MoRAGBench: A Benchmarking Framework for RAG Pipelines on Mobile Devices

by

Huzaifa Shaaban Kabakibo

Submitted to the Computer Networks Group
in partial fulfillment of the requirements for the degree of

Master of Sciences

at

Paderborn University

March 19, 2026

First reviewer: Prof. Dr. Lin Wang
Second reviewer: Prof. Dr. Marco Platzner

Computer Networks Group
Department of Computer Science
Paderborn University

MoRAGBench: A Benchmarking Framework for RAG Pipelines on Mobile Devices

Huzaifa Shaaban Kabakibo

The research reported in this thesis has been carried out in the Computer Networks group at the Department of Computer Science of Paderborn University.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

I certify that I have written this thesis without outside help and without using sources other than those specified and that the thesis has not been submitted in the same or a similar form to any other examination authority and has not been accepted by them as part of an examination. All statements that have been adopted verbatim or analogously are labeled as such.

Ort, Datum	Unterschrift
Place, Date	*Signature*

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Prof. Dr. Lin Wang, for his continuous guidance, support, and encouragement throughout this journey. His insights and mentorship have been invaluable to the completion of this work.

I am also grateful to Prof. Dr. Marco Platzner for reviewing this thesis and for his valuable feedback.

My deepest thanks go to my parents, whose unwavering support and sacrifices made this achievement possible. Their encouragement and belief in me have always been my greatest motivation.

Finally, I would like to thank my brother, Walid, for his constant support, and for helping with the daily cooking, which made this journey much easier and more enjoyable.

Abstract

Retrieval Augmented Generation (RAG) has emerged as an effective approach for improving the factual grounding and contextual relevance of Large Language Models (LLMs) by combining neural generation with external knowledge retrieval. While most existing RAG systems are designed for server or cloud environments, executing complete pipelines directly on mobile devices introduces significant challenges due to limited computational resources, memory constraints, hardware heterogeneity, and immature software stacks and optimizations. Despite growing interest in on-device intelligence, there remains limited systematic understanding of how individual RAG components behave and contribute to the overall performance under realistic mobile conditions.

This thesis presents MoRAGBench, a modular benchmarking framework for evaluating RAG pipelines on Android smartphones. The framework enables configurable experimentation across all stages of the pipeline, including document chunking, embedding generation, indexing and retrieval, augmentation, and LLM inference. To provide comprehensive analysis, MoRAGBench introduces two complementary evaluation modes: an approximate nearest neighbor benchmark that isolates retrieval performance, and an end-to-end task benchmark that measures downstream question answering quality and overall system efficiency.

Extensive experiments conducted on a modern smartphone reveal fundamental trade-offs between retrieval accuracy, latency, throughput, and memory consumption. The results show that efficient on-device RAG deployment is primarily a systems-level challenge in which performance improvement emerges from the interaction between pipeline components and hardware execution backends rather than from individual model improvements alone. The study further highlights limitations of current mobile inference-acceleration frameworks and demonstrates the importance of balanced RAG pipeline configurations and approximate similarity search methods for achieving practical performance.

By enabling systematic, reproducible evaluation of RAG systems under mobile constraints, MoRAGBench provides practical insights and a foundation for future research toward efficient, privacy-preserving, fully on-device intelligent assistants. MoRAGBench is fully open-sourced at https://github.com/upb-cn/MoRAGBench.

Contents

List of Figures

List of Tables

1

Introduction

Large Language Models (LLMs) have rapidly become a foundational technology for modern intelligent systems, enabling advanced capabilities in natural language understanding, reasoning, and generation [8, 2, 6, 30]. These models are trained on massive corpora of data [24] and require significant computational resources for both training and periodic re-training. However, the knowledge embedded within an LLM is inherently static and bounded by the time at which the training data was collected [4, 51]. Updating this knowledge through re-training or fine-tuning is prohibitively expensive in terms of data, computation, and time [18]. Retrieval-Augmented Generation (RAG) [26] has emerged as an effective solution to this limitation by combining LLMs with external knowledge retrieval mechanisms [19, 17]. Instead of relying solely on parametric memory, RAG enables models to dynamically retrieve relevant information from external data sources at inference time, thereby providing up-to-date, context-specific, and personalized responses without requiring model re-training.

At the same time, modern smartphones have evolved into highly capable computing platforms equipped with powerful CPUs, GPUs, NPUs, and large local storage capacities [48, 9]. These devices continuously accumulate vast amounts of user-specific data such as photos, documents, messages, browsing history, and application data. This creates an opportunity for highly personalized, on-device intelligent assistants that can answer queries such as "Show me pictures from my last visit to Japan" or "Summarize the notes I took during last week's meetings". Traditional approaches for such tasks rely heavily on keyword-based search and manually crafted indexing, which often fail to capture semantic relationships within the data. RAG, through similarity search over vector embeddings [22, 29, 21], offers a principled way to perform semantic retrieval over personal data, enabling natural, conversational interaction with information stored locally on the device.

However, deploying RAG pipelines on mobile devices presents significant challenges. Unlike server-class environments, mobile platforms operate under strict constraints on memory, computation, power consumption, and thermal limits. Additionally, support for vector databases and efficient execution of modern machine learning components, such as embedding models and LLM inference, is still fragmented across mobile hardware and software ecosystems, with limited availability of optimized GPU/NPU kernels and heterogeneous acceleration frameworks [48, 55, 20]. These constraints make it non-trivial to directly transfer RAG system designs and optimizations from cloud or desktop environments to mobile platforms.

Despite the growing interest in on-device intelligence, there is currently limited systematic understanding of how individual components of a RAG pipeline behave under mobile con-

straints [36]. It is unclear which parts of the pipeline (e.g., embedding generation, vector indexing, similarity search, LLM inference) become dominant bottlenecks under different hardware configurations and software stacks. Furthermore, design choices such as model size, embedding dimensionality, indexing strategy, and hardware acceleration backend can interact in complex ways, making performance highly sensitive to configuration. Existing evaluations of RAG systems largely focus on server-scale deployments [14, 7], leaving a gap in knowledge about their behavior in resource-constrained mobile environments.

These limitations motivate the following research question: *How do RAG pipeline components interact with each other under mobile constraints, and what bottlenecks and trade-offs emerge?* Answering this question requires a systematic investigation of the behavior of individual RAG pipeline components under realistic mobile deployment conditions, as well as an understanding of how different design choices and hardware acceleration strategies influence overall system performance.

In this thesis, we present MoRAGBench, a modular benchmarking framework designed to systematically evaluate RAG pipelines on mobile devices. The framework enables configurable experimentation across all major pipeline components, including text chunking, embedding generation, indexing and retrieval, augmentation, and LLM inference, while executing entirely on-device. By supporting both component-level benchmarking of approximate nearest neighbor retrieval and end-to-end evaluation of downstream question answering tasks, MoRAGBench provides a comprehensive view of performance trade-offs under realistic mobile constraints. Through extensive experiments on a modern smartphone, this work analyzes how design choices, model configurations, and hardware backends influence latency, throughput, memory usage, and task accuracy. The insights obtained contribute to a deeper understanding of efficient on-device RAG deployment and highlight the importance of holistic system-level optimization.

The remainder of this thesis is structured as follows. Chapter 2 introduces the background concepts underlying similarity search and RAG pipelines. Chapter 3 reviews related work on mobile RAG and existing benchmarking approaches. Chapter 4 presents the design and implementation of MoRAGBench. Chapter 5 validates the framework through extensive benchmarking experiments on mobile hardware. Chapter 6 discusses the broader implications of the findings and outlines future research directions. Finally, Chapter 7 concludes the thesis.

2

Background

This chapter provides the conceptual background necessary for the remainder of this thesis. First, in Section 2.1, we introduce RAG pipelines and describe their main components, including indexing, retrieval, augmentation, and generation. Understanding the structure of these pipelines is essential for analyzing how individual components contribute to overall system performance. Second, we present the fundamentals of similarity search in high-dimensional vector spaces in Section 2.2, which forms the core mechanism underlying the retrieval stage of RAG. Since similarity search algorithms and indexing strategies strongly influence latency, memory usage, and scalability, they play a central role in evaluating RAG systems under the resource constraints of mobile devices.

2.1 RAG Pipelines

RAG pipelines combine neural retrieval mechanisms with LLMs to incorporate external knowledge during inference time [26, 19, 17]. Instead of relying solely on the parametric memory of a language model, RAG systems retrieve relevant information from a knowledge source and use it to augment the model's input, enabling more accurate, up-to-date, and context-aware responses.

As illustrated in Figure 2.1, a typical RAG pipeline consists of two main stages: an offline indexing stage (steps A-C) and an online query stage (steps 1-5).

2.1.1 Offline Indexing Stage

The pipeline begins with raw data sources such as documents, images, audio recordings, or videos. Since these data sources are often large and unstructured, they are first processed through a chunking step (step A), where content is divided into smaller, manageable segments. Chunking improves retrieval granularity and ensures that retrieved context is relevant and concise [27].

Each chunk is then passed through an embedding model (step B), which converts the chunk into a dense vector representation in a high dimensional space, capturing its semantic meaning. These embeddings are then stored in a knowledge source (step C), typically a vector database or index that supports efficient similarity search techniques [22].

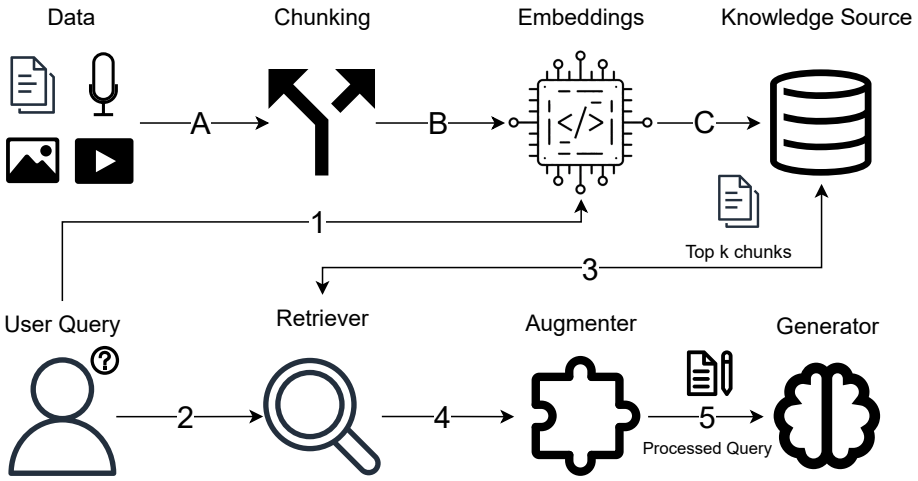


Figure 2.1: General workflow of a RAG pipeline.

2.1.2 Online Query Stage

When a user submits a query, the query is first transformed into an embedding vector using the same embedding model used in the offline stage (step 1). This query embedding is passed to a retriever (step 2), which performs similarity search over the indexed embeddings (step 3) and returns the top-k most relevant chunks. The parameter k determines how many retrieved chunks are returned and provided to the downstream pipeline; in practice, it is treated as a hyperparameter that is chosen based on empirical performance trade-offs between retrieval recall, prompt length, and computational cost. Later in Chapter 5, we analyze the trade-offs of this parameter.

The retrieved chunks are then passed to an augmenter (step 4). In its simplest form, the augmenter may directly append the retrieved chunks to the original user query to form an extended prompt [1]. In more advanced systems [43], this component may perform additional processing such as filtering irrelevant results, re-ranking retrieved chunks, de-duplication, or prompt templating to optimize the context provided to the language model.

Finally, the processed prompt is fed into the generator (step 5), typically an LLM, which produces the final response conditioned on both the query and the retrieved information [27, 19]. This modular pipeline allows RAG systems to dynamically incorporate external knowledge at inference time without requiring model re-training, while leveraging efficient similarity search techniques to scale retrieval to large knowledge sources.

2.2 Similarity Search in Vector Spaces

Similarity search is one of the core components of RAG pipelines (in particular, the retriever). The main task of similarity search is to retrieve data points from a collection that are most similar to a given query according to a defined distance metric, typically in a high-dimensional vector space. In modern machine learning systems, data such as text, images, audio, and documents are transformed into dense vector representations (embeddings) using neural models [10, 39], where semantic similarity between items corresponds to geometric proximity in the embed-

ding space. The resulting embedding vectors are typically stored in a data structure known as a vector index, which is designed to support efficient nearest neighbor retrieval over large collections of high-dimensional vectors. Given a query vector, similarity search algorithms traverse this index to retrieve the nearest vectors according to a chosen distance metric, such as cosine similarity or Euclidean distance.This paradigm enables semantic retrieval beyond traditional keyword-based approaches by operating directly on learned representations rather than symbolic tokens [41, 22].

Fundamentally, similarity search methods can be categorized into Exact Nearest Neighbor (ENN) search and Approximate Nearest Neighbor (ANN) search. ENN methods, often referred to as *Flat* search, compute the distance between the query vector and every vector in the database. While this approach guarantees perfect accuracy, it becomes computationally infeasible as the dataset grows due to its linear time complexity [22].

To address this limitation, ANN techniques trade a small amount of accuracy for significant improvements in speed and memory efficiency. Modern similarity search systems and vector databases predominantly rely on ANN methods due to their favorable trade-offs between accuracy, computation, and memory usage, making them especially suitable for deployment in resource-constrained environments such as mobile devices [36, 5, 49, 46]. Among the most popular ANN techniques used in practice are Inverted File Index (IVF) [42], Hierarchical Navigable Small World graphs (HNSW) [29], and Product Quantization (PQ) [21].

