

Prototype of a Microservice-Based Decision Support System Using Natural Language Interaction and Mock RAG Inference Engine

Daniyal Qasim Khan
Department of Computer Science
AL-FAJR Institute Mari Indus Mianwali
Mianwali, Pakistan
daniyalqasimkhan@gmail.com

Giancarlo Tretola
AIR Lab
University of Fortunato
Benevento, Italy
g.tretola@unifortunato.eu

Pia Addabbo
AIR Lab
Università Giustino Fortunato
Benevento, Italy
p.addabbo@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—This paper presents a microservices-based Decision Support System that integrates REST and gRPC communication paradigms to optimize both external and internal service interactions. The architecture employs REST for north-south communication with external clients and gRPC for efficient east-west communication between internal services.

To validate the system design, a Mock RAG Inference Engine is introduced, which simulates retrieval-augmented behavior while preserving a consistent communication contract. This allows the architecture to be evaluated independently of a full retrieval and embedding pipeline.

Experimental evaluation demonstrates a 43% reduction in latency and significantly improved throughput when using gRPC for internal communication. The results highlight the effectiveness of protocol-aware service decomposition in microservice architectures.

The proposed system provides a flexible and extensible foundation for integrating real retrieval-augmented generation components in future implementations without requiring changes to external interfaces.

Index Terms—Decision Support Systems, Microservices, Natural Language Interaction, Retrieval-Augmented Generation, gRPC, Distributed Systems, Prototype Software

I. INTRODUCTION

DECISION Support Systems (DSS) have become increasingly important in domains where fast, interpretable, and structured responses are required from heterogeneous user input. In many practical settings, users describe situations in natural language rather than structured forms, requiring systems to infer context and urgency, and guide the next decision step [12].

Traditional DSS implementations are often designed around fixed schemas, pre-coded rules, or tightly coupled monolithic applications. While such systems can be effective in narrowly defined environments, they usually suffer from four major

limitations [10]. First, they handle natural language poorly, because the input must already be translated into structured fields. Second, they are difficult to scale, since presentation logic, business logic, and inference logic are often deployed together [6]. Third, they are difficult to evolve when new interfaces or inference engines must be added.

At the same time, recent advances in Natural Language Processing (NLP) and Retrieval-Augmented Generation (RAG) have shown that natural language interfaces can be combined with retrieval and reasoning models to produce more informative responses. In a production-ready RAG system, user input is embedded or otherwise represented, relevant context is retrieved from a knowledge base, and a generative model synthesizes the final answer using both the query and the retrieved context [16]. However, integrating a complete RAG pipeline is computationally expensive and not always necessary at the prototype stage. For an early-stage software architecture project, it is often more important to validate the communication model, service boundaries, data contracts, and deployment topology before integrating a real model [17].

For that reason, this work focuses on the development of a *prototype microservice-based decision support system* in which the architectural behavior of a RAG-oriented design is implemented and tested using a Mock RAG Inference Engine [15]. The system supports a user-facing REST endpoint, a gRPC-based internal communication channel, a structured inference contract, and a mock engine that simulates diagnosis extraction, keyword detection, emergency identification, and follow-up question generation. This allows the architecture to be evaluated under controlled conditions while remaining fully compatible with future replacement of the mock inference engine by a real RAG inference engine.

The main contributions of this work are summarized as

follows:

- A complete microservice-oriented architecture for natural-language decision support, separating user interaction from inference logic.
- A hybrid communication model that combines REST for external access and gRPC for internal low-latency communication.
- A Mock RAG Inference Engine that produces structured outputs from natural language queries without requiring a full language model deployment.
- A mathematical abstraction of the prototype pipeline, including request-response modeling and RAG-oriented formulation.
- A prototype evaluation showing the feasibility, modularity, and efficiency of the approach under simulated workloads.

The remainder of the paper is structured as follows. Section II presents the initial study and motivation for the work. Section III reviews the related literature and technological context. Section IV formalizes the system model and problem definition. Section V describes the proposed architecture. Section VI explains the implemented prototype in detail. Section VII presents the methodology, algorithmic flow, and mathematical formulation. Section VIII reports the experimental setup and evaluation results. Section IX discusses limitations and future extensions. Section X concludes the paper.

