Baltic J. Modern Computing, Vol. 13 (2025), No. 3, 720-739
https://doi.org/10.22364/bjmc.2025.13.3.09

Private Microservice with Retrieval-Augmented Generation and Embedded LLM

Oleh CHAPLIA, Halyna KLYM

Department of Specialized Computer Systems, Lviv Polytechnic National University,
Bandera str., 12, Lviv, Ukraine, 79013

oleh.y.chaplia@lpnu.ua, halyna.i.klym@lpnu.ua

ORCID 0009-0005-9298-3538, ORCID 0000-0001-9927-0649

Abstract. This research introduces a modern private microservice architecture that integrates Retrieval-Augmented Generation and embedded local execution of Large Language Models based on MLX-LM, specifically optimised for Apple Silicon (M-series) hardware. By embedding RAG pipelines and LLM inference directly into a unified local microservice, the system achieves enhanced data confidentiality, private data, and embedding storage compared to conventional cloud-based deployments. Performance evaluations demonstrate that the architecture efficiently manages computational resources, maintaining availability even under intense workloads. Leveraging Apple's MLX framework, the solution attains great inference speed, reduced power consumption, and flexible deployment workflows. This study provides a practical foundation and clear guidelines for deploying secure, efficient, and privacy-centric AI microservices on local Apple Silicon infrastructure, highlighting significant opportunities for future research and industrial adoption.

Keywords: Cloud Computing, Microservices, Software Architecture, Artificial Intelligence, Large Language Models, Retrieval-Augmented Generation, Apple Silicon, MLX

1. Introduction

Modern Artificial Intelligence (AI) applications have witnessed a dramatic rise in the use of Large Language Models (LLMs) – extensive transformer-based networks trained on a massive corpus of data that excel in various natural language processing tasks (Guo et al., 2024). Their capacity to adapt to multiple domains, generate coherent responses, and engage in advanced reasoning has led to widespread deployment in fields such as healthcare, finance, and software development (Händler, 2023). Paired with microservices, LLMs can serve targeted, efficient, and maintainable solutions (Abgaz et al., 2023). Recently, systems have evolved even further towards serverless architectures, enabling highly scalable and event-driven deployment models where computational resources are dynamically allocated, significantly reducing operational overhead and further enhancing system agility and efficiency (Chaplia et al., 2024).

Furthermore, the Retrieval-Augmented Generation (RAG) concept has emerged as a powerful approach to mitigate LLM knowledge cutoffs and hallucinations by retrieving in external contexts or documents and then incorporating relevant facts into the generation pipeline (Liu et al., 2024).

Despite these advantages, private and secure RAG services are critical for organisations dealing with proprietary or regulated data (Josa and Bleda-Bejar, 2024). Traditional cloud-based deployments may inadvertently expose sensitive information, creating risks around data breaches and compliance violations. Security lapses can be damaging financially and in terms of reputation, especially in domains requiring strict data governance (e.g., healthcare, finance, and government). Consequently, the importance of privacy and security in RAG pipelines cannot be overstated, as even a single vulnerability in data handling or model inference could compromise entire microservice ecosystems (Buesser, 2024).

This push for secure, on-premises solutions has also heightened interest in specialised hardware and frameworks, such as Apple MLX (Feng et al., 2025). Unlike known platforms such as PyTorch, TensorFlow, and Keras, Apple MLX focuses on optimising computations specifically for Apple hardware (Apple Silicon) (Kenyon and Capano, 2022). Through native support for Core ML and tight integration with Apple's neural engine, MLX can offer significantly reduced latency, lower power consumption, and streamlined model deployment flows for macOS environments. For organisations seeking to host LLMs locally without reliance on external GPU servers, Apple MLX can further empower privacy-preserving RAG services, ensuring that model data and user queries remain confined to a secure, on-device environment.

Recent research underscores these emerging architectural concerns. Abgaz et al. (Abgaz et al., 2023) conducted a systematic review detailing how monolithic architectures can be decomposed into microservices. Their approach offers metrics and qualitative guidelines, emphasising that container-based and edge-oriented solutions could further enhance privacy in real-time scenarios. Building on the foundational idea of having small, independent services, we can structure minimal service containers that remain scalable and deployable even on on-premises hardware or local machines, thereby reducing external dependencies and potential data exposure. In parallel, "JAVIS Chat," presented by Aguirre and Cha (Aguirre and Cha, 2025), demonstrates how multi-LLM frameworks can remain cost-effective through distributed computing with Ray, yet might profit from container isolation and reduced runtime footprints to meet strict data-protection demands in sensitive environments. Focusing on the potential of smaller, domain-specific LLMs, Bucher and Martini (Bucher and Martini, 2024) highlight that fine-tuned models can outclass general zero-shot systems in classification tasks. Their toolkit lowers the entry barrier for practitioners seeking domain-focused solutions, though further emphasis on privacy and container-based secure deployments is needed. Meanwhile, Cheng et al. (Cheng et al., 2024) propose "RemoteRAG" to protect user embeddings during retrieval but note that containerisation could deliver greater data segmentation and isolation, especially for high-security use cases. Similarly, Chrapek et al. (Chrapek et al., 2024) show how Trusted Execution Environments (TEEs) can mitigate data leakage and intellectual property theft with minimal overhead, suggesting future synergy with container solutions for even finer-grained microservice isolation. Gunasekar et al. (Gunasekar et al., 2023) offer a complementary view, advocating for carefully curated "textbook" corpora to maintain model quality without ballooning parameter sizes – though container-based data handling could solidify data integrity within microservice ecosystems. Additional studies explore maintainability and scalability under real-world constraints. Hasan et al. (Hasan et al., 2023) provide structural metrics to measure how microservices affect development burdens, contending that secure container isolation can further improve compliance and data management. In the realm of edge computing,

Hossain et al. (Hossain et al., 2023) identify performance gains from microservices in resource-constrained settings, stressing that embedded encryption and robust orchestration can address data protection challenges at the network edge. The AIS model by Misnevs and Paskovskis introduces a modular, adaptive system for personalised information management, aligning well with the principles of private local microservices (Misnevs, 2025). Its architecture can be extended to integrate locally hosted LLMs, enabling intelligent, privacy-preserving data processing and user-specific automation. Lastly, Ieva et al. (Ieva et al., 2024) blend knowledge graphs, conversational interfaces, and RAG to manage energy infrastructure digital twins, arguing that container-based microservices could scale effectively and enhance security for sensitive operational data. These works highlight how microservices, containerisation, smaller LLM deployments, and privacy-preserving techniques converge to advance secure, efficient, and flexible AI systems.

Existing retrieval-augmented generation (RAG) deployments in privacy-sensitive contexts largely depend on cloud-hosted large language model (LLM) inference. This reliance introduces risks of data exfiltration and creates operational dependencies that are unacceptable for many organizations. Despite increasing adoption of Apple Silicon hardware and the rise of the MLX ecosystem, there is little empirical evidence on the feasibility and performance of fully local RAG deployments in such environments.

