NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Mobile AI-Generated Content (AIGC) Services

Zaw K [abbreviated for upload]

Student ID: [redacted]

Project Supervisor: [redacted]

Examiner: [redacted]

College of Computing and Data Science

Academic Year 2024/2025

NANGYANG TECHNOLOGICAL UNIVERSITY

Mobile AI-Generated Content (AIGC) Services

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering

in Computer Science of the Nanyang Technological University

by

Zaw K

College of Computing and Data Science

Academic Year 2024/2025

Abstract

This project undertakes the task of deploying a Stable Diffusion, a text-to-image generation model, on edge devices such as Android phones to enable on-device image generation without requiring active internet connection. Furthermore, this project also explores the techniques of utilizing large language models to improve the visual accuracy and aesthetic quality of generated images. To achieve this, an Android application was developed, integrating a diffusion image generation pipeline in Android native code. As an optional component of the image generation pipeline, this project also utilizes edge networks, such as a local area network (LAN), to run a lightweight large language model (LLM), enhancing prompt processing without requiring cloud-based resources, thus reducing latency and ensuring greater data privacy.

Due to the unavailability of a physical Android device during development, the project was solely developed on Android emulator. This precluded the exploration of mobile GPU-based inferencing, as Android AI inferencing libraries do not support GPU acceleration on Android Emulators. Consequently, all inference tasks were restricted to the CPU.

This report aims to elaborate on the design and development of Stable Diffusion image generation pipeline in Android native code, including model conversion for compatibility with Android and edge devices. It also covers the integration of a large language model into the pipeline for enhancing image generation. Additionally, the report discusses the limitations encountered due to the early stage of the Edge AI development ecosystem and proposes potential solutions for future work to optimize deployment efficiency and performance.

Acknowledgement

I would like to express my sincere gratitude to Professor [redacted] for his invaluable guidance and support throughout the course of this project. His expertise and mentorship were pivotal in helping me expand my knowledge in the field of edge AI and successful completion of this project.

I would also like to extend my heartfelt thanks to Dr. [redacted] and Dr. [redacted] for their invaluable help, especially during the early stages of the project when I faced roadblocks. Their support was instrumental in overcoming these challenges.

Lastly, I would like to acknowledge the contributions of the developers and communities behind PyTorch, ONNX, Hugging Face, and other tools that were essential to the success of this project. Their open-source contributions provided the foundation and resources needed to implement the image generation pipeline.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Generative AI has revolutionized the field of artificial intelligence and machine learning by enabling the creation of complex content from simple user prompts. This technology has unfolded enormous opportunities, extending far beyond the creative arts to diverse sectors such as scientific research, health care, security and education [1] [2]. Among the various models developed, text-to-image generation models stand out as one of the most resource intensive models in terms of training and inferencing. Furthermore, the rapid advancements and increase in demand for more capable and more intelligent models have led to the development of increasingly sophisticated models, which often require even more computational power.

The substantial computational demands of these models, especially for generating high-resolution images, typically necessitate model deployment in centralized data centers equipped with powerful and expensive hardware. However, this centralization leads to a significant bottleneck in resource allocation, often resulting in reduced availability and increased cost. To address these challenges, data centers typically offer image generation services through paid subscriptions, often with very limited credits or imposing restrictions on the number of free images that can be generated within a specific time frame.

Edge computing presents a promising alternative by enabling the deployment of AI models directly onto user devices, such as mobile phones, reducing the reliance on centralized infrastructure. While most modern mobile phones still lack the hardware capacity to handle computationally intensive on-device AI tasks in a short period, significant advancements in mobile processing power over the years have made these devices increasingly capable. As a result, mobile phones are becoming one of the most suitable candidates for on-device AI applications, including text-to-image generation.

1.2 Objectives and Aims

The main objective of this project is to design and develop a generative text-to-image Android application which leverages mobile processing power for local, on-device image generation, without requiring an active internet connection. The application will integrate Stable Diffusion version 1.4, a text-to-image generation model. Among the various and more capable Stable Diffusion models, this version was chosen due to its simpler architecture, model size and ability to generate high-quality images without requiring high-end hardware.

Additionally, this project also aims to enhance the aesthetic quality and accuracy of generated images by utilizing a language model (LLM). This LLM, deployed on a local area network (LAN), will enrich user-provided prompts by adding additional information such as distinct physical features and colours of objects which are mentioned in the user-provided prompt. The mobile device offloads the task of prompt enrichment to the LLM deployed in the LAN when available, improving the accuracy and quality of the generated images

While the project involves communication over LAN, security measures such as encryption and authorization are not implemented, as the primary focus is on implementation and to demonstrate the feasibility of using mobile devices for on-device AI applications rather than the security and privacy aspect of on-device AI applications.

1.3 Project Scope

The primary scope of this project is as follows:

- **Development of Android app**: Design and implement an Android application that allows users to input prompts, specify the seed for image generation, and adjust the number of denoising loops to balance quality and speed, and display the final generated image.
- **Integration of Stable Diffusion v1.4**: Most publicly available models are designed for inference and fine-tuning in Python environment. However, Android does not natively support Python execution for app development. Therefore, models must be converted to a format compatible with Android operating system for on-device inferencing.
- **Stable Diffusion Pipeline implementation**: Stable Diffusion pipeline consists of several interconnected components such as tokenizer, text encoder, UNet, scheduler and variational autoencoder. These components work together to transform text prompts into images through a step-by-step denoising process. To ensure compatibility with Android, this pipeline will be implemented using Android-native code, optimizing it for on-device execution.
- **Development of REST API server:** A lightweight REST API server will be developed for handling prompt enrichment requests from the Android application. This server will host a locally deployed large language model (LLM) on a local area network (LAN) and process user-provided prompts by adding additional context, such as detailed physical characteristics of objects. The enriched prompts will then be sent back to the mobile device for use in the Stable Diffusion image generation process.
- **Conversion of Model Output to Image Format**: After generating the image data from the Stable Diffusion model, this data needs to be transformed into a standard image format (such as PNG or JPEG) for display on the mobile device. This step will involve implementing a solution within the app to convert the model's output into a format that can be easily viewed and used by the user, ensuring the final image is ready for presentation.

1.4 Report Organization

This section provides an overview of chapters in this report, outlining the structure and contents. The report is organized as follows:

- Chapter 1: Introduction – Provides an introduction of the project, objectives, and scope.
- Chapter 2: Literature Review – Discusses the relevant technologies and literatures related to Stable Diffusion and edge computing.
- Chapter 3: Architecture and Technologies Used – Details the design and architecture of mobile application, model integration and overall system architecture.
- Chapter 4: Setup and Implementation – Describes the solutions implemented in this project.
- Chapter 5: Conclusion and Future Work – Concludes the report and provides suggestions for future improvement and research

Chapter 2

Literature Review

In order to generate images locally, this project relies solely on the computational power of edge devices. Although significant progress has been made in improving the processing capacity of edge devices in recent years, thermal constraints and battery consumptions remain the major obstacles to the efficient execution of deep learning models on-device. Furthermore, the deployment of deep learning models on edge devices is hindered by the early-stage edge AI development ecosystem, which lacks mature APIs, standardized interoperability, and comprehensive optimization tools. Existing popular machine learning frameworks such as TensorFlow, PyTorch, CoreML and ONNX offer foundational capabilities but remain fragmented, necessitating vendor-specific implementations which limit portability across devices and operating systems.

To better understand the feasibility and ways of deploying generative deep learning models on edge devices, it is essential to understand the technology and scrutinize the limitations and techniques to address the challenges. The following section examines technology underneath the image generative model, limitations, key constraints of edge devices, related work and the broader challenges associated with deploying generative deep learning models on edge devices.

2.1 Generative AI

Generative AI has become one of the significant research areas within the field of artificial intelligence, mainly driven by the advancements in deep learning techniques, the availability of large datasets and rapid growth of computational power. Nowadays, many versions of generative artificial intelligence exist, including but not limited to text-to-image, text-to-sound, text-to-video and large language models. The applications of generative models are vast and diverse, ranging from art creation and entertainment to science [3][4][5]. These models are increasingly being utilized, and it is expected that they will continue to play an important role in advancing both theoretical understanding and practical applications across various domains.

Generative AI is primarily utilized through prompts. Prompts are natural language sentences provided by the users. The ability of generative AI models to understand and process according to the prompts is one of its key strengths. This is achieved by transforming natural language into text embeddings, which are numerical representations of words and phrases that capture the semantic meaning [6]. After the transformation, the embeddings are used by the models to generate the output in various forms.

One of the most notable and revolutionary applications of generative AI is text-to-image generation, where models create images from textual descriptions. Over the decade, significant progress in the field of text-to-image generative AI models have yielded substantial improvements in realism and level of detail achievable in image generation. The following figure illustrates the evolution of image quality and detail overtime in image generative models.

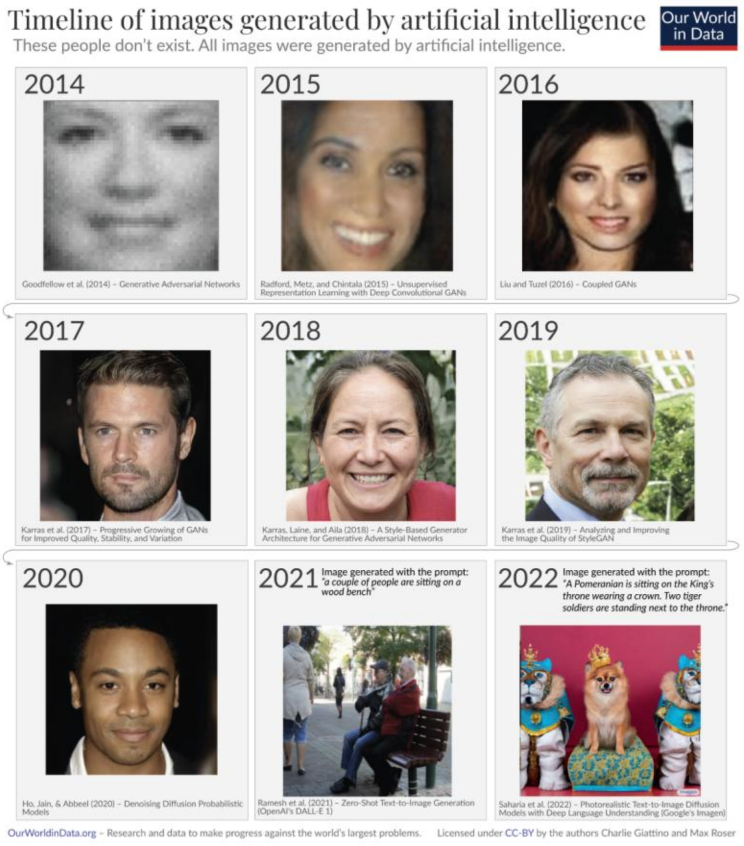

Timeline of images generated by artificial intelligence
These people don't exist. All images were generated by artificial intelligence.
Our World in Data
2014
Goodfellow et al. (2014) – Generative Adversarial Networks
2015
Radford, Metz, and Chintala (2015) – Unsupervised Representation Learning with Deep Convolutional GANs
2016
Liu and Tuzel (2016) – Coupled GANs
2017
Karras et al. (2017) – Progressive Growing of GANs for Improved Quality, Stability, and Variation
2018
Karras, Laine, and Aila (2018) – A Style-Based Generator Architecture for Generative Adversarial Networks
2019
Karras et al. (2019) – Analyzing and Improving the Image Quality of StyleGAN
2020
Ho, Jain, & Abbeel (2020) – Denoising Diffusion Probabilistic Models
2021
Image generated with the prompt: "a couple of people are sitting on a wood bench"
Ramesh et al. (2021) – Zero-Shot Text-to-Image Generation (OpenAI's DALL-E 1)
2022
Image generated with the prompt: "A Pomeranian is sitting on the King's throne wearing a crown. Two tiger soldiers are standing next to the throne."
Saharia et al. (2022) – Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding (Google's Imagen)
OurWorldinData.org – Research and data to make progress against the world's largest problems.
Licensed under CC-BY by the authors Charlie Giattino and Max Roser

Figure 1: Evolution of Image Generation Quality. Source: ourworldindata

2.2 Text-to-image Generative Models

The text-to-image generative models have made significant advancement in recent years, and it has unlocked the ability to create sophisticated, realistic and visually appealing images through a single prompt. Advancement in this field is made possible by several paradigm shifts in architectural approaches and optimization techniques.

2.2.1 Generative Adversarial Networks

Generative adversarial networks (GAN) was one of the prominent frameworks for image generative AI in the early days. The earlier versions of GAN did not make use of prompts for guiding image generation process. GAN consists of two neural networks, namely, generator and discriminator. During the training process, the former is responsible for creating a synthetic image from a random noise, while the latter is responsible for distinguishing between real and synthetic images. The two networks are trained in adversarial manner, which led to the generator producing realistic images [7].

Over time, variations of GAN emerged such as Conditional GAN (cGAN) and StyleGAN which allowed the models to be conditioned through external inputs, such as text prompts, enabling generation of prompt-driven images and a wider variation in styles [8][9]. However, GAN and its variations struggled with difficulties in training, complications with conditioning image generation on prompts and issue with complex scenes led to their decline. While GANs are less computationally intensive compared to newer image generative models, their limitations in producing diverse, highly detailed images make them unsuitable for this project.

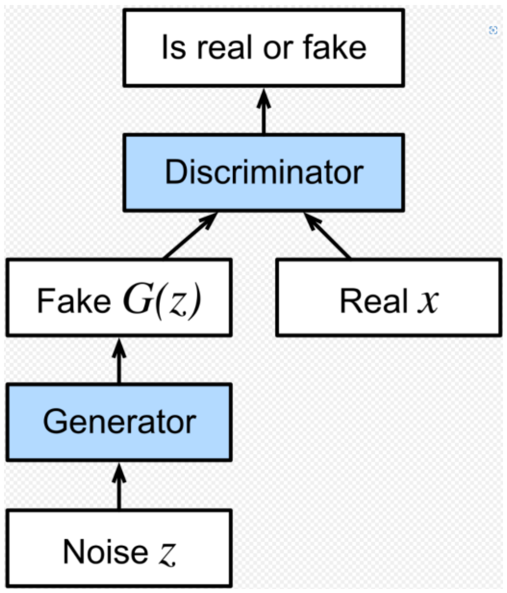


Figure 2: High-level overview of GAN training. Source: D2L.ai

2.2.2 Transformer based Models

DALL-E was one of the state-of-the-art (SOTA) models when it was first released to the public in 2021. It was among the first models with an ability to generate complex scenes and high fidelity from textual prompts. It leveraged transformer-based architecture which allowed the model to capture semantics of prompts and map them to corresponding visual representations effectively [10].

DALL-E utilized autoregressive transformers trained on a vast dataset of text-image pair [11]. Similar to the language models generating text, it synthesizes images by predicting the image tokens in a sequence. Although it was an influential model, it also had several limitations. Some of the limitations are as follows:

- High computational cost - due to its nature of the architecture
- Image coherence – sometimes struggle maintaining image coherence and complex scenes such as hands and symmetrical structures
- Limited resolution – Model is constrained by its training data's resolution. Upscaling the model to generate higher resolution images required datasets of higher resolution images

which increased the computational demand, making it costly and ineffective for widespread practical use.