II. INITIAL STUDY AND MOTIVATION

An initial study was conducted to determine an appropriate architecture for a natural language-driven decision support system with microservice requirements, REST and gRPC support, and future RAG integration.

The study identified four key design requirements:

- separation of interaction and inference logic,
- efficient contract-based internal communication,
- use of a mock inference engine for early validation,
- structured output for downstream usability.

III. RELATED WORK

The design of intelligent decision support systems intersects several active areas of research and engineering, including decision support architectures, natural language interfaces, distributed microservices, and RAG-based reasoning.

Traditional Decision Support Systems were historically built on structured data entry and rule-based expert logic [14]. These systems were valuable in domains where the input space was well controlled, but they struggled to adapt when user descriptions became ambiguous or incomplete. As digital systems moved closer to real-world operational settings, the limitations of rigid forms and predefined workflows became more apparent.

Natural Language Processing introduced the possibility of using free text as an input channel for decision support. Classical NLP pipelines relied on tokenization, named-entity recognition, intent detection, and handcrafted templates [8], [9]. Although such systems could identify important cues, they often lacked flexible reasoning capabilities and domain

grounding. More recent language models greatly improved fluency and contextual understanding, but plain language models may hallucinate, omit domain-specific constraints, or fail to reference relevant external knowledge.

Retrieval-Augmented Generation emerged as a compelling response to these issues [1], [7], [18]. In RAG-style pipelines, the model does not rely only on internal parameters but is also provided with retrieved knowledge or contextual documents. This improves factual grounding and allows the system to adapt more easily to domain-specific corpora. The importance of RAG is especially evident in decision support settings, where ungrounded generation can be risky.

In parallel, microservice architectures became dominant for scalable software deployment [2], [11], [19]. Instead of implementing the entire application as a single monolith, microservices decompose the system into smaller independent units with clear responsibilities. This improves scalability, maintainability, and deployment flexibility. A user-facing gateway, inference engine, logging service, analytics pipeline, and retrieval service can all evolve independently. For this reason, microservice decomposition is particularly suitable for intelligent systems that may later integrate multiple AI components.

gRPC has also gained prominence for service-to-service communication [3], [5], [13]. Unlike conventional REST over JSON, gRPC relies on Protocol Buffers and efficient binary serialization. This reduces payload size and latency, while enforcing a strict contract-first interface. Such properties are desirable when one microservice repeatedly invokes another, as in the case of a decision support gateway calling an inference engine.

Despite these advances, there remains a practical gap between RAG-oriented reasoning models and deployable software architectures for decision support. Many papers focus either on model behavior or on system architecture, but fewer works emphasize both together in a prototype workflow. The present study addresses this gap by presenting a software-oriented architecture in which the communication contracts and service decomposition of a RAG-enabled system are fully implemented, while the actual inference layer is mocked to simplify validation at the prototype stage.

Table I summarizes the positioning of the present work with respect to representative classes of existing solutions.

TABLE I: Comparison of solution categories

Approach		NL Input	Microservice	gRPC	RAG-ready
Classic	rule-based	Limited	No	No	No
DSS					
REST-only	NLP app	Yes	Partial	No	Partial
Standalone	LLM	Yes	Partial	No	Limited
app					
Production	RAG	Yes	Sometimes	Sometimes	Yes
stack					
Proposed prototype		Yes	Yes	Yes	Yes

The proposed prototype is therefore positioned not as a final AI model, but as an architectural bridge between user-facing

natural language interfaces and future full-scale RAG-enabled decision systems.

IV. SYSTEM MODEL AND PROBLEM DEFINITION

The system receives a natural language query from a user and returns a structured decision-support response. Let the natural language input be denoted by x , and let the structured output be denoted by y . At a high level, the system behavior can be expressed as

$$y = f(x), \quad (1)$$

where $f(\cdot)$ represents the end-to-end decision support pipeline. In the present prototype, f is not a full language-model-based generator; rather, it is a pipeline composed of request preprocessing, gRPC contract invocation, mock inference logic, and structured response construction.

