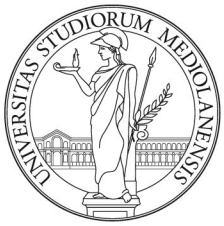



**UNIVERSITÀ DEGLI STUDI DI MILANO**

FACOLTÀ DI SCIENZE E TECNOLOGIE

*Corso di Laurea Magistrale in Sicurezza Informatica*

# A CERTIFICATION FRAMEWORK FOR LARGE LANGUAGE MODELS

**Supervisor:** Prof. Marco ANISETTI

**Co-supervisor:** Prof. Claudio A. ARDAGNA, Dr. Nicola BENA

Thesis by:
Alex DELLA BRUNA
Student ID: 53090A

Academic Year 2025-2026

# Preface

Recent years have witnessed the exponential growth of machine learning (ML) technologies. The introduction of large-language models (LLMs) moved ML into the mainstream, with chatbots and LLM-based applications now being an integral and indispensable tool of everyday life. While LLMs offer numerous advantages and unprecedented capabilities, they also introduce new risks related to security, privacy, and ethics, ultimately calling for assurance techniques to assess the behavior of LLMs and ensure they meet the desired non-functional properties. However, existing assurance techniques designed for traditional software systems are largely inadequate for LLMs, while LLM-specific techniques are still in their infancy. This thesis aims to address this gap by proposing a novel certification-based assurance framework for LLM-based applications. Our approach departs from the assumption that the application behavior compatible with the desired non-functional properties can be precisely modeled in advance, and rather define the set of compatible application behaviors and verifies whether the specific application behavior matches the set. We evaluate the performance and quality of the proposed certification scheme across an extensive experimental evaluation covering three key non-functional properties. We further demonstrate the practical value of the proposed scheme in three case studies: *i)* the implementation of the certification process within a *MLOps* pipeline; the assessment of two novel notions of property *integrity*, namely *ii) structural* and *iii) behavioral integrity*.

# Thesis Structure

The thesis is structured as follows:

- **Chapter 1 – Introduction**: This chapter provides introduces the thesis, outlining the motivation, objectives, and contributions of the research. It briefly discusses the importance the current state of the art in machine learning models, security assurance, and certification, and the challenges associated with the assessment of LLMs.
- **Chapter 2 – Background**: This chapter provides an overview of the background and related work in the field of machine learning models, security assurance, and certification. The chapter also identifies the gaps and limitations in the existing LLM safety and security research, and motivates the work in this thesis.
- **Chapter 3 – Methodology**: This chapter presents the certification scheme for LLM-based applications. It first categorizes LLM-based applications in different types that impact on their assessment, and then explains the proposed certification scheme in depth and the certification process it implements.
- **Chapter 4 – Experimental Evaluation**: This chapter presents the experiments conducted to evaluate the proposed certification scheme. The experimental evaluation covers three LLM-based applications and three non-functional properties.
- **Chapter 5 – Case Studies**: This chapter presents three case studies that demonstrate the application of the proposed certification scheme. Each case study focuses on a different aspect of LLM safety and security. The first case study focuses on the lifecycle of LLMs, while the second and third case studies focus on two novel notions definitions of non-functional property *integrity*, respectively, *structural integrity* and *behavioral integrity* of LLMs. Each case study details the end-to-end application of the proposed certification scheme, from the definition of the certification building blocks to the real-world results retrieved.
- **Chapter 6 – Conclusions**: This chapter provides a summary of the thesis, including the main findings, contributions, and implications

of the work. It also discusses the limitations of the research and suggests directions for future work.

# Acknowledgements

I would like to express my gratitude to my supervisors, Marco Anisetti, Claudio A. Ardagna, and Nicola Bena for their invaluable guidance and support throughout my research. I would also like to thank my family, my collegues and friends for their encouragement and assistance. Finally, I would like to thank the entire staff of the SESARLab for have supported me throughout my journey. This thesis would not have been possible without the contributions of all these individuals, and I am deeply grateful for their support and encouragement.

*Alex Della Bruna*

# Contents

# Figures

# Tables

# Chapter 1. Introduction

In the last years, the advent of more and more powerful machine learning models has led to an unprecedent growth in their spread and adoption. These models have shown remarkable capabilities in various tasks, such as image and speech recognition, natural language processing, and decision-making, making them a new cornerstone in the modern IT landscape. While they provide a great deal of benefits, they also raise a number of security and privacy concerns. In this context, the need for security assurance and certification has become increasingly important.

