Diploma Project

RescueAid: Using On-Device Vision-Language Models for Emergency Scene Description and Semantic Retrieval

Eleftherios Kalligiannakis

University of Cyprus
Department of Computer Science

Department of Computer Science

May 2026

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

Department of Computer Science

RescueAid: Using On-Device Vision-Language Models for Emergency Scene Description and Semantic Retrieval

Eleftherios Kalligiannakis

Advisor:

Dr. Panayotis Kolios

The Thesis was submitted in partial fulfillment of the requirements for obtaining the Computer Science degree of the Department of Computer Science of the University of Cyprus

May 2026

Declaration

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Panayotis Kolios, as well as to the faculty and staff of the department, for providing the academic environment and resources necessary for the completion of my studies.

A special thank you to my friends and colleagues for always motivating me and pushing me to grow and improve every day.

Finally, I would like to express my deepest appreciation to my family for their endless patience, encouragement, and unwavering support.

ABSTRACT

Emergency response scenarios demand rapid visual scene understanding under conditions where cloud connectivity may be unavailable, unreliable, or inappropriate due to privacy concerns. This thesis presents RescueAid, a fully offline mobile system that leverages on-device vision-language models to generate structured descriptions of emergency scenes from continuous camera input, encode them as semantic embeddings, and enable natural language retrieval over stored observations. The system is implemented as a React Native application with native Kotlin modules interfacing Google's LiteRT runtime, and operates entirely on-device without requiring external servers.

Four models from the Gemma family (Gemma 3n-E2B, Gemma 3n-E4B, Gemma 4-E2B, Gemma 4-E4B) were evaluated on a custom dataset of annotated first-responder images across police and rescue scenarios. Results reveal a consistent trade-off between descriptive richness and factual conservatism: models producing more detailed descriptions achieve better retrieval performance but higher hallucination rates, while more conservative models reduce errors at the cost of retrieval effectiveness.

RescueAid demonstrates that retrieval-augmented multimodal apps can be effectively deployed on mobile hardware, offering a practical foundation for privacy-preserving AI systems.

Keywords: On-Device AI, Retrieval-Augmented Generation, Vision-Language Models

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Important abbreviations used in the text are briefly explained below.

VLM	Vision-Language Model
RAG	Retrieval-Augmented Generation
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
TFLite	TensorFlow Lite
KV	Key-Value (used in KV-cache)

1 Introduction

1.1 Background and Motivation

In emergency situations, it is vital to understand dynamic environments, which are often visually complex, quickly and correctly. Whether it's a wearable camera, a mobile device or on-site observations, first responders often need visual information to determine the situation, whether it's an accident, medical emergency, fire, or public safety incident. In these settings, it is essential to interpret visual information quickly to make good decisions.

In recent years, with the development of multimodal machine learning, especially vision-language models (VLMs), automatic generation of image description text has become possible. Most current systems, however, are based on cloud computing, which poses problems like communication latency, network dependency, and privacy issues with sensitive visual information like body-worn camera footage.

Meanwhile, advancements in edge computing and the efficiency of large language models have made it more possible to deploy compact models directly on mobile devices. This provides the potential for locally processing visual data in real world environments faster and more privately, through multimodal inference.

The motivation for this thesis is to investigate the potential of on-device multimodal language models for structured scene understanding in emergency situations and to assess their ability to produce meaningful and reliable text descriptions of visual scenes given the computational constraints.

In addition, it examines whether these descriptions can be used as good semantic representations when embedded in embedding spaces, allowing to retrieve emergency-related visual information meaningfully based on similarity.

1.2 Problem Statement

Modern vision-language models can generate detailed descriptions of images, but most of these systems rely on cloud based processing. This introduces limitations when used in real world emergency scenarios.

One major issue is latency and dependency on network connectivity. In emergency situations, communication can be slow, unavailable, or unreliable, making cloud-based solutions impractical. Another concern is privacy, since emergency-related images or body-worn camera footage

may contain sensitive information that should not be transmitted to external servers.

In addition, most existing systems are not designed for deployment on mobile or edge devices with limited computational resources. As a result, there is a gap between state-of-the-art multimodal models and their practical use on devices such as smartphones.

Finally, while image captioning models can generate textual descriptions of scenes, there is limited exploration of how these descriptions behave when used as semantic representations for retrieval tasks in an on-device setting.

This thesis addresses these issues by implementing and evaluating an on-device multimodal pipeline that generates scene descriptions from images and encodes them into embeddings for similarity based retrieval.

1.3 Objectives

- **Design and implementation of an on-device visual RAG pipeline:** Develop a fully offline mobile system that processes image frames into structured semantic representations and enables retrieval-augmented querying over captured scenes.
- **Scene encoding and retrieval:** Use lightweight vision-language models to generate natural language scene descriptions, encode them into dense embeddings, and store them in a local database for cosine-similarity search.
- **Evaluation framework:** Evaluate the system on annotated police and rescue imagery using retrieval accuracy, semantic alignment, and error analysis metrics, including missing entities and hallucinations.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

- **Chapter 1** introduces the research problem, motivation, objectives, and structure of the thesis.
- **Chapter 2** reviews related work in multimodal learning, on-device artificial intelligence, embedding-based retrieval, and evaluation methods for multimodal systems.
- **Chapter 3** presents the system architecture, including the overall design and key components of the application.
- **Chapter 4** describes the evaluation methodology, experimental setup and results.

- **Chapter 5** concludes the thesis by summarizing the main findings, discussing limitations and outlining directions for future work.

1.5 Summary

This chapter introduced the motivation for the thesis, focusing on the challenges of interpreting visual data in emergency response scenarios and the limitations of existing cloud-based multimodal systems. It defined the problem of deploying such systems under privacy and resource constraints and motivated the need for on-device solutions.

The chapter also outlined the main objectives of the work, including the development of an on-device multimodal pipeline, structured scene description generation, embedding-based representation, and semantic retrieval. Finally, the structure of the remainder of the thesis was presented to guide the reader through the following chapters.

2 Literature Review

2.1 Introduction

This chapter reviews existing research related to vision-language models, on-device artificial intelligence, embedding-based representations and their applications in emergency scene understanding. The goal is to establish the theoretical background of the proposed system and identify limitations in current approaches.

2.2 Early Image Captioning Approaches

Early image captioning approaches combined convolutional neural networks (CNNs) for visual feature extraction with recurrent neural networks (RNNs) for sequence generation. In these architectures, CNNs were used to encode images into fixed-dimensional feature representations, which were then passed to an RNN decoder to generate natural language descriptions [2].

One of the foundational models in this area, Show and Tell, introduced an end-to-end CNN-RNN framework for image caption generation, demonstrating that neural networks could directly map visual content to textual output [3]. This work established the baseline architecture for neural image captioning systems.

Although these models demonstrated strong initial performance, CNN–RNN-based approaches are limited in their ability to capture long-range dependencies and global contextual relationships in language. In addition, their sequential decoding process restricts parallelization during both training and inference, limiting scalability compared to more recent transformer-based architectures [2].

2.3 Vision-Language Models

Vision-Language Models (VLMs) are multimodal systems designed to learn joint representations of visual and textual data. By aligning image and language information within a shared semantic space, VLMs enable a broad range of tasks including image captioning, visual question answering, and cross-modal retrieval [2].

2.3.1 Transformer-Based Vision-Language Models

The limitations of early CNN-RNN captioning systems motivated a shift towards transformer-based architectures capable of learning richer cross-modal representations. Modern VLMs are predominantly pre-trained on large-scale image-text datasets, enabling them to learn joint multimodal representations that generalise across multiple vision-language tasks.

Unlike CNN-RNN architectures, transformer-based models replace sequential recurrent processing with self-attention mechanisms, allowing the model to directly attend to all positions in an input sequence simultaneously. This removes the bottleneck of sequential decoding and enables full parallelization during training, significantly improving scalability. Furthermore, the self-attention mechanism allows transformers to capture long-range dependencies between visual regions and language tokens more effectively than recurrent networks, which struggled to maintain context over long sequences.

In the context of vision-language modeling, transformers are applied to both visual and textual modalities. Visual information is typically extracted using either a pre-trained object detector or a Vision Transformer (ViT), which processes images as sequences of patches rather than relying on convolutional feature maps. This unified patch-based representation allows the same attention mechanisms used for text to be applied directly to visual inputs, enabling tighter cross-modal integration. The result is a more flexible and expressive architecture capable of handling diverse vision-language tasks within a single unified framework [2].

2.3.2 Limitations of Vision-Language Models

Despite their strong performance, VLMs typically require substantial computational resources and are predominantly deployed in cloud-based environments. This introduces practical challenges including increased inference latency, dependency on network connectivity, and privacy concerns when processing sensitive visual data [2].

In addition, many state-of-the-art VLMs are designed for large-scale general-purpose reasoning rather than real-time decision support, resulting in inefficiencies when applied to time-critical scenarios. Their reliance on high-capacity architectures further limits their direct deployment on resource-constrained devices such as mobile platforms.

2.3.3 On-Device Vision-Language Models

Recent developments have focused on designing VLMs that are capable of running directly on edge devices without relying on cloud infrastructure. Google's Gemma 3n represents a significant step in this direction, introducing a mobile-first multimodal architecture that supports

text, image, audio, and video inputs while operating with an effective memory footprint of as little as 2GB [4]. Its novel MatFormer architecture enables elastic inference through nested sub-models, allowing dynamic trade-offs between performance and computational cost on resource-constrained hardware. Building on this, Gemma 4 extends these capabilities further with improved reasoning, a larger context window, and native support for structured agentic tasks, while retaining the efficient on-device design principles of its predecessor [5]. Despite these advancements, such models still face limitations in latency and resource usage when deployed on mobile hardware.

2.4 On-Device AI and Edge Computing

Edge AI refers to the deployment of machine learning models directly on resource-constrained devices such as smartphones, embedded systems, and wearables, rather than relying on remote cloud infrastructure. This paradigm has gained increasing importance in the last few years due to its ability to reduce inference latency, improve data privacy, and enable functionality in offline or connectivity-limited environments [6].

Despite these advantages, deploying LLMs on edge devices introduces significant challenges. Mobile and embedded hardware is constrained in terms of computational capacity, memory, and energy consumption.

To support on-device LLM inference, specialized runtimes have emerged that enable efficient use models on consumer hardware. These include lightweight C/C++ based runtimes such as llama.cpp, as well as platform-integrated solutions such as LiteRT-LM, which extends the LiteRT ecosystem to support generative language models within the broader Google AI Edge framework. Together, these approaches reflect a shift towards enabling practical LLM inference directly on mobile and edge devices under strict resource constraints.

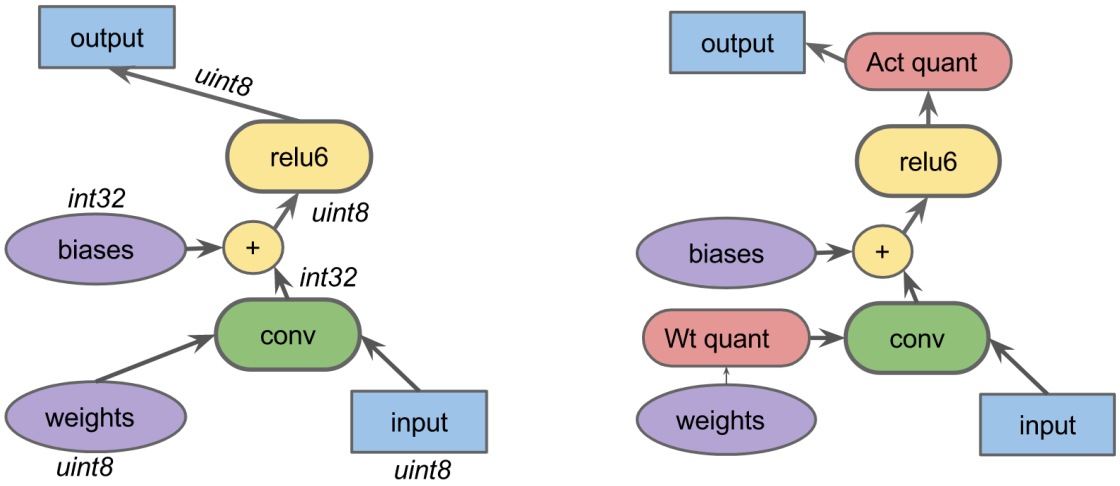
Overall, while these runtimes make on-device inference increasingly feasible, significant trade-offs remain between model size, latency, and output quality. This motivates further research into optimization techniques that enable efficient multimodal reasoning under strict resource constraints, which is discussed in the next section.

2.5 Model Optimization Techniques for Edge Deployment

To address the limitations of deploying LLMs on edge devices, model compression techniques are commonly used to reduce computational and memory requirements while maintaining acceptable performance. In this work, quantization is adopted as the primary optimization method to enable efficient on-device inference.

2.5.1 Quantization

Quantization is a model compression technique that reduces the numerical precision of neural network parameters, typically by converting weights and activations from high-precision floating-point representations (e.g., FP32) to lower-bit formats such as INT8 or INT4. This reduction in precision decreases the memory footprint of the model and enables more efficient computation, particularly on hardware that supports low-precision arithmetic.



(a) Quantized inference

(b) Training with simulated quantization

Figure 2.1: The distinction between post-training quantized inference and quantization-aware training, highlighting how quantization can either be applied after training or integrated into the training process to mitigate accuracy degradation [1].