3

Related Work

While RAG has been widely studied in server and cloud environments, where abundant computational resources enable large-scale retrieval and model inference [14, 7], relatively little work has explored the deployment of RAG systems on resource-constrained mobile devices. Only recently have a small number of studies begun to investigate how retrieval mechanisms and language model inference can be adapted to edge and mobile platforms. This section reviews four representative efforts, LEANN [49], MLLM [52], MobileRAG [36], and RAGPerf [28], that form the foundation and motivation for this research.

3.1 LEANN

LEANN [49] presents one of the first systematic studies on performing low-latency, high-accuracy similarity search over personal data with minimal storage overhead on edge devices. The proposed system introduces a compact graph-based ANN index that prunes redundant metadata by prioritizing the preservation of high-degree nodes and avoids storing embeddings by recomputing them on the fly. To mitigate the additional cost of recomputation, LEANN employs a two-level search mechanism with dynamic batching to balance latency and memory usage. While this approach effectively reduces storage requirements and improves retrieval efficiency, the work primarily focuses on optimizing the similarity search component in isolation rather than evaluating the behavior of a complete RAG pipeline that also includes embedding generation, augmentation, and language model inference. Furthermore, its evaluation primarily targets devices such as NVIDIA A10 GPUs and M1-based Mac laptops. Consequently, it does not fully capture the computational and energy constraints of modern smartphones, leaving an open question regarding its applicability to true mobile platforms.

3.2 MLLM

The MLLM framework [52] addresses the challenge of accelerating LLM inference on mobile hardware by leveraging efficient NPU offloading. The system introduces several key innovations, including a chunk-sharing graph for reuse of common computation paths, shadow outlier execution to minimize irregular workload stalls, and out-of-order subgraph execution for enhanced parallelism. These optimizations enable impressive inference speeds, exceeding 1000 tokens per second on mobile NPUs. However, the scope of MLLM is limited to optimizing the inference stage of LLMs. It does not address the broader RAG pipeline, which involves retrieval, augmentation, and generation components. As such, it offers only a partial solution to the challenges of deploying complete RAG systems on mobile devices.

3.3 MobileRAG

MobileRAG [36] represents a more direct attempt to enable full RAG pipelines on smartphones. The system introduces two core techniques: *EcoVector*, a memory- and energy-efficient vector search mechanism that partitions large-scale data into smaller, load-on-demand graph structures, and *Selective Content Reduction*, which dynamically re-chunks retrieved documents, recalculates similarity scores, and reconstructs the augmented prompt using only the most relevant content. These optimizations significantly reduce both inference latency and energy consumption, demonstrating the potential for efficient RAG execution on mobile platforms. Despite these contributions, MobileRAG remains a closed-source system and does not provide comprehensive benchmarking or component-level analysis. In particular, it lacks transparency regarding which pipeline stages contribute most to overall latency and power consumption, limiting its usefulness for systematic performance evaluation.

3.4 RAGPerf

RAGPerf [28] introduces a comprehensive benchmarking suite for evaluating the performance characteristics of RAG pipelines. The framework decomposes RAG systems into modular components, including embedding generation, vector indexing, similarity search, and LLM inference. It measures their latency, throughput, and resource utilization under different configurations. By varying parameters such as embedding models, index structures, and generator backends, RAGPerf provides valuable insights into how design choices affect end-to-end RAG performance. However, RAGPerf is designed and evaluated primarily in server-class environments equipped with high-performance CPUs and GPUs. The framework assumes abundant memory, stable power supply, and mature acceleration libraries, conditions that differ substantially from those of mobile devices. As a result, while RAGPerf provides an excellent foundation for understanding RAG behavior at scale, it does not capture the constraints, hardware heterogeneity, and energy limitations inherent to smartphones.

3.5 Summary and Motivation

Collectively, these studies highlight the growing interest in bringing intelligent retrieval and generation systems to mobile devices. However, none of these works provide a modular, extensible benchmarking framework that systematically evaluates the performance of RAG components under real mobile constraints. This paper aims to fill that gap by developing an open, configurable platform for profiling and comparing RAG pipelines directly on smartphones, thereby extending the applicability of on-device AI beyond the current state of the art.

4

System Design

This chapter presents the design and implementation of MoRAGBench. We first discuss the key challenges associated with deploying RAG systems under mobile constraints and explain how these challenges influence the architectural design of the framework (Section 4.1). We then provide an overview of the system architecture (Section 4.2) and describe the main components of the pipeline (Section 4.3), including the text chunker, embedding module, indexing and retrieval system, augmenter, and language model. Finally, we introduce the benchmarking methodology implemented in the framework (Section 4.4), which includes both component-level ANN benchmarking and end-to-end task evaluation.

4.1 Challenges of Mobile RAG Benchmarking

Deploying and evaluating RAG pipelines on mobile devices introduces several challenges that differ significantly from traditional server-based environments. These challenges directly influence the architectural decisions made in the design of MoRAGBench.

Resource Constraints. Mobile devices provide substantially fewer computational resources than server-class hardware. Components of a RAG pipeline, such as embedding generation, similarity search, and language model inference, are computationally intensive and can become bottlenecks when executed on-device. These limitations make it difficult to directly apply architectures and evaluation methodologies designed for powerful desktop or cloud environments.

Memory Limitations. RAG systems typically rely on large collections of embeddings, vector indices, intermediate data structures, and model weights. Storing and processing these representations can require significant memory, which may exceed the available capacity on mobile devices. As a result, system configurations that are practical in server environments may become infeasible in mobile settings, making memory efficiency a key consideration.

Hardware Heterogeneity. Modern smartphones integrate multiple specialized processing units, including CPUs, GPUs, and NPUs. However, the availability and maturity of software frameworks that efficiently utilize these hardware accelerators vary significantly across devices. This heterogeneity complicates both the implementation and evaluation, as performance characteristics can depend heavily on the underlying hardware and execution backend.

Complex Interactions Between Pipeline Components. A RAG pipeline consists of multiple interconnected stages, including chunking, embedding generation, indexing and retrieval,

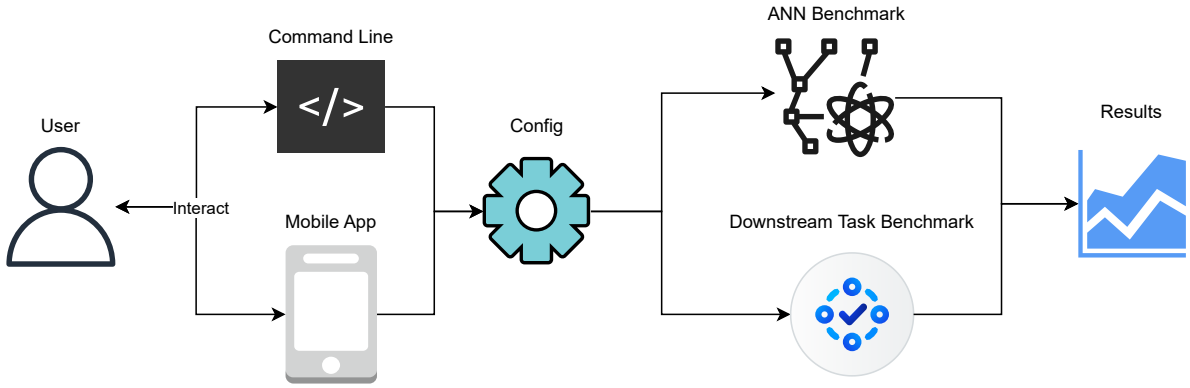


Figure 4.1: High-level architecture of MoRAGBench.

augmentation, and language model inference. The performance of the overall system does not depend solely on the efficiency of individual components, but also on how these components interact. For example, choices such as chunk size, embedding dimensionality, or indexing strategy can influence retrieval accuracy, memory consumption, and end-to-end latency in non-trivial ways.

Lack of Systematic Evaluation Tools. Despite growing interest in on-device intelligence, there is currently limited tooling for systematically evaluating RAG pipelines under realistic mobile constraints. Existing benchmarks are often designed for server environments and therefore fail to capture the unique limitations and trade-offs present on smartphones.

These challenges highlight the need for a structured approach to analyzing RAG pipelines in mobile environments. In the following section, we present the design of MoRAGBench, a framework developed to enable systematic benchmarking of RAG systems directly on mobile devices.

4.2 Overview

This work presents MoRAGBench, a modular benchmarking framework for evaluating RAG pipelines on Android devices. The framework is designed to systematically assess both the quality of ANN retrieval and the end-to-end performance of RAG-based question answering tasks under different pipeline configurations. It enables controlled experimentation by allowing users to define a complete configuration of the RAG pipeline and then evaluate it through two complementary types of benchmarks.

Figure 4.1 illustrates the overall architecture of the system. A user interacts with the framework through either a command-line interface or a mobile application. A configuration file specifies the parameters of the entire RAG pipeline, including document chunking strategy, embedding settings, indexing method, retrieval parameters, and LLM behavior. This configuration drives the execution of the selected benchmark, after which the system reports the evaluation results.

The framework supports two types of benchmarks:

- **ANN Benchmark:** This benchmark is based on this paper [3]. It isolates the retrieval component. Given a set of queries and a pre-built index, the system retrieves the top-k

nearest neighbors for each query. The retrieved results are then compared against ground truth neighbors, and metrics such as Recall@k are computed. This benchmark focuses solely on the effectiveness and efficiency of the indexing and retrieval mechanisms.

- **Downstream Task Benchmark:** This benchmark evaluates the complete RAG pipeline in an end-to-end setting. A corpus is constructed from the documents of a certain dataset. For each item in the dataset, the system goes through the whole RAG pipeline (see Figure 2.1), which performs query embedding, retrieves the top-k relevant document chunks, and feeds them to an LLM to generate an answer. The generated answers are then evaluated against ground truth answers using task-level metrics such as Exact Match, F1 score, and BLEU. This benchmark captures the combined effect of chunking, embedding, retrieval, and generation on downstream task performance.

The RAG pipeline in the framework is composed of the following modular components:

- **Text Chunker:** Responsible for splitting input documents into chunks of configurable token sizes.
- **Embedding:** Generates vector representations for document chunks and queries.
- **Indexing and Retrieval:** Builds and queries an index over the embeddings and manages the association between chunks and their source text.
- **Augmenter:** Augments the retrieved document with the input query before passing it to the LLM.
- **LLM:** Generates final answers based on the retrieved context and the input query.

Each component is independently configurable, enabling flexible experimentation with different RAG design choices. The overall architecture follows a highly modular design that facilitates straightforward extensibility. Moreover, every component is implemented as a stand-alone module that exposes high-level APIs for seamless integration and reuse.

4.3 Components

This section is going to describe each component in detail, including the design choices, technology used, configuration options, and role within the overall benchmarking framework.

4.3.1 Text Chunker

The Text Chunker is the first component in the RAG pipeline and plays a foundational role in determining the effectiveness of the subsequent retrieval and generation stages. In a typical RAG setting, the corpus consists of large, unstructured documents such as articles, manuals, reports, or web pages. Directly embedding and indexing entire documents is impractical and often ineffective, as relevant information is usually localized to small portions of the text. Therefore, a typical solution is to divide documents into smaller, semantically meaningful units called *chunks*, which serve as the atomic units for embedding, indexing, and retrieval [26].

The Text Chunker transforms raw documents into fixed-size chunks that can individually be embedded and stored in the vector index. During query time, retrieval operates over these chunks rather than whole documents, allowing us to return fine-grained and contextually relevant passages to the LLM. To ensure that chunks remain meaningful when retrieved in isolation, some systems [50] (including ours) introduce an overlap between consecutive chunks. This de-

Table 4.1: Text Chunker Configuration Options

Parameter	Type	Description
`method`	String	Chunking strategy: `word`, `character`, or `token`.
`size`	Integer	Number of words, characters, or tokens per chunk.
`overlap_enabled`	Boolean	Indicates whether consecutive chunks share overlapping content.
`overlap_size`	Integer	Number of words, characters, or tokens shared between adjacent chunks when overlap is enabled.

sign balances retrieval granularity with semantic coherence and aligns chunk boundaries with the token-based constraints of modern embedding models and LLMs.

Design and Implementation

We design the Text Chunker as a lightweight and fully configurable component that supports three chunking strategies: word-based, character-based and token-based chunking. In practice, token-based chunking is the most commonly used method because it directly corresponds to how embedding models and LLMs process input text.

Due to the simplicity and performance requirements of this component on mobile devices, we implement the chunking logic entirely in Kotlin. However, token-based chunking requires accurate tokenization compatible with transformer models. To support this, we cross-compile the Hugging Face tokenizer [32] to run on Android devices.

Cross-compilation refers to compiling a software library into a binary format that is compatible with a different target platform than the one used for development. In our case, this produces a native binary that executes efficiently within the Android environment.

Configurable Options

We expose all chunking parameters through the pipeline configuration file, allowing the Text Chunker to act as a controllable experimental variable in the benchmarking framework. Users can select the chunking method, define the chunk size, and enable or disable overlap between consecutive chunks. Table 4.1 summarizes the available configuration options.

This flexible design allows us to systematically study how different chunking strategies influence retrieval quality and end-to-end RAG performance.

4.3.2 Embedding

The Embedding component is responsible for converting text into numerical vector representations in a high-dimensional space. In a RAG pipeline, it's used to encode both the document chunks produced by the Text Chunker and the user query. These vector representations enable similarity search in the vector index, allowing the retrieval component to identify the most relevant chunks for a given query.

