IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

Automated Health Insurance Claim Processing Using Flask Microservices, Vector-Augmented Language Models

Sameer Singh[1], Vandana Swami[2], Rajendra Singh[3]
[1] Department of Computer Science and Engineering
[2] Department of Computer Science and Engineering
[3] Dean, Department of Computer Science and Engineering
Raffles University, Neemrana, Rajasthan, India
sameersingh09504@gmail.com, vandana.swami@rafflesuniversity.edu.in
rajendra.singh@rafflesuniversity.edu.in

Abstract: *The deployment of artificial intelligence systems in production cloud environments presents significant engineering challenges including memory management, latency optimization, and infrastructure cost minimization. Health insurance claim processing systems that incorporate large language models, vector databases, and optical character recognition pipelines are particularly demanding in terms of computational resources, making cost-effective cloud deployment a critical research problem. This paper presents the design, implementation, and performance evaluation of a scalable cloud-based automated health insurance claim processing system built on Flask microservices architecture and Docker containerization, deployed on Hugging Face Spaces at zero infrastructure cost. The system integrates a FAISS-powered vector retrieval pipeline for insurance policy handbook search, EasyOCR for medical bill digitization, and the Groq LLaMA 3.3 70B large language model for claim evaluation and executive report generation. A key architectural contribution is a three-stage claim validation pipeline that minimizes expensive language model API invocations by resolving obvious rejection cases at lightweight validation layers before escalating to language model inference. Performance benchmarking demonstrates median end-to-end processing latency of 3.2 seconds for claims requiring full AI evaluation and sub-10-millisecond processing for claims resolved at validation layers one and two. The deployed system sustains concurrent request handling without resource exhaustion on a free-tier instance providing 16 gigabytes of RAM. Comparative analysis against alternative deployment platforms confirms that Hugging Face Spaces provides the optimal balance of available memory, deployment simplicity, and zero operational cost for AI-intensive applications of this class..*

Keywords: Cloud Deployment, Flask Microservices, Docker, FAISS, Vector Database, LLaMA, Health Insurance, Scalability, Performance Optimization, Hugging Face

I. INTRODUCTION

The intersection of artificial intelligence and cloud computing has created unprecedented opportunities for deploying intelligent automation systems in domains previously dependent on manual human expertise. Health insurance claim processing represents one such domain, where the manual evaluation of claims against policy handbooks is both time-intensive and operationally expensive. The deployment of AI systems capable of replacing or augmenting this manual process requires careful consideration of cloud infrastructure choices, containerization strategies, memory management for large AI models, and latency optimization techniques.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

The challenge of deploying AI-intensive applications in production cloud environments is well documented. Large language models with tens of billions of parameters require gigabytes of memory for loading and inference. Vector databases such as FAISS require pre-computation and persistent storage of embedding indices. Optical character recognition pipelines incorporating deep neural networks add further memory pressure. The combination of these components in a single application creates a demanding deployment scenario that many free-tier cloud platforms cannot accommodate due to memory constraints.

This paper presents a complete engineering solution to this deployment challenge, demonstrating that a production-quality AI claim processing system incorporating all these components can be deployed at zero infrastructure cost by selecting an appropriate cloud platform and optimizing the application architecture for memory efficiency. The system processes insurance claims submitted through a web interface, applies a three-stage validation pipeline combining rule-based checks, semantic exclusion matching, and language model reasoning, and generates structured compliance reports in under 5 seconds.

The primary contributions of this paper are as follows. First, a detailed analysis of cloud platform alternatives for memory-intensive AI application deployment is presented, demonstrating that Hugging Face Spaces with 16 gigabytes of RAM provides significant advantages over alternative free-tier platforms limited to 512 megabytes. Second, a lazy initialization architecture for large AI model components is proposed and evaluated, demonstrating significant reduction in application startup time and memory overhead. Third, a Docker containerization strategy optimized for Python AI applications with large dependency trees is presented. Fourth, comprehensive performance benchmarking results characterize system behavior under realistic load conditions.

The remainder of this paper is organized as follows. Section II reviews related work in cloud deployment of AI applications. Section III describes the system architecture with emphasis on deployment and performance design decisions. Section IV presents implementation details. Section V reports performance benchmarking results. Section VI concludes with future work directions.

II. RELATED WORK

A. Cloud Deployment of AI Applications

The deployment of machine learning and deep learning applications in cloud environments has been an active research area since the widespread adoption of cloud computing infrastructure in the early 2010s. Moreno-Vozmediano et al. surveyed cloud deployment strategies for scientific computing workloads and identified memory availability as the primary bottleneck for data-intensive applications [1]. Their findings motivate the platform selection analysis presented in Section III of this paper.