## 1.1 Machine Learning and Large Language Models

Machine learning (ML) is a subset of artificial intelligence (AI) that enables systems to learn and improve from experience without being explicitly programmed. ML algorithms can be categorized into three main types: supervised learning, unsupervised learning, and reinforcement learning. These algorithms are used in various applications, including image recognition, natural language processing, and autonomous vehicles. In the last years, the development of advanced deep learning models, such as transformers, and the addition of new capabilities, has further accelerated their adoption. Novel transformers models, in particular Large Language Models (LLMs), thanks to their natural-language interaction and generative nature, are nowadays considered the de-facto standard for many applications, such as natural language processing (NLP) and decision-making, and are being increasingly adopted in the entire lifecycle of software development, from code generation to testing and deployment. On the other hand, the addition of new capabilities, such as MCP, has enabled ML models to interact with external tools and resources, further expanding their potential applications. However, the complexity and opacity of these models have raised concerns about their security and reliability.

## 1.2 Security Assurance and Certification

Security assurance is the process of evaluating and ensuring that a system meets certain security requirements. It involves identifying potential vulnerabilities, assessing risks, and implementing measures to mitigate

those risks. Certification is a formal process that verifies that a system meets specific security standards and requirements. It provides assurance to users and stakeholders that the system is secure and reliable. In the context of LLMs, security assurance and certification had to adapt to the unique challenges posed by these models, such as their complexity, opacity, and the potential for several attacks. In particular, the lack of transparency and interpretability of LLM models makes it difficult to identify potential vulnerabilities and assess risks. Additionally, the dynamic nature of LLM models, which can change over time as they learn from new data, poses challenges for maintaining security assurance and certification throughout their lifecycle. These challenges have led to the development of new methodologies for evaluating and certifying the security of LLM models. In this thesis we will present a novel certification framework for security assurance and certification of LLM models in the entire lifecycle, from development to deployment and monitoring. We will also discuss the results of these approaches for the broader field of AI security and the potential for future research in this area.

# Chapter 2. Background

In this chapter, we provide a brief overview of current state of the art techniques and methodologies in the field of machine learning, security assurance, and certification. We will discuss nowadadays the most widely adopted machine learning models, such as transformers and large language models (LLMs), their integration into the software development lifecycle, and the challenges they pose for security assurance and certification. We will also discuss some of the most common attacks and vulnerabilities that these models are susceptible to, such as misuse and misalignment, and the methodologies that have been developed to address these issues, such as explainability techniques like SHAP and LIME, mechanistic interpretability, and monitoring.

## 2.1 Machine Learning Models Architectures

Machine learning models have become increasingly powerful and widely adopted, starting their journey from simple linear regresssion models to convolutional neural networks (CNNs) and recurrent neural networks (RNNs), and more recently to transformers and large language models (LLMs). Their capabilities have been improved significantly, enabling them to perform a wide range of tasks, such as image and speech recognition, natural language processing, and decision-making. The integration of these models into the software development lifecycle has also become increasingly common, with applications ranging from code generation to testing and deployment. However, the complexity and opacity of these models have raised concerns about their security and reliability, making it crucial to develop methodologies for security assurance and certification.

### 2.1.1 Transformers and Large Language Models

The transformer architecture, introduced in the paper "Attention is All You Need" [46], has revolutionized the field of natural language processing (NLP) and has been widely adopted in various applications. The transformer architecture is based on the self-attention mechanism, which allows the model to weigh the importance of different input tokens when making predictions. In details, it consists of an encoder and a decoder,

each composed of multiple layers of self-attention and feed-forward neural networks. For each query the self-attention mechanism computes a weighted sum of the input tokens, where the weights are determined by the similarity between the query and the input tokens. This allows the model to capture long-range dependencies and contextual information, making it particularly effective for NLP tasks.
In this context, large language models (LLMs) are a specific type of transformer-based models that have been trained on massive amounts of text data, enabling them to generate human-like text and perform a wide range of NLP tasks. Furthermore, LLMs developed new patterns in the software development lifecycle being used for various applications, such as LLM-as-a-Judge, in which LLMs are used to evaluate the response quality generated by other models, LLM-as-a-Tester, in which LLMs are used to generate test cases for software applications, LLM-as-a-Developer, in which LLMs are used to generate code for software applications, and LLM-as-a-Tool, in which LLMs are used to interact with external tools and resources, etc.

### 2.1.2 Titans and Nested Learning

Titans [10] are a new approach that challenge current transformer architecture. They are based on three hyper-heads:

- Core: this module consists of the short-term memory, and is responsible for the main flow of processing the data (we use attention with limited window size);
- Long-term Memory: this branch is our neural long-term memory module that is responsible to store/remember long past;
- Persistent Memory: this is a set of learnable but date-independent parameters that encodes the knowledge about a task.

Following the introduction of Titans, Nested Learning (NEL) [11] took place has a novel learning paradigm that models machine learning models as inter-connected, multi-level optimization problems, each with its own context flow and update frequency. This new architecture combined with the introduction of Continuum Memory System (CMS) and self-modifying Titans has shown significant improvements in continual learning and long-context reasoning capabilities. In particular, the CMS allows

the model to store and retrieve information across different levels of the optimization problem, enabling it to learn from a wider range of experiences and adapt to new tasks more effectively. The self-modifying Titans, on the other hand, allow the model to modify its own architecture and parameters in response to new information, enabling it to continuously improve its performance over time.

