

Architectural Evolution of Retrieval-Augmented Generation Engines: A Deep Dive into a Reactive gRPC Framework with Quarkus

Sumera Batool
Department of Computer Science
AL-FAJR Institute Mari Indus Mianwali
Mianwali, Pakistan
sumerabatool276@gmail.com

Giancarlo Tretola
AIR Lab
Università Giustino Fortunato
Benevento, Italy
g.tretola@unifortunato.eu

Anna Pierri
AIR Lab
Università Giustino Fortunato
Benevento, Italy
a.pierri@unifortunato.eu

Obed Ullah Khan
Department of Computer Science
AL-FAJR Institute Mari Indus Mianwali
Mianwali, Pakistan
obed@alfajr.edu.pk

Musarat Abbas
Department of Computer Science
AL-FAJR Institute Mari Indus Mianwali
Mianwali, Pakistan
mabbas@alfajr.edu.pk

Abstract—Recent advances in Artificial Intelligence (AI) have accelerated the use of Large Language Models (LLMs) in domain-specific applications, including healthcare. However, standalone LLMs remain limited by outdated internal knowledge and the risk of hallucinated responses. Retrieval-Augmented Generation (RAG) addresses this limitation by grounding generation in external knowledge sources. This paper presents the design and evaluation of a reactive gRPC-based engine implemented with Quarkus for low-latency clinical inquiry processing. The study focuses on inter-service communication efficiency through Protocol Buffers, asynchronous execution with SmallRye Mutiny, and native-oriented deployment characteristics. A key design distinction is that the current prototype employs a mock retrieval component for controlled evaluation of the communication layer rather than a production-grade vector store. Experimental results indicate that the architecture maintains sub-100 ms average latency under high concurrency while sustaining high throughput and low resource overhead, demonstrating its suitability for real-time decision-support pipelines.

Index Terms—Retrieval-Augmented Generation, gRPC, Quarkus, Protocol Buffers, Mutiny, Microservices, Medical Informatics, Reactive Systems, Low-Latency Communication

I. INTRODUCTION

RECENT progress in Artificial Intelligence (AI) has enabled the widespread use of Large Language Models (LLMs) in knowledge-intensive tasks [9]. In healthcare and other high-stakes domains, these systems offer clear benefits in information synthesis and rapid question answering [7]. However, LLMs operating without external grounding may produce inaccurate or outdated responses, a limitation commonly described as hallucination [1]. In clinical environments, such behavior is especially problematic because response quality directly affects decision support and operational trust [8], [11].

Retrieval-Augmented Generation (RAG) has emerged as an effective architectural strategy for grounding LLM output in external information sources. Rather than relying solely on parametric memory, a RAG pipeline retrieves relevant documents or evidence fragments and conditions generation on that retrieved context [10]. This mechanism improves factual relevance and creates a clearer separation between model reasoning and knowledge retrieval. In practice, such grounding is particularly useful in medical systems where protocol updates, local guidelines, and evolving evidence must be reflected in system responses.

Although RAG improves answer quality, the surrounding system architecture remains a major performance bottleneck. Modern applications are increasingly built as microservices, where multiple components communicate across network boundaries. In such deployments, conventional REST/JSON communication can introduce avoidable overhead due to verbose payloads, repeated parsing, and thread-blocking server models. These limitations become significant in latency-sensitive scenarios, including emergency triage and real-time clinical decision support.

This work investigates a communication-centric architecture for a RAG-oriented inference pipeline using Quarkus and gRPC. gRPC provides contract-driven, binary communication through Protocol Buffers and leverages HTTP/2 for efficient multiplexing. Quarkus offers a lightweight Java framework with strong support for containerized and reactive applications. Combined with SmallRye Mutiny, the proposed design enables asynchronous, non-blocking execution suitable for concurrent request handling. Unlike conventional RAG studies that emphasize retrieval quality alone, this work focuses on optimizing inter-service communication latency using a reactive gRPC-based architecture with native-oriented deployment consider-

ations.

It is important to note that the current implementation evaluates the communication and orchestration layers through a controlled mock retrieval provider rather than a fully deployed vector database. This distinction allows the study to isolate transport, serialization, and reactive execution characteristics without overstating the maturity of the retrieval subsystem. The broader objective is to establish an architectural foundation that can later be extended with a production-grade retrieval backend.