As a result, quantized models often achieve significant reductions in memory usage and inference latency while maintaining performance comparable to their full-precision counterparts. When combined with techniques such as quantization-aware training, models can be trained to be robust to reduced precision, resulting in minimal degradation in accuracy [1].

In the context of large language models, recent studies have shown that aggressive quantization strategies can still preserve model quality under certain conditions, enabling more efficient deployment in resource-constrained environments [7].

2.6 Embedding-Based Representations and Retrieval

Embedding-based representations are a fundamental technique in modern natural language processing, enabling textual data to be encoded into dense, fixed-dimensional vectors that capture semantic meaning. Unlike sparse representations such as bag-of-words or TF-IDF, dense em-

beddings place semantically similar texts close together in a high-dimensional vector space, allowing similarity to be measured geometrically rather than through exact lexical overlap [8].

2.6.1 Text Embeddings

Text embedding models map variable-length textual inputs to fixed-size dense vectors in a continuous semantic space. Models such as Sentence-BERT [8] introduced the use of siamese and triplet network architectures built on top of transformer encoders, producing sentence-level embeddings that are well-suited for semantic similarity tasks. These models are trained using contrastive objectives, encouraging embeddings of semantically related sentences to cluster together while pushing dissimilar sentences apart.

In the system developed in this thesis, text embeddings are used to encode generated scene descriptions into vector representations. This enables the system to compare and retrieve descriptions based on semantic content rather than surface-level wording, which is particularly important in scenarios where multiple valid descriptions of the same scene may differ in phrasing. This representation forms the basis for efficient similarity search over previously observed events.

2.6.2 Semantic Similarity Search

Once text has been encoded into embedding vectors, similarity search enables efficient retrieval of semantically related content from a database. In this system, cosine similarity is used as the ranking function to compare query embeddings with stored embeddings and retrieve the most relevant results.

This approach forms the basis of vector-based retrieval systems, where items are ranked according to semantic proximity rather than lexical overlap. Compared to keyword-based search, embedding-based retrieval is more robust to variation in phrasing and can match conceptually similar queries even in the absence of shared terms [9].

2.6.3 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a framework that combines semantic retrieval with generative language models. In a standard RAG pipeline, a query is first encoded into an embedding, which is used to retrieve relevant documents or passages from a vector store. The retrieved context is then provided to a generative model as additional input, grounding its output in retrieved information rather than relying solely on parametric knowledge [9].

The system developed in this work adopts a lightweight, on-device variant of this approach. User queries are embedded and used to retrieve semantically similar previously generated scene

descriptions using cosine similarity. These retrieved records are then used to support and contextualise the model's responses, enabling grounded question answering over past events. Unlike traditional RAG systems, this approach operates entirely on-device and is designed for efficiency rather than large-scale document retrieval.

2.6.4 Lightweight Embedding Models for Edge Deployment

Deploying embedding models on resource-constrained devices introduces similar challenges to those encountered with generative vision-language models. While large embedding models provide strong semantic performance, their memory footprint and computational cost can make them unsuitable for real-time on-device use.

Recent work has therefore focused on developing smaller and more efficient embedding models that retain semantic quality while reducing inference cost, often through techniques such as quantisation and model distillation.

In this system, a lightweight transformer-based embedding model is used to encode generated scene descriptions. The model is deployed in LiteRT format.[10]

2.7 Overview of Evaluation Metrics for Multimodal Systems

Evaluating multimodal systems involves a combination of metrics depending on the task. For image captioning, commonly used metrics include BLEU, METEOR, and CIDEr, which measure the similarity between generated and reference captions.

For retrieval-based systems, evaluation focuses on ranking quality using metrics such as Recall@K, precision, and cosine similarity. These metrics assess how effectively the system retrieves relevant results from a database based on semantic similarity.

In real-time systems, additional performance indicators such as inference latency, throughput, and memory usage are also critical. These metrics are particularly important in on-device scenarios, where computational resources are limited and responsiveness is essential.

2.7.1 Text Generation Metrics

Text generation metrics are commonly applied in image captioning and vision-language tasks for measuring the similarity between generated sentences and one or more reference sentences written by humans. These measures are generally based on n-gram matching or consensus-based scoring, and offer a numerical assessment of textual similarity. But as noted in previous reviews of image captioning evaluation, these metrics can be insufficient to capture semantic correctness and may not align with human preference in open-ended generation tasks [11, 12].

2.7.1.1 BLEU

The BLEU metric is one of the first and most popular metrics for evaluating generated text. It is based on the precision of n-gram matches between a candidate sentence and reference sentences, often with a brevity penalty to penalise short sentences. While BLEU is good at measuring the surface-level overlap, it is sensitive to lexical choices and doesn't account for semantic equivalence of different expressions.

2.7.1.2 METEOR

METEOR extends BLEU by using other matching methods such as stemming and synonym matching. It aligns words between candidate and reference sentences, and penalises word order. This leads to better agreement with human judgement than BLEU, especially in tasks with natural language variation.

2.7.1.3 CIDEr

CIDEr (Consensus-based Image Description Evaluation) is tailored for image captioning. It uses TF-IDF weighted n-grams to calculate similarity, giving more weight to more informative words that are relevant to the image. CIDEr measures the agreement between a generated caption and a set of human references, making it more suitable for image captioning datasets that contain multiple valid descriptions of an image.

Despite their popularity, these metrics are based on surface-level n-gram matching and hence have difficulty assessing semantic equivalence when alternate valid descriptions are provided [12].

2.7.2 Semantic Similarity Metrics

Semantic similarity metrics are used to evaluate the degree of meaning overlap between two textual or multimodal representations. Cosine similarity over embedding representations is used as an evaluation metric to assess alignment between generated outputs and ground truth references.

Unlike lexical metrics such as BLEU or CIDEr, which rely on n-gram overlap, embedding-based similarity captures semantic relationships in a continuous vector space, enabling comparison of textual content even when wording differs [8]. This makes it particularly suitable for evaluating open-ended generation tasks in vision-language systems.

Given two embedding vectors $\mathbf{u}$ and $\mathbf{v}$, cosine similarity is defined as:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$$

2.7.3 Retrieval Evaluation Metrics

Retrieval performance is evaluated using ranking-based metrics that measure how effectively relevant items are surfaced within the top-K results.

Precision@K measures the proportion of retrieved items in the top-K results that are relevant to the query:

$$\text{Precision@K} = \frac{\text{relevant items in top K}}{K}$$

Recall@K measures the proportion of all relevant items that are successfully retrieved within the top-K results:

$$\text{Recall@K} = \frac{\text{retrieved relevant items in top K}}{\text{total relevant items}}$$

These metrics are used in this work to evaluate embedding-based image retrieval, where queries are matched against indexed scene descriptions using cosine similarity. They provide complementary views of performance, with precision capturing ranking accuracy and recall capturing coverage of relevant results[13].

2.7.4 Factual Consistency and Hallucination Metrics

Factual consistency metrics evaluate whether generated outputs are supported by the available input context or ground truth references. In vision-language and generative systems, a common failure mode is hallucination, where the model generates information that is not grounded in the input image or reference data [14].

Hallucination refers to the generation of objects, attributes, or actions that are not present in the source input. This issue has been widely studied in image captioning, where models may produce object words that are not visually present in the scene [14]. It is particularly important in safety-critical or real-world applications where incorrect descriptions may lead to misleading interpretations.

In this work, a lightweight heuristic-based approach is used to estimate factual consistency. This design is inspired by prior work on object hallucination in image captioning, which proposes overlap-based evaluation metrics such as CHAIR [14]. Entity-level and action-level tokens are extracted from both generated and reference texts, and overlap-based precision, recall, and F1 scores are computed. Additionally, hallucination rate is measured as the proportion of generated entities or actions that do not appear in the ground truth.

Although such rule-based methods do not capture all aspects of semantic correctness, they pro-

vide an interpretable and computationally efficient approximation of factual alignment. Prior work has shown that such metrics are useful for analyzing hallucination behavior in captioning models and complement standard evaluation metrics [14].

2.8 Ethics and Privacy

The development and deployment of vision-language systems in emergency response contexts raise important ethical and privacy concerns. These systems process visual data that may include people, private property, or incident locations. Because of this, careful handling of data and prevention of misuse are important design requirements.

One key concern is data privacy. Images and sensor data collected from real-world environments may contain personally identifiable information. To reduce this risk, data should be anonymized when possible, and access to raw data should be limited to trusted parts of the system. In this work, on-device processing is used, which helps improve privacy by eliminating the need to transmit sensitive data to external servers.

Finally, ethical use of such systems requires transparency and accountability. The outputs of AI models should be treated as suggestions to support human decision-making, rather than final or fully reliable answers. This helps ensure that responsibility remains with human operators, especially in high-stakes situations [15].

3 System Design

3.1 System Overview

RescueAid is an offline-first mobile intelligence system designed to transform real-world visual input into structured semantic memory for later retrieval and reasoning. The system is built around a continuous closed-loop pipeline, as shown in Figure 3.1.

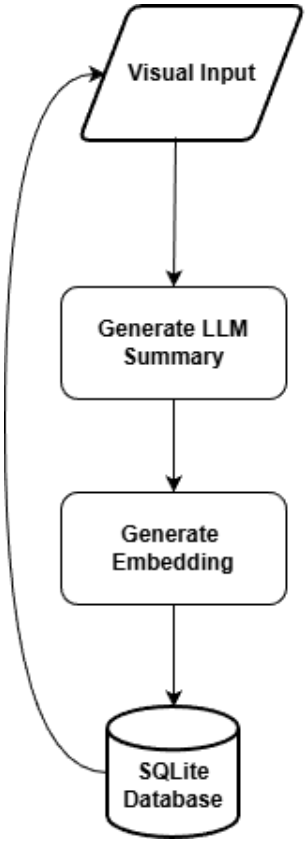


Figure 3.1: Inference pipeline: visual input is processed through LLM-based summarization, embedding generation, and stored in a local SQLite database for semantic retrieval.

Rather than relying on cloud services, all computation, including inference, embedding generation, and retrieval is performed entirely on-device. The system is implemented as a component-based architecture within a single React Native runtime, where core functionality is divided between Services written in Typescript and Kotlin-based native modules.

- **Native Modules**: Kotlin-based modules that interface the React Native runtime with Google's Edge GenAI frameworks. Built on LiteRT-LM and MediaPipe GenAI RAG, they enable fully offline language model inference and embedding generation [16, 17].

- **Service Layer**: A set of TypeScript modules responsible for the key functionality of the application.

  Most modules are implemented as class-based services that encapsulate long-lived, stateful resources such as the database (`DatabaseService`), inference engine (`llmService`), embedding engine (`EmbeddingService`), sensor interfaces (`CameraService`, `LocationService`, `GarminService`), and inference coordination components (`ChatService`).

  A small number of modules are implemented as React hooks where direct integration with the React runtime is required. These are used for exposing live sensor data to the UI (e.g., `useDeviceStats`, `useHrWindow`) and for coordinating the continuous inference loop (`useLLMLoop`), which requires access to up-to-date application state.

The remainder of this chapter describes the system bottom-up, following the dependency order of its layers. The native modules are presented first, as they are the foundation for key parts of the service layer.

3.2 Native Modules

The native module layer provides the low-level inference on which the rest of the system depends. Both modules are implemented as custom Expo native modules in Kotlin and expose asynchronous TypeScript interfaces to the service layer above.

3.2.1 Expo LiteRT-LM

The `expo-litertlm` module provides the bridge between the React Native application and the on-device LiteRT-LM runtime. It is implemented as a custom Expo native module in Kotlin and exposes a high-level TypeScript interface for loading models, managing conversations, and performing inference.

3.2.1.1 Native Module Implementation

At the native layer, the module wraps the LiteRT-LM `Engine` API and manages model lifecycle, conversation state, and inference execution. Each loaded model is represented by an engine instance, while each interaction is handled through a conversation abstraction.

The following excerpt illustrates the core model loading process:

```
AsyncFunction("loadModel") { modelPath: String, options: LoadModelOptions, promise:
    Promise ->
    val config = EngineConfig(
        modelPath = modelPath,
```

```
        backend = parseBackend(options.backend),
        visionBackend = parseBackend(options.visionBackend),
        maxNumTokens = options.maxTokens,
    )
    val engine = Engine(config)
    engine.initialize()

    val handle = nextHandle++
    engines[handle] = EngineEntry(engine, ...)
    promise.resolve(handle)
}
```

A conversation is then created from a loaded engine, encapsulating the dialogue state and decoding parameters:

```
AsyncFunction("createConversation") { engineHandle: Int, options:
    ConversationOptions, promise: Promise ->
    val entry = engines[engineHandle]
        ?: throw IllegalArgumentException("Invalid engine handle")

    val sessionConfig = SessionConfig(
        temperature = options.temperature,
        topK = options.topK,
        topP = options.topP,
        systemPrompt = options.systemPrompt,
    )
    val conversation = entry.engine.createConversation(sessionConfig)

    val convHandle = nextConvHandle++
    conversations[convHandle] = ConversationEntry(conversation, engineHandle)
    promise.resolve(convHandle)
}
```

Inference is executed within a conversation context, supporting multimodal inputs:

```
AsyncFunction("sendInConversation") { convHandle: Int, prompt: String, imageUri:
    String, promise: Promise ->
    val contents = if (imageUri.isNotEmpty())
        Contents.of(Content.ImageFile(imageUri), Content.Text(prompt))
    else
        Contents.of(Content.Text(prompt))

    val message = entry.conversation.sendMessage(contents)

    promise.resolve(mapOf(
```

```
        "response" to message.toString(),
        "benchmark" to benchmarkToMap(entry.conversation.getBenchmarkInfo())
    ))
}
```

This design abstracts low-level inference details while preserving flexibility in configuring decoding parameters such as temperature, top-K sampling, and token limits. Additionally, benchmark metrics such as time-to-first-token and decoding throughput are exposed to the upper layers for system evaluation.