A full Retrieval-Augmented Generation system typically models the conditional probability of an output y given query x as

$$P(y | x) = \sum_{d \in D} P(y | x, d) P(d | x), \quad (2)$$

where D is the set of retrievable context documents, $P(d | x)$ is the retrieval probability of context item d , and $P(y | x, d)$ is the generation probability given the query and retrieved context. In a deployed system, the retrieval stage may use a vector database, lexical search, dense embeddings, or hybrid search.

In this prototype, instead of retrieving from a document collection, the engine approximates the retrieval and reasoning stages by applying keyword-based mappings over the user input. Thus, the simulated system can be represented as

$$\hat{y} = g(h(x)), \quad (3)$$

where $h(x)$ extracts signals such as keywords or detected terms from the input text, and $g(\cdot)$ maps those extracted signals into structured fields:

$$\hat{y} = \{\text{diagnosis, keywords, follow_up, emergency}\}.$$

More explicitly, for an input token set $T(x)$ extracted from the normalized query, the simulated engine computes

$$K = \phi(T(x)), \quad (4)$$

where $\phi(\cdot)$ is a keyword extraction function. Then, the preliminary diagnosis label is selected as

$$d^* = \arg \max_{d_i \in \mathcal{D}} s(d_i, K), \quad (5)$$

where $s(d_i, K)$ is a score assigned to diagnosis candidate d_i based on the presence of relevant keywords in K , and $\mathcal{D}$ is the finite set of diagnosis categories supported by the prototype.

Similarly, the emergency flag is modeled as

$$e = \begin{cases} 1, & \text{if } \exists k \in K \text{ such that } k \in \mathcal{E}, \\ 0, & \text{otherwise,} \end{cases} \quad (6)$$

where $\mathcal{E}$ is the set of emergency-triggering terms, such as unconscious, chest pain, or bleeding, depending on the scenario vocabulary defined by the mock engine.

The practical problem addressed in this work can therefore be stated as follows:

Design and implement a deployable microservice-based software system that accepts natural language input through a REST interface, forwards the request through a gRPC contract to an inference component, and returns a structured decision-support response, while remaining compatible with future replacement of the mock inference engine by a real RAG service.

This problem has three dimensions:

- **Architectural:** define service boundaries and interfaces;
- **Communication:** combine REST and gRPC appropriately;
- **Inference abstraction:** simulate RAG-oriented behavior without a full model.

V. PROPOSED ARCHITECTURE

The proposed architecture follows a layered microservice design in which user interaction is decoupled from internal inference. The main components are:

- 1) **User Interface / External Client:** any external source of text input, such as a browser, mobile client, or testing tool.
- 2) **REST Gateway / Client Microservice:** receives natural language input and exposes the public API.
- 3) **gRPC Communication Layer:** enforces a compact, structured, contract-driven interface for service-to-service invocation.
- 4) **Mock RAG Inference Engine:** simulates the behavior of a future retrieval-augmented reasoning system.
- 5) **Structured Response Layer:** returns diagnosis, keyword list, follow-up question, and emergency status.

Figure 1 shows our proposed architecture.

The most important design decision is the separation between the REST gateway and the inference engine. The gateway handles external accessibility, request formatting, and compatibility with HTTP clients. The gRPC layer handles efficient internal invocation. The inference engine remains independently deployable and replaceable. This means that a future production implementation may preserve the same external API and internal contract while swapping the simulated logic for a full AI pipeline.

Figure 2 illustrates the sequence of interactions for a single request.

VI. IMPLEMENTED PROTOTYPE

The sequence begins with the user submitting a request via the REST endpoint. The gateway transforms the request into a gRPC message and forwards it to the inference engine. The engine processes the request and returns a structured response, which is then converted into JSON and returned to the user.