Container-based deployment using Docker has become the dominant paradigm for AI application deployment due to its reproducibility guarantees and environment isolation. Bernstein reviewed the evolution of container technologies and documented their advantages over virtual machine-based deployment for applications requiring rapid scaling and consistent runtime environments [2]. Docker's layer-based image construction is particularly beneficial for AI applications with large dependency trees, as unchanged layers are cached between builds, dramatically reducing rebuild times.

Serverless deployment frameworks have been proposed as an alternative to container-based deployment for reducing operational overhead. However, Hellerstein et al. identified significant limitations of serverless platforms for stateful AI applications including cold start latency of several seconds for large model loading and memory constraints that preclude the use of large neural network models [3]. These limitations make serverless architectures unsuitable for the AI model combination used in this system.

B. Memory Optimization for Large Language Models

The memory requirements of large language models present significant challenges for cost-constrained deployment scenarios. Frantar et al. proposed GPTQ, a post-training quantization technique that reduces model memory

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

requirements by 4 to 8 times with minimal accuracy degradation [4]. However, quantization typically requires specialized inference infrastructure not available through standard API services.

API-based inference, where the language model runs on provider infrastructure and the application communicates through a network API, transfers the memory burden to the provider while introducing network latency. Groq's Language Processing Unit architecture provides particularly low-latency API inference due to its deterministic execution model, making it well-suited for user-facing applications with response time requirements [5]. This motivates the use of Groq API for language model inference in the proposed system rather than local model loading.

C. Vector Database Deployment

FAISS, developed by Johnson et al. at Facebook AI Research, provides efficient similarity search over large collections of dense vectors [6]. For moderate-scale deployments with index sizes in the range of thousands to tens of thousands of vectors, FAISS IndexFlatL2 provides exact nearest neighbor search with sub-millisecond query times on CPU hardware without requiring GPU acceleration. This makes FAISS an appropriate choice for deployments on CPU-only cloud instances.

Alternative vector database solutions including Pinecone, Weaviate, and Chroma provide managed cloud services that eliminate the need to maintain vector indices within the application process. However, these services introduce additional API dependencies, network latency, and potential cost at scale. For the insurance handbook knowledge base used in this system, the relatively small index size makes in-process FAISS storage preferable to external vector database services.

D. Flask Application Architecture

Flask is a widely used Python web framework for building REST APIs and web applications. Grinberg's comprehensive analysis of Flask application patterns identified several architectural strategies relevant to AI-serving applications, including lazy initialization of expensive resources and connection pooling for database access [7]. The application-level factory pattern, where expensive AI model components are initialized once at application startup and shared across request handlers through global variables, is particularly relevant to the system described in this paper.

III. SYSTEM ARCHITECTURE AND DEPLOYMENT DESIGN

A. Overview

The system architecture is designed around three primary engineering objectives: minimizing application startup time through lazy component initialization, minimizing per-request latency through efficient pipeline design, and minimizing infrastructure cost through appropriate cloud platform selection. Figure 1 illustrates the overall system architecture with emphasis on the deployment stack.

The application consists of a single Flask process that serves both the web interface and the claim processing API. This monolithic process architecture was chosen over a microservices decomposition because the individual components — OCR, vector retrieval, and language model inference — are tightly coupled in the processing pipeline, and the overhead of inter-process communication would increase latency without providing meaningful scalability benefits at the expected request volumes.

B. Cloud Platform Selection

A systematic evaluation of candidate free-tier cloud platforms was conducted to identify the optimal deployment target for the AI component combination used in this system. Table I summarizes the evaluation results.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

TABLE I: CLOUD PLATFORM COMPARISON FOR AI APPLICATION DEPLOYMENT

Platform	Free RAM	EasyOCR	Sentence Transformers	FAISS	Docker Support	Cost
Render.com	512 MB	No — crashes	Yes	Yes	Yes	Free
Railway.app	512 MB	No — crashes	Yes	Yes	Yes	Free
Koyeb.com	512 MB	No — crashes	Yes	Yes	Yes	Free
Heroku	512 MB	No — crashes	Yes	Yes	Yes	Free
Hugging Face Spaces	16,384 MB	Yes	Yes	Yes	Yes	Free
AWS EC2 t2.micro	1,024 MB	Marginal	Yes	Yes	Yes	Free tier limited

EasyOCR requires approximately 1.5 gigabytes of RAM to load its deep learning models for text detection and recognition. Sentence Transformers requires approximately 400 megabytes. The Flask application and FAISS index require approximately 200 megabytes. The total memory requirement of approximately 2.1 gigabytes exceeds the 512-megabyte limit of Render, Railway, Koyeb, and Heroku free tiers, which results in out-of-memory termination during EasyOCR model initialization. AWS EC2 t2.micro with 1 gigabyte of RAM is marginally insufficient and results in system instability under concurrent requests.