3.2.1.2 TypeScript Interface and Usage

On the TypeScript side, the module is accessed through the `useLiteRTLM` hook, which provides a declarative interface for managing model lifecycle and performing inference within the React Native environment.

The hook encapsulates model loading and execution:

```
  const llm = useLiteRTLM({
    modelPath,
    modelUrl: config.modelUrl,
    maxTokens: config.maxTokens,
    temperature: config.temperature,
    topK: config.topK,
    randomSeed: config.randomSeed,
    enableThinking: config.enableThinking,
    backend: config.backend,
    visionBackend: config.visionBackend,
  });
```

A typical inference interaction is performed as follows:

```
const convId = await llm.createConversation(systemInstruction);

const { response, benchmark } =
  await llm.sendInConversation(convId, prompt, imageUri);
```

3.2.1.3 Role in System Architecture

This module serves as the bridge between the React Native application and the on-device LiteRT-LM runtime. It is responsible for loading the model, managing conversation state, and handling inference requests that may include both text and images.

3.2.2 Expo RAG Module

The `expo-rag` module provides the bridge between the React Native application and the on-device embedding runtime used for retrieval-augmented generation (RAG). It is implemented as a custom Expo native module in Kotlin and exposes a minimal asynchronous interface for initializing the embedding model and generating dense vector representations of text inputs.

This module is responsible for executing lightweight transformer-based embedding models entirely on-device. It enables semantic encoding of text without relying on external services, ensuring offline operation, low latency, and privacy-preserving retrieval. The generated embeddings are consumed by the higher-level Embedding Service to perform similarity search over indexed mission data.

3.2.2.1 Native Module Implementation

At the native layer, the module wraps a TensorFlow Lite-based embedding model and manages model lifecycle, asset provisioning, and inference execution. The model consists of two core components: the TFLite inference graph and a SentencePiece tokenizer, both of which are loaded from bundled application assets.

Model initialization is executed asynchronously to avoid blocking the UI thread. Once assets are prepared, the embedding engine is initialized:

```
embedder = GemmaEmbeddingModel(
    tfliteFile.absolutePath,
    spFile.absolutePath,
    USE_GPU
)
```

During initialization, the module emits structured log events to the JavaScript layer to provide visibility into model loading and readiness state.

3.2.2.2 Embedding Execution Pipeline

The primary functionality of the module is exposed through the `embed` function, which transforms input text into a high-dimensional semantic vector.

```
val embedData = EmbedData.create(text, EmbedData.TaskType.RETRIEVAL_DOCUMENT)
val request = EmbeddingRequest.create(listOf(embedData))

val rawList = embedder!!.getEmbeddings(request).get()
val result: List<Double> = List(rawList.size) { rawList[it].toDouble() }
```

3.2.2.3 TypeScript Interface and Usage

On the TypeScript side, the module is exposed through an Expo native module interface, providing a simple asynchronous API for initialization and embedding generation.

The module exposes three primary functions:

- `initialize()` — loads and prepares the embedding model
- `isInitialized()` — checks whether the module is ready for inference
- `embed(text)` — returns a semantic embedding vector

A typical usage pattern is shown below:

```
await rag.initialize();

const embedding = await rag.embed("search query text");
```

3.2.2.4 Role in System Architecture

Within the overall system architecture, the `expo-rag` module serves as the foundational semantic encoding layer. It transforms raw text into dense vector representations that are consumed by the Embedding Service to perform cosine similarity search over stored mission observations. By executing all embedding inferences locally, the system ensures offline capability and improved data privacy.

3.3 Service Layer

The Service Layer sits above the native modules and provides the application logic. Each service is initialized once at application startup and exposes an interface for synchronous and asynchronous access.

3.3.1 LLM Service

The `useLLMService` hook wraps the `expo-litertlm` native module and exposes model configuration and a stateless inference interface to the rest of the system.

3.3.1.1 Model Configuration

The hook is initialised with an `LLMConfig` object that controls all inference parameters, including model selection, decoding behaviour, and hardware backend assignment. Default values are defined as follows:

```
export const DEFAULT_LLM_CONFIG: LLMConfig = {
  modelName: "",
  modelUrl: "",
  maxTokens: 4092,
  temperature: 0.3,
  topK: 5,
  randomSeed: 42,
  enableThinking: false,
  backend: 'CPU',
  visionBackend: 'CPU',
};
```

Both text and vision inference are configured to execute on the CPU backend by default. Temperature and top-K sampling are kept deliberately low to bias the model toward factual, deterministic scene descriptions. A fixed random seed ensures reproducible outputs across inference runs, which is particularly important during controlled evaluation.

3.3.1.2 Hook Initialization

Internally, the hook delegates model loading and life-cycle management to `useLiteRTLM`, passing the resolved on-device model path alongside all parameters. The model path is constructed by appending the configured model filename to the application's local models directory:

```
const modelPath = MODELS_DIR.replace('file://', '') + config.modelName;
const llm = useLiteRTLM({
  modelPath,
  modelUrl: config.modelUrl,
  maxTokens: config.maxTokens,
  temperature: config.temperature,
  topK: config.topK,
  randomSeed: config.randomSeed,
  enableThinking: config.enableThinking,
  backend: config.backend,
  visionBackend: config.visionBackend,
});
```

3.3.1.3 Inference Interface

The hook exposes `inferOnce`, a helper that wraps a full conversation lifecycle into a single call. It opens a conversation handle, performs inference, and closes it on completion, returning the response text, any thinking tokens, and benchmark metrics.

```
async function inferOnce(
  prompt: string,
  imageUri?: string,
  systemInstruction?: string,
): Promise<{ response: string; thinking: string; benchmark: BenchmarkInfo }> {
  const convHandle = await llm.createConversation(systemInstruction);
  try {
    return await llm.sendInConversation(convHandle, prompt, imageUri);
  } finally {
    llm.closeConversation(convHandle).catch(() => {});
  }
}
```

This is the primary function used by the inference pipeline on each iteration to produce a scene description from a camera frame.

3.3.1.4 Role in System Architecture

The `useLLMService` hook sits at the center of the system's inference capability and is consumed by two distinct components in the architecture, each using it in a different context.

The first consumer is the `useLLMLoop` hook, which drives the continuous inference pipeline during active missions. On each iteration, the loop constructs a multimodal prompt using the latest camera frame and then calls `inferOnce` to execute a self-contained inference step.

```
const { response: responseText, thinking, benchmark } =
  await optsRef.current.llm.inferOnce(
    prompt,
    photoUri,
    systemInstruction,
  );
```

The resulting scene description is immediately forwarded to the embedding service to generate a vector representation, which is then stored in the SQLite database together with the GPS coordinates, as illustrated in Figure 3.1.

```
const embedding = await optsRef.current.embedding.embed(response.summary);
const insertId = await optsRef.current.db.insertRecord({
  ...response,
  mission_id: missionId,
  embedding: embedding ?? undefined,
});
```

Beyond inference, the `useLLMLoop` hook also manages device-level concerns during active op-

eration. On start, it activates a keep-awake lock to prevent the device from sleeping mid-mission and reduces screen brightness to 25% to conserve battery. Both are restored cleanly on stop:

```
const start = useCallback(async () => {
  await activateKeepAwakeAsync('llm-loop');
  savedBrightnessRef.current = await Brightness.getBrightnessAsync();
  await Brightness.setBrightnessAsync(0.25);
  iterationRef.current = runOnce();
}, []);

const stop = useCallback(async () => {
  deactivateKeepAwake('llm-loop');
  if (savedBrightnessRef.current !== null) {
    await Brightness.setBrightnessAsync(savedBrightnessRef.current);
  }
  await iterationRef.current;
}, []);
```

The second consumer is the ChatService, which accesses the underlying llm handle directly to manage its own persistent multi-turn conversation session. Unlike the inference loop, which uses inferOnce for stateless single-turn calls, the ChatService maintains an open conversation handle across multiple user queries, allowing the model to reference prior exchanges within the same session.

The third consumer is the History screen, which uses the inference capability to generate summaries over previously stored mission data. Unlike the continuous inference loop, this component operates on recorded observations and performs on-demand summarization for a selected time window.

3.3.2 Embedding Service

The `EmbeddingService` wraps the `expo-rag` native module and provides two capabilities to the rest of the system: embedding generation and semantic retrieval over stored mission observations.

3.3.2.1 Embedding Generation

Embedding generation is exposed through the `embed()` function, which converts a text string into a dense vector using the on-device embedding model:

```
async embed(text: string): Promise<number[] | null> {
  if (!this.isReady()) return null;
  try {
```

```
    return await ExpoRag.embed(text);
  } catch (e: any) {
    console.warn('RAG: embed failed:', e.message);
    return null;
  }
}
```

3.3.2.2 Embedding Cache

To avoid repeatedly loading and parsing the database on every search, the service keeps embeddings in memory. Each cached item includes the record ID, the original text summary, its embedding vector, and a mission ID.

```
private cache = new Map<number, {
  text: string;
  embedding: number[];
  mission_id?: number
}>();
```

The cache is populated from the SQLite database at startup:

```
loadFromDatabase(records: LLMRecord[]): void {
  for (const record of records) {
    if (record.id != null && record.embedding) {
      this.cache.set(record.id, {
        text: record.summary,
        embedding: record.embedding,
        mission_id: record.mission_id ?? undefined,
      });
    }
  }
}
```

This decision is justified by the nature of the data. Embedding vectors are fixed-size float arrays. The embedding model used produces 256-dimensional vectors, where each element is represented as a 32-bit floating point value, resulting in approximately 1 KB per embedding.

Even when accounting for JavaScript object overhead and associated metadata, each cached entry occupies only a few kilobytes. As a result, a mission with several hundred observations requires only a few megabytes of memory, which is well within the capabilities of modern mobile devices.

This makes a full in-memory cache practical, while avoiding the overhead of repeated SQLite

reads, JSON deserialisation of embedding data, and the latency this would introduce into every retrieval call. Since retrieval must complete before the LLM can generate a response, keeping the cache in memory ensures that semantic search remains fast and consistent regardless of database size. The retrieval operation remains $O(n)$, but operates entirely over in-memory data, making it significantly faster.

The cache is updated incrementally as new records are inserted by the inference loop, ensuring consistency with the database without requiring a full reload:

```
addToCache(id: number, text: string, embedding: number[], mission_id?: number): void {
  this.cache.set(id, { text, embedding, mission_id });
}
```

3.3.2.3 Similarity Search

Retrieval is performed as a brute-force cosine similarity search over all cached entries. Given a query string, the service first generates its embedding and then scores it against every cached entry:

```
function cosineSimilarity(a: number[], b: number[]): number {
  let dot = 0, normA = 0, normB = 0;
  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    normA += a[i] * a[i];
    normB += b[i] * b[i];
  }
  const denom = Math.sqrt(normA) * Math.sqrt(normB);
  return denom === 0 ? 0 : dot / denom;
}
```

Only results exceeding a configurable similarity threshold are retained. The final result set is sorted in descending order of relevance and truncated to the top-K matches:

```
return results
  .sort((a, b) => b.relevance - a.relevance)
  .slice(0, topK);
```

An optional mission identifier can be passed to restrict the search to observations from a specific mission, which is used to scope retrieval to a specific mission.

3.3.3 Chat Service (RAG)

The `ChatService` orchestrates a retrieval-augmented generation (RAG) pipeline. It accepts a natural language query from the user and returns a grounded response by retrieving semantically relevant mission records from the local SQLite database and injecting them as context into the on-device LLM. The complete pipeline is shown in Figure 3.2.

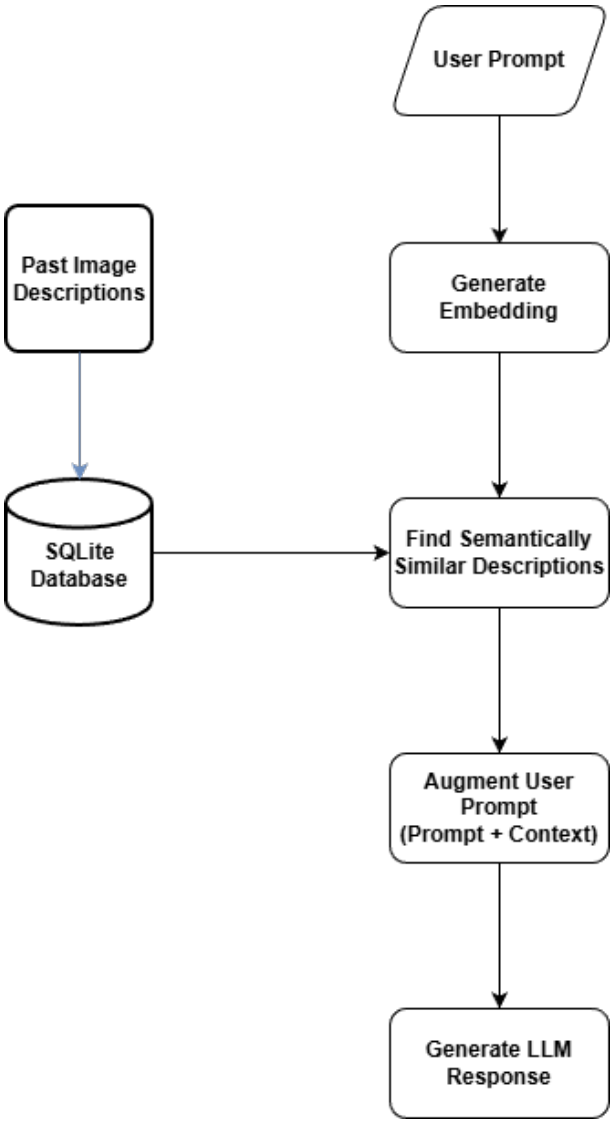


Figure 3.2: Retrieval-augmented generation pipeline: user queries are embedded and matched against stored frame descriptions via semantic similarity search. Retrieved context is augmented with the original prompt and passed to the LLM to generate a response.