To address this gap, this paper makes several contributions. This research presents a modern, secure, privacy-preserving microservice that integrates Retrieval-Augmented Generation (RAG) with local LLM inference optimised for Apple Silicon devices. The system leverages the MLX framework to enable efficient, low-latency, and on-device execution, eliminating the need for external data processing. Packaged as a self-contained unit, the microservice adheres to core architectural principles such as modularity, independence, and maintainability. Embedding the entire RAG pipeline and LLM within a single deployment ensures high data confidentiality.

The work provides empirical characterization, showing how ingestion time and response latency scale with input/output tokens and documenting a low, stable memory footprint alongside deployment notes and operational guidance sufficient for implementation and deployment. Finally, it articulates the security and privacy motivation for on-device RAG and maps those concerns to the architectural decisions taken in the implementation.

The study provides a practical foundation for locally deployed AI services and highlights performance outcomes and future development directions.

Unfortunately, this paper has a limited scope of research. Therefore, gaps like maintainability, monitoring, platform comparisons, and deployment settings related to the proposed microservice are planned to be addressed in future works and extended evaluations.

2. Challenges and Drawbacks

Deploying LLMs locally within microservice architectures, especially on Apple Silicon hardware, offers significant benefits in privacy and performance but introduces several notable challenges (Kasperek et al., 2023). Decomposing monolithic systems into minimal, container-based microservices facilitates isolation and independent scalability, yet it can also complicate software maintenance and coordination. Each service container, particularly when embedding specialised LLM runtimes, demands careful integration, frequent updates, and ongoing management, increasing overall operational complexity.

Hardware limitations represent another critical challenge. Although Apple's M-series chips deliver substantial computational power and efficient neural processing, local deployment of large-scale models can still strain system resources. Memory constraints become evident as larger models or multiple simultaneous services approach the device's resource ceiling. Additionally, despite optimised GPU and neural processing unit (NPU) usage through frameworks like Apple MLX, concurrent demands from various microservices (e.g., embedding computation, retrieval operations, database interactions) can create bottlenecks and degrade performance.

Balancing privacy with scalability poses further difficulties. Locally storing sensitive or proprietary data significantly reduces exposure risk but increases the complexity of managing data at scale. Maintaining sufficient local resources becomes more challenging as user demands or data volumes grow, potentially leading to degraded service quality unless additional hardware or sophisticated autoscaling mechanisms are implemented. Effective containerisation that preserves performance and strict data isolation also requires substantial iterative tuning and specialised configuration.

The lifecycle management of locally deployed LLMs introduces additional complexity. Frequent model updates, retraining cycles, or adjustments to domain-specific embeddings necessitate downtime or intricate update mechanisms to maintain continuous availability. Ensuring compatibility and version control, especially when integrating security mechanisms like Trusted Execution Environments (TEEs) or encryption-at-rest solutions, adds layers of overhead and demands rigorous attention from development and operations teams.

Finally, while adopting specialised optimisation frameworks such as Apple MLX is beneficial for performance optimisation for Apple Silicon, it reduces portability compared to widely used frameworks like PyTorch or Keras. This diminished ecosystem support complicates integration with third-party libraries or advanced retrieval mechanisms, limiting the adaptability of locally hosted services. Achieving peak performance often involves manual, iterative tuning, requiring deep technical expertise in Apple's hardware-specific APIs and runtime tools.

Security and governance of embedded RAG pipelines further exacerbate complexity. Although local deployment reduces external exposure, the threat of internal attacks or compromised endpoints within the local network persists, necessitating extensive security hardening. Additionally, maintaining accurate and high-quality datasets for retrieval-augmented generation is challenging in isolated local environments, potentially increasing the risk of outdated or inaccurate outputs.

Embedding LLMs into local microservices on Apple M-series hardware significantly enhances privacy and data control. Still, it demands meticulous architectural design, rigorous hardware resource management, specialised performance tuning, and robust security protocols. Addressing these challenges requires deliberate strategies and comprehensive expertise to ensure sustainable and efficient AI-driven microservices.

3. Goal

The primary goal of this research is to design, implement, and evaluate a secure, privacy-preserving microservice architecture that integrates Retrieval-Augmented Generation (RAG) techniques with embedded local LLM execution. A modern version of this work lies in its comprehensive approach, which includes local RAG and embedded LLM

inference capabilities within a single package. This architecture is specifically optimised for deployment on Apple Silicon (M-series) devices, leveraging the unique computational advantages of Apple's hardware ecosystem.

A key objective is to create a modular, minimalistic microservice architecture that aligns closely with fundamental microservice principles such as independence, scalability, and maintainability. The architecture will facilitate streamlined, local deployment, reducing external dependencies and network latency, thereby significantly enhancing data confidentiality and operational resilience. The solution ensures the secure handling of sensitive data within a tightly controlled environment by embedding the entire RAG pipeline along with the LLM inference model. It involves directly integrating and optimising local LLM execution through the Apple MLX framework. This choice not only harnesses the specialised computational acceleration offered by Apple's M-series chips but also ensures lower latency, power efficiency, and improved inference performance. Such a targeted optimisation further supports a fully self-contained microservice capable of robust, real-time RAG operations without external data exposure. Performance evaluations will examine throughput, resource efficiency, and inference latency under realistic conditions.

The research delivers descriptive, practical software architecture and design for organisations seeking to implement privacy-preserving, locally executed RAG microservices. It provides the strengths and potential limitations of this approach of AI-driven microservice architecture deployed on Apple Silicon and similar edge-computing platforms. Also, the study sets the stage for future improvements.

4. Software Architecture and System Design

The proposed software architecture outlined in Figure 1 depicts a self-contained microservice designed specifically for deployment on MacOS with Apple Silicon (M-series chips). This architecture leverages containerisation principles by integrating local LLM execution, Retrieval-Augmented Generation (RAG), and supporting tools into a single microservice. This unified structure ensures tight integration, minimal overhead, and robust data privacy.

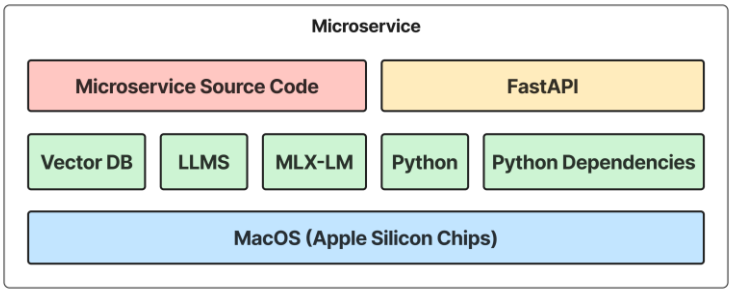

Figure 1. Layered microservice architecture

It is worth noting that the proposed microservice design follows key principles and best practices from microservice architecture. These include modular separation of concerns, clear interface and API boundaries, and isolated components for handling tasks

such as document processing, vector storage, and language model inference. The container image bundles all modules for on-device use, but components remain swappable. This preserves observability, fault isolation, and testability. As a result, the service remains simple, maintainable, and highly portable, making it well-suited for deployment in local or resource-constrained environments.

At the core of this architecture is the main source code of the microservice, implemented in Python and structured around the FastAPI framework. FastAPI provides the RESTful API endpoints, manages client requests, handles input validation, and orchestrates internal calls. Its asynchronous capabilities enable high concurrency and scalability, essential for responsive microservice operations.