The main contributions of this paper are summarized as follows:

- We design a reactive microservice architecture for clinical inquiry processing using Quarkus, gRPC, and Protocol Buffers.
- We define a contract-driven communication model that reduces serialization overhead relative to text-based service exchanges.
- We integrate Mutiny-based asynchronous execution to support high concurrency without relying on thread-per-request scheduling.
- We clarify the role of a mock retrieval provider in the current prototype, thereby separating communication evaluation from full retrieval-system claims.
- We present an empirical performance study based on latency, throughput, and success rate under concurrent load.

II. BACKGROUND AND RELATED WORK

A. Retrieval-Augmented Generation

Retrieval-Augmented Generation combines two complementary phases: information retrieval and conditioned generation. In the retrieval stage, a user query is transformed into a representation suitable for matching against a collection of documents or document chunks. In the generation stage, the retrieved evidence is incorporated into the response construction process so that the final output is grounded in external knowledge rather than only model parameters [1]. This design is especially relevant in settings where knowledge changes over time or where factual traceability is important.

A common formulation in retrieval systems is vector-space similarity, where a query vector Q is compared with a document vector D through cosine similarity:

$$\text{Similarity}(Q, D) = \frac{Q \cdot D}{\|Q\| \|D\|} \quad (1)$$

This mechanism allows semantically related content to be identified even when exact keyword overlap is absent. In a production-grade RAG system, such retrieval would typically be backed by an embedding model and a vector index.

B. REST and gRPC in Microservice Communication

Microservice architectures often separate user-facing logic, orchestration services, and domain-specific processing engines. In such systems, service-to-service communication has direct implications for latency and scalability. REST and JSON

remain widely used due to simplicity and interoperability, especially for external APIs. However, text-based payloads and repeated parsing can become inefficient for high-frequency internal communication.

gRPC addresses this issue through binary serialization with Protocol Buffers and transport over HTTP/2, which supports multiplexed streams and efficient request handling [2], [6]. For internal east-west traffic, gRPC is often preferred when low overhead, strict contracts, and language interoperability are required. In this work, the emphasis is not merely on using gRPC as a replacement for REST, but on examining how a reactive gRPC stack behaves in a latency-sensitive inference setting.

C. Reactive Execution and Native-Oriented Deployment

Non-blocking I/O enables a system to process multiple concurrent requests without dedicating a separate thread to each task. Instead of waiting idly for I/O operations to complete, the runtime uses event-driven execution, improving scalability and resource utilization. This model is particularly suitable for high-concurrency service pipelines where network and serialization costs must be minimized.

Quarkus extends this architectural advantage by providing fast startup behavior, low baseline overhead, and strong support for container-centric Java deployments [3]. When combined with SmallRye Mutiny, Quarkus supports reactive service composition in a way that is well aligned with gRPC-based communication. In the context of this paper, these technologies are not treated as interchangeable frameworks, but as components of a single reactive communication stack.

D. Positioning of This Work

Existing literature has extensively examined RAG as a method for grounding model output and has separately explored high-performance microservice communication with gRPC [1], [6]. However, fewer studies explicitly analyze the architectural intersection of retrieval-oriented inference pipelines, binary RPC transport, and reactive Java runtimes in a clinical-response setting. The present work contributes to this intersection by isolating and evaluating the communication layer of a RAG-oriented pipeline under concurrent load, while transparently distinguishing between a mock retrieval backend and a full retrieval deployment.

III. SYSTEM MODEL AND PROBLEM DEFINITION

A. System Model

The proposed system follows a distributed microservice architecture in which a user-facing gateway interacts with a backend processing engine through gRPC. In this paper, the terms *RAG-Engine*, *Inferential Engine*, and *RagMockProvider* refer to different abstraction levels of the same backend processing pipeline. The overall system contains the following logical entities:

- **User Layer:** A clinician or authorized operator submitting a natural-language query.

