permission errors with me (and usually emerging victorious). David, thank you for your responsiveness, and your steady insightful feedback throughout this process.

I want to thank Jennet Zamanova for helping curate a massive dataset of auto-labeled examples for our experiments, and Jacky Chen for building out the evaluation pipeline and helping analyze all the results. Shoutout to my LLM experiments and evals team—Kidus Yohannes and Catherine Zhu—couldn't have done it without our collective brainpower. And a shoutout to Ashley Granquist, whose work laid the foundation for much of what I built on—thank you for paving the way.

A special shoutout to the support of my friends, my housemates, family, the vibrant 2W wing community, and my emotional support plushies for getting me through slumps. Shoutout to the semesterly Teaholic boba runs (essential), to Vester for being the ever-reliable caffeine source (and to Arianna Scott for being a fellow dedicated Vester go-er :), and to the Capital One free drinks on Mondays. To the entire App Inventor team—thank you for being such a warm, supportive, and genuinely fun community to work with.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Overview

Artificial intelligence and mobile technologies have the potential to profoundly impact the way people around the world create, learn, and solve problems. However, access to these technical innovations remain deeply uneven because AI-powered tools often rely on high bandwidth, cloud compute, and technical expertise. Underserved communities continue to face systemic barriers to engaging with or benefiting from emerging technologies.

MIT App Inventor is a platform designed to bridge this gap, offering a block-based, beginner-friendly environment used by tens of millions of users for building mobile apps without code. In 2022, the platform introduced Aptly, an AI-powered tool that goes one step further: it uses large language models (LLMs) to generate entire mobile apps from natural language descriptions. However, Aptly in its current form depends on cloud-based LLM APIs such as OpenAI's GPT or Google Gemini. This reliance on constant and stable internet connectivity severely limits who can use the tool, especially in regions where WiFi infrastructure is sparse or expensive. As a result, the very users who would benefit most from accessible, AI-powered programming are often unable to access it.

Recent progress in TinyML and on-device inference has introduced a new paradigm for deploying large language models: running them directly on mobile phones. With quantization techniques, low-rank adaptation, and lightweight inference runtimes, it is now feasible to move beyond server-based models and support offline natural language processing entirely at the edge.

This thesis explores the feasibility, design, and implementation of running Aptly's natural language-to-app generation pipeline entirely on-device. Specifically, we investigate how to (1) embed a fine-tuned LLM capable of generating App Inventor code on an iPhone using MLC LLM, and (2) evaluate Aptly's performance through a benchmark and testing framework. The overarching goal is to push the boundaries of mobile app creation—making it truly accessible, independent of internet access, and powered by AI.

1.2 MIT App Inventor and Aptly

MIT App Inventor is a free, block-based programming platform that allows users to create fully functional mobile apps through a drag-and-drop interface. Launched in 2009, App Inventor was designed to make app development approachable for everyone, especially students and first-time programmers. Instead of writing code, users piece together visual programming blocks that represent logic, data, and UI components. Over the years, the platform has been used by tens of millions of people around the world to create apps that address real problems in their communities—from health tools and learning aids to games and utilities.

Beyond just building apps, the mission of App Inventor is to promote computational thinking and empower people—particularly those in underserved communities—to become creators of technology rather than just consumers. The platform is used in schools, after-school programs, and community centers globally, with a particular focus on inclusion and accessibility. As of 2023, MIT App Inventor has empowered over 18 million learners across more than 200 countries and regions to create mobile applications, with nearly half of these users residing in developing nations. Collectively, these users have built over 100 million apps, demonstrating the platform's significant role in democratizing mobile app development worldwide [1].

In 2022, MIT App Inventor introduced Aptly [2], a new AI-driven feature that allows users to generate apps by simply describing them in natural language. Aptly leverages large language models (LLMs) to translate freeform English descriptions into structured code that can be rendered in the App Inventor interface. For example, a prompt like "a quiz app with three multiple choice questions and a score tracker" can result in an automatically generated App Inventor project with screens, components, and blocks pre-populated to match the description. This lowers the barrier for beginners and opens the door to rapid app prototyping by users who may have never seen a block of code.

Aptly represents a powerful extension of App Inventor's vision: to democratize computing education and make meaningful app creation accessible to anyone, regardless of background.

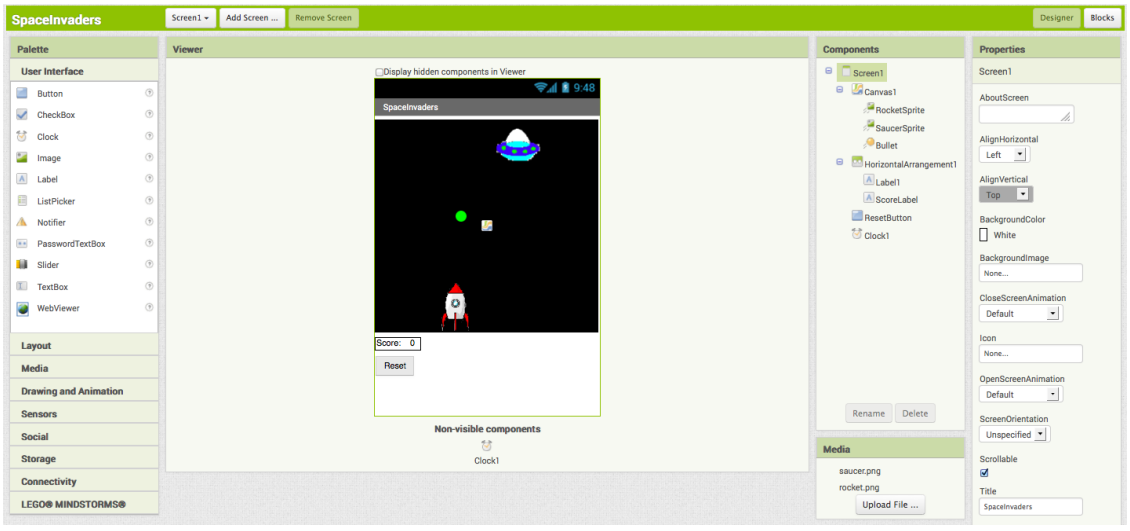


Figure 1.1: Recreating Space Invaders Game on the App Inventor web app

1.3 Research Questions

The core motivation behind this thesis stems from two fundamental questions regarding the accessibility and effectiveness of MIT App Inventor's Aptly tool:

1. **How can we enable offline natural language-to-app generation on mobile edge devices using Aptly and App Inventor?** Currently, Aptly requires constant internet connectivity to interact with cloud-based large language models (LLMs). This dependency severely restricts access, particularly in underserved areas with limited or unstable internet connections. The first motivating question addresses whether recent advancements in tiny machine learning and on-device inference can enable the deployment of Aptly's LLM directly on mobile devices, thus removing the reliance on cloud infrastructure.

2. **How can we effectively measure Aptly's performance?** Evaluating Aptly's ability to accurately generate functional mobile apps from natural language descriptions is challenging. Standard metrics may not fully capture the usability, correctness, or practical value of the generated apps. The second motivating question explores the need to create a tailored evaluation framework that can reliably assess the quality and accuracy of Aptly-generated code, thus ensuring continuous improvement and usability of the tool.

1.4 Contribution

This thesis makes four primary contributions aimed at addressing the motivating questions outlined above:

1. **A proof-of-concept implementation of Aptly running entirely offline on mobile devices.** Specifically, we fine-tune the open-source LLaMA 3B model using the QLoRA method on a domain-specific dataset, then leverage the MLC LLM framework to quantize and package the model for efficient inference within an iOS App Inventor application. We provide a live demonstration showcasing Aptly functioning fully offline.

2. **A custom evaluation framework for Aptly-generated code using custom CodeBLEU.** We introduce a tailored version of the CodeBLEU metric, integrating a newly created Tree-sitter parser specifically designed for the Aptly code format. This framework enables precise, structured evaluation of Aptly's performance, particularly suited to its unique visual programming syntax.

3. **Curation of datasets and a thorough benchmarking and analysis of Aptly's current code-generation methodologies.** We curate a few datasets (both manually labeled and auto labeled) that allow us to finetune and test various generation methods. Leveraging the new evaluation framework, we comprehensively evaluate Aptly across various model setups, including few-shot prompting versus rule-based prompting, comparing cloud-based models (OpenAI, Gemini, Claude) against locally deployed,

fine-tuned models. This analysis provides valuable insights into Aptly's strengths, weaknesses, and potential areas for improvement. In particular, we find that fine-tuned GPT-4.1 achieves the best overall performance, with a normalized CodeBLEU score of 0.36 ± 0.12 and over 81% syntactic validity, while QLoRA-finetuned LLaMA improves parseability by 11.7% over its base model but still lags in semantic fidelity.

4. **Preliminary user studies examining initial user reactions and interactions with Aptly.** As an additional contribution, we conduct early-stage qualitative user studies to better understand how novice programmers and students interact with and respond to Aptly's natural language-to-app capabilities. These insights help guide future developments and usability enhancements.

The full evaluation pipeline, research scripts, and associated datasets at https://github.com/mit-cml/eval-codegen-aptly.

The remainder of this thesis is organized as follows: Chapter 2 provides technical background on MIT App Inventor, Aptly architecture, relevant machine learning techniques, and code evaluation benchmarks. Chapter 3 describes the system architecture and implementation of Aptly on the phone. Chapter 4 presents the design of the Aptly evaluation framework and the process for building the dataset. Chapter 5 reports experimental results and user feedback. Finally, Chapter 6 discusses limitations, future work, and broader implications.

Chapter 2

Background & Preliminaries

2.1 Child-AI Co-Creation and Educational Theory

Programming has long served as both a cognitive scaffold and creative outlet for young learners. Constructionism, first introduced by Papert [3], frames learning as a process where students build meaningful artifacts. This idea underpins tools like Scratch [4], which allows children to create interactive media, and MIT App Inventor [5], which enables them to design functional mobile apps that address real-world problems. These platforms support what Tissenbaum et al. call *computational action*—applying code to engage with and transform the world [6].

Large Language Models (LLMs) offer a promising extension of this paradigm. Instead of acting as passive instruction tools, they can serve as collaborative co-creators—helping learners brainstorm ideas, scaffold logic, and articulate design goals. Conversational AI agents, such as smart speakers or chat-based assistants, have already been shown to help guide children through structured tasks and reading activities [7].

In the context of mobile app development, tools like Aptly [8] and App Planner [9] are pushing the boundaries of what learners can create. These systems use natural language prompts to generate working mobile applications, allowing students to iterate on ideas in real time and reflect on both technical and human-centered design considerations. Early user studies with high school students suggest that such tools increase confidence and broaden perspectives on the design process, especially when learners consider the societal impact of their apps.

As LLMs become more accessible and efficient, their role in educational settings may shift even further—from tutors to creative partners. This thesis builds on this vision by investigating how to make these tools available offline, empowering children and educators in bandwidth-limited environments to co-create with AI and take computational action into their own hands.

2.2 Machine Learning on Edge Devices

Large language models (LLMs) are AI systems trained on vast amounts of text data to understand and generate human-like text across various contexts. They use deep learning

techniques, particularly transformer architectures introduced by Google in 2017 [10], to predict and produce text based on input prompts. LLMs gained prominence around 2020 with the release of models like OpenAI's GPT-3, which demonstrated impressive capabilities in natural language processing, translation, and code generation. The significance of LLMs in accessible coding lies in their ability to assist people with little coding experience in writing functions, solving automated tasks, or even building apps. These models reduce the time and effort required for coding tasks and have the potential to democratize software development by lowering the barrier to entry.

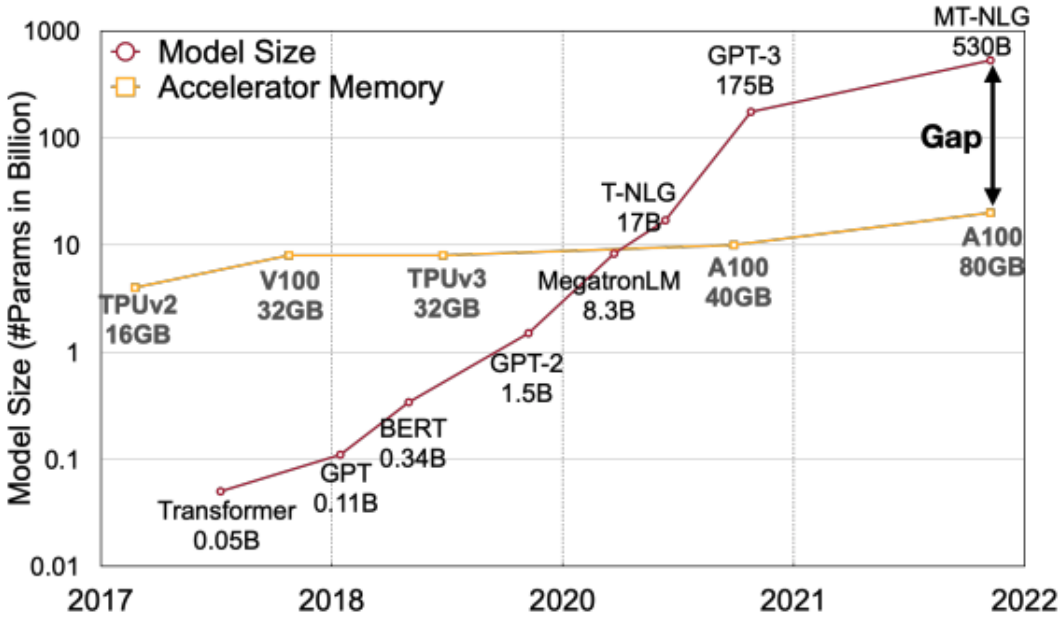


Figure 2.1: The model size of llms over the past few years have grown faster than our development of GPUs with more memory [11], causing a gap and a need for smaller models

The goal of this thesis is to enable offline, mobile-first natural language-to-code generation. However, realizing this vision requires overcoming substantial technical challenges: today's most powerful large language models (LLMs) demand compute and memory resources far beyond the capabilities of mobile hardware. Understanding these challenges—and the landscape of solutions developed to address them—is crucial for contextualizing the contributions of this work.

Large-scale LLMs such as GPT-3 [12], PaLM [13], and LLaMA [14] have achieved remarkable results on a wide variety of tasks, from natural language understanding to programming. Yet these models often have hundreds of billions of parameters, requiring hundreds of gigabytes of storage and trillions of floating-point operations per inference. For instance, GPT-3 (175B parameters) needs over 350GB of memory in standard floating-point format; OpenAI's GPT-4 has 1.76 trillion parameters, and Google's Gemini model, introduced in 2023 has 540 billion parameters [15].

Mobile devices, by contrast, impose strict constraints:

- **Memory:** 4–12 GB of RAM is typical for smartphones.

- **Compute:** ARM CPUs, mobile GPUs, and NPUs offer much less FLOPS than server-class accelerators.
- **Energy:** Power consumption must be minimized for battery longevity.
- **Thermal Budget:** Sustained high performance quickly leads to overheating and throttling.

Using the third party services provided by companies for their PPMs often require stable wifi, which isn't always possible in under privileged areas. These disparities motivate the need for significant model optimization before deployment on mobile platforms.

2.2.1 State of the Art for Mobile LLM Deployment

To enable LLMs to run on mobile and edge devices, the machine learning community has explored a diverse set of strategies that reduce memory, compute, and energy demands. These strategies aim to preserve task performance while adapting models to the unique constraints of on-device inference. Below, we briefly review four major technical directions:

- **Distillation** [16]: This approach involves training a compact *student* model to imitate the behavior of a larger, high-performing *teacher* model. The student is trained not just on task labels, but also on the teacher's soft outputs (probability distributions), which convey richer semantic information. For example, DistilBERT reduces BERT's size by 40% while retaining over 95% of its language understanding performance, making it far more practical for mobile inference. Distillation has also been explored in code generation contexts, such as training smaller Transformer models on GPT-3 outputs for program synthesis tasks.
- **Quantization** [11, 17, 18]: Quantization reduces the numerical precision of model parameters (e.g., from 16-bit or 32-bit floats to 8-bit or 4-bit integers), greatly reducing memory usage and increasing hardware efficiency. This technique, when applied post-training or during training (as in quantization-aware training), can significantly reduce model size and speed up inference with minimal loss in accuracy. Quantization is especially impactful in edge scenarios where memory bandwidth and thermal budgets are limited.
- **Pruning** [18–20]: Pruning eliminates weights or entire structures (such as attention heads or MLP channels) that have limited contribution to a model's output. It can be applied element-by-element or via structured approaches like channel or filter pruning, which maintain the network's architectural consistency. Structured pruning techniques, in particular, are favored for mobile inference because they align better with hardware acceleration and lead to more predictable compute reductions.
- **Efficient Model Architectures**: Instead of compressing large models, another strategy is to design inherently efficient networks tailored for constrained environments. MobileBERT [21] re-engineers BERT with bottleneck structures and inverted residual layers, achieving competitive performance with significantly fewer resources. More

recently, **TinyLlama** [22] offers a 1.1B parameter model trained from scratch on high-quality corpora, reaching strong accuracy on language benchmarks while being small enough for deployment on edge devices.

Beyond these algorithmic techniques, practical deployment of compressed LLMs requires compiler-level optimization and hardware integration. Emerging toolchains such as **MLC LLM** [23] address this need. MLC LLM allows developers to:

- Quantize and compile LLMs using TVM-based kernel fusion,
- Automatically generate platform-specific binaries (e.g., Metal for iOS, Vulkan for Android),
- Deploy models on-device with a small runtime, abstracting away hardware and driver complexity.

This framework represents an important step forward: it shifts LLM deployment from being a cloud-only capability to one that is viable in real-world, resource-constrained, offline-first settings. MLC LLM serves as the foundation for this thesis's system implementation, enabling our custom-trained Aptly model to run entirely offline on iOS devices.

This thesis builds on these methods to push the boundary of on-edge generation further: from general natural language understanding to practical, offline app creation for all.

2.3 Tiny Machine Learning Techniques

Because mobile deployments face severe memory and compute bottlenecks, a variety of TinyML techniques have been proposed to compress models without sacrificing too much accuracy. Understanding these methods provides critical tools for bringing models like Aptly offline. These techniques are essential in addressing the challenges posed by the limited computational power and memory of edge devices, enabling efficient on-device processing and real-time decision-making [18].

2.3.1 Quantization

Quantization reduces the bit-width of model parameters, decreasing memory usage and accelerating inference through more efficient computation. Model quantization typically converts high-precision floating-point numbers to lower precision integers, such as 8-bit values, thereby significantly cutting down memory and computation costs while often preserving accuracy [11].

The basic idea is to map floating-point weights w to discrete lower-precision representations $\hat{w}$:

$$\hat{w} = \text{round}\left(\frac{w - \min(w)}{\Delta}\right) \quad \text{where} \quad \Delta = \frac{\max(w) - \min(w)}{2^b - 1}$$

where b is the number of bits (e.g., $b = 8$ for int8 quantization).