Internally, the service integrates multiple components to deliver AI-powered functionality. First, the microservice embeds an LLM locally, directly utilising Apple's optimised MLX framework. MLX is designed to take advantage of Apple Silicon hardware, offering superior performance, lower latency, and enhanced power efficiency compared to more general frameworks such as PyTorch or Keras. The MLX-enabled LLM ensures that all inference processing occurs on-device, significantly enhancing data security by eliminating external communications during inference operations.

MLX-LM is a Python package that enables efficient LLM inference and fine-tuning on Apple Silicon using the MLX framework (Web, *Ml-Explore/Mlx-Lm*, 2025/2025). MLX-LM bridges the gap between modern models and on-device deployment, offering a streamlined path for running LLMs locally without reliance on cloud infrastructure. It integrates with the Hugging Face Hub (Jones et al., 2024), allowing users to download and deploy thousands of pre-trained LLMs with a single command. MLX-LM also supports model quantisation, reducing memory and compute requirements—an essential capability when running on devices with limited resources.

ChromaDB is the local vector database for the RAG operations (Web, *ChromaDB*, n.d.) that complements the embedded LLM. This database manages to embed storage and retrieval, allowing the querying of contextual data to support the model's responses. By collocating ChromaDB directly within the service, the system ensures secure access to contextual information, significantly enhancing privacy by ensuring data never leaves the local environment.

By default, ChromaDB uses the *"SentenceTransformerEmbeddingFunction"* from the sentence-transformers library as its embedding mechanism. This function wraps the pre-trained sentence embedding model *"all-MiniLM-L6-v2,"* widely known for providing efficient and high-quality vector representations of natural language texts.

This transformer-based model is optimised for creating dense embeddings that capture the semantic meaning of sentences or document chunks. The *"all-MiniLM-L6-v2"* model maps text inputs into a 384-dimensional vector space. It balances speed, accuracy, and computational efficiency, making it suitable for local and real-time RAG applications.

ChromaDB automatically handles embedding generation and vector insertion when using the default embedding function, enabling developers to have a straightforward, out-of-the-box experience. However, users can override this behaviour by defining a custom embedding function, such as one that calls an external API, integrates a local language model, or uses specialised domain-specific transformers.

When a user provides a natural language query string as a query to ChromaDB, the system uses the collection's embedding function to convert the text into a vector. This vector is then compared to the precomputed embeddings of stored documents or text chunks to identify the most semantically similar entries. The method returns the top n

results based on vector similarity, with options to include matching documents, metadata, and similarity scores. Filters can also be applied through metadata conditions to narrow the search scope. This querying capability is essential for RAG workflows, where relevant context must be fetched from a vector database before being passed to a language model for response generation.

By default, ChromaDB utilises L2 (Euclidean) distance as its similarity metric for vector searches (Web, *ChromaDB Docs*, n.d.). This means that when querying, ChromaDB calculates the straight-line distance between vectors to determine similarity, with smaller distances indicating higher similarity.

However, cosine similarity is often more appropriate for many natural language processing applications. Cosine similarity measures the cosine of the angle between two vectors, focusing on their orientation rather than magnitude (Jin and Lin, 2024). This is particularly useful when comparing text embeddings, as it accounts for the direction of the vectors, making it effective for identifying semantically similar texts regardless of their length. ChromaDB needs to be configured to use cosine similarity. The desired distance metric can be specified when creating a collection in the collection's metadata. It's important to note that ChromaDB returns cosine distance, which is calculated as *1* - cosine similarity. Therefore, to obtain the cosine similarity score from the returned distance, you would subtract the distance from 1. For instance, if the returned distance is *0.2*, the cosine similarity would be *1 - 0.2 = 0.8*.

ChromaDB was chosen because it functions as a *microdatabase*: a lightweight, in-process vector store with a low footprint that persists locally and requires no separate server or cluster. This embeds the store directly in our on-device microservice, simplifying packaging, installation, and reproducibility while keeping all data on the machine. By contrast, Neo4j is a full graph database with vector support that typically runs as a standalone service, bringing higher operational overhead and optimized primarily for rich relationship queries that are not required. Similarly, distributed or cloud vector databases (e.g., Milvus, Weaviate, Pinecone, Qdrant) offer scalability and advanced features but add unnecessary infrastructure and network dependencies for our offline target. We acknowledge that ChromaDB can be slower than specialized or distributed vector stores under larger collections or higher concurrency. Accepting this trade-off is intentional for the current scope. Performance and memory footprint remain as future optimizations in future research. We also accept that further research may motivate replacing ChromaDB with a more effective database. The storage layer is modular, so alternatives can be adopted with minimal changes.

All these components are orchestrated through a layer of clearly defined Python dependencies. These dependencies include the MLX libraries for model execution, FastAPI and its middleware for web service interaction, vector database clients such as ChromaDB, and supporting data processing utilities. Notably, the project uses the UV package manager (Web, *Astral-Sh/Uv*, 2023/2025), a fast and modern alternative to pip. Uv offers fast dependency resolution and efficient caching, significantly reducing build times and improving reliability in containerised environments. Its integration contributes to faster image creation and more predictable runtime behaviour. Finally, the entire architecture is encapsulated within a single service.

Overall, this architectural design addresses considerations of local execution, performance optimisation on Apple Silicon chips, data privacy, and microservice maintainability. By unifying RAG capabilities, embedded local LLM inference, and robust containerised deployment practices, the proposed microservice provides a highly

performant, secure, and efficient AI-driven solution suitable for deployment in sensitive, privacy-focused contexts.

Figure 2 provides a detailed representation of the microservice's internal architecture and control flow, highlighting the interactions between its RESTful API endpoints, dedicated controllers, and core backend components. Users directly engage with the system through defined HTTP endpoints. These include endpoints for chat interactions (*POST /chat*), retrieving stored data chunks (*GET /chunks*), clearing existing embeddings (*DELETE /clear_embeddings*), and uploading new files (*POST /upload_file*). Each endpoint is designed to handle specific user operations securely and efficiently.

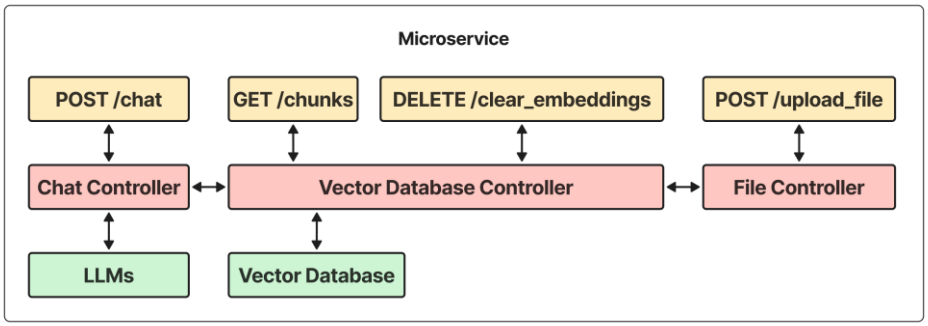

Figure 2. Microservice internal architecture and control flow

Internally, the microservice organises its logic into controllers to maintain a clear separation of concerns. The Chat Controller processes user interactions, receives chat requests, coordinates with the Vector Database Controller to retrieve contextual embeddings, and forwards the enriched queries to the embedded local LLM for inference.