Table 4.2: Embedding Component Configuration Options

Parameter	Type	Description
`model_name`	String	Name of the embedding model (e.g., `all-MiniLM-L6-v2`).
`dtype`	String	Numerical precision of the model weights (e.g., `fp32`, `int8`).
`backend`	String	Execution backend used for inference: `CPU`, `XNNPACK`, or `NNAPI`.

Embeddings play a central role in the effectiveness of RAG. The quality of the vector representation directly influences how well semantically related pieces of text are positioned near each other in the embedding space [31]. If the embedding model captures semantic meaning accurately, queries and relevant chunks will be close in this space, which leads to higher retrieval recall and, consequently, better downstream answer generation by the LLM [25].

Design and Implementation

We design the Embedding component as a thin wrapper around pre-trained embedding models that can be executed efficiently on Android devices. To achieve this, we adopt the **ONNX Runtime** framework [35], an open-source inference engine developed by Microsoft. We choose ONNX Runtime for three main reasons.

First, it is well-documented, easy to integrate, and actively maintained, which simplifies development and deployment on Android. Second, ONNX Runtime supports both embedding models and LLM inference within the same framework. Although we could theoretically use separate frameworks for embeddings and generation, using a single runtime significantly simplifies the implementation and reduces system complexity. Third, ONNX Runtime natively supports delegating parts of the computation to hardware-accelerated backends beyond the CPU.

In particular, in MoRAGBench, we support three different backends: CPU only, XNNPACK [16] and NNAPI [15]. XNNPACK is a high-performance neural network inference library optimized for ARM CPUs and mobile devices, providing faster execution for quantized and floating-point models. NNAPI (Neural Networks API) is a hardware abstraction layer provided by Android that enables neural network workloads to be accelerated using available on-device hardware such as GPUs, DSPs, and NPUs through vendor drivers. By delegating execution to specialized accelerators when present, NNAPI can significantly reduce latency and power consumption compared to CPU-only inference. This flexibility allows us to benchmark how different hardware backends affect embedding performance on mobile devices.

Currently, MoRAGBench supports two embedding models: *all-MiniLM-L6-v2* [45] and *all-MiniLM-L12-v2* [44]. However, the system is easily extensible to other models, as long as they can be converted to the ONNX format. This design allows us to keep the framework model-agnostic while maintaining compatibility with efficient mobile inference.

Configurable Options

We expose the embedding configuration through the pipeline configuration file, allowing users to control which model and hardware backend are used during benchmarking. This makes the Embedding component another important experimental variable in the framework. Table 4.2 summarizes the available configuration options.

This modular and hardware-aware design allows us to systematically evaluate how embedding models, numerical precision, and hardware acceleration influence retrieval quality and end-to-end RAG performance on Android devices.

4.3.3 Indexing and Retrieval

The Indexing and Retrieval component forms the core of the RAG pipeline. After the knowledge source has been converted into vector representations by the Embedding component (see Section 4.3.2), we need a fast and effective mechanism to retrieve the most semantically similar chunks for a given user query. The role of this component is to take the embedded user query and perform similarity search over the indexed document vectors, returning the closest matches. These retrieved chunks are then passed to the LLM to generate the final answer.

In practice, a knowledge source may contain thousands or even millions of chunks. Performing similarity comparisons against every vector in the database can be prohibitively expensive on resource-constrained mobile devices. Therefore, we first add all chunk embeddings to an index structure that is optimized for fast similarity search. At query time, this index allows us to retrieve the top-k most relevant chunks efficiently.

As discussed in Section 2.2, indexing methods generally fall into two main categories: *Exact Nearest Neighbor* (ENN) search and *Approximate Nearest Neighbor* (ANN) search. In ENN, sometimes referred to as a *Flat* index, the system computes the similarity metric (such as cosine similarity or Euclidean distance) between the query and every vector in the database. This approach produces the most accurate results, but it becomes computationally expensive as the database grows [22].

ANN methods address this issue by sacrificing a small amount of accuracy in exchange for significantly faster retrieval. Common ANN techniques include:

- **IVF (Inverted File Index) [42]:** Partitions the vector space into clusters and restricts the search to a subset of the most relevant clusters, reducing the number of comparisons.
- **HNSW (Hierarchical Navigable Small World) [29]:** Organizes vectors in a multi-layer graph structure that enables fast approximate search by navigating through nearest neighbors.
- **PQ (Product Quantization) [21]:** Compresses vectors into compact codes, allowing faster distance computations and reduced memory usage.

Design and Implementation

To support these indexing methods on Android, we evaluated available libraries for similarity search. To the best of our knowledge, there are very few ready-to-use Android libraries that support multiple ANN methods. However, existing solutions usually only support a single indexing method, such as Spotify's *Voyager* library [47], which only supports HNSW.

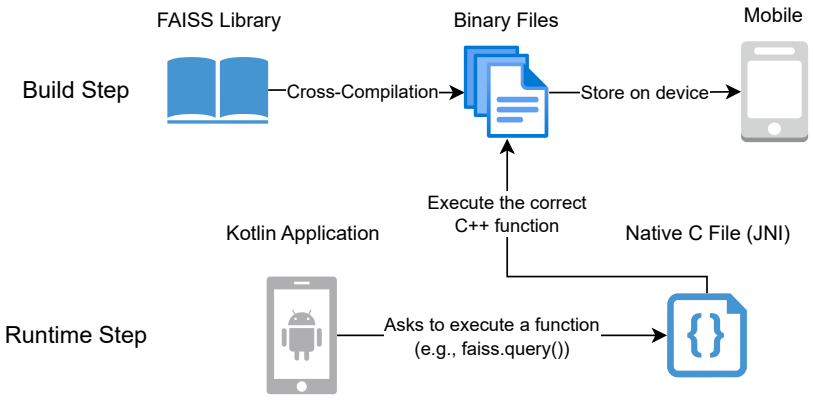


Figure 4.2: High-level workflow of FAISS execution.

On the other hand, one of the most widely used libraries for similarity search is **FAISS** [12]. It is an efficient library that supports a wide range of indexing methods, provides extensive configuration options, and is easy to use. However, FAISS is primarily designed for desktop and server environments, and there is no official Android version.

To overcome this limitation, we cross-compile FAISS into Android-compatible binary files. These binaries contain the optimized FAISS routines for vector indexing and similarity search, allowing us to reuse its mature and highly optimized implementation. The main limitation of this approach is that the current cross-compiled version only supports CPU execution, since the official library does not provide native GPU or NPU kernels for Android. We leave the integration of such kernels as future work.

Since the Android application is written in Kotlin and FAISS is implemented in C++, we introduce an interface layer using the Java Native Interface (JNI). JNI provides a standard mechanism for Android applications to call native C or C++ code. In our design, the JNI layer acts as a bridge between the Kotlin code and the native FAISS library. On the native side, a small wrapper exposes selected FAISS functions through JNI. On the Android side, a Kotlin class declares the corresponding native methods and invokes them as needed. The JNI layer forwards these calls to the FAISS library and returns the results back to the Kotlin code.

Through this structure, the Android application interacts with FAISS using simple Kotlin method calls, while the complexity of the native C++ implementation remains hidden behind the JNI interface. Figure 4.2 summarizes this procedure.

Configurable Options

Currently, MoRAGBench supports three index types: Flat, IVF, and HNSW. Later in Chapter 5, we compare these methods in terms of retrieval speed and accuracy.

We also expose several indexing and retrieval parameters to the user, allowing fine-grained control over the search behavior and performance characteristics. These include the index type, similarity metric, number of retrieved chunks, and additional hyperparameters specific to each indexing method. Table 4.3 summarizes the available configuration options.

This flexible and hardware-aware design allows us to systematically evaluate how different indexing methods and hyperparameters affect retrieval speed, memory usage, and overall RAG

Table 4.3: Indexing and Retrieval Configuration Options

Parameter	Type	Description
`method`	String	Indexing method: `flat`, `ivf`, or `hnsw`.
`metric`	String	Distance metric used for similarity search: `L2` (Euclidean distance) or `inner_product`.
`top_k`	Integer	Number of nearest chunks retrieved for each query.
`batch_size`	Integer	Number of vectors processed per batch during indexing or search.
`backend`	String	Execution backend (currently CPU only).
`use_cache`	Boolean	Enables caching of index data to avoid rebuilding the index.
`ivf.nlist`	Integer	Number of clusters in the IVF index.
`ivf.nprobe`	Integer	Number of clusters searched during query time in IVF.
`ivf.num_training_vectors`	Integer	Number of vectors used to train the IVF clusters.
`hnsw.m`	Integer	Number of connections per node in the HNSW graph.
`hnsw.ef_construction`	Integer	Controls graph quality during index construction.
`hnsw.ef_search`	Integer	Controls search accuracy and speed at query time.

performance on mobile devices.

4.3.4 Augmenter

The Augmenter is the component responsible for transforming the retrieved external knowledge into a form that can be effectively consumed by the LLM. It operates between the retriever, which selects the most relevant document chunks, and the generator, which produces the final response (see Figure 2.1). While the retriever focuses on identifying relevant information, the augmenter determines how that information is organized, filtered, and presented to the model.

The design of the augmenter has a direct impact on the quality of the generated answers [26]. Even if the retriever returns relevant chunks, poorly structured or redundant context can reduce the effectiveness of the LLM [33]. Therefore, the augmenter controls how much context is used, in what order it appears, and how it is formatted within the final prompt.

Several augmentation strategies have been proposed in the literature:

- **Simple concatenation:** The retrieved chunks are directly concatenated and inserted into the prompt along with the user query. This is the most common and lightweight approach, requiring no additional computation.
- **Reranking-based augmentation [34]:** The retrieved passages are rescored using a more precise model, such as a cross-encoder, and then reordered according to their estimated

relevance to the query. This often improves retrieval quality but introduces additional computational overhead.

- **Multi-step or iterative augmentation [13, 54]:** Instead of a single retrieval–augmentation step, the system performs multiple rounds of retrieval and context construction. The LLM may generate intermediate queries or refine the context iteratively, which can improve reasoning but significantly increases latency and resource usage.

In MoRAGBench, we adopt the simple concatenation strategy. This choice is motivated by the constrained computational and memory resources available on mobile devices. More advanced augmentation techniques, such as reranking or iterative retrieval, require additional models or multiple inference steps, which would substantially increase latency and energy consumption. Since the goal of this work is to benchmark RAG pipelines on mobile hardware, we prioritize a lightweight and efficient augmentation strategy.

As a result, the Augmenter component in the current system does not expose any user-configurable parameters. It simply concatenates the retrieved chunks with the query to form the final prompt that is passed to the LLM.

4.3.5 LLM

After the Retriever selects the most relevant document chunks and the Augmenter organizes them together with the user query, the resulting prompt is passed to the LLM. The LLM is responsible for generating the final answer based on the provided context. In a RAG pipeline, the quality, latency, and resource consumption of the LLM have a direct impact on the overall system performance, especially on mobile devices with limited computational resources.

As discussed in Section 4.3.2, we use the same inference framework for both the embedding model and the LLM, namely ONNX Runtime [35]. This decision reduces operational complexity, simplifies integration, and minimizes the number of external dependencies. Although different frameworks could be used for embedding and generation, relying on a single runtime makes the system easier to maintain and deploy on Android devices.

Design and Implementation

We design the LLM module to be modular and easy to integrate within the benchmarking framework. The module exposes a small set of high-level interfaces, such as `llm.initialize` and `llm.generate`, which abstract away the underlying inference details. This design allows us to easily swap models or adjust generation parameters without affecting other components of the pipeline.

The LLM runs through ONNX Runtime and supports different execution backends, including CPU, XNNPACK, and NNAPI (see Section 4.3.2 for more details about each backend). Similar to the Embedding component, this enables hardware-aware benchmarking by allowing the system to offload parts of the computation to optimized mobile backends when available.

Currently, MoRAGBench supports lightweight language models that are suitable for on-device inference. The framework is designed to remain model-agnostic, and any model that can be converted to the ONNX format can be integrated into the system.

Configurable Options

We expose a set of generation and runtime parameters through the configuration file, allowing users to control model selection, decoding behavior, and memory-related settings. These options enable systematic evaluation of how different model configurations affect latency, memory usage, and answer quality. Table 4.4 summarizes the available configuration options.

Table 4.4: LLM Configuration Options

Parameter	Type	Description
`model_name`	String	Name of the language model used for generation.
`aug_method`	String	Augmentation strategy used to construct the prompt (currently concatenation only).
`backend`	String	Execution backend: `CPU`, `XNNPACK`, or `NNAPI`.
`dtype`	String	Numerical precision of the model weights (e.g., `fp32`, `int8`).
`use_sampling`	Boolean	Enables stochastic decoding instead of deterministic generation.
`repetition_penalty`	Float	Penalty applied to repeated tokens to reduce repetition in generated text.
`temp`	Float	Sampling temperature controlling the randomness of the output.
`top_p`	Float	Nucleus sampling parameter that limits token selection to a cumulative probability mass.
`top_k`	Integer	Limits token selection to the top-k most probable tokens during sampling.
`max_tokens`	Integer	Maximum number of tokens generated for each response.
`ignore_eos`	Boolean	Indicates whether the model should ignore end-of-sequence tokens during generation.
`generate_until`	List of Strings	Stops generation when one of the specified sequences is produced.
`kv_window`	Integer	Size of the KV cache window used to reduce memory usage.
`prefill_chunk_size`	Integer	Number of tokens processed per iteration during the prefill phase.
`system_prompt`	String	Instructional prompt that defines the model's behavior during generation.