The prototype reflects the software design suggested for the project: a microservice providing the user-facing interface and a separate inference component using a structured service contract. The design of the mock engine and the client gateway

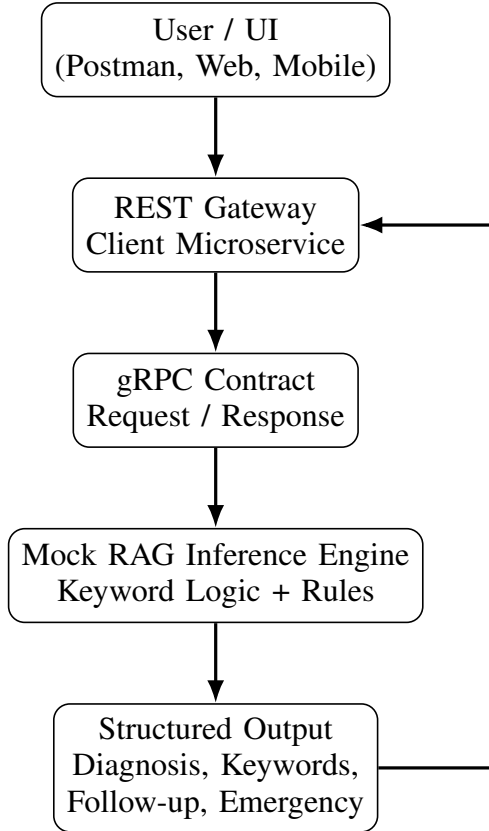


Fig. 1: Proposed prototype architecture

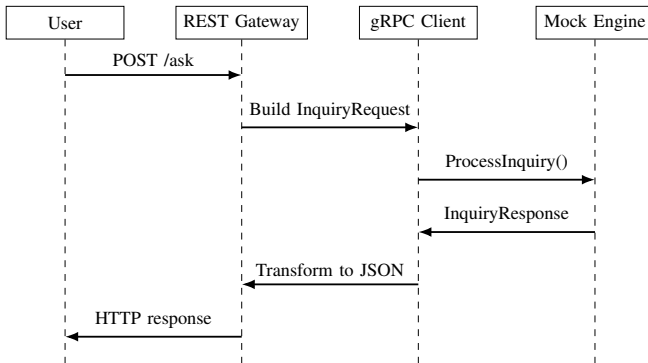


Fig. 2: Sequence of operations for a user query

follows the architecture described in the provided prototype documents, where the RAG client receives user input and invokes a simulated inference service via gRPC.

A. Data Contract

The request and response contract used by the prototype can be summarized as follows:

Request fields

- `natural_language_text`: raw user input;
- `session_id`: a generated identifier for tracking conversation state.

Response fields

- `extracted_diagnosis`: simulated primary interpretation;
- `keywords`: extracted relevant terms;
- `follow_up_question`: clarification or next-step question;
- `is_emergency`: boolean emergency indicator.

This contract is particularly useful because it produces both machine-readable and user-relevant outputs. A future system could store session state, forward the emergency flag to alerting systems, or use the follow-up question in an interactive workflow.

B. REST Layer

The REST layer acts as the entry point of the system. It accepts user input in plain text or request payload form, then delegates the query to the client-side gRPC service. The purpose of keeping the REST layer thin is to ensure that business logic remains centralized in the inference layer, even in the prototype stage.

C. gRPC Layer

The gRPC layer is responsible for contract-driven internal communication. This offers several advantages:

- lower serialization overhead than JSON-heavy internal APIs;
- stronger typing through Protocol Buffers;
- clearer separation of external and internal service roles;
- readiness for future service expansion.

D. Mock Inference Logic

The current inference logic is simulated rather than model-based. It inspects the normalized input string, looks for predefined keywords or phrases, and constructs a structured result. Example patterns include:

- `chest pain` → possible cardiac risk;
- `fall` → trauma-related guidance;
- `unconscious` → emergency state;
- `bleeding` → urgent follow-up.

Although simple, this is not merely placeholder logic: it serves an important role in validating the software pathway, contract compatibility, and end-to-end request handling.