There were also other autoregressive transformer-based image generative models such as Pathways Autoregressive Text-to-Image (PARTI) and Make-A-Scene. Despite the improvements over the original DALL-E, they also shared similar downsides.

For this project, which focuses on deploying a text-to-image generation model on mobile devices for on-device inference, these transformer-based models are not suitable. The mobile devices are often limited by their processing power, available memory, storage and battery. Therefore, high computational requirements of autoregressive transformer models make it impractical for use on mobile devices. Furthermore, the large size of the model and the need for a significant amount of computing resources would prevent efficient execution on the device, making them unsuitable for the goal of generating local, on-device image generation on mobile platforms.

2.2.3 Stable Diffusion

Transformer-based models like DALL-E, PARTI, and Make-A-Scene primarily operated in pixel space or image tokens for synthesizing images. Pixels or image tokens are generated one at a time and future tokens are generated sequentially to form a complete image which makes them computationally expensive.

Stable Diffusion, on the other hand, adopts a more efficient approach by working in a latent space and it is the key innovation behind Stable Diffusion. It utilizes a latent diffusion model where the images are generated in a compressed and lower-dimensional space known as latent space rather than operating directly in pixel space [12]. During the image generation process, Stable Diffusion progressively denoise an initial random noise in a latent space through multiple steps, which are known as denoising steps, and a synthesized image is achieved at the end of denoising step. Due to this property, significant computational costs are reduced and making it possible to generate high-quality images with fewer resources as well as making it ideal for deployment on resource constrained edge and mobile devices.

Stable Diffusion image generation pipeline is composed of several key components such as text tokenizer, text encoder, UNet, Scheduler, and Variational Autoencoder. The following sections provide a high-level overview of each component and its role in image generation process.

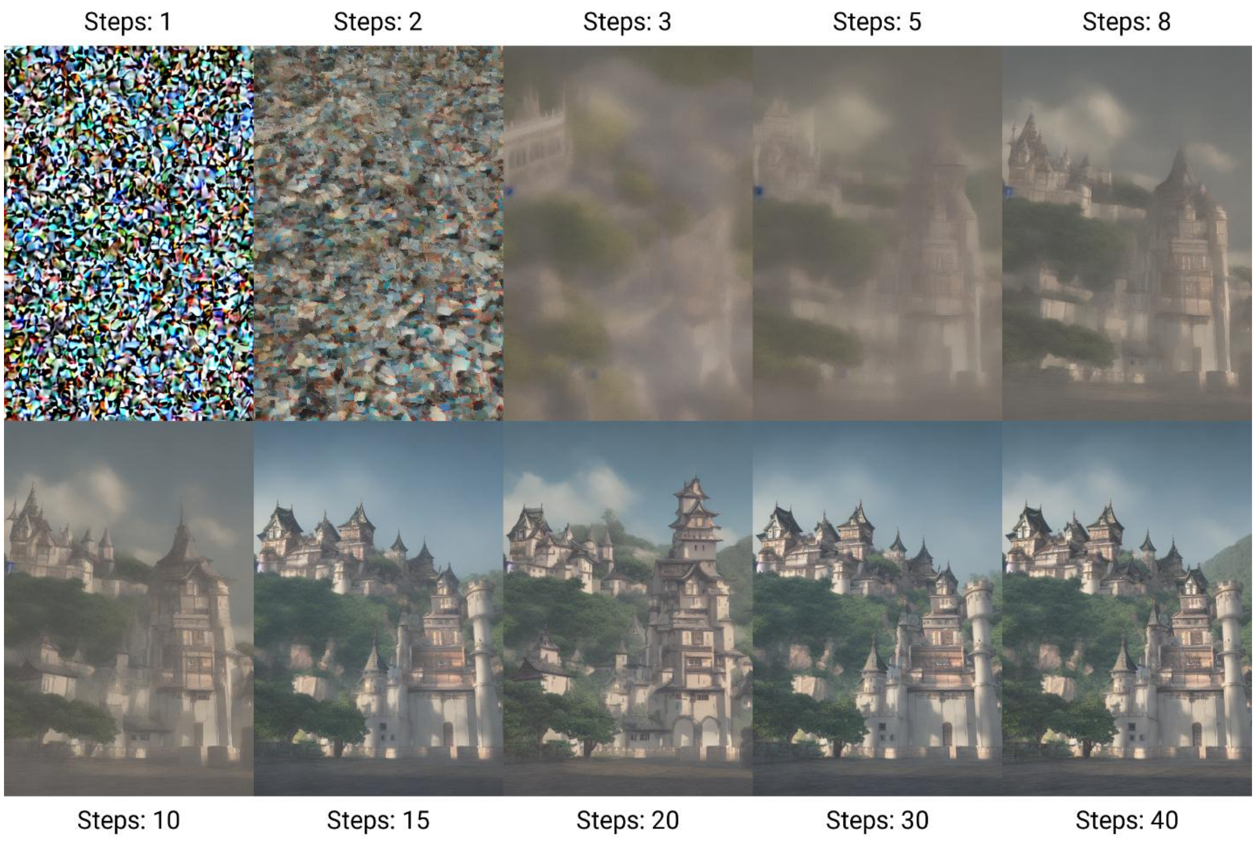


Figure 3: Stable Diffusion image generation process. Source: Stable Diffusion Wikipedia

2.2.3.1 Text Tokenizer and Encoder

Text tokenization and encoding make up the first stage of Stable Diffusion image generation pipeline. Since user-provided prompts are in textual format, an efficient algorithm is required to break the text into meaningful tokens or chunks which the text encoder can process without losing the intricate details of the input text. These tokens play a crucial role in enabling the text encoder to generate embeddings, which in turn condition the other components to generate images that closely align with the user-provided prompts. Byte Pair Encoding (BPE) is employed to perform tokenization tasks, offering a balance between character-level and word-level tokenization to efficiently represent text while maintaining semantic integrity [13].

After the prompts are tokenized, the next part is to encode the tokens into embeddings. Embeddings are dense vector representations of tokenized texts, capturing both semantic meaning and contextual relationships between words. Stable Diffusion uses Contrastive Language-Image Pretraining (CLIP) text encoder. CLIP is a model trained on large datasets of images and their corresponding captions to learn the representation of words and corresponding images [14]. These embeddings are used as conditioning inputs for image generation process.

2.2.3.2 UNet

The UNet is the second stage of Stable Diffusion image generation pipeline and actual image generation happens in this stage. It is a type of convolutional neural network (CNN), originally developed for biomedical image segmentation tasks [15]. In the Stable Diffusion pipeline, the UNet is responsible for predicting the noise present in the latent representation at each step of the denoising process.

At the initial stage of image generation, a random noise following a Gussian distribution is generated in latent space. The UNet then iteratively predicts the noise component of this latent representation. Each noise prediction is guided by conditions, including the text embeddings (from the text encoder) and the current timestep. The timestep, as part of the input to the U-Net, helps the model understand how much noise to predict and remove at each step. The actual denoising process is guided by the U-Net's predictions in conjunction with the scheduler, which controls the level of noise removal at each step. This iterative process continues until the noise is sufficiently removed, yielding a final image that aligns with the given prompt.

2.2.3.3 Scheduler

The Scheduler plays an important role in the Stable Diffusion image pipeline, particularly in the denoising stage. It defines the noise schedule, determining how much noise should be retained or removed at each denoising step. While the Scheduler does not directly communicate this schedule to the U-Net, it provides the timestep as input to the U-Net, which indirectly guides the noise removal process. The U-Net uses the timestep to adjust its predictions of the noise to be removed, iteratively refining the latent noise based on the information from the Scheduler.

The choice of Scheduler is crucial, as it directly impacts the quality and computational efficiency of the image generation. A well-designed Scheduler helps the model efficiently transform random noise into a coherent image, reducing unnecessary computation while improving the accuracy of the final result. Some commonly used schedulers include DDIM Scheduler (Denoising Diffusion Implicit Models), PNDM Scheduler (Pseudo Numerical Methods for Diffusion Models), and PLMS Scheduler (Pseudo Linear Multi-Step). PNDM Scheduler is chosen for this project as it offers a balanced trade-off between computational efficiency and image quality.

2.2.3.4 Variational Autoencoder

Variational Autoencoder (VAE) is the last stage of Stable Diffusion image generation pipeline. It consists of two components, namely, the encoder and the decoder [16]. During the training phase of Stable Diffusion, the VAE encoder maps the images into a lower dimensional latent space. Random noise is then added to these latent representations according to a forward diffusion process. The UNet is trained to predict and remove this noise in reverse order, learning to gradually denoise the latent representations back to their original state.

However, during the inferencing phase of image generation, only the VAE decoder is used. After the UNet has completed the denoising steps, the output exists in the latent space. The VAE decoder is then used to convert the denoised latent representation back into pixel space. Once the pixel data is obtained, the image can be reconstructed and presented as the final output.

2.3 Edge and Mobile Device Deployment of Generative Models

Generative models typically require a significant number of resources in terms of computational power, memory, energy and storage. Deploying these models on edge and mobile devices is a very challenging task due to the hardware and energy limitations. However, various techniques of optimization are available today such as model quantization, pruning and knowledge distillation [17]. Although these techniques make inferencing generative models on edge and mobile devices more accessible, the model tends to suffer from loss of accuracy, and irregular neural network structures sometimes result in suboptimal performance.

Pruning the model involves removing the redundant or less important weights from the model [18]. However, the model will suffer from loss of some accuracy as a result of pruning.

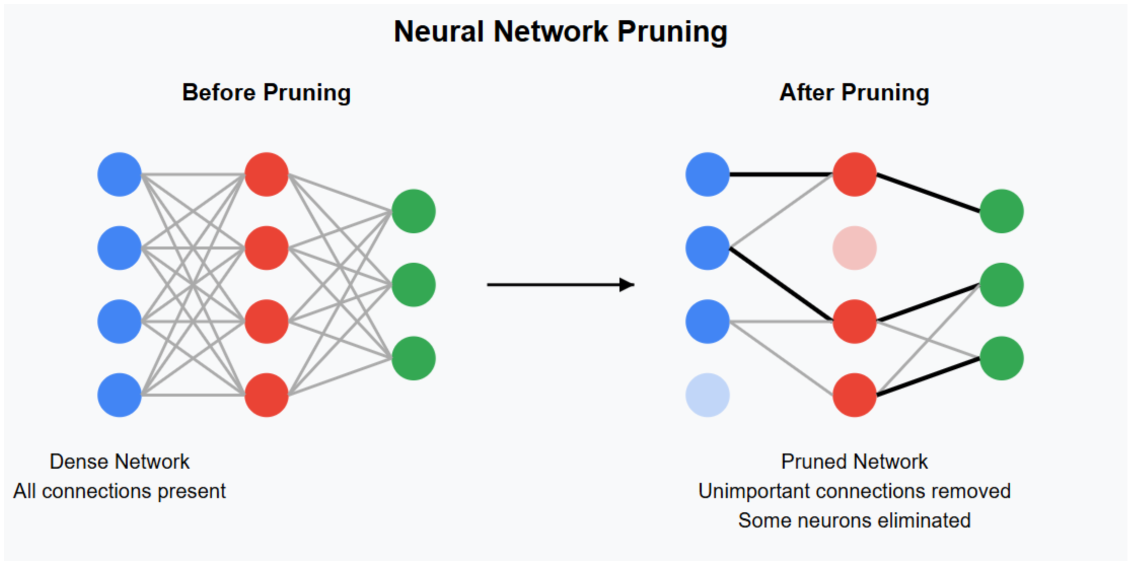


Figure 4: Neural Network Pruning Illustration

Quantization is another technique of optimization where a model operates on a reduced precision [19]. For instance, replacing 32-bit floating point with 8-bit integer can significantly reduce the memory footprint and improve inference speed although the model will suffer from loss of accuracy. Therefore, it is important to re-evaluate the model to assess whether the model still performs at a reasonable level of accuracy.

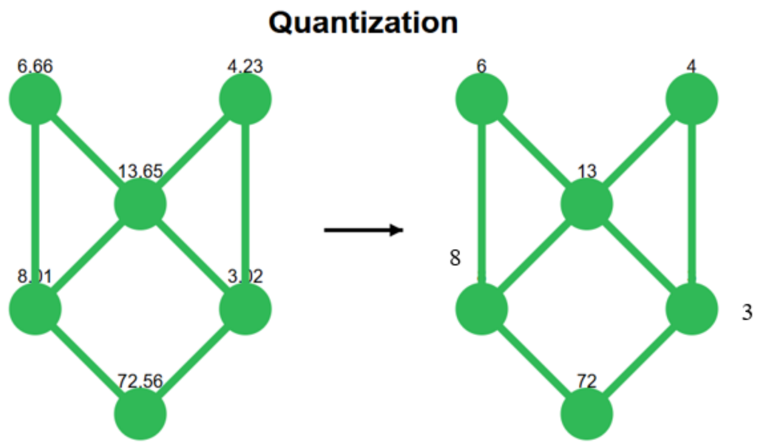


Figure 5: Neural Network Quantization Illustration

Knowledge distillation involves training a new smaller model from a larger model [20]. Although this technique can effectively transfer knowledge and reduce model size, it may also lead to a loss of fine-grained information, as the smaller model might struggle to capture complex patterns learned by the larger model. Additionally, computational resources are still required to train the smaller model, making the knowledge distillation optimization technique less suitable and accessible for individuals with limited hardware. Therefore, quantization and pruning stand out as the preferred techniques for this project if optimization is required for successful deployment of Stable Diffusion model onto mobile phone.

Another technique to ensure successful deployment on edge devices is the hybrid approach, which offloads certain heavy tasks to on-premises resources or cloud services while keeping lightweight operations on the edge or mobile devices. However, this approach introduces additional challenges, such as network dependency, potential security and privacy risks. Therefore, this project will perform the entire image generation process on-device without offloading computational heavy tasks to the network. However, additional features which require heavy computations will utilize resources within the Local Area Network (LAN) to strike a balance between project development speed, efficiency and computational resources.

2.4 Related Works

Several implementations of text-to-image generation on mobile devices have been made in recent years. Notably, there are four existing implementations that demonstrate the feasibility of on-device text-to-image generation, each employing a distinct design approach. The following section outlines the strategies used by these implementations and evaluates their potential for adoption in this project.

2.4.1 Mobile Diffusion

Mobile Diffusion, developed by Google Research, is an internal implementation of diffusion model optimized for mobile devices [21]. The model is capable of generating image in less than a second on device locally. Although very efficient, the model is not publicly available impeding the application of this model in the project. While not available for use in this project, it demonstrated that near instant image generation is achievable through extensive optimizations in both architecture and sampling technique.

Due to the limitations of available resources and the author’s current knowledge, replicating this level of optimization within the scope of this project was not feasible. The optimizations required, particularly in terms of architectural adjustments and advanced sampling methods, would require significant computational expertise and hardware resources beyond the scope of the present work.

