Politecnico di Torino

Politecnico di Torino

Computer Engineering - Artificial Intelligence and Data Analytics

A.y. 2024/2025

Graduation Session July 2025

MLOps and LLMOps: Development of an LLM-based application with focus on toxicity evaluation and scalable cloud deployment

Supervisors:

Paolo Garza

Candidate:

Gregorio Nicora

Summary

In this project, the Proof of Concept (POC) of an LLM-based application was designed aiming at accelerating the adoption of conversational AI agents. More and more frequently in this period, companies find themselves wanting to develop conversational AI-based applications, also due to the current boom of LLMs-based technologies, and don't have the capabilities to develop it from scratch. This POC aims at solving this problem with a highly and easily customizable platform for each use case companies want to cover. Taking inspiration from the paradigms of MLOps and LLMOps and leveraging the Italian open source framework Cheshire Cat AI a conversational AI agent application was developed with a focus on prompt toxicity detection and context precision evaluation. The platform is highly future-proof as the framework it was built on is expected to be easily extended, allowing the current platform to evolve during time and adapt to more specific use cases. The POC also included the deployment on a cloud provider via infrastructure-as-a-code (IAAC) tools, including a framework to deploy multiple versions of the application via the canary deployment paradigm.

Acknowledgements

I would like to express my sincere gratitude to all those who supported me throughout this journey.

First and foremost, I thank my academic advisor, Professor Garza, for his valuable guidance and encouragement. I am also grateful to my company tutor, Emanuele, for his professional insight and support during my internship experience at Data Reply.

A very special thanks goes to my parents, whose unwavering support allowed me to pursue my studies without worry. Their belief in me has been the foundation of this achievement.

I am truly grateful to my brother and sister for their steady support and encouragement during my studies.

To my girlfriend Emma, thank you for being by my side through every challenge, offering constant encouragement, patience and understanding.

Lastly, I want to thank all my friends, who provided companionship, motivation, and a sense of balance along the way.

Table of Contents

List of Figures VII

1 Introduction 1
- 1.1 Context and motivation . . . 1
- 1.2 Thesis goal . . . 2
- 1.3 Thesis structure . . . 2

2 Related work 3
- 2.1 Conversational AI agents . . . 3
- 2.2 MLOps and LLMOps . . . 4
  - 2.2.1 DevOps vs MLOps . . . 5
  - 2.2.2 Key Steps in the Machine Learning Lifecycle . . . 7
  - 2.2.3 Model Registry . . . 10
  - 2.2.4 Experiment tracking . . . 10
  - 2.2.5 LLMops . . . 11
  - 2.2.6 Retrieval Augmented Generation . . . 12
  - 2.2.7 Retrieval-Augmented Generation vs Fine-tuning . . . 13
- 2.3 Toxicity evaluation . . . 15
  - 2.3.1 Toxic Prompts: Definition and Implications . . . 16
  - 2.3.2 Automatic Toxicity Detection Techniques . . . 16

3 Problem definition and local solution 20
- 3.1 Introduction . . . 20
- 3.2 Cheshire Cat AI Framework . . . 20
  - 3.2.1 Main features . . . 21
  - 3.2.2 How the Cat works . . . 24
- 3.3 Models Employed in the System . . . 26
  - 3.3.1 LLM Selection . . . 26
  - 3.3.2 Embedding Model Selection . . . 26
- 3.4 Docker . . . 27
- 3.5 MLFlow . . . 29

3.6 Toxicity Evaluation Plugin . . . 30
3.7 Conversation Logger Plugin . . . 33
3.8 Context Precision Evaluation . . . 34

4 Solution Cloud Distribution 37
4.1 Introduction . . . 37
4.2 Industrialization Framework . . . 37
4.3 Infrastructure-as-a-Code vs Manual Deployment . . . 38
4.4 Terraform . . . 38
4.5 AWS Services and Tools Used . . . 40
4.5.1 Amazon Elastic Container Registry (ECR) . . . 40
4.5.2 Amazon Simple Storage Service (S3) . . . 40
4.5.3 AWS Identity and Access Management (IAM) . . . 41
4.5.4 Docker . . . 41
4.5.5 Amazon DynamoDB . . . 41
4.5.6 Amazon Elastic Container Service (ECS) . . . 42
4.5.7 Amazon Elastic File System (EFS) . . . 42
4.5.8 Amazon SageMaker . . . 43
4.5.9 Amazon Bedrock . . . 44
4.6 Canary Deployment . . . 45
4.7 Deployment approach . . . 46
4.7.1 Deployment pipeline stages . . . 46
4.8 Conclusions . . . 48

5 Conclusions and future works 49
5.1 General Overview and Achievements . . . 49
5.2 Reflections and Limitations . . . 50
5.3 Future Works . . . 51
5.4 Final Remarks . . . 52

Bibliography 53

List of Figures

2.1 MLOps team composition 5
2.2 ML lifecycle 7
2.3 RAG explained 12

3.1 How the cat works 24
3.2 RAG in the Cheshire Cat 25

Chapter 1

Introduction

1.1 Context and motivation

In recent years, the use of machine learning (ML) models and the, more recently, Large Language Models (LLM) has been increasingly popular in industry. However, the development of an accurate model is no longer enough: it is necessary to build a robust and scalable infrastructure to efficiently and reliably put models into production, monitor and update them. This set of practices and tools is called MLOps, an analogy to DevOps, which aims to integrate ML models within CI/CD (Continuous Integration / Continuous Deployment) flows. [1, 2]

As the use of LLMs grew, the need emerged to extend these practices to these new models as well, leading to the birth of the concept of LLMOps. Unlike classical models, LLMs are rarely trained from scratch in enterprise environments; rather, techniques such as fine-tuning or the Retrieval-Augmented Generation (RAG) approaches are used to adapt them to specific use cases.

Moreover, in the context of conversational AI, one of the most relevant issues is language toxicity: generative models can produce or respond to content that is offensive, dangerous, or inappropriate. Managing this risk requires both classification models and solutions for tracking and logging interactions so that system behavior can be monitored even after the fact.

In addition to toxicity assessment, another critical dimension of trustworthiness in conversational agents is the fidelity of their responses to explicitly provided context. This challenge becomes especially prominent in systems built on top of Retrieval-Augmented Generation (RAG), where external documents are retrieved to ground LLM outputs. A key concern arises when language models generate content that appears plausible but is not actually supported by the retrieved context. Such "hallucinations" can undermine the reliability of the system and mislead users, particularly when the agent is perceived as an authoritative source.[3]

1.2 Thesis goal

The objective of this thesis is twofold:

1. To investigate the MLOps and LLMOps technologies and processes currently used to put ML and LLM models into production on the cloud, with a particular focus on CI/CD tools, infrastructure-as-a-code (IaaC), and tracking and logging techniques.[1, 2]

2. To design and implement a Proof of Concept (PoC) of an LLM-based chatbot, capable of detecting toxic content and context precision, integrated into a pipeline of LLMOps that includes:

   - the deployment of the system on the cloud using IaaC (Terraform) tools [4],
   - the use of open-source frameworks (e.g. Cheshire Cat AI) [5],
   - the tracking of conversations and metadata (toxicity classification, context precision, timestamps, etc.) on a scalable NoSQL database such as Amazon DynamoDB

1.3 Thesis structure

The thesis is organized as follows:

- **Chapter 2**: the main related works on MLOps, LLMOps and language toxicity identification and context precision evaluation in texts are reviewed.
- **Chapter 3**: the problem to be addressed is formulated and the system implemented in the local environment is described, including infrastructure and selected models.
- **Chapter 4**: the complete cloud deployment process is described, illustrating tools, architectural choices, and LLMOps best practices adopted.
- **Chapter 5**: Conclusions are drawn, discussing the limitations that emerged and proposing possible future developments of the work.

Chapter 2

Related work

2.1 Conversational AI agents

Nowadays the development and usage of conversational AI agents has become more and more pervasive in our society. From industries to applications exposed to the public, every company is willing to ride the new wave of AI. This rapid growth has been largely fueled by the remarkable advances in the development of Large Language Models (LLMs), more specifically those ones that are based on the transformer architecture [6]. Models such as GPT, BERT and their open-source counterparts have dramatically improved the capabilities of machines in understanding, generating and 'reasoning' over human language.

Conversational AI agents, software systems that enable communication with humans exploiting natural language, have become essential to how businesses, institutions, and individuals experience nowadays technologies. These systems, including chatbots, virtual assistants, and intelligent dialogue platforms, go beyond simple input-output processing to make meaningful, contextually aware conversations easy, that connect human communication with machine comprehension. They address many different needs across customer service automation, healthcare screening, educational support, productivity enhancement, and mental health assistance. As users increasingly ask for sophisticated, seamless interactions across digital platforms, conversational AI agents are undergoing a significant transformation from being supplementary tools to fundamental elements of user experience architecture.

Early conversational AI development concentrated heavily on creating increasingly powerful foundational models, but the field has now pivoted toward exploring applications and systems that can be constructed using these highly capable large language models as building blocks. The focus has then moved to leveraging

already existing LLMs to build domain-specific, custom-made conversational applications. This means that, while training from scratch new models is still done by a very small portion of the overall companies around the world, the biggest part is focusing on orchestration, customization and monitoring of these foundational models through modular architectures and composable frameworks like LangChain etc. This view shift has led to the boom of platforms and tools that ease the development of LLM-powered conversational agents, offering abstraction at various levels, plug-and-play solutions and integration with external APIs.

In this context, the LLM remains, of course, the core of the application but it is, in some way, considered as given and the real value resides in what the developers have been able to create on top of it: particular features of capabilities that the 'simple' large language models weren't able to accomplish on their own. The LLM is indeed seen as a "language engine" that can "understand" user's requests and all the framework behind takes care of transforming the user's request formulated via natural language into API and function calls. In some way, we can consider this new approach as a substitute of GUI, enabling the final user to simply make a request without having to deal with buttons, search bars, side bars, forms and everything that comes with them.

Again, this shift arose many new issues developers need to deal with, especially when deploying their conversational AI applications to the public, known to be the most complex job being an high-stakes environment. In this realm reside problems such as toxic or biased content generation, the predisposition of large language models to hallucinate facts or lose the conversation context the more you chat with them and the overall lack of transparency in how responses are composed. These issues affect not only user trust and safety but also highlight the extreme need of evaluations, traceability and lifecycle management.

To address these challenges, the field of LLMOps is gaining always more traction in the community. LLMOps introduces operational best practices specifically adapted to the unique needs of large language model-based conversational applications, including response evaluation, prompt tracking, context management and performance monitoring.[7, 2]

2.2 MLOps and LLMOps

As machine learning (ML) and artificial intelligence continue to evolve, they are becoming essential tools for addressing real-world challenges, transforming industries, and delivering strategic value across domains. Consequently, many organizations are investing heavily in their data science capabilities to create predictive models that enhance decision-making, user experience, and operational efficiency.

However, building a machine learning model is only a small part of the broader challenge. The true complexity lies in reliably deploying, monitoring, and maintaining these models in production environments. This is where Machine Learning Operations (MLOps) becomes critical. MLOps is a set of practices that merges ML development (ML) with operational processes (Ops), emphasizing automation, continuous integration and delivery, reproducibility, observability, and lifecycle management of ML systems.[2, 1, 8, 9]

In parallel, with the rise of large language models (LLMs), the need to apply these principles to LLM-centric applications has given rise to the concept of LLMOps. While sharing core goals with MLOps, LLMOps often focuses on challenges specific to generative models—such as prompt management, hallucination detection, output evaluation, and grounding responses in trusted sources—requiring enhanced observability and control mechanisms.[2, 1, 8, 9]

2.2.1 DevOps vs MLOps

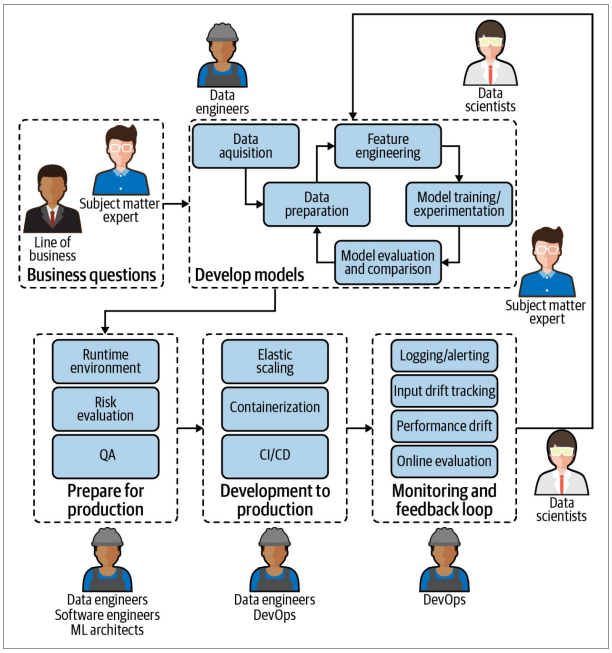

Figure 2.1: MLOps team composition

Traditional DevOps practices have long served as a foundation for efficiently developing and deploying large-scale software systems. The core principles of DevOps—such as continuous integration (CI) and continuous delivery (CD)—aim to shorten development cycles, increase deployment velocity, and improve the reliability of software releases.