Quantization can be applied post-training (Post-Training Quantization, PTQ) or during training (Quantization Aware Training, QAT). Recent advancements have demonstrated that even aggressive 4-bit or 3-bit quantization can preserve the capabilities of LLMs, particularly when combined with calibration techniques [17].

2.3.2 Pruning

Pruning further reduces model size by removing parameters deemed unnecessary. It can be unstructured—zeroing individual weights—or structured—removing entire heads, neurons, or channels. Structured pruning methods like channel or filter pruning maintain the original network topology, which improves compatibility with hardware accelerators and maintains real-time performance [20].

Formally, pruning solves:

$$\min_{M,W} \mathcal{L}(M \odot W; D) \quad \text{subject to} \quad ||M||_0 \leq k$$

where M is a binary mask over the weight matrix W, and $\mathcal{L}$ is the loss over dataset D.

Pruned models often exhibit surprising resilience, motivating research into the "Lottery Ticket Hypothesis" [19]: the idea that sparse subnetworks exist which can train as effectively as full networks.

2.3.3 Knowledge Distillation

Distillation provides another route to smaller models by teaching a compact student network to replicate a teacher model's soft predictions [24]. This approach often retains much of the original model's performance while drastically reducing the number of parameters—making it highly compatible with edge deployments.

The training loss combines standard cross-entropy with a Kullback-Leibler divergence term:

$$\mathcal{L}_{\text{KD}} = \lambda \mathcal{L}_{\text{CE}}(y, \hat{y}) + (1 - \lambda)\mathcal{L}_{\text{KL}}(p_T, p_S)$$

where p_T and p_S are teacher and student softmax outputs, respectively.

Together, these techniques provide a powerful arsenal for adapting LLMs to constrained mobile environments—an essential step toward offline Aptly deployment.

2.4 Fine-tuning Techniques for LLMs

Compressing and optimizing a model for mobile deployment is only part of the challenge. To perform natural language-to-App Inventor code generation effectively, a model must also be *adapted* to this specific domain.

2.4.1 Standard Fine-tuning

Traditional fine-tuning retrains **all** parameters of the model on a task-specific dataset. Although straightforward, this approach is prohibitively resource-intensive for LLMs with billions of parameters.

2.4.2 Continued Pretraining

Continued pretraining [25] bridges the gap between general pretraining and task-specific fine-tuning. It involves further unsupervised training of a base model on domain-relevant text corpora. In the context of Aptly, continued pretraining on App Inventor-specific datasets could help the model better internalize the unique patterns of block-based programming.

2.4.3 Parameter-Efficient Fine-Tuning (PEFT)

To overcome the impracticality of full fine-tuning, Parameter-Efficient Fine-Tuning (PEFT) techniques have been developed.

LoRA [26] introduces small, trainable low-rank matrices (A, B) into each layer:

$$\Delta W = AB \quad \text{with} \quad A \in \mathbb{R}^{d \times r}, B \in \mathbb{R}^{r \times d}, r \ll d$$

where r (rank) is typically small (e.g., $r = 4$), drastically reducing the number of trainable parameters.

QLoRA [27] builds on LoRA by:

- Quantizing the model to 4-bit precision.
- Training LoRA adapters on top of frozen quantized weights.

QLoRA achieves remarkable efficiency, enabling fine-tuning of models as large as 65B parameters on consumer-grade GPUs (24GB VRAM).

These techniques are critical for this thesis, where domain-specific adaptation must be achievable within reasonable computational budgets.

2.5 Code Generation Benchmarks

Evaluating the output of a natural language-to-code generation system is fundamentally different from evaluating traditional NLP tasks. While standard text generation tasks (e.g., translation or summarization) often rely on surface-level metrics like n-gram overlap, code generation introduces stricter and more nuanced requirements. Code must not only resemble the correct answer but also compile, execute, and perform the intended behavior across diverse environments and use cases.

2.5.1 Motivation

Unlike natural language, source code is a highly structured language governed by strict syntax rules and unambiguous semantics. An incorrect indentation, bracket, or variable reference can render code invalid or cause it to behave incorrectly—even if it shares most of its surface features with the correct solution. Moreover, there are often many valid ways to implement the same functionality, from algorithmic variations to stylistic differences. As a result, metrics that rely purely on text similarity (like BLEU) are often misleading in code generation contexts, as they may penalize correct programs that differ in formatting or variable naming.

Therefore, effective evaluation of code generation systems must go beyond surface-level resemblance. It must account for both **syntactic correctness** (*does the code parse and compile?*) and **semantic correctness** (*does the code perform the correct task?*). These requirements have led to the development of code-specific benchmarks and metrics designed to assess generated programs more meaningfully.

2.5.2 Survey of Metrics

BLEU [28] score, originally developed for machine translation, is still widely used in code generation due to its simplicity and speed. It computes the n-gram overlap between generated code and a reference solution, providing a proxy for similarity. However, BLEU fails to capture the hierarchical and symbolic nature of programming languages, and it cannot assess whether the generated code compiles or behaves correctly. As a result, BLEU is often considered a weak signal for code generation quality, especially when used in isolation.

To address these shortcomings, the **HumanEval** benchmark [29] introduced a functional evaluation protocol for Python code generation. It consists of a curated set of programming problems, each paired with test cases that verify correctness. The primary metric, *pass@k*, estimates the probability that at least one out of k generated samples passes all unit tests:

$$\text{pass@k} = 1 - \prod_{i=0}^{k-1} \left(1 - \frac{c}{n-i}\right)$$

where n is the total number of generations and c is the number of correct ones. This metric focuses on functional accuracy, making it more aligned with real-world code usage, though it requires executable code and predefined test suites for every task.

CodeBLEU [30] expands upon BLEU by incorporating language-specific structure and semantics. It combines traditional n-gram matching with additional components that assess abstract syntax tree (AST) similarity, data flow consistency, and code syntax rules. This hybrid metric allows CodeBLEU to better reflect both the structure and behavior of programs, making it more robust than BLEU in scenarios where multiple implementations are possible.

Beyond these, several other datasets have been introduced to further benchmark code generation across domains and difficulty levels. **MBPP** (Mostly Basic Programming Problems) [31] provides short Python tasks with input-output-based test cases, geared toward beginner-level reasoning. The **APPS** dataset [32] contains a wide range of real-world programming interview questions with corresponding solutions, covering more complex logic

and algorithmic thinking. Meanwhile, **CodeContests** [33] focuses on competition-style problems drawn from platforms like Codeforces and Leetcode, pushing models to reason under constraints and optimize solutions.

Together, these benchmarks form a diverse and evolving ecosystem for assessing code generation models. They reflect the growing recognition that natural language-to-code evaluation requires a tailored approach—one that balances surface similarity, structural alignment, and functional correctness. In this thesis, we build on these foundations by developing a customized evaluation framework tailored specifically for App Inventor code, which presents its own structural and visual logic challenges not captured by existing benchmarks.

Table 2.1: Comparison of Code Generation Evaluation Metrics

Metric / Benchmark	**Syntax-Aware**	**Exec.-Based**	**Multi-Output Eval**	**Language-Specific**
BLEU [28]	×	×	×	×
HumanEval [29]	×	✓	✓	✓(Python)
CodeBLEU [30]	✓	×	✓	✓
MBPP [31]	×	✓	✓	✓(Python)
APPS [32]	×	✓	✓	✓(Python)
CodeContests [33]	×	✓	✓	✓(multi-lang)

These evaluation methods serve as important references when developing Aptly's custom evaluation framework and benchmarks in Chapter 4

2.6 Summary

This chapter reviewed the foundational concepts and techniques necessary for enabling offline, AI-powered mobile app creation. We began by outlining the computational and memory constraints of running large language models on edge devices, motivating the need for TinyML. We then explored three key model compression strategies—quantization, pruning, and knowledge distillation—as well as efficient fine-tuning methods suited for domain adaptation. Finally, we examined why conventional NLP benchmarks fall short for evaluating code generation, and surveyed specialized metrics that better capture program correctness. Together, these components establish the technical groundwork for designing, implementing, and evaluating an offline version of Aptly for MIT App Inventor.

Chapter 3

System Design: Aptly on Phone

This chapter presents the system design behind enabling Aptly's the natural language-to-app generation to run offline directly on the phone. Our core motivation is to support AI-powered mobile app generation in settings with limited or no internet access, thereby extending MIT App Inventor's mission by making AI-powered mobile app creation accessible to underserved communities. This required careful design choices across multiple dimensions: selecting an appropriate base model that balances performance with mobile feasibility, exploring fine-tuning techniques to specialize the model to Aptly's unique code format, and identifying lightweight inference frameworks capable of executing models on iOS devices. We began by analyzing the system constraints of modern phones and evaluating candidate models that could run efficiently under these limitations. A range of open-source language models was evaluated to identify a base model suitable for both fine-tuning and on-device deployment. We then fine-tuned our chosen model—LLaMA 3B—on a curated Aptly dataset using QLoRA to accommodate our small-scale training corpus. Finally, we leveraged MLC LLM, a mobile inference library, to successfully deploy the model onto an iPhone. The following sections walk through each design decision and engineering step, culminating in a live demonstration of Aptly running entirely offline on a phone, and closing with reflections and future directions for this work.

3.1 Background on Aptly and System Architecture

3.1.1 MIT App Inventor and the Role of Aptly

MIT App Inventor is a widely used visual programming environment that democratizes mobile app development by allowing users—especially novices—to build applications using drag-and-drop programming blocks. It has been used globally by millions of students and educators to create interactive Android and iOS applications [1]. While this block-based approach greatly reduces the entry barrier for non-programmers, creating apps still requires understanding control structures, event-based programming, and UI component configuration.

To lower this barrier even further, **Aptly** was developed as a natural language interface layered on top of MIT App Inventor [8]. The aim of Aptly is to enable users to create, edit, and understand App Inventor programs through conversational natural language commands,

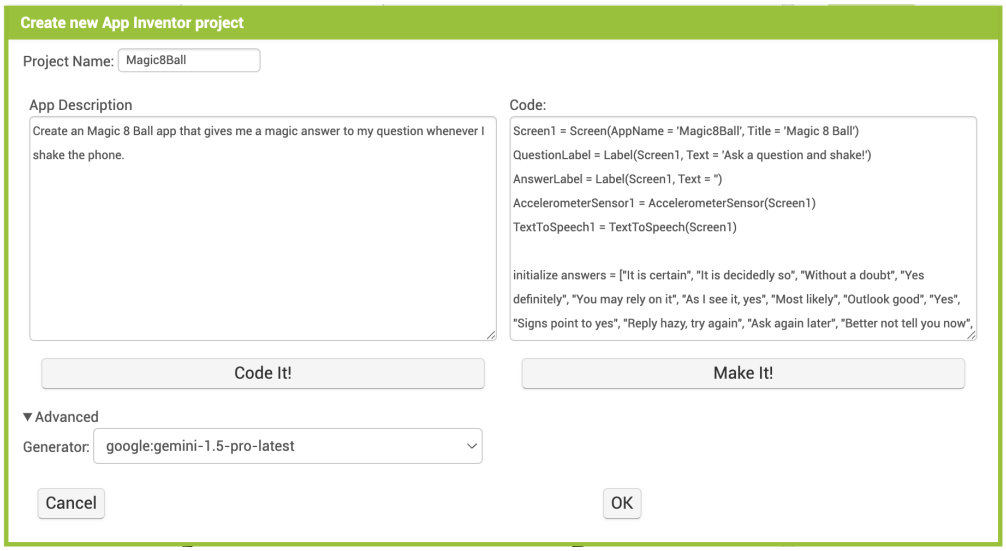


Figure 3.1: User can use natural language to describe their app, click on *Code It!*, and Aptly will help generate it.

thus broadening access to computational action [6]. When a user visits the Aptly site (aptly.appinventor.mit.edu), they are able to input the app description, click *Code It!*, and have Aptly generate their app, reducing the friction towards creating impactful mobile apps (as displayed in Figure 3.1. In the context of this thesis, Aptly serves as a bridge between large language model (LLM)-based code generation and visual programming, and is the core system evaluated for its feasibility in offline and mobile-first educational environments.

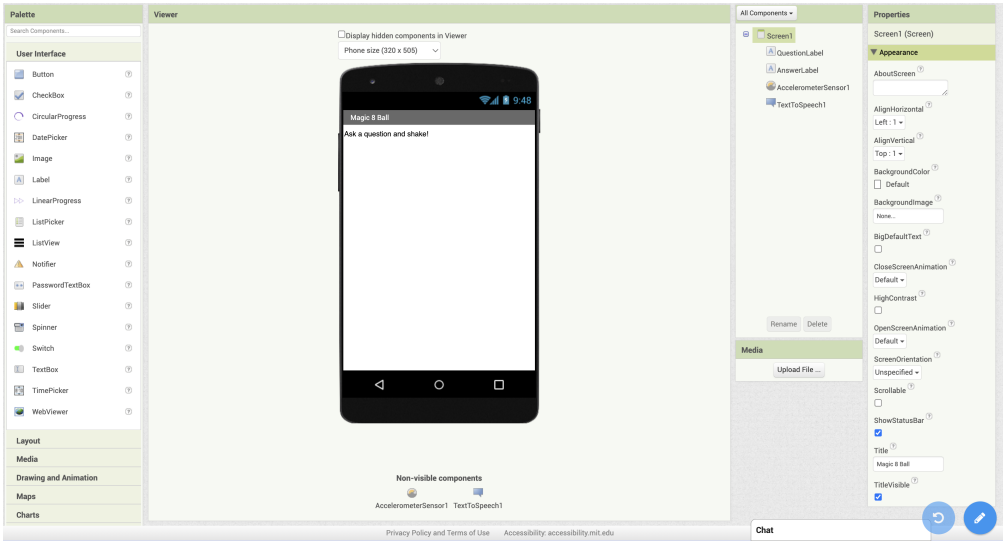


Figure 3.2: Magic 8 Ball App created by Aptly from description in Figure 3.1

3.1.2 The Aptly Language and Code Generation Pipeline

At the heart of Aptly lies a Python-inspired textual language that mirrors the structure of App Inventor blocks. This design choice was deliberate: Python's clean, pseudocode-like syntax enhances LLM compatibility, while the language itself was constrained to maintain a one-to-one mapping with App Inventor's block semantics. Each valid Aptly program is also a valid App Inventor program and vice versa. An example can be seen below:

```
Screen1 = Screen()
HA1 = HorizontalArrangement(Screen1)
Label1 = Label(HA1, Text = "Weight in lbs: ")
EarthWeight = TextBox(HA1, NumbersOnly = True)
PlanetList = ListView(Screen1, ElementsFromString = "Mercury, Venus, Mars,
    Jupiter, Saturn, Uranus, Neptune")
Calculate = Button(Screen1, Text = 'Calculate')
PlanetaryWeight = Label(Screen1)

initialize gravities = {
  "Mercury": 0.38, "Venus": 0.91, "Mars": 0.38,
  "Jupiter": 2.34, "Saturn": 0.93, "Uranus": 0.92,
  "Neptune": 1.12
}

to compute_weight(earth_lbs, planet):
  return earth_lbs * dictionaries_lookup(planet, global gravities, "not found"
      )

when Calculate.Click():
  set PlanetaryWeight.Text = call compute_weight(EarthWeight.Text, PlanetList.
      Selection)
```

Listing 3.1: Aptly program calculating planetary weight

The Aptly code above corresponds to the blocks below that would populate on the App Inventor interface.

To convert natural language to Aptly code, Aptly uses **few-shot prompt engineering**. When a user issues a request (e.g., "Make an app that translates my speech into one of four languages"), Aptly constructs a prompt consisting of:

- The user's natural language description D
- A set of example pairs $\langle d_i, c_i \rangle$ where d_i is a description and c_i is corresponding Aptly code

These example pairs are selected using **semantic similarity** computed via cosine distance on generated embeddings from either OpenAI or a third party service. This synthesized prompt is sent to an LLM (i.e. OpenAI's Codex/GPT models), which return Aptly code, subsequently parsed and rendered as App Inventor blocks [12, 34]. Refer to Fig 3.4 for the full flow.

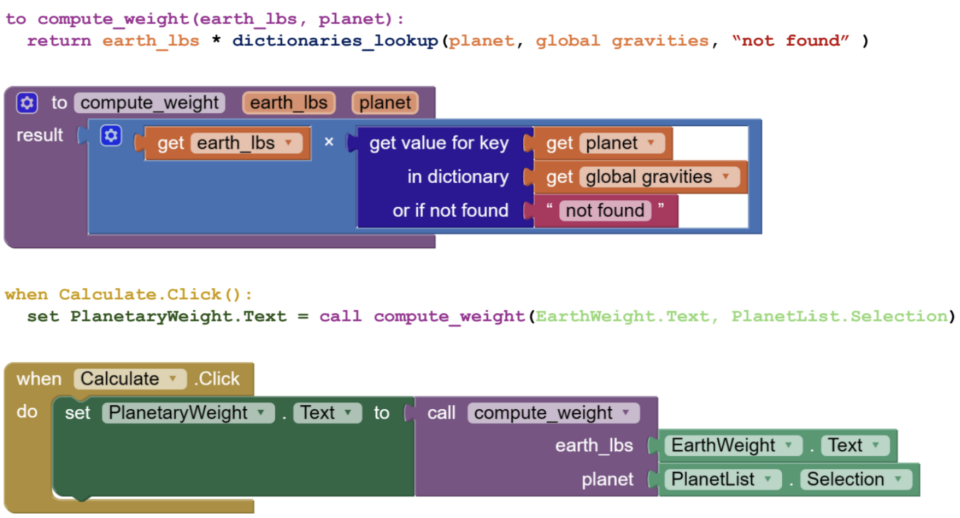


Figure 3.3: An example of the Aptly code in Listing 3.1.2 and its correspondence to the App Inventor interface.

3.1.3 System Architecture and Integration with App Inventor

Aptly is implemented as an external agent that participates in a collaborative App Inventor session using the Real-Time Collaboration (RTC) system. The system comprises five key components (see Figure 3.5):

- **MIT App Inventor**: The standard visual block-based IDE.
- **RTC Server**: Enables real-time collaborative editing by synchronizing actions across clients, including Aptly.
- **Aptly Server**: Converts natural language to Aptly code, computes diffs, and emits edit events to the RTC.
- **Third Party LLM**: Generates Aptly code given natural language and a few-shot prompt.
- **User Interface**: The user issues natural language commands from the web or mobile App Inventor client.

When a user provides a command, the following sequence occurs:

1. The App Inventor client transmits the current project (AIA file) and user command to the Aptly server.
2. The Aptly server translates the project to Aptly code using its internal `ProjectReader` and `Serializer`.
3. An edit prompt is generated with top-k similar example pairs and the current code.