2.4.2 MediaPipe On-Device Text-to-Image Generation

MediaPipe is an end-to-end framework designed to facilitate the deployment of machine learning models on mobile and edge devices. It provides support for a wide range of on-device inference tasks, including but not limited to object detection, image segmentation, face detection, text classification, and image generation [22]

Despite its broad capabilities, MediaPipe's on-device text-to-image generation feature is still an experimental feature. As a result, it is currently not fully supported by Android emulators, limiting its use in development environments that rely on simulated devices. This experimental nature also means that the feature may not be as stable or optimized as other more mature implementations, but it demonstrates the potential for on-device image generation with minimal reliance on cloud resources.

2.4.3 Stable Diffusion on Termux

Termux is an Android terminal emulator and Linux environment app [23]. This app enables users to execute Python in a terminal environment. As a result, this app allows execution of various Python programs, including AI models like Stable Diffusion on Android devices. However, requiring additional set-up configurations, command-line interface and limited level of customization and optimization makes this approach unsuitable for this project.

2.4.4 Qualcomm AI Engine Direct

Qualcomm AI Engine Direct is a proprietary software development kit developed by Qualcomm Technologies [24]. It provides a unified development experience of on-device AI inference tasks, allowing efficient execution of machine learning models, including text-to-image generation model, on Qualcomm Snapdragon processors. Since this is specifically designed for Qualcomm chips, its application is primarily limited to devices powered by Snapdragon processors. Consequently, this makes it unsuitable for adoption in the project, as our development should support all Android phones, which use a variety of chipsets, including Qualcomm Snapdragon, Google Tensor, and others.

Chapter 3

Architecture and Technologies Used

In this chapter, the overall architecture of the system is described, along with the technologies and frameworks utilized in its development. The system consists of multiple components that work together to facilitate on-device image generation, with optional support from a locally deployed large language model (LLM) for prompt enrichment. The chapter provides an in-depth discussion of the software architecture, development frameworks, and the emulated hardware configuration used for development.

3.1 System Architecture

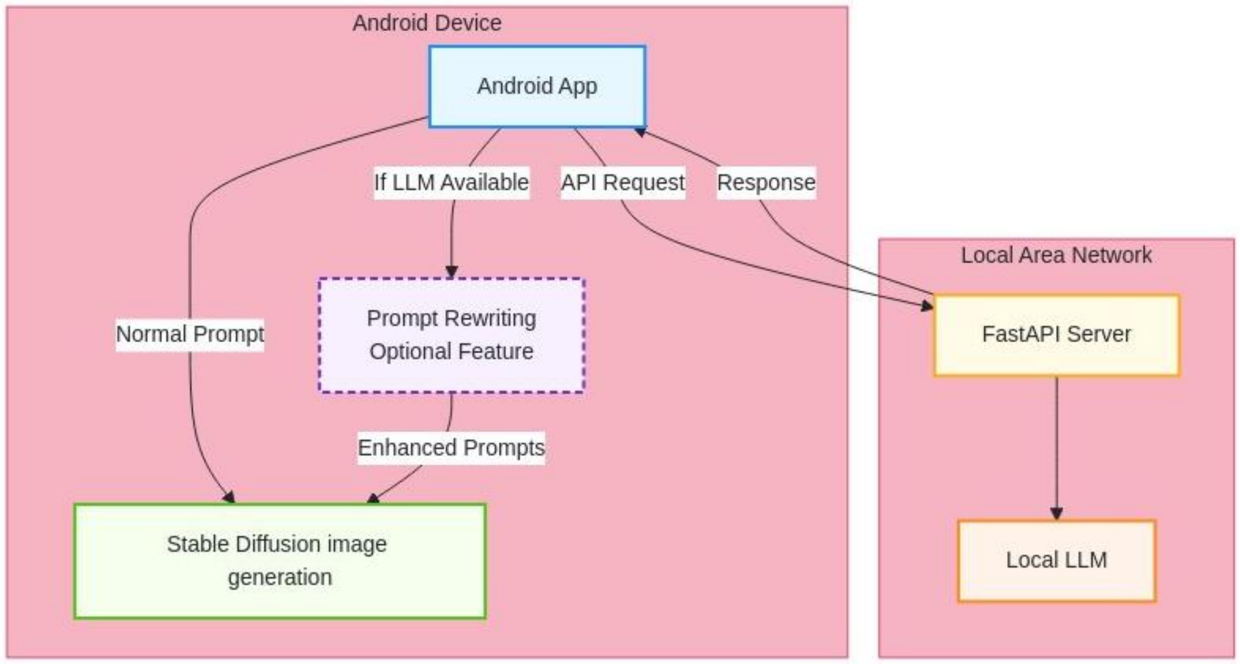


Figure 6: System Architecture

The system consists of two components: the Android application and the FastAPI server. The Android application is a self-contained system capable of generating images from user-provided prompts without requiring any connection to the server. Optionally, the app utilizes a large language model (LLM) deployed on the local network for prompt enrichment tasks when it is available. This design choice was made because it is storage and computationally intensive to carry out both image generation and LLM inferencing tasks on mobile devices.

Upon launching the Android application, an API request is made to the FastAPI server deployed on the local area network to check for its availability. If the server is accessible, the app enables prompt rewriting-optional feature, and the server provides the necessary resources for prompt enhancement tasks via RESTful API communication. REST APIs are stateless, and each request from the application to the server is independent and contains all the information required for processing the request [25]. The server does not maintain session information or state between requests, which enhances the scalability and flexibility of the system by allowing multiple requests to be processed independently.

When the user chooses to rewrite the prompt, it is sent to FastAPI server. The LLM deployed on the LAN is instructed with special instructions designed to enhance its ability to enrich prompts for image generation. These instructions guide the LLM to focus on rewriting the specific aspects of the input prompt to improve image accuracy, such as physical characteristics, colours, and distinct physical features of objects mentioned in the prompts. Once the prompt has been enriched with additional information, it is returned back to the Android app where the image generation occurs.

3.1.1 MVVM Architecture

In this project, Model View View Model (MVVM) architectural pattern is applied for an Android application. In the MVVM architectural pattern, the responsibility of each component can be outlined as follows:

- Model – Responsible for managing data and business logic execution. It typically interacts with databases, cloud APIs or local storage. The Model directly communicates with View Model layer, providing the data that the View Model requires and handles data storage and execution of business logic as requested by View Model.

- View Model – Acts as a bridge between Model and View layer. Additionally, the View Model also holds instances which need to survive configuration changes such as when the phone screen rotates. View Model exposes simple API for View to observe, allowing it to receive data change updates without directly managing or interacting with the underlying business logic and data.
- View – Responsible for UI representation and handling user interactions such as button click, scroll, and text input. It observes the View Model for updates and updates the UI and UI states accordingly.

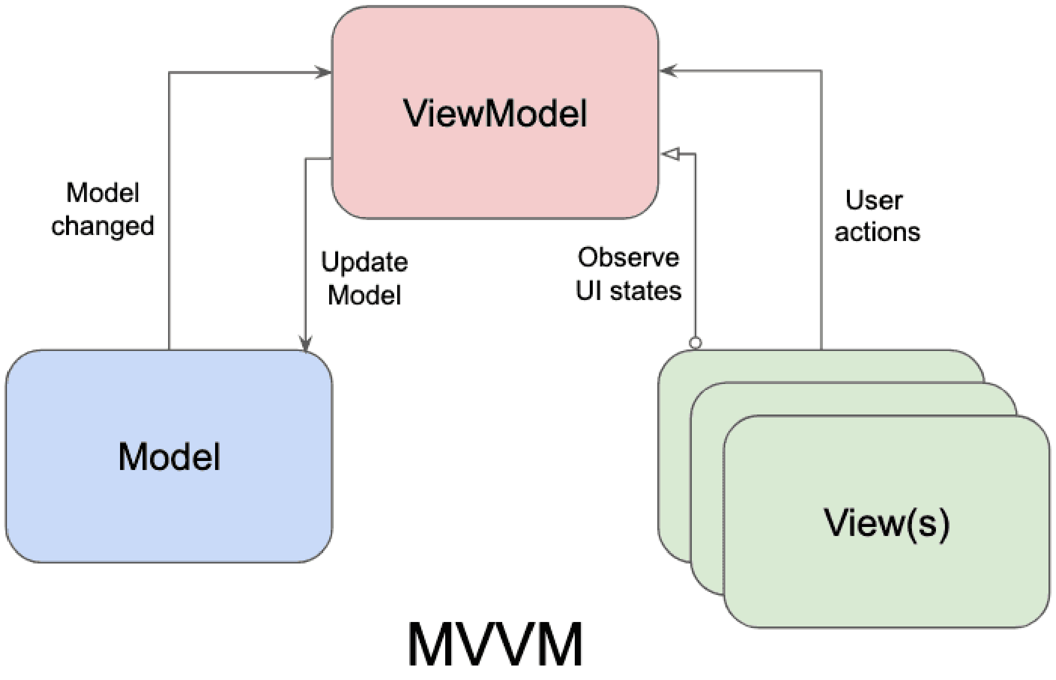


Figure 7: MVVM Architectural Pattern

3.1.2 Android App Architecture

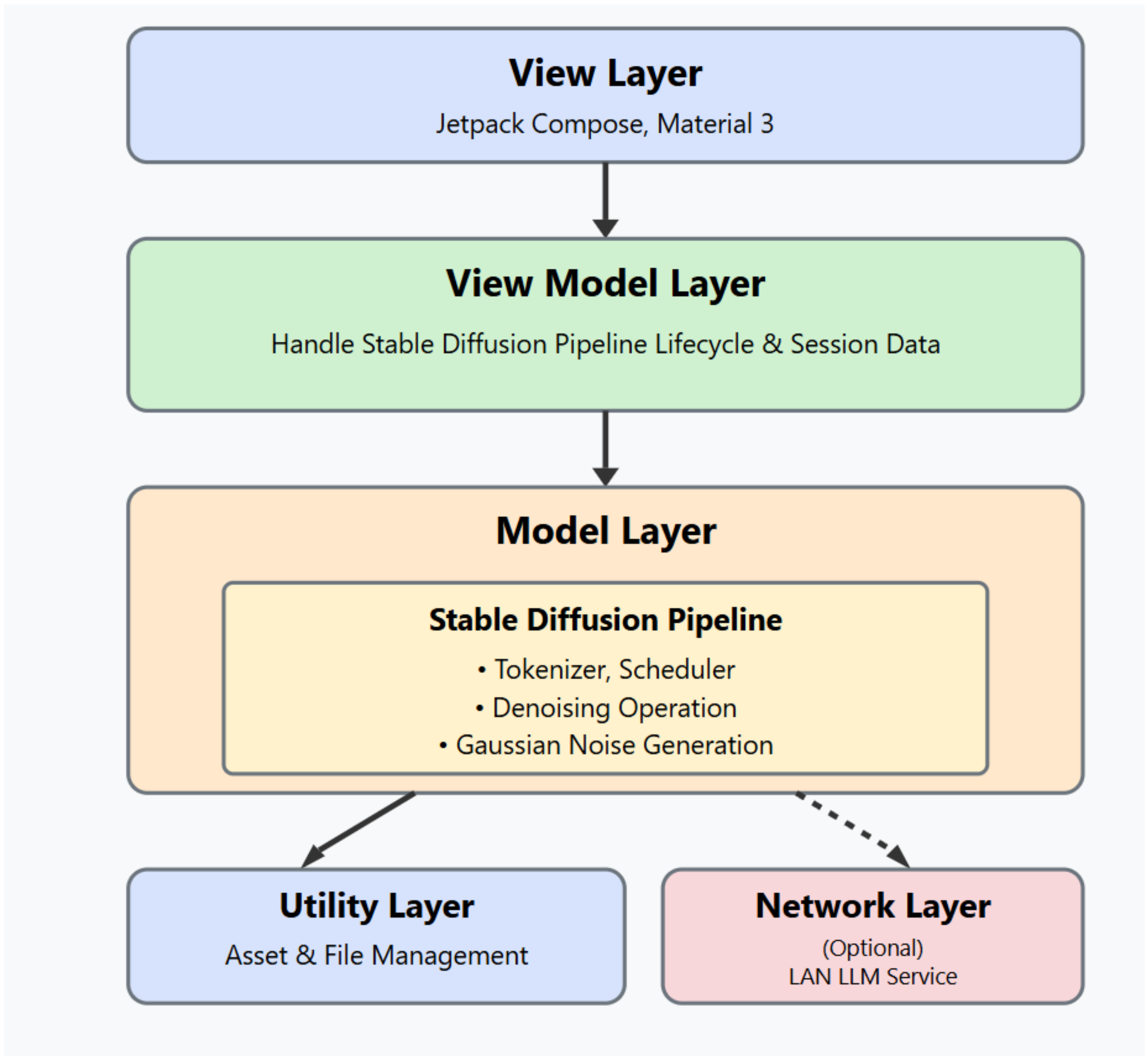


Figure 8: Overall Android App Architecture

As shown in Figure 7, the View layer continuously observes changes in the View Model layer to update the UI accordingly. User interface is composed by utilizing UI frameworks such as Jetpack compose and Material 3 theme. The UI layer calls the APIs or uses abstracted instances provided by the View Model layer when triggered by user interactions such as button click.

View Model layer serves as the intermediary between the Model layer and the View layer. This layer manages the lifecycle of the pipeline instance and session data such as LAN LLM status to avoid re-creating multiple pipeline instances and retain session data, during configuration changes such as screen rotation, app pausing and background processing.

The Model layer encapsulate core business logic for image generation. This layer implements the Stable Diffusion pipeline along with other utility functions necessary for image generation. Business logic is abstracted and hidden from both View and View Model layers, ensuring that the internal workings of the pipeline are not directly exposed. As a result, this separation allows for flexibility, such as the ability to replace or modify the pipeline implementation without affecting the View or View Model layers.

Among the several benefits achieved through the application of MVVM architecture, the main benefits are outlined as follows:

- Separation of concern: MVVM ensures that business logic and UI logic remain separated. As a result, it becomes more manageable and scalable.
- Testability: As the business logic and UI logic are separated, each logic can be tested independently without relying on other.
- Android Lifecycle Awareness: On Android devices, configuration changes such as screen rotation can trigger the recreation of UI components, leading to the loss of transient data and the need for reinitialization. By using the View Model to manage instances of business logic (e.g., the Stable Diffusion pipeline), we ensure that critical components are preserved across these lifecycle changes. This approach prevents data loss and optimizes app performance, allowing for smoother user experience even during configuration changes.

3.2 Tools and Frameworks

3.2.1 Kotlin

Kotlin is a new statically typed programming language developed as an alternative to Java. Interoperability with Java allows Kotlin to be operated on any machines with Java virtual machine.

It is also now the preferred language, recommended by Google for Android apps development. The main advantage of Kotlin is that codes written are more compact compared to its ancestor Java [26]. Due to these several qualities, it is now adopted by more than 60% of professional Android Developers, making it an ideal choice of language for this project [27].