This configurable and modular design allows us to benchmark different language models, backends, and decoding strategies, and to study their impact on end-to-end RAG performance on mobile devices.

4.4 Benchmarking Methodology

In MoRAGBench, we provide two complementary benchmarks that target different aspects of a RAG pipeline. The first benchmark focuses exclusively on vector indexing and retrieval performance, while the second benchmark evaluates the complete end-to-end behavior of the system on downstream question answering tasks. Together, these benchmarks allow us to decouple retrieval quality from generation quality and to analyze system performance under realistic mobile constraints.

4.4.1 ANN Benchmark

The ANN Benchmark evaluates the performance of approximate nearest neighbor search methods in isolation. This benchmark follows a well-established evaluation protocol and uses standard ANN datasets [3], each of which is characterized by a fixed embedding dimension, a set of training vectors, and a set of test vectors. For every test vector, the dataset provides the 100 nearest neighbors computed using exact search, which serve as the ground truth.

To run the benchmark, we first build an index using the training vectors. Then, for each test vector, we retrieve the top-k nearest neighbors from the index using the selected indexing method. The retrieved neighbors are compared against the ground truth neighbors provided by the dataset. This benchmark exercises only the indexing and retrieval component of the RAG pipeline, allowing us to study retrieval accuracy and efficiency independently of embedding quality or language model behavior.

One of the most commonly reported metrics in ANN evaluation is Recall@k. It measures the probability that the true nearest neighbor appears within the top-k retrieved results. In our implementation, this metric is computed per query and then averaged over all queries. It's defined as follows: Let N be the total number of queries. For a given query i, let $R_{i,k}$ denote the set of the top-k retrieved neighbors, and let g_i denote the closest ground-truth neighbor for that query. Recall@k is defined as:

$$\text{Recall@}k = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(g_i \in R_{i,k}) \tag{4.1}$$

where $\mathbf{1}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

In addition, we report several performance-related measurements, including total benchmark runtime, index construction time, average retrieval latency, query embedding time, and queries per second (QPS). We also report both average and peak memory usage during the benchmark execution. A detailed analysis of these metrics is presented in Chapter 5.

4.4.2 Task Benchmark

The Task Benchmark evaluates the full RAG pipeline in an end-to-end setting. Unlike the ANN benchmark, this benchmark includes all components of the system: text chunking, embedding, indexing and retrieval, augmentation, and generation with an LLM.

This benchmark operates on question answering datasets that are formulated as information extraction tasks. Each item consists of a set of documents, a question, and one or more ground

truth answers that can be found within the documents. Currently, MoRAGBench supports three datasets: SQuAD [37, 38], TriviaQA [23], and HotpotQA [53], but the framework is easily extensible to additional datasets.

When running the benchmark, we first collect all documents from the dataset and construct a shared knowledge source. The documents are chunked, embedded, and indexed according to the selected configuration. For each question, we embed the query, retrieve the top-k most relevant chunks, and pass them, together with the question, to the LLM to generate an answer. The generated answer is then compared against the ground truth provided by the dataset.

To evaluate answer quality, we report several widely used task-level metrics:

- **Exact Match (EM):** Measures whether the generated answer exactly matches the ground truth answer.
- **F1 Score:** Computes the harmonic mean of precision and recall at the token level between the generated answer and the ground truth.
- **BLEU:** Measures n-gram overlap between the generated answer and the reference answer, commonly used in text generation tasks.
- **ROUGE:** Evaluates recall-oriented n-gram overlap, emphasizing coverage of the reference answer.
- **Contains:** Indicates whether the ground truth answer appears as a substring within the generated answer.

In addition to these accuracy metrics, we report the same performance-related measurements as in the ANN benchmark, including total runtime, latency breakdowns, QPS, and both average and peak memory usage. This allows us to analyze the trade-offs between retrieval accuracy, generation quality, and system efficiency when deploying RAG pipelines on mobile devices.

5

Evaluation

In this chapter, we evaluate the proposed benchmarking framework and analyze the performance of different RAG pipeline configurations on a modern mobile device. As described in Section 4.4, the framework provides two complementary types of benchmarks: the ANN Benchmark, which focuses on the indexing and retrieval component, and the Task Benchmark, which evaluates the complete end-to-end RAG pipeline.

The goal of this chapter is to investigate how different hardware backends, indexing methods, model configurations, and different other parameters affect both retrieval quality and downstream task performance. We report a variety of metrics, including task accuracy, Recall@k, memory usage, latency, and throughput. Through these experiments, we aim to identify the main performance bottlenecks when running RAG pipelines on mobile devices and to understand the trade-offs between accuracy, latency, throughput and efficiency.

This chapter is structured as follows. First, we describe the experimental setup, including the hardware platform and general evaluation methodology. We then present the results of the ANN benchmarks, focusing on the retrieval component across different indexing methods and configurations. Finally, we evaluate the full RAG pipeline using the task benchmarks and analyze how different system components influence end-to-end performance.

5.1 Experimental Setup

All experiments are conducted on a **OnePlus 13** Android smartphone. The device is equipped with the following hardware specifications:

- **Platform:** Snapdragon® 8 Elite Mobile Platform (with integrated NPU)
- **CPU:** Qualcomm® Oryon™ CPU @ 4.32 GHz
- **GPU:** Adreno™ 830
- **RAM:** 16 GB LPDDR5X
- **Storage:** 512 GB UFS 4.0

In order to run the benchmarks, we deploy a lightweight HTTP server directly on the mobile device. This server exposes a set of routes that allow external clients to control and monitor benchmark execution. To interact with the server, we provide a Python-based client script.

The client script is responsible for preparing the experimental environment. This includes downloading the required datasets, embedding models, and LLMs, and transferring them to

Table 5.1: Selected ANN Benchmark Datasets

Dataset	Dimensions	Train Size	Test Size	Neighbors	Distance
Fashion-MNIST	784	60,000	10,000	100	Euclidean
GloVe-100	100	1,183,514	10,000	100	Angular
NYTimes	256	290,000	10,000	100	Angular
LastFM	65	292,385	50,000	100	Angular
SIFT	128	1,000,000	10,000	100	Euclidean

the mobile device using `adb` commands. The user only needs to provide a RAG configuration file, and the script automatically handles all setup steps, including model preparation and data transfer.

Once the environment is ready, the client triggers the benchmark by sending a request to the HTTP server, for example through an endpoint such as `/start_benchmark`. This setup allows experiments to be launched, monitored, and repeated in a controlled and automated manner, while keeping all computation on the mobile device.

5.2 ANN Benchmark Evaluation

In this section, we evaluate the performance of the indexing and retrieval component using the ANN benchmark described in Section 4.4.1. The objective of these experiments is to analyze how different index structures and configurations affect retrieval accuracy, latency, throughput, and memory usage on a mobile device.

Although the full ANN benchmark suite [3] provides 15 different datasets, we select a representative subset of five datasets for this study due to space and time constraints. These datasets vary in dimensionality, data distribution, and scale, allowing us to evaluate the behavior of different indexing methods under diverse conditions. Table 5.1 summarizes the characteristics of the selected datasets.

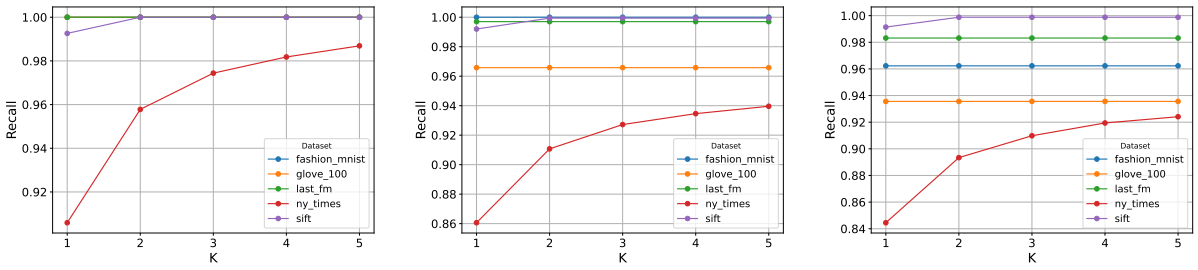
Across these datasets, we compare three indexing methods supported by MoRAGBench: Flat (exact search), IVF, and HNSW. For each method, we evaluate both retrieval accuracy and system-level performance metrics.

In the following subsections, we present the results of each experiment and discuss the observed trade-offs between accuracy, latency, throughput, and memory usage.

5.2.1 Recall@k Across Index Types

In this experiment, we analyze how Recall@k changes with different values of k for each indexing method (Flat, IVF, and HNSW) across the selected datasets. For each index type, we fix a single configuration and vary the value of k from one to five. The results are summarized in Figure 5.1

Flat Index. The Flat index performs exact nearest neighbor search (ENN). In theory, this should result in a Recall@k of 1.0 for all values of k, since the exact distances are computed against all vectors in the database. This behavior is almost observed in Figure 5.1(a).



(a) Flat index (b) IVF index (c) HNSW index

Figure 5.1: Recall@k for different index types across the selected ANN datasets.

For most datasets, such as Fashion-MNIST and LastFM, the recall reaches 1.0 even for $k = 1$. However, minor deviations appear in certain cases. For example, in the SIFT dataset, Recall@1 is approximately 0.99 before reaching 1.0 for larger values of k. Similarly, in the NYTimes dataset, Recall@1 is around 0.91 and gradually increases to approximately 0.99 at $k = 5$.

These small discrepancies may be caused by several factors, such as metric mismatches between the benchmark and the index configuration, differences in floating-point precision, or ties and near-ties in distances between neighboring vectors.

IVF Index. For the IVF index, we select a configuration for each dataset based on its size and dimensionality, following common heuristics used in prior work [40, 11]. We then vary only the value of k.

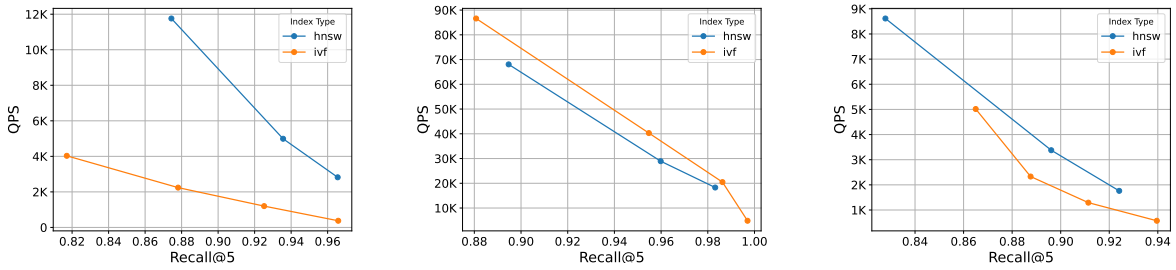
As shown in Figure 5.1(b), the IVF index exhibits a very similar trend to the Flat index. Most datasets achieve recall values close to 1.0 even for small values of k. For example, Fashion-MNIST and SIFT reach nearly perfect recall at $k = 2$. The LastFM dataset maintains a recall of approximately 0.99 across all values of k.

The NYTimes dataset shows lower recall compared to the other datasets, starting at roughly 0.91 for $k = 1$ and increasing to about 0.94 at $k = 5$. The GloVe-100 dataset remains stable at approximately 0.97 across all values of k. Overall, IVF maintains high recall while benefiting from approximate search efficiency.

HNSW Index. We perform the same experiment using the HNSW index with a fixed configuration for each dataset. As illustrated in Figure 5.1(c), HNSW follows a trend similar to both Flat and IVF indices.

For example, the SIFT dataset achieves a recall of about 0.99 at $k = 1$ and reaches nearly 1.0 at larger values of k. The LastFM dataset maintains a recall of approximately 0.98 across all values of k, while Fashion-MNIST and GloVe-100 remain stable at around 0.96 and 0.94 respectively.

The NYTimes dataset again shows the lowest recall among the datasets, starting at approximately 0.84 for $k = 1$ and increasing to around 0.92 at $k = 5$. Despite these differences, HNSW consistently improves recall as k increases and demonstrates behavior comparable to IVF.



(a) GloVe-100 (b) LastFM (c) NYTimes

Figure 5.2: Throughput (QPS) vs. Recall@5 for IVF and HNSW indices across three representative datasets.

Summary. Across all index types, Recall@k increases with larger values of k, as expected. The Flat index achieves near-perfect recall due to exact search, while IVF and HNSW provide similar recall levels with approximate search. The results indicate that both ANN methods can closely match the accuracy of exact search under appropriate configurations.

5.2.2 Throughput vs. Recall Trade-off

In this experiment, we analyze the trade-off between retrieval throughput, measured in queries per second (QPS), and retrieval accuracy, represented by Recall@5. Since the value of k is fixed in this experiment, the Flat index is not included, as it does not provide tunable parameters that allow a meaningful accuracy–throughput trade-off.

We therefore focus on the two approximate indexing methods: IVF and HNSW. For each dataset, we vary one key hyperparameter that directly affects the search effort and observe how this impacts both recall and throughput.

IVF Index. For the IVF index, we fix the number of clusters (*nlist*) and the number of training vectors, and vary the *nprobe* parameter. The *nprobe* parameter controls how many clusters are searched per query. A larger *nprobe* means that more clusters are examined, which typically improves recall but increases query latency, resulting in lower QPS.