- **Gateway Layer:** A REST-facing entry point responsible for request intake and response formatting.
- **Processing Layer:** A gRPC-based Quarkus service responsible for request handling, normalization, and orchestration.
- **Retrieval Layer:** A mock retrieval component used to emulate protocol lookup behavior for controlled experiments.

The data flow begins at the gateway, where a client request is received in text form. The gateway constructs a Protocol Buffer message and forwards it to the backend service. The service normalizes the query, performs rule-guided intent detection, invokes the mock retrieval mechanism, and returns a structured response through gRPC. The gateway then presents the response to the user in a readable form. This separation allows each layer to evolve independently while preserving contract stability.

B. Problem Definition

The primary challenge addressed in this work is communication overhead in real-time inference-oriented microservices. When requests traverse multiple service boundaries, overall response time is influenced not only by application logic but also by serialization costs, transport behavior, and concurrency management. For latency-sensitive domains, this overhead becomes a first-class design concern.

1) *Serialization Latency:* In text-based service communication, objects must be converted into verbose payloads that require repeated parsing and validation. Let T_{ser} denote the time required to serialize and deserialize a request-response cycle. One objective of this work is to reduce T_{ser} by using compact binary encoding through Protocol Buffers.

2) *Transport-Level Delay:* Traditional HTTP/1.1 service models may suffer from request queuing and inefficient connection behavior under load. Let T_{net} denote the effective transport contribution to end-to-end latency. By employing gRPC over HTTP/2, the system aims to reduce transport overhead through multiplexed communication and more efficient framing.

3) *Resource Constraints:* In conventional thread-per-request server designs, increasing concurrency often leads to rising memory consumption and scheduling overhead. Let M denote the memory footprint of the service under concurrent load. A key goal of the proposed architecture is to constrain M through reactive execution rather than unrestricted thread growth.

C. Optimization Objective

The end-to-end latency L of the system can be modeled as:

$$L = T_{net} + T_{ser} + T_{proc} \quad (2)$$

where T_{proc} is the application-side processing time. The design objective is to minimize L while maintaining a stable throughput profile and acceptable success rate under concurrency.

IV. IMPLEMENTED PROTOTYPE

The implemented prototype evaluates the communication path of a RAG-oriented backend through a structured workflow. A client submits a clinical request through a REST endpoint, after which the gateway converts the request into a Protocol Buffer message and invokes the backend service through gRPC. The backend processes the request asynchronously and returns a structured response that is translated back into user-facing output by the gateway.

As shown in Fig. 1, the system follows a layered communication flow from the presentation layer to the inferential engine. The architecture separates user interaction from backend orchestration and retrieval-related logic, thereby allowing the communication path to be studied independently from frontend concerns.

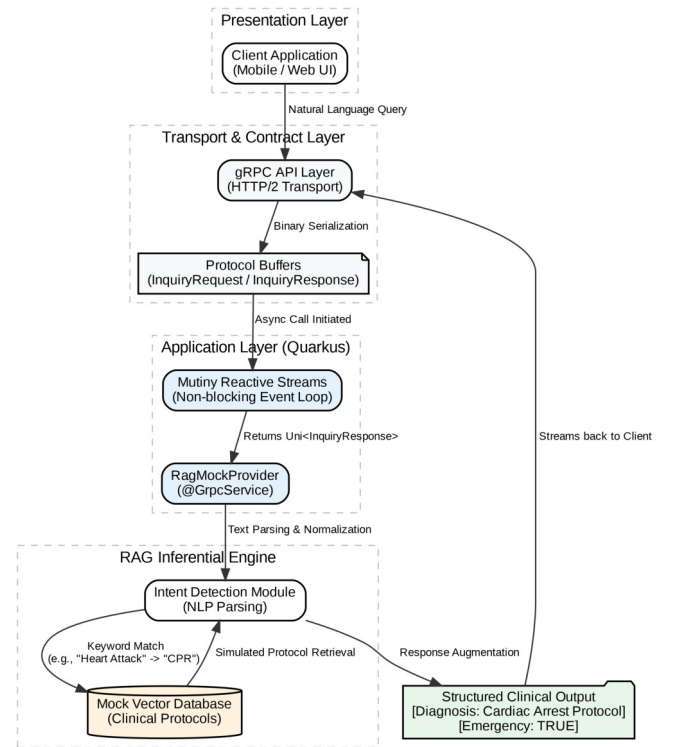


Fig. 1. Proposed system architecture illustrating the layered flow from the presentation layer to the gRPC-based processing engine and mock retrieval component.