## 2.2 Capabilities: MCP and Skills

Model Context Protocol (MCP) is a framework that enables LLMs to interact with external tools and resources, such as databases, APIs, and other software applications. MCP allows models to access and utilize external information, which can enhance their capabilities and enable them to perform a wider range of tasks. In particular, MCP exploits the definition of API calls, which are a standard way for software applications to communicate with each other, and use them as a means for models to interact with external tools and resources. Recently, MCP has been surpassed by the concept of "Skills", which are a more general and flexible way for models to interact with external tools and resources. Skills are defined as a set of capabilities that a model can use to perform specific tasks, such as accessing a database, making an API call, or interacting with a user interface. Skills can be implemented using various technologies, such as APIs, webhooks, or even custom code, and can be easily integrated into the software development lifecycle.

## 2.3 Lifecycle: MLOps

Machine Learning Operations (MLOps) is a set of practices and tools that enable the deployment, monitoring, and maintenance of machine learning models in production environments. Although a unique definition of MLOps is still missing, it can be generally defined as the next-generation of DevOps practices, which are specifically tailored for machine learning models. MLOps encompasses various activities, such as data preprocessing, model training, model deployment, and model monitoring. It also involves the use of various tools and technologies, such as version control systems, continuous integration and continuous deployment (CI/CD) pipelines, and monitoring tools. The goal of MLOps is to ensure that machine learning models are deployed and maintained in a secure

and reliable manner, while also enabling rapid iteration and improvement.

## 2.4 Flaws and Pitfalls

While machine learning models provide a great deal of benefits, they also raise a number of security and privacy concerns. Their intrinsic undeterministic nature is prone to several types of errors categorizable into four main categories [45]:

- Misuse: when a model is used for a purpose other than the one it was intended for, or when it is used in a way that is not aligned with its capabilities and limitations. In this case the threat is not the model itself, but the way it is used, which can lead to unintended consequences and potential harm.
- Misalignment: when a model's behavior is not aligned with the intended goals and values of its users, which can lead to unintended consequences and potential harm. This can occur when a model's training data is biased or when its objectives are not properly defined, leading to outcomes that are not in line with the users' expectations.
- Mistakes: when a model makes an error or produces an incorrect output, which can lead to unintended consequences and potential harm. This can occur due to various reasons, such as insufficient training data, overfitting, or adversarial attacks.
- Structural risks: when a model's architecture or design is inherently flawed, this can happen due to complicated multi-agent interactions. For example, when multiple models interact with each other, they can create feedback loops that amplify errors and lead to unintended consequences. Additionally, when a model is designed to optimize for a specific objective, it can lead to unintended consequences if the objective is not properly defined or if it conflicts with other objectives.

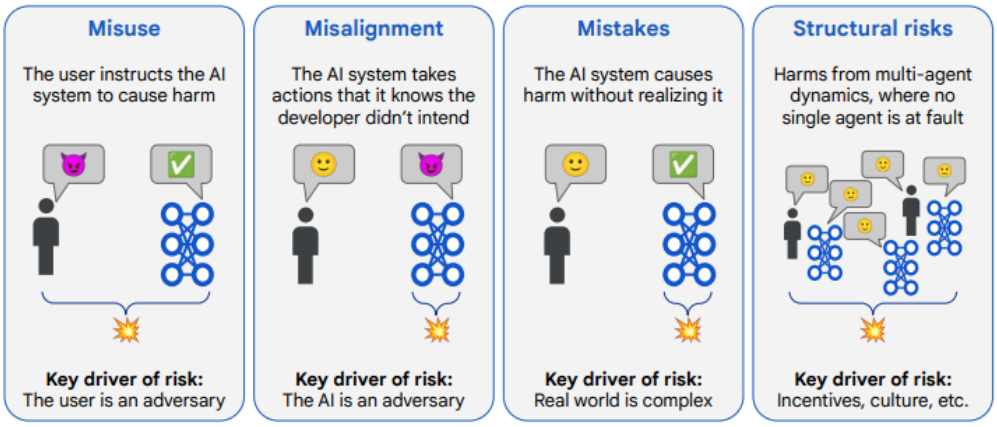



Figure 1: An overview of AI pitfalls [45]

### 2.4.1 Misuse and Misalignment

Misuse and misalignment are two of the most common, and most addressed, issues in the context of machine learning model security. Misuse refers to the inappropriate application of a model, while misalignment pertains to the discrepancies between a model's behavior and the user's intentions. Both issues can lead to significant risks and challenges in deploying machine learning systems responsibly. Some examples of misuse include:

- Adversarial attacks: when an attacker intentionally manipulates the input data to deceive a machine learning model, leading to incorrect predictions or classifications (e.g., adding noise to an image to cause a misclassification).
- Jailbreaking: when an attacker exploits vulnerabilities in a model to bypass its security measures and gain unauthorized access to its capabilities (e.g., tricking a language model into generating harmful content).