3.2.2 Jetpack Compose

Jetpack Compose is a modern and declarative toolkit developed by Google for building android native UI interfaces. It is also the Android's recommended toolkit for building UI elements. Contrary to the old practice of using XML for designing user interface, jetpack compose reduces the need for boilerplate code. Additionally, it offers live previews of UI when changes are made to allow easier iteration and faster development time [28]. Hence, Jetpack Compose is the primary choice for UI development of this project.

3.2.3 PyTorch and PyTorch Mobile

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab and it integrates seamlessly with Python. A mobile version of PyTorch is also available, namely, PyTorch Mobile. In this project, PyTorch is used for initial experimentation of Stable Diffusion models on desktop environment and converting certain components of Stable Diffusion into a format compatible with Android OS. PyTorch mobile is also used for inferencing tasks of certain Stable Diffusion components on Android environment.

3.2.4 Open Neural Network Exchange (ONNX)

Open Neural Network Exchange (ONNX) is an open-source, interoperable format designed to facilitate the deployment of machine learning models across different frameworks and hardware platforms. It enables models trained in various deep learning frameworks, such as PyTorch and

TensorFlow, to be converted into a standardized ONNX format, ensuring compatibility across multiple environments, including edge devices and mobile platforms like Android.

In the context of this project, ONNX is used for the deployment of some Stable Diffusion model components on Android OS. By converting the model into ONNX format, it enables efficient execution with optimized runtime support, leveraging ONNX Runtime (ONNX RT) for accelerated inference. This approach significantly enhances model portability while reducing framework dependencies, making it well-suited for this project. Additionally, ONNX facilitates hardware acceleration through backend optimizations, allowing better performance on resource-constrained devices.

3.2.5 Hugging Face Diffusers and Transformers

Hugging Face's Diffusers and Transformers libraries provide a comprehensive framework for inferencing generative AI models, including Stable Diffusion. These libraries were initially utilized in a desktop environment to experiment with and analyze the Stable Diffusion pipeline. Additionally, their official GitHub repository served as a reference for implementing a Stable Diffusion pipeline in Android's native Kotlin environment.

However, since Hugging Face's libraries are primarily developed in Python and optimized for desktop and cloud-based execution, direct integration into an Android-native application presents several challenges. The libraries contain extensive code sections that are unnecessary for this project, leading to inefficiencies when deployed on resource-constrained mobile devices. To address this, only the necessary components are rewritten in Kotlin, ensuring efficient on-device inference while minimizing overhead.

3.2.6 FastAPI

FastAPI is a Python framework for building RESTful API server. It follows RESTful principles, ensuring that requests are processed without having to maintain session data. This characteristic enhances scalability, as multiple concurrent requests can be handled efficiently without shared server-side state.

3.3 Virtual Hardware Configuration

The development of this project primarily took place in Android Studio leveraging the Android emulator feature due to the unavailability of physical Android devices. Therefore, a software emulated android device is used for this project development. The Android emulator provides a virtual environment which replicates the behavior of a physical device and allows developing, testing and debugging under various configurations.

However, the emulator does not support GPU acceleration for ONNX Runtime and PyTorch mobile due to its reliance on software-based graphics rendering rather than direct access to hardware-accelerated APIs such as Vulkan or OpenCL. As a result, optimizations involving GPU acceleration could not be implemented, and all model inferences were executed using CPU-based processing.

Specification	**Value**
Device Model	Pixel 9 Pro
Android Version	Android 15
CPU Architecture	x86_64 (Emulated)
RAM	10 GB
Internal Storage + SD	10GB + 6GB

Table 1: Emulated Device Configuration

Chapter 4

Setup and Implementation

This chapter covers the implementation process and the practical aspects of implementing the system are discussed. The focus is on deploying Stable Diffusion for on-device image generation, building the pipeline and integrating a Large Language Model (LLM) hosted on a local area network (LAN).

4.1 Stable Diffusion Model Conversion and Selection for Android Deployment

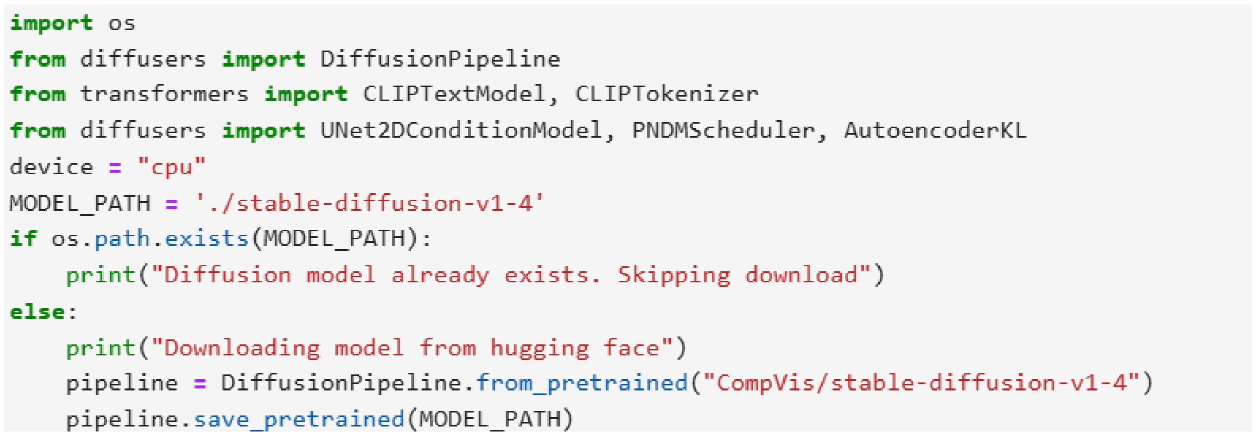
4.1.1 Model Selection

Among the Stable Diffusion Models, the CompVis/stable-diffusion-v1-4 model was selected for this project. This model is primarily chosen due to its balance between image fidelity and computational efficiency. Additionally, this model is publicly available on HuggingFace, making it more accessible.

4.1.2 Model Conversion

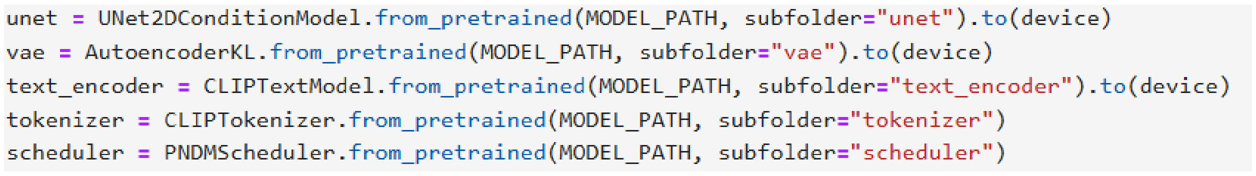
Before deploying the model to Android devices, it was necessary to convert model into a format compatible with Android environment. This is due to the hardware architectural difference between desktop and mobile platforms, which require optimizations for efficient deployment on devices with limited resources such as memory and processing power.

The model is first downloaded from HuggingFace and saved on the PC. By saving it on PC, it makes it easier to access and allow manual loading of each Stable Diffusion Components as discussed in Section 2.2.3 of the Literature Review.



```python
import os
from diffusers import DiffusionPipeline
from transformers import CLIPTextModel, CLIPTokenizer
from diffusers import UNet2DConditionModel, PNDMScheduler, AutoencoderKL
device = "cpu"
MODEL_PATH = './stable-diffusion-v1-4'
if os.path.exists(MODEL_PATH):
    print("Diffusion model already exists. Skipping download")
else:
    print("Downloading model from hugging face")
    pipeline = DiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4")
    pipeline.save_pretrained(MODEL_PATH)
```

Figure 9: Downloading Model from HuggingFace



```python
unet = UNet2DConditionModel.from_pretrained(MODEL_PATH, subfolder="unet").to(device)
vae = AutoencoderKL.from_pretrained(MODEL_PATH, subfolder="vae").to(device)
text_encoder = CLIPTextModel.from_pretrained(MODEL_PATH, subfolder="text_encoder").to(device)
tokenizer = CLIPTokenizer.from_pretrained(MODEL_PATH, subfolder="tokenizer")
scheduler = PNDMScheduler.from_pretrained(MODEL_PATH, subfolder="scheduler")
```

Figure 10: Loading Stable Diffusion Components Separately

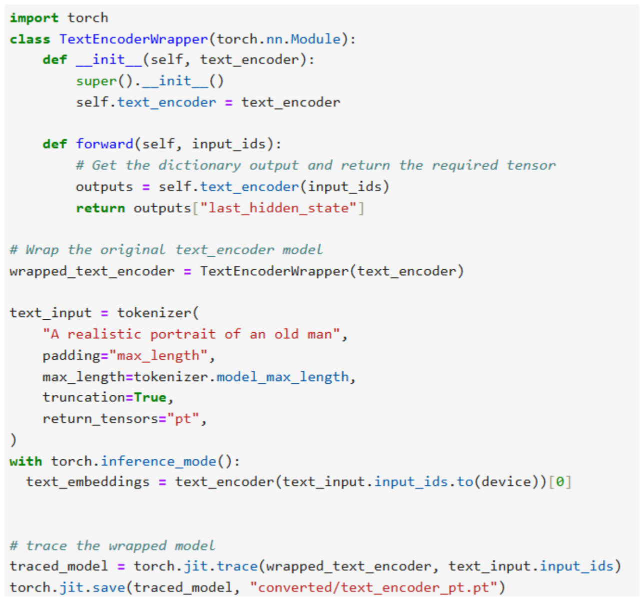
Among the components, UNet, Variational Autoencoder (VAE), and text encoder are models which require conversion into a format compatible with the mobile environment. The other components such as tokenizer and scheduler are functional components that do not require conversion but instead need to be re-implemented in Android app to ensure proper functionality in the mobile environment.

4.1.2.1 Text Encoder Conversion

As described in section 2.2.3.1of Literature Review, user-prompt needs to be tokenized before being passed to the text encoder. The downloaded model uses CLIP Text Model, a transformer based neural network model for encoding the tokenized prompts into text embeddings.

PyTorch just-in-time tracing (JIT tracing) technique is used for converting the model. PyTorch JIT tracing works by recording the operations performed during a model's forward pass when provided with a sample input, creating a static computation graph [29]. This process results in an optimized, platform-independent model which can be executed in non-Python environments, and it is most effective for models with static control flow as it cannot capture dynamic behavior dependent on input values.

As described in Figure 11, the text encoder must be wrapped as PyTorch JIT tracing requires a traceable output, and the model returns a dictionary, which is not directly compatible with the tracing mechanism. By defining a custom TextEncoderWrapper class, we explicitly specify how to process the dictionary output, such as extracting the "last_hidden_state" tensor, ensuring compatibility with JIT tracing. Without this wrapper, the tracer would encounter errors when attempting to trace the dictionary structure. This approach simplifies the model's output to a tensor, enabling successful tracing and conversion.



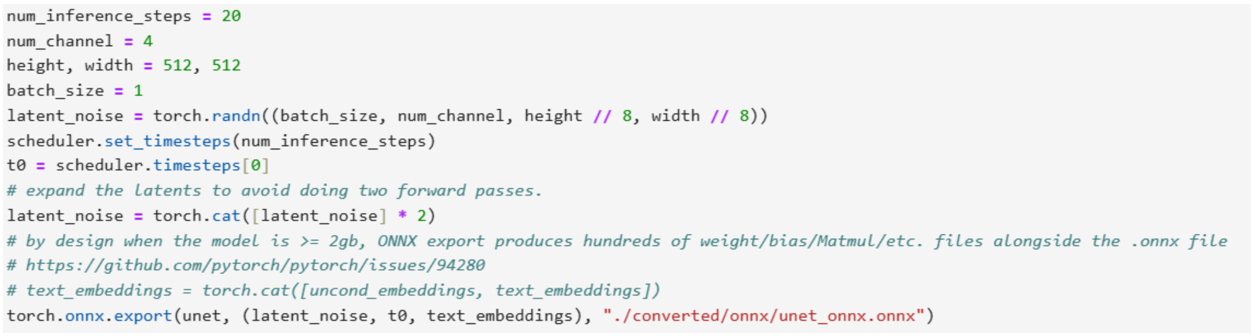
```
import torch
class TextEncoderWrapper(torch.nn.Module):
    def __init__(self, text_encoder):
        super().__init__()
        self.text_encoder = text_encoder

    def forward(self, input_ids):
        # Get the dictionary output and return the required tensor
        outputs = self.text_encoder(input_ids)
        return outputs["last_hidden_state"]

# Wrap the original text_encoder model
wrapped_text_encoder = TextEncoderWrapper(text_encoder)

text_input = tokenizer(
    "A realistic portrait of an old man",
    padding="max_length",
    max_length=tokenizer.model_max_length,
    truncation=True,
    return_tensors="pt",
)
with torch.inference_mode():
  text_embeddings = text_encoder(text_input.input_ids.to(device))[0]


# trace the wrapped model
traced_model = torch.jit.trace(wrapped_text_encoder, text_input.input_ids)
torch.jit.save(traced_model, "converted/text_encoder_pt.pt")
```

Figure 11: Text Encoder Conversion

4.1.2.2 UNet Conversion

As described in section 2.2.3.2 of Liter Review, UNet relies on embedding data from Scheduler and Text Encoder. Primarily, there are two ways of conversion techniques provided by PyTorch, namely, tracing and scripting. Due to the structural complexities of UNet model and its dependency on dynamic input, converting it using PyTorch presents limitations.

Due to the early stages of the Edge AI ecosystem, and rapid evolution, the tools and frameworks often go major changes or stop being supported in favour of newer tools and frameworks. PyTorch Mobile is no longer actively maintained and lacks certain features required for converting the UNet model. As an alternative approach, the model is exported to an ONNX format which is also compatible with Android devices for inferencing tasks.