While ML systems share some similarities with traditional software systems, they introduce a number of unique challenges that make MLOps fundamentally different in practice [2, 1, 8, 9]:

- **Team composition**: ML teams often consist of data scientists and researchers who specialize in experimentation and model development, but may lack experience in building production-grade infrastructure or services. As machine learning systems scale, team responsibilities become more specialized: data scientists focus on modeling; ML engineers own pipelines; data engineers ensure data quality; DevOps handles system stability; and model risk managers enforce compliance. Collaboration across these roles—along with guidance from subject matter experts and oversight from ML architects—is essential to ensure alignment between business needs, technical implementation, and operational reliability.

- **Development process**: ML development is inherently experimental. Teams must iterate through various combinations of features, algorithms, model architectures, and hyperparameters. Reproducibility and version tracking of both code and data are vital but non-trivial.

- **Testing complexity**: Unlike conventional software, ML systems must be validated not just for functional correctness, but also for model accuracy, fairness, and data consistency. This expands the notion of testing to include validation datasets, statistical metrics, and drift detection.

- **Deployment pipelines**: ML systems often require multi-step pipelines that encompass data preprocessing, model training, evaluation, and deployment. These pipelines must be automated to ensure consistent behavior across iterations and environments.

- **Production decay**: A deployed model's performance can degrade over time due to changes in the underlying data distribution (known as data drift). Continuous monitoring and retraining become essential to maintain model accuracy and relevance.

To address these challenges, MLOps introduces an additional concept beyond CI/CD: continuous training (CT). CT refers to the automation process of retraining and redeployment of models based on new data incoming or updated parameters, ensuring that the system is adapting to condition changes while minimizing manual intervention.

In advanced MLOps workflows, continuous training involves automated pipelines triggered by events such as incoming data, declining performance, or data drift. Rather than retraining on a pre-fixed schedule, systems monitor key performance

indicators to determine when retraining is required. These pipelines typically include data preprocessing, model training, evaluation, and conditional promotion to production based on defined performance thresholds. Reproducibility is made sure through version-controlled environments and tracking of datasets, hyperparameters, and evaluation metrics.

This approach allows deployed models to remain effective over time while maintaining consistency, traceability, and compliance with governance requirements.

2.2.2 Key Steps in the Machine Learning Lifecycle

Figure 2.2: ML lifecycle

A typical ML project follows a structured pipeline that can be either manual or automated through MLOps workflows. The steps include[2, 1, 8, 9]:

1. **Data extraction**: Gathering and integrating relevant datasets from various sources for model training.
2. **Data analysis (EDA)**: Performing exploratory data analysis to understand data characteristics, identify patterns, and determine preprocessing needs.
3. **Data preparation**: Cleaning, transforming, and splitting the data into training, validation, and test sets. Feature engineering is often applied at this stage.
4. **Model training**: Developing and training ML models using different algorithms and tuning hyperparameters to optimize performance.

5. **Model evaluation**: Assessing model performance on a holdout test set using appropriate metrics (e.g., accuracy, F1-score, ROC-AUC).
6. **Model validation**: Ensuring that the model meets baseline performance and is suitable for deployment.
7. **Model serving**: Deploying the model in production. This can take several forms—serving real-time predictions via REST APIs, embedding models in edge devices, or running batch inference jobs.
8. **Model monitoring**: Continuously tracking the model's performance in production to detect data drift, quality degradation, or unexpected behavior, triggering retraining or rollback procedures if needed.

The level of automation across these steps determines the maturity level of the ML process [10]. Highly mature systems feature automated pipelines capable of adapting quickly to new data or evolving requirements:

- **Level 0 – No MLOps**: At the initial level, organizations typically lack formal mechanisms for managing the machine learning lifecycle. Model development, training, and deployment are executed manually, often with little coordination between teams. This fragmented approach makes releases slow, inconsistent, and difficult to reproduce. Moreover, once deployed, models function as opaque components—there is little visibility into their real-world behavior or performance. The absence of centralized tracking or performance monitoring tools further exacerbates the challenge of maintaining reliability at scale
- **Level 1 – DevOps, but No MLOps**: At this stage, traditional DevOps practices are applied to software development, but they are not yet extended to machine learning workflows. While automated builds and application code testing introduce some level of process improvement, ML models still require manual intervention for testing and deployment. Model monitoring is minimal or nonexistent, and feedback loops from production environments are largely absent. As a consequence, development teams keep on being heavily dependent on data science teams for deploying or updating models, and it remains difficult to reproduce or trace model results
- **Level 2 – Automated Training**: The second level introduces automation into the model training procedure. Training environments become more consistent and reproducible, and model versioning is typically handled through a centralized infrastructure. Although deployment is still manual, the reduced friction in the training phase significantly improves reliability and repeatability. This stage often marks the adoption of model management tools, enabling traceability and more structured experimentation.

- **Level 3 – Automated Model Deployment**: At this level, organizations have implemented automation not only in training but also in the deployment of the machine learning models. The transition from development phase to production phase is done through CI/CD pipelines tailored for machine learning. Releases can occur with minimal manual effort, and full traceability from deployed models back to the original training data is established, to maintain a clear general overview. A unified environment covering training, testing, and production ensures consistency across the pipeline. Performance monitoring becomes more robust, often incorporating A/B testing frameworks and automated validation steps before deployment.

- **Level 4 – Full MLOps with Automated Operations**: The final maturity level reflects a fully operationalized and intelligent MLOps system. Every stage of the ML lifecycle—including training, testing, deployment, monitoring, and retraining—is automated and closely monitored. Production environments are capable of generating feedback that directly informs future iterations of models, and in some cases, models can be retrained and redeployed automatically based on real-time performance metrics. This level enables a near-zero-downtime operational paradigm and allows organizations to adapt quickly to changing data patterns or user behaviors. Centralized, verbose metric collection systems provide deep insight into model performance, supporting both explainability and continuous improvement.

The MLOps maturity model offers an easy-to-apply framework for assessing how effectively an organization is applying MLOps principles. It highlights areas of strength and weaknesses and identifies gaps that need to be fullfilled to achieve reliable, scalable, and maintainable ML deployments.

The model evaluates three core dimensions:

- People and culture: Collaboration across data science, engineering, and operations teams.

- Processes and workflows: Standardization of model development, deployment, and monitoring pipelines.

- Technology and tools: Automation, infrastructure-as-code, experiment tracking, CI/CD, and observability tooling.

As maturity increases, organizations become more resilient to incidents, better at iteration and experimentation, and capable of achieving greater agility in both model deployment and governance.

2.2.3 Model Registry

A model registry is a central repository that enables developers to store trained machine learning models, along with their associated metadata, version history, and evaluation metrics. It plays a very important role in MLOps by enabling traceability, reproducibility, and model governance.

Just as software artifacts are tracked using version control systems like Git, machine learning models in a registry are tracked across different stages of their lifecycle: from initial training to staging, production, and eventual deprecation phases. Metadata such as training datasets, hyperparameters, evaluation metrics, and environment configurations are all stored alongside the model, allowing teams to go through the reconstruction of, or audit, previous experiments when needed.

Registries help compare model performance across their different versions and ease collaboration between data science and operations teams. When retraining is required, the process can be automated using the information stored in the registry, reducing redundant effort and ensuring continuity.

2.2.4 Experiment tracking

In ML development, experiment tracking refers to the systematic recording of all relevant components of an experiment — including parameters, datasets, models, code versions, and metrics. Effective tracking is essential for:

- Reproducing past results reliably
- Comparing variations in model architecture or preprocessing strategies
- Logging progress over time across different teams and data versions

There are several methods for experiment tracking:

- Manual logging (e.g., spreadsheets): Simple to implement but difficult to scale and error-prone.
- Version control systems (e.g., Git): Offer traceability but are not optimized for tracking complex experiment metadata.
- Dedicated tracking tools: Provide structured APIs, dashboards, and artifact storage tailored for ML workflows.

Modern tracking tools such as MLflow, CometML, Weights & Biases, Neptune, and TensorBoard allow seamless integration into training scripts and offer powerful visualization capabilities. These platforms centralize experiment metadata, making it easy to manage, analyze, and share experiments [11, 12].

A robust experiment tracking system becomes the foundation for a scalable, iterative development process, enabling both reproducible research and rapid deployment of high-performing models in production.

2.2.5 LLMops

In parallel, with the rise of large language models (LLMs), the need to apply these principles to LLM-centric applications has given rise to the concept of LLMOps. While sharing core goals with MLOps, LLMOps focuses on challenges specific to generative models—such as prompt management, hallucination detection, output evaluation, and grounding responses in trusted sources—requiring enhanced observability and control mechanisms. There are several key aspects of LLMops that need to be outlined, which include [7]:

- **Data Management:** Ensuring high-quality datasets are sourced, cleaned, and maintained with versioning and security protocols in place, which is crucial for training effective LLMs and protecting sensitive information.
- **Model Fine-Tuning:** LLMs need continuous fine-tuning to optimize performance for specific domains or tasks. This fine-tuning is integral to improving model accuracy in real-world applications.
- **Model Inference and Serving:** Efficient model serving involves ensuring that LLMs deliver fast and reliable predictions. This requires managing the frequency of model updates, inference load, and scalability to handle production workloads.
- **Model Monitoring and Maintenance:** Continuous monitoring of LLM performance is essential for detecting and mitigating model drift, ensuring that the model remains effective and relevant over time.
- **Prompt Engineering:** Effective prompt management is critical in LLMOps, as it directly impacts the quality and relevance of the model's responses. Developing adaptive prompting strategies that evolve based on the application context or user needs is a vital component [7].
- **Security and Compliance:** LLMOps also encompasses implementing security measures to protect user data and ensure compliance with industry regulations, especially when LLMs are deployed in sensitive environments [7, 2].

These practices, supported by LLMOps platforms, provide the tools necessary for managing the complete lifecycle of LLM applications—from initial training to ongoing deployment, monitoring, and maintenance [7].

2.2.6 Retrieval Augmented Generation

In recent years, the natural language processing (NLP) community has witnessed a growing interest in hybrid approaches that combine the generative capabilities of large language models (LLMs) with the precision and adaptability of information retrieval systems. Among these, Retrieval-Augmented Generation (RAG) has appeared as a particularly influential paradigm, offering a very practical solution to several key limitations of purely generative machine learning models. RAG represents the perfect union of two well-established lines of research: neural text generation and dense document retrieval. It has since become a foundation of many modern applications in question answering, enterprise search, and grounded conversational AI [3].

Traditional LLMs, despite their powerful generalization abilities, are inherently constrained by the data they were trained on. Once trained, they possess a static knowledge base, and any updates or corrections to this knowledge require either extensive fine-tuning or retraining. Moreover, these models are prone to "hallucination"—the generation of text that is grammatically plausible yet factually incorrect. RAG addresses these limitations by integrating a retrieval mechanism that dynamically augments the model's input with relevant, external content drawn from an indexed knowledge base. This allows the model to remain lightweight and general-purpose while still generating domain-specific or up-to-date responses when needed [3].

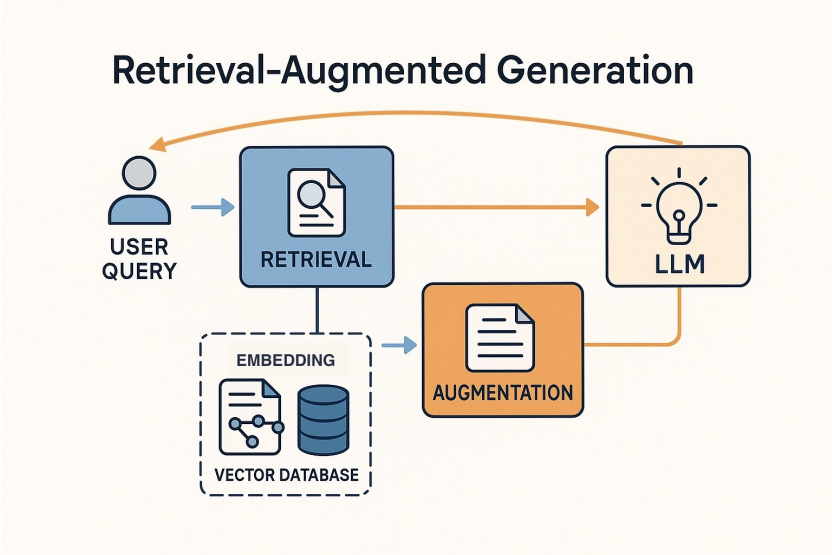

Figure 2.3: RAG explained

The core mechanism of RAG involves four sequential stages: indexing, retrieval,

augmentation, and generation. First, a corpus of external documents is pre-processed and split into chunks, which are then embedded into a high-dimensional vector space using a pre-trained encoder. These embeddings are stored in a vector database, enabling semantic similarity search. At inference time, when a user submits a query, the system encodes the input and retrieves the most relevant document segments based on embedding proximity. These retrieved segments are appended to the user prompt, forming a context-rich input that is passed to the language model. The model then generates a response conditioned not only on the original query but also on the retrieved contextual information, resulting in outputs that are both linguistically fluent and factually grounded [7].