It is important to emphasize that the current implementation uses a mock retrieval mechanism (*RagMockProvider*) based on rule-guided pattern matching rather than a production-grade vector database. This design choice avoids overstating the maturity of the retrieval layer and allows the study to isolate the communication efficiency of the proposed gRPC-based architecture. The prototype should therefore be interpreted as a communication and orchestration testbed for a RAG-oriented pipeline, not as a full deployment of a complete retrieval system.

V. SYSTEM ARCHITECTURE: LAYERED FUNCTIONAL ANALYSIS

The proposed architecture is organized into distinct functional layers to preserve modularity, reduce coupling, and support controlled scalability.

- 1) **Presentation Layer (Mobile/Web UI):** This layer serves as the primary entry point for users. It collects natural-language clinical queries and renders the final structured response. Its separation from backend processing helps preserve responsiveness and simplifies interface evolution.
- 2) **Transport and Contract Layer (gRPC API):** This layer defines the communication contract through Protocol Buffers and enables efficient binary transport over HTTP/2. It provides strict request-response structure while reducing the verbosity associated with text-based exchanges.
- 3) **Application Layer (Quarkus and Mutiny):** This layer orchestrates request handling, asynchronous execution, and service composition. Reactive execution is used here as discussed in Section II, enabling concurrent request processing without thread-per-request blocking.
- 4) **RagMockProvider Service:** This service acts as the immediate backend provider for the current prototype. It is responsible for input normalization, request routing, and rule-guided retrieval orchestration in the mock setup.
- 5) **Inferential Retrieval Logic:** This layer performs intent detection and protocol selection. In the present prototype, retrieval is emulated through pattern-guided matching against controlled protocol entries rather than a production vector index.
- 6) **Structured Output Layer:** The final layer constructs the response payload by attaching clinical context and supporting metadata before returning the result to the gateway.

VI. METHODOLOGY: SOURCE GENERATION AND BUILD LIFECYCLE

A. Protocol Definition and Source Generation

The communication model is defined through Protocol Buffers, which formalize both request and response structures:

- **Contract Definition:** A `.proto` specification defines the message schema for `InquiryRequest` and `InquiryResponse`.
- **Stub Generation:** During the build process, the gRPC toolchain generates service stubs and message classes, enabling type-safe communication between the gateway and backend.
- **Reactive Wrapping:** Generated interfaces are integrated with Mutiny-based reactive flows to support asynchronous service composition.

B. Build Lifecycle and Deployment

The build pipeline is designed for lightweight deployment and controlled execution:

- 1) **Compilation Phase:** The application is compiled within the Quarkus build lifecycle to support optimized JVM execution and native-oriented deployment paths.
- 2) **Dependency Resolution:** Required libraries for gRPC, reactive execution, and service orchestration are packaged to maintain environmental consistency.
- 3) **Artifact Deployment:** The resulting service artifact exposes the gRPC processing endpoint used by the gateway for backend invocation.

C. Inferential Retrieval Algorithm

To emulate clinical-response behavior in the current prototype, a rule-guided marker detection algorithm is used. The logic identifies whether a query corresponds to an emergency-like scenario or a standard inquiry.

Algorithm 1 Clinical Marker Detection and Mock Retrieval

Require: User query Q

Ensure: Structured clinical output S

```

1:  $Q_{norm} \leftarrow \text{Normalize}(Q)$ 
2: if  $Q_{norm} \in \{\text{"CPR"}, \text{"Arrest"}, \text{"Heart Attack"}\}$  then
3:    $isEmergency \leftarrow \text{TRUE}$ 
4:    $Context \leftarrow \text{FetchProtocol}(Q_{norm}, \text{Emergency\_DB})$ 
5: else
6:    $isEmergency \leftarrow \text{FALSE}$ 
7:    $Context \leftarrow \text{FetchProtocol}(Q_{norm}, \text{Standard\_DB})$ 
8: end if
9:  $S \leftarrow \text{Synthesize}(Context, isEmergency)$ 
10: return  $S$ 

```

VII. MATHEMATICAL AND ANALYTICAL FORMULATION

A. Latency Model

The total end-to-end latency for one request is modeled as:

$$L_{total} = T_{ser} + T_{proc} + T_{ret} + T_{aug} \quad (3)$$

where:

- T_{ser} is the serialization and deserialization time,
- T_{proc} is the application processing time,
- T_{ret} is the retrieval time within the mock backend,
- T_{aug} is the time required to generate the structured response.