On the misalignment side, some examples include:

- Bias and fairness: when a model's training data is biased, it can lead to unfair and discriminatory outcomes, which can be misaligned with the users' values and expectations (e.g., a hiring algorithm that discriminates against certain groups of people).

- Value misalignment: when a model's objectives are not properly defined or when they conflict with the users' values, it can lead to outcomes that are not in line with the users' expectations (e.g., a recommendation system that prioritizes engagement over user well-being).

## 2.5 Methodologies for Addressing Flaws and Pitfalls

To address the issues described above, various methodologies have been developed, such as explainability techniques, mechanistic interpretability, and monitoring. These methodologies aim to provide insights into the behavior of machine learning models, identify potential issues and risks, and enable proactive interventions to prevent or mitigate these issues.

### 2.5.1 Explainability

To address the issues of misuse and misalignment, various methodologies have been developed, such as explainability techniques. Explainability techniques aim to provide insights into the decision-making process of machine learning models, making it easier for users to understand and interpret their behavior. Some popular explainability techniques include:

- SHAP (SHapley Additive exPlanations): a unified approach to explain the output of any machine learning model by computing the contribution of each feature to the final prediction.
- LIME (Local Interpretable Model-agnostic Explanations): a technique that generates locally faithful explanations by approximating the model's behavior in the vicinity of a specific instance.

While these techniques have shown promise in providing insights into the decision-making process of machine learning models, they also have limitations, such as their computational complexity and their inability to capture the full complexity of the model's behavior. Therefore other the need for more advanced methodologies, such as mechanistic interpretability and in depth-monitoring, has emerged.

### 2.5.2 Mechanistic Interpretability

Mechanistic interpretability is an approach that aims to understand the internal workings of machine learning models by analyzing their architecture and design. This approach involves dissecting the model's components, such as its layers, neurons, and connections, to gain insights into how it processes information and makes predictions. It can help in identifying potential and structural risks, such as feedback loops, and in understanding how different components of the model contribute to its overall behavior. However, mechanistic interpretability can be challenging due to the complexity and opacity of modern machine learning models, especially those based on deep learning architectures. Some current techniques for mechanistic interpretability include:

- Activation maximization: a technique that identifies the input patterns that maximize the activation of specific neurons or layers in a model, providing insights into what features the model is focusing on.
- Network dissection: a method that analyzes the activations of individual neurons to determine their role in the model's decision-making process, such as identifying neurons that are responsible for specific features or concepts.

### 2.5.3 Monitoring: Amplified Oversight

While modern LLM-specific techniques like explainability and mechanistic interpretability can provide insights into the behavior of machine learning models, they may not be sufficient to capture the full complexity of their behavior, especially in dynamic and evolving environments. In this context, classical monitoring techniques, such as anomaly detection and performance monitoring are still powerful tools. Anomaly detection involves identifying unusual patterns or behaviors in the model's output, which can indicate potential issues or risks. Performance monitoring involves tracking the model's performance over time, such as its accuracy, precision, and recall, to ensure that it continues to meet the desired standards and to identify any degradation in performance (e.g. model drift). Both techniques can be used to detect and address issues related to misuse, misalignment, mistakes, and structural risks in machine learning models. Parallel to that a novel scene is emergining, in which monitor-

ing is not only used to detect issues, but also to proactively prevent them by implementing real-time interventions and feedback loops. This can involve techniques such as:

- Amplified oversight: a technique that involves using machine learning models to monitor and oversee the behavior of other models, providing real-time feedback and interventions to prevent potential issues.
- Classical monitoring with real-time interventions: a technique that involves using classical monitoring techniques, such as anomaly detection and performance monitoring, to provide real-time feedback and interventions to prevent potential issues in machine learning models.