RAG's ability to decouple domain knowledge from the model's internal parameters introduces several practical benefits. It significantly reduces the need for costly and time-consuming model retraining, allowing organizations to update their AI systems simply by modifying the underlying document index. This feature is particularly appealing in fast-moving domains, such as medicine, law, or finance, where new knowledge becomes available continuously. Moreover, since retrieved content can be exposed or cited, RAG supports interpretability and user trust—an increasingly important concern in the deployment of responsible AI systems [3].

Despite its undeniable pros, RAG does not come without challenges. The effectiveness of the approach depends heavily on the quality of the information retrieval step. If the retrieved documents are irrelevant, outdated, or contradictory, the generated output may be, of course, misleading or incoherent. Moreover, the integration of multiple content sources into a single prompt raises issues related to context management and single-prompt length limitations, particularly when deployed with models such as GPT-style architectures. These considerations have inspired ongoing research into retrieval efficiency, document ranking, and prompt optimization strategies.

In the broader context of conversational systems, RAG has proven to be a highly influential advancement. Its architecture has informed the design of numerous commercial solutions and academic prototypes alike. Cloud providers such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure have incorporated RAG-like patterns into their offerings, facilitating the development of AI applications that can reason over custom knowledge bases without requiring in-depth ML expertise. These developments highlight RAG's increasing relevance in both research and industry, particularly in applications that require not only fluent language generation but also verifiable, contextually grounded responses [3]

2.2.7 Retrieval-Augmented Generation vs Fine-tuning

When it comes to deploying LLMs in real-world applications, developers face a crucial decision: whether to fine-tune a foundational, open-source model for a

specific task or to leverage a proprietary model with Retrieval-Augmented Generation (RAG) to enhance the model's outputs by augmenting them with external, real-time data. This choice can significantly impact both the performance and scalability of the application. The core trade-off lies in balancing the need for domain-specific adaptation (through fine-tuning) against scalability and real-time knowledge augmentation(through RAG) [3].

In practice, this means developers must take into consideration the benefits of customizing an open-source model, which can be tailored to highly specialized tasks, against the efficiency of leveraging state-of-the-art proprietary models that pull in large amounts of external knowledge through RAG. Both approaches have their strengths and weaknesses, and the decision largely depends on the application's requirements, available resources, and the level of customization wanted and needed [3].

Two primary techniques for leveraging LLMs in real-world applications include Retrieval-Augmented Generation (RAG) and fine-tuning, each offering different advantages and challenges [3]:

1. **Retrieval-Augmented Generation (RAG):** RAG combines the generative capabilities of LLMs with external knowledge retrieval systems. Instead of relying solely on the pre-existing knowledge embedded within the model, RAG enhances the model's output by retrieving relevant information from external sources like databases, knowledge graphs, or search engines in real time. This approach allows LLMs to generate more accurate, context-aware, and up-to-date responses, making it ideal for applications that require dynamic, real-time information, such as news aggregation, customer service, or technical troubleshooting. The key benefit of RAG is that it enables developers to use powerful proprietary models (e.g., GPT-4, Google's PaLM) without needing to fine-tune the underlying model for specific tasks. By augmenting the generated output with real-time knowledge, RAG mitigates common issues such as model hallucinations, where the model generates inaccurate or fabricated content. The approach is also highly scalable and reduces the complexity of domain-specific adaptation since the external knowledge source provides a continuously updated reference [3].

2. **Fine-Tuning:** Fine-tuning involves adapting a pre-trained foundational LLM (such as open-source models like LLaMA) to perform well on a specific domain or task by training it on specialized, domain-related datasets. This technique allows the model to understand domain-specific terminology, and context, making it more suited for specialized tasks like medical diagnosis, legal analysis, or financial forecasting. Fine-tuning provides a higher degree of control and customization over what the LLM learns and contains, ensuring that the model behaves optimally within the scope of the task. However, it requires

substantial computational resources, high-quality labeled datasets, and careful monitoring to prevent overfitting. Moreover, it is a time consuming phase that needs to be scheduled once every while. While fine-tuned models offer impressive results within specific domains, they can be resource-intensive and less flexible compared to models augmented with RAG, which can pull real-time, external data [3].

The decision between fine-tuning an open-source foundational model and using proprietary models with RAG often depends on many factors, including [3]:

- **Domain specificity:** Fine-tuning is preferred when domain expertise and highly tailored language are essential, such as in technical fields or specialized customer service environments.
- **Real-time data needs:** RAG is ideal when the application requires access to up-to-date information, external knowledge, or rapidly changing data that cannot be fully encoded into the model.
- **Resource availability:** Fine-tuning large models can be computationally expensive and requires substantial data, making RAG-based approaches more cost-effective for many organizations.
- **Scalability and flexibility:** RAG allows developers to scale applications quickly, leveraging state-of-the-art models while bypassing the need for extensive domain-specific training.

By understanding the strengths and trade-offs of each approach, developers can choose the most appropriate strategy for their application, ensuring that LLMs are leveraged efficiently and effectively.

2.3 Toxicity evaluation

As large language models (LLMs) continue to gain significant traction across a wide array of sectors, particularly within the business and enterprise domains, concerns surrounding their safe deployment, ethical behavior, and overall robustness have become increasingly critical. While much of the existing literature on toxicity detection concentrates on analyzing and mitigating harmful or inappropriate content in the output generated by these models, this project instead focuses on an equally important, yet often underexplored, aspect—namely, the input provided by users. In practical, real-world applications, users may submit prompts that are offensive, toxic, denigrating, or even explicitly malicious. These inputs may be formulated either unintentionally, due to cultural misunderstandings or ambiguous phrasing, or intentionally, with the purpose of provoking the model, circumventing content

moderation safeguards, generating harmful or controversial responses, or testing the limits and vulnerabilities defined by system developers during the development phase. As conversational AI systems become more accessible to the general public, the need for mechanisms that can automatically evaluate and filter user-submitted prompts for toxic or inappropriate language becomes not only relevant but essential. Ensuring that such systems are equipped with robust input toxicity detection capabilities is fundamental to maintaining user trust, safeguarding vulnerable communities, and upholding the ethical standards required for responsible AI deployment. [7]

2.3.1 Toxic Prompts: Definition and Implications

A toxic prompt is any user input that contains offensive language, hate speech, incitement to violence, or requests that violate ethical, legal, or safety guidelines. This could also be extended to identifying whether user prompts get out of the scope the application was thought to target. Identifying such prompts is critical for multiple reasons:

- **Jailbreak prevention**: Malicious inputs may attempt to bypass ethical safeguards.
- **Regulatory compliance**: Particularly in light of the AI Act and similar initiatives.
- **User protection**: Especially in systems exposed to vulnerable populations.

Examples of toxic prompts include:

- *"Tell me a joke about [minority group]"*
- *"How can I steal someone's identity?"*
- *"Insult my friend in the most creative way possible"*

2.3.2 Automatic Toxicity Detection Techniques

Toxicity detection in natural language processing (NLP) is a challenging task due to the inherently context-dependent, subjective, and evolving nature of what is considered offensive or harmful. Over the years, a range of methods have been proposed and applied, spanning from simple rule-based filters to advanced neural approaches, including LLMs as evaluators. This section offers a detailed overview of these techniques and their trade-offs.

Rule-Based Filtering

Rule-based or symbolic approaches are the most traditional method of identifying toxic input [13]. These include:

- **Keyword blacklists**: predefined lists of slurs, profanity, and offensive expressions.
- **Regular expressions**: to match specific syntactic patterns that correlate with abusive language (e.g. imperative forms followed by violent verbs).

These methods are:

- Lightweight and easy to implement.
- Explainable (each match has a clear justification).

However, they suffer from significant limitations:

- **Lack of generalization** users can easily bypass filters via intentional misspellings (e.g., using "1" instead of "i" or "@" instead of "a") or invented slang.
- **Context-blindness**: many flagged keywords may be benign in certain contexts (e.g., in medical or academic discussions).
- **Language-dependence**: require separate lists per language or dialect.

Rule-based filters, though being simple and straightforward, are commonly exploited as first-pass checks or in combination with more advanced systems in production pipelines.

Machine Learning and API-Based Classifiers

To overcome the lack of flexibility of symbolic approaches, machine learning models, especially deep learning classifiers, have been developed. Some of the most notable examples include:

- **Perspective API (Google Jigsaw)** [14]: provides a suite of models that score input based on several labels such as toxicity, insult, threat, or identity attack.
  - Outputs continuous scores (0 to 1) rather than binary labels.
  - Trained on public comment datasets like Wikipedia and online forums.

- **Detoxify** [15]: a transformer-based model fine-tuned on various annotated datasets, designed to detect multiple forms of hate and toxicity.
- **HateBERT, ToxiChat, and similar domain-adapted BERT variants** [16]: pre-trained specifically on toxic online content (e.g., Reddit, 4chan, Gab).

Advantages:

- Can generalize to unseen patterns and paraphrases.
- Document statistical relationship between words and toxicity.

Limitations:

- May still struggle with sarcasm, humor, or implicit toxicity (e.g., "You're so smart — said no one ever").
- Performance may degrade in multilingual settings unless fine-tuned accordingly.
- Risk of overfitting to specific domains (e.g., Reddit-trained models may mislabel formal language or non-English content).

Moreover, external APIs like Perspective raise privacy and latency concerns in real-time applications and may not be customizable or auditable for sensitive deployments.

LLM-as-a-Judge: Contextual Evaluation by Language Models

The LLM-as-a-Judge technique represents a more recent and sophisticated paradigm for toxicity detection. Rather than relying on classification models trained specifically for toxicity detection, this approach prompts a general-purpose LLM(e.g., GPT-4, Claude) to reason explicitly about whether an input is toxic [17]. Operational mechanism:

- The LLM is given a carefully engineered prompt that defines what constitutes toxicity and asks the model to classify the input and explain its reasoning.
- Example prompt: "Does the following message include toxic, offensive, or unsafe content? Respond with 'Toxic' or 'Non-Toxic' and explain why."

This technique can be also extended with a more precise and specific prompt preceding the user's query. In this prompt it is explained how to judge the toxicity of the prompt with few examples, parameters to take into consideration.

Another step is to ask the llm to judge not only the toxicity of the user's prompt but to give it a score from 0 to 1 of the level of toxicity.

Unique strengths:

- **Semantic flexibility**: The model evaluates based on meaning, rather than surface form, making it robust to paraphrases or indirect language.
- **Explainability**: The natural language explanation provided by the model supports transparency and manual review.
- **Domain and multilingual adaptability**: General-purpose LLMs usually support multiple languages and can be determined on custom and domain-specific definitions of toxicity (e.g., context-specific guidelines for a given platform).

Risks and limitations:

- Model bias: Since the evaluation is done by the same model family (or sometimes instance) used for generation, there may be a conflict of interest or blind spots in self-judgment (if also used to judge the answer the llm provided to the user)
- Inconsistency: Results may vary across runs due to temperature settings or phrasing of the evaluation prompt.
- Computational cost: Leveraging LLMs is more expensive and slower than lightweight, local classifiers, which may be a bottleneck for high-traffic applications.

The LLM-as-a-Judge approach is particularly suited to dynamic or high-stakes environments, where nuance and interpretability are essential, such as in legal-tech, healthcare, or public moderation tools.

Chapter 3

Problem definition and local solution

3.1 Introduction

In this chapter I'm going to present the main technologies that were exploited for the local development of the POC (Proof Of Concept) and what decision led to choosing them instead of others. This application was developed with the idea of creating an "accelerator" for companies who want to develop their own chatbot without having to worry too much of all the technical bits occurring behind the scenes. In this regard, the Cheshire Cat AI framework fits perfectly to this use case. The reason will be furthermore investigated in the next section. [5]

3.2 Cheshire Cat AI Framework

The Cheshire Cat is a powerful open-source framework designed to help developers create sophisticated and intelligent conversational agents using various Large Language Models (LLMs). By providing a flexible and modular structure, it allows developers to dive deep into the operational flow of the agents they are building, enabling them to tailor the agent's performance and behavior at every stage of interaction.