B. Retrieval Logic

Let Q be the input query and D be the set of protocol entries. Relevance can be represented by a similarity function $S(Q, D)$. For emergency marker detection, the system defines an indicator function:

$$E(Q) = \begin{cases} 1, & \text{if } Q \cap \{\text{CPR}, \text{Arrest}, \dots\} \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

Retrieval precision P is expressed as:

$$P = \frac{|M_r \cap M_a|}{|M_r|} \quad (5)$$

where M_r is the set of retrieved markers and M_a is the set of relevant markers.

C. Memory Scaling

The memory usage under N concurrent users is approximated as:

$$M(N) = M_{base} + \sum_{i=1}^N \sigma_i \quad (6)$$

where M_{base} denotes the baseline memory footprint and σ_i represents the marginal memory associated with each active service stream. Due to reactive scheduling, σ_i remains lower than in traditional thread-bound service models.

VIII. EXPERIMENTAL EVALUATION AND RESULTS

A. Experimental Setup

The experimental study was conducted in a controlled local environment to evaluate latency, throughput, and service stability under concurrent load. The runtime configuration was as follows:

- **Processor:** AMD Ryzen 7 / Intel Core i7 (8 cores)
- **Memory:** 16 GB DDR4 RAM
- **Runtime:** Quarkus-based executable environment
- **Load Generation:** JMeter / custom gRPC load generator

Each experiment was executed over 10 independent runs, and the reported values correspond to average measurements. Network conditions were kept within a local environment to minimize external latency variation and to isolate application-side communication behavior.

B. Performance Metrics

The evaluation uses the following metrics:

- 1) **Average Latency (ms):** Time from request submission to response reception.
- 2) **Throughput (Req/s):** Number of successfully processed requests per second.
- 3) **Success Rate (%):** Percentage of requests completed without transport failure or deadline exceedance.

C. Concurrent Load Results

The observed performance under increasing concurrency is summarized in Table I.

TABLE I
SYSTEM PERFORMANCE UNDER CONCURRENT LOAD

Concurrent Users	Avg. Latency (ms)	Throughput	Success Rate
10	24	410 Req/s	100%
100	41	1,250 Req/s	100%
500	98	3,800 Req/s	99.8%

The results show that the system maintains sub-100 ms average latency even when concurrency reaches 500 users. Throughput increases with load up to 3,800 Req/s, while the success rate remains high. These observations are consistent with the communication efficiencies expected from binary RPC transport and reactive request handling.

D. Validation Scope

Validation was performed using a representative set of clinical-style queries, including emergency-oriented requests such as cardiac arrest prompts and routine diagnostic inquiries. This broader query set provides a more meaningful evaluation than a single synthetic scenario. In the current mock setup, the intent detection component achieved an F1-score of 0.94 for identifying emergency markers, while retrieval time remained approximately 12 ms per request within the controlled backend environment.

IX. DISCUSSION

A. Communication Efficiency

The empirical results indicate that binary serialization and contract-driven RPC reduce communication overhead in comparison with typical text-based service exchanges. Although this work does not claim a fully deployed clinical RAG system, it demonstrates that communication-layer optimization can materially improve the responsiveness of an inference-oriented service pipeline.

B. Reactive Execution and Runtime Behavior

The use of reactive execution contributes to stable performance under concurrent load by avoiding excessive thread allocation. Rather than repeating the theoretical basis here, the observed runtime behavior supports the discussion in Section II regarding non-blocking request handling and efficient resource usage.