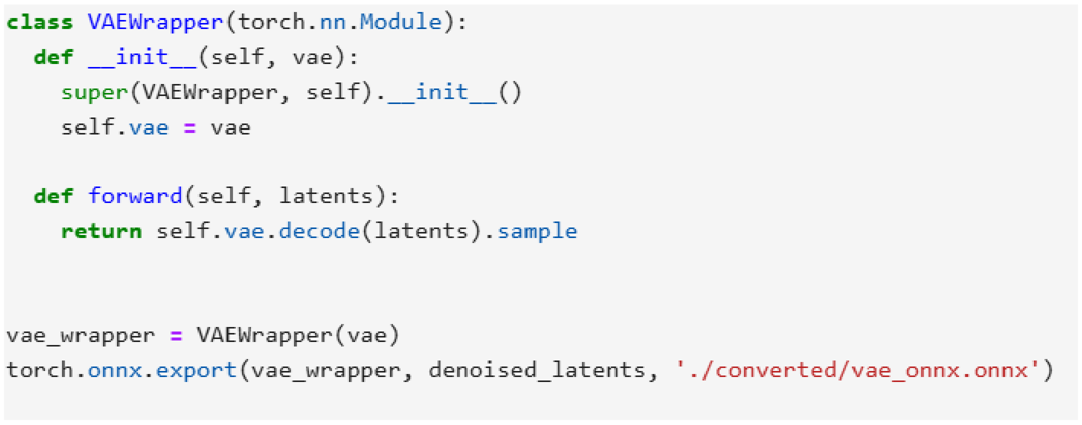


```
num_inference_steps = 20
num_channel = 4
height, width = 512, 512
batch_size = 1
latent_noise = torch.randn((batch_size, num_channel, height // 8, width // 8))
scheduler.set_timesteps(num_inference_steps)
t0 = scheduler.timesteps[0]
# expand the latents to avoid doing two forward passes.
latent_noise = torch.cat([latent_noise] * 2)
# by design when the model is >= 2gb, ONNX export produces hundreds of weight/bias/Matmul/etc. files alongside the .onnx file
# https://github.com/pytorch/pytorch/issues/94280
# text_embeddings = torch.cat([uncond_embeddings, text_embeddings])
torch.onnx.export(unet, (latent_noise, t0, text_embeddings), "./converted/onnx/unet_onnx.onnx")
```

Figure 12: Export UNet to ONNX format

4.1.2.3 Variational Autoencoder Conversion

As described in section 2.2.3.4 of Literature Review, Variational Autoencoder (VAE) is needed for scaling up the latent data output from UNet into pixel space so that the generated image can be re-constructed. As an input, VAE expects a latent generated from UNet. Due to the same limitations as described in the UNet conversion section, the model cannot be directly converted to a mobile compatible format using tracing or scripting techniques. Hence, the VAE model is also exported to mobile compatible format, ONNX. To facilitate the conversion, the VAE model is encapsulated within a wrapper class before being exported. This wrapper ensures that only the required functionality, specifically the decoding process, is retained while removing unnecessary complexities that may hinder conversion. By wrapping the model, the exported ONNX representation remains streamlined and optimized for inference on mobile devices.



```python
class VAEWrapper(torch.nn.Module):
  def __init__(self, vae):
    super(VAEWrapper, self).__init__()
    self.vae = vae

  def forward(self, latents):
    return self.vae.decode(latents).sample


vae_wrapper = VAEWrapper(vae)
torch.onnx.export(vae_wrapper, denoised_latents, './converted/vae_onnx.onnx')
```

Figure 13: Export VAE to ONNX format

4.2 System Implementation

4.2.1 Android App Project Structure

As illustrated in Figure 14, the Android application follows a modular MVVM (Model-View-View Model) architecture, ensuring a clear separation of concerns and maintainability. The project's structure is organized into distinct packages, each serving a specific purpose in handling the diffusion model, network operations, user interface components, and application logic.

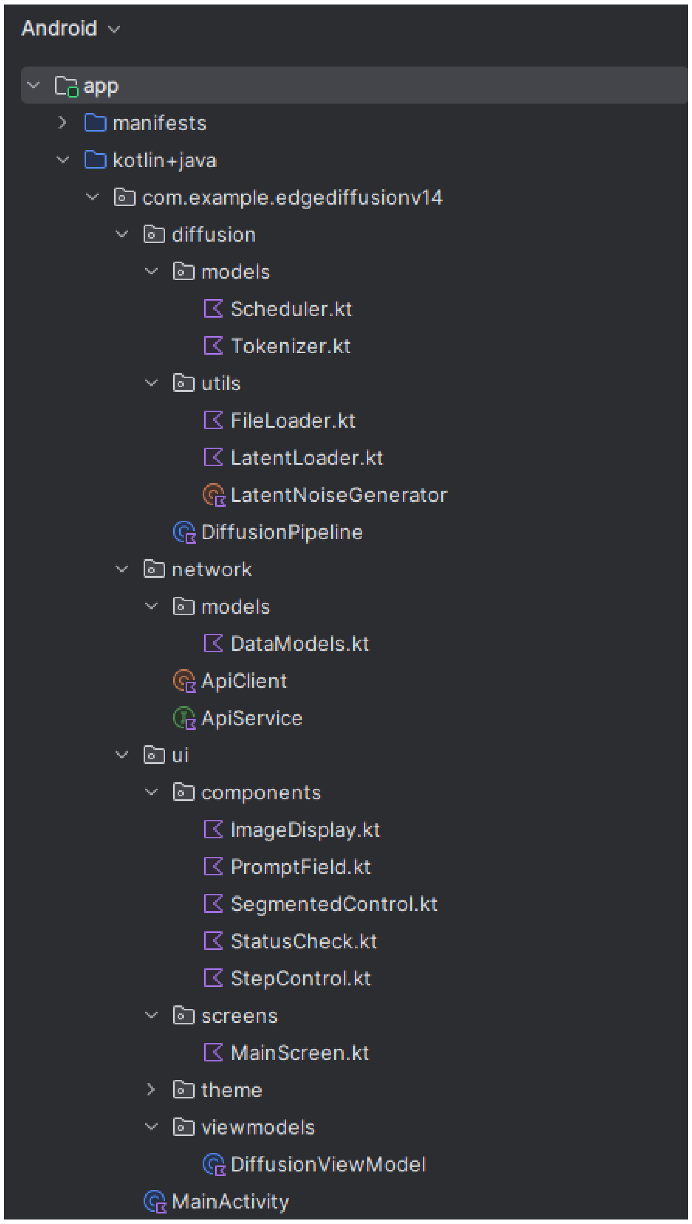
The high-level overview and structure of the project is as follows:

1. com.example.edgediffusionv14 (Root Package)
    - This is the main package containing all submodules for image generation, networking, UI components, and app life cycle management.
2. Diffusion (Model Processing Logic – Model Layer)
    - Models: Contains the core components required for Stable Diffusion image generation.
        - Scheduler.kt: Implementation of Scheduler - Pseudo Numerical Methods for Diffusion Models (PNDM Scheduler).

        - Tokenizer.kt: Tokenizes user input prompts before passing them to the text encoder.
    - Utils: Contains helper utilities for processing input and generating latent noise.
        - FileLoader.kt: Manages model loading operations into the App.
        - LatentLoader.kt: Handles the loading of pre-seeded latent noise from the desktop environment to ensure consistency and enable reproducibility between desktop and mobile environments.
        - LatentNoiseGenerator.kt: Generates Gussian noise based on the seed.
    - DiffusionPipeline: A primary pipeline class orchestrating the diffusion model image generation execution.
3. Network (Communication with LAN – Part of Model Layer)
    - Models
        - DataModels.kt: Define data structures used for API communication with the FastAPI server deployed on LAN.
    - ApiClient.kt: Manages API request tool configuration
    - ApiService.kt: Defines API endpoints for exchanging data with LAN FastAPI server.
4. UI (User Interface – View Layer)
    - Components: contain reusable UI components adhering DRY (Don't Repeat Yourself) principle. The reusable components are modular, and they can be easily integrated into various parts of the application without repeating the redundant codes.
        - ImageDisplay.kt: Handles display of generated image.
        - PromptField.kt: Provides input field for user prompts.
        - SegmentedControl.kt: Manage segmented button logic and display.
        - StatusCheck.kt: Handle display of LAN LLM availability status.
        - StepControl.kt: Handles the display and configuration of denoising steps specified by the user.
    - Screens
        - MainScreen.kt: The main user interface, piecing together all the reusable components.

- Theme: Contains theme resources for styling
- Viewmodels (View Model Layer)
    - DiffusionViewModel.kt: Manages application session states, and life cycle of Stable Diffusion Pipeline.

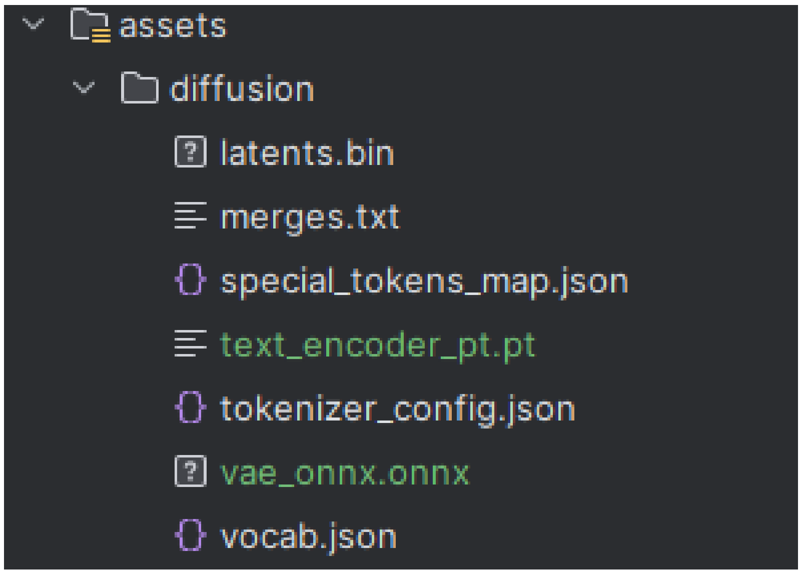
5. MainActivity.kt
    - The entry point of the application.



```
Android ⌄
⌄ app
  › manifests
  ⌄ kotlin+java
    ⌄ com.example.edgediffusionv14
      ⌄ diffusion
        ⌄ models
            Scheduler.kt
            Tokenizer.kt
        ⌄ utils
            FileLoader.kt
            LatentLoader.kt
            LatentNoiseGenerator
          DiffusionPipeline
      ⌄ network
        ⌄ models
            DataModels.kt
          ApiClient
          ApiService
      ⌄ ui
        ⌄ components
            ImageDisplay.kt
            PromptField.kt
            SegmentedControl.kt
            StatusCheck.kt
            StepControl.kt
        ⌄ screens
            MainScreen.kt
        › theme
        ⌄ viewmodels
            DiffusionViewModel
        MainActivity
```

Figure 14: Android App Project Structure

Additionally, Stable Diffusion configuration files and converted models are stored in the asset folder. However, due to the large size of the UNet converted model, it cannot be bundled in the App. Therefore, it is stored on the device's local storage, outside of the App directory.



```
assets
  diffusion
    latents.bin
    merges.txt
    special_tokens_map.json
    text_encoder_pt.pt
    tokenizer_config.json
    vae_onnx.onnx
    vocab.json
```

Figure 15: Configuration and Model files

4.2.2 App Interface Design

As described in figure 16, the main user interface is composed by reusable components and each component's functionality can be outlined as follows:

- LAN LLM Availability status indicator: Indicate whether LLM is available when mobile device is connected to a local area network such as Wi-Fi
- Denoising step adjustment: Communicate with the Stable Diffusion Pipeline to specify the number of time embeddings to be generated and set up configurations
- Seed entry field: Allow user to enter the desired seed. This seed determines the random Gussian noise generated, and it is used for the denoising process. The same seed results in the same random noise generated, enabling reproducible generated images.
- Prompt Field: A text field where user can enter a description of images, he/she wishes to generate

- Prompts enrich/rewrite button: This button is only active when LAN LLM is available. Otherwise, it remains disabled. It sends user-entered prompt from prompt field to LAN LLM and retrieves the rewritten/enhanced prompt. The enhanced prompt is then displayed back to the user in the prompt field.
- Start button: Start the image generation process by communicating with Stable Diffusion image generation pipeline.
- Negative prompt toggle button: Upon toggling, Prompt field switches to negative prompt field. Negative prompt field is used for excluding certain objects, and colours from the generated image as shown in figure 17.
- Progress Status: This box is used for reporting the current status of image generation process as described in figure 18.
- Seed type selector segmented button: This feature enables the user to choose between two seed types. When the option "x86 Seed 0" is selected, the app does not generate random noise but instead utilizes noise generated on a desktop system using PyTorch with seed 0. This is included to address the reproducibility between two different platforms during the early stage of development. The noise generation method differs between desktop and mobile due to the use of PyTorch on desktop and java.util.Random on Android. These different noise generators result in variations in the generated images, even when the same seed is used. The alternative option, "Custom," allows the user to input a custom seed value, with Gaussian noise generated on the mobile device using java.util.Random.

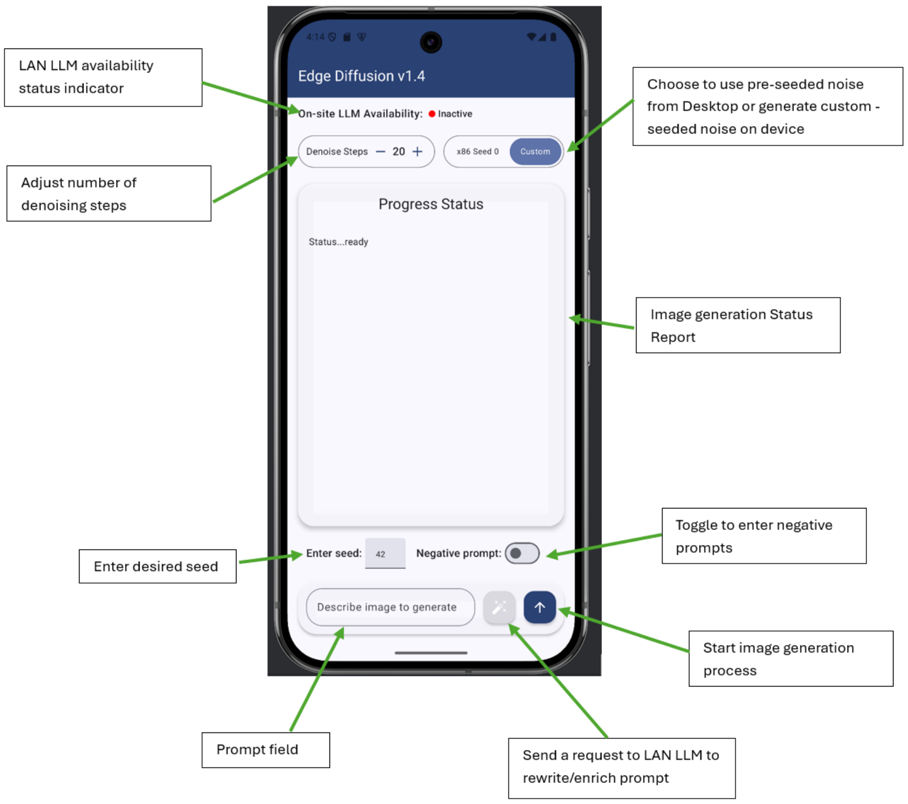

LAN LLM availability status indicator
Adjust number of denoising steps
Choose to use pre-seeded noise from Desktop or generate custom - seeded noise on device
Image generation Status Report
Toggle to enter negative prompts
Enter desired seed
Start image generation process
Prompt field
Send a request to LAN LLM to rewrite/enrich prompt
Edge Diffusion v1.4
On-site LLM Availability: ● Inactive
Denoise Steps — 20 +
x86 Seed 0
Custom
Progress Status
Status...ready
Enter seed: 42
Negative prompt:
Describe image to generate