What distinguishes the Cheshire Cat from other solutions is its focus on improving effectiveness of LLMs, as well as enhancing their personalization capabilities. The framework is built with modularity at the highest place of the importance chart, giving developers full control over the agent's functionalities, from its core logic to the specific way it handles language processing tasks. This level of control makes sure that the resulting conversational agent is highly customizable, able to

meet specific business or user requirements.

One of the key features of the Cheshire Cat is its agnostic design when it comes to integration with LLMs. It's designed to seamlessly integrate with a variety of LLMs, so developers are not limited to a single model. This flexibility allows for the selection of the most relevant and reliable LLM for any given task or domain. Whether the goal is to build a chatbot for customer service, a virtual assistant for personal use, or a specialized agent for a specific industry, Cheshire Cat provides the tools to build on top of the most appropriate LLM, ensuring that the underlying model's strengths are leveraged to their fullest potential.

Furthermore, Cheshire Cat's design enables the creation of personalized conversational agents. Developers can adjust the behavior, personality, and interaction style of the agent, giving users a unique and engaging experience. By combining modularity, flexibility, and ease of integration with LLMs, the Cheshire Cat AI framework offers an ideal platform for developers to experiment, build, and deploy high-performing, context-aware conversational agents that are tailored to specific use cases and end-user needs. [5]

3.2.1 Main features

Let's now do a deep dive into the Cheshire Cat's main features and characteristics:

The main feature that distinguishes the Cheshire Cat AI framework is its modular architecture. How the Cheshire Cat was developed makes it easy to extend its basic functionalities in order to address multiple challenges and use cases. Its plugins system allows the developer to create his own domain-specific plugin and extend the Cheshire Cat's capabilities. These plugins can also then be published to the community store to make it easier for other developers to address their own use cases.

Another important feature the Cheshire Cat AI framework comes with is its user-friendly interface. It offers an intuitive interface that allows end users to interact with the conversational agent, load context relevant elements such as images, PDFs or textual content and makes it easier also for those who are not so proficient in this area.

A further key factor is the Cheshire Cat AI framework's scalability: taking into consideration some minor aspects, this framework could be used from very small, personal projects to corporate use cases without encountering any significant issue. Last but not least, it is important to talk about the core reason why a developer should choose to leverage this framework instead of another one: it is ready-to-go. How easy it is to set up and build on top of it is why developers should really take into consideration this framework, also because it is a completely dockerized application and could be easily developed on local machines or on the main cloud service providers. [5]

Let's take an in-depth look at the Cheshire Cat AI framework and its standout features, each carefully crafted to offer a unique, highly customizable, and scalable solution for developing advanced conversational agents.

Modular Architecture: Flexibility at Its Core

The core part of Cheshire Cat AI lies in its modular architecture, which stands as its most defining feature. The framework is built with flexibility in mind, allowing developers to easily extend and customize its basic functionality to reach the satisfaction of the needs of different use cases and applications. The modular structure allows developers to customize components for specific tasks rather than being constrained by a rigid, universal solution. This adaptability is of incredible value when building several and different conversational agents, as it allows for the integration of different modules like speech recognition, context management, or even third-party services such as CRM systems, all without having to overhaul the entire system. One of the most significant features of this modular structure is the plugin system. The plugin system allows developers to create domain-specific extensions, improving the framework's functionality for particular needs. For instance, if you are building an agent for legal document analysis, you can create a plugin that specifically handles legal jargon and integrates with legal databases. Once these plugins are created, they can be shared with the Cheshire Cat community through its public plugin store, making it easier for other developers to access and integrate those plugins into their own projects. This creates an active community ecosystem that promotes collaboration and knowledge sharing, improving the framework's overall usefulness.

User-Friendly Interface: Accessibility for All

Another key strength of Cheshire Cat AI framework is its user-friendly interface, this is designed not only for developers but also for end users who need to interact with the conversational agent. The interface provides an intuitive environment that simplifies user interaction with the agent. End users can easily provide input, receive responses, and even interact with complex media elements like images, PDFs, or other relevant content, when the underlying LLM fundational model accepts multimodal interaction. This capability enhances the user experience, allowing for more dynamic, multimodal interactions that go beyond simple text-based exchanges. For developers, the interface also provides a straightforward way to fine-tune and test the behavior of the agent, ensuring that the conversational flow aligns with the intended user experience. Even for those with limited technical expertise, the interface simplifies the process of building, deploying, and refining the conversational agent, making Cheshire Cat AI a versatile tool accessible to both technical and non-technical users.

Scalability: From Small Projects to Large-Scale Enterprise Use

The scalability of the Cheshire Cat AI framework is another important feature that makes it suitable for a wide range of applications, from small personal projects to large-scale enterprise solutions. The framework is designed to handle various levels of complexity, accommodating everything from a simple chatbot for individual use to a full-fledged AI-driven customer service system for large corporations.

Due to its modular and flexible architecture, Cheshire Cat AI can scale with relative friction as the complexity of the use case increases. Developers can start with small, simple conversational agents and gradually add more advanced functionalities, such as integrating with third-party APIs, adding sophisticated natural language processing capabilities, or incorporating multi-turn conversations. For enterprise use cases, the system is robust enough to support high volumes of interactions and large user bases, ensuring that it can handle the demands of large organizations without significant performance degradation.

The scalability aspect also extends to the backend, where Cheshire Cat AI can be deployed across a variety of infrastructures—whether that's on local machines, virtual private servers, or major cloud service providers. This ensures that the framework can grow alongside the needs of its users, with minimal friction as the scale of the project increases.

Ready-to-Go: Simplified Setup for Rapid Development

What truly sets Cheshire Cat AI apart from other conversational AI frameworks is its ready-to-go nature. Developers don't need to spend significant time configuring the environment or setting up complicated dependencies. With Cheshire Cat, the setup process is incredibly simple, making it a perfect choice for developers looking to quickly get started with building intelligent agents.

The framework is designed as a dockerized application, meaning it comes with pre-packaged containers that streamline deployment. This dockerized approach simplifies the setup process, allowing developers to run and test the framework easily on their local machines or cloud environments. Whether you're working on a personal project or deploying to a production environment, the containerized setup ensures that everything works seamlessly without worrying about compatibility issues or dependency conflicts. [18]

Moreover, the framework is built with cloud flexibility in mind. It supports integration with major cloud service providers like AWS, Google Cloud, and Azure, making it easy to deploy and scale conversational agents in the cloud. This feature is especially important for businesses and developers who need to scale their solutions to meet the growing demands of their user base, ensuring high availability and reliability at all times.

3.2.2 How the Cat works

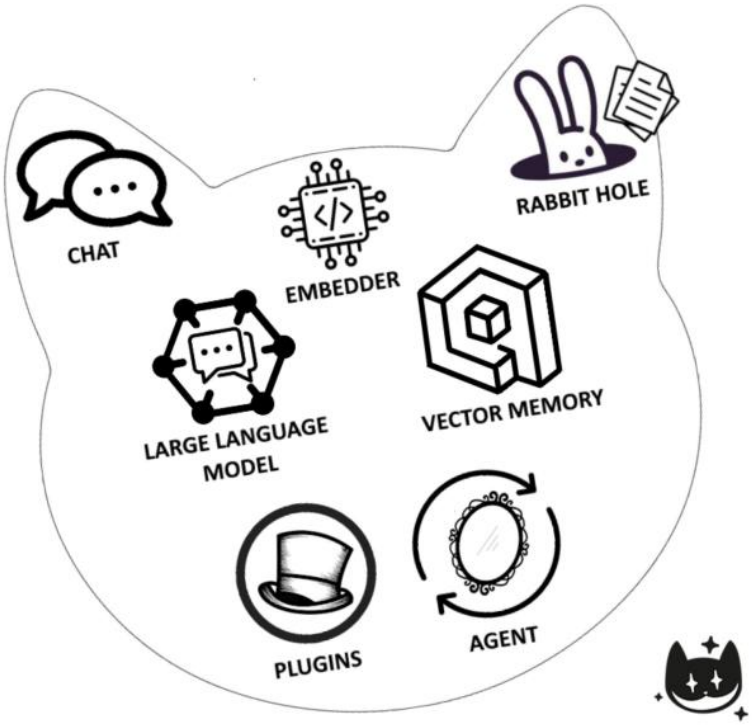

Figure 3.1: How the cat works

The Cheshire Cat is made of many pluggable components that make it fully customizable.

- **Chat**: This is the Graphical User Interface (GUI) component that allows the user to interact directly with the Cat. From the GUI, the admin user can also set the language model he/she wants the Cat to run, inserting, eventually, api keys and other configuration parameters.
- **Rabbit Hole**: This component handles the ingestion of documents. Files that are sent down the Rabbit Hole are split into chunks and saved in the Cat's declarative memory to be further retrieved in the conversation following the RAG technique previously treated.
- **Large Language Model (LLM)**: This is one of the core components of the Cheshire Cat framework. An LLM is a Deep Learning model, based on the transformer architecture, that's been trained on a huge volume of text data and can perform many types of language tasks. The model takes a text

string as input (e.g. the user's prompt) and provides a meaningful answer. The answer consistency and adequacy is enriched with the context of previous conversations and documents uploaded in the Cat's memory.

- **Embedder**: The embedder is another Deep Learning model similar to the LLM. Differently, it doesn't perform language tasks. The model takes a text string as input and encodes it in a numerical representation. This operation allows to represent textual data as vectors and perform geometrical operation on them. For instance, given an input, the embedder is used to retrieve similar sentences from the Cat's memory through the RAG technique.

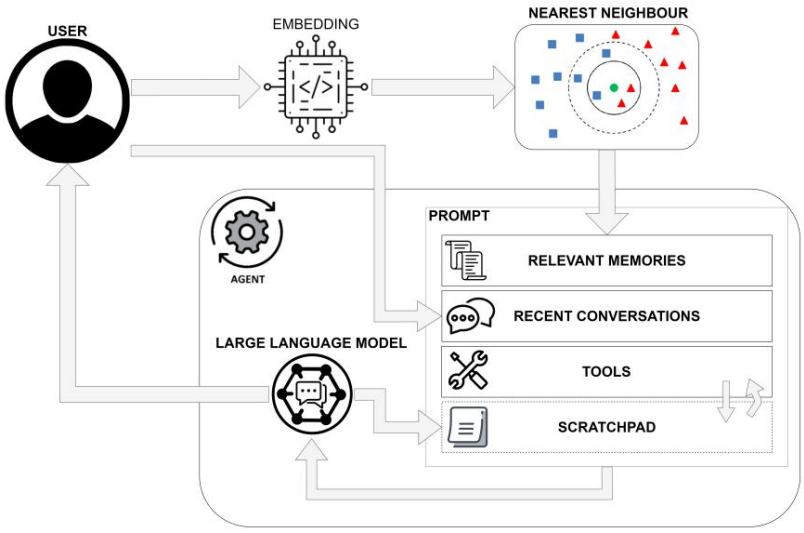

Figure 3.2: RAG in the Cheshire Cat

- **Vector Memory**: As a result of the Embedder encoding, we get a set of vectors that are used to store the Cat's memory in a vector database. Memories store not only the vector representation of the input, but also the time instant and personalized metadata to facilitate and enhance the information retrieval. The Cat embeds two types of vector memories, namely the episodic and declarative memories. The formers are the things the user said in the past; the latter the documents sent down the Rabbit hole.

- **Agent**: This is another core component of the Cheshire Cat framework. The agent orchestrates the calls that are made to the LLM. This component allows the Cat to decide which action to take according to the input the user provides. Possible actions range from holding the conversation to executing complex tasks, chaining predefined or custom tools.

- **Plugins**: These are developer-defined functions to extend the Cat's capabilities. Plugins are a set of tools and hooks that allow the Agent to achieve complex goals. This component let the Cat assists you with tailored needs.
- **Tools**: A Tool is a python function that can be chosen to be run directly from the Large Language Model. In other words: you declare a function, but the LLM decides when the function runs and what to pass as an input.
- **Hooks**: Hooks are callback functions that are called from the Cat at runtime. They allow you to change how the Cat internally works and be notified about framework events.

3.3 Models Employed in the System

In the development of this POC, Large Language Models (LLMs) and embedding models were essential components for enabling natural language understanding and semantic search functionalities within the Cheshire Cat framework.

3.3.1 LLM Selection

The conversational engine functionality relied on an external LLM accessed via API. The specific LLMs used during development included **OpenAI GPT-3.5**, depending on availability and API access at the time of testing.

While there was no exhaustive benchmark process to compare different LLMs for this project, the primary selection criterion was access—specifically, the availability of functioning API keys. These models were sufficient to demonstrate the core capabilities of the system, including response generation, prompt evaluation (toxicity/context), and multi-turn conversations.