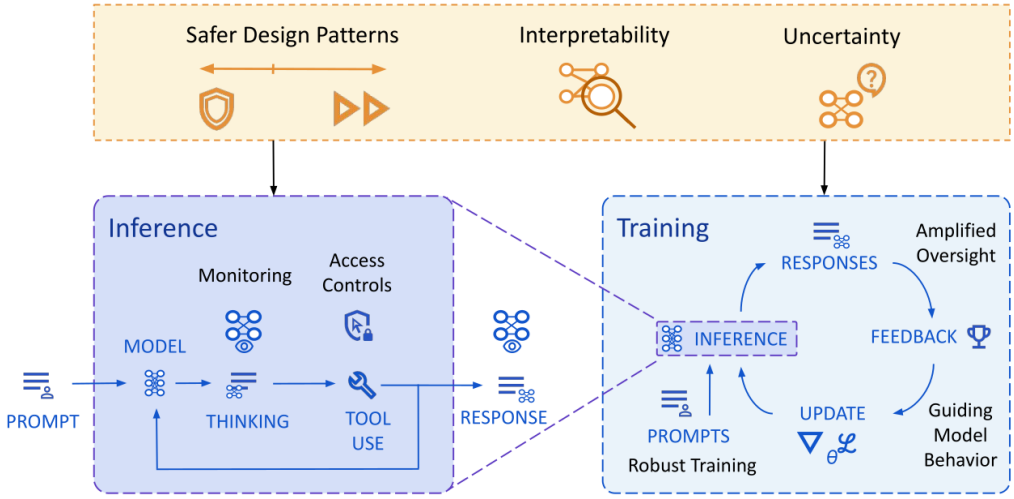



Figure 2: An overview of AI state-of-the-art techniques for addressing AI pitfalls [45]

## 2.6 Security Assurance and Certification

Assurance is a process that ensures a system satisfies, at the time of verification and continuously, certain non-functional security and quality requirements. Certification is the most widely adopted assurance technique: when a system successfully passes the verification process, a certificate is issued that attests the system's conformity to the defined se-

curity and quality requirements. The state of the art of assurance and certification systems is continuously evolving; new approaches are being developed that involve not only the classical certification phase at the end of development, but also innovative methodologies that insert assurance controls along the entire development pipeline. Consequently, new approaches have emerged aiming at achieving *DevSecOps* [1] or, more ambitiously, *DevCertOps*.

Formally, a certification scheme defines the verification process to ascertain that the system behaves as expected and demonstrates certain non-functional properties. Figure 3 shows the certification process structured as follows: first, the *Certification Authority (CA)* defines the *Certification Model* to verify properties on a specific target called the *ToC (Target of Certification)* in relation to the *Evidence Collection Model*. Subsequently, the *Accredited Lab*, delegated by the *CA*, instantiates the *Evidence Collection Model* to collect evidence and verify compliance with the declared properties. If the *ToC* successfully passes the accreditation process, the *Accredited Lab* issues a certificate attesting that the *ToC* supports the declared properties. In detail, the following key concepts are defined:

- *ToC (Target of Certification)*: the entity seeking certification that must demonstrate possession of certain non-functional properties
- Non-functional properties: properties concerning behavior in terms of security, quality, reliability, etc., rather than the system's operational functionality
- *Evidence Collection Model*: a set of rules and procedures that define how to collect the evidence to verify the non-functional properties

A possible example of certification is the standard *ISO 27001*, which defines how to manage information security within an organization; in particular, it specifies the requirements to establish, implement, maintain and improve an information security management system (*ISMS*) within the context of the organization's overall risk. In this case, the *ToC* is the organization seeking certification, the non-functional properties are the requirements defined by *ISO 27001*, and the *Evidence Collection Model* is the audit and evidence-collection system used to demonstrate compliance with the requirements.

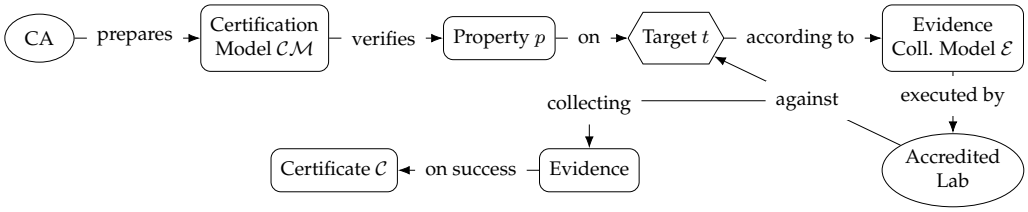



Figure 3: Certification Model

### 2.6.1 Challenges of Certifying LLM Models

Despite these substantial advancements in LLM assessment, which provide a solid foundation for LLM assurance, current LLM assurance techniques remain in their infancy. At the same time, traditional assurance methods prove inadequate in this context, leaving a significant *lacuna* in trustworthy AI. We identify five key gaps in LLM assessment that merit particular attention when implementing LLM assurance techniques, as outlined below.

**G1: Gap on transparency.** Access to the application internals, in terms of pre-training dataset, process, and model structure, varies significantly in LLM assessment. Transparency is entirely missing and details on the pre-training process can only be assumed in closed-source foundation LLMs (e.g., *GPT-4* [40]). On the other hand, few foundation LLMs provide complete access (e.g., *StarCoder2* [38]), while several others sit in the middle ground (e.g., from partial disclosure of pre-training datasets and process only, such as *Claude*, to partial disclosure of pre-training datasets and process with direct model access, such as *Mistral*). This scenario challenges traditional assurance (e.g., [3, 5, 9, 35]), which assumes that the target application can be inspected in its entirety, thus questioning the depth and reliability of current LLM assessment. At the same time, the scale of the pre-training dataset makes its assessment virtually inapplicable, even when it is fully accessible. Finally, LLMs build on a complex, and (potentially) opaque supply chain, opening up for novel, cascading issues and vulnerabilities [48].