Figure 16: App UI description

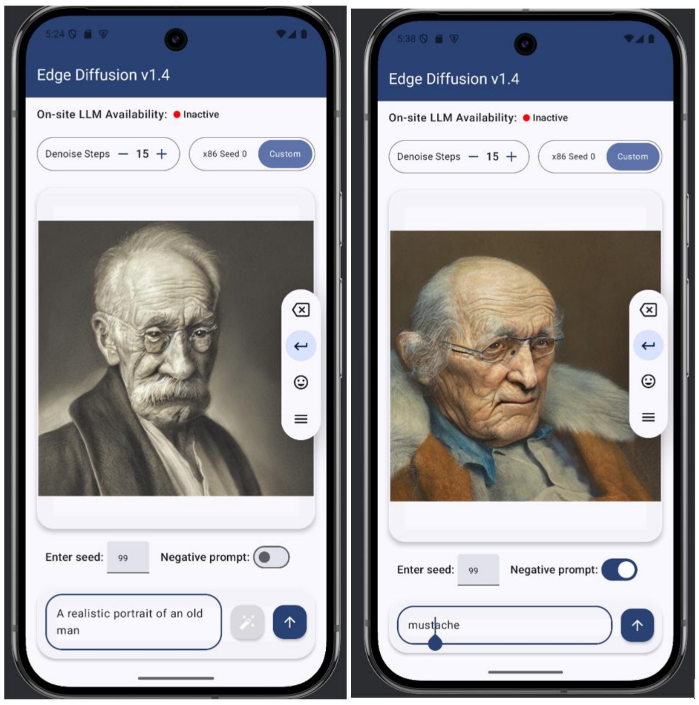
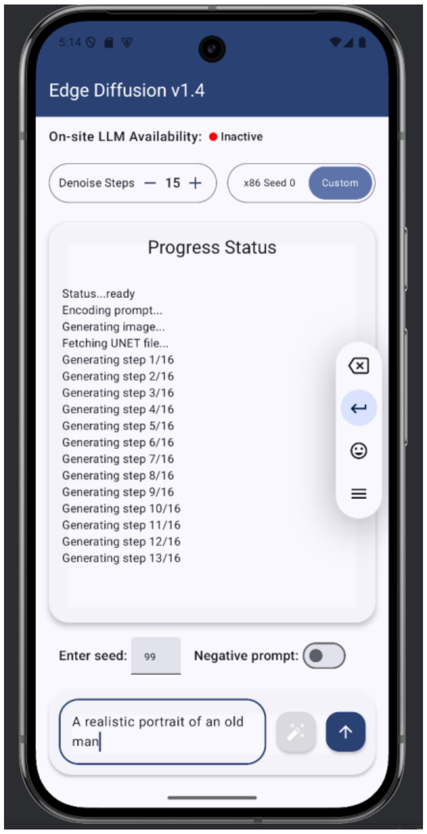


Figure 17: Usage of negative prompt



On-site LLM Availability: ● Inactive

Denoise Steps — 15 +

x86 Seed 0 | Custom

Progress Status

```
Status...ready
Encoding prompt...
Generating image...
Fetching UNET file...
Generating step 1/16
Generating step 2/16
Generating step 3/16
Generating step 4/16
Generating step 5/16
Generating step 6/16
Generating step 7/16
Generating step 8/16
Generating step 9/16
Generating step 10/16
Generating step 11/16
Generating step 12/16
Generating step 13/16
```

Enter seed: 99 Negative prompt:

A realistic portrait of an old man

Figure 18: Status Progress report

4.2.3 Stable Diffusion Pipeline Implementation

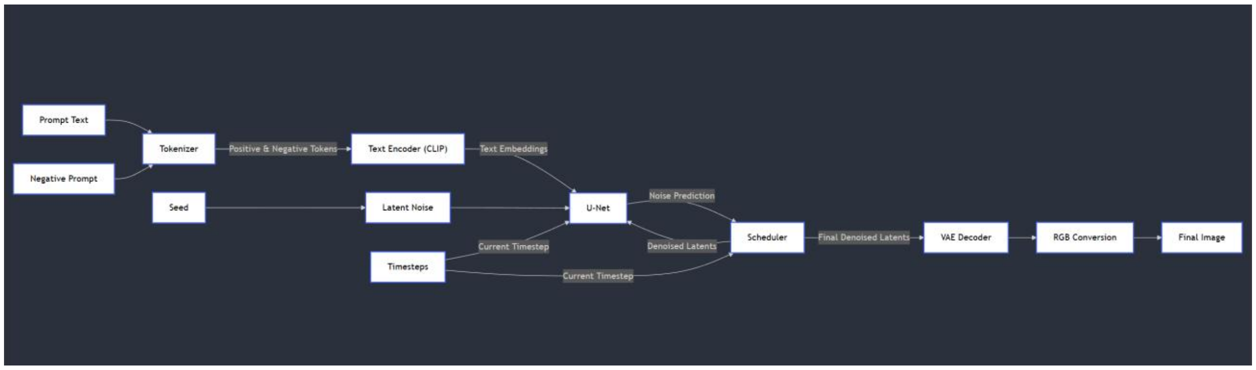


Figure 19: Stable Diffusion Pipeline Implementation in Android App

Stable Diffusion image generation pipeline is made of piecing multiple components together as described in figure 19. The Stable Diffusion image generation pipeline begins with processing prompt and negative prompts. They are tokenized and transformed into Text Embeddings using a Text Encoder (CLIP). After the encoding task, a seed is used to initialize a pseudo random noise in a latent space, which serves as the starting point for image generation. This noise is passed to the denoising loop which begins the image generation process.

At each iteration, the current timestep along with text embeddings are used to guide the denoising process. The U-Net, a convolutional neural network, predicts the noise in the latent space, while the Scheduler reduces the noise based on the U-Net's predictions and the current timestep.

Once the denoising process is completed, the Scheduler outputs the final denoised latents. These latents are then decoded into pixel data using a VAE Decoder. Finally, RGB conversion is applied on the pixel data to produce the final image.

The following sections explain the implementation of key steps and Stable Diffusion components in Android application.

4.2.3.1 Tokenizer and Encoder implementation

The tokenizer employs CLIP BPE, a variant of the Byte-Pair Encoding (BPE) algorithm, rather than pure BPE. BPE operates by iteratively merging the most frequent adjacent character pairs within a predefined vocabulary, thereby generating a compact token representation. The tokenizer relies on a predefined vocabulary file, which is included in the asset folder of the application. This vocabulary file, sourced from CompVis/Stable-Diffusion-v1-4 on Hugging Face, contains token mappings that are specifically aligned with the CLIP text encoder. Additionally, the caching mechanism is also implemented in tokenizer to increase efficiency on mobile devices. Previously tokenized words are cached to avoid redundant computations, ensuring that if the same word appears multiple times, the tokenizer retrieves its precomputed tokenized representation from the cache.

Once both prompt and negative prompts are tokenized, they are merged and passed to the CLIP text encoder model to generate text embedding. This is the encoder model which we converted in section 4.1.2.1 and stored in the asset folder of the app.

4.2.3.2 Scheduler Implementation

The scheduler is implemented using Pseudo Numerical Methods for Diffusion (PNDM). This implementation is adapted from the PNDM Scheduler in the Hugging Face diffusers repository, which was originally written in Python. The code has been ported to Kotlin to suit the requirements of this project, with irrelevant components removed for optimization. Additionally, several adjustments were made to accommodate the Kotlin environment, such as modifying tensor operations and ensuring compatibility with the project's specific architecture.

4.2.3.3 Denoising Loop Implementation

Before the denoising loop begins, time steps and random latent noise are generated. The number of time steps generated is determined by the number of denoising steps set by the user while the random noise is determined by the seed set by the user. The denoising loop iterates through each timestep, progressively refining the noise.

In each iteration, the converted UNet model from section 4.1.2.2 (loaded from device’s local storage) predicts the noise present at the current timestep, and the Scheduler reduces this noise based on the UNet's predictions and the current timestep. This iterative process continues until it reaches the number of denoising steps set by the user. Therefore, it is important to set a denoising steps high enough to allow sufficient iterations for the model to gradually refine the latent representations. Setting denoising steps too high will result in a better picture with less artifact but it will also take longer and more computation to generate. Similarly, denoising steps too low will result in faster generation at the cost of image quality. The experimentation result of denoising step counts, time taken to generate and resulting image is presented in section 4.3.

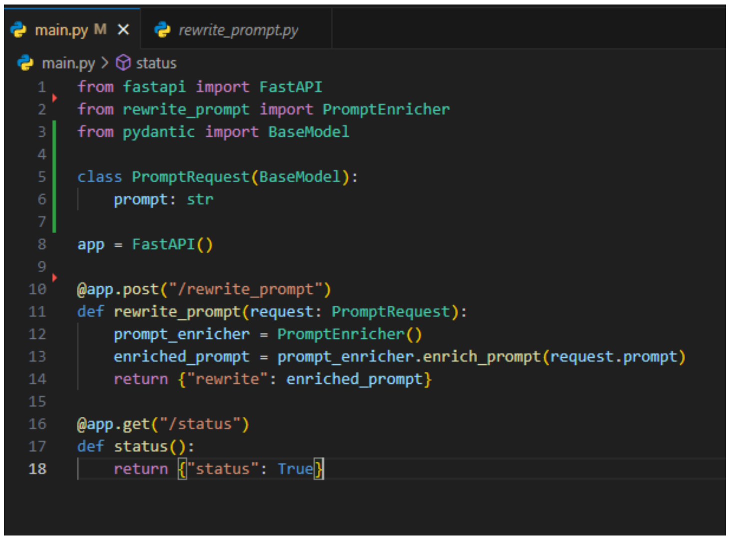
4.2.3.4 VAE Decoding and Image Construction Implementation

At the end of the denoising loop, refined latent is generated. At this point, VAE, a converted model from section 4.1.2.3 stored in the assets folder, is loaded into the App. This model takes in refined latent as an input and run inference on it, producing pixel data at the end.

The produced pixel data is in NCHW format (N = batch size, C = number of channels, H & W = height & width of the image). Therefore, an image conversion function is developed to process the produced NCHW-format data and convert it into a viewable image format. This function extracts the pixel values from each channel, normalizes them, and then assembles the image in a standard RGB format. The final image is then returned as a bitmap, ready for display in the application, marking the end of the Stable Diffusion image generation pipeline.

4.2.3 Local LLM Server and Prompt Enhancement Implementation

FastAPI framework is used to provide RESTful API service to the mobile app. The server has two endpoints to ask for prompt rewriting service and to check for status of the server.

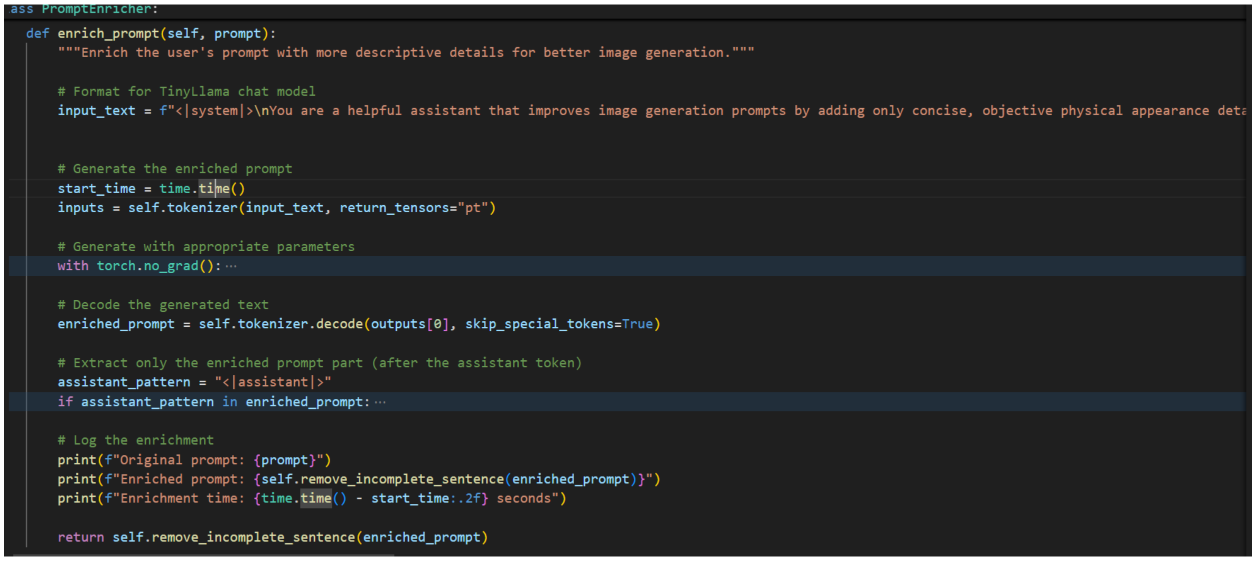


```python
from fastapi import FastAPI
from rewrite_prompt import PromptEnricher
from pydantic import BaseModel

class PromptRequest(BaseModel):
    prompt: str

app = FastAPI()

@app.post("/rewrite_prompt")
def rewrite_prompt(request: PromptRequest):
    prompt_enricher = PromptEnricher()
    enriched_prompt = prompt_enricher.enrich_prompt(request.prompt)
    return {"rewrite": enriched_prompt}

@app.get("/status")
def status():
    return {"status": True}
```

Figure 20: LLM Server Endpoints

A lightweight LLM, TinyLlama/TinyLlama-1.1B-Chat-v1.0 is employed to perform the prompt re-writing and enrichment task. It is instructed with system prompt to rewrite and enrich the prompt with additional details and information. The full system instruction is as follows:

Full system prompt : f"<|system|>\nYou are a helpful assistant that improves image generation prompts by adding only concise, objective physical appearance details to the objects mentioned in the prompt. Describe only factual attributes such as size, shape, color, texture, and arrangement. Do not introduce any new objects, background elements, or narrative context. \n<|user|>\nEnrich this prompt with visual details: {prompt}\n<|assistant|>"