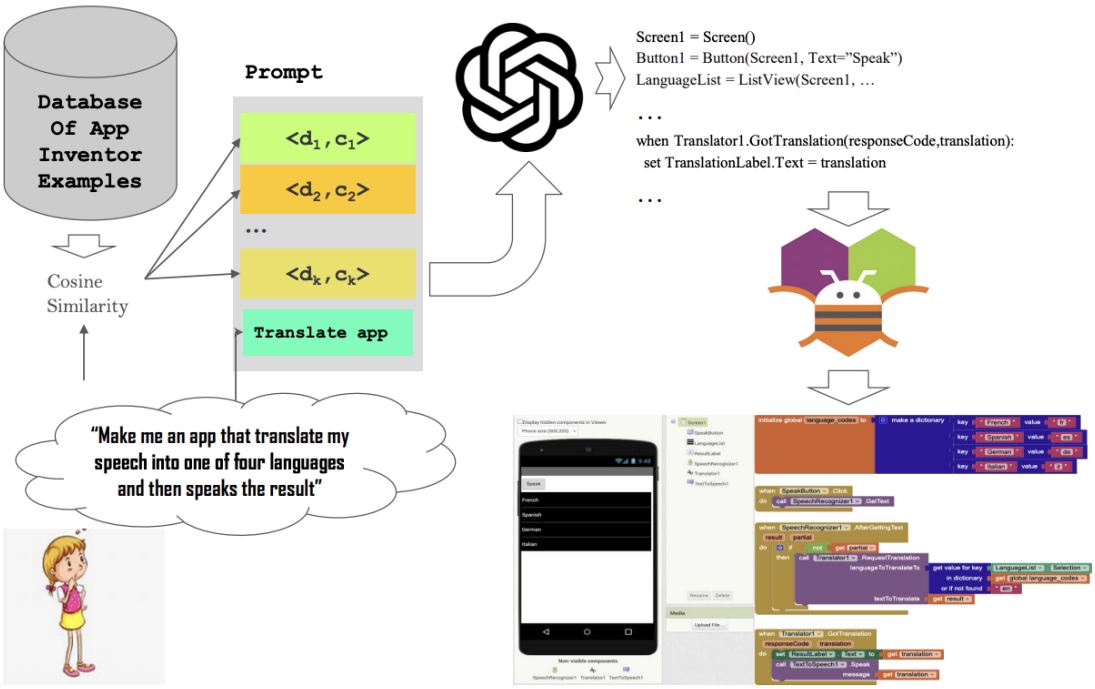


Figure 3.4: User flow for Aptly app generation: first the user offers a description for what they want to create, then we use semantic search to add several example pairs for few shot prompting; the constructed prompt is sent to a third party LLM, which return the Aptly code that gets parses into the App Inventor app on the web app [8].

4. The LLM returns updated Aptly code.
5. The Aptly server uses the Zhang-Shasha tree edit distance algorithm [35] to compute a sequence of INSERT, REMOVE, UPDATE, and MATCH operations between ASTs.
6. These are compiled into RTC events (e.g., `ComponentAdd`, `ComponentProperty`) and streamed into the live project.

This system allows Aptly to not only synthesize full apps from scratch but also perform **fine-grained edits** to existing projects. For example, given the request "add a label below the kitty that says 'pet the kitty,'" Aptly can parse the new code, calculate diffs, and emit RTC events to dynamically update the live project with minimal user intervention.

3.2 Aptly Only on the Phone Architecture

As mentioned in Section 3.1.3, the mobile version of Aptly is powered by three main services coordinated through the App Inventor app (see the architecture diagram) in Fig 3.5: the App Inventor server, the Aptly server, and the App Inventor Real-Time Collaboration (RTC) server.

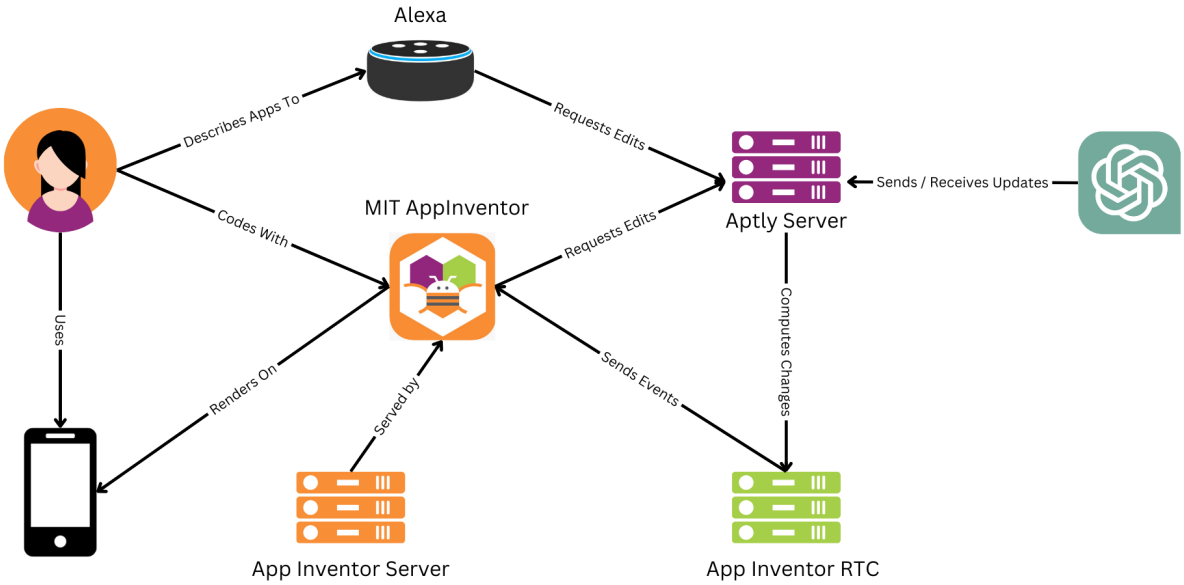


Figure 3.5: Interaction diagram for MIT App Inventor with Aptly [8].

The **App Inventor server** forms the core backend of the MIT App Inventor web interface, and is responsible for serving the web-based development environment and compiling projects from the visual block-based editor into executable app packages. It includes frontend assets that power the drag-and-drop design experience, as well as backend Java services for project compilation and persistence. In the context of Aptly on the phone, this component is only partially needed—specifically, the UI layer for displaying the generated app can be preserved without requiring the full web development interface.

The **Aptly server** is responsible for the natural language-to-code generation capabilities of the system. It is implemented in Python and acts as the orchestration layer for embedding user prompts, selecting relevant few-shot examples from a small internal dataset, formatting prompts for the LLM, and parsing the model's response into Aptly-compatible code structures. It acts as both a retrieval system and a prompt constructor. Initially, this server was designed to interface with cloud-based LLM APIs (e.g., OpenAI's GPT models), which poses an accessibility barrier in offline or low-connectivity environments. In the mobile version, we replicate this functionality locally by replacing the cloud LLM with a fine-tuned, on-device model, and executing the Aptly server logic within the mobile app itself. While the current implementation retains the core logic in Python, porting it to JavaScript or a mobile-native framework would be required for a fully integrated mobile stack—this is discussed in Chapter 6.

The **App Inventor Real-Time Collaboration (RTC) server** enables multiple users to edit the same project simultaneously [36]. It emits and listens for editing events, such as adding, deleting, or moving blocks, and ensures that all connected clients maintain a synchronized view of the project. Aptly participates in this architecture as a virtual collaborator, generating and inserting blocks in response to user interactions in real time.

The RTC server is implemented in JavaScript and designed for web-based collaboration. In the context of offline mobile use, this component is less critical—especially since users are not expected to perform real-time edits through the mobile UI.

Together, these three components define the full Aptly system. The App Inventor server serves and renders the UI, the Aptly server interprets natural language into code using a large language model, and the RTC server handles editing events and enables synchronous collaboration. In the traditional web-based workflow, a user enters a prompt into the Aptly interface (served by the App Inventor server), which is forwarded to the Aptly server for processing. The generated blocks are then injected into the shared editing environment via the RTC server, appearing in real time on the user's screen.

For an offline version of Aptly to run entirely on the phone, each of these components must either be migrated or reimagined for a local, mobile-only context. For the scope of this thesis, we address the following: (1) we approximate the functionality of the App Inventor server by extracting and bundling the necessary frontend components into the mobile app itself (in section 3.2.1), thereby bypassing the need for the full server or live web editing; (2) we omit the RTC server entirely, as real-time collaboration is unnecessary when editing is not being performed directly by the user; (3) most critically, we remove Aptly's dependency on a cloud LLM by designing and integrating an on-device inference engine powered by a fine-tuned version of LLaMA 3B (in sections 3.4). The full server is not migrated to JavaScript; future directions for porting the Aptly server and integrating RTC functionality are discussed in Chapter 6.

The remainder of this chapter details how each of these scoped components was implemented. Section 3.3 discusses model selection and system constraints. Section 3.4 describes the fine-tuning process with QLoRA. Section 2.2.1 introduces MLC LLM and outlines the deployment process to iOS. The chapter concludes with a working demo of Aptly running entirely offline and a discussion of next steps.

3.2.1 App Inventor Server on Mobile

To support offline usage of Aptly on a mobile device, we bypassed the need for the full App Inventor server by isolating only the essential frontend assets responsible for rendering the app interface. Rather than replicating the entire Java-based server environment, which typically runs on Google App Engine and handles a variety of backend services, we extracted the compiled JavaScript frontend—specifically `ode.js` and `index.js`—which powers the drag-and-drop interface in the browser.

These assets were then bundled into the mobile app and served locally through an embedded webview. This approach enables the rendering of Aptly-generated projects within the standard App Inventor UI, allowing users to view and interact with their generated apps directly on the phone without relying on any external server or internet connectivity.

To accomplish this, we configured a lightweight local server within the app to serve the static files, ensuring compatibility with the expected paths and dependencies of the App Inventor frontend. This solution provides a minimal but functional App Inventor frontend experience, sufficient to render and test generated applications locally. A working demo video can be found here. In future work, we plan to explore deeper integration of backend services—such as local project saving or on-device compilation—by replacing or

re-implementing server logic in mobile-native code.

3.3 Model Selection Under Mobile Constraints

3.3.1 System Constraints and Base Model Selection

Deploying large language models (LLMs) on smartphones requires careful consideration of the device's hardware limitations. Unlike data centers with dedicated GPUs and abundant memory, edge devices like modern laptops and mobile phones are limited in terms of power and memory bandwidth. A standard smartphone may only have 6–12 GB of RAM shared between the operating system and applications; higher end iPhones may feature up to 16 GB of RAM, but typical applications are restricted to around 5–6 GB of RAM usage [37]. As a result, deploying LLMs on these devices requires substantial model compression, quantization, and architectural optimization.

Given these constraints, our model selection criteria focused on two primary factors:

- **Runtime Memory Usage (Activations):** The model's peak RAM consumption during inference must remain within the per-app memory limits to prevent crashes.
- **On-Disk Storage Size (Weights):** The model's storage footprint must be sufficiently small to fit within the app size limitations, allowing room for other essential assets.

We ran preliminary experiments several compact LLMs, including TinyLlama (1.1B parameters), Anthropic Claude Instant (8B), Gemini Pro (8B), Gemini Flash (8B), and LLaMA 3 Instruct (available in 3B, 8B, and 70B variants).

While smaller models like TinyLlama offer fast inference and low memory usage, they lack the representational capacity to generalize across diverse domains. On the other hand, larger models such as Mistral 7B push the limits of edge deployment and require distillation or aggressive quantization. After empirical testing, we selected **LLaMA 3B** as the model of choice due to its balanced trade-off between expressivity and deployability. Additionally, LLaMA 3B is open-source, which enables fine-tuning on domain-specific tasks such as Aptly code generation and flexible integration into mobile deployment frameworks. Its relatively smaller footprint also makes it suitable for quantization without a significant loss in performance.

3.4 Finetuning LLaMA 3B

While LLaMA 3B offers impressive general-purpose capabilities, it lacks any understanding of the Aptly domain; Aptly's programming language is not open-source, so its internal syntax and structure are unfamiliar to off-the-shelf models. Thus, we perform domain-specific fine-tuning to enable the model to generate functional and valid code in the Aptly format.

Among the various finetuning methods (including the ones introduced in section 2), we decided to adopt the the QLoRA (Quantized Low-Rank Adapter) fine-tuning method [27] due to our dataset constraints (reference Section 4.1). Given the limited size of our dataset

(on the order of a few hundred natural language and Aptly code pairs), we chose the QLoRA fine-tuning method [27] because it is designed to train LLMs efficiently with minimal memory and compute resources. QLoRA uses quantization-aware low-rank adapters and offloads much of the training computation to GPU-efficient routines, making it feasible to fine-tune billion-parameter models on a single GPU node.

We conducted fine-tuning on Amazon SageMaker using a ml.g5.2xlarge instance equipped with a single NVIDIA A10G GPU. Training the LLaMA 3B model with QLoRA adapters converged within approximately 2 hours.

After training, we merged the LoRA adapters with the base LLaMA 3B model weights to create a standalone model ready for inference. The merged model demonstrated robust understanding of natural language prompts related to Aptly and produced structured, valid code blocks consistent with the platform's component and block paradigm. Below we provide a few illustrative examples comparing model outputs before and after fine-tuning.

Example prompt: Create an app that lets users draw on a canvas. There are 4 buttons: red, green, blue, and black. When each button is clicked, the canvas background changes to the selected color.

```
Screen1 = Screen(AppName = 'BallFling', ScreenOrientation = 'portrait', Title
    = 'Screen1')
Canvas1 = Canvas(Screen1, Height = -2, Width = -2)
Ball1 = Ball(Canvas1, PaintColor = '&HFF0000FF', Radius = 10, X = 190, Y =
    133)

when Ball1.Flung(x, y, speed, heading, xvel, yvel):
  set Ball1.Heading = heading
  set Ball1.Speed = speed

when Ball1.EdgeReached(edge):
  call Ball1.Bounce(edge)
```

Listing 3.2: Finetuned Llama Output Parses Correctly

```
Screen1 = Screen(AppName = 'BallFling', Sizing = 'Fixed', Title = 'Screen1')
Canvas1 = Canvas(Screen1, Height = -2, Width = -2)
Ball1 = Ball(Canvas1, PaintColor = '&HFF0000FF', Radius = 10, X = 190, Y =
    133)

when Ball1.Flung(x, y, speed, heading, xvel, yvel):
  set Ball1.Heading = heading
  set Ball1.Speed = speed

when Ball1.EdgeReached(edge):
  call Ball1.Bounce(edge)

when Button1.Click():
  set Ball1.Heading = math_random_int(225, 315)
  set Ball1.Speed = 10

when Button2.Click():
  set Ball1.Heading = math_random_int(225, 315)
  set Ball1.Speed = 10

```

```
when Button3.Click():
  set Ball1.Heading = math_random_int(225, 315)
  set Ball1.Speed = 10

when Canvas1.Dragged(startX, startY, prevX, prevY, currentX, currentY,
    draggedAnySprite):
  call Ball1.Flung(currentX, currentY, Ball1.Speed, Ball1.Heading, Ball1.Xvel,
      Ball1.Yvel)
```

Listing 3.3: Out of box Llama with few shot prompting generated syntatically incorrect code

3.5 Deploying LLM on the Edge

To enable offline inference on iOS devices, we utilized the MLC LLM framework [38], a mobile-friendly runtime that compiles LLMs into optimized, platform-specific binaries using Apache TVM and WebGPU. MLC LLM supports a range of models, including LLaMA variants, and enables quantization to 4-bit weights, drastically reducing memory usage while retaining accuracy.

We scoped our deployment to iOS devices and used MLC LLM's PyTorch-to-TVM compilation pipeline to convert the fine-tuned LLaMA 3B model into a format compatible with iOS. This involved the following steps:

1. *Model Quantization*: We applied symmetric 4-bit quantization to the merged LLaMA weights to reduce the memory footprint to 1.2 GB.

2. *TVM Compilation*: The model was compiled to iOS Metal shaders, enabling GPU-accelerated inference via Apple's Metal Performance Shaders.

3. *Swift Integration*: The compiled model was embedded into a Swift-based demo app, allowing natural language prompts to be inputted directly on-device.

The quantized and sharded model weights are available on Hugging Face at https://huggingface.co/wallie18/Aptly-Finetuned-3B-q4f16_1-MLC.

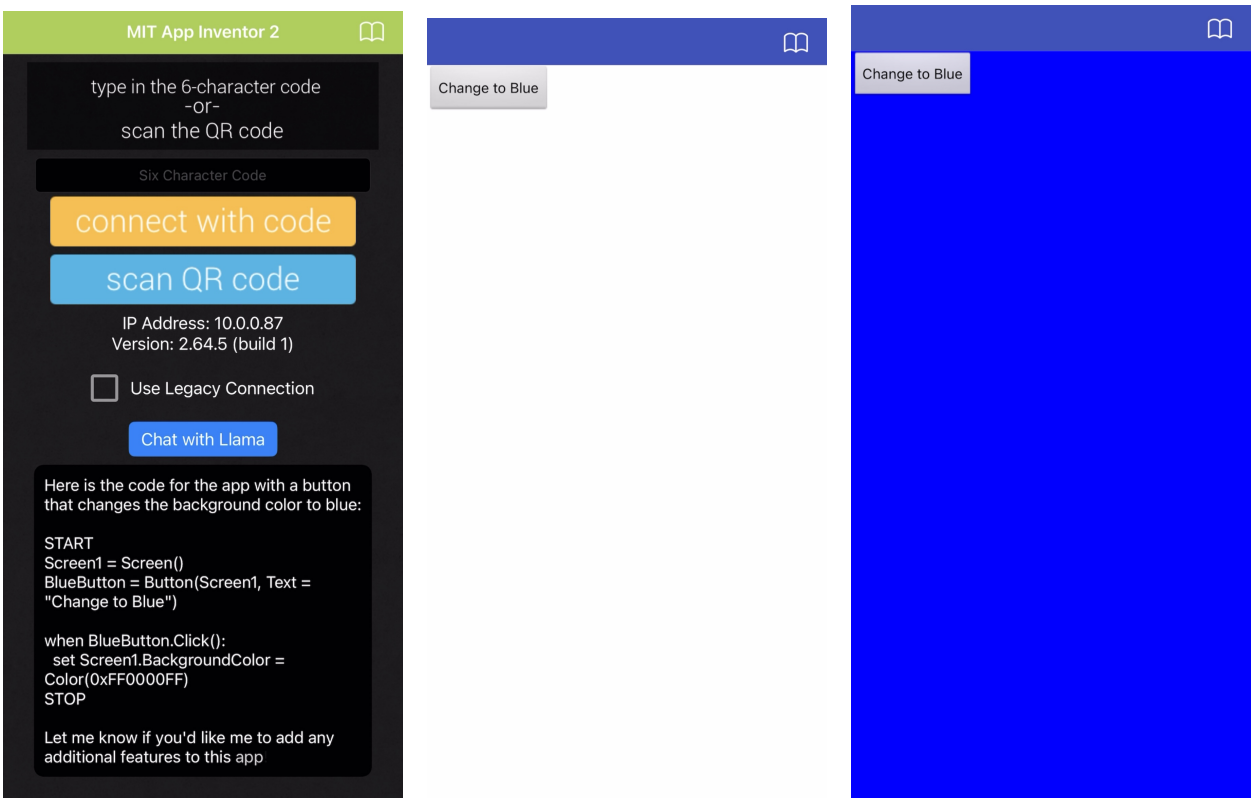

```
  {
"version": "0.1.0",
"model_type": "llama",
"quantization": "q4f16_1",
"model_config": {
  "hidden_size": 3072,
  "intermediate_size": 8192,
  "num_attention_heads": 24,
  "num_hidden_layers": 28,
  "rms_norm_eps": 1e-05,
  "vocab_size": 128256,
  "tie_word_embeddings": true,
  "position_embedding_base": 500000.0,
  "rope_scaling": {
    "factor": 32.0,
    "high_freq_factor": 4.0,
```

```
      "low_freq_factor": 1.0,
      "original_max_position_embeddings": 8192,
      "rope_type": "llama3"
    },
    "context_window_size": 3072,
    "prefill_chunk_size": 128,
    "num_key_value_heads": 8,
    "head_dim": 128,
    "tensor_parallel_shards": 1,
    "pipeline_parallel_stages": 1,
    "max_batch_size": 128,
    "disaggregation": false
  },
  ...
}
```

Listing 3.4: Snippet of Config file settings for finetuned Llama-3B for Aptly (reference Appendix B.2)

Deployment required several custom adjustments to package imports and memory management. Nonetheless, we successfully ran inference on-device.