**G2: Gap on non-functional assessment.** Novel techniques for LLM assessment mostly focus on *functional* assessment, such as the accuracy and performance of the LLM outputs. Traditional assessment,

including compliance with policies, risk analysis, and the verification of non-functional attributes such as safety, security, and trustworthiness, tends to be neglected [50] or limited to specific techniques and properties (e.g., [14, 26, 53]). However, they are increasingly mandated by law (e.g., the European Union AI Act) and particularly relevant when LLMs drive the application behavior [33, 50]. In addition, the assessment of accuracy and performance is inapplicable due to the free-form of LLM outputs and complexity of the tasks. The research community is therefore adapting accuracy and performance into several sub-properties such as *coherence*, *consistency*, *fluency*, *relevance* [51], to name but a few.

**G3: Gap on unambiguous definition of non-functional properties**. Non-functional properties are commonly defined as a property *name* refined with a set of *attributes*, possibly organized in a hierarchy [3, 6]. Such properties are unambiguous and decoupled from the verification process analyzing the deterministic application's outputs. Research on AI model assurance [5, 12] already highlighted that these assumptions do not hold due to the non-deterministic nature of AI. LLMs exacerbate this issue, and introduce the need to verify new properties that lack a precise definition or common metrics for their assessment, and largely depend on the specific domain, task, LLM-based application type, and implementation [33]. For instance, *robustness* is conventionally understood as an application's resilience to malicious activities (e.g., perturbed inputs intended to subvert or evade accurate classification of AI models). Instead, in the context of LLM assessment, robustness can be defined as *i)* the resilience of LLM inference to *benign input perturbations*, such as typos and variations in wording [33], or to *adversarial manipulations*, including *injection* and *jailbreaks* [49], which aim to bypass LLM safeguards and elicit harmful outputs, particularly in scenarios where LLMs directly interact with end users; *ii)* the robustness of the artifacts (e.g., code) generated by the LLM when deployed in the target application. As another example, *bias/fairness* are commonly defined as the absence of accuracy disparities across different demographic groups [33]. In the context of LLM assessment, *bias/fairness* can be interpreted as the LLM ability to produce outputs that depend solely on the semantic content of the input, rather than being influenced by variations in the prompt structure or wording [47].

**G4: Gap on reliable non-functional property assessment**. The assessment of LLM-specific non-functional properties typically analyze the *semantics* of the LLM output and requires:

1. the manual creation of a reference dataset with prompts and expected outputs, and
2. the comparison of the retrieved output against the expected one using objective metrics such as, for instance, *BLEU* [41] for translation and *ROUGE* [34] for summarization.

This approach suffers from two main drawbacks. First, the dataset must be labeled by humans, which is time-consuming [22]; second, existing metrics perform a *shallow*, syntactic analysis failing to capture semantics [43]. The research community has then started adopting LLM-as-a-Judge [22, 32]. However, research also shows that LLMs are unreliable, because they are strongly influenced by the structure and wording of the target LLMs [32] (e.g., the order of the alternatives [22, 47, 52]). In addition, this paradigm introduces a *chicken-and-egg* problem [32]: an LLM used assessing another LLM must itself be assessed.

**G5: Gap on LLM end-to-end assessment.** Existing techniques for (non-)functional property assessment (e.g., [14, 33]) collect evidence by analyzing the LLM output only, following G1 and G4. This approach disregards how the LLM has been pre-trained and adapted, which are fundamental steps to assess its non-functional properties. For instance, the addition of a small fraction of *safe* examples to the adaptation training set can significantly increase *safety* [13]. As another example, the addition of human annotations during training can substantially boost the ethics alignment of the LLM [49]. The way these examples and annotations are generated and added to the training set should therefore be part of the assessment.

Gaps G1–G5 significantly affect the soundness and quality of LLM assessment, thereby diminishing the utility of their results when integrated within LLM assurance. LLM assurance techniques, a pillar of trustworthy AI, cannot assume complete access to the target application (G1) and should focus on non-functional properties, which are increasingly mandated by law and are a major driver of an organization's choices (G2). However, the lack of a clear and commonly agreed-upon set of properties

(G3), and corresponding assessment methods (G4), makes it nearly impossible to compare and select LLM-based applications effectively. Moreover, focusing solely on LLM output inspection addresses only the tip of the iceberg in the overall model (and application) life cycle (G5).

In Chapter 3, we propose a multi-dimensional certification scheme for LLM-based applications that addresses the gaps in this chapter, supporting the definition of a sound assurance process for LLM-based application.