3.3.3.1 RAG Pipeline

When a query is received, the system first delegates to the `EmbeddingService` to perform a semantic similarity search over stored mission observations. The query is embedded into the

same vector space as stored records, and a nearest-neighbor search is executed over the local SQLite database. This returns the top-k most relevant observations above a configurable similarity threshold.

```
const hits = await this.embedding.searchRelevant(question, 4, 0.35, missionId);
const ids = hits.map(h => h.id);
const records = ids.length > 0 ? await this.db.getRecordsByIds(ids) : [];
```

The retrieved records are then formatted into a structured observation list, which is combined with the user's query to form an augmented prompt. Each observation is indexed to allow explicit referencing during generation.

```
function buildChatTurnPrompt(question: string, records: LLMRecord[]): string {
  const observations = records
    .map(r => `[${r.id}] ${r.summary}`)
    .join('\n');

  return `Observations: ${observations || '(none)'} Operator: ${question}`;
}
```

This augmented prompt is passed to the LLM, enabling retrieval-augmented generation where responses are conditioned on the retrieved semantic memory.

To ensure grounded reasoning and reduce hallucination, the system applies a strict instruction set to the language model. This instruction constrains the model to operate exclusively over retrieved observations and enforces explicit citation behavior:

> `You are an assistant reviewing a rescue operation. Each turn the operator provides mission observations and asks a question. - Answer using observations provided in the current turn or any observation already discussed earlier in this conversation. - Cite observations by their id like [12] or [47]; ids are stable, so reuse the same id when referring back to an earlier source. - If the observations don't contain the answer, say so briefly. - Be concise. Synthesize a focused answer; do not enumerate observations one by one unless asked.`

This design ensures that all responses are grounded in retrieved mission data, preventing hallucination and enforcing traceability of generated outputs to specific stored observations.

An example of the constructed augmented prompt is shown below. The system retrieves semantically relevant observations and formats them into an indexed list, which is then combined with the user's query:

Observations:

```
[1] A roadside incident involving an individual who appears to be
in a distressed physical state, with surrounding people reacting
and attempting to provide assistance while the exact cause is unclear...
[2] Emergency medical services are present at the location, with
responders actively coordinating around a central point of attention
and preparing intervention procedures...
[3] The environment shows signs of an unfolding medical emergency,
with bystanders gathered and movement focused toward a single affected
individual on the ground...
```

User Query:

Are there any emergency medical situations?

3.3.3.2 Session Management

Conversation state is maintained across turns through a persistent session handle. On the first query of a session, a new conversation is created with the system instruction.

```
if (this.sessionId === null) {
  this.sessionId = await this.llm.createConversation(CHAT_SYSTEM_INSTRUCTION);
}

const { response } = await this.llm.sendInConversation(this.sessionId, prompt);
```

The session is reset explicitly when the user starts a new conversation or the service is disposed, at which point the underlying LLM conversation handle is closed and released.

3.3.4 Database Service

This service manages all persistent storage using a local SQLite database (**rescueaid.db**) via react-native-sqlite-storage.

The database is designed around four core data structures, shown in Figure 3.3:

- **missions**: stores mission metadata, including start and end timestamps, and acts as the top-level grouping for all collected data.
- **inferences**: stores each LLM output along with its summary, associated camera frame, GPS coordinates, mission reference, and serialized embedding vector for semantic retrieval.

- **inference_stats**: stores detailed performance metrics for each inference call, including latency, token statistics, and device resource utilization.
- **mission_stats**: stores aggregated statistics computed at the end of a mission, summarizing overall system performance and device state.
- **hr_samples** stores time-series heart rate measurements collected during missions.

Figure 3.3: Local SQLite database schema.

Each inference generated by the system is persisted immediately after execution, including its associated embedding vector to support later semantic retrieval. Performance metrics are recorded separately at inference time.

Mission-level statistics are derived from the collected inference metrics when a mission concludes, allowing higher-level analysis of system performance.

Heart rate data is continuously logged during operation and later utilized.

3.3.5 Camera Service

The `CameraService` manages visual input from the device camera. It captures a photo on a fixed interval timer. Each frame is captured as a compressed JPEG and persisted to a dedicated `photos/` directory within the application's local storage:

```
this.intervalId = setInterval(async () => {
  const photo = await this.takePhoto();
  if (photo && this.onPhotoTaken) {
    this.onPhotoTaken(photo);
  }
}, 5000);
```

The service exposes a `getLatestPhotoUri()` accessor used by the inference loop to retrieve the most recent frame on each iteration.

In addition to live capture, an `ExperimentCameraService` is provided for deterministic evaluation. This service replays pre-recorded image sequences from a fixed dataset, enabling reproducible benchmarking of the inference pipeline under controlled visual input conditions.

3.3.6 Location Service

The `LocationService` tracks geographic position using high-accuracy GPS, sampling latitude, longitude, altitude, and accuracy every 5 seconds or every 5 metres of movement:

```
Location.watchPositionAsync(
  {
    accuracy: Location.Accuracy.BestForNavigation,
    timeInterval: 5000,
    distanceInterval: 5,
  },
  (location) => {
    const locationData: LocationData = {
      latitude: location.coords.latitude,
      longitude: location.coords.longitude,
      accuracy: location.coords.accuracy || 0,
      timestamp: Date.now(),
      altitude: location.coords.altitude || 0,
    };
    this.lastKnownLocation = locationData;
    onUpdate(locationData);
  }
);
```

3.3.7 Garmin Service

This service retrieves heart-rate data from a Garmin watch over Bluetooth Low Energy. The watch must have heart-rate broadcasting enabled, after which the service scans for devices advertising the standard Heart Rate profile (service UUID `0x180D`) and subscribes to notifications on the heart rate measurement characteristic (`0x2A37`). Incoming values are throttled to one sample per second and decoded from the BLE packet format into beats per minute.

Over the last 60 seconds of samples, a helper function computes average, minimum and maximum heart rate:

```
return {
  hrAvg: Math.round(hrAvg),
  hrMin,
  hrMax,
  windowSeconds: Math.round(windowSeconds),
};
```

3.3.8 Inference Stats Collector Service

The Inference Stats Collector monitors device-level performance during each inference iteration when statistics collection is enabled. In this mode, it samples CPU usage, memory consumption, and GPU utilization at fixed intervals over the duration of the inference call.

After completion, it aggregates these measurements into summary statistics (minimum, maximum and average values) and combines them with LLM runtime metrics such as token counts, decoding speed, time-to-first-token, and total inference latency.

If enabled, the resulting data is stored in the local SQLite database for subsequent system evaluation.

3.4 Interface and Features

The following section presents the application's main features, interface and its components.

3.4.1 Record Screen

The Record screen is the main screen of the application and the screen used during an active mission. It displays the most recent assessments produced by the on-device model, the device's camera feed and the operator's heart rate data. From here, the operator picks the operation type, starts and stops a mission, and configures the application using the configuration modal.

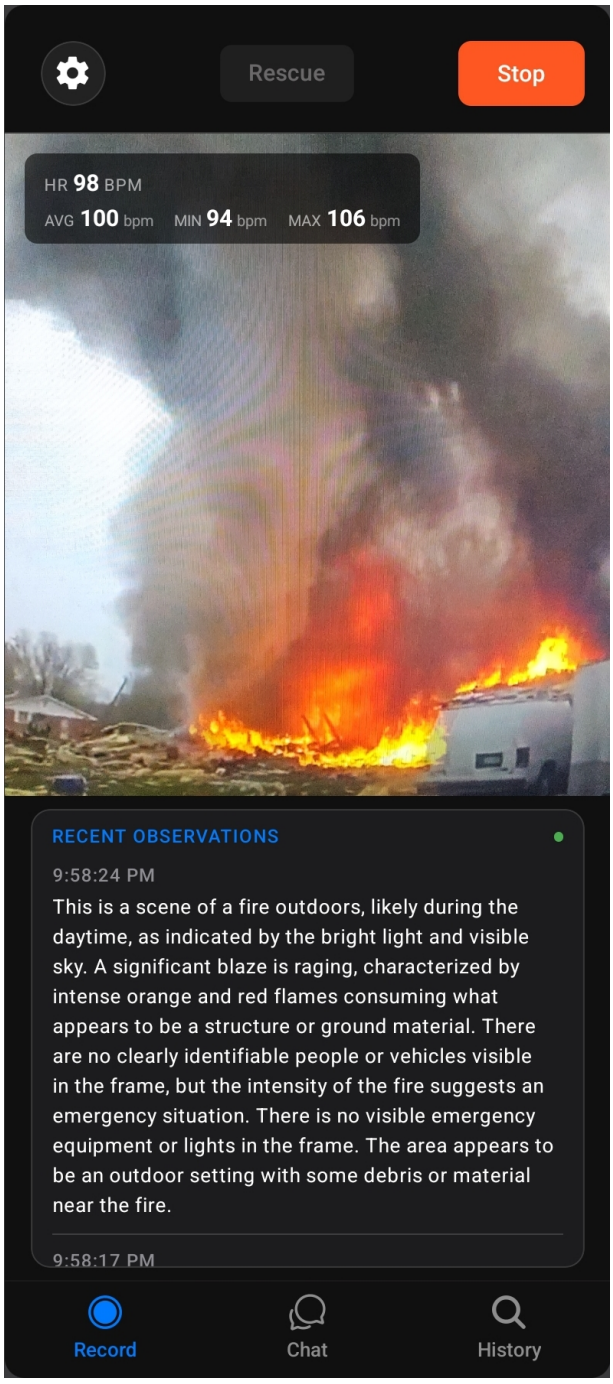


Figure 3.4: The Record screen during an active mission.

3.4.1.1 Header Bar

A compact top bar that combines the settings button, the operation-type chip (Rescue or Police), and the prominent Start/Stop button, so all mission-level controls are reachable in one place.

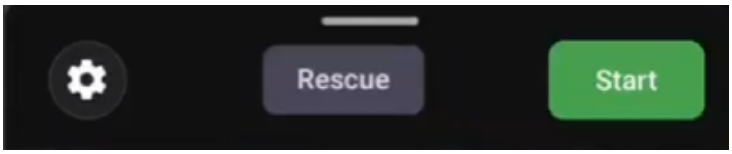


Figure 3.5: Header with settings, operation chip, and record button.

3.4.1.2 Device Stats

A thin strip showing real-time CPU, RAM, GPU memory, and battery, giving the operator at-a-glance feedback on how heavily the device is being used. This is only visible if stats collection for system evaluation is enabled.

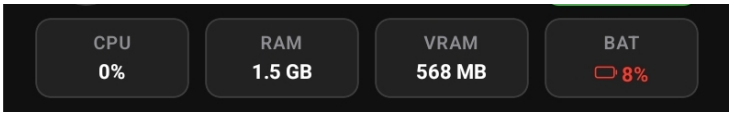


Figure 3.6: Stats bar displaying real-time CPU, RAM, GPU memory, and battery.

3.4.1.3 Camera View

The live camera preview that the model uses as its visual input. It occupies the central area of the screen and shows what the device is currently observing.

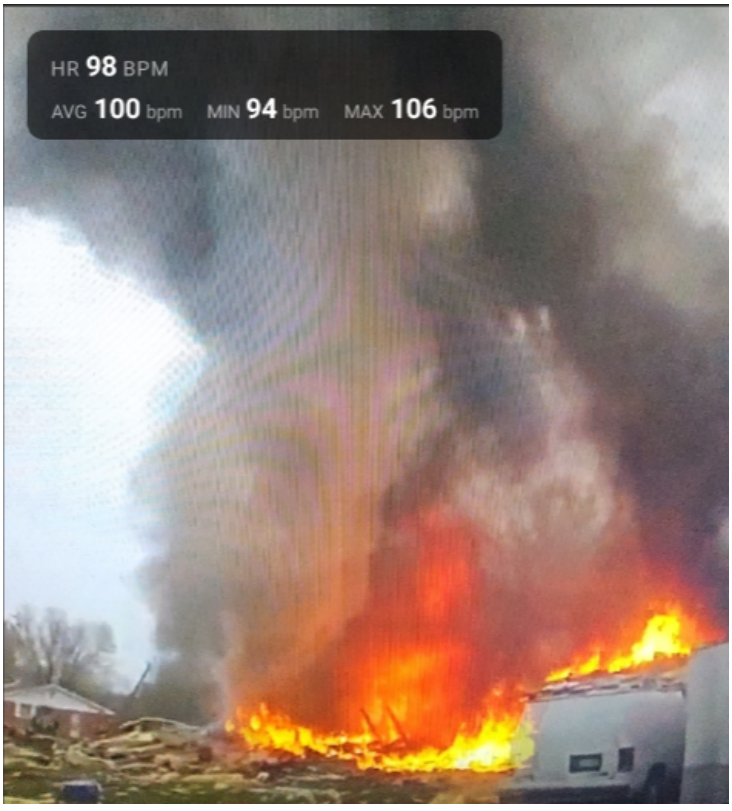


Figure 3.7: The camera view component.