Figure 3.6 shows a sequence of screenshots from the working demo, where the user spoke the prompt: *Make me an app with a button that changes the background color to blue*, and the system returns Aptly-generated code before parsing it to blocks rendered on-device; the rendered button is able to change the background color of the app blue. The full video demonstration is available at: https://www.youtube.com/shorts/lve7uS41sKk.



(a) Local Llama Generated Code (b) Generated Code Blocks (c) Fully Functional App

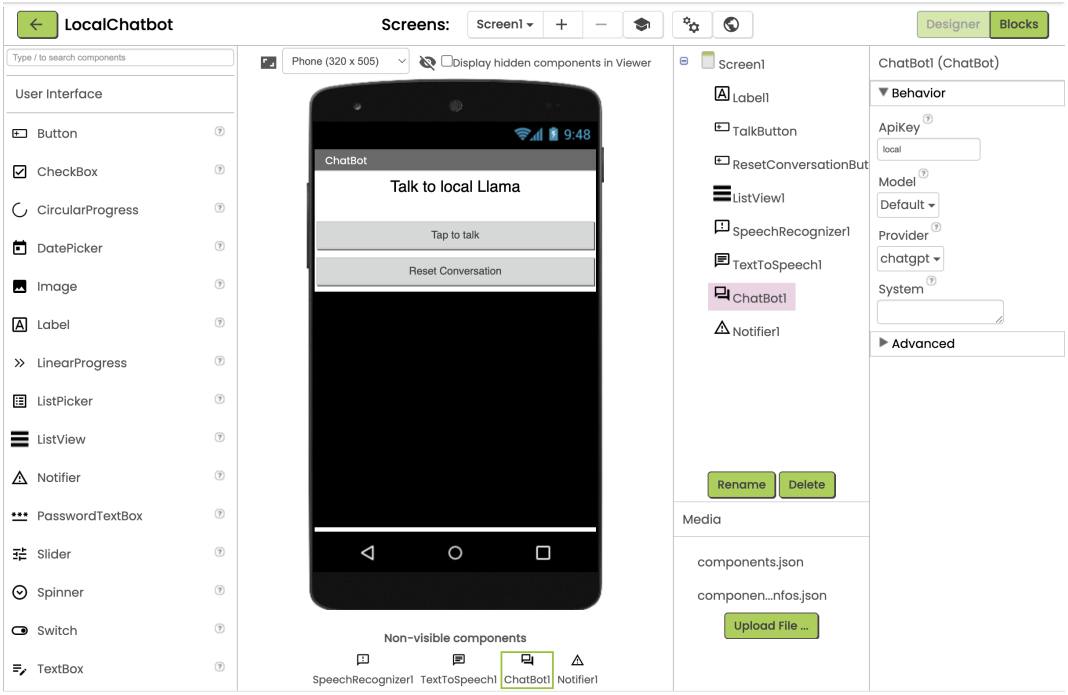
Figure 3.6: Aptly on the Phone: LLaMA 3B running locally through MLC LLM on iOS. Screenshots from the live demo, which can be viewed at this link.

3.5.1 Offline Chatbot via Local LLaMA Integration

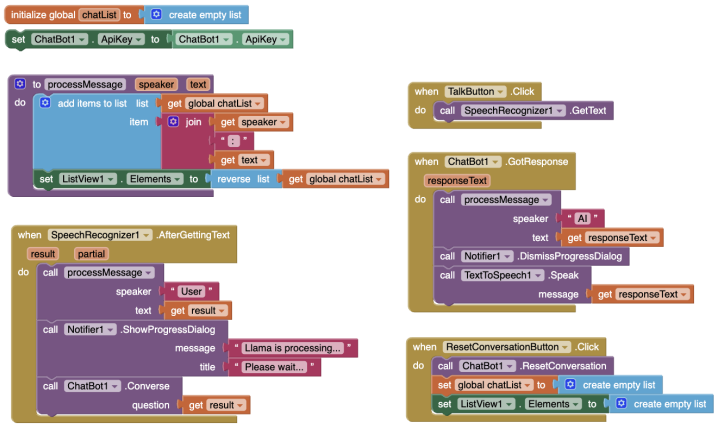
In addition to running Aptly-generated code blocks from speech, we also enabled the Aptly Chatbot Component to interact with a fully local LLaMA model. This functionality allows users to select `"local"` as the `ApiKey` within the ChatBot component. When this is done, the system routes all inference requests to a local version of LLaMA instead of relying on cloud-based APIs like OpenAI's ChatGPT. Figure 3.7 shows the LocalChatbot app interface, where speech input is transcribed, passed to the local LLaMA model, and the response is rendered in a simple chat UI.

The integration requires no code changes—only setting the `ApiKey` to `"local"`—and works entirely offline. After scanning the app with the MIT AI2 Companion and launching it on a device, users can switch their phone to airplane mode and continue chatting with the local LLaMA model, making this a true offline experience. This demonstrates that with only a few configuration changes, developers and educators can leverage the same Aptly interface for both online and offline use cases.

This deployment proves that it is technically feasible to support AI-driven mobile app generation entirely offline. Through this integration, Aptly becomes accessible not just to users with high-end devices or stable internet, but also to children in rural classrooms, students in refugee camps, and innovators in areas of limited technological infrastructure.



(a) Component Layout



(b) Blocks Editor

Figure 3.7: LocalChatbot App showing how to run a local LLaMA model offline. The API key is set to `"local"`, enabling fully on-device inference without any internet connection.

Chapter 4

Evaluation Framework

Evaluating Aptly-generated code presents a unique set of challenges that make existing code evaluation frameworks insufficient. Aptly is a grammar and tool built specifically for MIT App Inventor, enabling natural language prompts to be translated into an intermediate code format. This Aptly code, inspired by a Python-like syntax, can be parsed and tokenized into `.aia` files —project files that MIT App Inventor can render into fully functional mobile apps. Aptly thus bridges natural language and App Inventor projects, supporting both code generation from prompts and parsing existing apps back into Aptly descriptions.

However, evaluating Aptly outputs is particularly difficult due to several factors:

- **Non-standard, non-open-source language**: Aptly's grammar and tooling are not fully open-source. Unlike common programming languages such as Python or JavaScript, Aptly cannot leverage off-the-shelf parsers, linters, or evaluation frameworks. There is also no direct programmatic way to verify if generated code behaves correctly without manual inspection in the App Inventor environment.
- **Lack of ground truth data**: There is no large-scale, publicly available dataset pairing natural language prompts with correct Aptly code. Without ground truth examples, traditional supervised evaluation is not possible.
- **No automatic execution testing**: Unlike conventional code generation tasks, there is no straightforward way to run unit tests or check execution outputs. App Inventor apps must be deployed inside the platform to test functionality, making scalable automated evaluation infeasible.
- **Partial correctness**: Even when generated code is not fully correct, it may still achieve parts of the intended functionality. Standard exact-match metrics fail to capture degrees of partial correctness or semantic similarity between generated and reference programs.

Given these limitations, a custom evaluation framework is necessary to meaningfully assess Aptly's performance. In response, we developed a dedicated evaluation pipeline that addresses each of these challenges:

- **Dataset creation** (section 4.1: We constructed a benchmark dataset consisting of natural language prompts paired with carefully designed reference Aptly code, addressing the lack of ground truth data.

- **Evaluation metric adaptation** (section 4.2: We surveyed evaluation techniques from the broader natural language-to-code literature, including HumanEval and CodeBLEU, and selected CodeBLEU as the basis for our framework. CodeBLEU's structure, syntax, and semantic-aware components are well-suited for capturing partial correctness.

- **Tree-sitter parser for Aptly** (section 4.3: To enable syntactic and structural analysis, we implemented a custom Tree-sitter parser for the Aptly language. This parser tokenizes and produces abstract syntax trees (ASTs) from Aptly code, a crucial step for metric computation.

- **Tailored evaluation pipeline** (section 4.4: Finally, we combined the dataset, parser, and adapted metrics into a full evaluation pipeline capable of assessing Aptly outputs both syntactically and semantically.

This evaluation framework lays the groundwork for systematically measuring Aptly's strengths, identifying failure modes, and guiding future improvements in offline natural language-to-app generation.

4.1 Dataset Construction

4.1.1 Dataset Overview

The datasets collected for this thesis consist of paired natural language descriptions and Aptly code snippets. Aptly is a domain-specific language designed to express the component structure and event logic of MIT App Inventor apps in a readable and structured format. Each data point is a (description, code) pair: the description conveys the intended app behavior in plain English, and the code provides an abstract yet executable specification that captures the app's logic and layout.

For instance, the following example illustrates a "Magic 8 Ball" app that speaks a random answer aloud when the user shakes the device and asks a question:

> *"Create a magic 8 ball app with a picture of the magic 8 ball that gives the user an answer that a real 8 ball would give when they ask the app a question out loud and shake the device."*

```
START
Screen1 = Screen()
Magic8BallImage = Image(Screen1, Picture = "magic8ball.png")
QuestionLabel = Label(Screen1, Text = "Ask the magic 8 ball a question, then
    shake the phone")
AnswerLabel = Label(Screen1, Text = "")
TextToSpeech1 = TextToSpeech(Screen1)
AccelerometerSensor1 = AccelerometerSensor(Screen1)

initialize PredictionsList = ["Outlook certain", "Yes", "No way", "It is
    certain",
```

```
                                "Not likely", "Reply hazy", "Maybe", "Without a
                                    doubt"]

when AccelerometerSensor1.Shaking():
    set AnswerLabel.Text = lists_pick_random_item(global PredictionsList)
    call TextToSpeech1.Speak(AnswerLabel.Text)
STOP
```

Listing 4.1: Aptly code for the Magic 8 Ball app

This structure supports the training and evaluation of models capable of generating full Aptly programs from natural language input, facilitating automatic app creation and providing a foundation for studies in natural language programming.

4.1.2 Datasets

To evaluate the Aptly code generation system, we curated four distinct datasets representing a range of difficulty levels, use cases, and origins. The first dataset, **repo_examples** (164 files), consists of simple, curated examples originally used within the Aptly project to support few-shot prompting. These examples are concise, typically focusing on a narrow component of MIT App Inventor, and were designed to maximize coverage across the Aptly syntax space. We split this dataset using an 85%/15% train/test split. To avoid data leakage, we explicitly removed the 15% reserved for testing from all few-shot prompt contexts and training data. The examples range in length from 5 to 64 lines of code, with a mean of 16.57 lines and a standard deviation of 10.87.

The second dataset, **tutorial_examples** (18 files), consists of comprehensive educational examples used to teach MIT App Inventor to beginners. These include more complex projects such as making a drawing canvas or building an AI chatbots. Owing to their complexity and the small total sample size, we adopted a 60%/40% train/test split. These examples are longer, with line counts ranging from 24 to 106, a mean of 51.83, and a standard deviation of 26.14.

The third dataset, **manual_labeled** (45 files), contains examples created by real (anonymized) users of App Inventor. We manually labeled these files with natural language descriptions to support supervised learning evaluation. These examples were selected to capture a diverse range of app functionalities and structural completeness. Some examples of descriptions of the apps include: *A note taking apps that lets users write and save notes, read old notes, and make new notes* and *A music player with buttons to play different genres like metal, rap, and adrenaline-pumping tracks*. Like the tutorial examples, we used a 60%/40% train/test split. This dataset has a wider range in complexity, with file lengths ranging from 9 to 147 lines, a mean of 48.69, and a standard deviation of 27.12. The spreadsheet with the dataset info can be found here

After individual processing, we aggregated these 3 training sets into a unified training set, and all testing sets into a single held-out evaluation set. This structure was chosen to allow models to generalize across varying complexities and content types while maintaining a clear and fair separation between seen and unseen data.

The fourth dataset, **ai_labeled** (2045 files), was constructed from anonymized, complete app examples created by real users. To ensure diversity and reduce redundancy, we filtered the

raw examples by computing pairwise cosine similarities between their embeddings, generated using a pre-trained CodeBERT model [39]. Specifically, snippets with a similarity score greater than 0.99 were considered redundant and removed. After filtering, we generated natural language descriptions for the remaining examples using few-shot prompting with OpenAI models, thereby creating new (description, code) pairs. Compared to the manually curated datasets, `ai_labeled` is substantially larger, with code examples varying from 3 to 174 lines, a mean length of 12.28 lines, and a standard deviation of 8.45. We randomly partitioned this dataset into five equal parts for use in 5-fold cross-validation.

Table 4.1: Line Count Statistics for Each Dataset

Dataset	Mean	Std	Min	25%	50% (Median)	75%	Max
`repo_examples (164)`	16.57	10.87	5	9.75	13.00	21.00	64
`tutorial_examples (18)`	51.83	26.14	24	32.50	43.00	60.75	106
`manual_labeled (45)`	48.69	27.12	9	34.00	45.00	55.00	147
`ai_labeled (2045)`	12.28	8.45	3	7.00	11.00	16.00	174

4.2 Metric Selection and Adaptation

Unlike standard natural language generation tasks, code generation (as in the case with Aptly) imposes stricter structural, syntactic, and semantic constraints. A code snippet may be superficially similar to a reference output yet fail to execute correctly, or conversely, may differ in wording but still correctly implement the intended functionality. As a result, traditional natural language evaluation metrics—such as BLEU, ROUGE, or METEOR—are insufficient for assessing code quality.

In evaluating Aptly, we needed a metric that could capture not just token-level similarity, but also structural and functional correctness. To guide our decision, we conducted a preliminary literature review of natural language to code benchmarks and evaluation methodologies (as detailed in section 2.5). This survey helped us identify key requirements for code evaluation: sensitivity to syntax and structure, robustness to minor surface variations, and, where possible, reflection of functional correctness.

4.2.1 Preliminary Experimentation: HumanEval and CodeBLEU

To better understand the behavior of available metrics, we conducted preliminary experiments using out-of-box large language models and standard code generation datasets. Specifically, we experimented with: *OpenAI GPT-3.5, OpenAI GPT-4, Gemini Pro, Gemini Flash, Meta Llama3 8B Instruct, Anthropic Claude 3.5, Anthropic Claude Instant.*

We evaluated model outputs on existing Python datasets, including CONCODE [40] and MBPP (Multi-lingual Python Programming Benchmark) [41]. Our goal was to investigate the strengths and limitations of different evaluation strategies—specifically surface-form matching (e.g., BLEU score), structural matching (e.g., CodeBLEU), and executable correctness (e.g., HumanEval).

HumanEval Metrics

HumanEval [29] is a benchmark designed for code generation evaluation through execution-based tests. Each problem consists of a Python function signature and a hidden set of test cases. Generated code is assessed based on its ability to pass these tests. HumanEval primarily reports the *pass@k* metric, which measures the probability that at least one of k generated samples passes all test cases.

For example, if a model generates five candidate functions for a prompt like "reverse a linked list," *pass@5* measures whether any one of those five successfully reverse the list under hidden tests.

Pros: The benefits are that this captures true functional correctness, and is robust to stylistic variations.

Cons: The drawbacks are that it requires an executable environment, which Aptly lacks outside of App Inventor. The utility also depends heavily on the quality and coverage of test cases, making it infeasible for Aptly without significant infrastructure development (discussed in Future Work, Chapter 6).

Thus, while HumanEval metrics are ideal in theory, they are impractical for current Aptly evaluation.

CodeBLEU Metrics

CodeBLEU [30] extends the traditional BLEU score by incorporating signals crucial for code:

- **Weighted n-gram match**: Like BLEU, but adjusts weights for code-specific tokens.
- **Syntax match**: Compares abstract syntax trees (ASTs) for structural correctness.
- **Data-flow match**: Considers variable and data dependencies in the code.
- **Semantic match**: Optionally considers equivalence transformations (though rarely feasible without execution).

To illustrate the advantage over plain BLEU, consider the following example:

- **Reference Code**:

```
def find_Volume(l, b, h):
    return ((l * b * h) / 2)
```

- **Candidate Code**:

```
def triangular_prism_volume(base, height, depth):
    return (base * height * depth) / 2
```

The corresponding evaluation results are:

- BLEU Score: **0.0255**
- CodeBLEU Score: **0.3892**

- N-gram Match Score: 0.0255
- Weighted N-gram Match Score: 0.0312
- Syntax Match Score: 0.5
- Dataflow Match Score: 1.0

This example highlights that although the BLEU score is very low due to minimal n-gram overlap (e.g., different variable and function names), CodeBLEU assigns a significantly higher score by recognizing structural and semantic similarities. Specifically, the syntax match (0.5) and perfect dataflow match (1.0) demonstrate that the candidate code preserves the intended functionality of the reference despite surface-level differences. Thus, CodeBLEU better reflects functional correctness, while BLEU remains sensitive only to superficial token matches.

Pros: CodeBLEU captures structural and syntactic similarity, and is more tolerant of stylistic variation than BLEU, but does not require executable outputs.

Cons: : However, CodeBLEU does not guarantee functional correctness, and is still surface-level in cases of deep semantic divergence.

Through these experiments, we concluded that CodeBLEU provides the best balance between feasibility and informativeness for Aptly evaluation.