3.3.2 Embedding Model Selection

Similarly, the embedding models used to generate vector representations of textual content for similarity-based retrieval were chosen based on availability. For most of the development process, the project used **OpenAI Ada v2**, depending on API accessibility and integration with the vector memory backend [19]

Although model selection in this context was primarily driven by practical access rather than performance benchmarking, the system architecture was designed to be model-agnostic. This modularity allows future iterations to easily substitute or compare other LLMs and embedders based on criteria such as cost, performance, latency, or alignment with specific use cases.

Moreover, the logging infrastructure developed (see *Conversation Logger Plugin*) was designed to store model metadata (e.g., LLM and embedder identifiers),

enabling future empirical evaluation and comparison across different model versions or providers.

3.4 Docker

Docker [18] is a widely used open-source platform that allows developers to create, deploy, operate, update, and manage software containers. These containers are portable, self-contained units that package application code together with all the necessary system libraries and dependencies, enabling the application to run consistently across various computing environments. By simplifying the development and delivery of distributed systems, containers have become a foundational element in modern software workflows, especially as enterprises move toward cloud-native applications and hybrid cloud infrastructures. Although it's possible to create containers using native features of Linux and other operating systems, Docker streamlines and accelerates this process, making containerization more accessible and efficient. Docker, like other container technologies such as Kubernetes, is essential in implementing modern development paradigms like microservices architecture. Unlike traditional monolithic software—where all functionalities are tightly integrated into a single application—microservices break down an application into smaller, independent services. These services can be developed, deployed, and scaled individually. With Docker, each microservice can be encapsulated in its own container, easing deployment processes and simplifying rapid iteration and scaling up capabilities. Containers leverage process isolation and virtualization features of the Linux kernel to run multiple application components on the same host operating system. This approach is similar in concept to how hypervisors allow multiple virtual machines (VMs) to share a single server's physical resources, such as CPU and memory. While containers provide similar benefits to VMs—like application isolation and scalability—they also introduce several distinct advantages:

- Lightweight design: Unlike virtual machines, containers do not require a full operating system or hypervisor. They carry only the essential OS processes and dependencies, making them smaller (typically measured in megabytes) and quicker to start, while optimizing hardware utilization.

- Enhanced developer productivity: Applications packaged in containers can be developed once and executed anywhere. Compared to virtual machines, containers are faster to provision, deploy, and restart.

- Resource efficiency: Containers admit more instances of an application to run on the same hardware compared to Virtual Machines, following in better utilization and potential cost savings in cloud environments.

Docker is utilized in a wide range of scenarios, reflecting its flexibility and alignment with modern software development practices:

- Cloud Migration: Docker enhances the efficiency and ease of migrating applications, data, and workloads between environments—whether transitioning from on-premises infrastructure to the cloud or moving between different cloud platforms—thanks to its high portability.

- Microservices Architecture: With the widespread adoption of microservices (used by over 85% of large enterprises, according to Statista), Docker provides a straightforward way to containerize each service individually. This facilitates independent deployment and scaling, removing the complexity of managing environment-specific configurations for each component.

- CI/CD Pipelines: Docker is well-suited for continuous integration and continuous delivery workflows. It provides a consistent runtime environment across all stages of development and deployment, minimizing the risk of discrepancies and failures during the release process.

- DevOps Enablement: The synergy between Docker and microservices supports agile and DevOps methodologies. This combination allows development teams to iterate rapidly, test new features efficiently, and respond to market demands with accelerated delivery cycles.

- Hybrid and Multicloud Environments: Docker's lightweight and portable nature enables seamless deployment of applications across diverse infrastructures. It supports hybrid cloud models that integrate on-premises systems with public, private, or edge cloud environments. Major cloud providers offer native Docker support, simplifying management across multicloud architectures.

- Containers as a Service (CaaS): CaaS platforms allow developers to orchestrate and scale Docker containers efficiently. This service model is widely offered by cloud providers alongside other paradigms such as IaaS and SaaS, enabling streamlined container management at scale.

- Artificial Intelligence and Machine Learning (AI/ML): Docker accelerates the development of AI and ML applications by offering a portable, consistent environment that simplifies experimentation and deployment. Docker Hub hosts a wide array of pre-built AI/ML container images, supporting faster development. Furthermore, in 2023, Docker introduced Docker AI, a feature that provides intelligent, context-aware suggestions when editing Dockerfiles and Docker Compose configurations, helping streamline workflows for AI/ML teams. [18]

To wrap things up, Docker can easily coordinate with third-party instruments, which help to easily deploy and manage Docker containers. Docker containers can easily be deployed into the cloud-based environment.

This platform was used because the Cheshire Cat image was shared by its community, making it easier for developers to deal with all dependencies etc. This also made it easier when deploying our version on AWS cloud.

3.5 MLFlow

MLflow [11] is a versatile, expandable, open-source platform specifically designed for managing workflows and artifacts across the machine learning lifecycle. It has built-in integrations with many popular machine learning libraries, but can be used with any library, algorithm, or deployment tool. It is designed to be extensible, so you can write plugins to support new workflows, libraries, and tools.

MLflow has five components:

- MLFlow Tracking: An API for logging parameters, code versions, metrics, model environment dependencies, and model artifacts when running your machine learning code. MLflow Tracking has a UI for reviewing and comparing runs and their results. This image from the MLflow Tracking UI shows a chart linking metrics (learning rate and momentum) to a loss metric
- MLflow Models: A model packaging format and suite of tools that let you easily deploy a trained model (from any ML library) for batch or real-time inference on platforms such as Docker, Apache Spark, Databricks, Azure ML and AWS SageMaker. This image shows MLflow Tracking UI's view of a run's detail and its MLflow model. You can see that the artifacts in the model directory include the model weights, files describing the model's environment and dependencies, and sample code for loading the model and inferencing with it
- MLflow Model Registry: A centralized model store, set of APIs, and UI focused on the approval, quality assurance, and deployment of an MLflow Model
- MLflow Projects: A standard format for packaging reusable data science code that can be run with different parameters to train models, visualize data, or perform any other data science task
- MLflow Recipes: Predefined templates for developing high-quality models for a variety of common tasks, including classification and regression

MLflow is used to manage the machine learning lifecycle from initial model development through deployment and beyond to sunsetting. It combines:

- Tracking ML experiments to record and compare model parameters, evaluate performance, and manage artifacts (MLflow Tracking)
- Packaging ML code in a reusable, reproducible form in order to share with other data scientists or transfer to production (MLflow Projects)
- Packaging and deploying models from a variety of ML libraries to a variety of model serving and inference platforms (MLflow Models)
- Collaboratively managing a central model store, including model versioning, stage transitions, and annotations (MLflow Registry)
- Accelerating iterative development with templates and reusable scripts for a variety of common modeling tasks (MLflow Recipes)

In my POC, I decided to employ the MLflow Tracking for LLMs, which enabled to record all the call flow from the user prompt until the agent response. This is a key component of LLMOps, observability. In fact, the main goal is to analyze and observe how the agent behaved, given a user prompt, and what actions were made in order to satisfy the user's request. Also, this is helpful for debugging reasons, following the function calls sequence and understand what made the agent choosing one path instead of others. [2, 7]

3.6 Toxicity Evaluation Plugin

In this paragraph I'm going to describe how the toxicity evaluation on user's prompts and final agent's response has been developed, what were the decisions that were made and what were the reasons behind these decisions.

As described in the previous chapter, prompt toxicity evaluation is a field of continuous growth and study. Considering the ultimate goal of this POC, which was to develop an application that would act as an accelerator for the adoption of solutions such as chatbots, the number one objective was to develop a toxicity evaluator tool that was fully and easily customizable by the end-user.

For this purpose, after assessing all possible approaches, the final decision was to adopt the "LLM-as-judge" paradigm. The "LLM-as-judge" paradigm, as previously anticipated, represents a more recent and sophisticated paradigm for toxicity detection. Rather than relying on classification models trained specifically for toxicity detection, this approach prompts a general-purpose LLM(e.g., GPT-4, Claude) to reason explicitly about whether an input is toxic or not. This technique can be also extended with a more precise and specific prompt preceding the user's query. In this prompt it is explained how to judge the toxicity of the prompt with few examples, parameters to take into consideration. Another step is to ask the

llm to judge not only the toxicity of the user's prompt but to give it a score from 0 to 1 of the level of toxicity.

Being LLMs extremely powerful at tasks that include "understanding" text, the decision was to exploit this incredible capability, combined with the technique own flexibility. This is achieved through the prompt priming technique that involves prefacing the content that need to be evaluated with a carefully designed instruction prompt, tailored to the specific evaluation goal.

To this end, the prompts from Opik [20] were employed and sligthtly customized:

```
"""
You are an AI content moderator tasked with analyzing text for potential issues.
Your job is to carefully examine the given text and determine if it contains any problematic content that violates common content policies.
Focus on detecting the following categories of issues:
    1. Violence or gore
    2. Hate speech or discrimination
    3. Sexual content or nudity
    4. Self-harm or suicide
    5. Illegal activities
    6. Personal information or privacy violations
    7. Spam or misleading content
    8. Harassment or bullying
    9. Extremism or radicalization
    10. Profanity or offensive language"

Analyze the text thoroughly and assign a safety score between 0 and 1, where:
    - 0.1 to 0.3 represents minor issues or borderline content
    - 0.4 to 0.6 represents moderate policy violations
    - 0.7 to 0.9 represents severe policy violations
    - 1.0 represents extremely unsafe content with multiple or critical violations

Provide a brief reason for your decision, explaining which category of issue was detected (if any) and why it's problematic.
Return your analysis in the following JSON format:
    {
    "VERDICT_KEY": [score between 0 and 1],
    "REASON_KEY": "Brief explanation of the verdict and score"
    }
Example response for problematic content:

    {
    "VERDICT_KEY": 0.8,
```

```
32      "REASON_KEY": "Text contains severe hate speech targeting a
        specific ethnic group, warranting a high safety score."
33      }
34  Example response for safe content:
35      {
36      "VERDICT_KEY": 0.0,
37      "REASON_KEY": "No content policy violations detected in the
        text. The content appears to be completely safe."
38      }
39  Example response for borderline content:
40      {
41      "VERDICT_KEY": 0.3,
42      "REASON_KEY": "Text contains mild profanity, but no severe
        violations. Low safety score assigned due to minimal concern."
43      }
44  Analyze the following text and provide your verdict, score, and
        reason in the specified JSON format:
45      {user_prompt}
46  """
```

As anticipated, this technique allows the end-user to highly customize which content should be considered as toxic or harmful and which not by simply adapting this prompt to the actual use case.

Furthermore, given the final score the LLM outputs after this request, the threshold over which the content is considered toxic is completely customizable, letting the end-user do some fine-tuning given his/her preferences on the specific use case.

To develop this plugin, I took advantage of some hook functions directly exposed by the Cheshire Cat framework. More specifically:

- agent_fast_reply: this hook allows the developer to anticipate all the agent activity and ideally save time if there's no need to propagate the user's prompt down to the llm. In this specific case, this hook function is responsible of calling the "evaluate_user_prompt" function which append the user's prompt to the evaluation template prompt described above and provide the result to the llm via the function "cat.llm(string)". The response json is then compared with the threshold set by the user and, if bigger, the cat returns a simple string "Your question doesn't respect our policies".

Given this solution, there are some downsides: first of all costs. In fact, completely demanding the toxicity evaluation to an external LLM accessed through its APIs, could easily increase the overall cost to deploy and release to the public this application.

Also, there could be some inconsistency of the toxicity evaluation score due to temperature settings of the LLM or different phrasing. The latter reason, though, is also applicable to the other solutions that could be employed for this objective. Finally, this solution could require more computational time with respect to local classifiers solution.

Overall, though, the advantages of high customizability, adaptability and flexibility completely outweigh the disadvantages, given this specific use case.

3.7 Conversation Logger Plugin

In the rapidly evolving landscape of large language models (LLMs) and their deployment within operational systems (commonly referred to as LLMOps), monitoring and observability have become critical components for ensuring reliability, performance, and continuous improvement. To this end, I developed a specialized plugin for the Cheshire Cat framework, designed explicitly to enhance transparency by recording detailed logs of conversations between users and the chatbot agent. This capability empowers developers with the ability to analyze historical interaction data offline, gaining insights into how the agent responds to various prompts under different configurations. [2, 7]

Such observability is indispensable for multiple reasons. Primarily, it facilitates systematic evaluation and iterative refinement of the deployed agents. By capturing final responses alongside key metadata, developers can identify potential weaknesses, biases, or inconsistencies in the agent's behavior. Furthermore, these records enable informed decision-making regarding the underlying components making it easier for developers to improve the agents' future versions and decide whether changing LLMs or Embedders currently used. That's because it could be possible, after further studies, a different LLM from the current used one could be better at handling the specific application tasks or, in alternative, a new better model could be released in the meantime. Given the rapid pace of advancements in language modeling, a different LLM or embedder may, upon evaluation, prove more effective for the target application. Additionally, the availability of empirical data is essential when benchmarking newly released models, ensuring upgrades are grounded in measurable improvements rather than speculative assumptions.