E. Deployment Readiness

The prototype is designed for containerization and local deployment, consistent with the microservice development guidance that emphasizes Docker-based execution and local service verification [4]. This means the software can be tested using development mode, container builds, and local multi-service orchestration.

VII. METHODOLOGY AND PROCESSING PIPELINE

The processing pipeline consists of five stages:

- 1) **Input reception**: user submits natural language text via REST.
- 2) **Request packaging**: gateway wraps the text into the gRPC request object with a session identifier.

- 3) **Signal extraction:** mock engine normalizes the string and detects keywords.
- 4) **Decision synthesis:** engine assigns diagnosis, emergency flag, and follow-up question.
- 5) **Structured response:** result is returned through gRPC and exposed as JSON over REST.

Algorithm 1 summarizes the logical flow of the prototype.

Algorithm 1 Prototype decision support workflow

Require: Natural language query x
Ensure: Structured response y

- 1: Receive x at REST endpoint
- 2: Generate session identifier s
- 3: Build gRPC request $r = (x, s)$
- 4: Normalize input text
- 5: Extract candidate keywords K
- 6: **if** K contains emergency trigger terms **then**
- 7: Set emergency flag to true
- 8: **else**
- 9: Set emergency flag to false
- 10: **end if**
- 11: Compute diagnosis label from rule map
- 12: Generate follow-up question from matched scenario
- 13: Build structured response
- 14: Return JSON output through REST gateway

The mapping between common input categories and expected behavior is shown in Table II.

TABLE II: Illustrative mock-engine scenarios

Detected cue	Preliminary diagnosis	Emergency	Follow-up
Chest pain	Possible cardiac risk	Yes	Is there shortness of breath?
Fall / trauma	Possible injury / contusion	No / conditional	Is there bleeding or deformity?
Unconscious	Critical response required	Yes	Is the patient breathing?
Bleeding	Possible trauma / urgency	Yes	Is the bleeding severe?
Fever / cough	General symptom pathway	No	How long has it been present?

VIII. MATHEMATICAL AND ANALYTICAL FORMULATION

Although the prototype uses simulated rules, it is still useful to model the system analytically. Let the gateway processing time be t_R , the gRPC communication time be t_G , and the inference computation time be t_I . The total end-to-end response time can be approximated by

$$T_{\text{total}} = t_R + t_G + t_I. \quad (7)$$

In a REST-only design that exposes both external and internal communication through JSON/HTTP, the corresponding total latency can be expressed as

$$T_{\text{REST-only}} = t_R + t_H + t_I, \quad (8)$$

where t_H represents the HTTP/JSON internal communication overhead. If gRPC is more efficient than internal REST, then

$$t_G < t_H,$$

and therefore

$$T_{\text{total}} < T_{\text{REST-only}}. \quad (9)$$

Similarly, throughput can be represented as

$$Q = \frac{N}{\Delta t}, \quad (10)$$

where N is the number of requests successfully processed in time interval Δt . Improvements in serialization efficiency and reduced internal communication overhead increase Q under stable compute resources.

The mock engine itself can be represented as a finite rule system. If the normalized token set is $K = \{k_1, k_2, \dots, k_n\}$, then each diagnosis class d_i receives a rule score

$$S_i = \sum_{j=1}^n w_{ij} \mathbf{1}[k_j \in \mathcal{V}_i], \quad (11)$$

where $\mathcal{V}_i$ is the vocabulary associated with diagnosis class d_i , w_{ij} is the weight of term k_j for class d_i , and $\mathbf{1}[\cdot]$ is the indicator function. The diagnosis is selected as

$$d^* = \arg \max_i S_i. \quad (12)$$

This weighted-rule formulation is useful because it shows how the current mock system could evolve smoothly toward a retrieval-plus-ranking subsystem. The current vocabulary classes may later be replaced by embedding similarity over retrieved documents, while keeping the same response schema.