4.2.2 Adapting CodeBLEU for Aptly

While CodeBLEU provided a strong foundation, it was not directly applicable to Aptly. Aptly's Python-inspired syntax, customized for block-based code generation, does not align with standard programming language grammars. To address this:

- We implemented a custom **Tree-sitter parser** for Aptly, capable of producing structured ASTs.
- We modified CodeBLEU's syntax matching and n-gram tokenization modules to align with Aptly's conventions.
- We recalibrated the weights across CodeBLEU's sub-metrics to emphasize structural matching and de-emphasize surface token matches.

Additionally, recognizing the importance of program loadability in the absence of execution testing, we integrated **parsing success** into our evaluation pipeline: only programs that successfully parse into valid App Inventor projects without errors are considered viable outputs.

In summary, we chose to adapt CodeBLEU rather than rely on out-of-box metrics because of the considerable stylistic variability in Aptly-generated code. Small differences such as screen names, variable naming conventions, and minor block variations could otherwise significantly distort evaluation. By tightly integrating Aptly's syntax into CodeBLEU, we established a strong foundation for a rigorous and extensible evaluation framework, upon which future executable or functionality-based extensions can be layered.

Table 4.2: Comparison of Evaluation Metrics for Code Generation

Metric	**BLEU**	**HumanEval (pass@k)**	**CodeBLEU**
Type	Surface-level n-gram match	Execution-based correctness	Structural and semantic match
Execution Required	No	Yes	No
Captures Syntax	No	Indirectly	Yes (via AST match)
Captures Semantics	No	Yes (via test cases)	Partially (via data-flow match)
Variable Renaming	No	Yes	Yes
Language Agnostic	Yes	Limited (primarily Python)	Requires language-specific parsers
Advantages	Simple to compute; widely used	Measures actual functionality	Considers structure and semantics
Limitations	Ignores code structure and meaning	Needs executable environment	Requires AST and data-flow analysis

4.3 Tree-sitter Parser for Aptly

Evaluating Aptly-generated code at a structural level required more than token comparisons – we needed a way to parse Aptly code into structured representations, such as abstract syntax trees (ASTs). While Aptly includes its own parser for converting natural language into App Inventor code, it lacks the capabilities provided by Tree-sitter, such as incremental parsing and integration with tools like CodeBLEU that rely on detailed syntax trees to evluate syntax and dataflow matches.

To address this, we developed a custom Tree-sitter parser for Aptly. Tree-sitter [42] is an open-source incremental parsing library that generates parsers based on user-defined grammars. The process of developing a Tree-sitter grammar for Aptly involved:

- **Defining the Grammar**: We wrote a formal grammar specification for Aptly in `grammar.js`, covering core constructs such as screen declarations, UI component initialization, property settings, and event handler definitions.
- **Generating the Parser**: Utilizing Tree-sitter's CLI tools, we generated C-based parsing code from the grammar file.
- **Testing the Parser**: We built and tested the parser by running it against example Aptly code snippets to ensure accurate syntax tree generation.

The full parser code is open-sourced at https://github.com/joyce-yuan/tree-sitter-aptly. For instance, given the following Aptly code:

```
Screen1 = Screen()
Label1 = Label(Screen1, text="Hello, World!")
Button1 = Button(Screen1, text="Click Me")

when Button1.Click():
    call Notifier1.ShowMessageDialog("Hello, World!", "Greeting", "OK")
```

the Tree-sitter parser produces the following syntax tree:
The resulting syntax tree is:

```
program
|--- screen_block
|   |--- component_decl
|   |   |--- identifier
|   |   \--- identifier
|   |--- component_decl
|   |   |--- identifier
|   |   |--- identifier
|   |   \--- component_args
|   |       \--- designer_properties
|   |           \--- designer_property
|   |               |--- identifier
|   |               \--- designer_value
|   |                   \--- string
|--- component_decl
|   |--- identifier
|   |--- identifier
|   \--- component_args
|       \--- designer_properties
|           \--- designer_property
|               |--- identifier
|               \--- designer_value
|                   \--- string
|   |--- identifier
|   |--- identifier
|   \--- commented_statement
|       \--- statement
|           \--- method_call
|               |--- identifier
|               |--- identifier
|               \--- argvalues
|                   |--- expression
|                   |--- expression
|                   \--- expression
```

This parser forms the backbone of the structural analysis in our evaluation framework, enabling syntax tree comparisons and facilitating customized CodeBLEU scoring.

4.4 Custom CodeBLEU-Based Evaluation Pipeline

Building on the Aptly Tree-sitter parser, we constructed a custom CodeBLEU-based evaluation pipeline tailored to Aptly's syntax and semantics. The goal was to integrate parsing, weighted

n-gram matching, and structural analysis into a unified framework for evaluating natural language-to-Aptly code generation.

To implement the full evaluation framework, we made several modifications based on the original CodeBLEU infrastructure [43]:

- **Building the Aptly Parser**: First, we generated a compiled shared library version of our Tree-sitter Aptly parser. This involved running `tree-sitter generate` to produce parser C files, compiling these into a shared object file (`aptly.so`), and placing the compiled parser into the CodeBLEU evaluation directory. This step ensured that the CodeBLEU scripts could dynamically load and parse Aptly code during evaluation, as described previously in Section 4.3.

- **Defining aptly_dfg.py**: We created a module, `aptly_dfg.py`, to extract data flow graphs (DFGs) from Aptly code. A DFG captures the flow of variables and values through a program—mapping how data is defined, used, and passed across different parts of the code. Writing `aptly_dfg.py` involved walking the Aptly syntax trees produced by Tree-sitter, identifying variable assignments and usages, and constructing corresponding DFG edges. This enabled CodeBLEU's dataflow matching component to operate correctly for Aptly.

- **Providing aptly.txt**: We defined a file `aptly.txt` listing Aptly-specific keywords and operators. This file was used to compute weighted n-gram matching, giving greater importance to critical tokens like `Screen`, `Label`, `Button`, and `when`.

Once the infrastructure was set up, the evaluation pipeline could be run using a command such as:

```
python calc_code_bleu.py \
    --refs path/to/reference.txt \
    --hyp path/to/hypothesis.txt \
    --lang aptly \
```

where `reference.txt` and `hypothesis.txt` contain one Aptly program per line, corresponding to the ground truth and model-generated outputs, respectively.

An example output from the evaluation pipeline (example placeholder):
As an illustration, consider the following evaluation:

- **reference.txt**:

```
Screen1 = Screen()
Label1 = Label(Screen1, text="Hello, World!")
Button1 = Button(Screen1, text="Click Me")

when Button1.Click():
    call Notifier1.ShowMessageDialog("Hello, World!", "Greeting", "OK")
```

- **hypothesis.txt**:

```
Screen1 = Screen()
Label1 = Label(Screen1, text="Hello!")
Button1 = Button(Screen1, text="Press Me")

when Button1.Click():
    call Notifier1.ShowMessageDialog("Hello!", "Message", "OK")
```

- **Evaluation Results**:
  - N-gram Match Score: 0.4301
  - Weighted N-gram Match Score: 0.4410
  - Syntax Match Score: 1.0
  - Dataflow Match Score: 1.0
  - **Overall CodeBLEU Score**: 0.7178

The example above shows that, with a single evaluation command, we can now assess Aptly-generated code across multiple dimensions—including syntactic validity, structural alignment, and dataflow consistency—without requiring manual inspection or executable test environments. This addresses key limitations we previously identified with traditional evaluation metrics and enables scalable, reproducible evaluation of offline natural language-to-code generation.

By adapting CodeBLEU for Aptly and integrating our custom parser and DFG tool, we established a reliable and extensible evaluation framework for measuring the quality of Aptly code generation outputs, forming the basis of the experiments and benchmarks presented in this thesis.

Chapter 5

Results and Analysis

This section takes a deep dive into how well Aptly's code generation performs—both with models we use today and with newer, fine-tuned versions designed to run fully offline. These experiments directly address the two motivating questions from Section 1.3: *(1) Can we make Aptly work effectively on mobile devices without an internet connection?* and *(2) How can we reliably evaluate the quality of the code it generates?*

To explore these questions, we evaluate Aptly's performance from multiple angles. First, we use our custom evaluation framework (introduced in Section 4.2.2)—which combines a Tree-sitter parser for Aptly and a modified version of CodeBLEU—to benchmark out-of-box large language models such as GPT-4, Claude, and Gemini. Next, we evaluate the robustness of our fine-tuning pipeline via a 5-fold cross-validation study on a fine-tuned LLaMA model. This controlled analysis helps establish whether the model generalizes consistently across data splits and if improvements in parseability and structure are systematic. Finally, we compare a range of fine-tuned models—including on-device deployments—against commercial APIs to assess their effectiveness.

[Placeholder: This is where we'll summarize key takeaways—like the best-performing model setups, how much improvement fine-tuning achieved, and lessons learned from failure cases or parsing issues.]

The rest of this section thoroughly explores the results through a multi-perspective evaluation:

- **Section 5.1: Evaluation of Out-of-Box Models.** We benchmark several widely-used models across different prompting methods and highlight their strengths and weaknesses when applied to Aptly.
- **Section 5.2: Cross-Validation Evaluation of Fine-tuned LLaMA.** We assess whether fine-tuning on Aptly-specific data leads to consistent performance and structural improvements across multiple data splits.
- **Section 5.3: Evaluation of Fine-Tuned Models.** We show how models customized with Aptly-specific training data—especially those optimized for mobile use—perform relative to commercial APIs.
- **Section 5.4: Preliminary User Studies and Additional Experiments.** We

include some exploratory experiments and early user study results to round out our understanding of Aptly's capabilities and usability.

By the end of this section, we aim to:

- Understand how well Aptly currently works in generating valid and useful code;
- Identify which models and prompting strategies give the best results;
- See how much of a difference fine-tuning makes, especially for offline use on edge devices.

5.1 Evaluation of Out-of-Box Models

To understand the baseline performance of Aptly's current generation pipeline, we utilize our newly developed evaluation framework (Section 4) to evaluate a variety of out-of-box LLMs that are used in the current pipeline. Previously, Aptly lacked a reliable metric given the tool's non-pulic syntax, but our developed Aptly-specific CodeBLEU variant, combined with parse success tracking via our Tree-sitter parser, enables us to rigorously benchmark model performance across both syntactic validity and semantic fidelity.

5.1.1 Models and Prompting Methods

We evaluate a representative set of state-of-the-art LLMs from three major providers:

- **OpenAI**: *gpt-3.5-turbo*, *gpt-4*
- **Anthropic / Amazon Bedrock**: *claude-3-sonnet*, *claude-3.5-sonnet*, *meta.llama3-70b-instruct*
- **Google Gemini**: *gemini-1.5-pro-latest*

Each model is tested using three distinct prompting strategies, reflecting the different ways Aptly has historically been used to generate code:

1. **Examples (Few-Shot Prompting)**: We retrieve a small set of similar natural language–code examples using embedding-based similarity search. These examples are prepended to the prompt to guide the model's output.
2. **Rules (Rules-Based Prompting)**: We include a formal definition of Aptly's grammar as a system message, instructing the model to follow the rules when generating code. A sample excerpt is shown below (full system text can be found in Appendix B.1):

```
Aptly is a programming language used to describe programs for MIT App Inventor.
Aptly can be described by the following grammar:

program = screen_block code_block*
screen_block = component_decl* '\n'
component_decl = identifier '=' identifier '(' component_args? ')'
component_args = parent_component | designer_properties | ...
```

Listing 5.1: Excerpt from Aptly System Prompt

3. **Examples + Rules (Hybrid Prompting)**: We combine both strategies, providing grammar rules in the system message and similar examples in the user prompt. This hybrid approach reflects the current production setup used in Aptly's deployed version.

These strategies represent three different tradeoffs: examples alone provide grounded reference points but little structural guidance; rules alone emphasize correctness but may lack semantic nuance; the combined approach aims to balance both.

5.1.2 Evaluation Setup

For each model-prompting pair, we evaluate on a held-out test set of [insert number] natural language prompts and their corresponding Aptly programs. We record:

- **CodeBLEU Score**: As adapted in Section 4.2.2, this measures structural and semantic correctness of the generated Aptly code.
- **Parsing Success Rate**: Whether the generated code parses successfully into a valid `.aia` file that MIT App Inventor can open and render.

These dual metrics give us a holistic view: CodeBLEU captures semantic intent, while parse success measures syntactic correctness and viability in deployment.

5.1.3 Results Summary

In particular, we are interested in identifying (1) which models generate the most accurate and syntactically correct Aptly code, and (2) which prompting strategy yields the best results.

Figure 5.1 are the normalized CodeBLEU scores across evaluated across a held out test set of 25 examples, and whether or not the code parsed.

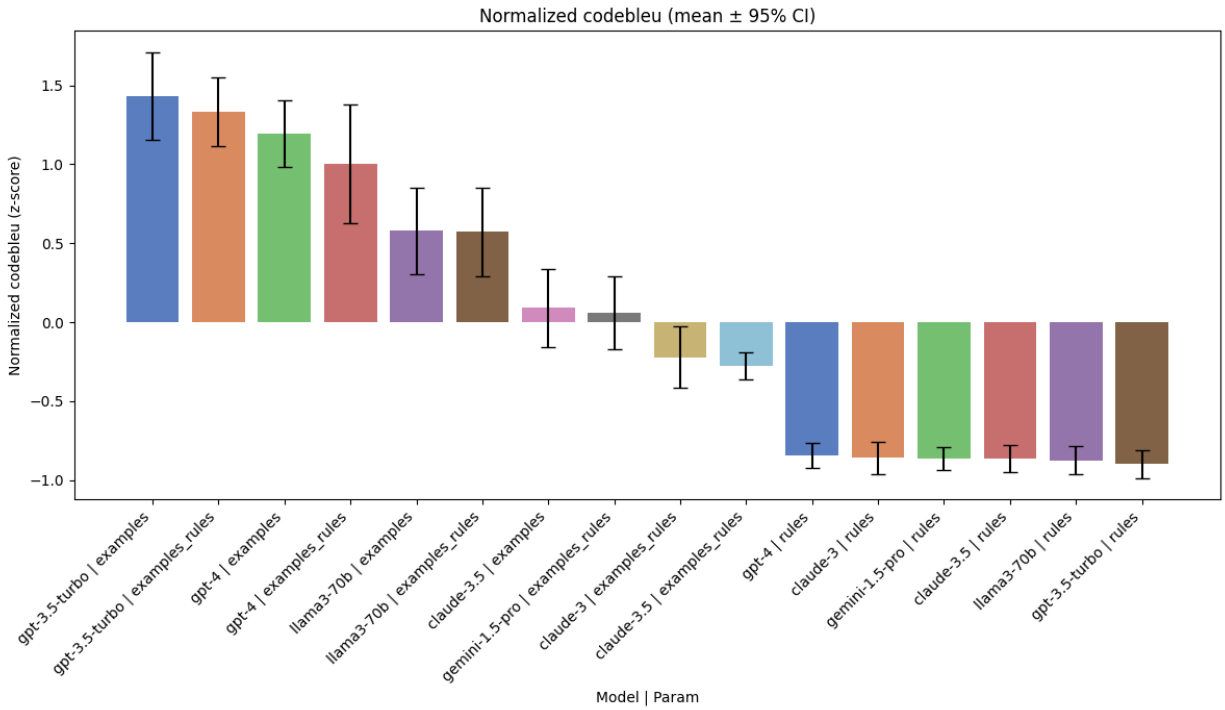


Figure 5.1: CodeBLEU Scores Across Out-of-Box Models and Prompting Strategies

Label	**Z-score**
gpt-3.5-turbo | examples	1.4287
gpt-3.5-turbo | examples_rules	1.3163
gpt-4 | examples	1.2017
gpt-4 | examples_rules	1.0040
llama3-70b | examples	0.5829
llama3-70b | examples_rules	0.5765
claude-3.5 | examples	0.1025
gemini-1.5-pro | examples_rules	0.0679
claude-3 | examples_rules	-0.2041
claude-3.5 | examples_rules	-0.2623
gpt-4 | rules	-0.8509
claude-3 | rules	-0.8656
gemini-1.5-pro | rules	-0.8696
claude-3.5 | rules	-0.8745
llama3-70b | rules	-0.8849
gpt-3.5-turbo | rules	-0.9064

Table 5.1: Normalized codebleu score for each model and prompting strategy

CodeBLEU Scores:

- **GPT-3.5-turbo** consistently outperforms all other models in terms of normalized

CodeBLEU score, across both example-only and hybrid (examples + rules) prompting strategies.

- Providing grammar rules in the prompt sometimes *reduces* performance, particularly when combined with examples—likely due to exceeding context window limits or introducing ambiguity.
- Across all models, **rule-only prompts perform the worst**, indicating that current LLMs struggle to generate semantically coherent Aptly code when given only formal specifications without examples.

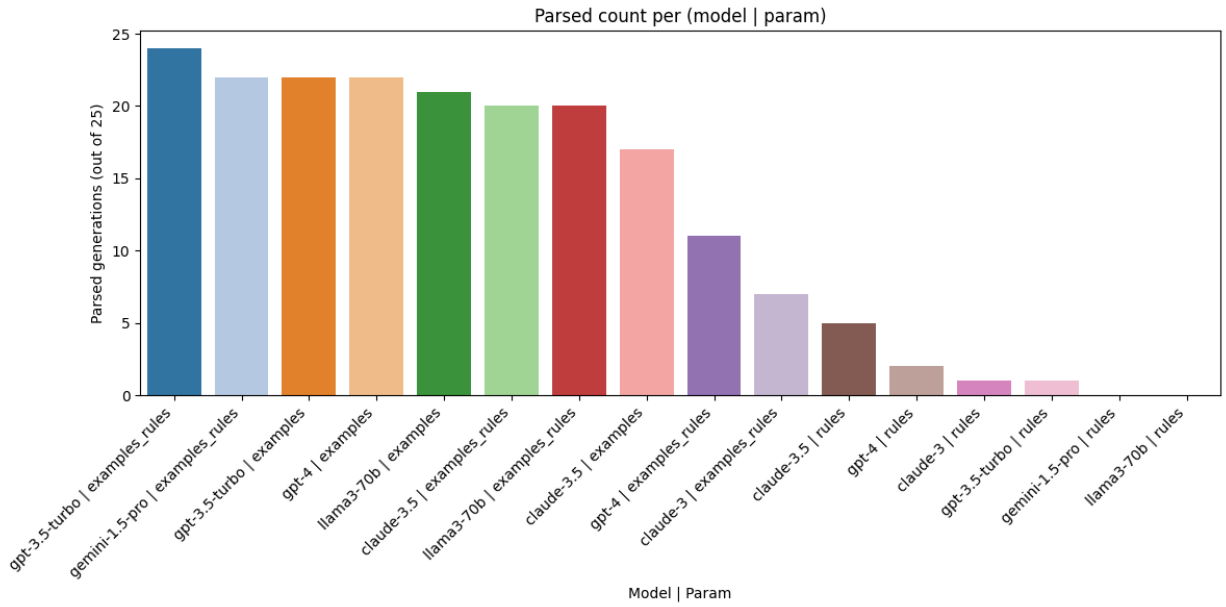


Figure 5.2: Parsing Success Rate Across Out-of-Box Models and Prompting Strategies

Parse Success Rates:

- **GPT-3.5-turbo** also achieves the highest parse success rate, producing syntactically valid Aptly code in 24 out of 25 cases.
- **Gemini 1.5 Pro** ranks second in parsing success, despite being less frequently used in our system—suggesting it may deserve more consideration going forward.
- Unlike with CodeBLEU, **providing grammar rules generally helps parsing**, particularly in the hybrid (examples + rules) setting, reinforcing the idea that rules improve syntactic structure even if they do not aid semantic accuracy.