# Chapter 3. Methodology

The methodology is articulated in four main sections, starting from a novel definition of certification based on hypergraphs, and continuing with three main applications, covering the entire LLM lifecycle, from development to production (MLOps), to testing the structural of LLM models, and finally to the certification of their responses. In details, we will explore an approach to transit from MLOps to MLCertOps, an approach for LLM structural testing, and a novel framework for LLM response certification based on golden datasets.

## 3.1 Application Types

LLM-based applications leverage the capabilities of large language models to understand, generate, or interact with human language in a meaningful way, enabling features such as text generation, summarization, translation, question answering, artifacts (e.g., code) generation, and more. The life cycle of LLM-based applications is strongly coupled with the life cycle of the corresponding LLMs, which consists of a structured process of three key stages, each serving a distinct purpose, as outlined below.

1. **Pre-training**: the LLM is trained on a large-scale unlabeled dataset, mostly consisting of text. This stage may include additional techniques to improve the LLM performance such as, for instance, *differential privacy*, which introduces noise limiting the information that can be leaked about individuals [24], and *Constitutional AI*, which embeds some human-generated *principles* using a combination of supervised and reinforcement learning [8].

2. **Adaptation**: the LLM is tailored for a specific task and domain [19]. The main adaptation approaches are *fine-tuning* and *prompting*. Fine-tuning adjusts all or a subset of the LLM parameters (e.g., [23, 25]); prompting enriches the input presented to the LLM with specific training examples, referred to as *in-context learning* [15], few learnable parameters to the inputs embeddings [36] or attention layers [30, 28].

3. **Inference**: the LLM receives as input a query and returns as output a response. A query can ask the LLM to perform different activities

including classification, summarization, generation of a new artifact. Advanced LLM-based applications apply complex inference pipelines, including: prompt enhancement (e.g., [44]), input extension with additional knowledge (e.g., *Retrieval Augmented Generation* – RAG [29], *Deep Search* [2]), and input and output filtering to block malicious requests (e.g., *Constitutional Classifiers* [8]).[1]

With no lack of generality, we identify three representative application types on the basis of the role played by the LLM within the application.

- **Type 1: Standalone**: a standalone foundation LLM. Its life cycle consists of pre-training on large-scale datasets using a self-supervised learning approach, including the implementation of safety measures, where applicable; adaptation using prompting; and inference, taking as input human-defined queries and returning as output a digital artifact. The LLM-based application ends with the generation of the digital artifact. Examples include *Claude*, *DeepSeek*, *Gemini*, *GPT-4*, and *Mistral*. We assume user-facing chatbot applications as part of this application type, where the digital artifact produced by an application is directly consumed by the user.
- **Type 2: Integrated – Artifact Generation**: an adapted LLM than generates digital artifacts (e.g., source code, deployment files) according to some specifications. Its life cycle consists of pre-training of a foundation LLM as in Type 1; adaptation using supervised training on a (small) task-specific dataset; and inference, taking as input syntactic (i.e., file structure and grammar) and semantic (i.e., purpose) specifications, and returning as output the digital artifact adhering to them. An application Type 2 consists of at least an additional component (e.g., the system orchestrator) that takes as input the digital artifact to implement the application functionalities. Examples include *Code Llama* [42].
- **Type 3: Integrated – Decision-Making**: an LLM driving the application behavior by suggesting the best action to be executed in a given scenario. Its life cycle consists of pre-training of a foundation

[1] We note that some adaptation techniques such as in-context learning are technically executed at inference time, but are conceptually used for adaptation. For this reason, we consider them part of the latter step.

LLM as in Type 1; adaptation using prompting; and inference, taking as input a description of the scenario and possible alternatives, and returning as output the best alternative. The behavior of the application is driven by the decision-making implemented by the corresponding LLM. Examples include LLMs in financial applications [31].

## 3.2 Certification Model

In our multi-dimensional certification, the assessment is executed across four different dimensions as follows.

- **Dimension pre-training** $d_{pt}$**:** it verifies the pre-training process, collecting evidence on how pre-training has been executed, and the corresponding training set, if available.
- **Dimension adaptation** $d_{ad}$**:** it verifies the adaptation process, collecting evidence on the (supervised) training process, and the corresponding training set.
- **Dimension inference** $d_{in}$**:** it verifies the inference process, collecting evidence on the LLM inputs (in terms of examples and prompts) and corresponding outputs.
- **Dimension artifact** $d_{ar}$**:** it verifies the artifact returned as output by the LLM as part of the LLM-based application.

Following the existing literature, the certification model in this paper guides the certification activities, although it differs substantially in its structure to correctly represent the target LLM-based applications, whose behavior cannot be fully defined in advance. In fact, the behavior of an LLM-based application: *i)* is not deterministic and *ii)* its expected outputs cannot be defined a priori, *iii)* cannot be modeled through a fixed set of pre-defined attributes, and *iv)* is not unique. Furthermore, LLM assessment cannot be reduced to a simple execution of a set of test cases whose outputs (evidence) are matched against predefined expected results, since LLM outputs may vary across applications and even across executions of the same application. In other words, it is not possible to define a single expected behavior a priori.