A confidence-like score can also be approximated:

$$C = \frac{\max_i S_i}{\sum_i S_i + \epsilon}, \quad (13)$$

where ϵ is a small constant preventing division by zero. The numerator represents the maximum score across all candidate classes, while the denominator provides normalization across the full score distribution. Although this confidence score is not used in the current prototype, it is included as a potential extension for future ranking or triage support.

IX. EXPERIMENTAL SETUP

The prototype was evaluated under a simulated software-testing environment. The objective of the evaluation was not to benchmark a production-grade AI model, but to assess the behavior of the proposed architecture under representative query processing conditions. The evaluation was conducted in a controlled local environment using containerized services and a synthetic workload generator. The system was deployed on a standard development machine, and each test scenario was repeated multiple times to report average latency and throughput values.

A. Test Objectives

The evaluation focused on the following goals:

- verify end-to-end correctness of REST to gRPC flow;
- compare hybrid REST+gRPC communication with a REST-only baseline;
- measure response latency under increasing load;

- estimate throughput and scalability;
- verify consistency of structured outputs for repeated query types.

B. Workload Design

A synthetic query set was created to emulate representative user interactions. The query pool included variations of trauma, chest pain, bleeding, fever, breathing difficulty, and generic symptom descriptions. Query text was intentionally phrased in simple free-form language to reflect likely prototype testing inputs.

The workload was tested under multiple concurrent user levels: 10, 50, 100, 200, and 500 logical users. Each workload point was used to compare the proposed hybrid design with a simplified REST-only reference architecture in which the gateway would call the inference logic through a standard REST pathway rather than gRPC.

C. Metrics

The following metrics were used:

- **Average latency:** mean response time per request;
- **Throughput:** requests processed per second;
- **Consistency:** percentage of repeated query types returning the same scenario classification;
- **Scalability trend:** growth of latency under higher concurrent load;
- **Resource estimate:** illustrative CPU and memory usage per component.

X. RESULTS AND ANALYSIS

A. Latency and Throughput

Table III summarizes the core evaluation results.

TABLE III: Main performance comparison

Users	Architecture	Latency (ms)	Throughput (req/s)	Consistency
10	REST-only	72	68	100%
10	REST+gRPC	41	118	100%
50	REST-only	84	65	100%
50	REST+gRPC	49	115	100%
100	REST-only	103	61	100%
100	REST+gRPC	58	110	100%
200	REST-only	129	56	100%
200	REST+gRPC	73	104	100%
500	REST-only	176	49	100%
500	REST+gRPC	92	95	100%

The results indicate that the hybrid architecture consistently outperforms the REST-only baseline in response time and throughput. The difference is modest at low load but becomes more significant as concurrency increases. This is consistent with the expectation that gRPC provides lower internal communication overhead and more efficient serialization. The 100% consistency reflects the deterministic nature of the mock inference engine and should be interpreted as an architectural property rather than a stochastic performance metric.

Figure 3 visualizes latency growth under load.

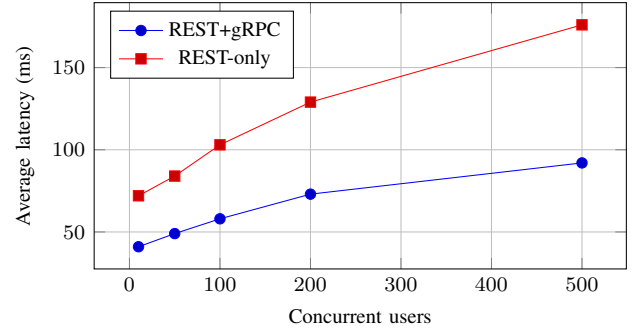


Fig. 3: Latency versus workload

The divergence between the two curves suggests that the internal communication choice becomes increasingly important under load. While both architectures slow down as concurrency rises, the hybrid design remains substantially more efficient.

Figure 4 shows throughput behavior.