The Vector Database Controller manages all interactions related to embeddings stored within the local ChromaDB database. This controller is a central point for embedding operations, facilitating efficient retrieval and management of embeddings for various system components. It supports operations invoked directly through API endpoints and indirectly via requests from other controllers, such as the Chat and File Controllers.

Managing file-related tasks, the File Controller specifically handles user-uploaded documents. Upon receiving files through the *POST /upload_file* endpoint, this controller processes and transforms uploaded content into vector embeddings suitable for storage. It communicates directly with the Vector Database Controller to ensure these embeddings securely persist, are properly indexed, and are ready for future retrieval operations.

Figure 3 illustrates the overall processing flow of the *POST /chat* request within the microservice. It outlines the sequential steps from receiving a user prompt to generating and returning a context-aware response, highlighting interactions with the local LLM and ChromaDB components.

When a request is received, it first generates a set of five relevant keywords by prompting the local LLM with the following instruction: *"Return only 5 keywords as separate words, use a comma as a separator. Relevant to this text (for RAG): {message}"*. These keywords are then used to search within ChromaDB, which returns the top three most relevant document chunks based on L2 distance similarity. Once the relevant context is retrieved, the Chat Controller issues a second call to the LLM using a structured prompt: *"Context:\n{context}\n\nUser Query:\n{message}\n\nAnswer the query based on the*

context above." The LLM processes the prompt and context together and generates a context-aware response. This final answer is then returned to the user. All these operations, including keyword extraction, vector search, and LLM inference, are executed serially for each request without parallelisation. While this ensures clarity and simplicity in processing, it also contributes to the overall response latency, particularly under heavier workloads.

A few words on why keyword generation by LLM was chosen for this approach. Many approaches exist and can be applied depending on the tasks. Early methods for automatic keyword or keyphrase extraction relied on statistical and graph-based algorithms. These classical approaches are simple and domain-independent, but they often miss important context-specific phrases, resulting in providing inaccurate or irrelevant responses (Aftab et al., 2024).

Recent years have seen a shift to deep learning and transformer-based methods, which better capture contextual and semantic information. Supervised models treat keyword and keyphrase extraction as a sequence labelling or sequence-to-sequence task. Likewise, various transformer-based models leverage large pre-trained language models as encoders, dramatically boosting extraction accuracy by using context from the entire document. There are also neural unsupervised approaches, such as embedding-based methods, which rank phrases by their semantic similarity to the document. Overall, deep learning approaches that incorporate contextualized embeddings have been shown to outperform most classical algorithmic and graph methods, addressing context-dependent limitations (Giarelis and Karacapilidis, 2024).

Researchers have also tailored keyword/keyphrase extraction to specific domains. For example, in legal text, where vocabulary and phraseology are very domain-specific, custom pipelines have been proposed to improve keyphrase identification (Santosh and Hernandez, 2025). These domain-specific efforts demonstrate that while general algorithms provide a foundation, incorporating domain knowledge or constraints can further enhance the relevance of extracted keywords. Based on the existing research results, in this experiment, LLM was used to generate keywords and use them for the RAG approach.

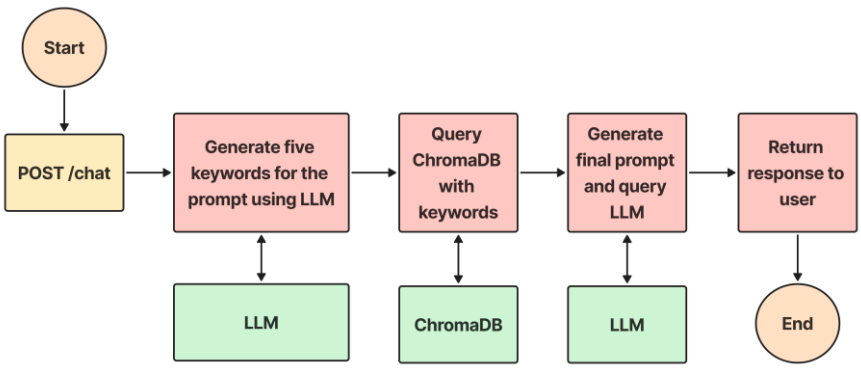

Figure 3. Chat execution flow

This modular structure enhances maintainability and promotes scalability. It is also important to note that Docker cannot currently execute the MLX-based LLM on Apple Silicon hardware, despite the service being designed for containerised deployment. This

limitation stems from the lack of full support for Apple's MLX framework within Docker's virtualisation layer on macOS, particularly for accessing GPU and Neural Engine acceleration. As a result, while the codebase and system architecture are built to be container-ready and follow microservice best practices, the actual execution of the MLX model must take place natively on the host system. This constraint highlights a current gap in tooling compatibility. It reinforces the importance of local optimisation and portability in the service design, ensuring it remains efficient and functional outside containerised environments.

The microservice lifecycle and monitoring are out of scope for this study. Therefore, processes of updating, upgrading, and patching were not designed and evaluated in this research paper. Nevertheless, the internal system's modular architecture was chosen to allow components to be replaced or extended with minimal coupling. Clear boundaries between retrieval, embedding, inference, storage, and API layers are defined. A systematic investigation of lifecycle management (automated updates, schema and version migrations, compatibility testing, and operational rollout strategies) was left to future work and subsequent papers.

Overall, monitoring may be implemented by instrumenting the microservice to expose optional "/health" and "/metrics" endpoints. These endpoints may provide information in a generic format, covering a small set of signals such as request counts, latency percentiles (p50, p95, p99), tokens per second, vector-store latency, cache hit ratio, error rate, memory, and queue depth. Logs could be anonymized, sensitive fields omitted, or hashed and written as structured JSON. A lightweight, local stack might then scrape and visualize this data. Simple threshold alerts could be set for spikes in p95 latency or error rate, slow vector queries, or low memory and disk. To keep overhead low and privacy high, all components may run on the same Mac with default redaction and time/size-based log rotation. A small maintenance CLI could provide status, reindex, backup/restore, and config checks, with brief runbooks documenting responses to common issues.

Selecting an appropriate LLM model for a local microservice involves balancing several key factors, including model size, task requirements, hardware constraints, and deployment goals. Smaller models, such as those in the 1–3 billion parameter range, are often preferred for on-device execution due to their lower memory footprint and faster inference times, critical for resource-constrained environments like laptops or edge devices. The choice also depends on the nature of the task, whether it is classification, summarisation, question answering, or open-ended generation, and whether domain-specific fine-tuning is required. Compatibility with the underlying framework is another essential consideration. Licensing and community support may influence the selection, as open-source models offer flexibility for research and commercial use, with pre-trained weights readily accessible and optimised tools for local inference.

LLaMA 3.2 1B is a compact version of Meta's LLaMA family, designed to deliver strong language modelling capabilities in a significantly reduced parameter footprint (Grattafiori et al., 2024). With approximately 1 billion parameters, this model is part of the LLaMA 3.2 release, which incorporates architectural improvements over previous iterations, including better tokenisation, optimised training efficiency, and enhanced instruction-following behaviour.