This trend is clearly visible in Figure 5.2. For example, in the GloVe-100 dataset (Figure 5.2a), increasing recall from approximately 0.82 to 0.96 reduces throughput from about 4,000 QPS to below 500 QPS. A similar pattern appears in the NYTimes dataset (Figure 5.2c), where throughput drops from around 5,000 QPS at recall 0.86 to roughly 600 QPS at recall 0.94. The LastFM dataset (Figure 5.2b) exhibits the same trade-off, with throughput decreasing from approximately 86,000 QPS at recall 0.88 to about 5,000 QPS at near-perfect recall. These results demonstrate the expected behavior of IVF: higher search effort leads to better accuracy at the cost of reduced throughput.

HNSW Index. For the HNSW index, we fix the graph construction parameters (M and $ef\_construction$) and vary the ef_search parameter. The ef_search parameter controls the size of the candidate list explored during search. Higher values allow the algorithm to explore

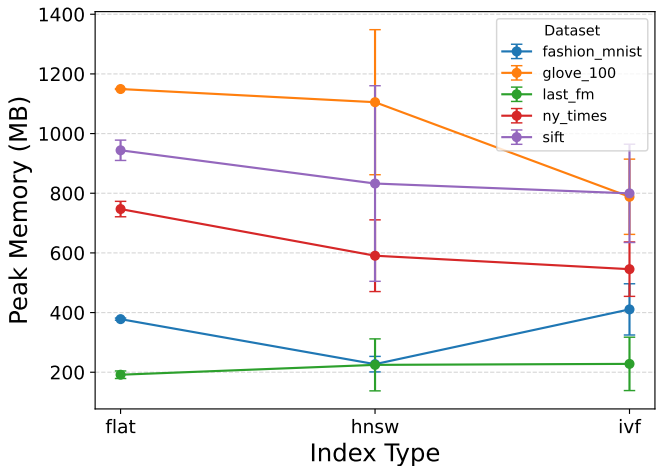


Figure 5.3: Peak memory usage for different index types across the selected datasets. Error bars represent the variation across different configurations.

more nodes in the graph, which typically improves recall but increases computation time per query.

The same trade-off is observed across all datasets. In the GloVe-100 dataset (Figure 5.2a), increasing recall from approximately 0.87 to 0.96 reduces throughput from about 11,800 QPS to around 2,800 QPS. In the NYTimes dataset (Figure 5.2c), throughput drops from roughly 8,600 QPS at recall 0.83 to about 1,800 QPS at recall 0.92. In the LastFM dataset (Figure 5.2b), the throughput decreases from approximately 68,000 QPS at recall 0.89 to around 18,000 QPS at recall 0.98. This confirms that increasing ef_search improves recall at the cost of lower throughput.

Summary. Across all datasets, both IVF and HNSW exhibit a clear accuracy–throughput trade-off. Higher recall levels require more extensive search, which increases per-query computation and reduces QPS.

5.2.3 Peak Memory usage

In this experiment, we analyze the peak memory usage of all three index types across the five selected datasets. For each index type, we run the benchmark under multiple configurations by tuning the available hyperparameters (see Table 4.3). We then average the peak memory measurements across these configurations to obtain a representative value for each dataset and index type.

The results are shown in Figure 5.3. Overall, memory usage is strongly correlated with the size of the dataset, which is primarily determined by the product of the vector dimensionality and the number of training vectors (see Table 5.1). Datasets with larger vector counts or higher dimensionality, such as GloVe-100 and SIFT, consistently exhibit higher memory usage than smaller datasets like LastFM.

Across index types, the Flat index generally exhibits predictable memory usage, since it stores the full set of vectors without additional data structures. For example, the GloVe-100 dataset reaches peak memory usage of approximately 1.15 GB, while the SIFT dataset uses around 0.95 GB. Smaller datasets such as LastFM and Fashion-MNIST remain below 400 MB.

The HNSW index shows greater variation in memory usage across configurations, as reflected by the larger error bars. This is expected because the graph structure introduces additional memory overhead that depends on parameters such as M and $ef\_construction$. For instance, GloVe-100 ranges roughly between 900 MB and 1.3 GB, while SIFT ranges between approximately 500 MB and 1.1 GB.

The IVF index typically consumes slightly less memory than the Flat index for most datasets. For example, GloVe-100 and SIFT both use around 800 MB under IVF, compared to higher values in the Flat configuration. This reduction is due to the clustered structure of the index, which can store vectors more compactly depending on the configuration.

Summary. Peak memory usage is primarily driven by dataset size, while the index structure introduces secondary overhead. Flat indexing provides predictable memory usage, HNSW incurs additional graph-related overhead, and IVF can offer modest memory savings depending on the configuration.

5.2.4 Retrieval Latency vs. Recall Trade-off

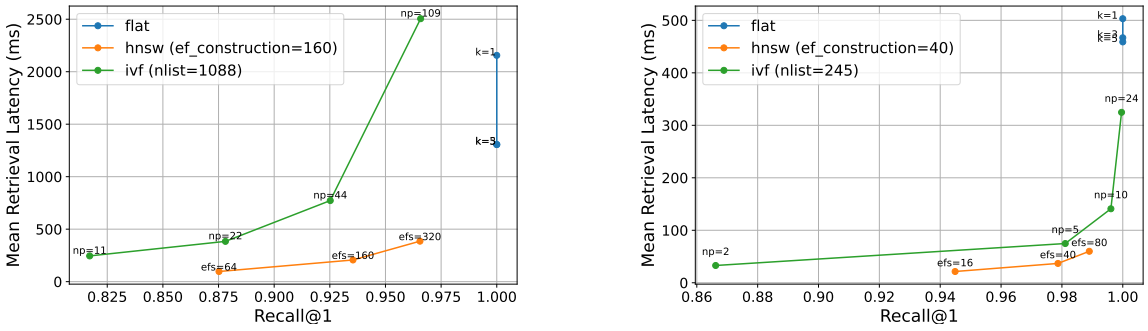
In this experiment, we analyze how the average per-query retrieval latency changes as Recall@1 increases. We present results for two representative datasets: GloVe-100 and Fashion-MNIST.

For the Flat index, we vary the value of k between 1, 3, and 5. Since the Flat index performs exact search, the primary factor affecting latency is the number of neighbors retrieved. For the IVF index, we fix the number of clusters (*nlist*) and vary the *nprobe* parameter, which controls the number of clusters searched per query. For the HNSW index, we fix $ef\_construction$ and vary the ef_search parameter, which determines how many candidate nodes are explored during the graph traversal. Larger values of these parameters generally improve recall at the cost of higher latency.

GloVe-100 Dataset. As shown in Figure 5.4(a), the Flat index achieves perfect recall but at a very high latency, exceeding 1.3 seconds per query for $k = 5$ and more than 2 seconds for $k = 1$. This reflects the cost of exact search over a large dataset containing more than one million vectors.

The IVF index shows a clear trade-off. At a lower *nprobe* value, it achieves recall around 0.82 with a latency of roughly 250 ms. As *nprobe* increases, recall improves to approximately 0.96, but latency rises sharply to about 2.5 seconds. This demonstrates that higher search effort in IVF can approach the accuracy of exact search, but with comparable latency at extreme settings.

The HNSW index provides a more favorable trade-off. At recall around 0.88, the latency is approximately 100 ms. Increasing ef_search improves recall to about 0.96, while latency increases



(a) GloVe-100 (b) Fashion-MNIST

Figure 5.4: Mean per-query retrieval latency vs. Recall@1 for different index types.

to roughly 400 ms. Compared to IVF at similar recall levels, HNSW achieves significantly lower latency.

Fashion-MNIST Dataset. A similar pattern is observed in Figure 5.4(b). The Flat index again achieves perfect recall, but with relatively high latency, around 460–500 ms per query.

The IVF index shows increasing latency as *nprobe* grows. At low recall (around 0.87), latency is approximately 35 ms. As recall approaches 1.0, latency increases to over 300 ms.

The HNSW index provides the lowest latency across most recall levels. At recall around 0.95, latency is approximately 25 ms, and even at near-perfect recall, it remains below 70 ms. This indicates that HNSW is particularly effective at maintaining low latency while achieving high recall on this dataset.

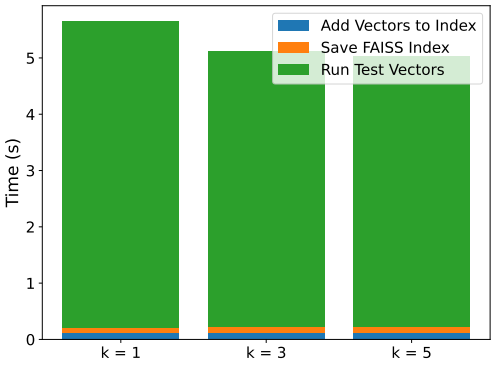
Summary. Across both datasets, the Flat index consistently achieves the highest recall but at the cost of significantly higher latency. The IVF index provides a controllable trade-off between recall and latency, but its latency increases rapidly at high recall levels. NSW generally offers the best latency–recall trade-off, achieving high recall with substantially lower latency than both Flat and IVF indices.

Additionally, overall latency is noticeably lower for the Fashion-MNIST dataset compared to GloVe-100. This is primarily due to the smaller number of vectors in Fashion-MNIST (60K $<$ $\sim$1.2M), which reduces the search space and leads to faster retrieval across all index types.

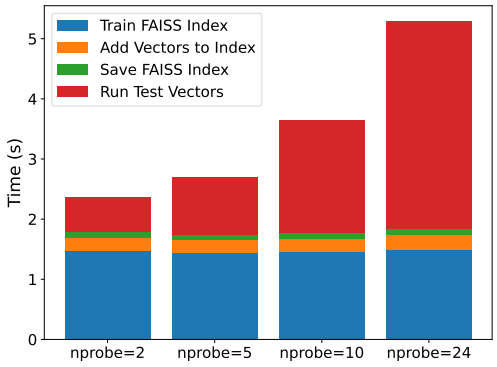
5.2.5 Total Benchmark Time Breakdown

In this experiment, we analyze the total time spent in each stage of the ANN benchmark. The benchmark consists of three main stages:

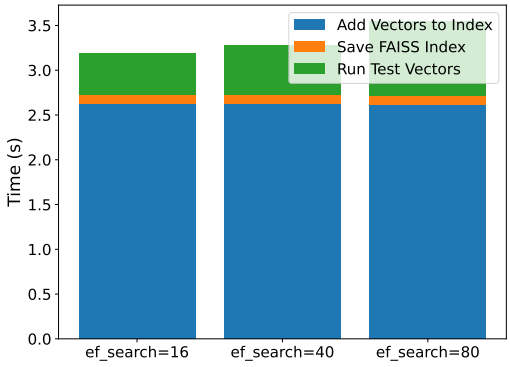
- **Index construction:** Adding all training vectors to the index.
- **Index serialization:** Saving the constructed index to disk for future use.
- **Benchmark execution:** Running queries for each test vector and retrieving the top-k nearest neighbors.



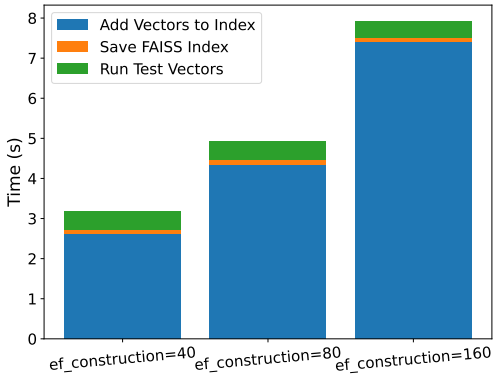
(a) Flat index



(b) IVF index



(c) HNSW ($ef\_construction = 40$)



(d) HNSW ($ef\_search = 16$)

Figure 5.5: Total benchmark time breakdown for the Fashion-MNIST dataset under different index types and configurations.

For the IVF index, there is an additional step before index construction:

- **Index training:** A clustering step that learns the centroids used to partition the vector space. This is typically done using a subset of the training vectors and is required to build the inverted lists before vectors can be added to the index.

The results for the Fashion-MNIST dataset are shown in Figure 5.5. Similar trends are observed across the other datasets but are omitted for space.

Flat Index. For the Flat index, we vary the value of k between 1, 3, and 5 (Figure 5.5a). Index construction and serialization are very fast, taking only around 0.2 s in total. This is expected because the Flat index simply stores the vectors without building any additional data structures.

The dominant cost comes from the benchmark execution stage. The total runtime is approximately 5.7 s for $k = 1$, and around 5 s for $k = 3$ and $k = 5$. The majority of this time is spent computing distances between each query and all vectors in the index.

Interestingly, the runtime for $k = 1$ is slightly higher than for $k = 3$ or $k = 5$. This may be caused by implementation details such as heap or partial sorting optimizations, caching effects, or vectorized distance computations that become more efficient when retrieving multiple neighbors.

IVF Index. For the IVF index, we fix $nlist = 245$ and $k = 5$, and vary *nprobe* (Figure 5.5b). The index training, vector addition, and index serialization stages remain nearly constant across at 1.8 s all configurations, since these steps do not depend on *nprobe*.

For small values of *nprobe* (e.g., $nprobe = 2$), the training stage dominates the total runtime. The total time is approximately 2.4 s, with the testing stage taking around 0.5 s. As *nprobe* increases, the testing stage becomes more expensive. For example, at $nprobe = 24$, the total runtime increases to about 5.3 s, with the testing stage alone taking more than 3.4 s. This behavior is expected, since higher *nprobe* values cause more clusters to be searched per query, increasing the computational cost of the retrieval stage.