Hugging Face Spaces CPU Basic tier provides 16 gigabytes of RAM at no cost, providing approximately 7.6 times the required memory and sufficient headroom for concurrent request handling. This platform selection is therefore the only viable option among free-tier alternatives for the complete AI component combination used in this system.

C. Docker Containerization Strategy

The Docker image is built from a python:3.11-slim base image to minimize the base layer size. System-level dependencies gcc and g++ are installed in a single RUN instruction to minimize layer count. The requirements.txt file is copied and installed before the application code to exploit Docker's layer caching mechanism — unchanged dependencies are not reinstalled on subsequent builds, reducing build time from approximately 15 minutes on the first build to approximately 3 minutes on subsequent builds with cached dependency layers.

The application exposes port 7860, which is the default port expected by Hugging Face Spaces. The CMD instruction launches the Flask application directly using Python rather than a production WSGI server, as Flask's built-in server is sufficient for the expected request volumes in this deployment scenario.

D. Lazy Initialization Architecture

A key performance optimization in the system architecture is the lazy initialization of the EasyOCR reader component. EasyOCR model loading requires approximately 30 seconds on the first invocation due to downloading and initializing

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

the CRAFT text detection model and ResNet-LSTM recognition model. If initialized at application startup, this delay would extend the time between container start and the application becoming available to serve requests.

The lazy initialization pattern defers EasyOCR loading until the first image bill upload request is received. A global variable ocr_reader is initialized to None at module load time. The extract_text_from_bill function checks whether ocr_reader is None before each use and initializes it if necessary. This pattern reduces application startup time by approximately 30 seconds while adding at most 30 seconds of latency to the first image processing request, which is an acceptable tradeoff given the infrequency of first requests relative to total system uptime.

The FAISS index and Sentence Transformer model, by contrast, are initialized eagerly at application startup because they are required for every claim processed at validation layer three. These components require approximately 5 seconds to load from pre-built index files stored on the container filesystem, representing an acceptable startup latency.

E. Three-Stage Validation Pipeline

The three-stage validation pipeline is designed to minimize language model API invocations, which are the highest-latency component of the processing pipeline at 1.5 to 4 seconds per invocation. Stage one performs amount validation using arithmetic operations completing in under 1 millisecond. Stage two performs exclusion keyword matching using string search operations completing in under 5 milliseconds. Claims resolved at stages one or two generate rejection responses without any network communication, achieving sub-10-millisecond end-to-end latency. Only claims passing both stages proceed to stage three, which invokes the FAISS retrieval pipeline and the Groq LLaMA API.

This design significantly reduces the average per-request API cost and latency when the claim population includes a substantial proportion of excluded conditions or amount violations, which is the expected case in production deployments.

IV. IMPLEMENTATION

A. Application Structure

The application is implemented in a single Python module app.py of approximately 500 lines. Global variables maintain references to the Flask application instance, the Groq API client, the FAISS index, the chunk list, the Sentence Transformer embedder, and the EasyOCR reader. These globals are initialized at module load time for eager components and set to None for lazy components.

The Flask application defines four routes. The root route serves the single-page HTML frontend. The /submit route handles POST requests containing claim form data and optional file upload. The /setup route returns API connection status. The /health route returns system readiness status including whether the FAISS index is loaded.

B. RAG Pipeline Implementation

The RAG pipeline consists of a build phase and a query phase. The build phase is triggered at startup when no pre-built index files are present on the filesystem. The insurance policy handbook PDF is read using PyPDF and split into 500-character chunks with 100-character overlaps. Each chunk is encoded using the all-MiniLM-L6-v2 Sentence Transformer producing 384-dimensional vectors. The vectors are stored in a FAISS IndexFlatL2 and serialized to disk alongside the chunk list for reuse across application restarts.

The query phase encodes the claim diagnosis text and retrieves the five most similar handbook chunks by L2 distance. These chunks are concatenated and included in the language model prompt as policy context. The complete query phase completes in approximately 60 milliseconds on CPU hardware, dominated by the Sentence Transformer encoding step.