Conclusion. Taken together, these results highlight GPT-3.5-turbo as the strongest overall performer across both semantic and syntactic metrics with this smaller test set. Interestingly, while grammar rules alone are insufficient for generating meaningful code, they appear to help models produce parseable output when combined with examples. These findings also surface promising alternatives—such as Gemini 1.5 Pro—that may be underutilized in Aptly's current pipeline and merit deeper exploration in future iterations.

Discussion and Future Work

Our initial results indicate that `gpt-3.5 turbo` combined with `both examples and rules` offers the best tradeoff between syntactic validity and semantic correctness. However, parse failures remain non-trivial, motivating future directions including:

- Expanding the example pool for few-shot prompting
- Experimenting with different ways and lengths of prompting.
- Leveraging reinforcement learning to optimize generation toward syntactic validity based on Aptly grammar

5.2 Cross-Validation Evaluation of Finetuned Llama

To assess whether fine-tuning leads to consistent improvements in local token-level code generation, we conducted 5-fold cross-validation using the `ai_labeled` dataset, which contains 2,045 natural language-to-Aptly examples. The dataset was randomly partitioned into five folds, ensuring each fold served once as a test set while the others were used for training. For each fold, we compared the *weighted n-gram match* component of CodeBLEU between a fine-tuned model and its corresponding baseline (out-of-box) model on the same test data.

We chose weighted n-gram match as our primary cross-validation metric to isolate the effects of fine-tuning from potential implementation variability in the new Tree-sitter Aptly parser, avoiding conflation with correctness-sensitive metrics like syntax or dataflow match. In addition, we evaluated parsing success across folds as a proxy for syntactic validity, reflecting whether generated Aptly code could be structurally parsed into valid App Inventor projects.

5.2.1 Fine-Tuning Robustness Across Folds

To evaluate the consistency and stability of our fine-tuning process, we analyzed the performance of the five fine-tuned models—one trained on each fold of the 5-fold cross-validation split—using the *weighted n-gram match* score from CodeBLEU and the number of parseable outputs. These metrics help assess how well each model generalizes to unseen data and whether the fine-tuning procedure exhibits convergence regardless of fold composition.

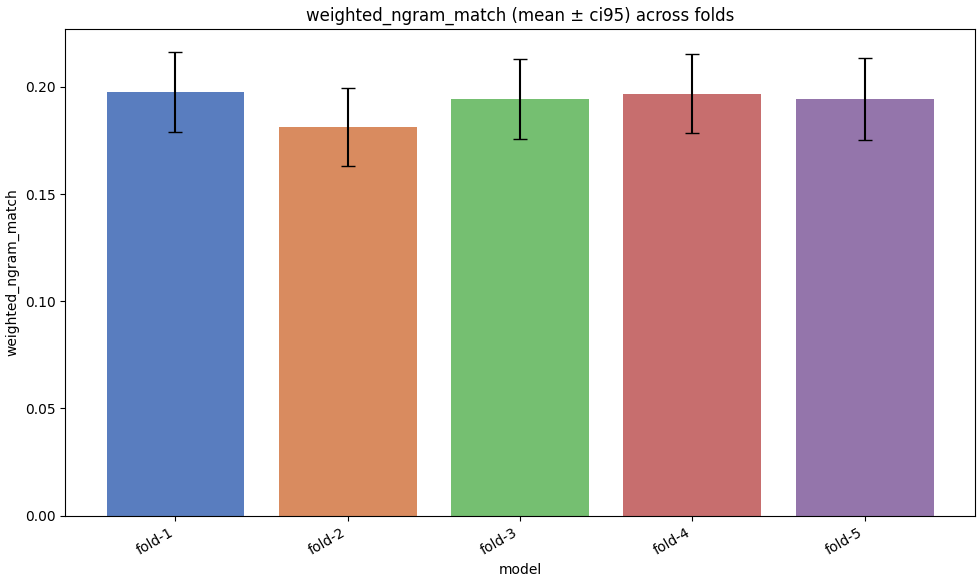


Figure 5.3: Weighted n-gram match (mean ± 95% CI) for each fine-tuned model across the five folds.

Figure 5.3 shows the weighted n-gram match scores for the five fine-tuned models, along with their 95% confidence intervals. The scores across folds are relatively close in magnitude, with all values ranging from 0.181 to 0.198 and overlapping confidence intervals. This suggests that the models converge to similar performance levels across the different folds, reinforcing our claim that the fine-tuning process is robust and not overly sensitive to a specific partition of the data.

Table 5.2: Weighted n-gram match and parseable output count for each fine-tuned model (out of 409 examples).

Fold	**Weighted N-gram Match**	**95% CI**	**Parseable Outputs**
Fold-1	0.198	± 0.0185	67
Fold-2	0.181	± 0.0181	74
Fold-3	0.195	± 0.0187	67
Fold-4	0.197	± 0.0182	61
Fold-5	0.195	± 0.0191	68

Additionally, we measured the number of parseable completions out of 409 total examples per fold. While there is mild variation, the parseability rates are generally consistent, ranging from 61 to 74 completions across folds. These counts are summarized in Table 5.2.

Conclusion: Together, these results indicate that fine-tuning yields reliable training behavior and stable test-time outcomes across data splits.

5.2.2 Fine-tuned vs Baseline Performance Comparison Across Folds

To evaluate whether fine-tuning meaningfully improves model performance compared to out-of-box baselines, we compared each fine-tuned model with its respective baseline across all five folds, using both the weighted n-gram match metric and the number of parseable outputs. These comparisons were paired by test set and evaluated using both per-fold differences and a paired t-test for statistical significance.

Weighted N-gram Match. Figure 5.4 shows the weighted n-gram match scores for all 10 models (5 fine-tuned and 5 baselines), grouped by fold. While fold-level differences exist, the scores are relatively similar across models, and no consistent trend favoring fine-tuned models emerges. As summarized in Table 5.3, the average difference across folds was -0.0007 ± 0.0025, slightly favoring baselines. A paired t-test confirms this difference is not statistically significant ($t = -0.595$, $p = 0.584$).

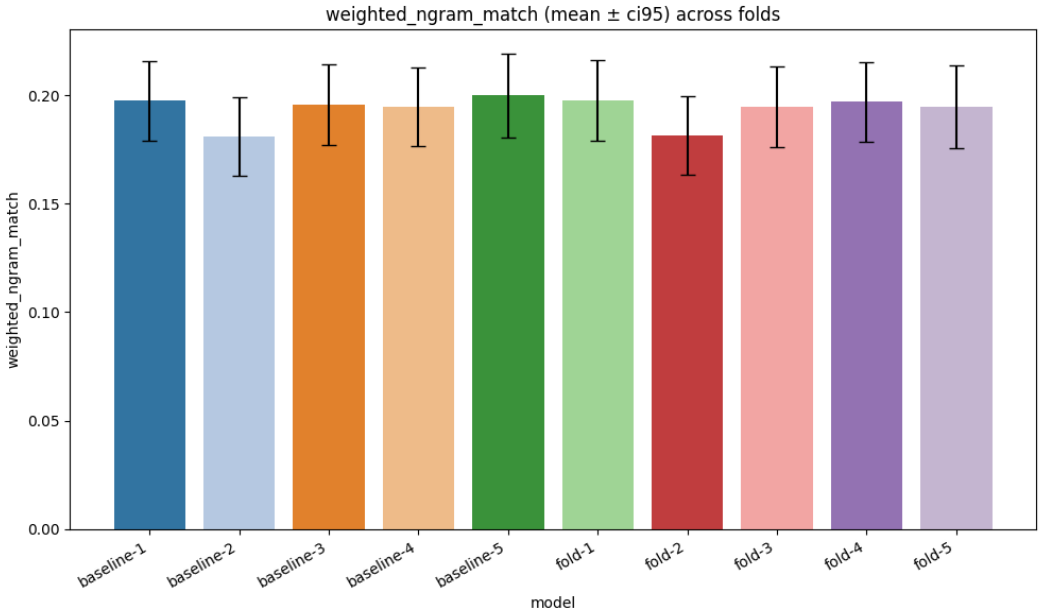


Figure 5.4: Weighted n-gram match (mean ± 95% CI) for fine-tuned and baseline models across all folds.

Table 5.3: Weighted n-gram match comparison per fold.

Fold	**Baseline**	**Fine-tuned**	**(Fine - Base)**
1	0.1974	0.1976	+0.0002
2	0.1811	0.1813	+0.0002
3	0.1955	0.1945	-0.0010
4	0.1947	0.1969	+0.0022
5	0.1998	0.1945	-0.0053
Mean ± Std	-0.0007 ± 0.0025		

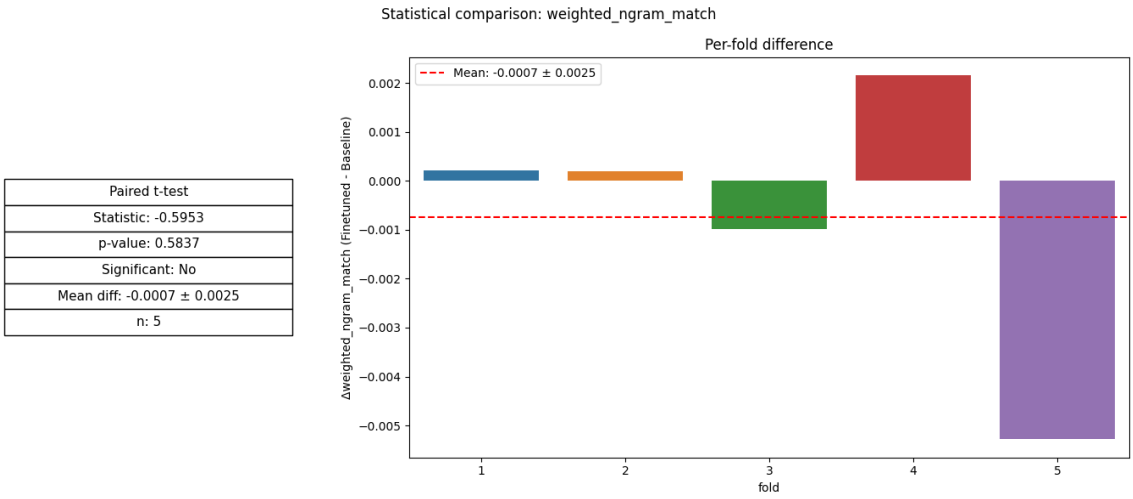


Figure 5.5: Per-fold weighted n-gram match differences (fine-tuned minus baseline), with t-test results.

Parsed Output Counts. Parseability serves as a proxy for syntactic correctness, reflecting whether model-generated Aptly code could be successfully parsed into a valid App Inventor project. Table 5.4 shows the number of parseable completions per model (out of 409 examples). In 4 out of 5 folds, the fine-tuned model had more parseable outputs than its baseline, with the largest gain seen in Fold 2 (+8 completions).

A paired t-test across folds yielded a mean improvement of $+3.6 \pm 4.50$ parsed outputs, which again was not statistically significant ($t = 1.60$, $p = 0.185$). Still, the consistent improvement in 4 of 5 folds suggests a promising trend: fine-tuned models may be more likely to generate well-formed code, even if the effect is not yet conclusive.

Table 5.4: Parseable output comparison per fold.

Fold	**Baseline Parsed**	**Fine-tuned Parsed**	**(Fine - Base)**
1	60	67	+7
2	66	74	+8
3	61	67	+6
4	65	61	-4
5	67	68	+1
Mean ± Std	$+3.6 \pm 4.50$		

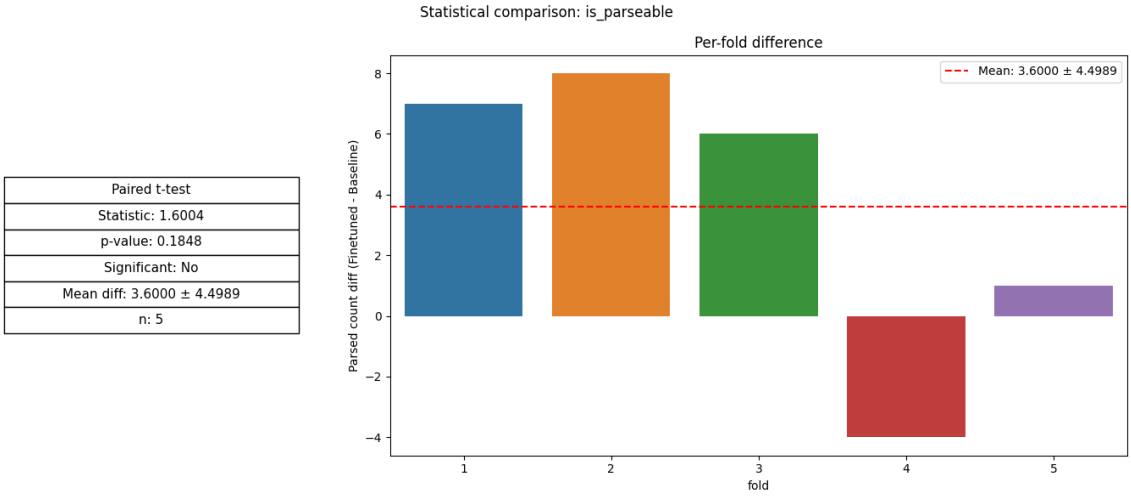


Figure 5.6: Per-fold parseability differences (fine-tuned minus baseline), with t-test results.

Summary. While fine-tuned models do not significantly outperform their baselines in either weighted n-gram match or parseability under statistical testing, we observe early signs of structural improvements, particularly in parsing success. The consistent parseability gains across most folds suggest that fine-tuning may help reinforce well-formed output, even if token-level surface metrics remain flat. These insights point to the value of incorporating structural metrics and downstream task-based evaluation in future iterations.

5.3 Evaluation of Fine-Tuned Models

Beyond out-of-box usage, we explore how fine-tuning affects model performance in both cloud and offline settings. In this section, we evaluate the performance of fine-tuned models relative to commercial API baselines using two key metrics: normalized CodeBLEU scores and parseable output counts. This comparison includes fine-tuned GPT-4.1 and GPT-3.5 variants, as well as baseline models like GPT-4, GPT-3.5-turbo, and locally deployable versions of LLaMA.

Fine-Tuning Cloud-Based Models

We fine-tuned an OpenAI GPT model using the company's fine-tuning API with the curated train dataset, and the *ai_labeled* dataset (references in Section 4.1). The dataset includes hundreds of natural language-app pairs representing real use cases and diverse app types. Figure 5.7 shows the loss curve of an example of the finetuned model, indicating that the model converged after around 300 examples.

Figure 5.7: Train loss and accuracy curve of finetuning job for OpenAI

The final model can be referenced by ID *ft:gpt-4.1-2025-04-14:mit-app-inventor::BU3BkNlZ*.

Fine-Tuning and Deploying Local LLaMA with QLoRA

We also fine-tuned a local version of Meta's LLaMA 3B model using QLoRA, a lightweight method for scalable adaptation of large models. The training data was the same as used for OpenAI fine-tuning.

Model Details: *Base model:* `LLaMA 3B` | *Training method:* `QLoRA` | *Deployment framework:* `MLC LLM`

Weighted N-gram Match / CodeBLEU (Semantic Fidelity): As shown in Figure 5.8, the fine-tuned GPT-4.1 model trained on the `examples` dataset outperforms all baselines, achieving the highest average normalized CodeBLEU score of 0.36 ± 0.12. Both GPT-4 baselines also perform well, with scores of 0.22 and 0.20 respectively, while GPT-3.5 and GPT-3.5-turbo perform more modestly. LLaMA 3B and QLoRA-finetuned LLaMA perform similarily in terms of CodeBLEU score.

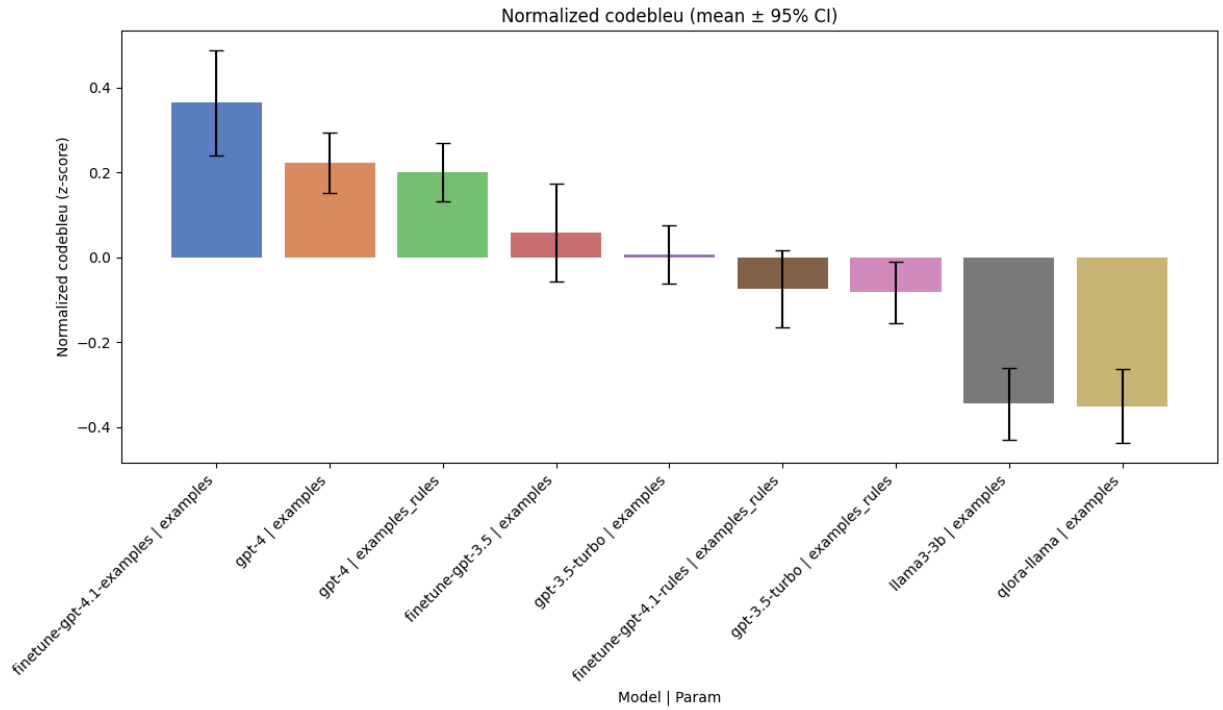


Figure 5.8: Normalized CodeBLEU (z-score) across model and prompting configurations (mean ± 95% CI).