3.4.1.4 Heart-Rate Overlay

A small overlay on top of the camera view that shows the wearer's current heart rate together with rolling minimum, average, and maximum values from the recent window.

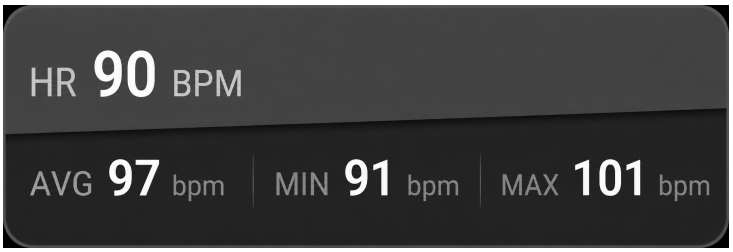


Figure 3.8: Heart rate Overlay displaying the current Hr window

3.4.1.5 Latest Observations

A scrollable panel below the camera that displays the most recent inferences generated by the model, so the operator can read what the system observed without leaving the screen.

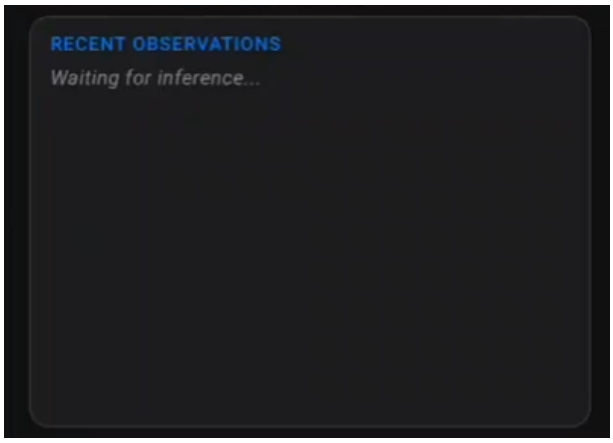


Figure 3.9: Initial state of the Latest Observations panel

3.4.1.6 Configuration Modal

A full-screen modal that gathers everything the operator might need before a mission: model selection and download, generation parameters, heart-rate device pairing, experiment mode, and database export.

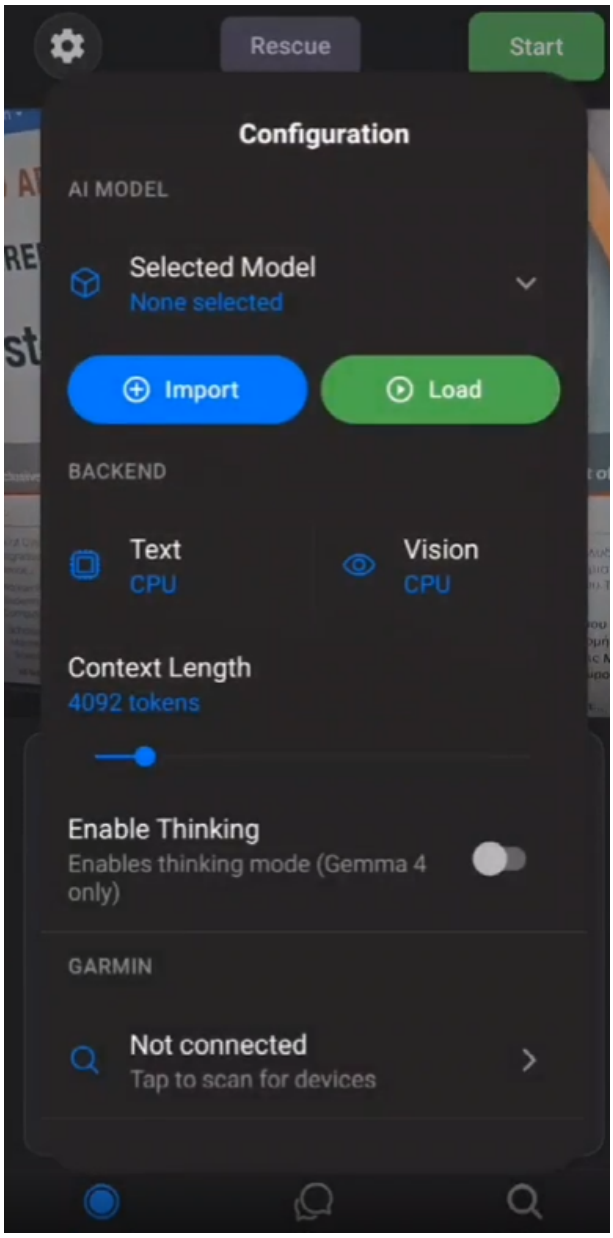


Figure 3.10: Initial state of the configuration modal.

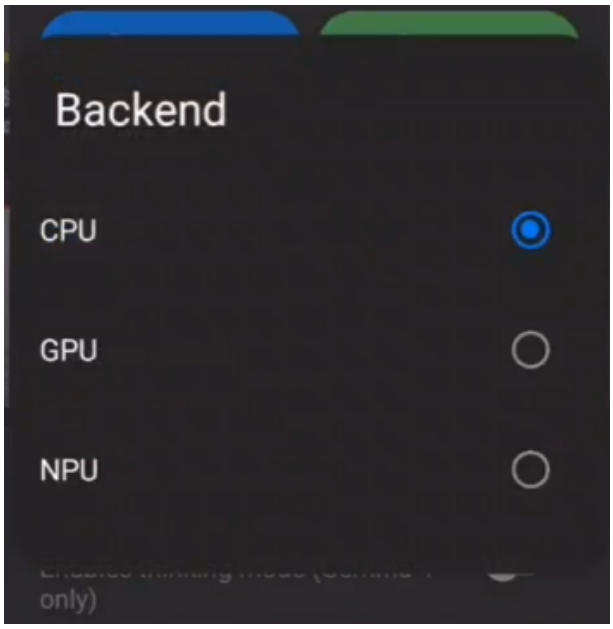


Figure 3.11: Text backend selection.

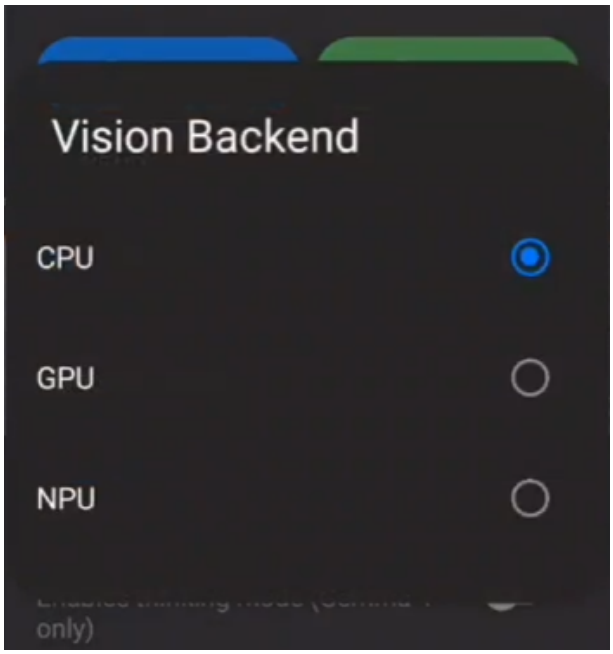


Figure 3.12: Vision backend selection.

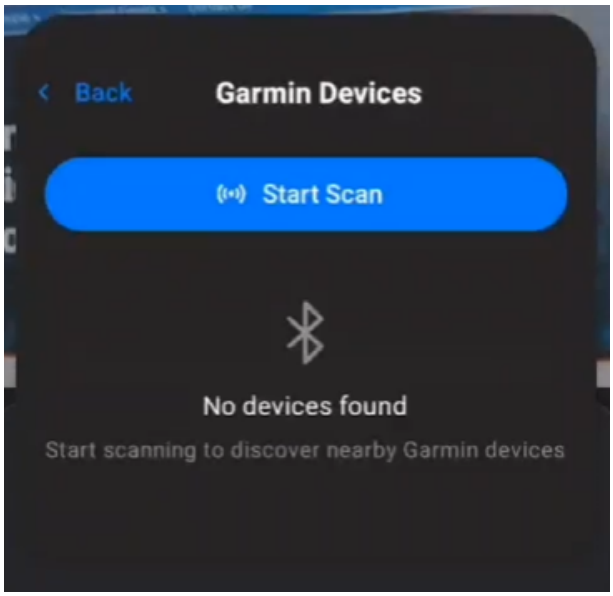


Figure 3.13: Garmin configuration before connecting.

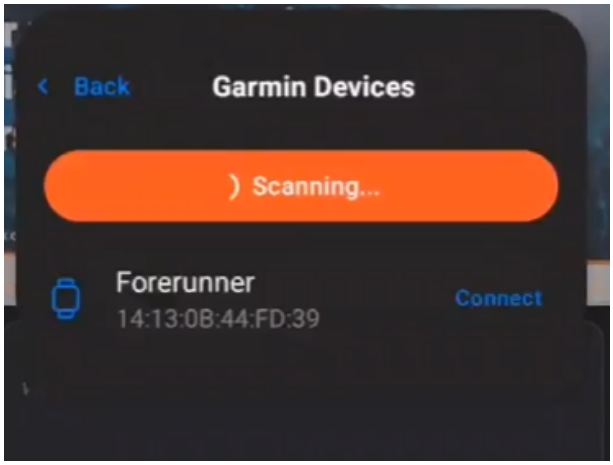


Figure 3.14: Garmin configuration after connecting.

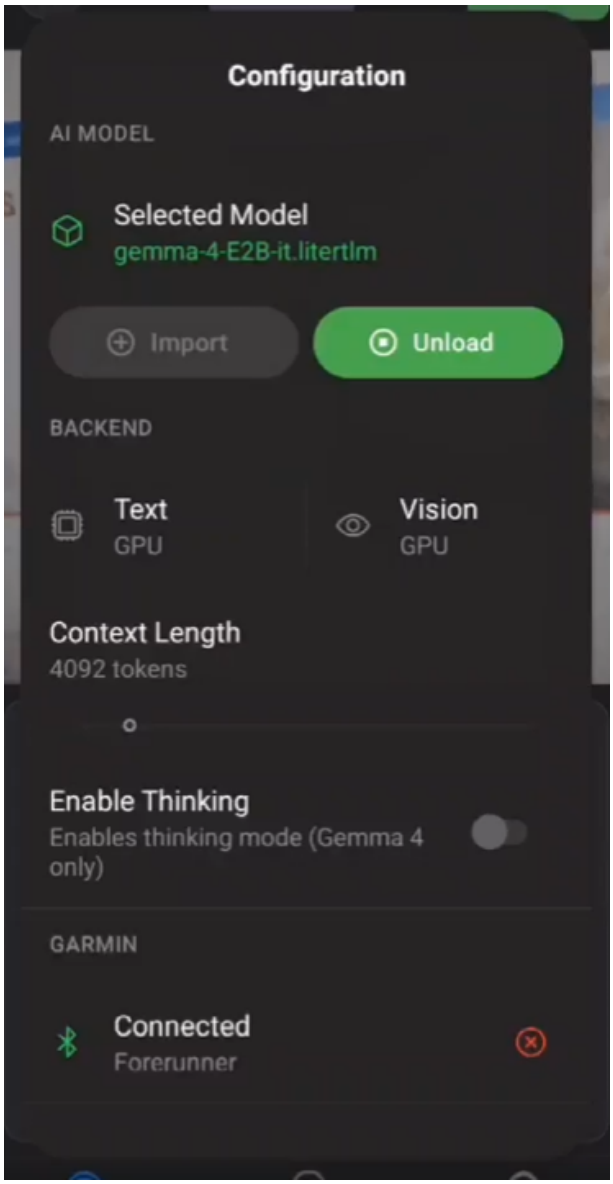


Figure 3.15: The configuration modal after configuration.

3.4.1.7 Onboarding Overlay

A first launch overlay that briefly walks new users through what the app does and how to get started.