Despite its smaller size, LLaMA 3.2 1B performs strongly on a range of foundational NLP tasks. It is especially well-suited for edge deployments and resource-constrained environments with limited memory and computing. Its design allows for faster inference and lower power consumption while preserving contextual understanding and fluency in

generation. The 1B variant is particularly advantageous for local execution on devices powered by Apple Silicon, as it aligns well with the M chips' memory constraints and computational throughput. When used with the MLX and MLX-LM toolchain, it can be quantised and executed efficiently using CPU, GPU, or the Neural Engine, balancing performance, responsiveness, and privacy. This makes LLaMA 3.2 1B a practical choice for embedding LLMs into lightweight, secure microservice architectures.

5. Testing and Evaluation Methodology

The article provides a limited, research-oriented test set. It contains end-to-end tests of the full pipeline to verify functionality and performance benchmarks measuring latency, throughput, and memory under typical workloads. It does not include a full functional test suite derived from formal requirements. These are outside the current scope and may be conducted in the future.

The testing approach is structured into two parts. The first part evaluates the performance of parsing PDF files, chunking and inserting their content into the ChromaDB, and simulating the ingestion of external knowledge into the system. This includes measuring insertion times, file sizes, number of chunks, chunk size, and the success rate of the document upload process. The second part tests the chat functionality, where the embedded LLM generates responses based on previously uploaded and indexed data. This test assesses the execution times, input and output tokens, peak memory, and responsiveness of the LLM in generating context-aware answers using the stored embeddings. However, this evaluation has important limitations. The accuracy or factual correctness of the LLM responses was not explicitly tested or validated by automated tests.

The dataset used for ingestion was limited to 100 PDF files. The PDF files used in the evaluation were sourced from open and publicly accessible scientific repositories, including platforms such as arXiv, ScienceDirect, and IEEE Xplore. The smallest file in the collection was approximately 0.07 MB, while the largest file reached 13.03 MB. This variation ensured the system was evaluated across lightweight and heavy content scenarios. Regarding content density, the minimum word count among the files was 0, typically indicating either an empty document or one composed solely of non-extractable images or metadata. On the other end of the spectrum, the maximum word count was 63,907 words.

An automated testing approach was implemented using several Python scripts to evaluate the performance of the data insertion. This approach was designed to simulate realistic usage conditions by uploading multiple PDF files. The test begins with a *DELETE /clear_embeddings* endpoint request to clear the database and remove all existing embeddings.

The test loop processes each *.pdf* file located in a folder. For every file, the script calculates its size in megabytes and initiates a *POST* request to upload it via the */upload_file* endpoint. The total execution time for each upload is recorded using timestamps before and after the API call. The system's response code determines whether the upload was successful or failed, and the results are logged accordingly.

To analyse the relationship between file size and upload time, the script generates a scatter plot with file size on the x-axis and upload time on the y-axis. To evaluate the system's ingestion performance, another scatter plot presents the relationship between the length of extracted text from PDF documents (in characters) and the corresponding insertion time into ChromaDB. Additionally, all performance data are saved to CSV files.

This methodology provided quantitative and visual evidence of system performance, reliability, and resource behaviour under actual usage scenarios.

The second part of the evaluation focuses on testing the chat functionality, where the embedded LLM generates responses based on previously uploaded and indexed knowledge in the vector database. This phase assesses the system's responsiveness and performance by measuring key metrics such as execution time, the number of input and output tokens, and the system's ability to handle varied prompts.

The prompt set consists of 300 unique queries, each automatically generated by an LLM. These prompts vary in complexity, length, and domain (e.g., *"What is the purpose of a hash function in computer science?" "Describe the differences between relational and non-relational databases."*). During each iteration, one unique prompt is selected for each previously uploaded PDF file. For each interaction, the script records execution time, token usage, and the success or failure of the response. These metrics are logged into the CSV file and visualised.

These prompts collectively contained a total of 2,850 words. Analysis of the vocabulary reveals that the most frequent terms included common English words as well as domain-relevant terms such as *"What," "Explain," "concept," and "Describe,"* reflecting a focus on informational and explanatory queries. The average prompt length was 59 characters, with an average word count of 10 words per prompt. The shortest prompt contained 6 words, while the longest included 14. This distribution highlights a consistent and concise structure across the dataset, suitable for benchmarking LLM responsiveness and performance in a controlled manner.

Figure 4 presents two histograms summarising the sentence length distribution in the testing prompt dataset. Subfigure (a) illustrates the relationship between word count and the number of sentences. Most prompts fall within the 9 to 10-word range, with 9-word prompts being the most frequent. Subfigure (b) displays the distribution of character counts per sentence, showing that most prompts are between 45 and 70 characters long. These visualisations confirm that the dataset consists of short, concise prompts with a consistent structure, making it suitable for evaluating the performance and responsiveness of the RAG-based system.

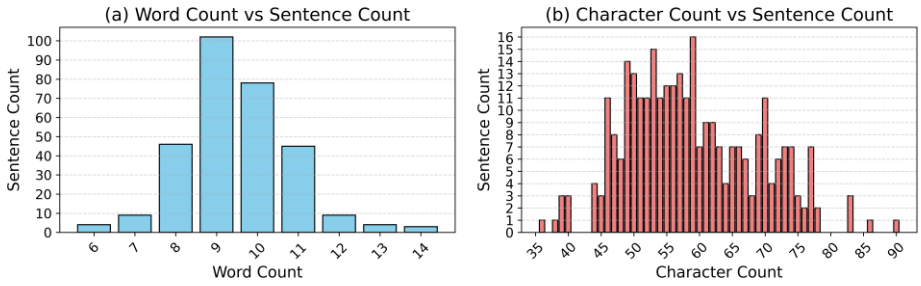

Figure 4. Distribution of sentence lengths in the prompt dataset

The requests library sends each prompt as a JSON payload to the *POST /chat* endpoint. Execution time is measured by recording timestamps before and after the API call. Upon receiving a response, the script checks the HTTP status code and a response.

The service writes its runtime output to a log file, which is later parsed to extract performance metrics. This design choice is necessary because MLX-LM, the framework used to execute the local language model, outputs key performance data – such as memory

usage, input and output token counts, and processing speed – directly to the console during inference. The standard output is redirected to a log file to capture this information programmatically.

Following the logging and parsing process, the extracted performance metrics are converted into a structured CSV file and then visualised using Python's Matplotlib library.

6. Results and Evaluation

This section presents and discusses results from testing the implemented privacy-preserving microservice based on the proposed software architecture and system design.

Testing was conducted on a laptop running Apple Silicon with 16 GB of RAM, but with multiple applications open concurrently. This setup reflects a realistic usage scenario but may have introduced performance variability due to system resource contention. As such, while the tests effectively demonstrate the microservice's responsiveness and operational integrity, they do not provide a complete picture of the accuracy of LLM responses or performance under dedicated or optimised hardware conditions.