C. Claim Processing Pipeline

The process_claim function implements the complete three-stage validation pipeline. Amount parsing handles multiple currency symbol formats including Rs., INR, and rupee symbol variants. The bill amount cross-verification compares

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

the claimed amount against the OCR-extracted bill total with a 10% tolerance to accommodate OCR extraction inaccuracies. The exclusion keyword list contains 24 terms covering the principal excluded condition categories defined in the policy handbook. The language model prompt is structured to elicit responses in five clearly labeled sections for reliable parsing.

D. Frontend Implementation

The frontend provides a professional dark-themed single-page interface. The left panel contains the claim submission form with fields for patient name, treatment date, address, medical facility, claim amount, and diagnosis. A drag-and-drop file upload zone accepts PDF and image format medical bills. The right panel displays the AI-generated compliance report including a color-coded verdict box, patient information grid, OCR bill analysis box, and four structured report sections. Three quick-test buttons auto-fill the form for demonstration purposes.

V. PERFORMANCE EVALUATION

A. Startup Performance

Application startup time was measured from container start to first successful HTTP response across five repetitions. With no pre-built FAISS index, startup time averaged 68 seconds due to handbook PDF processing, embedding generation, and index construction. With pre-built index files present, startup time averaged 8 seconds, dominated by Sentence Transformer model loading. These measurements confirm the importance of persisting FAISS index files across container restarts where the deployment platform supports persistent storage.

B. Request Processing Latency

End-to-end request processing latency was measured for 50 requests at each validation layer. Table II presents the results.

TABLE II: REQUEST PROCESSING LATENCY BY VALIDATION LAYER

Validation Layer	Median Latency	95th Percentile	Min	Max
Layer 1 — Amount validation	0.4 ms	0.9 ms	0.2 ms	1.8 ms
Layer 2 — Exclusion matching	3.1 ms	6.8 ms	1.9 ms	12.4 ms
Layer 3 — Full AI pipeline (no OCR)	2,840 ms	4,210 ms	1,620 ms	5,890 ms
Layer 3 — Full AI pipeline (with OCR)	4,120 ms	6,380 ms	2,940 ms	8,210 ms
FAISS retrieval only	0.6 ms	1.1 ms	0.3 ms	2.2 ms
Sentence Transformer encoding	72 ms	118 ms	48 ms	195 ms
Groq API response	2,140 ms	3,890 ms	1,310 ms	5,420 ms
EasyOCR image processing	1,180 ms	2,260 ms	740 ms	3,490 ms

The results confirm that Groq API response time is the dominant latency contributor for Layer 3 requests, accounting for approximately 75% of total processing time. FAISS retrieval and Sentence Transformer encoding together contribute approximately 10% of Layer 3 latency. The remaining 15% is attributed to Flask request handling, JSON serialization, and network transfer.

DOI: 10.48175/IJARSCT-35219

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

C. Memory Consumption Analysis

Memory consumption was profiled using Python's tracemalloc module over a 30-minute period including 100 simulated claim requests. Table III presents the results.

TABLE III: MEMORY CONSUMPTION PROFILE

Component	Memory Consumed	Notes
EasyOCR models	1,512 MB	After first image request
Sentence Transformer model	412 MB	Loaded at startup
FAISS index	8 MB	45 vectors, 384 dimensions
Flask application	142 MB	Including Python runtime
Peak during OCR processing	2,318 MB	Transient image buffers
Stable idle consumption	2,074 MB	After EasyOCR initialization

The stable memory consumption of approximately 2.1 gigabytes is well within the 16-gigabyte RAM allocation of Hugging Face Spaces CPU Basic tier, providing approximately 7.6 times headroom for concurrent request processing and OS overhead. This confirms that the platform selection is appropriate and the system can handle concurrent requests without memory pressure.

D. Comparative Platform Performance

To validate the platform selection analysis presented in Section III, deployment was attempted on Render.com free tier with 512 megabytes of RAM. The application terminated with an out-of-memory error during EasyOCR model initialization, confirming that the 512-megabyte limit is insufficient for this component combination. This validates the selection of Hugging Face Spaces as the only viable free-tier deployment platform.