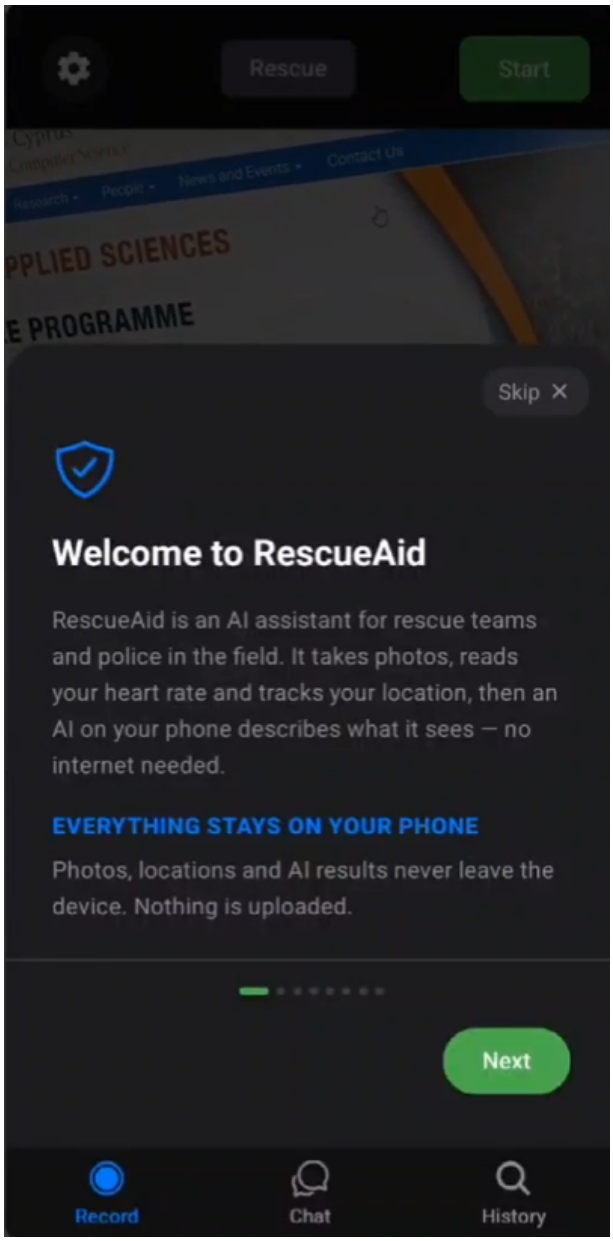


Figure 3.16: First-launch onboarding overlay.

3.4.2 Chat Screen

The Chat screen is where the operator can ask questions about what happened during a mission, after the fact. It behaves like a familiar messaging interface but draws its answers from the assessments the system recorded earlier, citing the specific observations it references.

Figure 3.17: The Chat screen.

Figure 3.18: The Chat screen while loading an LLM response.

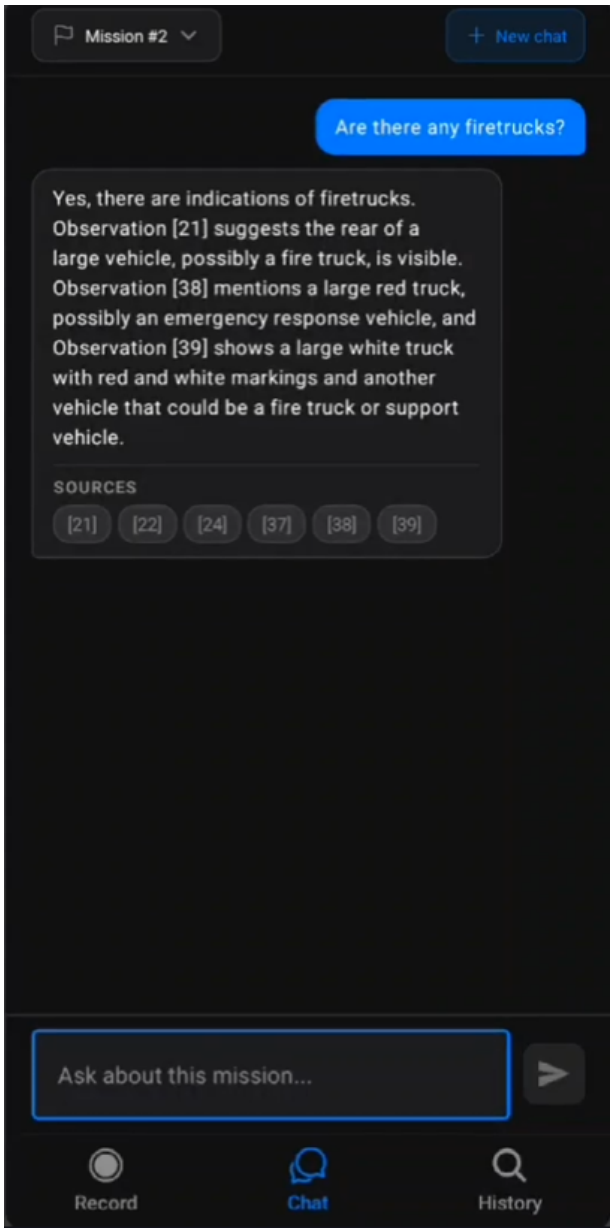


Figure 3.19: The Chat screen after a message.

3.4.2.1 Mission Picker Modal

A pop-up list of recorded missions, used to switch the conversation's scope. Each entry shows when the mission took place and how many observations it contains.

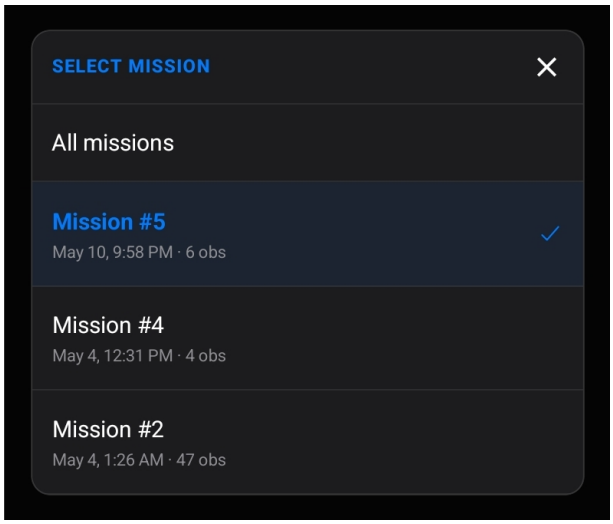


Figure 3.20: Picking a mission scope for the conversation.

3.4.2.2 Citation Detail Modal

A modal opened by tapping a citation pill. It shows the full observation behind the citation, including its photo, map location, and original text, so the operator can verify where an answer came from.

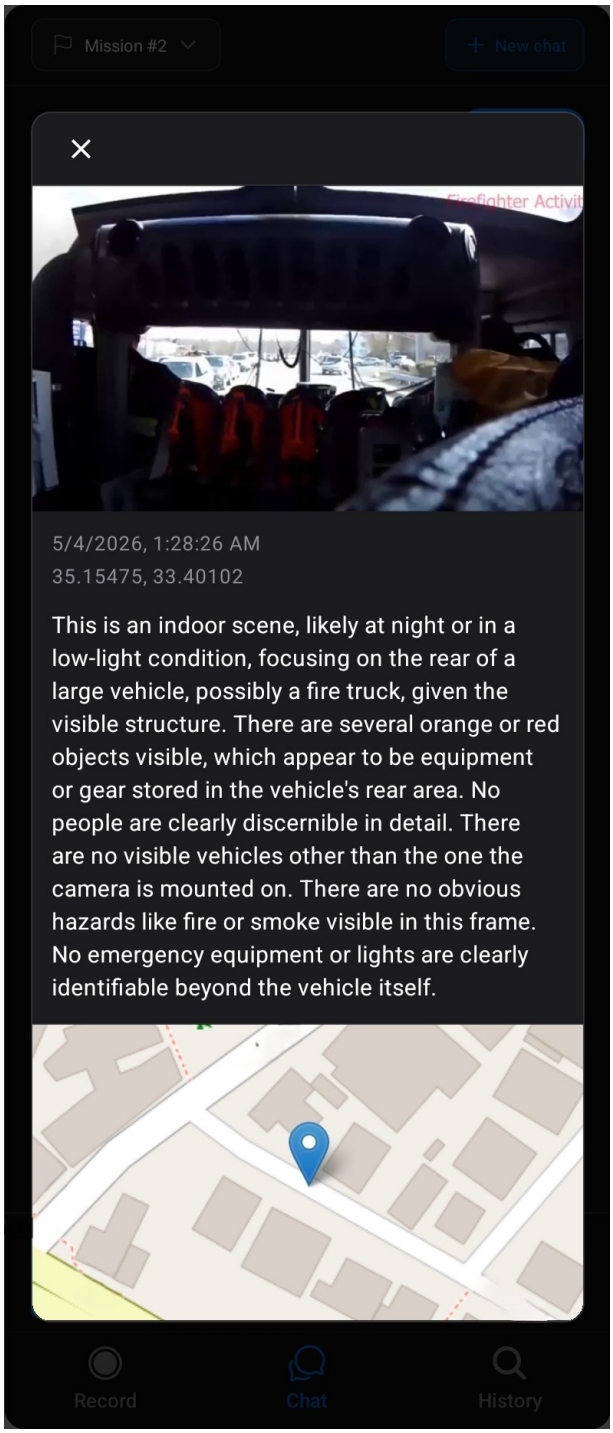


Figure 3.21: The citation modal of a chat response.

3.4.3 History Screen

The History screen is the searchable archive of every assessment the system has ever recorded. The operator can scroll through past observations or search across all of them in natural language to find specific moments.

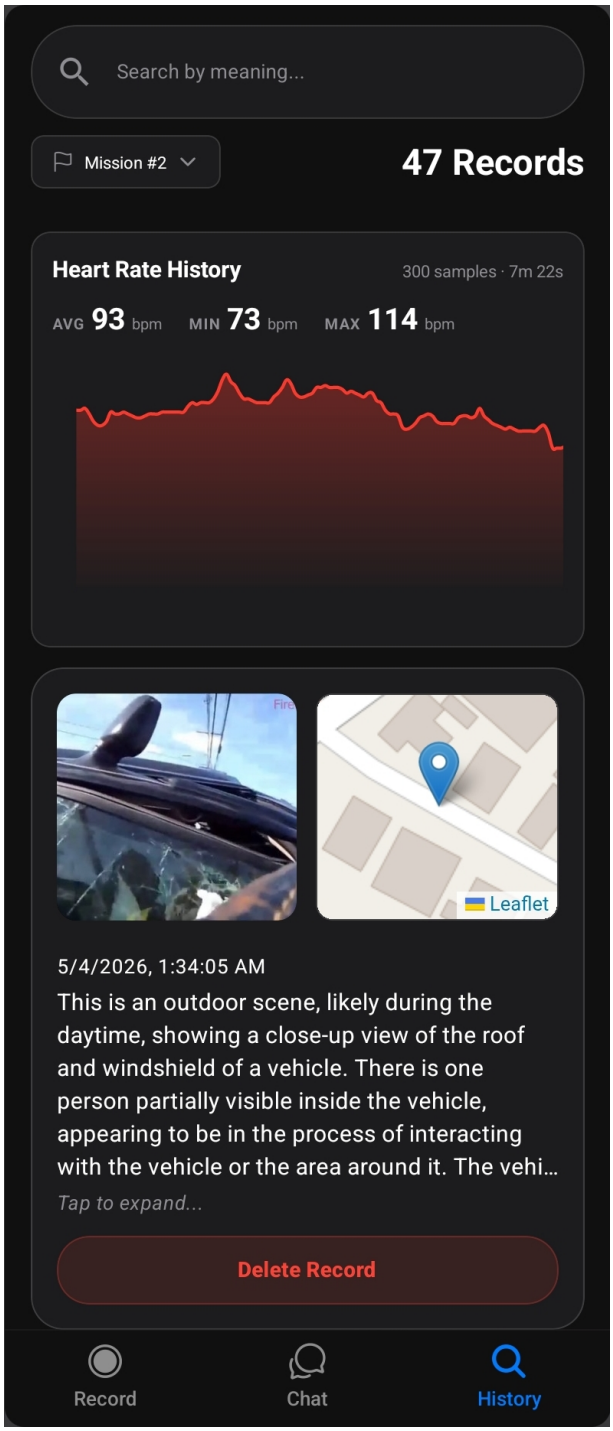


Figure 3.22: The history screen.

3.4.3.1 Semantic Search

The History screen supports natural language querying, allowing the operator to describe a situation in plain terms rather than constructing keyword filters. When a search is submitted, the system computes semantic similarity between the query and all stored assessment records, then

re-ranks and displays the results ordered by relevance. This enables the operator to surface conceptually related moments even if the exact wording differs.

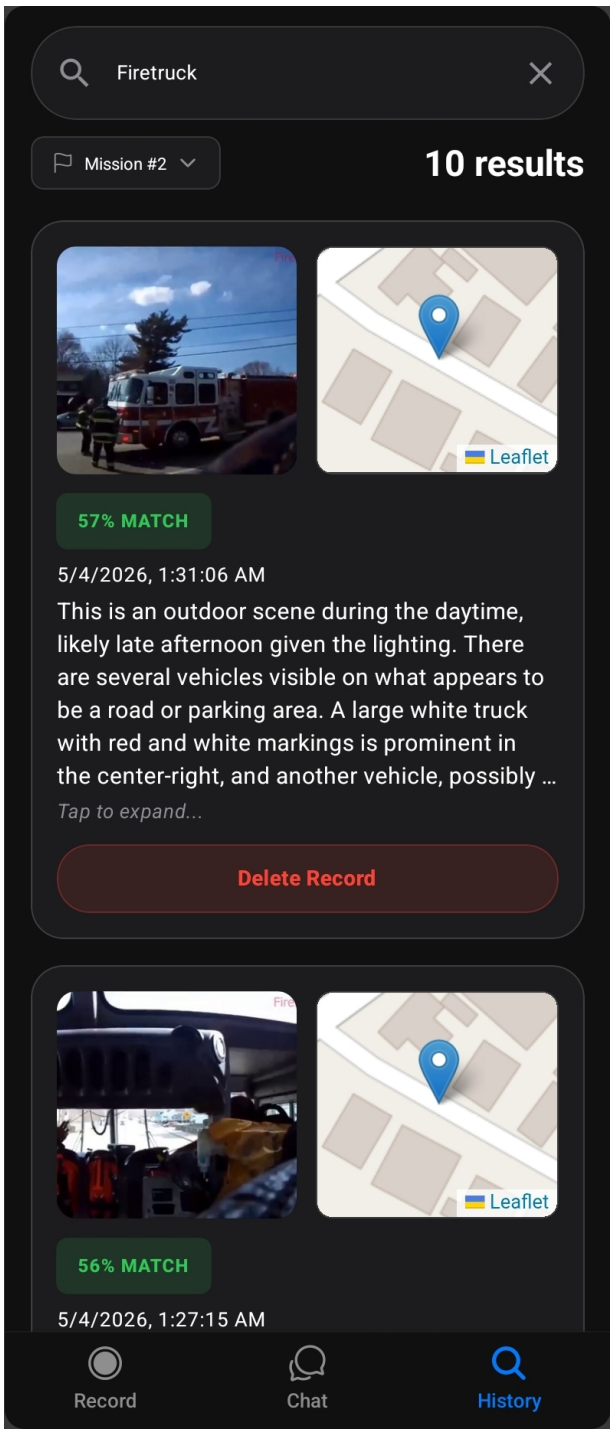


Figure 3.23: The history screen during semantic search, records sorted based on similarity.