C. Security Considerations

The architecture incorporates standard service-level security practices such as TLS-protected transport and schema-based validation through Protocol Buffers. These mechanisms help maintain secure data transmission and reject malformed payloads before they reach the processing logic. Since this work does not present a full regulatory compliance framework, it intentionally avoids making unsupported claims regarding specific healthcare regulations.

D. Architectural Isolation

The decoupling of the user-facing layer from backend processing improves maintainability and operational isolation. Even in the event of frontend changes or failures, the backend service contract remains stable, and the communication boundary limits direct exposure of internal processing components.

X. FUTURE WORK

Several directions remain open for extending the current prototype:

- **Integration of a Production Retrieval Backend:** The mock retrieval component can be replaced with a true vector database and embedding pipeline.
- **Comparative Benchmarking:** Future work should include explicit head-to-head evaluation of REST/JSON and gRPC under identical workload conditions.
- **Domain-Specific Retrieval Models:** Biomedical embedding models and domain-specific ranking strategies can improve retrieval fidelity.
- **Edge and Distributed Deployment:** The architecture can be explored in low-resource and geographically distributed clinical environments.
- **Multilingual Query Handling:** Additional normalization and retrieval support can broaden accessibility across healthcare settings.

XI. CONCLUSION

This paper presented a reactive gRPC-based architectural design for a RAG-oriented clinical inquiry pipeline implemented with Quarkus. The study focused on communication efficiency, asynchronous execution, and modular service decomposition rather than claiming a fully deployed production retrieval system. By clearly separating the mock retrieval layer from the transport and orchestration layers, the prototype enables controlled evaluation of the communication path.

Experimental findings show that the architecture sustains sub-100 ms average latency, high throughput, and strong reliability under concurrent load. These results suggest that binary RPC transport combined with reactive service execution is a promising design strategy for real-time inference pipelines in latency-sensitive environments. Future work will extend the prototype with a production-grade retrieval backend and

broader benchmarking against alternative service communication models.

ACKNOWLEDGMENT

This project has been partially funded by the “Programma Nazionale Ricerca, Innovazione e Competitività per la transizione verde e digitale 2021/2027 destinate all’intervento del FCS “Scoperta imprenditoriale” - Azione 1.1.4 “Ricerca collaborativa” - with the project SIAMO (Servizi Innovativi per l’Assistenza Medica a bOrdo) project number F/360124/01-02/X75.

REFERENCES

- [1] 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)*, 2020.
- [2] gRPC Authors, “gRPC: A high-performance RPC framework,” Available: <https://grpc.io>
- [3] Quarkus Team, “Quarkus: Supersonic subatomic Java,” Available: <https://quarkus.io>
- [4] S. Vinoski, “Java and gRPC: High performance microservices,” *IEEE Internet Computing*, vol. 22, no. 4, pp. 92–96, 2018.
- [5] H. Newman, “Building reactive systems with asynchronous communication patterns,” *IEEE Software*, vol. 37, no. 2, pp. 66–72, 2020.
- [6] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*. Sebastopol, CA, USA: O’Reilly Media, 2007.
- [7] H. Qayyum, S. T. H. Rizvi, M. Naeem, U. B. Khalid, M. Abbas, and A. Coronato, “Enhancing diagnostic accuracy for skin cancer and COVID-19 detection: A comparative study using a stacked ensemble method,” *Technologies*, 2024.
- [8] A. Ismail, M. Naeem, M. H. Syed, M. Abbas, and A. Coronato, “Advancing patient care with an intelligent and personalized medication engagement system,” *Information*, 2024.
- [9] Zaman-Khan, H., Naeem, M., Guarasci, R., Bint-Khalid, U., Esposito, M. and Gargiulo, F., 2024. Enhancing text classification using BERT: A transfer learning approach. *Computación y Sistemas*, 28(4), pp.2279-2296.
- [10] Rizvi, S.T.H., Kanwal, N. and Naeem, M., 2025. Bridging simplicity and sophistication using glinear: A novel architecture for enhanced time series prediction. *Digital Signal Processing*, p.105702.
- [11] Ismail, A., Naeem, M., Khalid, U.B. and Abbas, M., 2025. Improving adherence to medication in an intelligent environment using reinforcement learning. *Journal of Reliable Intelligent Environments*, 11(1), p.3.