HNSW Index (varying ef_search). In the first HNSW experiment, we fix $ef\_construction = 40$ and $k = 5$, and vary ef_search (Figure 5.5c). In all configurations, index construction is the dominant stage, taking approximately 2.6 s.

As ef_search increases from 16 to 80, the testing time also increases, from about 0.45 s to nearly 0.85 s. Consequently, the total runtime rises from roughly 3.2 s to 3.6 s. This is expected, since larger ef_search values require the algorithm to explore more candidate nodes during the search.

HNSW Index (varying $ef\_construction$). In the second HNSW experiment, we fix $ef\_search = 16$ and $k = 5$, and vary $ef\_construction$ (Figure 5.5d). The $ef\_construction$ parameter controls the size of the candidate list during index construction. Higher values lead to more thorough graph connections, improving search quality but increasing construction time.

This effect is clearly visible in the results. When $ef\_construction = 40$, the total runtime is approximately 3.2 s. Increasing it to 80 raises the total time to around 5 s, and at $ef\_construction = 160$, the total runtime reaches nearly 8 s. In all cases, index construction remains the dominant stage.

Summary. Across all index types, the dominant time component varies depending on the index structure. For the Flat index, the retrieval stage dominates due to exhaustive distance computations. For IVF, the dominant stage shifts from index training at low *nprobe* values to retrieval at higher *nprobe* values. For HNSW, index construction is consistently the main bottleneck, especially when using larger $ef\_construction$ values.

These results highlight that different index types exhibit different performance bottlenecks, which should be considered when selecting an indexing strategy for mobile RAG systems.

5.3 Task Benchmark Evaluation

In this section, we evaluate the complete RAG pipeline in an end-to-end setting using the Task Benchmark described in Section 4.4.2. Unlike the ANN evaluation, which isolates the retrieval component, this benchmark measures the interaction between all pipeline stages, including chunking, embedding, indexing, augmentation, and answer generation with the LLM.

Our goal is to analyze system-level trade-offs between task accuracy, throughput, time-to-first-token (TTFT), and memory consumption when running a full RAG pipeline on a mobile device.

Although MoRAGBench supports three downstream datasets (SQuAD, TriviaQA, and HotpotQA), we report results for the **TriviaQA** dataset in this section due to space and time constraints. The remaining datasets exhibit similar trends. For computational feasibility on mobile hardware, we sample the first 1000 questions from TriviaQA, resulting in a knowledge corpus composed of 1457 documents.

5.3.1 Baseline Configuration

All experiments start from a baseline RAG configuration. We summarize the baseline parameters in Tables 5.2–5.5.

Table 5.2: Downstream Task Configuration

Parameter	Value
Dataset	TriviaQA
Sampling Method	First-N samples
Number of Questions	1000
Number of Documents	1457

Table 5.3: Embedding and Chunking Configuration (Baseline)

Component	Configuration
Embedding Model	all-MiniLM-L6-v2
Embedding Backend	CPU
Precision	int8
Chunking Method	Token-based
Chunk Size	256 tokens
Overlap	Enabled (50 tokens)

Table 5.4: Indexing Configuration (Baseline)

Parameter	Value
Index Type	Flat
Backend	CPU
Distance Metric	Inner Product
Top-k Retrieved Chunks	3
Batch Size	2000
Index Cache	Disabled

Table 5.5: LLM Configuration (Baseline)

Parameter	Value
Model	Qwen2.5-0.5B
Backend	CPU
Precision	int8
Augmentation Method	Concatenation
Max Generated Tokens	50
KV Cache Window	4096
Prefill Chunk Size	1024
Decoding Strategy	Deterministic

This configuration serves as the reference point against which all subsequent experiments are compared.

5.3.2 Alternative Pipeline Baselines

To understand how individual components affect end-to-end performance, we construct several alternative baselines by modifying one parameter at a time while keeping the remaining configuration unchanged.

Embedding Variants

- Replace embedding model with *all-MiniLM-L12-v2* [44], which doubles the embedding dimensionality.
- Execute embeddings using the XNNPACK backend.
- Execute embeddings using the NNAPI backend.

Indexing Variants

- IVF index with $nlist = 109$ and $nprobe = 10$.
- HNSW index with $M = 12$, $ef\_construction = 80$, and $ef\_search = 24$.

LLM Variants

- Replace the model with Qwen2.5-1.5B.
- Use an int4-quantized version of the language model.
- Execute the LLM using XNNPACK backend.
- Execute the LLM using NNAPI backend.

Each configuration isolates a specific design decision, enabling us to measure how hardware acceleration, model size, quantization, and indexing strategies influence overall RAG performance.

In the following subsections, we present experimental results and analyze the trade-offs between task accuracy, latency, throughput, TTFT, and memory usage.

5.3.3 Throughput (Decoding Speed)

In this experiment, we evaluate decoding throughput, measured as the number of generated tokens per second. Unlike retrieval metrics, decoding throughput is measured *after* query embedding and document retrieval have completed. Consequently, embedding and indexing configurations do not influence this metric. The only relevant factors affecting decoding speed are modifications to the LLM itself. Figure 5.6 presents the decoding throughput for all LLM-related baseline configurations.

Baseline Performance. The baseline configuration using the Qwen2.5-0.5B model running on the CPU backend achieves a decoding throughput of 10.7 tokens/s. This serves as the reference point for comparing all other configurations.

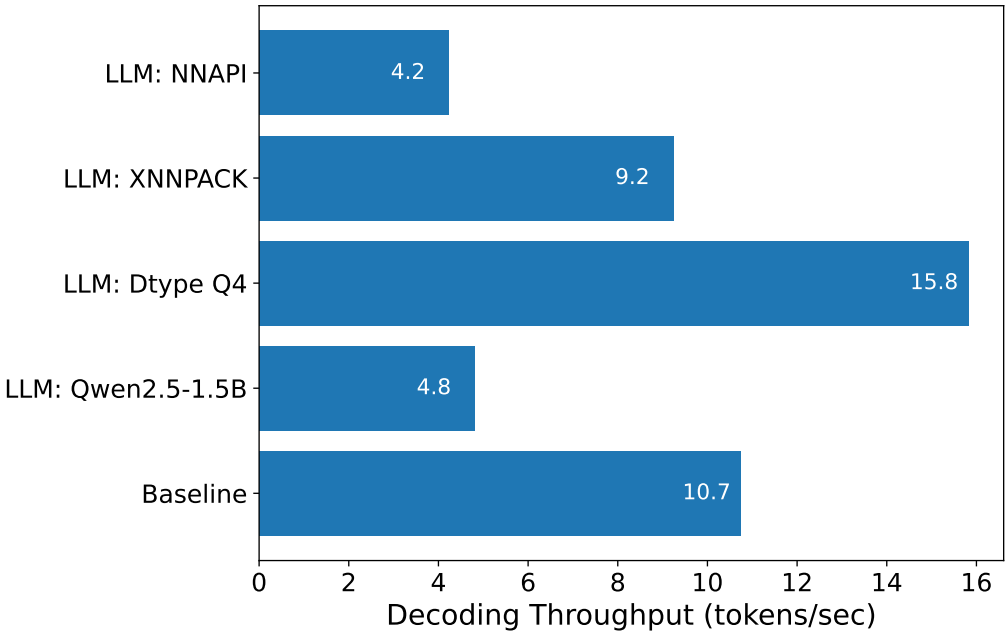


Figure 5.6: Decoding throughput (tokens/sec) for different LLM configurations.

Effect of Model Size. Increasing the model size to Qwen2.5-1.5B significantly reduces decoding performance, lowering throughput to approximately 4.8 tokens/s. This decrease is expected, as larger transformer models require substantially more matrix multiplications and memory bandwidth per generated token. The result highlights one of the primary challenges of deploying LLMs on mobile devices: model scaling directly translates into slower user response times.

Effect of Quantization. Quantization provides the largest performance improvement among all evaluated configurations. Using the int4 (Q4) quantized model increases throughput to approximately 15.8 tokens/s, representing nearly a 50% improvement over the baseline.

This improvement stems from reduced memory bandwidth requirements and faster arithmetic operations, both of which are critical bottlenecks in mobile inference workloads. These results demonstrate that aggressive quantization is one of the most effective optimizations for on-device RAG systems.

Effect of Execution Backend. We also evaluate alternative execution backends for the LLM. Running the model using XNNPACK achieves approximately **9.2 tokens/s**, which remains close to CPU performance. XNNPACK provides optimized CPU kernels for mobile processors, but in this setup it does not significantly outperform the default execution path. In contrast, the NNAPI backend performs substantially worse, achieving only **4.2 tokens/s**. This degradation is likely caused by additional communication overhead between the application and the Android neural network abstraction layer, combined with limited acceleration benefits for autoregressive decoding workloads. These results suggest that LLM workload is not very well optimized in XNNPACK and NNAPI backend.

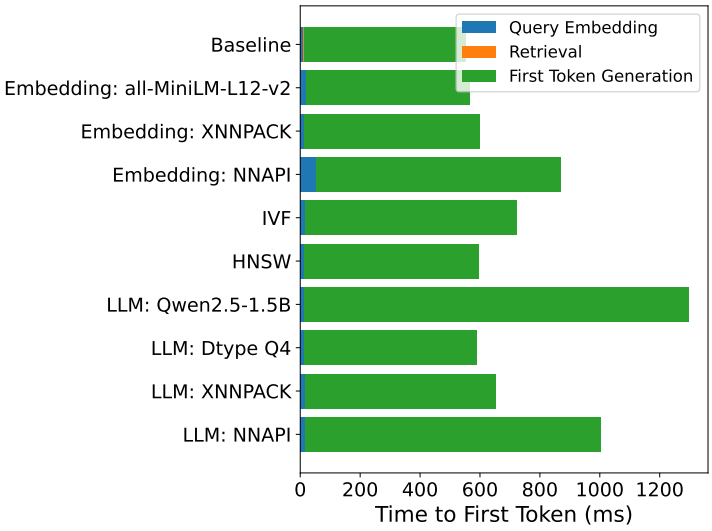


Figure 5.7: TTFT breakdown across different RAG configurations.

Summary. Overall, decoding throughput is determined almost entirely by LLM characteristics rather than earlier RAG pipeline components. Model size and quantization dominate performance, while backend selection plays a secondary role. Among all evaluated configurations, low-bit quantization provides the most significant improvement in decoding speed, making it a key enabler for efficient on-device RAG deployment.

5.3.4 TTFT Breakdown

In this experiment, we analyze the *Time-To-First-Token* (TTFT), which measures the latency between issuing a query and receiving the first generated token from the LLM. TTFT is one of the most important user-facing metrics for interactive RAG systems, as it determines the perceived responsiveness of the application.

We decompose TTFT into three stages:

- **Query Embedding:** encoding the user query into a vector representation.
- **Retrieval:** searching the index and selecting relevant document chunks.
- **First Token Generation (Prefill Phase):** processing the prompt and generating the first token of the response.

Figure 5.7 illustrates the contribution of each stage across all evaluated configurations.

Retrieval Cost. Across all configurations, the retrieval stage is effectively instantaneous and is barely visible in Figure 5.7. This confirms the results from the ANN benchmark: optimized vector search on-device introduces negligible latency compared to language model inference. Even when switching between Flat, IVF, and HNSW indices, retrieval contributes only a small fraction of total TTFT.

Query Embedding Cost. Query embedding is also very fast, typically contributing only a few milliseconds to the total latency. Changing the embedding model from all-MiniLM-L6-v2 to

all-MiniLM-L12-v2 has almost no observable impact on TTFT, demonstrating that embedding dimensionality is not a bottleneck in the end-to-end pipeline. The only noticeable deviation appears when using the NNAPI backend for embeddings, suggesting again the NNAPI backend is not well optimzied for embedding models.

Prefill Phase Dominance. The dominant component of TTFT is the **prefill phase**, i.e., the computation required by the LLM to process the entire prompt and generate the first token. As shown in Figure 5.7, nearly the entire TTFT is attributed to this stage.

Model-related changes strongly affect TTFT. Increasing the model size to Qwen2.5-1.5B dramatically increases TTFT to approximately 1300 ms, reflecting the higher computational cost of larger transformer layers. Conversely, using the int4-quantized model reduces TTFT to around 600 ms, demonstrating that quantization accelerates not only decoding throughput but also prompt processing.

Backend selection also plays a significant role. The XNNPACK backend slightly increases TTFT compared to the baseline, while the NNAPI backend results in substantially higher latency (around 1000 ms), again indicating that NNAPI is not well optimized for autoregressive transformer workloads in this setup.

Summary. Overall, TTFT is overwhelmingly dominated by the LLM prefill computation rather than retrieval or embedding stages. Retrieval and query encoding together contribute only a small fraction of the total latency, while model size, quantization, and inference backend determine the responsiveness of the system. These results highlight that optimizing the LLM inference path is the most effective strategy for reducing perceived latency in mobile RAG applications.

5.3.5 Mean Memory Usage

In this experiment, we analyze the mean memory increase observed during execution of the full RAG pipeline. Memory usage is measured as the average increase in RAM consumption relative to the idle application state while processing the benchmark workload. Figure 5.8 summarizes the memory footprint across all evaluated configurations.

Effect of Embedding Models. Embedding configuration has a moderate but noticeable impact on memory consumption. Switching from the baseline embedding model to the larger *all-MiniLM-L12-v2* increases memory usage from approximately 1000 MB to around 1500 MB. This behavior is expected because the embedding dimensionality doubles, requiring larger embedding tensors, intermediate buffers, and a larger vector index.