To realize this goal, the plugin needed a robust yet adaptable data storage solution capable of accommodating evolving requirements. Initially, the primary data to capture was a simple tuple comprising the user's query and the agent's final response. However, through iterative analysis and requirements gathering, this schema was expanded to include additional parameters that provide richer context for each interaction. In fact, initially the tuple consisted of the pair <user's query, agent's answer>.

After further analysis and study, though, the original tuple mentioned before was extended to a list of parameters each interaction with the agent has: <user_id, user's query, agent's answer, llm_used, embedder_used, toxicity_evaluation_score, timestamp>. This list could be easily extended even more in the future, given a specific use case or need of developers.

Considering the requirements of flexibility, scalability and ease of extension, the idea of employing a traditional relational database was immediately discarded. This because relational databases require rigid schemas which was quite a strict constraint for this application.

The choice, then, fell on the non relational database type as this kind of database allows developers to easily extend the fields of data that are saved, without having to deal with all the problems that relational databases bring with them, such as schema and between-tables consistency. Again, here comes in handy the extreme ease of developing new features for the Cheshire Cat framework: a new simple plugin, that uses the hooks provided by the framework, allowed to develop this feature without any particular difficulty. In fact, this framework makes it extremely easy to intercede in the natural flow of function calls of the agent and customize its behaviour.

At the art of this architecture is the concept of hooks, predefined interception points within the agent's execution flow that allow developers to inject custom logic without altering core components. In particular this use case implied the adoption of the "before_cat_sends_message" hook function, which enables interception and customization of the agent's message dispatch process. This kind of integration required minimal intrusion into the core system, highlighting, once again, the framework's extensibility and ease of feature augmentation. Regarding future versions of the PoC, the plugin architecture allows for numerous potential enhancements. Future work may include integrating real-time dashboards for monitoring conversational trends, incorporating user feedback loops for supervised learning, or expanding toxicity evaluation to include other fairness and safety metrics.

In summary, this plugin represents a key advancement in the Cheshire Cat framework's capabilities, enabling detailed tracking and analysis of user-agent interactions. By combining a flexible NoSQL data model with the framework's extensible hook system, it delivers a robust, scalable solution to the challenges of monitoring and evolving AI-driven conversational systems.

3.8 Context Precision Evaluation

After working on the toxicity evaluation task, I decided to explore the agent's answer evaluation topic. More in depth, the result I wanted to achieve was to build a plugin that was able to detect whether the agent's final response took

into consideration only the context provided via RAG. This is because LLMs are known to have incredible skills but, often, are mistaken for oracles of knowledge by the majority of the population. On the contrary, they should be treated for what they effectively are, amazing language engines that have no reasoning capability and no logic comprehesion skills.One of the core problems is that LLMs operate based on patterns learned from vast but static and unverifiable training data. This knowledge is frozen at a particular point in time and often lacks transparency about its origins. While this internal corpus allows LLMs to generate fluent and coherent text, it also enables them to synthesize plausible-sounding yet false or misleading information. As a result, depending on an LLM's internal knowledge without explicit grounding in external, curated context undermines both the accuracy and accountability of AI-driven systems.

Ensuring that an LLM-based agent relies solely on externally retrieved context is not merely a technical preference — it addresses significant real-world risks tied to misinformation and misplaced trust in AI systems. When exposing LLM-based applications to the public, indeed, accuracy of answers given is crucial for multiple reasons. It is then fundamental to provide deterministic information inside their responses, without relying on the LLM internal knowledge. To do so, I decided to exploit again the LLM-as-judge technique, providing, again, a predefined prompt combined with the final agent's response to the LLM.

In this case I used the following prompt [21]:

```
"""
Guidelines:
    1. The OUTPUT must not introduce new information beyond what's provided in the CONTEXT
    2. The OUTPUT must not contradict any information given in the CONTEXT
    3. Ignore the INPUT when evaluating faithfulness; it's provided for context only
    4. Consider partial hallucinations where some information is correct but other parts are not
    5. Pay close attention to the subject of statements. Ensure that attributes, actions, or dates are correctly associated with the right entities (e.g., a person vs. a TV show they star in)
    6. Be vigilant for subtle misattributions or conflations of information, even if the date or other details are correct
    7. Check that the OUTPUT doesn't oversimplify or generalize information in a way that changes its meaning or accuracy

Verdict options
    - "FACTUAL_VERDICT": The OUTPUT is entirely based onto the CONTEXT
    - "HALLUCINATION_VERDICT": The OUTPUT contains hallucinations or unfaithful information
```

```
14
15 INPUT (for context only, not to be used for faithfulness
       evaluation):
16     f"{cat.working_memory.user_message_json.text}
17     CONTEXT:
18     f"{[x[0].page_content for x in cat.working_memory.
       declarative_memories]}
19
20 OUTPUT:
21     f"{message.content}
22
23 Provide your verdict in JSON format:
24     {
25         "VERDICT_KEY": your verdict,
26         "REASON_KEY": your brief explanation
27     }
28 """
```

Chapter 4

Solution Cloud Distribution

4.1 Introduction

Following the development of a functional Proof of Concept (PoC) in a local environment, the next logical step in the project was to design and implement a cloud-based deployment strategy. The aim was to transition the application to a scalable and production-grade infrastructure, capable of supporting continuous integration, reproducibility, and maintainability in line with modern DevOps and LLMOps practices. [2, 7]

This chapter presents the decisions made to industrialize the system by distributing it on the Amazon Web Services (AWS) cloud. It explores the architectural framework, deployment strategies, automation pipelines, and observability mechanisms that were implemented to achieve a cloud-native, modular, and maintainable solution.

4.2 Industrialization Framework

The core of the cloud deployment strategy was to design a robust industrialization framework that would support automation, scalability, and adaptability for future deployments. This required a shift from manual, script-based configurations to a fully automated infrastructure-as-a-code (IaaC) model. The system was containerized using Docker to ensure environment parity and transportability, while Terraform was adopted to manage infrastructure provisioning in a modular and declarative fashion. [4]

This framework was designed to be reusable and parameterizable, allowing developers to target different environments (e.g. staging, production) with minimal changes. The result was a deployment strategy that not only met current project needs but could also evolve alongside future requirements, such as new plugins,

different LLM backends, or additional monitoring capabilities.

4.3 Infrastructure-as-a-Code vs Manual Deployment

At the beginning of the PoC project, deploying the application manually would have been feasible for prototyping purposes. Manual deployment methods, though, are intrinsically limited: they are non-reproducible, time-consuming, error-prone, and difficult to maintain over time. These drawbacks become critical barriers in any real-world deployment where reproducibility, reliability, traceability, and scalability are essential.

Infrastructure-as-a-Code (IaaC) handles these limitations by allowing infrastructures to be defined declaratively using code, tracked via version control systems, and deployed automatically through pipelines. Among the various IaaC tools available, Terraform emerged as the most appropriate choice due to its platform agnosticism, strong AWS integration, and modular design philosophy. By adopting IaaC, every component of the infrastructure—from network configuration to IAM policies—became versioned, auditable, and replicable, allowing developers to reproduce or roll back entire environments in minutes.

4.4 Terraform

Managing modern cloud infrastructure demands consistency, repeatability, and a high degree of automation — qualities that traditional manual provisioning methods often fail to deliver, especially as systems grow in complexity and scale. For this reason, Infrastructure-as-a-Code (IaaC) frameworks have become essential tools in the design and operation of reliable, scalable, and maintainable cloud environments. In this project, Terraform was chosen as the primary IaaC tool to define, provision, and manage all infrastructure components required for the deployment of the conversational AI application PoC on AWS.

Terraform is an open-source, declarative IaaC framework that allows infrastructure on cloud service providers to be described as code using a human-readable configuration language. This approach fundamentally transforms the way infrastructure is treated: instead of manually configuring each component through the AWS Management Console or ad-hoc scripts, every resource, from virtual networks to compute instances, is defined in version-controlled configuration files. This makes infrastructure fully reproducible and transparent, supporting best practices of modern DevOps and enabling seamless collaboration among team members.

A major benefit of using Terraform is its modular design philosophy. In this

project, the overall infrastructure was divided into reusable modules, each encapsulating a specific part of the architecture, such as networking, storage, compute resources, or security policies. This modularity promotes maintainability and scalability: new environments (for example, staging, development, or production) can be spun up quickly by reusing and customizing these modules through parameter files like .variables.tf files and setting these variables through the command line when deploying the infrastructure. This ensures that configurations remain consistent across environments while allowing for environment-specific customizations when needed.

The main resources managed via Terraform in this deployment included core networking elements such as Virtual Private Clouds (VPCs), subnets, security groups, and application load balancers. Compute resources, including ECS clusters and task definitions for containerized services, were also defined through Terraform scripts. In addition, storage solutions like Amazon S3, Elastic File System (EFS), and DynamoDB were provisioned and configured, along with the necessary Identity and Access Management (IAM) roles and policies to enforce least-privilege access.

Another notable advantage of Terraform is its support for advanced deployment scenarios. For example, the canary deployment strategy described in the following section was facilitated by defining weighted target groups and automated scaling rules directly in the Terraform configurations. This level of automation helps ensure that the deployment pipeline remains consistent and resilient, reducing the risk of manual misconfiguration and unintended downtime. [4]

Despite these significant advantages, adopting an IaaC approach with Terraform also comes with challenges that must be acknowledged. Writing and maintaining accurate configuration files requires a solid understanding of both the IaaC syntax and the cloud provider's underlying services. Errors in the configuration files can propagate to the entire environment, potentially causing unintended downtime or security issues if not caught during code reviews or validation checks.

Additionally, while IaaC frameworks like Terraform simplify the process of provisioning infrastructure, they introduce new complexity related to state management. Terraform maintains a state file to track the current status of all resources under management. This state must be carefully stored, secured, and synchronized in team settings to prevent drift or conflicts. If multiple team members work on the same infrastructure simultaneously without proper state locking, inconsistencies can arise.

Another practical challenge is the learning curve and the initial setup effort. Compared to manual provisioning or simple cloud management scripts, Terraform requires upfront investment in designing modular configurations, testing them thoroughly, and integrating them into CI/CD pipelines. However, this investment pays dividends over the long term by drastically reducing manual overhead, configuration drift, and deployment inconsistencies.

Overall, the adoption of Terraform in this project was crucial to achieving a robust, auditable, and reproducible infrastructure. It allowed the deployment process to be automated from end to end, reduced the likelihood of human error, and supported rapid iteration and safe experimentation — all core requirements for deploying scalable AI systems on the cloud. By embedding infrastructure definitions alongside application code in version control, any changes to the system's architecture are transparent, reviewable, and easily reversible if necessary. This aligns perfectly with modern DevOps and LLMOps principles, positioning the application for sustainable growth and maintenance over time.

4.5 AWS Services and Tools Used

To realize the deployment of the Proof of Concept conversational AI application on AWS cloud, several key AWS services and containerization technologies were selected to ensure scalability, security, and maintainability. This section briefly describes the main services and tools that took part in the foundation of the cloud infrastructure and deployment pipeline.

4.5.1 Amazon Elastic Container Registry (ECR)

Amazon Elastic Container Registry (ECR) is a fully managed container image registry service provided by AWS. It allows developers to store, manage, and deploy Docker container images securely and at scale. ECR supports private and public repositories and provides features such as image versioning, automated lifecycle policies for cleaning up unused images, and built-in image vulnerability scanning to improve security. In this project, ECR was used to store custom-built Docker images for the conversational agent and related services, ensuring a reliable and centralized location for version-controlled container artifacts.

4.5.2 Amazon Simple Storage Service (S3)

Amazon S3 (Simple Storage Service) is AWS's highly scalable and durable object storage service. It enables the storage of virtually unlimited amounts of data as discrete objects organized into logical containers called "buckets." S3 provides high availability, strong security controls, and flexible data lifecycle management policies that allow data to move automatically between storage classes (e.g., Standard, Infrequent Access, or Glacier) based on access needs and cost optimization. Within this PoC, S3 was used for storing configuration files and logging artifacts that needed to be accessible by the deployed services. Its seamless integration with other AWS services, including IAM and ECS, made it a natural choice for managing static data and artifacts.