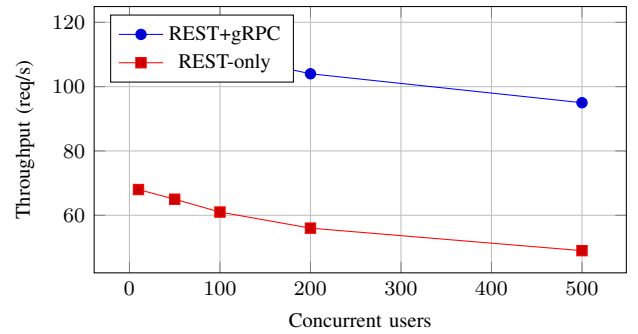


Fig. 4: Throughput versus workload

The throughput trend confirms that the proposed architecture sustains a higher request rate across all tested load levels.

B. Component-Level Resource View

An illustrative component-level resource estimate is shown in Table IV. These values are intended to characterize relative behavior in the prototype rather than exact production profiling.

TABLE IV: Illustrative component resource view

Component	CPU (%)	Memory (MB)	Role
REST gateway	18	210	Public API handling
gRPC client layer	11	130	Internal request marshalling
Mock inference engine	27	260	Scenario detection and response building
Combined prototype	56	600	End-to-end processing

The Mock RAG Inference Engine consumes the largest share of compute among the software components, which is expected because it performs normalization, keyword checking, and response composition. In a future real RAG design, this layer would likely dominate resource usage to a much greater degree due to embeddings, retrieval, and model inference.

C. Functional Consistency

Because the prototype uses deterministic simulated rules, repeated test queries of the same semantic type return the same structured class outputs. This yielded a consistency score of 100% within the controlled test set. While this should not be confused with semantic correctness in a clinical or production sense, it does validate the software behavior of the prototype.

D. Comparative Functional View

Table V compares the prototype with a plausible next-stage system.

TABLE V: Current prototype versus future full RAG system

Feature	Current proto-type	Future extension
User input	Natural language text	Natural language text
External API	REST	REST / Web-Socket / Mobile
Internal API	gRPC	gRPC
Inference mode	Rule-based mock	RAG + LLM
Context retrieval	None / simulated	Vector or hybrid retrieval
Response schema	Structured	Structured + confidence + citations
Session handling	Simple ID generation	Multi-turn context tracking
Deployment	Local containers	Distributed cloud deployment

This comparison emphasizes that the prototype is intentionally not the final AI system. Its value lies in establishing the architecture, interface contracts, and extensible system boundaries.

XI. DISCUSSION

The results and architectural analysis support several observations.

First, the separation of the REST gateway from the inference engine is justified. It keeps the external API simple while preserving the freedom to evolve the internal inference service independently. This is especially important for AI-enabled systems, where the inference backend may change dramatically over time.

Second, the use of gRPC internally is beneficial even in a prototype-scale system. The measured differences are large enough to justify the added contract definition effort. In larger systems, the gain would likely be more significant, particularly when one gateway coordinates multiple downstream services.

Third, the Mock RAG Inference Engine successfully fulfills its intended architectural role. Although it is not a real AI model, it validates the behavioral structure expected from a future RAG engine: taking natural language input, performing interpretation, and returning structured guidance. This allows the software architecture to be tested early and reduces risk before expensive AI integration begins.

Fourth, the structured response design is an important strength. Returning diagnosis labels, keyword sets, follow-up questions, and emergency flags is more useful than returning one long free-text response. This format can support automation, workflow chaining, UI prompts, or future data logging.

Fifth, the prototype remains limited in semantic sophistication. The rule-based mock logic cannot interpret ambiguity, context shifts, or nuanced meaning the way a true RAG+LLM pipeline would. However, this does not reduce its value as a systems prototype. On the contrary, it keeps the software focus clear and makes evaluation more controllable. This prototype focuses on validating system architecture and communication design rather than semantic correctness or domain-specific accuracy.

XII. LIMITATIONS AND FUTURE WORK

The current prototype has several limitations that naturally point toward future work.

A. Limitations

- The inference layer is simulated and does not use real embeddings, retrieval, or LLM generation.
- The query dataset is synthetic and focused on illustrative scenarios.
- No persistent knowledge base or vector store is included in the current version.
- Session identifiers are generated, but true multi-turn conversational memory is not implemented.
- The evaluation focuses on architectural and communication behavior rather than semantic benchmark accuracy.