Figure 5 illustrates the relationship between PDF file size (in megabytes) and insertion time (in seconds) into the ChromaDB vector database with L2 distance for the similarity metric. Each data point represents a single PDF file processed by the system, which was successfully inserted. The chart reveals that smaller files (under 2 MB) generally exhibit shorter and more consistent insertion times, typically below 100 seconds. However, as file size increases, insertion time becomes more variable, with some larger files taking over 200 seconds to process. A few outliers, including one exceeding 300 seconds, suggest that factors beyond file size – such as document structure, encoding, or number of extractable tokens – may also impact processing time. Importantly, all insertions were successful, indicating high reliability across various input sizes.

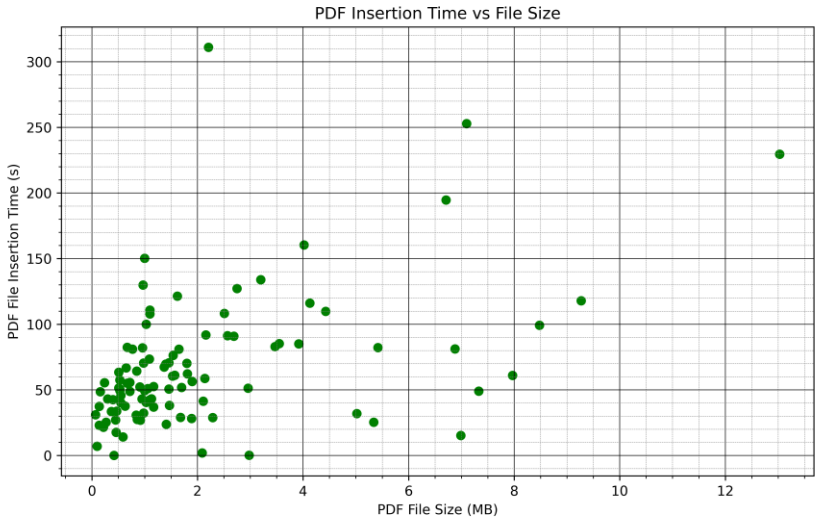

Figure 5. Insertion time vs PDF file size comparison

Figure 6 shows the relationship between the length of extracted PDF text (measured in number of characters) and the total insertion time (in seconds) into the ChromaDB. Each data point represents a single document processed and inserted by the system. As illustrated, there is a clear upward trend – longer text documents require more time to be parsed, chunked, embedded, and stored. It is important to note that the text extraction process excluded all images, tables, and other graphical elements. Only raw textual content was parsed, meaning the insertion time reflects text-based processing.

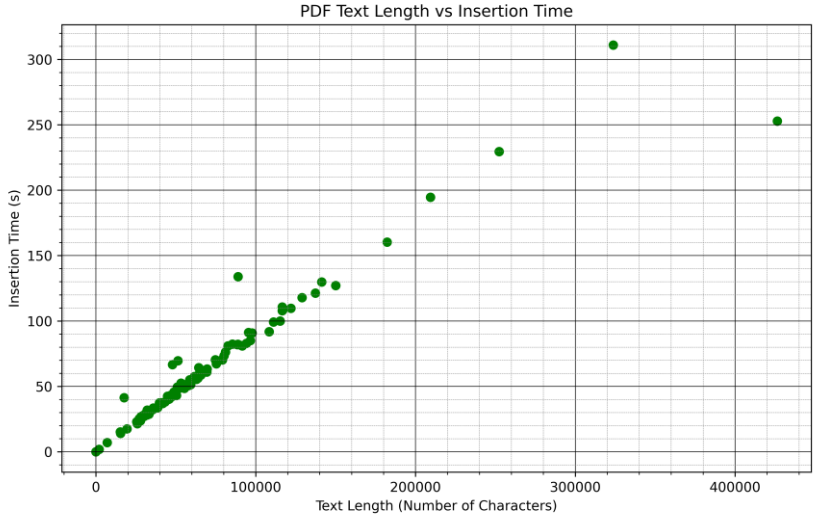

Figure 6. Text length vs insertion time comparison

The distribution indicates a correlation for small to medium-length documents, where insertion time increases consistently with text size. However, for very large texts (exceeding 200,000 characters), the insertion time shows higher variability, with some documents taking over 300 seconds. This behaviour reflects the computational cost of chunking large files and generating embeddings sequentially for each segment.

This result highlights a performance bottleneck in the current implementation of vector database, especially when dealing with large academic documents. All insertions in this test were performed serially, meaning each PDF document was processed and uploaded one at a time. This sequential approach was intentionally chosen to simulate a realistic and minimal setup on a local machine. This means the system processes one file simultaneously before moving on to the following document. While this approach simplifies testing and avoids concurrency-related complexity, it does not leverage the full potential of modern multicore processors or asynchronous processing frameworks.

ChromaDB performance can appear slow during large document insertions due to several contributing factors inherent to its architecture and the nature of embedding-based indexing. First, ChromaDB performs real-time vectorisation of text chunks, which involves generating embeddings for each segment using a preloaded language model or external embedding service. This process is computationally intensive, especially for large PDF files with many tokens or complex formatting.

Additionally, text extraction and chunking from PDF files are often bottlenecked by I/O operations and CPU-bound parsing tasks. The preprocessing pipeline becomes significantly slower if documents contain embedded images, tables, or non-standard encodings. ChromaDB's in-memory or disk-backed storage layer may also introduce latency, particularly when rapidly handling write-heavy operations or persisting data across multiple chunks.

In local deployments, especially on general-purpose laptops, these delays are further exacerbated by limited concurrency and shared resource contention – such as when multiple background applications consume CPU, memory, or I/O bandwidth. The observed insertion slowness results from compound effects from embedding generation, document preprocessing, and system-level resource constraints typical of single-node deployments.

Figure 7 illustrates four key relationships between token counts and system performance metrics during local LLM inference. Each subplot provides a specific perspective on how input and output token quantities affect execution time and memory usage, offering a comprehensive view of runtime behaviour.

In the first subplot (a), titled *Input Tokens vs Execution Time*, we observe that even though the input token range is relatively narrow (between 67 and 75 tokens), execution times vary substantially, from around 35 seconds to 150 seconds. This inconsistency suggests that factors beyond input size, such as system conditions, prompt complexity, or output generation behaviour, may influence execution duration.

The second subplot (b), *Input Tokens vs Peak Memory*, shows that peak memory usage remains relatively constant at around 1.6 GB regardless of input size. The on-disk size of the quantised model used with MLX-LM is approximately 1.32 GB. This indicates that the underlying LLM and runtime environment manage memory usage efficiently and are not significantly affected by small variations in input token length.

A few outliers are visible below the 1.5 GB line, which may correspond to cases where the model response was minimal or the prompt triggered significantly shorter processing paths. Similarly, one or two higher-token inputs remain within the standard memory range, highlighting that token count alone does not necessarily drive peak memory usage beyond typical operational limits.

In subplot (c), *Output Tokens vs Execution Time*, a more pronounced pattern emerges. Execution time increases with the number of output tokens, confirming that generating longer responses incurs a higher computational cost. This is consistent with expectations for autoregressive models, where each additional token requires forward-pass computation through the model.

Finally, subplot (d), *Output Tokens vs Peak Memory*, demonstrates that memory usage remains consistent across a wide range of output sizes, up to over 700 tokens. This further reinforces the observation that while execution time scales with output length, memory consumption does not, highlighting the model's efficient handling of token generation in constrained environments.