- **Finetuned GPT-4.1 with example prompting achieves the best overall semantic performance**, outperforming all other models with the highest normalized weighted n-gram match score and relatively tight confidence intervals.
- **Fine-tuning led to semantic gains in OpenAI models, but not in LLaMA**, reinforcing that OpenAI's fine-tuning pipeline effectively adapts models to domain-specific tasks, while QLoRA and LLaMA variants failed to show similar improvements in semantic fidelity.
- **Surprisingly, fine-tuned GPT-4.1 with grammar rule prompting underperformed relative to the base model**, suggesting that injecting handcrafted grammar examples may hinder learning in large-context models—an unexpected result that merits further investigation.

Parse Success Rate (Syntactic Validity): Parseability reflects the syntactic validity of generated Aptly programs. Figure 5.9 and Table 5.5 shows the number of parseable completions (out of 409) for each model. Again, the fine-tuned GPT-4.1 `examples` model leads with 331 parsed outputs—over 80% parseability—followed by GPT-4 baselines at 321 and 311. This is consistent with CodeBLEU trends and further validates the robustness of the fine-tuned GPT-4.1 model. However, some models show divergence between semantic fidelity and syntax correctness—for example, out of box models with grammar rules improves parsing but not CodeBLEU.

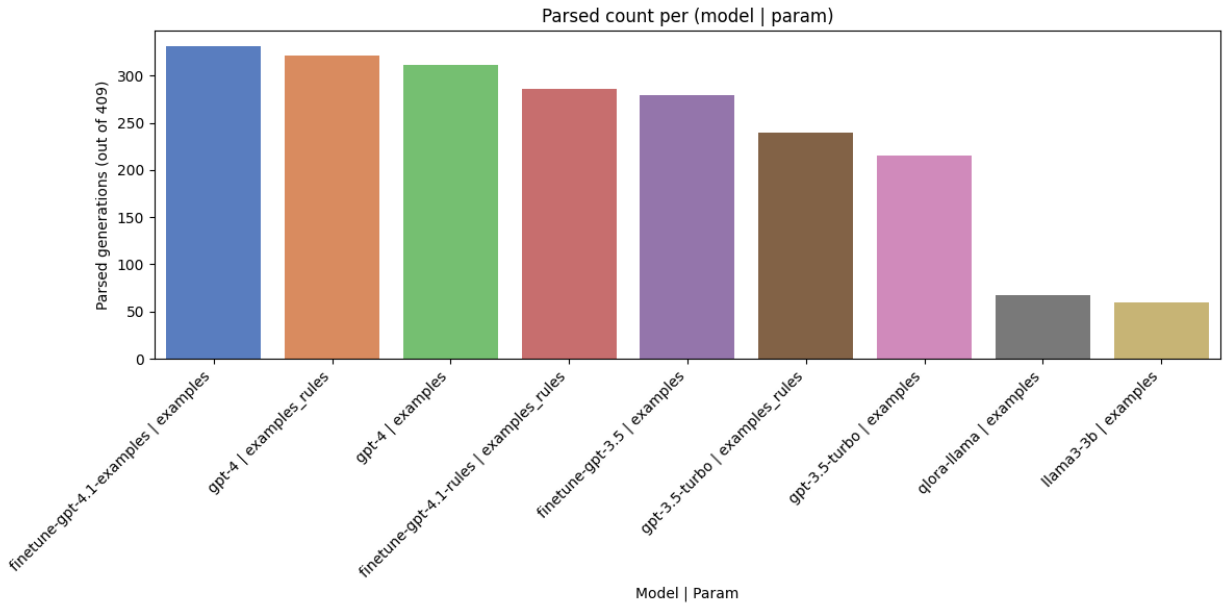


Figure 5.9: Parseable completions (out of 409) per model and prompting setup.

Table 5.5: Parseable output counts (out of 409) per model.

Model | Prompting	**Parseable Completions**
finetune-gpt-4.1-examples | examples	331
gpt-4 | examples_rules	321
gpt-4 | examples	311
finetune-gpt-4.1-rules | examples_rules	286
finetune-gpt-3.5 | examples	279
gpt-3.5-turbo | examples_rules	240
gpt-3.5-turbo | examples	215
qlora-llama | examples	67
llama3-3b | examples	60

- **Finetuned GPT-4.1 also leads in parsing success**, generating valid Aptly code in over 80% of cases, substantially outperforming both the base GPT-4 and all other models.
- **Grammar rules appeared to help parseability in out-of-box GPT models**, but this trend did not carry over to fine-tuned variants, possibly due to conflicting inductive biases introduced during training.
- **QLoRA-finetuned LLaMA improves slightly in parseability over the base model**, indicating that even though semantic accuracy remains low, fine-tuning can help align syntax—though still far behind OpenAI's results.

- This trend reinforces the observation from Section 5.1 that **LLMs often struggle to follow Aptly's grammar rules precisely**, even after domain-specific training—suggesting an opportunity for reinforcement learning or constrained decoding in future work.

Summary These results confirm that fine-tuning can yield significant gains in code generation performance. However, syntactic correctness remains a challenge. Finetuned GPT-4.1 emerges as the most capable model across both dimensions, while the persistent parsing issues in smaller models point to potential benefits from integrating rule-following incentives during training or postprocessing. Grammar rule-based prompting can sometimes improve parseability but may interfere with semantic fidelity during fine-tuning. However, future work may investigate distillation or adapter-based strategies to retain these benefits in smaller, locally deployable models.

5.4 Preliminary User Studies and Additional Experiments

In addition to our core evaluations, we conducted supplementary experiments and early-stage user studies to better understand Aptly's usability, effectiveness, and technical feasibility—both from a system-level and user-centered perspective. These efforts serve to contextualize our fine-tuning and deployment results, and identify paths for improvement in real-world use.

5.4.1 Preliminary User Studies

To assess Aptly's usability and interpretability, we conducted a series of early-stage user studies with MIT undergraduate students (n=5). The full test plan, including task prompts and protocols, is available at https://docs.google.com/document/d/1JBbG-nVy985twt9p0BLko_i6AEPY202VPW1CK0N5GdM/edit?usp=sharing, and study materials, surveys, and recordings are archived at https://drive.google.com/drive/folders/1ilRmofaA4AOYEs-SPIsq7MEJ4_FhHPSo?usp=drive_link.

Each participant was asked to:

1. Describe an app they wanted to build using natural language,
2. Inspect and debug the generated Aptly code blocks,
3. Provide feedback on output quality, prompt design, and their expectations.

Key findings included:

- *Visual mapping clarity:* Users appreciated the transparency between their input and the resulting code blocks.
- *Syntax friction:* Several users were confused by certain tokens in the Aptly-specific syntax and struggled to debug errors without deeper documentation

- *Prompt expectations:* Participants often expected a more "fully fleshed out" app and were surprised when additional prompt specificity was required

These observations motivate a need for improved onboarding tools and scaffolding, and suggest future work on longitudinal usability studies. While qualitative, this feedback confirms Aptly's potential for real-world deployment and highlights directions for interface refinement and future user research.

5.4.2 Technical Supplementary Experiments

Python Baseline Experiments

To calibrate CodeBLEU's behavior, we evaluated it on a set of Python programs and HumanEval tasks. These experiments helped validate the feasibility of adapting CodeBLEU for Aptly-specific evaluation and provide grounding for the metric's interpretation. See Appendix A.2 for charts and further details.

Model Pruning Trials

We experimented with sparsity-aware pruning on our fine-tuned LLaMA model to reduce memory footprint and inference latency. While compression gains were significant, initial results showed notable accuracy degradation at higher sparsity levels. Further study is required to balance performance and efficiency in future deployments. See Appendix A.3 for configuration and preliminary results.

5.4.3 Conclusion

The experimental results presented here confirm both the promise and the limitations of current approaches to mobile-first, offline AI code generation. Our comparative analysis across models, prompting strategies, and tuning setups lays a strong empirical foundation for future optimizations, including RL-based training, prompt engineering, and on-device model refinement.

Overall, the evaluation results highlight that fine-tuning, particularly when paired with well-crafted prompting strategies, can meaningfully enhance both the semantic fidelity and syntactic validity of Aptly-generated code. Fine-tuned GPT-4.1 models consistently outperformed all other baselines across both metrics: achieving the highest normalized CodeBLEU score (0.36 ± 0.12) and generating syntactically valid output in 331 out of 409 cases (81% parse success), outperforming the base GPT-4 by over 5%. While fine-tuned GPT-3.5 showed marginal gains, QLoRA-finetuned LLaMA improved in parseability from 60 to 67 completions—an 11.7% relative gain—despite no improvement in CodeBLEU. Cross-validation of the fine-tuned LLaMA models further demonstrated training stability, with consistent weighted n-gram match scores across folds ($\mu = 0.193$, $\sigma \approx 0.02$), though differences from baseline models were not statistically significant (paired $t = -0.595$, $p = 0.584$ for CodeBLEU; $t = 1.60$, $p = 0.185$ for parseability). These findings confirm the effectiveness of OpenAI's fine-tuning pipeline for Aptly, while also pointing to the need for improved training objectives or decoding strategies for smaller models. Future work may explore grammar-aware

loss functions, distillation from GPT-4.1, or constrained generation to preserve correctness in low-resource, offline-friendly deployments.

Chapter 6

Discussion & Future Work

6.1 Discussion

This thesis explored how to enable offline, natural language–to–app generation on mobile devices using Aptly and MIT App Inventor. We built and evaluated a system that runs a quantized large language model directly on-device, introduced a custom evaluation framework for Aptly's grammar, and compared a range of models and prompting methods to understand what works best.

Out-of-the-box models like GPT-3.5-turbo and GPT-4 performed strongly, especially when given prompting examples, but fine-tuned GPT-4.1 outperformed all other models across both semantic and syntactic metrics—achieving a normalized CodeBLEU score of 0.36 ± 0.12 and over 81% parseable completions. While fine-tuning smaller models like LLaMA with QLoRA led to modest gains in parseability (an 11.7% relative improvement), they continued to struggle with semantic fidelity. These results suggest that fine-tuning open source models has a highly promising direction for future offline code generation. However, high semantic similarity does not guarantee syntactic correctness—many models still failed to produce fully parseable Aptly code, even after training. Bridging this gap will require new approaches, such as grammar-aware loss functions, reinforcement learning, or constrained decoding techniques to better align generation with both structure and function.

On the engineering side, we showed that it's now feasible to deploy a working model fully offline. While the Aptly server still runs outside the phone, the rest of the flow demonstrates that local generation is practical with current hardware and quantization tools. Our evaluation framework, built with a custom Tree-sitter parser and grammar-aware CodeBLEU, gives us a way to track progress and compare model outputs in a structured, reliable way.

This work provides the technical groundwork for Aptly to be used offline, and raises new questions about how to build models that are not just fluent, but structurally precise. We return to the broader goals and implications of this work in the conclusion.

6.2 Future Work

While this thesis establishes a foundation for mobile-first, offline code generation using Aptly, there are several promising directions for future research and development across engineering,

modeling, and human-centered evaluation.

6.2.1 Engineering Extensions

To fully realize an end-to-end mobile deployment pipeline, there are still engineering challenges to address. Currently, the Aptly server runs externally and must be queried for each code generation request. A natural next step is to remove this dependency by either embedding a lightweight Python interpreter directly into the mobile runtime or translating the entire Aptly parser and generator into JavaScript. This would allow the complete flow—from prompt to App Inventor code—to run natively on-device. For the purpose of the initial generation, we also did not require logic from the App Inventot Real Time Collaboration server, but to enable editing apps after generation, we would need to embed this server directly on the mobile app itself as well.

Additionally, the current system has been developed using a relatively limited dataset. As we expand the number and diversity of natural language–to–Aptly code examples, we can improve model robustness and enable more flexible training and evaluation strategies.

6.2.2 Model Improvement and Optimization

Improving on-device model performance remains an open and critical area of work. As we grow the training dataset, we can explore more comprehensive fine-tuning workflows, including full parameter tuning or adapter-based approaches tailored to mobile constraints.

Another key area for exploration is model compression. While initial experiments with QLoRA and quantization enabled local deployment, we aim to take a deeper dive into model pruning. A systematic study of pruning techniques could help us understand the trade-offs between compression, accuracy, and parsing success—paving the way for a more efficient, deployment-ready model.

Finally, an exciting direction involves using **reinforcement learning (RL)** to improve syntactic correctness. By defining a grammar-aware reward function—e.g., based on parsing success or AST structure—we can train the model to better internalize Aptly's formal rules. This could significantly boost the generation of syntactically valid programs, especially in rule-heavy use cases.

6.2.3 Studying User Impact and Accessibility

To assess the real-world utility and accessibility of this tool, further user studies are essential. Future work includes testing how novice programmers use Aptly on-device to create functional mobile apps, exploring how prompt phrasing affects success, and gathering feedback on interpretability and trust.

In addition, we are interested in evaluating Aptly's utility in low-resource settings. Field testing in environments with limited or no internet access can help us better understand what users are able to build when operating purely offline—ultimately guiding the design of tools that truly empower learners everywhere.

6.3 Conclusion

This thesis demonstrates that it is possible to generate App Inventor apps from natural language entirely offline using a fine-tuned language model deployed on a mobile device. We developed and evaluated a working pipeline that brings together model compression, custom prompting strategies, and a tailored evaluation framework designed specifically for Aptly. Along the way, we benchmarked model performance across prompting setups, assessed the impact of fine-tuning, and identified clear gaps—particularly in syntactic validity—that future work can address.

More broadly, this work pushes toward a more accessible model of computing. In many parts of the world, reliable internet and powerful desktop machines are not a given. Tools like Aptly, when made fully offline and mobile-compatible, can offer a new entry point: one where learners can go from an idea to a working app with just a phone and a prompt.

The system presented here is only a first step, but it shows that this kind of access is within reach. Continued progress—on both the modeling and deployment side—can help expand who gets to build with code, and how. That is the larger vision: to make computing creation possible for anyone, anywhere, regardless of connectivity or background.

Appendix A

Resources and Additional Experiments

A.1 Resources and Reproducibility

All source code, models, evaluation scripts, and study materials are publicly available to ensure transparency and reproducibility of this work. Below is a list of relevant resources:

Code and Evaluation Framework

- **GitHub Repository:** All source code, evaluation scripts, and fine-tuning pipelines are available at https://github.com/mit-cml/eval-codegen-aptly

Demonstration Videos

- **Aptly-on-the-Phone Demo:** Live demo of Aptly generating functional apps using a local LLaMA model on mobile https://youtube.com/shorts/lve7uS41sKk
- **Legacy App Inventor Mobile Server Demo:** Prior demo showing App Inventor served fully from a phone https://drive.google.com/file/d/1hfnLz0NBkJHUJrwAbloYpWEYyFXseYMc view?usp=sharing

Presentation Materials

- **Research Presentation Slides (Current Work):** *Aptly on the Phone: Enabling AI-Assisted App Creation Without the Internet* https://www.canva.com/design/DAGlzbEdipU/0h17i6mUm9Xf13pA8Un71Q
- **Architecture Slides (Prior Work):** Final presentation from previous year outlining Aptly architecture and mobile deployment feasibility https://www.canva.com/design/DAGE-7klJP4/Ym1mn_UjG6yrUpODiUOgsA

A.2 Python Baseline Experiments

To understand how CodeBLEU performs on traditional code generation benchmarks, we ran evaluations on both the Concode and MBPP Python datasets. These results served as a

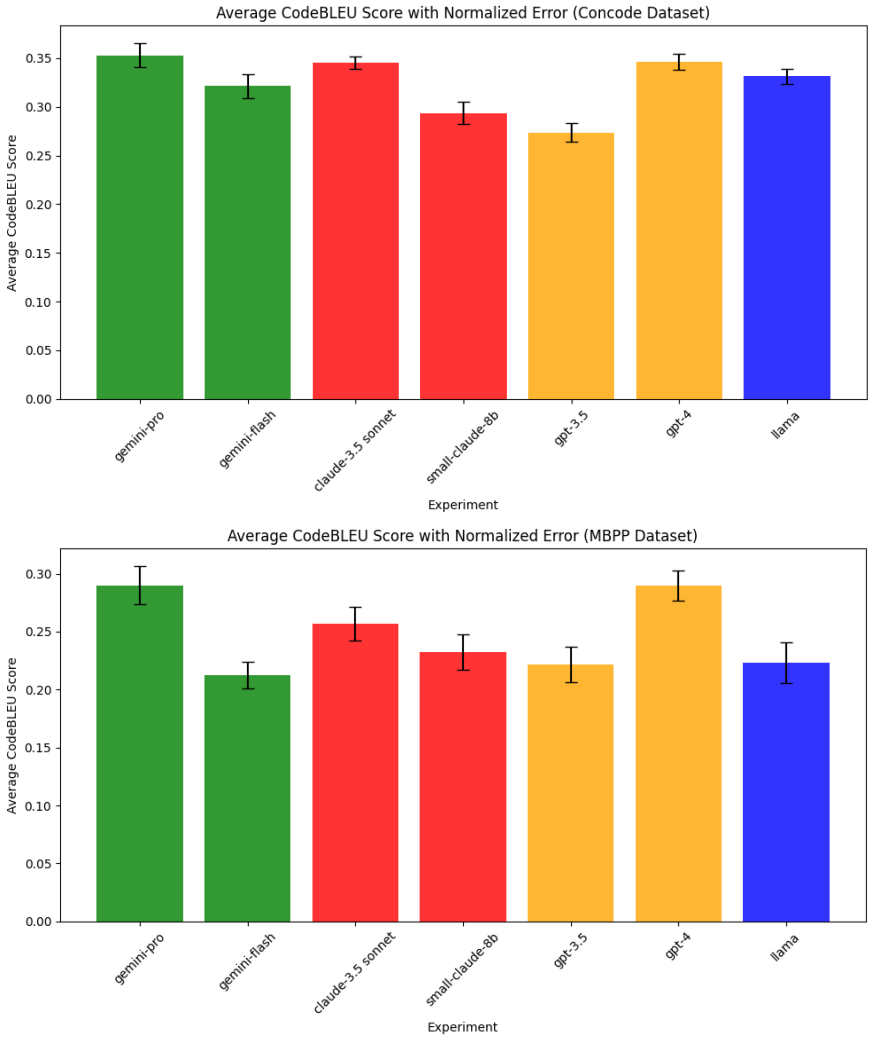


Figure A.1: CodeBLEU evaluation results on the Python Concode dataset (top) and MBPP dataset (bottom).

calibration point for interpreting CodeBLEU's reliability and helped justify our adaptations for Aptly-specific grammar.