3.4.3.2 Heart Rate History Chart

The Heart Rate history chart provides a view of the operators's cardiac activity across the whole mission. By tapping anywhere on the timeline the operator can get a summary of the inferences around that point in time.

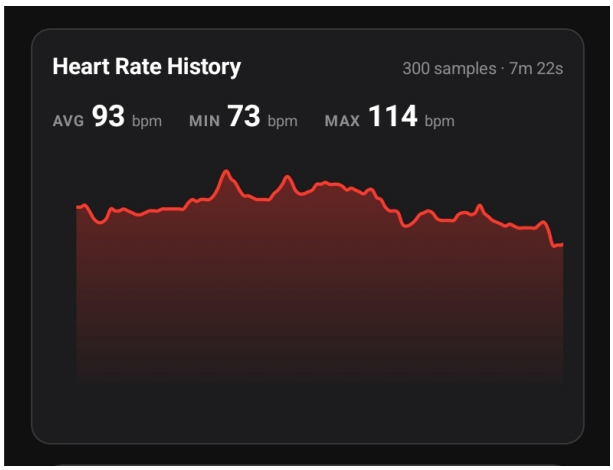


Figure 3.24: Heart Rate history chart.

3.4.3.3 Summary Modal

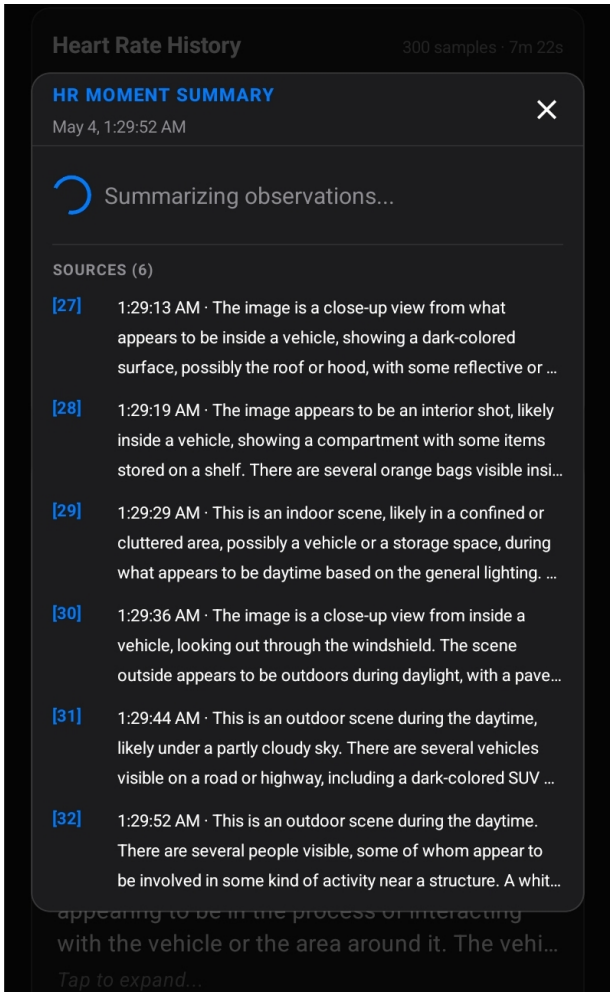


Figure 3.25: Moment summary modal, during inference.

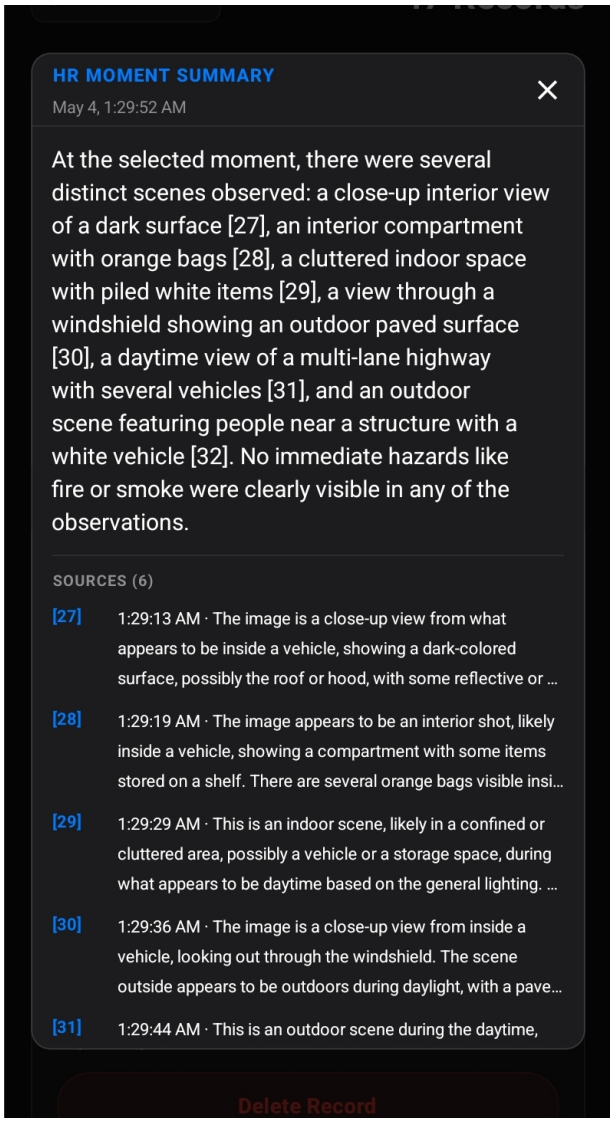


Figure 3.26: Moment summary modal, after inference.

4 Evaluation Methodology and Results

4.1 Introduction

This chapter presents the experimental setup, evaluation metrics, and results for the RescueAid system. The evaluation focuses on three main components: the vision-language model responsible for generating structured scene descriptions, the embedding-based retrieval system operating on the generated representations, and the overall runtime performance of the system.

The objective of this evaluation is not to establish absolute measures of semantic correctness but to perform a comparative analysis of model behavior under a consistent evaluation pipeline.

The chapter is structured as follows: Section 4.2 describes the dataset, models, and evaluation metrics. Section 4.3 presents quantitative results across models, categories, and image quality conditions, including retrieval performance. Section 4.4 interprets the findings in a system-level context.

4.2 Experimental Setup

4.2.1 Dataset

The evaluation uses the RescueAid dataset, constructed specifically for this work. It consists of 50 images extracted from various real bodycam videos of first-response situations. The dataset is divided into two operational categories:

- **Police**
- **Rescue**

Each image is additionally annotated with an image quality label representing real-world capture conditions:

- Clear
- Blurry
- Dark
- Obstructed

For each image, a human-authored ground-truth description is provided. These descriptions serve as reference annotations for evaluating relative differences in generated outputs.

4.2.1.1 Example from the Dataset

Figure 4.1 shows a representative example from the dataset.

Figure 4.1: Example image from the RescueAid dataset (category: Rescue, quality: clear).

Ground-truth description:

> This body camera image captures the aftermath of a multi-vehicle traffic collision outdoors on a highway or major road. The most prominent hazard is extensive debris, including vehicle parts, shattered glass, and fluid, scattered across multiple lanes of the asphalt roadway. No fire or smoke is visible. At least five vehicles are visible, with varying degrees of damage. Three individuals are visible standing on the right side of the road in the mid-ground, observing the scene. No emergency lights or specific emergency equipment are clearly visible within the immediate crash area.

This example illustrates the inherent complexity of the dataset, including multiple interacting entities, spatial relationships, and the absence of explicit emergency indicators in some cases.

4.2.1.2 Retrieval Evaluation Queries

In addition to the image descriptions, the retrieval evaluation uses a set of 30 manually defined natural-language queries. These queries represent information needs that could occur in an emergency-response setting, such as searching for fire scenes, water rescues, vehicle crashes, medical aid, or police activity.

For each query, the relevant images were manually selected from the dataset. The retrieval system then ranked stored image descriptions using cosine similarity between the query em-

bedding and the description embeddings. The top five retrieved images were evaluated using Precision@5 and Recall@5.

Example retrieval query:

severe car crash on highway with debris

Relevant images:

Rescue_021.jpg, Rescue_022.jpg, Rescue_023.jpg, Rescue_024.jpg, Rescue_025.jpg

This example shows how retrieval is evaluated: the system should return images that match the meaning of the query, even if the exact query words do not appear in the generated descriptions. This tests whether the generated descriptions are useful as searchable semantic representations, not only as standalone captions.

4.2.2 Models Evaluated

This evaluation considers four models from the Gemma family, spanning two architectural generations (Gemma 3n and Gemma 4) and two model scales (2B and 4B parameters). The selected variants are Gemma 3n-E2B-int4, Gemma 3n-E4B-int4, Gemma 4-E2B, and Gemma 4-E4B [4, 5].

These models are compared under identical evaluation conditions in order to isolate relative differences in generation behavior and downstream retrieval impact.

4.2.3 Evaluation Metrics

Seven metrics are used to evaluate system performance across generation and retrieval components.

For vision-language generation:

- **Semantic Similarity:** Cosine similarity between embeddings of generated and ground-truth descriptions. This measures alignment in embedding space rather than exact lexical overlap. This is our main metric.
- **Precision:** Fraction of extracted entities in the generated output that are also present in the ground truth.
- **Recall:** Fraction of ground-truth entities recovered in the generated output.
- **F1 Score:** Harmonic mean of precision and recall.
- **Hallucination Rate:** Fraction of extracted entities in the generated output not present in the ground-truth entity set.

For embedding-based retrieval:

- **Precision@5 (P@5):** Fraction of top-5 retrieved items that are relevant.
- **Recall@5 (R@5):** Fraction of all relevant items retrieved within the top-5 results.

For on-device runtime performance:

- **Average Inference Time:** Average time required to generate a description for one image.
- **Tokens/s:** Average number of output tokens generated per second.
- **Time to First Token (TTFT):** Time between starting inference and receiving the first generated token.

4.3 Results

4.3.1 Comparative Model Performance

Table 4.1: Overall Model Performance

Model	Sim.	Precision	Recall	F1	Halluc.
Gemma 3n-E2B	74.39	58.68	47.04	51.20	41.32
Gemma 3n-E4B	74.89	60.44	48.09	52.42	39.56
Gemma 4-E2B	70.33	66.46	36.61	46.16	33.54
Gemma 4-E4B	73.59	61.31	38.86	46.47	38.69

Table 4.1 shows the overall generation performance of the evaluated models. Gemma 3n-E4B achieves the highest semantic similarity and F1 score, giving the strongest overall balance between semantic alignment and entity coverage.

The Gemma 4 models show a different behavior. They achieve higher precision and lower hallucination rates, but also lower recall and F1 scores. This suggests that they produce more conservative descriptions, mentioning fewer uncertain details and therefore missing more scene information.

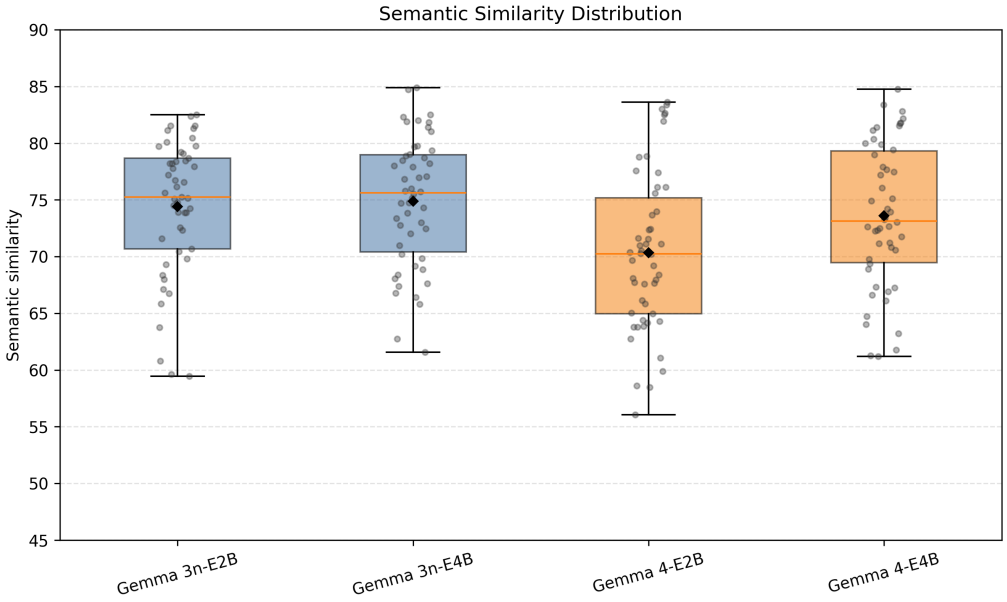


Figure 4.2: Distribution of semantic similarity scores across evaluated models.