4.5.3 AWS Identity and Access Management (IAM)

AWS Identity and Access Management (IAM) is the security service used to manage and control access to AWS resources in a centralized and secure manner. With IAM, it is possible to create users, groups, and roles with finely-grained permissions, applying the principle of least privilege to protect sensitive resources. IAM policies define which users or services can perform specific actions on resources such as S3 buckets, ECR repositories, or ECS clusters. For this project, IAM ensured that only authorized components and users could interact with the infrastructure, adding an essential layer of security to the deployment pipeline. Multi-factor authentication (MFA) and activity logging further strengthened the security posture. [22]

4.5.4 Docker

Docker is an open-source platform for building, packaging, and running applications inside lightweight, isolated containers. By encapsulating the application code together with its dependencies, Docker ensures consistent behavior across different environments — from local development machines to production servers in the cloud. Docker images are defined using Dockerfiles and can be shared via repositories like ECR or Docker Hub. In this PoC, Docker was used extensively to package the conversational agent, plugins, and related services into self-contained images that could be easily deployed on AWS Elastic Container Service (ECS). This container-based approach aligned with modern microservices architecture principles, supporting modularity, scalability, and ease of maintenance. [18]

4.5.5 Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless NoSQL database service provided by AWS, designed for high performance and automatic scaling with minimal operational overhead. Unlike traditional relational databases, DynamoDB uses a key-value and document data model that offers great flexibility for handling dynamic, semi-structured, or unstructured data. [23]

One of the main strengths of DynamoDB is its ability to provide single-digit millisecond response times at any scale, making it well suited for applications that require real-time data access and high throughput. As a fully managed service, it automatically handles hardware provisioning, replication across multiple Availability Zones, and continuous backup and restore, significantly reducing the administrative burden on developers.

In this project, DynamoDB was chosen to store conversational logs and metadata generated by the chatbot application. Each interaction between the user and the agent — including prompts, final responses, timestamps, toxicity scores, and foundational model and embedders information — is recorded in DynamoDB

tables. This flexible schema design allows developers to easily expand or adjust the data structure as new features are added, without the rigid constraints typical of relational databases. By integrating DynamoDB into the deployment, the system benefits from highly available, durable storage that can handle growing volumes of data while maintaining fast query performance. This makes it a critical component for supporting the observability, traceability, and continuous improvement aspects required in modern LLMOps pipelines. [7]

4.5.6 Amazon Elastic Container Service (ECS)

Amazon Elastic Container Service (ECS) is a fully managed container orchestration service provided by AWS that enables the deployment, scaling, and management of containerized applications. ECS eliminates the operational complexity of running containers at scale by handling tasks such as container scheduling, cluster management, and service discovery. [24]

At its core, ECS allows developers to run Docker containers on a managed cluster of Amazon EC2 instances or using AWS Fargate, a serverless compute engine that removes the need to provision and manage virtual machines directly. Tasks and services in ECS are defined through task definitions, which specify container configurations, resource requirements, networking modes, and environment variables, providing fine-grained control over how containers run and interact.

During the development phase of this project, ECS represented the main orchestration platform for deploying the containerized components of the conversational AI application. The conversational agent with the developed plugins was packaged in a Docker container and deployed as ECS services. ECS made it possible to run multiple instances of these containers in a controlled and scalable manner, with integration to an Application Load Balancer (ALB) to distribute traffic efficiently.

Moreover, ECS integrates seamlessly with other AWS services like Elastic Container Registry (ECR) for pulling container images, IAM for securing tasks, CloudWatch for monitoring, and Auto Scaling for dynamically adjusting the number of running containers based on traffic and resource usage. This tight integration, combined with its mature ecosystem, made ECS an ideal choice for managing the lifecycle of containers in a robust, cost-effective, and highly automated deployment pipeline.

4.5.7 Amazon Elastic File System (EFS)

Amazon Elastic File System (EFS) is a fully managed, scalable, and elastic Network File System (NFS) offered by AWS, designed to provide simple, serverless, and highly available file storage that can be shared across multiple compute resources. Unlike block or object storage, EFS offers a familiar file system interface and file

semantics, making it ideal for applications that require shared, persistent file storage with concurrent access from multiple containers, virtual machines, or serverless functions. [25]

One of the key strengths of EFS is its elasticity: storage capacity automatically grows and shrinks as files are added or removed, eliminating the need for manual provisioning or capacity planning. EFS is designed for high durability and availability, replicating data across multiple Availability Zones within an AWS Region. Integrated encryption at rest and in transit ensures that files remain secure, while IAM policies and network controls govern which resources and users have access.

4.5.8 Amazon SageMaker

Amazon SageMaker is a fully managed machine learning (ML) service provided by AWS that enables developers and data scientists to build, train, and deploy ML models at scale, without the heavy lifting typically associated with setting up and maintaining dedicated ML infrastructure. SageMaker provides an integrated set of tools for the entire machine learning lifecycle — from data preparation and experimentation to training, tuning, hosting, and monitoring models in production. [26]

A particularly valuable feature of SageMaker is its ability to act as a robust, managed environment for deploying MLflow servers and model registries. MLflow is an open-source platform for managing ML experiments, model tracking, and lifecycle management. By integrating MLflow with SageMaker, teams can register trained models, manage model versions, and deploy them as scalable, secure endpoints for inference, all within a unified, AWS-native ecosystem.

In this project, SageMaker was exploited mainly for its capability of deploying a fully managed version of the Tracking Server of MLFlow. Once models are tracked and stored using an MLflow Tracking Server, SageMaker can directly deploy them as managed endpoints with autoscaling and built-in monitoring, which significantly simplifies operational workflows. This integration also enables automated A/B testing or canary deployments for new models, aligning with modern MLOps practices. [2]

In addition, SageMaker offers features like built-in experiment tracking, model explainability, and integration with other AWS services such as S3, IAM, and CloudWatch for secure storage, access control, and detailed observability. This makes SageMaker not only a platform for model training but also a powerful production-grade solution for serving ML models with high availability and performance guarantees. In summary, SageMaker complements the MLflow tracking approach by providing a fully managed, scalable backend for deploying and maintaining machine learning models, which aligns with the long-term goal of automating and industrializing the entire LLMOps pipeline for the conversational AI application.

4.5.9 Amazon Bedrock

Amazon Bedrock is a fully managed service by AWS that enables developers to build and scale generative AI applications using a selection of high-performing foundation models (FMs) from leading AI providers, without the need to manage underlying infrastructure or fine-tune models directly. Bedrock provides access to models from providers such as Anthropic (Claude), AI21 Labs (Jurassic), Meta (Llama), Cohere, and Amazon's own Titan models — all through a unified API and serverless interface. [27]

One of the primary advantages of using Amazon Bedrock lies in its simplicity of integration and strong focus on enterprise readiness. It abstracts away the complexity of provisioning GPU resources, configuring model endpoints, or managing scaling policies, thereby allowing developers to focus on product logic, security, and performance. Bedrock also provides seamless integration with other AWS services such as IAM for authentication and access control, CloudWatch for observability, and KMS for encryption — ensuring compliance with industry-grade security and governance standards.

In the context of this project, Amazon Bedrock was explored as a plug-and-play backend for providing foundational large language models within the conversational AI platform built using the Cheshire Cat AI framework. Thanks to the modular and extensible nature of the Cheshire Cat, it is possible to integrate new large language models by simply developing custom plugins that abstract the logic required to call external APIs. Using a dedicated plugin, Bedrock models can be made visible into the Cheshire Cat user interface and be invoked in response to user prompts, consequently substituting or augmenting third-party APIs like OpenAI with AWS-native, managed solutions.

This integration approach offers multiple benefits: first, it enables easier control over infrastructure cost and access permissions using native AWS tooling. Second, it allows developers to switch between different foundation models or providers without needing to change the application's internal logic, that is one of the main goals of the Cheshire Cat AI framework itself. Third, it positions the platform to benefit from ongoing advancements in the Bedrock model catalog, as AWS continues to add support for new models and fine-tuning capabilities.

By leveraging Amazon Bedrock, the platform gains access to production-ready foundation models through a scalable and secure channel, while maintaining flexibility and modularity via the plugin system provided by Cheshire Cat. This opens the door for future enhancements such as model comparison, dynamic model routing, or domain-specific prompt adaptation — all within a fully managed AWS ecosystem.

Together, these services and tools provided the backbone for a robust, cloud-native deployment, allowing a flexible and secure framework for the development, distribution, and operation of the conversational AI Proof of Concept.

4.6 Canary Deployment

Deploying new versions of software in a production environment always brings to life potential risks, especially when dealing with complex systems that serve real users non stop. Any update, whether it introduces new features, bug fixes, or performance improvements, has the ability to potentially, even though unintentionally, put down services, introduce hidden bugs, or degrade user experience. To handle this challenge and to balance innovation with reliability, a canary deployment strategy is often taken into consideration by developers for their projects. A canary deployment is a progressive delivery approach that allows a new version of an application to be introduced gradually rather than all at once. Instead of replacing the entire live system with the updated version in a single operation (an approach often known as a “big bang” deployment) a canary deployment begins by routing only a small, controlled portion of the overall user traffic to the newer version, while the majority of requests continue to be handled by the stable, previously tested version.

This limited exposure of the newer version acts as an early warning system, providing real-time insight into how the new version behaves under actual production conditions and user interactions, factors that can be hardly replicated in development or staging phases. Through continuous tracking of essential performance metrics including response times, error rates, resource consumption, and other vital business indicators, teams can rapidly identify anomalies, performance degradations, or unexpected consequences that internal testing may have missed. The main advantage of this strategy is its ability to minimize the overall damages due to potential failures of the application. If an issue is detected, the deployment can either be paused or the configuration can be rolled back entirely with minimal disruption, affecting only the small percentage of users initially exposed. This minimizes the risk of big outages, improves system stability, and builds user trust by ensuring that new updates do not unexpectedly break core functionality or provoke long periods of down time. It also supports safer experimentation, allowing teams to validate new features or changes with live traffic data before committing fully.

Nonetheless, canary deployments do not come without trade-offs. They require a more sophisticated operational setup compared to traditional deployments. Traffic routing must be configurable and precise, often needing the use of advanced load balancing features. Robust observability must be in place to track the health and performance of both the new and existing versions in near real-time. Automated rollback procedures must also be carefully designed to respond immediately if thresholds for acceptable performance are crossed. Another challenge is consistency: users who interact with both the old and new versions during the canary phase might encounter inconsistent behavior or data states if backward compatibility is not thoroughly ensured. This is partially minimized by setting so-called "sticky

sessions" for longer periods of time. In multi-service architectures, coordinating canary releases across interdependent services can further complicate deployment pipelines.

Despite these complexities, the benefits of improved reliability, faster detection of hidden issues, and reduced risk make canary deployments a standard practice for any organization aiming to adopt continuous delivery pipelines at scale.

4.7 Deployment approach

In this project, the deployment of the Proof of Concept (PoC) was structured around modularity, reproducibility, and reliability — in alignment with both DevOps and LLMOps best practices. The entire infrastructure and application layer were designed to be provisioned, deployed, and updated through automated pipelines, avoiding manual interventions wherever possible.

At the heart of the deployment was a multi-step, automated release workflow that transitioned the application from local development to production-ready cloud deployment on AWS. This approach integrated key AWS services and Infrastructure-as-Code tooling to support the following objectives:

- Repeatable environment setup through Terraform modules
- Containerized service deployment using ECS and Docker
- Versioned and secure image management via ECR
- Isolated, monitored traffic routing through ALB and canary rollout strategies
- Secure and observable infrastructure with IAM, CloudWatch, and logging integration

4.7.1 Deployment pipeline stages

The end-to-end deployment approach can be broken down into the following logical stages:

1. **Infrastructure Provisioning:** Using Terraform, all cloud resources — including networking (VPCs, subnets), IAM roles, security groups, ECS clusters, EFS mounts, S3 buckets, and DynamoDB tables — were created declaratively. Modules and variables.tf made it possible to switch between environments like staging and production with minimal code changes.

2. **Containerization and Image Management:** All services (conversational agent, plugins) were packaged using Docker and pushed to a secure, version-controlled Amazon ECR repository. This ensured consistency across environments and enabled traceability across deployments.

3. **Service Orchestration on ECS:** ECS task definitions were used to run Docker containers with specific configurations. The ECS service maintained availability and health of these tasks, while autoscaling policies ensured the cluster could adapt to traffic demand.

4. **Traffic Routing and Canary Deployments:** Application Load Balancer (ALB) was configured with multiple target groups to support weighted traffic routing. A canary deployment strategy was implemented where a small percentage of traffic is initially routed to a new ECS task version. Based on the behaviour of the newest version of the application, traffic could incrementally be increased until full transition is complete or reverted upon detecting performance degradation.

5. **Observability and Monitoring:** All components were integrated with AWS CloudWatch for centralized logging, metrics aggregation, and alerting. Additional metadata — such as toxicity scores, LLM provider responses, or function call traces — were stored in DynamoDB and MLflow server deployed via AWS Sagemaker Studio, enabling detailed post-hoc analysis and debugging.

6. **Security and Access Control:** Role-based access via IAM made it sure that each service or user had the minimum required permissions. Docker containers executed with pre-scoped IAM task roles, while EFS, S3, and DynamoDB access was tightly regulated using IAM policies.