These results confirmed that our Aptly-specific modifications to CodeBLEU were necessary and well-justified.

Implementation Details. All experiments were run using our evaluation pipeline in the public repository: https://github.com/mit-cml/eval-codegen-aptly/tree/fall_24. The repository supports generation, evaluation, and visualization for code generation tasks using models such as GPT-3.5, GPT-4, Claude, Gemini, and LLaMA.

- **Generation:** Run via `generate_samples.py`, which supports datasets `human_eval`, `mbpp`, and `concode`. You can specify model, dataset, number of tasks, samples per task, and whether to include CodeBLEU evaluation inline.
- **Evaluation:** Conducted with `evaluate_samples.py`, supporting both HumanEval (pass@k) and CodeBLEU metrics. Outputs are saved in `results/`.
- **Visualization:** Interactive charts comparing model performance (average scores, standard deviation) can be generated using `display_results.ipynb`.
- **Training and Pruning:** A dedicated notebook, `CodeGenerationWithSparsity.ipynb`, allows experimentation with sparsity-aware pruning of open-source LLMs.

Supported Models. The pipeline currently supports GPT-3.5, GPT-4, Gemini Pro, Gemini Flash, Claude 3.5 (Sonnet), Claude Instant (8B), and LLaMA 3B/8B models. Adding a new model involves editing the `generate_one_completion` function in `generate_sample_helpers.py`.

A.3 Model Pruning Trials

We applied sparsity-aware pruning methods to our fine-tuned LLaMA model in order to reduce inference latency and memory requirements. While compression gains were significant, test performance dropped sharply beyond 30% sparsity. Further work is needed to co-train or fine-tune under pruning constraints to retain task-specific performance.

Implementation Details. Experiments were conducted using the `CodeGenerationWithSparsity.ipynb` notebook in the `fall_24` branch of our repository: https://github.com/mit-cml/eval-codegen-aptly/tree/fall_24.

- **Models:** We used Unsloth's 4-bit quantized models for efficient training and inference, including LLaMA 3.1/3.2, Mistral, Phi, and Gemma variants.
- **Training:** Fine-tuning was done using `SFTTrainer`.
- **Pruning:** Structured sparsity was applied using `prune_model(model, sparsity)`.
- **Evaluation:** Supported metrics include BLEU and CodeBLEU via `evaluate_bleu` and `evaluate_codebleu`.

- **Inference:** Evaluation on test samples was performed using `run_inference_on_sample`.

Preliminary Results. At 10% sparsity:

- CodeBLEU score: **0.1002**
- Memory allocated: **4669.5 MB**
- Memory reserved: **5213.5 MB**

These early results highlight the trade-off between sparsity and code generation quality. Future work will include more granular pruning schedules and co-training strategies to improve performance under compression.

Appendix B

Configurations

B.1 System Text for Aptly Rule-Based Prompting

```
Aptly is a programming language used to describe programs for MIT App Inventor.  Aptly can be
     described by the following grammar:

START of GRAMMAR

program = screen_block code_block*
screen_block = component_decl* '\n'
component_decl = identifier '=' identifier '(' component_args? ')'
component_args = parent_component | designer_properties | parent_component  ','
    designer_propreties
parent_component = identifier
designer_properties = designer_property ( ',' designer_property )*
designer_property = identifier '=' designer_value
code_block = '\n'+ comment? decl '\n'+
decl = global_decl | procedure_decl | event_decl | generic_event_decl
global_decl = 'initialize' identifier '=' value
procedure_decl = 'to' identifier '(' arglist? '):\n' scoped_statements
scoped_statements = commented_statement+ ( return | do_return_expr)?
return = comment? 'return:\n' scoped_expression
scoped_expression = comment? expression
arglist = identifier ( ',' identifier )*
event_decl = 'when' identifier '.' identifier '(' arglist? '):\n' scoped_statements
generic_event_decl = 'when' 'any' identifier '.' identifier '(' arglist '):\n'
    scoped_statements
commented_statement = comment? statement
statement = method_call | local_var_decl | setter | cond_stmt | loop_stmt
method_call = 'call' identifer ( '.' identifier ) '(' argvalues? ')'
argvalues = expression ( ',' expression )*
local_var_decl = 'let' letlist ':' scoped_statements
letlist = local_variable '=' expression ( ',' letlist )?
setter = 'set' location '=' expression
cond_stmt = 'if' expression ':' scoped_statements ('elif' expression ':' scoped_statements)*
    ( 'else:' scoped_statements )?
location = local_variable | global_variable | component_property
```

```
local_variable = identifier
global_variable =  'global' identifier
component_property =  identifier '.' identifier
loop_stmt = for_each_stmt | for_item_stmt | for_key_value_stmt | while_stmt
for_each_stmt = 'for' identifier 'from' expression 'to' expression [ 'by' expression ]? ':'
    scoped_statements
for_item_stmt = 'for' identifier 'in' expression ':' scoped_statements
for_key_value_stmt = 'for' identifier 'with' identifier 'in' expression ':' scoped_statements
while_stmt = 'while' expression ':' scoped_statements
expression = cond_expression
cond_expression = or_expression ( 'if' or_expression 'else' expression )*
or_expression = and_expression ( 'or' and_expression )*
and_expression = add_expression ( 'and' add_expression )*
add_expression = mul_expression ( [+-] mul_expression )*
mul_expression = paren_expression ( '^' paren_expression )*
paren_expression = number | location | method_call | primative_call | '(' expression |
    local_var_decl | do_return_expr ')'
do_return_expr = 'do:\n' scoped_statements+ return
primative_call = identifier '(' argvalues? ')'
comment = '#' comment_text
designer_value = boolean | number | string
value = boolean | number | string | list | dictionary | identifier
list = '[' list_entries ']'
list_entries = value ( ',' value )*
dictionary = '{' dictionary_entries? '}'
dictionary_entries = dictionary_entry ( ',' dictionary_entry )*
dictionary_entry = key ':' value
key = identifier | number | string
boolean = 'True' | 'False'
number = [+-]?[0-9]+('.'[0-9]+)?
string = "([^"]|\\")*"
identifier = [A-Za-z][A-Za-z0-9]*
comment_text = [^\n]*


END OF GRAMMAR

In addition to the grammar above, there are semantic rules related to Aptly, as well as
    information about the framework within which App Inventor (and therefore Aptly) programs
    operate.

Here are some semantic rules about Aptly:
A statement or a return within a scoped_statement production MUST be indented from the
    previous scoped_statement block, and SHOULD be indented by +4 spaces.
Local variables (i.e. variables declared using the 'let' keyword) can only be referenced
    within the scope of their let statement.  That means that those local variable references
     MUST be indented under their declarations. Here is an example declaration and use of a
    local variable:
```
let x = 1
    let y = x + 1
        call print(y)
```
```

```
Global variables (i.e. variables declared using the 'initialize' keyword) MUST be referenced
    by preceding their name with the keyword 'global'.  For example:
```
initialize z = 1
let x = 1
    set global z = x + 1
    call print(global z)
```

Here is some information about Aptly, related to its connection to the MIT App Inventor
    environment:
An Aptly program consists of components, which are basically objects which encapsulate visual
     UI appearance and behavior and other non-visual behaviors.
Components have three major types of code associated with them:
Properties, which have getters and setters
Methods, which can be called to provide certain component-related behavior.
Event handlers, which are executed when certain events occur
UI components can be nested.  The roots of the UI component trees are the "Screen" components
    .  Other UI components are descendents of Screen components.  The initial Screen
    component is named "Screen1".  Other Screen components can have programmer-defined names.
      The other main UI components that can contain child components are:
    HorizontalArrangement and VerticalArrangement.  They are layout components that cause
    their children to be laid out horizontally and vertically.  Note also that the Screen
    component lays its children out vertically.
The first argument of a component declaration is its parent component.  Subsequent arguments
    are property initializers.
Aptly has a number of built in types: Text, Math (i.e. numbers), Logic (i.e. booleans), Lists
    , Dictionaries and Colors.
For information about the full set of components that Aptly can use, use your knowledge of
    MIT App Inventor.
For information about the full set of functions and operations available to use with the
    built in Aptly types, use your knowledge of MIT App Inventor.

Here are some things to note about the grammar:
All components must be declared before any of the rest of the code


Make sure that any generated program follows the rules of the grammar, semantics and
    environment described above. Also make sure to use your knowledge of MIT App Inventor
    when choosing components and functions.

Verify that all components, methods, events and properties are actually valid in MIT App
    Inventor.  If you are unsure, consult the MIT App Inventor documentation.

Also make sure that for any generated program that you create, you prepend a line that
    contains the text "START" and
append a line that contains the text: "STOP"
```

B.2 MLC Chat Config

```
    {
```

```
  "version": "0.1.0",
  "model_type": "llama",
  "quantization": "q4f16_1",
  "model_config": {
    "hidden_size": 3072,
    "intermediate_size": 8192,
    "num_attention_heads": 24,
    "num_hidden_layers": 28,
    "rms_norm_eps": 1e-05,
    "vocab_size": 128256,
    "tie_word_embeddings": true,
    "position_embedding_base": 500000.0,
    "rope_scaling": {
      "factor": 32.0,
      "high_freq_factor": 4.0,
      "low_freq_factor": 1.0,
      "original_max_position_embeddings": 8192,
      "rope_type": "llama3"
    },
    "context_window_size": 3072,
    "prefill_chunk_size": 128,
    "num_key_value_heads": 8,
    "head_dim": 128,
    "tensor_parallel_shards": 1,
    "pipeline_parallel_stages": 1,
    "max_batch_size": 128,
    "disaggregation": false
  },
  "vocab_size": 128256,
  "context_window_size": 3072,
  "sliding_window_size": -1,
  "prefill_chunk_size": 128,
  "attention_sink_size": -1,
  "tensor_parallel_shards": 1,
  "pipeline_parallel_stages": 1,
  "temperature": 0.6,
  "presence_penalty": 0.0,
  "frequency_penalty": 0.0,
  "repetition_penalty": 1.0,
  "top_p": 0.9,
  "tokenizer_files": [
    "tokenizer.json",
    "tokenizer_config.json"
  ],
  "tokenizer_info": {
    "token_postproc_method": "byte_level",
    "prepend_space_in_encode": false,
    "strip_space_in_decode": false
  },
  "conv_template": {
    "name": "llama-3",
    "system_template": "<|start_header_id|>system<|end_header_id|>\n\n{system_message}<|
    eot_id|>",
    "system_message": "You are a helpful, respectful and honest assistant.",
```

```
    "system_prefix_token_ids": [
      128000
    ],
    "add_role_after_system_message": true,
    "roles": {
      "user": "<|start_header_id|>user",
      "assistant": "<|start_header_id|>assistant"
    },
    "role_templates": {
      "user": "{user_message}",
      "assistant": "{assistant_message}",
      "tool": "{tool_message}"
    },
    "messages": [],
    "seps": [
      "<|eot_id|>"
    ],
    "role_content_sep": "<|end_header_id|>\n\n",
    "role_empty_sep": "<|end_header_id|>\n\n",
    "stop_str": [
      "<|end_of_text|>",
      "<|eot_id|>"
    ],
    "stop_token_ids": [
      128001,
      128009
    ],
    "function_string": "",
    "use_function_calling": false
  },
  "pad_token_id": 0,
  "bos_token_id": 128000,
  "eos_token_id": [
    128001,
    128008,
    128009
  ]
}
```


Listing B.1: Full MLC Chat Cofig settings file for finetuned Llama-3B for Aptly

References

[1] App Inventor Foundation. *Annual Impact Report 2023*. https://www.appinventorfoundation.org/news/annual-impact-report-2023. Accessed: 2025-04-19. 2023.

[2] E. Patton, A. Granquist, M. Kelleher, H. Abelson, and the MIT App Inventor Team. *Speak Your Mind: Introducing Aptly, the Software Platform that Turns Ideas into Working Apps*. https://appinventor.mit.edu/blogs/hal/2022/03/21/Aptly. Accessed: 2025-04-19. 2022.

[3] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980.

[4] M. Resnick et al. "Scratch: programming for all". In: *Commun. ACM* 52.11 (Nov. 2009), pp. 60–67. ISSN: 0001-0782. DOI: 10.1145/1592761.1592779. URL: https://doi.org/10.1145/1592761.1592779.

[5] S. C. Pokress and J. J. D. Veiga. *MIT App Inventor: Enabling Personal Mobile Computing*. 2013. arXiv: 1310.2830 [cs.CY]. URL: https://arxiv.org/abs/1310.2830.

[6] M. Tissenbaum, J. Sheldon, and H. Abelson. "From computational thinking to computational action". In: *Commun. ACM* 62.3 (Feb. 2019), pp. 34–36. ISSN: 0001-0782. DOI: 10.1145/3265747. URL: https://doi.org/10.1145/3265747.

[7] Y. Xu and M. Warschauer. "Exploring young children's engagement in joint reading with a conversational agent". In: *Proceedings of the Interaction Design and Children Conference*. IDC '20. London, United Kingdom: Association for Computing Machinery, 2020, pp. 216–228. ISBN: 9781450379816. DOI: 10.1145/3392063.3394417. URL: https://doi.org/10.1145/3392063.3394417.

[8] E. W. Patton, D. Y. J. Kim, A. Granquist, R. Liu, A. Scott, J. Zamanova, and H. Abelson. *Aptly: Making Mobile Apps from Natural Language*. 2024. arXiv: 2405.00229 [cs.HC]. URL: https://arxiv.org/abs/2405.00229.

[9] D. Y. Kim, P. Ravi, R. Williams, and D. Yoo. "App Planner: Utilizing Generative AI in K-12 Mobile App Development Education". In: *Proceedings of the 23rd Annual ACM Interaction Design and Children Conference*. IDC '24. Delft, Netherlands: Association for Computing Machinery, 2024, pp. 770–775. ISBN: 9798400704420. DOI: 10.1145/3628516.3659392. URL: https://doi.org/10.1145/3628516.3659392.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[11] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. 2024. arXiv: 2211.10438 [cs.CL]. URL: https://arxiv.org/abs/2211.10438.

[12] T. B. Brown et al. "Language Models are Few-Shot Learners". In: (2020). arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[13] A. Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: (2022). arXiv: 2204.02311 [cs.CL]. URL: https://arxiv.org/abs/2204.02311.

[14] H. Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: (2023). arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[15] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, and A. M. Dai. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.

[16] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: (2020). arXiv: 1910.01108 [cs.CL]. URL: https://arxiv.org/abs/1910.01108.

[17] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers". In: (2023). arXiv: 2210.17323 [cs.LG]. URL: https://arxiv.org/abs/2210.17323.

[18] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, and S. Han. "Tiny Machine Learning: Progress and Futures [Feature]". In: *IEEE Circuits and Systems Magazine* 23.3 (2023), pp. 8–34. ISSN: 1558-0830. DOI: 10.1109/mcas.2023.3302182. URL: http://dx.doi.org/10.1109/MCAS.2023.3302182.

[19] J. Frankle and M. Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: (2019). arXiv: 1803.03635 [cs.LG]. URL: https://arxiv.org/abs/1803.03635.

[20] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. *Pruning and Quantization for Deep Neural Network Acceleration: A Survey*. 2021. arXiv: 2101.09671 [cs.CV]. URL: https://arxiv.org/abs/2101.09671.

[21] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou. "MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices". In: (2020). arXiv: 2004.02984 [cs.CL]. URL: https://arxiv.org/abs/2004.02984.

[22] P. Zhang, G. Zeng, T. Wang, and W. Lu. *TinyLlama: An Open-Source Small Language Model*. 2024. arXiv: 2401.02385 [cs.CL]. URL: https://arxiv.org/abs/2401.02385.

[23] MLC team. *MLC-LLM*. 2023-2025. URL: https://github.com/mlc-ai/mlc-llm.

[24] G. Hinton, O. Vinyals, and J. Dean. "Distilling the Knowledge in a Neural Network". In: (2015). arXiv: 1503.02531 [stat.ML]. URL: https://arxiv.org/abs/1503.02531.

[25] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith. "Don't Stop Pretraining: Adapt Language Models to Domains and Tasks". In: (2020). arXiv: 2004.10964 [cs.CL]. URL: https://arxiv.org/abs/2004.10964.

[26] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. "LoRA: Low-Rank Adaptation of Large Language Models". In: (2021). arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[27] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *arXiv preprint arXiv:2305.14314* (2023). URL: https://arxiv.org/abs/2305.14314.

[28] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. "BLEU: a method for automatic evaluation of machine translation". In: ACL '02 (2002), pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://doi.org/10.3115/1073083.1073135.

[29] M. Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[30] 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". In: *CoRR* abs/2009.10297 (2020). arXiv: 2009.10297. URL: https://arxiv.org/abs/2009.10297.

[31] J. Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732.

[32] D. Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: (2021). arXiv: 2105.09938 [cs.SE]. URL: https://arxiv.org/abs/2105.09938.

[33] Y. Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097. ISSN: 1095-9203. DOI: 10.1126/science.abq1158. URL: http://dx.doi.org/10.1126/science.abq1158.

[34] OpenAI et al. "GPT-4 Technical Report". In: (2024). arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[35] K. Zhang and D. Shasha. "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems". In: *SIAM Journal on Computing* 18.6 (1989), pp. 1245–1262. DOI: 10.1137/0218082. eprint: https://doi.org/10.1137/0218082. URL: https://doi.org/10.1137/0218082.

[36] X. Deng and E. W. Patton. "Enabling Multi-User Computational Thinking with Collaborative Blocks Programming in MIT App Inventor". In: *Proceedings of the 1st International Conference on Computational Thinking in Education*. July 13–15. Hong Kong, China, July 2017. URL: https://appinventor.mit.edu/papers/CTE_2017_paper_84.pdf.

[37] B. Mayo. *Apps can request access to more RAM with iOS 15 entitlement, exceeding normal system memory limits*. Accessed: 2025-04-18. 2021. URL: https://9to5mac.com/2021/06/25/apps-can-request-access-to-more-ram-with-ios-15-entitlement-exceeding-normal-system-memory-limits/.

[38] MLC team. *MLC-LLM*. 2023. URL: https://github.com/mlc-ai/mlc-llm.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 1536–1547.

[40] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. "Mapping Language to Code in Programmatic Context". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1643–1652. DOI: 10.18653/v1/D18-1192. URL: https://aclanthology.org/D18-1192/.

[41] J. Austin et al. "Program Synthesis with Large Language Models". In: *CoRR* abs/2108.07732 (2021). arXiv: 2108.07732. URL: https://arxiv.org/abs/2108.07732.

[42] M. Brunsfeld. *Tree-sitter: Incremental Parsing System*. https://tree-sitter.github.io/tree-sitter/creating-parsers. Accessed: 2025-04-26.

[43] 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. arXiv: 2009.10297 [cs.SE]. URL: https://arxiv.org/abs/2009.10297.