E. Functional Test Results

Eight functional test cases were executed to validate system correctness across the principal claim processing scenarios. Table IV presents the complete results.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

TABLE IV: FUNCTIONAL TEST RESULTS

ID	Scenario	Claim Amount	Expected Verdict	Actual Verdict	Layer	Status
TC-01	Viral Fever — covered	Rs.4,000	ACCEPTED	ACCEPTED	3	PASS
TC-02	HIV — excluded	Rs.9,450	REJECTED	REJECTED	2	PASS
TC-03	Amount over limit	Rs.2,50,000	REJECTED	REJECTED	1	PASS
TC-04	Claim exceeds bill	Rs.6,000	REJECTED	REJECTED	1	PASS
TC-05	Diabetes — covered	Rs.8,500	ACCEPTED	ACCEPTED	3	PASS
TC-06	Cosmetic surgery — excluded	Rs.45,000	REJECTED	REJECTED	2	PASS
TC-07	Cancer chemotherapy — covered	Rs.1,80,000	ACCEPTED	ACCEPTED	3	PASS
TC-08	Bone fracture — covered	Rs.15,000	ACCEPTED	ACCEPTED	3	PASS

All eight test cases produced the expected verdict, demonstrating 100% functional correctness. The system correctly resolved four cases at validation layers one and two with sub-10-millisecond latency and correctly evaluated four complex cases at layer three using the full AI pipeline.

VI. CONCLUSION AND FUTURE WORK

This paper presented the design, implementation, and performance evaluation of a scalable cloud-based automated health insurance claim processing system. The central engineering contributions are a systematic cloud platform selection methodology demonstrating that Hugging Face Spaces is the only viable free-tier platform for AI component combinations requiring more than 512 megabytes of RAM, a lazy initialization architecture that reduces application startup time by 30 seconds without compromising steady-state performance, a Docker containerization strategy optimized for Python AI applications with large dependency trees, and a three-stage validation pipeline that minimizes expensive language model API invocations.

Performance benchmarking demonstrates median end-to-end latency of 3.2 seconds for full AI evaluation and sub-10-millisecond latency for claims resolved at validation layers one and two. Memory profiling confirms stable consumption of approximately 2.1 gigabytes, well within the 16-gigabyte allocation of the selected deployment platform. Functional testing across eight scenarios demonstrates 100% correctness. The system is deployed live at zero infrastructure cost, confirming the practical viability of the proposed approach for resource-constrained deployment scenarios.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

Future work will explore several directions. First, integration of a production WSGI server such as Gunicorn with worker process management would improve concurrent request handling under high load. Second, implementation of FAISS IVF indexing would reduce query latency for larger knowledge bases with hundreds of thousands of handbook chunks from multiple insurance providers. Third, evaluation of quantized language model variants such as LLaMA 3.3 8B would characterize the accuracy-latency tradeoff for scenarios where the 4-second language model latency is unacceptable. Fourth, integration with persistent cloud storage for FAISS index files would eliminate the 60-second index rebuild on fresh container deployments. Fifth, a load balancing layer with multiple Flask instances behind a reverse proxy would enable horizontal scaling for production deployments requiring higher throughput.

Acknowledgment

I would like to sincerely thank **Vandana Swami Ma'am, Department of Computer Science and Engineering, Raffles University**, for her valuable guidance, continuous support, and helpful suggestions throughout this project.

I am also grateful to **Rajendra Singh Sir, Dean, Department of Computer Science and Engineering, Raffles University**, for his encouragement, academic support, and motivation during this research work.

REFERENCES

[1] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Key challenges in cloud computing: Enabling the future internet of services," IEEE Internet Computing, vol. 17, no. 4, pp. 18-25, 2013.

[2] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.

[3] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," arXiv preprint arXiv:1812.03651, 2018.

[4] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," arXiv preprint arXiv:2210.17323, 2022.

[5] Groq Inc., "Groq LPU Inference Engine: Architecture and Performance," Technical Report, 2024. [Online]. Available: https://groq.com/technology/

[6] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535-547, 2021.

[7] M. Grinberg, Flask Web Development: Developing Web Applications with Python, 2nd ed. O'Reilly Media, 2018.

[8] 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," Advances in Neural Information Processing Systems, vol. 33, pp. 9459-9474, 2020.

[9] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in Proc. EMNLP 2019, arXiv:1908.10084, 2019.

[10] Jaided AI, "EasyOCR: Ready-to-use OCR with 80+ supported languages," GitHub, 2020. [Online]. Available: https://github.com/JaidedAI/EasyOCR

[11] Meta AI Research, "LLaMA: Open and efficient foundation language models," arXiv preprint arXiv:2302.13971, 2024.

[12] Pallets Projects, "Flask 3.0 Web Framework Documentation," 2024. [Online]. Available: https://flask.palletsprojects.com

[13] Hugging Face, "Spaces: Docker SDK Documentation," 2024. [Online]. Available: https://huggingface.co/docs/hub/spaces-sdks-docker

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," Advances in Neural Information Processing Systems, vol. 30, 2017.