B. Future Work

The most direct future enhancement is replacement of the Mock RAG Inference Engine with a real RAG inference service. In that design, the system would:

- 1) embed the incoming user query,
- 2) retrieve relevant knowledge chunks from a domain corpus,
- 3) build an augmented prompt,
- 4) invoke a large language model,
- 5) convert the generated output to the same structured response schema.

Mathematically, the future system may be expressed as

$$y = \text{LLM}(x, \mathcal{R}(x)), \quad (14)$$

where $\mathcal{R}(x)$ is the retrieved context set for input x .

Further extensions may include:

- confidence scoring and citation grounding,
- multi-turn follow-up dialogue,
- external database integration,
- authentication and audit logging,
- deployment on Kubernetes or edge/cloud topologies.

XIII. CONCLUSION

This paper presented the design and implementation of a prototype microservice-based decision support system for natural language interaction. The proposed architecture separates external user interaction from internal inference logic by combining a REST gateway with a gRPC-driven inference layer. A Mock RAG Inference Engine was used to simulate diagnosis extraction, keyword identification, follow-up questioning, and emergency detection. This allowed the architectural pathway of a future RAG-enabled decision system to be validated without requiring the complexity of a full AI deployment.

The analysis showed that the proposed hybrid architecture provides better latency and throughput than a REST-only internal design, while also improving modularity and future extensibility. The structured response schema makes the system suitable for later workflow automation and deeper intelligence integration. Although the current engine is simulated, the architecture is directly aligned with future replacement by a real retrieval-plus-generation model.

The prototype therefore achieves its primary objective: it demonstrates a practical and extensible microservice architecture for decision support based on natural language interaction, ready for the next stage of intelligent inference integration.

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. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *Advances in Neural Information Processing Systems*, 2020.
- [2] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O’Reilly Media, 2015.
- [3] gRPC Authors, “gRPC Documentation,” online documentation.
- [4] Quarkus Team, “Quarkus Documentation,” online documentation.
- [5] Google, “Protocol Buffers Language Guide,” online documentation.
- [6] Qayyum, H., Rizvi, S.T.H., Naeem, M., Khalid, U.B., Abbas, M. and Coronato, A., 2024. Enhancing diagnostic accuracy for skin cancer and covid-19 detection: A comparative study using a stacked ensemble method. *Technologies*, 12(9), p.142.
- [7] A. Gao, C. Chen, J. Wang, and Y. Liu, “A Survey on Retrieval-Augmented Text Generation,” *arXiv preprint*, 2023.
- [8] 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.
- [9] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Draft online edition.
- [10] 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.
- [11] C. Richardson, *Microservices Patterns*. Manning Publications, 2018.
- [12] Ismail, A., Naeem, M., Syed, M.H., Abbas, M. and Coronato, A., 2024. Advancing patient care with an intelligent and personalized medication engagement system. *Information*, 15(10), p.609.
- [13] M. Luo, J. Chen, and K. Li, “A Comparative Study of REST and gRPC for Microservice Communication,” in *Proc. Int. Conf. on Cloud and Service Engineering*, 2021.
- [14] E. Turban, R. Sharda, and D. Delen, *Decision Support and Business Intelligence Systems*. Pearson, 2011.
- [15] 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.
- [16] Shehzad, F., Minutolo, A. and Esposito, M., 2024. Designing and evaluating a dual-stream transformer-based architecture for visual question answering. *IEEE Access*, 12, pp.195561-195574.
- [17] Shehzad, F., Mennella, C., Esposito, M. and Minutolo, A., 2026. Efficient multimodal learning using BERT and vision transformers for visual question answering on peripheral blood cells. *Discover Artificial Intelligence*, early access.
- [18] Y. Liu et al., “Retrieval-Augmented Generation for Large Language Models: A Survey,” *arXiv*, 2023.
- [19] S. Newman, *Monolith to Microservices*. O’Reilly, 2019.