Effect of Indexing Methods. Changing the indexing method introduces additional memory overhead due to index-specific data structures. Both IVF and HNSW consume slightly more memory than the Flat index, reaching approximately 1150–1250 MB.

This overhead arises from auxiliary structures such as inverted lists in IVF and graph connections in HNSW. Nevertheless, the impact remains relatively small compared to the memory footprint of the language model.

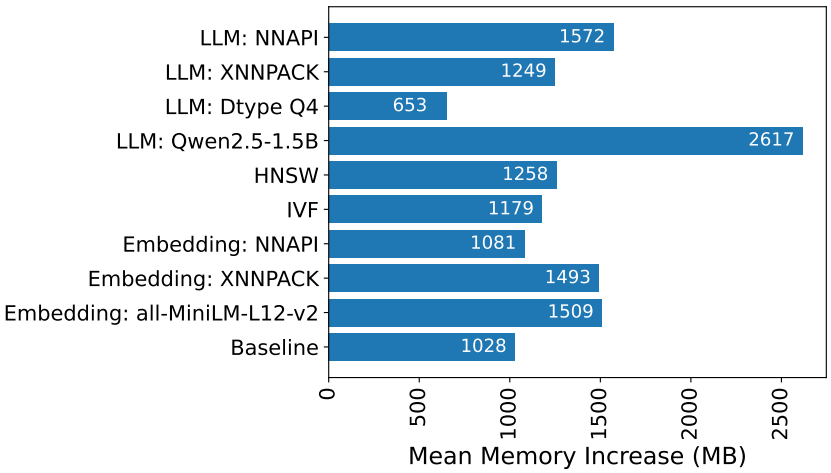


Figure 5.8: Mean memory increase for different RAG configurations.

Effect of LLM Configuration. The dominant factor affecting memory usage is the LLM. Increasing the model size to Qwen2.5-1.5B dramatically raises memory consumption to approximately 2600 MB, more than doubling the baseline requirement. This increase results from larger model weights, expanded activation tensors, and larger key–value caches used during generation.

Conversely, quantization significantly reduces memory requirements. The int4 (Q4) model lowers memory usage to roughly 650 MB, demonstrating that low-bit quantization effectively reduces both storage and runtime memory footprint.

Backend selection influences memory behavior. The NNAPI backend shows slightly higher memory usage compared to CPU and XNNPACK, likely due to additional runtime buffers and data transfers required by the Android neural network abstraction layer.

Summary. Overall, memory consumption in on-device RAG systems is primarily dominated by the LLM, while embedding models and indexing methods contribute secondary overheads. Larger embedding dimensions moderately increase memory usage, indexing structures add small additional costs, and quantization emerges as the most effective technique for reducing memory requirements. These findings reinforce the importance of model compression when deploying RAG pipelines on memory-constrained mobile devices.

5.3.6 Task Accuracy

In this experiment, we evaluate the downstream task performance of the complete RAG pipeline using the *Contains Accuracy* metric described in Section 4.4.2. This metric measures whether the generated answer contains the ground-truth answer span and serves as an indicator of factual correctness in generated responses. Figure 5.9 presents the accuracy results across all evaluated configurations.

Effect of Hardware Backends. Hardware-related modifications have negligible influence on task accuracy. Changing execution backends for either the embedding model or the LLM

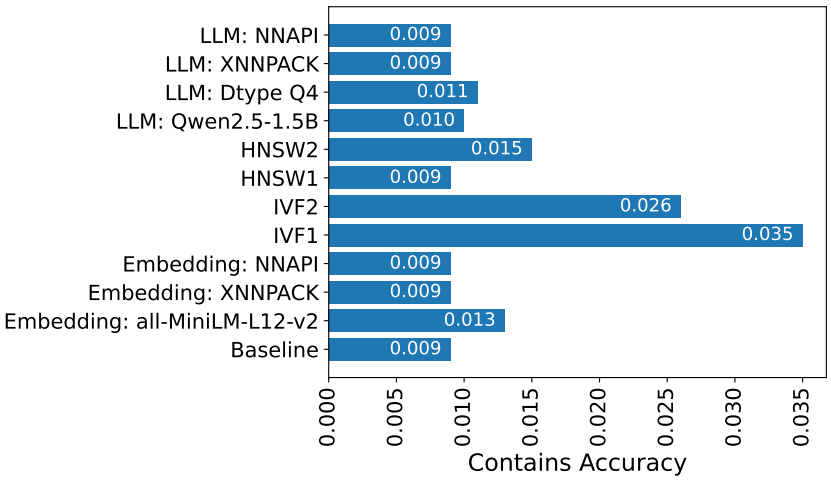


Figure 5.9: Task accuracy (Contains metric) across different RAG configurations.

(CPU, XNNPACK, or NNAPI) results in nearly identical accuracy values, all remaining close to the baseline performance of approximately **0.009**.

This behavior is expected because hardware backends primarily affect computational efficiency rather than model semantics. As long as numerical precision remains sufficiently stable, inference acceleration does not significantly alter generated answers.

Effect of Language Model Variants. Changes to the LLM configuration produce only minor variations in accuracy. Increasing model size to Qwen2.5-1.5B slightly improves accuracy to approximately **0.010**, while quantization (Q4) yields a comparable result of around **0.011**. These results suggest that, for this task setup, scaling the generator alone does not substantially improve factual correctness. This observation highlights an important property of RAG: generation quality is often limited more by retrieval quality than by model capacity.

Effect of Indexing Strategy. The most significant accuracy differences arise from the retrieval component. Switching from the Flat index baseline to approximate indexing methods leads to noticeable improvements. For example, HNSW achieves approximately **0.015** accuracy, while IVF configurations reach up to **0.035**, representing nearly a fourfold improvement over the baseline.

These results indicate that retrieval effectiveness plays a dominant role in downstream task performance. Improved indexing strategies allow the system to retrieve more useful contextual passages, which directly benefits answer generation.

Recall vs. Task Accuracy. Interestingly, higher retrieval recall does not always translate into higher task accuracy. While Flat index achieves the best recall among index types (see Section 5.2.1, this does not guarantee better task accuracy. In some cases, additional retrieved passages may introduce noise or distract the LLM, leading to limited or inconsistent gains in final task performance.

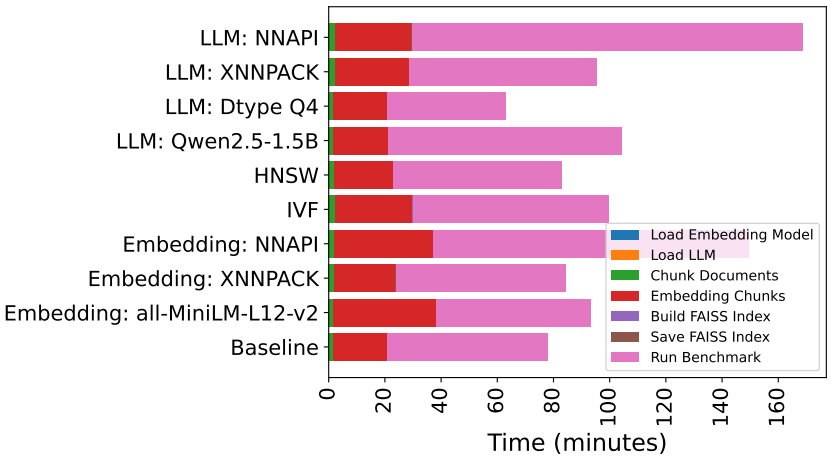


Figure 5.10: Total execution time breakdown of the end-to-end RAG benchmark.

Summary. Overall, retrieval strategy is an important factor in determining task accuracy in mobile RAG systems. Hardware choices mainly influence efficiency metrics such as latency and throughput, while indexing decisions determine the relevance of retrieved knowledge and therefore have the greatest impact on end-to-end accuracy.

5.3.7 Total Time Breakdown

In the final experiment, we analyze the total execution time of the full RAG pipeline by decomposing the benchmark into its major stages. Unlike previous experiments that focused on isolated performance metrics, this analysis provides a holistic view of where computation time is spent during end-to-end execution.

The evaluated stages include:

- Loading the embedding model,
- Loading the LLM,
- Document chunking,
- Embedding document chunks,
- Building and saving the FAISS index,
- Running the downstream benchmark (query processing and generation).

Figure 5.10 presents the runtime breakdown across all configurations.

Generation as the Primary Bottleneck. Across all configurations, the benchmark execution stage dominates the total runtime. This phase includes prompt construction, LLM prefill computation, and autoregressive decoding for each query. As shown in Figure 5.10, generation accounts for the majority of execution time, often exceeding 60–70% of the total benchmark duration.

Configurations using larger models, such as Qwen2.5-1.5B, exhibit the longest execution times, confirming that LLM inference remains the primary bottleneck for on-device RAG systems.

Impact of Embedding Computation. Embedding-related stages constitute the second largest contributor to runtime. In particular, embedding document chunks requires substantial computation because every document in the corpus must be encoded before indexing. Different embedding models noticeably affects total runtime; for example, running the experiments with the larger *all-MiniLM-L12-v2* model almost doubled the embedding time.

Chunking vs. Index Construction. An interesting observation is that document chunking requires more time than building the vector index itself. Although indexing involves constructing search structures, chunking processes the entire raw document corpus and performs token-level operations, making it computationally expensive despite its conceptual simplicity.

In contrast, FAISS index construction and serialization represent only a small fraction of the total runtime. This indicates that indexing is not a major bottleneck once embeddings have been generated.

Summary. Overall, the total runtime of the mobile RAG pipeline is dominated by language model generation, followed by embedding computation. Retrieval and indexing operations contribute comparatively little to overall execution time. These findings reinforce a central conclusion of this work: optimizing LLM inference and embedding computation provides the greatest opportunity for improving end-to-end performance of on-device RAG systems.

6

Discussion

This chapter discusses the broader implications of the work presented in this thesis. The goal is to interpret the findings from a system-level perspective and to reflect on what they reveal about deploying RAG pipelines on mobile devices. The discussion also highlights the role of MoRAGBench within the emerging landscape of on-device intelligent systems and outlines promising directions for future research.

6.1 On-Device RAG as a Systems Problem

One of the main insights of this thesis is that deploying RAG pipelines on smartphones is fundamentally a *system-level design problem* rather than purely a machine learning challenge. While much recent research focuses on improving LLM capabilities or retrieval algorithms independently, mobile deployment exposes strong inter-dependencies between all components of the pipeline.

Unlike server environments, mobile platforms operate under strict constraints in computation, memory capacity, thermal limits, and energy consumption. These constraints transform design priorities. Techniques that are effective in cloud deployments cannot be transferred directly to mobile devices without careful reconsideration of trade-offs.

The development of MoRAGBench demonstrates that understanding mobile RAG behavior requires holistic analysis across the entire pipeline. Embedding generation, indexing, retrieval, augmentation, and generation all contribute to overall performance. Optimizing a single component in isolation often shifts the bottleneck elsewhere in the system. Consequently, efficient on-device intelligence depends on coordinated optimization rather than individual model improvements.

6.2 The Role of Benchmarking in Mobile Intelligence

Another central contribution of this work is highlighting the importance of systematic benchmarking for on-device AI. Existing evaluation frameworks for RAG systems largely assume server-class hardware, abundant memory, and stable power availability. However, mobile devices introduce hardware heterogeneity and software fragmentation that significantly influence runtime behavior.

MoRAGBench addresses this gap by providing a modular framework capable of evaluating both individual pipeline components and complete end-to-end workflows directly on smartphones. This enables reproducible experimentation under realistic deployment conditions.

Beyond the specific implementation, the framework illustrates a broader methodological shift. As AI workloads move from centralized infrastructure to edge devices, benchmarking becomes a prerequisite for meaningful system optimization. Without detailed measurement and analysis, it is difficult to understand how architectural choices interact with mobile hardware characteristics.

In this sense, benchmarking frameworks play a similar role for on-device AI as profiling tools do in traditional systems engineering: they expose hidden performance limitations and guide informed design decisions.

6.3 General Observations from the Evaluation

Although Chapter 5 presented extensive experimental results, several general observations emerge when considering the evaluation as a whole.

First, the performance of mobile RAG pipelines is governed by trade-offs rather than single optimal configurations. Improvements in retrieval accuracy, model size, or decoding quality frequently introduce higher latency or memory usage. The most effective configurations therefore balance competing objectives instead of maximizing individual metrics.

Second, the results indicate that approximate solutions are essential for practical deployment. Exact computation, while theoretically optimal, often proves infeasible under mobile constraints. Efficient approximations enable systems to achieve sufficiently high quality while maintaining interactive performance.

Third, hardware acceleration plays an important but inconsistent role. Different execution backends can significantly influence runtime behavior, yet their effectiveness depends strongly on implementation maturity and device-specific support. This highlights that software optimization remains as important as hardware capability in the current mobile AI ecosystem.

Overall, the evaluation confirms that successful on-device RAG deployment requires viewing the pipeline as an integrated system shaped by resource constraints, hardware characteristics, and configuration choices.

6.4 Implications for Future On-Device Assistants

The findings of this thesis suggest that mobile RAG systems represent a promising direction toward privacy-preserving and personalized intelligent assistants. By executing retrieval and generation entirely on-device, such systems can operate without continuous reliance on cloud infrastructure, reducing latency and improving data privacy.

However, achieving this vision requires continued advances in efficiency, software tooling, and hardware-aware optimization. Frameworks such as MoRAGBench provide an important foundation by enabling researchers and developers to systematically study how intelligent pipelines behave under real-world mobile constraints.