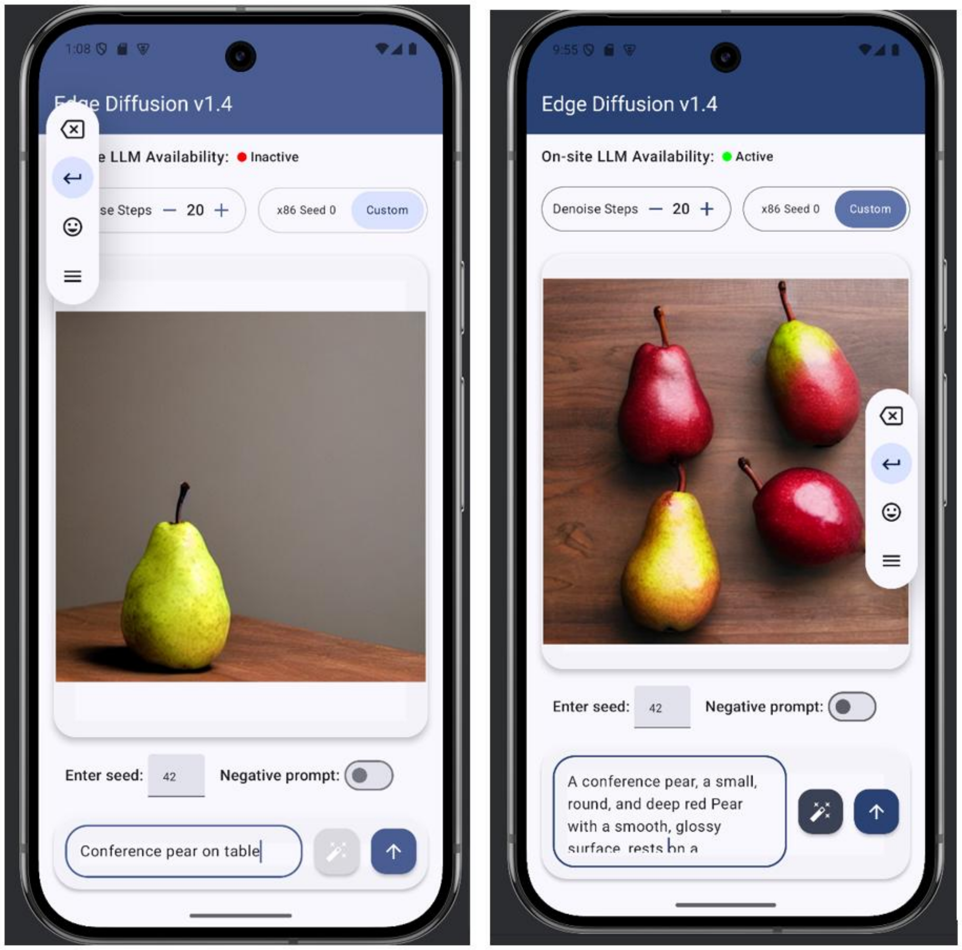


```
ass PromptEnricher:
    def enrich_prompt(self, prompt):
        """Enrich the user's prompt with more descriptive details for better image generation."""

        # Format for TinyLlama chat model
        input_text = f"<|system|>\nYou are a helpful assistant that improves image generation prompts by adding only concise, objective physical appearance deta


        # Generate the enriched prompt
        start_time = time.time()
        inputs = self.tokenizer(input_text, return_tensors="pt")

        # Generate with appropriate parameters
        with torch.no_grad():···

        # Decode the generated text
        enriched_prompt = self.tokenizer.decode(outputs[0], skip_special_tokens=True)

        # Extract only the enriched prompt part (after the assistant token)
        assistant_pattern = "<|assistant|>"
        if assistant_pattern in enriched_prompt:···

        # Log the enrichment
        print(f"Original prompt: {prompt}")
        print(f"Enriched prompt: {self.remove_incomplete_sentence(enriched_prompt)}")
        print(f"Enrichment time: {time.time() - start_time:.2f} seconds")

        return self.remove_incomplete_sentence(enriched_prompt)
```

Figure 21: LLM instructed to enrich prompt

4.2.3.1 Effects of Prompt Enrichment on Generated Images

The rewriting of prompts with additional information by LLM before passing it to image generation pipeline does not always result in the desired improvement as shown in figure 22 . While the enriched prompts can enhance the accuracy and aesthetic quality of the generated images by providing more information, in some cases, they can introduce ambiguity or overly specific details that may lead to artifacts or distortions in the output. Although, in some cases, it works well because the additional context helps the model generate a more accurate representation of the object or scene, improving visual coherence and realism as shown in figure 23.



e LLM Availability: Inactive

se Steps — 20 +

x86 Seed 0

Enter seed: 42

Negative prompt:

Conference pear on table

Edge Diffusion v1.4

On-site LLM Availability: Active

Denoise Steps — 20 +

x86 Seed 0

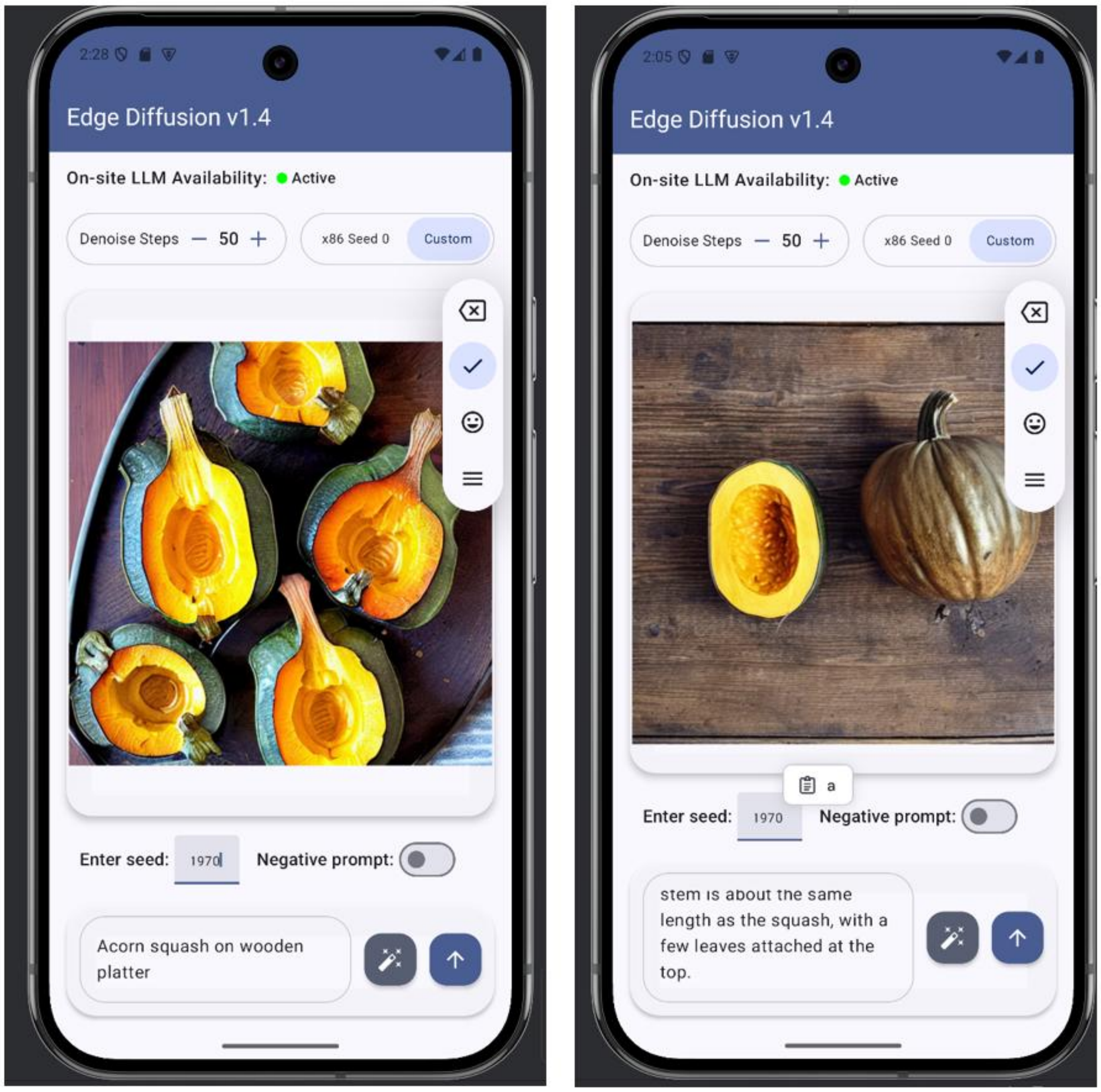
Enter seed: 42

Negative prompt:

A conference pear, a small, round, and deep red Pear with a smooth, glossy surface, rests on a

Figure 22: Conference Pear. Normal prompt vs Enhanced Prompt Effect

- Normal Prompt (Left): Conference pear on table
- Enhanced Prompt by LLM (Right): A conference pear, a small, round, and deep red Pear with a smooth, glossy surface, rests on a rectangular wooden table. The conference pear is arranged in a symmetrical pattern, with the stem pointing towards the center of the table. The conference pear is surrounded by a cluster of other pears, each with a similar color and texture.


2:28

Edge Diffusion v1.4

On-site LLM Availability: Active

Denoise Steps — 50 +

x86 Seed 0 | Custom

Enter seed: 1970

Negative prompt:

Acorn squash on wooden platter

2:05

Edge Diffusion v1.4

On-site LLM Availability: Active

Denoise Steps — 50 +

x86 Seed 0 | Custom

Enter seed: 1970

Negative prompt:

stem is about the same length as the squash, with a few leaves attached at the top.

Figure 23: Acorn Squash. Normal prompt vs Enhanced Prompt Effect

- Normal Prompt (Left) : Acorn squash on wooden platter
- Enhanced Prompt by LLM (Right): A small, round acorn squash sits on a wooden platter, its skin smooth and shiny, with a few small, brown spots dotting its surface. The squash is about the size of a small pumpkin, with a slightly curved shape and a slightly pointed end. The stem is about the same length as the squash, with a few leaves attached at the top.

4.3 Android on-device Inference Experiment Data

In order to collect the relationship between denoising steps and the time taken to generate an image, 10 runs of each denoising step count are conducted while keeping seed and prompt the same for each run. As described in table 2 and figure 24, the relationship is observed to be linear. This is expected because denoising step involves a fixed number of computations, including noise prediction by the UNet and noise reduction by the Scheduler. Since these operations are repeated for each step, the total processing time increases proportionally with the number of denoising steps. Additionally, the linear relationship suggests that there are no significant overheads or bottlenecks affecting performance as the step count increases.

Denoising Steps	**Time Taken (minutes)**
10	4.5
20	7.42
50	16.7

Table 2: Average time over 10 runs

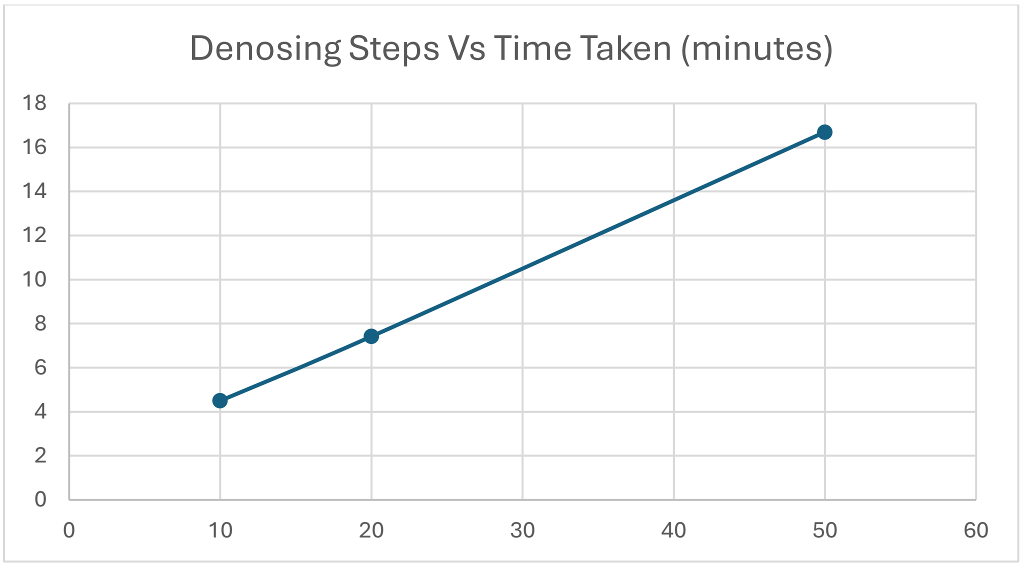
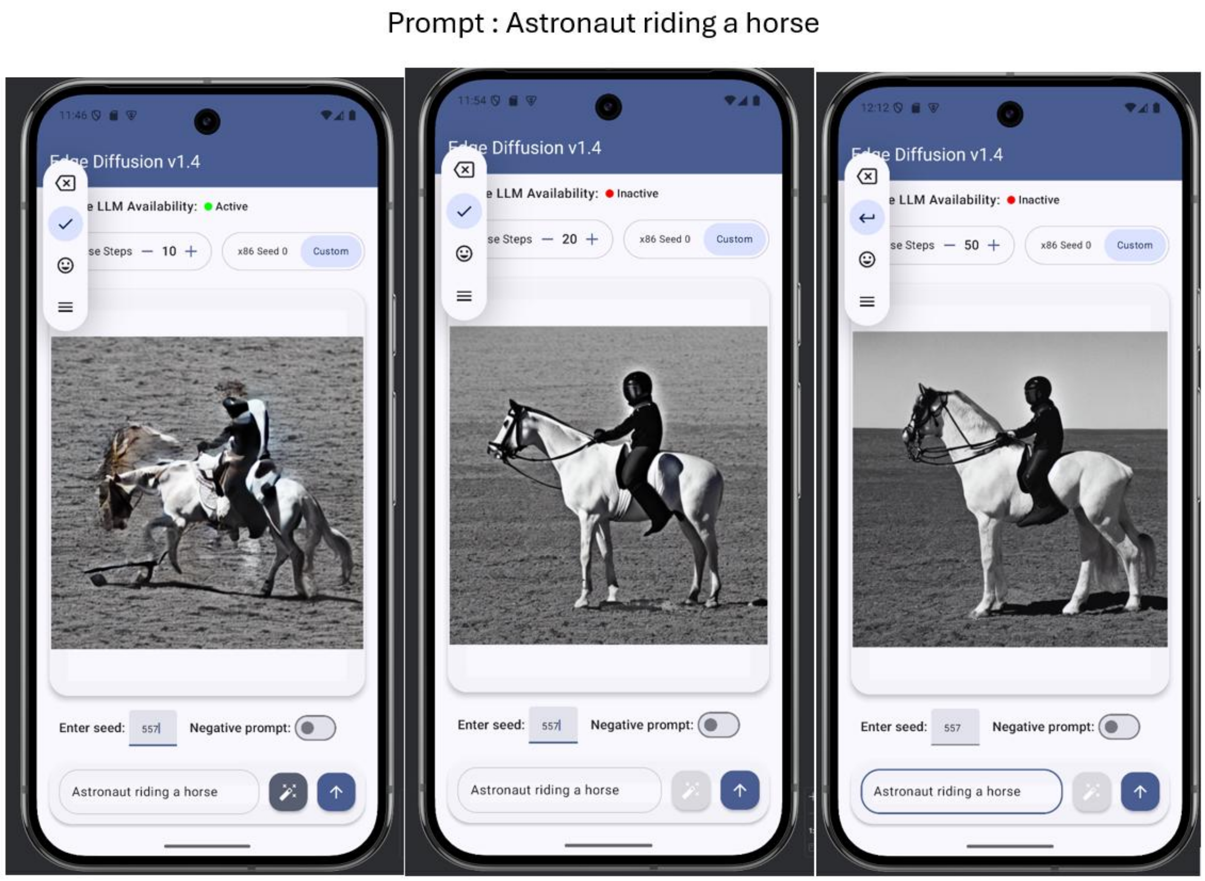


Figure 24: Plot of denoising step and time taken relationship

4.3.1 Sample Generated Images



Edge Diffusion v1.4
LLM Availability: Active
se Steps — 10 +
x86 Seed 0
Custom
Enter seed: 557
Negative prompt:
Astronaut riding a horse