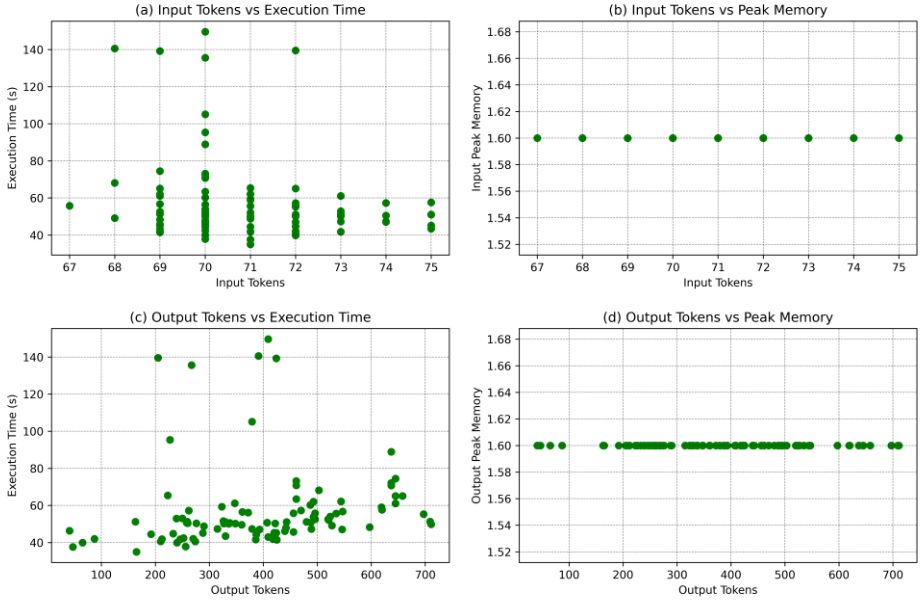

Figure 7. Performance metrics of LLM execution based on token count

Overall, the figure provides valuable insight into the performance characteristics of local LLM inference. It indicates that execution time is primarily influenced by output length, whereas memory usage is stable across both input and output token ranges.

The figure suggests that execution time scales with output length while memory usage remains flat and efficient, regardless of input or output size. These characteristics position the LLaMA 3.2 1B model with MLX as a strong candidate for real-time local deployment, especially in applications where consistent memory performance is critical, even under varying workloads.

Figure 8 displays the relationship between the number of input tokens and the number of output tokens generated by the LLM during testing. Each data point represents an interaction between the user and the microservice, with input lengths varying widely across the dataset. Despite this narrow input range, the model outputs vary significantly, from as few as ~50 to over 700 tokens.

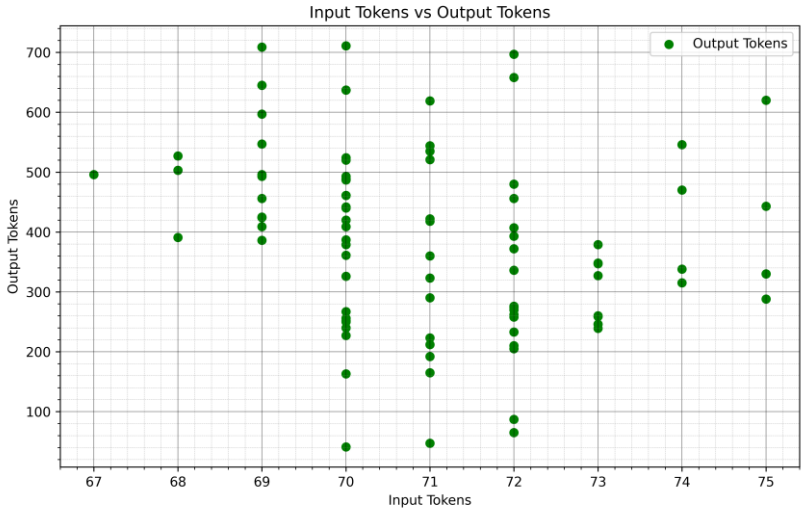

Figure 8. Input tokens vs output tokens

This wide dispersion of output lengths suggests that while the input size remains relatively constant, the content or semantics of the prompt heavily influence the length of the generated response. Some prompts likely triggered more elaborate explanations or step-by-step breakdowns, while others elicited more concise replies. Importantly, this variation is typical for LLMs, especially when the model operates with a relatively unconstrained decoding strategy, such as a high temperature or no strict output length cap.

The figure confirms that output size is not linearly determined by input size, highlighting the generative and interpretative flexibility of the underlying LLM. From an application standpoint, this reinforces the need to account for potentially significant variances in output length when planning for performance and memory requirements in real-time, local deployments.

Overall, this figure suggests that the LLM used in this microservice delivers memory-efficient performance, even with varying input lengths. This characteristic is particularly valuable for edge and local deployment scenarios where system memory may be limited, such as macOS devices with Apple Silicon.

The testing and evaluation of the proposed secure RAG microservice demonstrated that it can effectively support privacy-preserving document ingestion and LLM-based querying. Results from the document ingestion tests showed that insertion times generally scaled with file size. However, high variability was observed due to the complexity of PDF content and the serial processing approach. ChromaDB's performance was influenced by preprocessing, tokenisation, and real-time embedding generation. Memory use during these operations remained within acceptable local limits, supporting the system's suitability for on-device deployments.

In the LLM response evaluation, input prompts varied in length, and the output token distribution reflected a wide range of responses, from concise answers to extended explanations. Response generation times and memory consumption remained stable across different input sizes. The average peak memory during inference was approximately 1.6 GB and token throughput was consistent at most prompt lengths. This confirms that MLX-

LM's integration with LLaMA 3.2 1B delivers efficient and reproducible performance on MacOS.

Overall, the results affirm the feasibility of running private RAG services entirely locally, with predictable resource usage and sufficient capacity to support real-world applications in secure or disconnected environments.

7. Conclusion

This study introduced a comprehensive, privacy-preserving microservice architecture for local deployment on Apple Silicon hardware, incorporating embedded LLM execution and RAG. Integrating these features into a single, unified microservice significantly advances the practicality and security of deploying AI-driven services locally, especially in sensitive domains requiring stringent data protection.

Evaluations demonstrated that the implemented system can effectively handle typical and peak workloads with acceptable latency. Utilising Apple's MLX framework provided noticeable performance and power-efficiency advantages. The RAG component provides better grounding and context to the LLM model. The model can provide context-relevant responses that may contain private data based on the additional context.

Resource utilisation analyses underscored the suitability of this architecture for constrained environments, showing moderate CPU and memory demands even during intensive use. The system eliminated risks associated with external data exposure and cloud-based deployments by isolating model inference and data storage within encrypted local disks.

In conclusion, this work demonstrates the viability of a locally deployable, privacy-centric microservice architecture that aligns closely with modern software development principles of modularity, maintainability, and security. The presented results provide a practical foundation for future studies and industrial applications, suggesting clear pathways for further optimisations in model fine-tuning, retrieval accuracy enhancement, and advanced security mechanisms.

Acknowledgements

The authors are grateful for the support from the Education and Science of Ukraine (Project No 0125U001883).