Figure 4.2 shows the spread of semantic similarity scores for each model. This complements the average values in Table 4.1 by showing how consistent each model is across the dataset.

4.3.2 Performance Across Categories

Table 4.2: Performance by Category

	Police			Rescue		
Model	Sim.	F1	Halluc.	Sim.	F1	Halluc.
Gemma 3n-E2B	72.00	48.03	47.75	76.70	54.24	35.16
Gemma 3n-E4B	72.30	50.80	42.39	77.48	54.05	36.72
Gemma 4-E2B	68.53	41.45	36.51	72.14	50.87	30.57
Gemma 4-E4B	71.08	42.37	40.77	76.10	50.57	36.61

Table 4.2 shows that Rescue scenes achieve higher semantic similarity and F1 scores than Police scenes across all models, while also producing lower hallucination rates. This suggests that Rescue scenes are easier to describe consistently, likely because important elements such as damaged vehicles, fire, smoke, water, or rescue activity are usually more visually distinct. Police scenes are more challenging due to overlapping people, body-camera motion, and actions that are harder to infer from a single frame.

4.3.3 Impact of Image Quality

Table 4.3: Semantic Similarity and Entity-Level F1 by Image Quality

Model	Clear	Blurry	Dark	Obstructed
Gemma 3n-E2B	75.09 / 54.35	73.88 / 46.42	75.92 / 48.89	72.44 / 53.22
Gemma 3n-E4B	74.87 / 52.47	74.72 / 52.95	75.85 / 47.04	74.27 / 56.36
Gemma 4-E2B	68.82 / 48.03	74.39 / 44.61	71.50 / 39.62	67.80 / 50.40
Gemma 4-E4B	72.90 / 46.59	75.31 / 39.81	74.72 / 42.57	72.07 / 55.66

Table 4.3 reports semantic similarity and entity-level F1 for each image-quality group. Each cell is shown as semantic similarity / F1. Semantic similarity is the main metric, while F1 is included as a secondary indicator of entity keyword overlap.

The results show that image quality affects model behavior, but the pattern is not uniform across all models. Blurry and dark images often reduce F1, suggesting that low-quality images make it harder for the models to extract specific scene details. However, semantic similarity remains relatively stable, meaning that the generated descriptions can still be broadly close to the ground truth even when exact entity overlap is lower.

The Gemma 4 models generally achieve lower semantic similarity than the Gemma 3n models, especially Gemma 4-E2B. This is likely because the Gemma 4 models produce shorter and more conservative descriptions and avoid including details they are not certain about, a behavior that is more apparent in the smaller variant. As a result, they miss some information present in the ground-truth summaries, which lowers semantic similarity and recall even when the generated content is not incorrect.

Since the dataset is small and the F1 score is based on keyword overlap, these results should be interpreted in a comparative manner.

4.3.4 Findings from Description Evaluation

Overall, there is a clear pattern:

- Gemma 3n models give more complete descriptions but also include more errors, since they tend to include more entities and details from the scene.

- Gemma 4 models are more careful, making fewer errors but missing more details. As a result, they often produce more ambiguous descriptions, as they avoid including uncertain entities or interpretations.

4.3.5 Retrieval Performance

The retrieval system uses the generated descriptions to search for relevant images.

Table 4.4: Retrieval Performance per Model

Model	P@5	R@5
Gemma 3n-E2B	0.380	0.761
Gemma 3n-E4B	0.353	0.691
Gemma 4-E2B	0.287	0.523
Gemma 4-E4B	0.353	0.661

This result is consistent with the findings above. The Gemma 3n models generally produce more detailed descriptions, which can provide more semantic cues for retrieval. In contrast, the more conservative Gemma 4 models often omit uncertain details. While this can reduce unsupported entities, it may also make the generated descriptions less useful for retrieving relevant images.

4.3.6 Runtime Performance Evaluation

In addition to generation quality and retrieval performance, the runtime behavior of each model was evaluated during on-device inference.

The runtime evaluation focuses on inference latency, token generation throughput, and time to first token. Due to hardware limitations and runtime stability constraints on the evaluation device, Gemma 3n-E4B and Gemma 4-E4B were executed only on CPU, while Gemma 3n-E2B and Gemma 4-E2B were executed on both CPU and GPU.

Table 4.5: Runtime Performance per Model

Model	Backend	Avg. Time (s)	Tokens/s	TTFT (s)
Gemma 3n-E2B	CPU	19.00	11.99	6.25
Gemma 3n-E4B	CPU	26.85	8.04	7.65
Gemma 4-E2B	CPU	15.20	19.44	6.51
Gemma 4-E4B	CPU	37.10	7.18	16.20
Gemma 3n-E2B	GPU	18.34	10.90	1.70
Gemma 4-E2B	GPU	9.23	21.16	0.84

Table 4.5 shows clear differences in runtime performance across models and execution backends. In the comparable E2B setting, Gemma 4-E2B is faster than Gemma 3n-E2B on both CPU and GPU. GPU execution also greatly reduces time to first token for both E2B models, making

the system more responsive. The E4B variants are slower on CPU, especially Gemma 4-E4B, which makes them less practical for real-time on-device use in this setup.

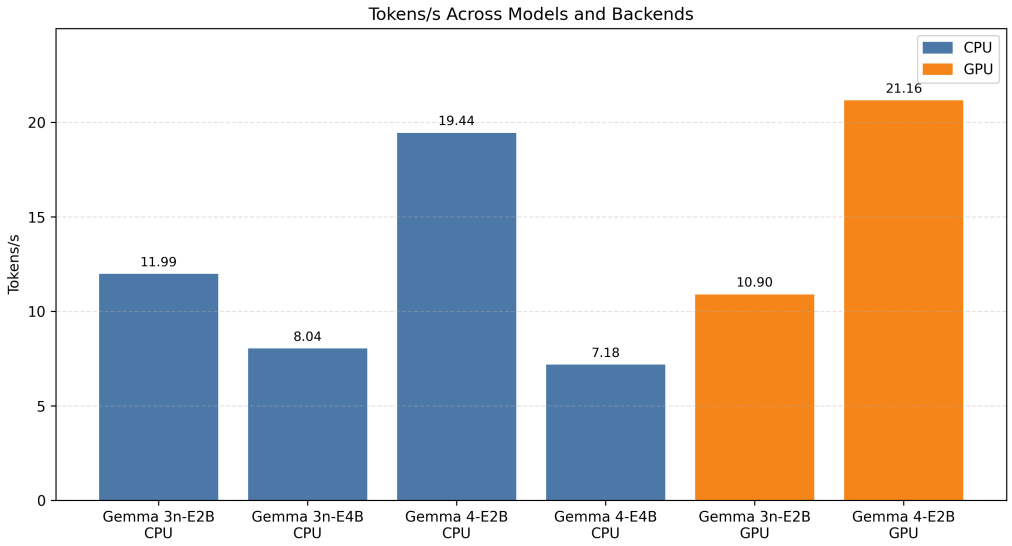


Figure 4.3: Tokens/s Across Models and Backends

4.4 Discussion

The results show a clear trade-off between detail and caution. The Gemma 3n models usually produce more detailed descriptions. This helps semantic similarity and retrieval, because the generated text contains more useful information for search.

The Gemma 4 models are more conservative. They include fewer uncertain details, which lowers hallucination, but they also miss more relevant information. This explains their lower recall and weaker retrieval performance.

Rescue scenes are easier for the models than Police scenes. Rescue images often contain clearer visual cues, such as damaged vehicles, fire, smoke, or water. Police scenes are harder because they often contain overlapping people, motion blur, and actions that are difficult to understand from a single frame.

Image quality also affects the results. Blurry and dark images make it harder to extract specific details, although semantic similarity often remains relatively stable.

In the comparable E2B setting, the average inference times of Gemma 3n-E2B and Gemma 4-E2B are relatively close on CPU, and GPU. The main benefit of GPU execution is the much lower time to first token, which makes the system feel more responsive during use.

Overall, no single model is best for every part of the system. Gemma 3n is stronger when detailed descriptions and retrieval coverage are important, while Gemma 4 is better when more

conservative output and faster inference are preferred.

5 Conclusion and Future Work

5.1 Summary of the Work

This work presents an end-to-end on-device system for transforming visual observations into semantically searchable representations. The system combines vision-language description generation, embedding, and local vector storage for offline retrieval.

The proposed approach was implemented in a fully offline mobile environment and evaluated using different model variants, scene types, and image qualities. The evaluation focused on how well generated descriptions support semantic similarity and retrieval.

5.2 System Contributions

This work develops an end-to-end on-device system that transforms visual observations into semantically searchable representations through vision-language description generation, embedding, and local vector storage for offline retrieval.

The primary contributions of this work include:

- A fully offline multimodal pipeline using the LiteRT runtime for on-device image description generation and embedding without cloud dependency.
- A lightweight prompting strategy for small vision-language models that enforces structured, retrieval-ready outputs suitable for semantic indexing.
- A retrieval-augmented generation (RAG) pipeline that enables citation-grounded conversational responses over the locally stored visual description embeddings.
- A comparative evaluation of four Gemma models in on-device scenarios, analyzing trade-offs between descriptive quality, factual consistency, and retrieval performance.

5.3 Key Findings from Evaluation

Overall, the evaluation shows a trade-off between descriptive detail and output caution. Gemma 3n produces more detailed descriptions, which improves semantic similarity and retrieval performance, while Gemma 4 is more conservative, reducing hallucinations but also lowering recall.

In general, more detailed outputs benefit retrieval-based tasks, while more cautious generation improves reliability. Across models, no single approach performs best in all settings, as each model reflects a different balance between coverage and precision.

5.4 Future Work

Several directions could further extend this work and improve both system performance and evaluation robustness.

A key area for future improvement is the development of a more comprehensive evaluation framework. The current evaluation primarily relies on a relatively small, manually curated dataset and heuristic-based metrics for hallucination and factual consistency. While this provides useful insights, it does not fully capture real-world performance or long-term system behavior. Future work should incorporate larger and more diverse datasets, as well as more advanced evaluation methods such as learned factuality metrics, human-in-the-loop assessment, and scenario-based testing in realistic operational environments.

Another important direction is the use of fine-tuned vision-language models. In this work, general-purpose models from the Gemma family were used without task-specific adaptation. While these models demonstrate strong baseline performance, they are not explicitly optimized for emergency response scenarios. Fine-tuning could significantly improve entity recognition, reduce hallucination rates, and enhance consistency in structured scene descriptions. This would also likely improve retrieval performance by producing more semantically rich and relevant embeddings.

BIBLIOGRAPHY

[1] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *arXiv preprint arXiv:1712.05877*, 2017. [Online]. Available: https://arxiv.org/abs/1712.05877

[2] A. Sharma, H. Singh, and M. Pant, "Pixels to prose: A comprehensive survey of image captioning techniques with deep learning and generative artificial intelligence," *Neurocomputing*, vol. 667, p. 132385, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231225030577

[3] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[4] Gemma Team, "Gemma 3n," 2025. [Online]. Available: https://ai.google.dev/gemma/docs/gemma-3n

[5] ——, "Gemma 4," 2026. [Online]. Available: https://ai.google.dev/gemma/docs/core

[6] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[7] J. Lang, Z. Guo, and S. Huang, "A comprehensive study on quantization techniques for large language models," *arXiv preprint arXiv:2411.02530*, 2024.

[8] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2019, pp. 3982–3992. [Online]. Available: https://arxiv.org/abs/1908.10084

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[10] Google DeepMind, "Embeddinggemma," https://deepmind.google/models/gemma/embeddinggemma/, 2026.

[11] R. Bernardi, R. Cakici, D. Elliott, A. Erdem, E. Erdem, N. Ikizler-Cinbis, F. Keller, A. Muscat, and B. Plank, "Automatic description generation from images: A survey of

models, datasets, and evaluation measures," *Journal of Artificial Intelligence Research*, vol. 55, pp. 409–442, 2016.

[12] P. Anderson, B. Fernando, M. Johnson, and S. Gould, "Spice: Semantic propositional image caption evaluation," in *European conference on computer vision*. Springer, 2016, pp. 382–398.

[13] V. Raghavan, P. Bollmann, and G. S. Jung, "A critical investigation of recall and precision as measures of retrieval system performance," *ACM Trans. Inf. Syst.*, vol. 7, no. 3, p. 205–229, Jul. 1989. [Online]. Available: https://doi.org/10.1145/65943.65945

[14] A. Rohrbach, L. A. Hendricks, K. Burns, T. Darrell, and K. Saenko, "Object hallucination in image captioning," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 4035–4045. [Online]. Available: https://aclanthology.org/D18-1437/

[15] A. Jobin, M. Ienca, and E. Vayena, "The global landscape of ai ethics guidelines," *Nature machine intelligence*, vol. 1, no. 9, pp. 389–399, 2019.

[16] Google AI Edge, "Litert-lm: On-device language model inference," https://ai.google.dev/edge/litert-lm, 2024.

[17] ——, "Mediapipe genai: Retrieval-augmented generation (rag)," https://ai.google.dev/edge/mediapipe/solutions/genai/rag, 2024.

APPENDICES