As mobile processors continue integrating dedicated AI accelerators, the distinction between edge and cloud intelligence is likely to diminish. Understanding how to efficiently combine retrieval mechanisms with compact LLMs will therefore remain an important research direction.

6.5 Future Work

While MoRAGBench establishes a foundation for benchmarking mobile RAG pipelines, several opportunities exist for extending and improvingthe framework.

6.5.1 Optimized Execution Backends

The evaluation revealed that current mobile acceleration backends, including XNNPACK and NNAPI, are not yet fully optimized for modern LLM inference and embedding workloads. Although these frameworks provide hardware acceleration abstractions, their performance remains limited by generic kernel implementations.

Future work will focus on developing customized kernels tailored specifically to transformer-based workloads. Writing specialized operators for attention mechanisms, matrix multiplications, and embedding computation can enable better utilization of mobile CPUs, GPUs, and NPUs while reducing memory overhead and execution latency.

6.5.2 Extending Model and Dataset Support

Another important direction is expanding the range of supported experimental configurations. Future versions of MoRAGBench will incorporate additional datasets, LLMs, embedding models, indexing methods, and configuration options.

Supporting a wider variety of workloads will allow researchers to evaluate how different model architectures and retrieval strategies behave across diverse application scenarios. Increased configurability will also strengthen the framework's role as a general-purpose benchmarking platform for mobile RAG research.

6.5.3 Native Implementation and System Acceleration

Currently, parts of the inference orchestration logic are implemented in Kotlin for ease of integration within the Android ecosystem. However, high-performance mobile systems often benefit from lower-level implementations that provide tighter control over memory management and execution flow.

Future work therefore aims to migrate critical components from Kotlin to native C implementations. Native execution can reduce overhead introduced by language abstractions, improve cache locality, and enable more aggressive memory optimizations. This transition is expected to accelerate execution and provide finer control over resource usage, which is particularly important for long-running on-device AI workloads.

6.6 Summary

In summary, this thesis demonstrates that deploying RAG pipelines on mobile devices introduces challenges that extend beyond model performance alone. Efficient operation requires holistic system understanding, careful benchmarking, and hardware-aware optimization.

MoRAGBench contributes to this goal by enabling systematic evaluation of mobile RAG pipelines and by revealing the complex trade-offs that shape on-device intelligence. The insights gained through this work provide a basis for future research toward efficient, private, and fully on-device AI assistants.

7

Conclusion

The rapid advancement of large-scale language models has enabled powerful intelligent systems capable of understanding and generating natural language. At the same time, modern mobile devices have evolved into capable computing platforms equipped with specialized AI hardware and large local storage. These developments create an opportunity to move intelligent assistants from cloud-centric architectures toward fully on-device solutions.

This thesis addresses the challenge of evaluating and understanding RAG pipelines under mobile constraints. While RAG has become a dominant paradigm for improving factual grounding and contextual awareness of LLMs, most existing systems and evaluation methodologies assume server-class environments. Consequently, limited insight exists into how complete RAG pipelines behave when executed directly on smartphones.

To bridge this gap, this work introduces MoRAGBench, a modular benchmarking framework designed specifically for analyzing RAG systems on Android devices. The framework enables configurable experimentation across all pipeline components, including document chunking, embedding generation, indexing and retrieval, augmentation, and language model inference. By providing both component-level and end-to-end benchmarking modes, MoRAGBench allows systematic investigation of performance trade-offs under realistic mobile hardware conditions.

The results of this thesis demonstrate that deploying RAG pipelines on mobile devices is fundamentally a systems problem shaped by resource constraints and hardware heterogeneity. Efficient execution cannot be achieved through improvements to individual models alone. Instead, overall performance emerges from complex interactions between retrieval mechanisms, model inference, memory usage, and execution backends.

Several key insights arise from this work. First, approximate methods are essential for practical on-device retrieval, as exact computation quickly becomes infeasible under mobile limitations. Second, balanced configurations that jointly optimize latency, memory usage, and accuracy are more effective than maximizing individual performance metrics. Third, current mobile AI software stacks remain insufficiently optimized for modern transformer-based workloads, indicating substantial opportunities for systems-level optimization.

The broader implication of this work is that fully on-device intelligent assistants are becoming increasingly feasible. By combining local retrieval with compact LLMs, future systems may provide personalized, privacy-preserving assistance without continuous dependence on cloud infrastructure. Achieving this vision will require continued advances in hardware-aware optimization, efficient model design, and integrated system engineering.

In conclusion, this thesis demonstrates that systematic benchmarking is a critical step toward efficient mobile RAG deployment. Through the design and evaluation of MoRAGBench, this work advances the understanding of how retrieval and generation pipelines operate under real mobile constraints and provides a foundation for future research on scalable, efficient, and fully on-device AI systems.

Bibliography

[1] Raviteja Anantha, Tharun Bethi, Danil Vodianik, and Srinivas Chappidi. "Context tuning for retrieval augmented generation". In: *arXiv preprint arXiv:2312.05708* (2023).

[2] Anastasiya Zharovskikh. *Best applications of large language models*. Accessed: 2026-01-31. 2023. URL: https://indatalabs.com/blog/large-language-model-apps.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms". In: *Information Systems* 87 (2020), p. 101374.

[4] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. "On the dangers of stochastic parrots: Can language models be too big?" In: *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. 2021, pp. 610–623.

[5] Dongqi Cai, Shangguang Wang, Chen Peng, Zeling Zhang, Zhenyan Lu, Tao Qi, Nicholas D Lane, and Mengwei Xu. "Ubiquitous memory augmentation via mobile multimodal embedding system". In: *Nature Communications* 16.1 (2025), p. 5339.

[6] CellStrat. *Real-World Use Cases for Large Language Models (LLMs)*. Accessed: 2026-01-31. 2023. URL: https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2.

[7] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. "Benchmarking large language models in retrieval-augmented generation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 16. 2024, pp. 17754–17762.

[8] Daivi. *7 Top Large Language Model Use Cases And Applications*. Accessed: 2026-01-31. 2024. URL: https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887.

[9] Yunbin Deng. "Deep learning on mobile devices: a review". In: *Mobile Multimedia/Image Processing, Security, and Applications 2019* 10993 (2019), pp. 52–66.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[11] Nawaz Dhandala. *How to Create HNSW Index*. https://oneuptime.com/blog/post/2026-01-30-vector-db-hnsw-index/view. 2026.

[12] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. "The Faiss library". In: (2024). arXiv: 2401.08281 [cs.LG].

[13] Zhangyin Feng, Xiaocheng Feng, Dezhi Zhao, Maojin Yang, and Bing Qin. "Retrieval-generation synergy augmented large language models". In: *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2024, pp. 11661–11665.

[14] Robert Friel, Masha Belyi, and Atindriyo Sanyal. "Ragbench: Explainable benchmark for retrieval-augmented generation systems". In: *arXiv preprint arXiv:2407.11005* (2024).

[15] Google. *Android Neural Networks API (NNAPI)*. https://developer.android.com/ndk/guides/neuralnetworks. Accessed: 2026-02-04. 2025.

[16] Google. *XNNPACK: High-efficiency floating-point neural network inference operators*. https://github.com/google/XNNPACK. Accessed: 2026-02-04. 2026.

[17] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. "Retrieval augmented language model pre-training". In: *International conference on machine learning*. PMLR. 2020, pp. 3929–3938.

[18] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. "An empirical analysis of compute-optimal large language model training". In: *Advances in neural information processing systems* 35 (2022), pp. 30016–30030.

[19] Gautier Izacard and Edouard Grave. "Leveraging passage retrieval with generative models for open domain question answering". In: *Proceedings of the 16th conference of the european chapter of the association for computational linguistics: main volume*. 2021, pp. 874–880.

[20] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, et al. "MLPerf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device AI". in: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 352–369.

[21] Herve Jegou, Matthijs Douze, and Cordelia Schmid. "Product quantization for nearest neighbor search". In: *IEEE transactions on pattern analysis and machine intelligence* 33.1 (2010), pp. 117–128.

[22] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547.

[23] Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. "triviaqa: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension". In: *arXiv e-prints*, arXiv:1705.03551 (2017), arXiv:1705.03551. arXiv: 1705.03551.

[24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. "Scaling laws for neural language models". In: *arXiv preprint arXiv:2001.08361* (2020).

[25] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. "Dense Passage Retrieval for Open-Domain Question Answering." In: *EMNLP (1)*. 2020, pp. 6769–6781.

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[28] Shaobo Li, Eric Zhou, Yuan Xu, Nelson Mimura Gonzalez, Daniel Waddington, Swaminathan Sundararaman, Hubertus Franke, and Jian Huang. "RAGPerf: An End-to-End Benchmarking Framework for Retrieval-Augmented Generation Systems". In: (2026). URL: %5Bhttps://github.com/platformxlab/ragperf%5D(https://github.com/platformxlab/ragperf).

[29] Yu A Malkov and Dmitry A Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs". In: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018), pp. 824–836.

[30] Ben Mann, Nick Ryder, Melanie Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. "Language models are few-shot learners". In: *arXiv preprint arXiv:2005.14165* 1.3 (2020), p. 3.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26 (2013).

[32] Anthony Moi and Nicolas Patry. *HuggingFace's Tokenizers*. Version 0.13.4. Apr. 2023. URL: https://github.com/huggingface/tokenizers.

[33] Abdelrahman Abdallah Bhawna Piryani Jamshid Mozafari and Mohammed Ali Adam Jatowt. "How good are llm-based rerankers? an empirical analysis of state-of-the-art reranking models". In: (2025).

[34] Rodrigo Nogueira and Kyunghyun Cho. *Passage Re-ranking with BERT*. 2020. arXiv: 1901.04085 [cs.IR]. URL: https://arxiv.org/abs/1901.04085.

[35] ONNX Runtime developers. *ONNX Runtime*. Nov. 2018. URL: https://github.com/microsoft/onnxruntime.

[36] Taehwan Park, Geonho Lee, and Min-Soo Kim. "MobileRAG: A Fast, Memory-Efficient, and Energy-Efficient Method for On-Device RAG". in: *arXiv preprint arXiv:2507.01079* (2025).

[37] Pranav Rajpurkar, Robin Jia, and Percy Liang. "Know What You Don't Know: Unanswerable Questions for SQuAD". in: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Iryna Gurevych and Yusuke Miyao. Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 784–789. DOI: 10.18653/v1/P18-2124. arXiv: 1806.03822 [cs.CL]. URL: https://aclanthology.org/P18-2124.

[38] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Ed. by Jian Su, Kevin Duh, and Xavier Carreras. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264. arXiv: 1606.05250 [cs.CL]. URL: https://aclanthology.org/D16-1264.

[39] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 2019, pp. 3982–3992.

[40] Facebook Research. *FAISS Wiki*. https://github.com/facebookresearch/faiss/wiki/Faiss-indexes. 2026.

[41] Gerard Salton and Christopher Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information processing & management* 24.5 (1988), pp. 513–523.

[42] Gerard Salton, Edward A Fox, and Harry Wu. "Extended boolean information retrieval". In: *Communications of the ACM* 26.11 (1983), pp. 1022–1036.

[43] Payel Santra, Madhusudan Ghosh, Debasis Ganguly, Partha Basuchowdhuri, and Sudip Kumar Naskar. "HF-RAG: Hierarchical Fusion-based RAG with Multiple Sources and Rankers". In: *arXiv preprint arXiv:2509.02837* (2025).

[44] Sentence-Transformers. *all-MiniLM-L12-v2: Sentence Embedding Model*. https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2. Accessed: 2026-02-04. 2026.

[45] Sentence-Transformers. *all-MiniLM-L6-v2: Sentence Embedding Model*. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Accessed: 2026-02-04. 2026.

[46] Michael Shen, Muhammad Umar, Kiwan Maeng, G Edward Suh, and Udit Gupta. "Hermes: Algorithm-System Co-design for Efficient Retrieval-Augmented Generation At-Scale". In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 2025, pp. 958–973.

[47] Spotify. *Voyager: A library for performing fast approximate nearest-neighbor searches on an in-memory collection of vectors*. https://github.com/spotify/voyager. 2026.

[48] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. "Empowering edge intelligence: A comprehensive survey on on-device ai models". In: *ACM Computing Surveys* 57.9 (2025), pp. 1–39.

[49] Yichuan Wang, Zhifei Li, Shu Liu, Yongji Wu, Ziming Mao, Yilong Zhao, Xiao Yan, Zhiying Xu, Yang Zhou, Ion Stoica, et al. "LEANN: A Low-Storage Vector Index". In: *arXiv preprint arXiv:2506.08276* (2025).

[50] Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. "Multi-passage BERT: A globally normalized BERT model for open-domain question answering". In: *arXiv preprint arXiv:1908.08167* (2019).

[51] Walter F Wiggins and Ali S Tejani. "On the opportunities and risks of foundation models for natural language processing in radiology". In: *Radiology: Artificial Intelligence* 4.4 (2022), e220119.

[52] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. "Empowering 1000 tokens/second on-device llm prefilling with mllm-npu". In: *arXiv e-prints* (2024), arXiv–2407.

[53] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering". In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2018.

[54] Tian Yu, Shaolei Zhang, and Yang Feng. *Auto-RAG: Autonomous Retrieval-Augmented Generation for Large Language Models*. 2024. arXiv: 2411.19443 [cs.CL]. URL: https://arxiv.org/abs/2411.19443.

[55] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen. "A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities". In: *Proceedings of the IEEE* 110.3 (2022), pp. 334–354.