References

Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., Clarke, P. (2023). Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*, *49*(8), 4213–4242. https://doi.org/10.1109/TSE.2023.3287297

Aftab, W., Apostolou, Z., Bouazoune, K., Straub, T. (2024). Optimizing biomedical information retrieval with a keyword frequency-driven prompt enhancement strategy. *BMC Bioinformatics*, *25*(1), 281. https://doi.org/10.1186/s12859-024-05902-7

Aguirre, J., Cha, W. C. (2025). JAVIS Chat: A Seamless Open-Source Multi-LLM/VLM Deployment System to Be Utilized in Single Computers and Hospital-Wide Systems with Real-Time User Feedback. *Applied Sciences*, *15*(4), Article 4. https://doi.org/10.3390/app15041796

Astral-sh/uv. (2025, April 17). Astral-Sh/Uv. https://github.com/astral-sh/uv (Original work published 2023)

Bucher, M. J. J., Martini, M. (2024). *Fine-Tuned "Small" LLMs (Still) Significantly Outperform Zero-Shot Generative AI Models in Text Classification* (No. arXiv:2406.08660). arXiv. https://doi.org/10.48550/arXiv.2406.08660

Buesser, B. (2024). Private Information Leakage in LLMs. In A. Kucharavy, O. Plancherel, V. Mulder, A. Mermoud, V. Lenders (Eds.), *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation* (pp. 75–79). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-54827-7_7

Chaplia, O., Klym, H., Elsts, E. (2024). Serverless AI Agents in the Cloud. *Advances in Cyber-Physical Systems*, *9*(2), 115–120. https://doi.org/10.23939/acps2024.02.115

Cheng, Y., Zhang, L., Wang, J., Yuan, M., Yao, Y. (2024). *RemoteRAG: A Privacy-Preserving LLM Cloud RAG Service* (No. arXiv:2412.12775). arXiv. https://doi.org/10.48550/arXiv.2412.12775

Chrapek, M., Vahldiek-Oberwagner, A., Spoczynski, M., Constable, S., Vij, M., Hoefler, T. (2024). *Fortify Your Foundations: Practical Privacy and Security for Foundation Model Deployments In The Cloud* (No. arXiv:2410.05930). arXiv. https://doi.org/10.48550/arXiv.2410.05930

ChromaDB. (n.d.). ChromaDB. Retrieved April 17, 2025, from https://trychroma.com

ChromaDB Docs. (n.d.). ChromaDB Docs. Retrieved April 17, 2025, from https://docs.trychroma.com

Feng, D., Xu, Z., Wang, R., Lin, F. X. (2025). *Profiling Apple Silicon Performance for ML Training* (No. arXiv:2501.14925). arXiv. https://doi.org/10.48550/arXiv.2501.14925

Giarelis, N., Karacapilidis, N. (2024). Deep learning and embeddings-based approaches for keyphrase extraction: A literature review. *Knowledge and Information Systems*, *66*(11), 6493–6526. https://doi.org/10.1007/s10115-024-02164-w

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., … Ma, Z. (2024). *The Llama 3 Herd of Models* (No. arXiv:2407.21783). arXiv. https://doi.org/10.48550/arXiv.2407.21783

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Giorno, A. D., Gopi, S., Javaheripi, M., Kauffmann, P., Rosa, G. de, Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., Li, Y. (2023). *Textbooks Are All You Need* (No. arXiv:2306.11644). arXiv. https://doi.org/10.48550/arXiv.2306.11644

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., Zhang, X. (2024). *Large Language Model based Multi-Agents: A Survey of Progress and Challenges* (Version 2). arXiv. https://doi.org/10.48550/ARXIV.2402.01680

Händler, T. (2023). *Balancing Autonomy and Alignment: A Multi-Dimensional Taxonomy for Autonomous LLM-powered Multi-Agent Architectures* (No. arXiv:2310.03659). arXiv. https://doi.org/10.48550/arXiv.2310.03659

Hasan, M. H., Osman, M. H., Admodisastro, N. I., Muhammad, M. S. (2023). From Monolith to Microservice: Measuring Architecture Maintainability. *International Journal of Advanced Computer Science and Applications*, *14*(5). https://doi.org/10.14569/IJACSA.2023.0140591

Hossain, Md. D., Sultana, T., Akhter, S., Hossain, M. I., Thu, N. T., Huynh, L. N. T., Lee, G.-W., Huh, E.-N. (2023). The role of microservice approach in edge computing: Opportunities, challenges, and research directions. *ICT Express*, *9*(6), 1162–1182. https://doi.org/10.1016/j.icte.2023.06.006

Ieva, S., Loconte, D., Loseto, G., Ruta, M., Scioscia, F., Marche, D., Notarnicola, M. (2024). A Retrieval-Augmented Generation Approach for Data-Driven Energy Infrastructure Digital Twins. *Smart Cities*, *7*(6), Article 6. https://doi.org/10.3390/smartcities7060121

Jin, X., Lin, Z. (2024). SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach. *SimLLM: Calculating Semantic Similarity in Code Summaries Using a Large Language Model-Based Approach*, *1*(FSE), 62:1376-62:1399. https://doi.org/10.1145/3660769

Jones, J., Jiang, W., Synovic, N., Thiruvathukal, G., Davis, J. (2024). What do we know about Hugging Face? A systematic literature review and quantitative validation of qualitative claims. *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 13–24. https://doi.org/10.1145/3674805.3686665

Josa, A. D., Bleda-Bejar, M. (2024). Local LLMs: Safeguarding Data Privacy in the Age of Generative AI. A Case Study at the University of Andorra. *ICERI2024 Proceedings*, 7879–7888. 17th annual International Conference of Education, Research and Innovation. https://doi.org/10.21125/iceri.2024.1931

Kasperek, D., Antonowicz, P., Baranowski, M., Sokolowska, M., Podpora, M. (2023). Comparison of the Usability of Apple M2 and M1 Processors for Various Machine Learning Tasks. *Sensors*, *23*(12), Article 12. https://doi.org/10.3390/s23125424

Kenyon, C., Capano, C. (2022). Apple Silicon Performance in Scientific Computing. *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–10. https://doi.org/10.1109/HPEC55821.2022.9926315

Liu, F., Kang, Z., Han, X. (2024). *Optimizing RAG Techniques for Automotive Industry PDF Chatbots: A Case Study with Locally Deployed Ollama Models* (No. arXiv:2408.05933). arXiv. https://doi.org/10.48550/arXiv.2408.05933

Misnevs, B. (2025). Conceptual Model: Personal Information Management Using Adaptive Information Systems. *Baltic Journal of Modern Computing*, *13*(1). https://doi.org/10.22364/bjmc.2025.13.1.07

Ml-explore/mlx-lm. (2025). [Python]. ml-explore. https://github.com/ml-explore/mlx-lm (Original work published 2025)

Santosh, T.Y.S.S., Hernandez, E. Q. (2025). LexKeyPlan: Planning with Keyphrases and Retrieval Augmentation for Legal Text Generation: A Case Study on European Court of Human Rights Cases. In W. Che, J. Nabende, E. Shutova, & M. T. Pilehvar (Eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (pp. 425–436). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.acl-short.32

Received April 23, 2025, revised August 21, 2025, accepted September 25, 2025