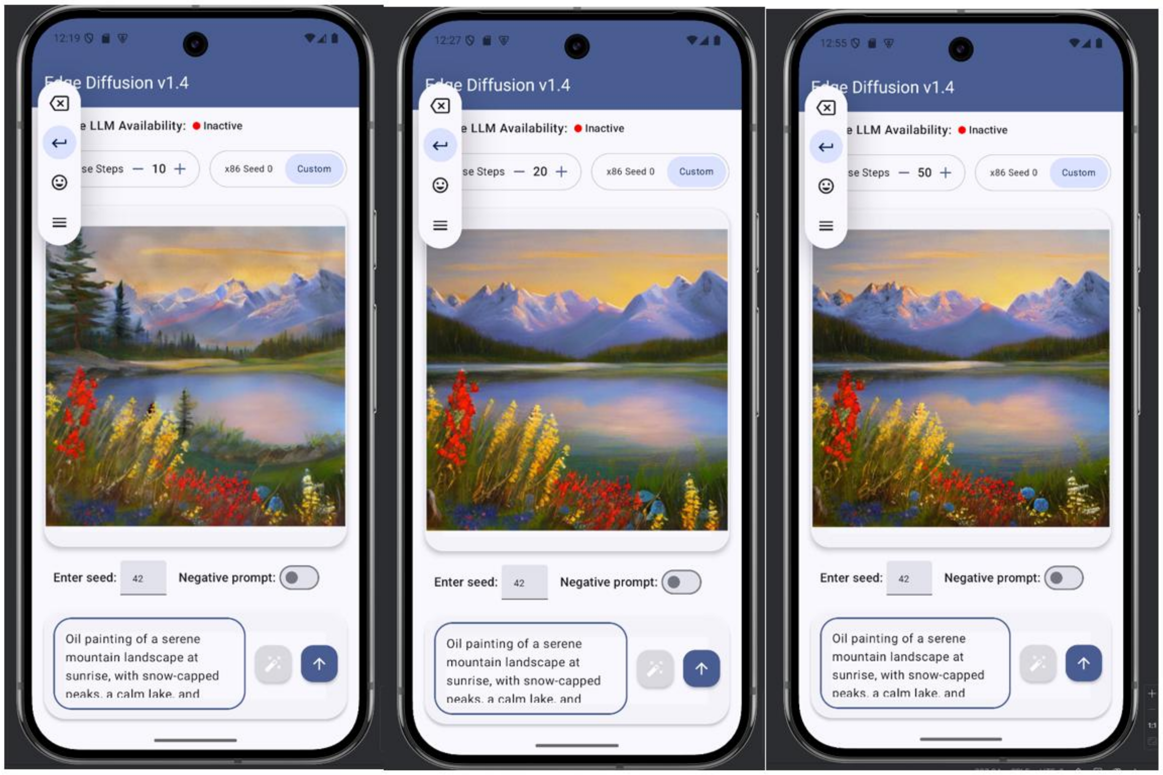
Edge Diffusion v1.4
LLM Availability: Inactive
se Steps — 20 +
x86 Seed 0
Custom
Enter seed: 557
Negative prompt:
Astronaut riding a horse

Edge Diffusion v1.4
LLM Availability: Inactive
se Steps — 50 +
x86 Seed 0
Custom
Enter seed: 557
Negative prompt:
Astronaut riding a horse

Figure 25: Astronaut riding a horse. Increasing denoise steps from left to right

As figure 25 illustrates, it can be observed that higher denoise steps lead to higher quality and less noisy image. The higher denoising steps results in smoother transitions, sharper outlines and more defined structures at the cost of higher computational resources and processing time.

Prompt: Oil painting of a serene mountain landscape at sunrise, with snow-capped peaks, a calm lake, and wildflowers in the foreground.



e LLM Availability: Inactive

se Steps — 10 +

se Steps — 20 +

se Steps — 50 +

Enter seed: 42

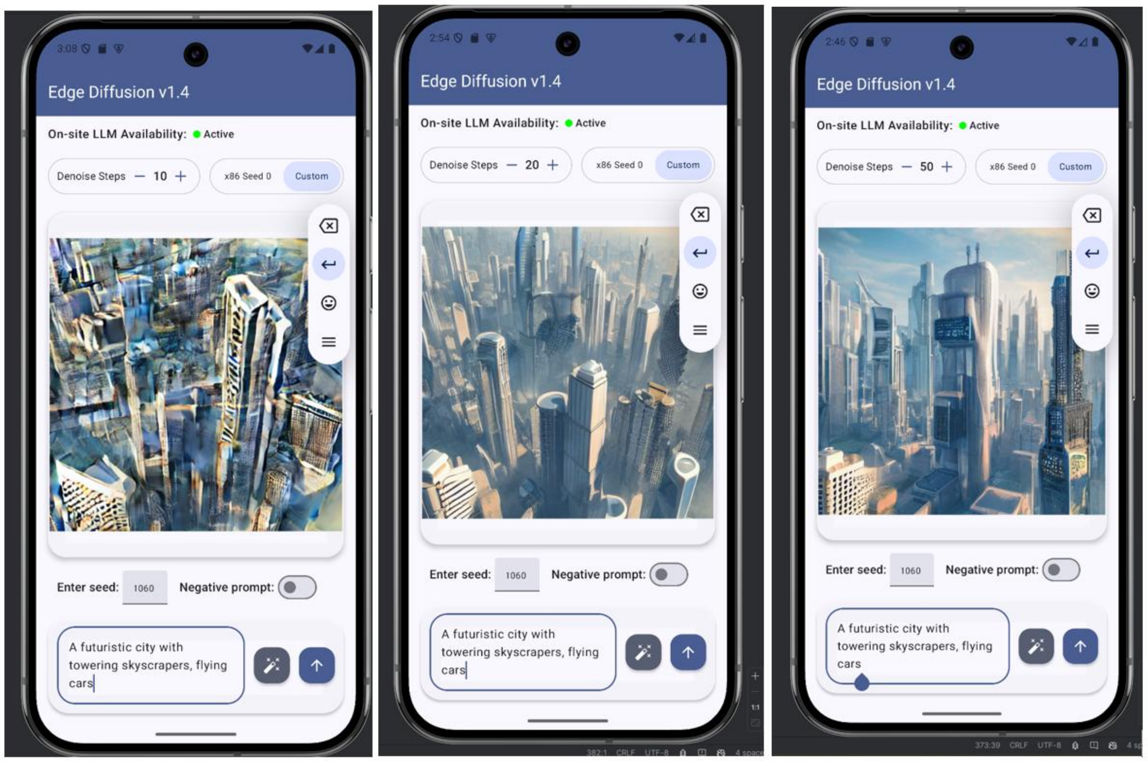
Negative prompt:

Oil painting of a serene mountain landscape at sunrise, with snow-capped peaks, a calm lake, and

Figure 26: Oil painting of nature. Increasing denoise steps from left to right

In figure 26, it can be observed that the difference between the three generated picture is minimal in terms of quality and noise presence. This is likely due to the nature of the prompt, which specifies an oil painting style. Oil paintings, by design, often have a degree of abstraction and texture that does not require fine details to convey their aesthetic. As a result, a coherent and less noisy picture is achieved at low denoising steps.

Prompt: A futuristic city with towering skyscrapers, flying cars


Edge Diffusion v1.4

On-site LLM Availability: ● Active

Denoise Steps — 10 +

x86 Seed 0 Custom

Enter seed: 1060 Negative prompt:

A futuristic city with towering skyscrapers, flying cars

Edge Diffusion v1.4

On-site LLM Availability: ● Active

Denoise Steps — 20 +

x86 Seed 0 Custom

Enter seed: 1060 Negative prompt:

A futuristic city with towering skyscrapers, flying cars

Edge Diffusion v1.4

On-site LLM Availability: ● Active

Denoise Steps — 50 +

x86 Seed 0 Custom

Enter seed: 1060 Negative prompt:

A futuristic city with towering skyscrapers, flying cars

Figure 27: Futuristic City. Increasing denoise steps from left to right

It can be observed that figure 27 requires more denoising steps to produce a less noisy image. This is likely because the model needs to generate more intricate details, such as lighting effects, textures, and reflections, demanding additional refinement to ensure clarity and prevent artifacts.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This project has successfully demonstrated the feasibility of local, on-device image generation on resource constrained mobile devices , with a particular focus on Android native code implementation using Kotlin. Through the exploration and technical analysis of various text-to-image generation models, optimization techniques, and core components of Stable Diffusion, this project has managed to convert these models into mobile-compatible formats, enabling efficient deployment on Android. Stable Diffusion image generation pipeline was also successfully implemented in Android native code, Kotlin. Additionally, utilization of large language models to improve the accuracy and aesthetic quality of generated images is also implemented and experimented.

This project also highlighted the difficulties faced due to the early stage of Edge AI development ecosystem. Despite the challenges, the successful implementation of this project demonstrates the potential for further advancement in the field of Edge AI and inference on resource constrained devices. Ultimately, this work lays the groundwork for further exploration and improvement in the field, particularly regarding optimizing generative models for mobile devices and the continued evolution of on-device AI capabilities.

5.2 Future Work

This section discusses the future work and explorations that can be undertaken to further improve and expand upon this project.

5.2.1 Model Optimization for Mobile Devices

In this project, optimization techniques such as pruning, quantization and knowledge distillation were not applied due to the limited project timeline. Future work could focus on incorporating these techniques to further optimize the Stable Diffusion model for more efficient inference on resource-constrained mobile devices.

5.2.2 Training LoRA

In this project, the use of LLM to improve accuracy of generated image is implemented and experimented. However, this approach does not always yield the desired results, necessitating additional computation on LAN server. This issue could be mitigated through the use of smaller, fine-tuned models tailored for specific use cases. One such fine-tuning technique is LoRA (Low-Rank Adaptation), which enables finetuning with relatively low computational overhead. Therefore, it can be performed on consumer-grade level GPU, making it a viable option for enhancing model performance without requiring high-end hardware.

5.2.3 GPU Acceleration for Inference Taks

Due to the lack of a physical Android device, the development for this project was primarily conducted on an emulator. Currently, major frameworks such as TF Lite, PyTorch mobile and ONNX mobile support GPU acceleration only on real hardware, restricting the exploration and implementation of GPU-accelerated inferencing in this project.

Future work can be carried out in deploying the model on actual Android hardware to take advantage of GPU acceleration. By utilizing GPU acceleration, the inference speed could be significantly improved, reduce computational load on the CPU and enable more efficient on-device generation.

5.2.4 Experimenting Prompt Enrichment with Larger LLMs

In this project, a small light weight LLM is used for prompt enrichment task. Despite its low requirement for computational and memory resources, the LLM can sometimes hallucinate or return inaccurate rewritten prompt. This is likely due to model's limited parameter and using a larger model would likely yield more accurate and contextually enriched outputs. However, hardware constraints prevented the simultaneous execution of both a larger LLM and the Android emulator, limiting the choice of models for this task.

Future work could explore integrating more powerful LLMs by leveraging model deployment on dedicated hardware, such as secondary PC or server on premise, to enhance prompt enrichment accuracy and quality.

References

[1] Y. Lu, “Artificial intelligence: a survey on evolution, models, applications and future trends,” *Journal of Management Analytics*, vol. 6, no. 1, pp. 1–29, 2019.

[2] M. Cheng, “The creativity of artificial intelligence in art,” in *Proceedings*, 2022, vol. 81, no. 1, p. 110.

[3] L. Ding, C. Lawson, and P. Shapira, “Rise of Generative Artificial Intelligence in Science,” *arXiv preprint arXiv:2412.20960*, 2024.

[4] Z. Epstein *et al.*, “Art and the science of generative AI,” *Science*, vol. 380, no. 6650, pp. 1110–1111, 2023.

[5] D. Yim, J. Khuntia, V. Parameswaran, A. Meyers, and others, “Preliminary evidence of the use of generative AI in health care clinical services: systematic narrative review,” *JMIR Medical Informatics*, vol. 12, no. 1, p. e52073, 2024.

[6] T. Mikolov, W. Yih, and G. Zweig, “Linguistic regularities in continuous space word representations,” in *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, 2013, pp. 746–751.

[7] I. Goodfellow *et al.*, “Generative adversarial networks,” *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[8] M. Mirza and S. Osindero, “Conditional generative adversarial nets,” *arXiv preprint arXiv:1411.1784*, 2014.

[9] T. Karras, S. Laine, and T. Aila, “A style-based generator architecture for generative adversarial networks,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4401–4410.

[10] A. Gillioz, J. Casas, E. Mugellini, and O. Abou Khaled, “Overview of the Transformer-based Models for NLP Tasks,” in *2020 15th Conference on computer science and information systems (FedCSIS)*, 2020, pp. 179–183.

[11] A. Ramesh *et al.*, “Zero-shot text-to-image generation,” in *International conference on machine learning*, 2021, pp. 8821–8831.

[12] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10684–10695.

[13] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[14] A. Radford *et al.*, “Learning transferable visual models from natural language supervision,” in *International conference on machine learning*, 2021, pp. 8748–8763.

[15] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, 2015, pp. 234–241.

[16] D. P. Kingma, M. Welling, and others, *Auto-encoding variational bayes*. Banff, Canada, 2013.

[17] Y. C. Shin and C. Xu, *Intelligent systems: modeling, optimization, and control*. CRC press, 2017.

[18] Y. LeCun, J. Denker, and S. Solla, “Optimal brain damage,” *Advances in neural information processing systems*, vol. 2, 1989.

[19] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.

[20] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv preprint arXiv:1503.02531*, 2015.

[21] Y. Zhao, Y. Xu, Z. Xiao, H. Jia, and T. Hou, “Mobilediffusion: Instant text-to-image generation on mobile devices,” in *European Conference on Computer Vision*, 2024, pp. 225–242.

[22] Google AI, "Mediapipe Solutions Guide | Google AI Edge | Google AI for developers," *Google AI Edge*. [Online]. Available: https://ai.google.dev/edge/mediapipe/solutions/guide. [Accessed: Mar. 21, 2025].

[23] “Docs,” Termux Wiki. Accessed: Mar. 23, 2025. [Online]. Available: https://wiki.termux.com/wiki/Main_Page

[24] “Qualcomm Documentation,” AI Engine Direct SDK. Accessed: Mar. 23, 2025. [Online]. Available: https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-50/introduction.html

[25] A. Neumann, N. Laranjeiro, and J. Bernardino, “An analysis of public REST web service APIs,” *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, 2018.

[26] L. Ardito, R. Coppola, G. Malnati, and M. Torchiano, “Effectiveness of Kotlin vs. Java in android app development tasks,” *Information and Software Technology*, vol. 127, p. 106374, 2020.

[27] “Kotlin and Android,” *Android Developers*. Accessed: Mar. 23, 2025. [Online]. Available: https://developer.android.com/kotlin

[28] “Why Compose,” *Android Developers*. Accessed: Mar. 23, 2025. [Online]. Available: https://developer.android.com/develop/ui/compose/why-adopt#accelerate-development

[29] “torch.jit.trace ,” PyTorch 2.6 documentation. Accessed: Mar. 23, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.jit.trace.html