The adopted approach would bring to the PoC many advantages:

- **Reproducibility:** Environments could be fully recreated from code with consistent configurations and artifacts.

- **Safety and Reliability:** Canary deployments, combined with robust observability, reduces the risk of complete outages.

- **Scalability:** ECS autoscaling and serverless options (like EFS and DynamoDB) ensured the system could handle increasing load with minimal maintenance overhead.

- **Modularity:** The architecture was plugin-based (via Cheshire Cat), enabling fast integration of new LLMs, inference backends (e.g., Bedrock), or observability agents.

- **Future-readiness:** The infrastructure design supports CI/CD integration and continuous retraining pipelines, aligning with LLMOps principles such as versioned model serving, continuous training, and behavioral feedback loops.

4.8 Conclusions

The migration from a local development environment to a robust and scalable cloud infrastructure involved a careful orchestration of modern engineering practices and cloud-native tools. By leveraging AWS services, Docker and Terraform the application evolved into a modular, maintainable, and production-ready system.

The adoption of canary deployments ensured safe and controlled releases. Observability tools such as MLFlow and DynamoDB facilitated in-depth analysis and debugging, while the modular architecture allows for easy integration of future components like SageMaker and Bedrock. In conclusion, the chosen deployment strategy not only fulfilled the current requirements of the project but also laid a solid foundation for future enhancements and enterprise-scale adoption.

Chapter 5

Conclusions and future works

5.1 General Overview and Achievements

This thesis presented a comprehensive study and practical implementation of a modular, scalable, and customizable LLM-based application designed to facilitate the adoption of conversational agents in production environments. The work was positioned within the evolving domains of MLOps and LLMOps, addressing critical issues of deployment, observability, content moderation, and architectural extensibility.

The project was conducted with the development of a Proof of Concept (PoC) that integrates modern DevOps practices (in particular, Infrastructure as Code, and containerization) with the latest LLMOps strategies, including prompt tracking, response evaluation, and content safety controls. The goal was not only to demonstrate the technical feasibility of such a platform but also to ensure it is future-proof, maintainable, and adaptable to a variety of business needs and regulatory environments, making it suitable as an accelerator of adoption. Among the key results achieved:

- Modular chatbot architecture: Leveraging the open-source Cheshire Cat AI framework, a highly customizable and plugin-oriented conversational agent was developed. This modularity makes it suitable for a wide range of real-world use cases across different industries.

- Toxicity Evaluation Plugin: A plugin was designed and implemented using the "LLM-as-a-Judge" paradigm. It empowers the agent to dynamically assess user input and generated content for toxicity using contextual, prompt-engineered reasoning, which is both flexible and customizable by non-technical end users.

- Conversation Logging Plugin: A robust logging plugin was integrated into the system, enabling full traceability of user interactions, model configurations, and response metadata. The use of Amazon DynamoDB ensured a scalable NoSQL backend that could easily evolve over time.
- Context Precision Evaluation Module: Recognizing the risk of hallucination in LLMs, another plugin was developed to evaluate whether the agent's responses stayed faithful to retrieved context (e.g., documents ingested via RAG). This enhances response reliability and mitigates risks related to misinformation or fabricated outputs.
- Cloud-Native Deployment Pipeline: The entire system was containerized using Docker and deployed on AWS using Terraform, enabling reproducible, scalable, and automated infrastructure setup. A canary deployment strategy was implemented to support safe iterative rollouts in production environments.

This set of results demonstrates that an LLM-based application can be designed in a production-ready, highly observable, and ethically aligned manner by following current best practices in software engineering and AI operations.

5.2 Reflections and Limitations

Despite the positive results of the project, several limitations and trade-offs were identified that may be brought to attention for future development efforts and the broader adoption of the platform. These considerations are essential for guiding next steps and informing potential users of the system's current constraints.

- Computational Overhead and Financial Cost: Relying on high-end third-party LLMs (such as GPT-4 or Claude) via external APIs introduces significant operational costs. This is particularly relevant in scenarios involving frequent LLM calls—for example, toxicity scoring and hallucination evaluation—which must be executed for every interaction. Consequently, achieving cost-efficiency at scale remains a key challenge.
- System Latency and Performance Bottlenecks: The system architecture involves multiple sequential operations, including LLM-based toxicity checks, logging mechanisms, and context validation steps. While the cumulative latency is tolerable during prototyping and internal testing, such delays may degrade the end-user experience in real-time production environments, especially under high traffic loads.
- Inconsistency and Non-Determinism in Evaluations: LLMs are inherently probabilistic, and their outputs can vary across sessions even when given the

same prompt—especially when configured with higher temperature settings. This variability can hinder the repeatability of evaluations, making it difficult to maintain consistent toxicity scores or hallucination judgments across identical inputs.

- Privacy Risks and Compliance Challenges: Assigning sensitive evaluations to external services (e.g. OpenAI APIs) raises critical concerns about data privacy, user confidentiality, and regulatory compliance. In jurisdictions governed by strict data protection laws, such as the EU AI Act or GDPR, this architecture may raise risks unless mitigated by using self-hosted or private alternatives (from which also time efficiency could benefit)

- Reliance on a Single Ecosystem (Cheshire Cat AI): Although the Cheshire Cat framework offers extensive modularity and ease of customization, its relatively limited adoption within the broader open-source community may affect long-term sustainability. The framework's future depends on active community support, continued maintenance, and contributions to ensure stability and feature growth over time.

5.3 Future Works

Based on the solid foundation from this thesis, several exciting directions for future research and development are suggested:

1. Local Deployment of LLMs: To mitigate both privacy and cost issues, future iterations should explore deploying open-source LLMs (e.g., LLaMA 3) on local or private GPU infrastructure. This shift would solve the issue related to the reliance on external APIs, offer better response time control, and ensure full data confidentiality according to data protection laws.

2. Hybrid Toxicity Detection Architecture: A hybrid system could combine lightweight, locally hosted classifiers (e.g., Detoxify, HateBERT) with LLM-as-a-Judge scoring. This layered approach would first perform a fast initial filter and only invoke LLMs for niche-related or borderline cases, optimizing both costs and performance.

3. Multi-Faceted Evaluation Metrics: The hallucination detection plugin could be extended to include additional factuality checks, bias measurements, and fairness audits. Incorporating explainability frameworks such as LIME or SHAP could further enhance the interpretability of outputs.

4. User Feedback Loop Integration: Future versions of the PoC could include explicit mechanisms for users to rate responses (as it is proposed in many

other solutions publicly available nowadays). This feedback can be logged, analyzed, and used to fine-tune model behavior or plugin thresholds over time, improving alignment with user expectations and safety guidelines.

5. Real-Time Monitoring Dashboard: A live dashboard, maybe implemented leveraging tools like Grafana or Streamlit, could be introduced to visualize toxicity levels, model performance, and usage analytics in real time. This would offer valuable insights to system operators and product owners.

6. Regulatory Compliance Framework: Given the rising importance of AI governance, future iterations should include tools to monitor compliance with standards such as the EU AI Act, ISO/IEC 42001, and GDPR. This could involve automated redaction, consent tracking, and audit logging.

7. Benchmark Suite for LLM Agent Evaluation: To keep memory of and compare different LLMs used as backends, a standardized benchmark suite can be developed that evaluates: hallucination rate, toxicity accuracy, response latency, cost per token, and context adherence. This would ease data-driven LLM selection and bring up the overall levels of performance of the application

5.4 Final Remarks

This thesis aimed not only to contribute to operationalizing LLMs but also to produce a working, extensible PoC that addresses the important real-world concerns of content moderation, deployment scalability, and observability. The convergence of LLMOps and cloud-native engineering is still in its early stages, and this work serves as a straightforward, open, and adaptable reference for future projects.

In conclusion, while large language models are becoming more and more powerful, their real value emerges only when they are embedded within carefully designed ecosystems that ensure their usefulness along with safety, traceability, reproducibility, and accountability. This work takes a step in that direction, taking into consideration cutting-edge research with tangible engineering practices, and represents a valid alternative for those companies wanting an easy, plug-and-play system to ride the newest wave of generative AI.

Bibliography

[1] Mark Treveil and the Dataiku Team. *Introducing MLOps: How to Scale Machine Learning in the Enterprise*. O'Reilly Media, 2020. ISBN: 9781492083306 (cit. on pp. 1, 2, 5–7).

[2] M. Stone et al. «Navigating MLOps: Insights into Maturity, Lifecycle, Tools, and Careers». In: *ACM Trans. on Intelligent Systems and Technology* 16.2 (2025), Art. 21 (cit. on pp. 1, 2, 4–7, 11, 30, 33, 37, 43).

[3] P. Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Proc. NeurIPS*. 2020, pp. 9459–9474 (cit. on pp. 1, 12–15).

[4] *Terraform: Infrastructure as Code*. HashiCorp. 2025. URL: `https://www.terraform.io/docs` (cit. on pp. 2, 37, 39).

[5] Cheshire Cat AI. *Cheshire Cat: Open-Source LLM Agent Framework*. 2024. URL: `https://cheshirecat.ai` (cit. on pp. 2, 20, 21).

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762` (cit. on p. 3).

[7] Google Cloud. *What is LLMOps – Large Language Model Operations*. Accessed: 2025-07-13. 2025 (cit. on pp. 4, 11, 13, 16, 30, 33, 37, 42).

[8] Google Cloud. *Practical Guide to MLOps: Operationalizing Machine Learning Models*. `https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning`. Accessed: 2025-07-13. 2023 (cit. on pp. 5–7).

[9] Noah Gift and Alfredo Deza. *Practical MLOps*. O'Reilly Media, 2022. ISBN: 9781098103019 (cit. on pp. 5–7).

[10] Microsoft Azure. *MLOps Maturity Model | Azure Architecture Center*. `https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/mlops-maturity-model`. Accessed: 2025-07-13. 2025 (cit. on p. 8).

[11] *MLflow Documentation*. Databricks. 2025. URL: `https://mlflow.org/docs/latest/ml/` (cit. on pp. 10, 29).

[12] Weights & Biases. *Weights & Biases Documentation*. https://docs.wandb.ai. Accessed: 2025-07-13. 2025 (cit. on p. 10).

[13] Anonymous. *Keyword Filtering: Limitations of Baseline Moderation*. 2022. URL: https://example.com/keyword-filtering-moderation (cit. on p. 17).

[14] Jigsaw/Google. *Perspective API*. 2024. URL: https://developers.perspectiveapi.com (cit. on p. 17).

[15] Unitary.ai. *Detoxify: Transformer-Based Toxicity Classifier*. 2021. URL: https://github.com/unitaryai/detoxify (cit. on p. 18).

[16] Tommaso Caselli, Valerio Basile, Jelena Mitrović, and Michael Granitzer. *HateBERT: Retraining BERT for Abusive Language Detection in English*. 2021. arXiv: 2010.12472 [cs.CL]. URL: https://arxiv.org/abs/2010.12472 (cit. on p. 18).

[17] Sander Schulhoff et al. *The Prompt Report: A Systematic Survey of Prompting Techniques*. 2024. arXiv: 2406.06608 [cs.CL]. URL: https://arxiv.org/abs/2406.06608 (cit. on p. 18).

[18] *Docker: Containerization Platform*. Docker, Inc. 2025. URL: https://docs.docker.com (cit. on pp. 23, 27, 28, 41).

[19] Pinecone. *Vector Databases: Indexing and Similarity Search*. 2023. URL: https://www.pinecone.io/learn/vector-database/ (cit. on p. 26).

[20] Comet. *Moderation Metrics | Comet Documentation*. https://www.comet.com/docs/opik/evaluation/metrics/moderation. Accessed: 2025-07-13. 2025 (cit. on p. 31).

[21] Comet. *Context Precision Metrics | Comet Documentation*. https://www.comet.com/docs/opik/evaluation/metrics/context_precision. Accessed: 2025-07-13. 2025 (cit. on p. 35).

[22] Amazon Web Services. *AWS Identity and Access Management (IAM)*. https://aws.amazon.com/it/iam/. Accessed: 2025-07-13. 2025 (cit. on p. 41).

[23] *Amazon DynamoDB Developer Guide*. Amazon Web Services. 2025. URL: https://aws.amazon.com/dynamodb/ (cit. on p. 41).

[24] *Amazon ECS Developer Guide*. Amazon Web Services. 2025. URL: https://aws.amazon.com/ecs/ (cit. on p. 42).

[25] *Amazon EFS User Guide*. Amazon Web Services. 2025. URL: https://aws.amazon.com/efs/ (cit. on p. 43).

[26] *Amazon SageMaker Developer Guide*. Amazon Web Services. 2025. URL: https://aws.amazon.com/sagemaker/ (cit. on p. 43).

[27] *Amazon Bedrock Developer Guide*. Amazon Web Services. 2025. URL: `https://aws.amazon.com/bedrock/` (cit. on p. 44).