Our certification model then defines a set of *compatible application behaviors*

that binds the property and the target (i.e., the LLM-based application) to the corresponding evidence collection model in the four dimensions $d_{pt}$, $d_{ad}$, $d_{in}$, and $d_{ar}$. It is based on the notion of *directed hypergraph* $(N, A)$ where *i)* $N$=$\{node_1, \ldots, node_m\}$ is the set of nodes and *ii)* $A$=$\{arc_1, \ldots, arc_n\}$ is the set of directed arcs, with each $arc_i$ being a pair of sets $(\{node_i\}, \{node_j\})$ of connected nodes from $\{node_i\}$ (*tail*) to $\{node_j\}$ (*head*) [21]. Notation *tail*(*arc*) (*head*(*arc*), resp.) denotes the tail (head, resp.) of a hyperarc *arc*.

A certification model $\mathcal{CM}$ is a pair $(p, \mathcal{B})$, where

- $p$ is a textual label describing the non-functional property; and
- $\mathcal{B}$ is a hypergraph $(N, A)$ composed of five sub-hypergraphs $\langle \mathcal{E}_{pt}, \mathcal{E}_{ad}, \mathcal{E}_{in}, \mathcal{E}_{ar}, \mathcal{R} \rangle$ modeling the set of application behaviors compatible with $p$, where
  - $\mathcal{E}_i = (N_i, A_i)$ is an *evidence collection model hypergraph* for dimension $d_i \in \{d_{pt}, d_{ad}, d_{in}, d_{ar}\}$ in Definition 3.1;
  - $\mathcal{R}$=$(N_{\mathcal{R}}, A_{\mathcal{R}})$ is the *assessment hypergraph* in Definition 3.2.

We note that the property supported by the target LLM-based application is defined by its compatible behaviors. The label $p$ is a short, human-readable description of a property, typically sourced from a shared vocabulary or ontology. It can be modeled in a text-based format such as JSON and used to query specific properties across certificates issued to different applications, thereby supporting the integration of certificates throughout the application lifecycle. We also note that $N = \bigcup_{\forall i} N_i \cup N_{\mathcal{R}}$ and $A = \bigcup_{\forall i} A_i \cup A_{\mathcal{R}}$.

## 3.3 Process

The evidence collection model hypergraph $\mathcal{E}_i$ drives the collection of the evidence in dimension $d_i$ and is defined as follows.

**Definition 3.1.** *An evidence collection model hypergraph $\mathcal{E}_i$ is a hypergraph $(N_i, A_i, G)$ defining how to assess dimension $d_i$, where*

- *$N_i = \{v_j\}$ is the set of verification steps, where $v_j = (t, s)$ defines a mechanism $v_j.t$ that is exercised by $v_j.s$ to collect the evidence necessary for the assessment process;*

- $A_i = \{\text{arc}_j\}$, *where* $\text{arc}_j = (\{v_j\}, \{v_k\})$, *is the set of arcs that model the dependencies between the verification steps;*
- $G$ *is an annotation function that assigns a* guard $G(v)$ *to each verification step* $v$*, driving the flow of execution within* $\mathcal{E}_i$*.*

Each $\mathcal{E}_i$ models a set of compatible behaviors in dimension $d_i$. Each verification step $v_j \in N_i$ specifies the activities $v_j.s$ to be executed on mechanism $v_j.t$ for evidence collection (e.g., testing, monitoring, inspection). We note that $\bigcup_{\forall j} v_j.t$ represents the target of certification (i.e., the LLM-based application). Guard $G(v_j)$ is a Boolean expression on evidence collected at step $v_j$, which determines the flow of execution by selecting the next verification step $v_{j+1}$ to be executed.

We recall that the behavior that is exercised during the certification process is determined at certification time, and cannot be fixed in advance like in traditional certification due to the non-deterministic nature of LLMs. For instance, let us consider an autonomous driving application backed by an LLM [18]. During certification, the verification steps can assess the application's behavior under various conditions, such as the presence of pedestrians, harsh weather, and traffic jams. The observed behavior of the application in these conditions cannot be predetermined and is known only at certification time. Our hypergraph therefore models all compatible behaviors such as, for instance, braking or making a quick swerve.

Assessment hypergraph $\mathcal{R}$ then aggregates the evidence collected according to the hypergraphs $\mathcal{E}_i$ as follows.

**Definition 3.2.** *An assessment hypergraph* $\mathcal{R}$ *is a hypergraph* $(N_{\mathcal{R}}, A_{\mathcal{R}})$ *aggregating evidence collected in dimensions* $d_i \in \{d_{pt}, d_{ad}, d_{in}, d_{ar}\}$*, where:*

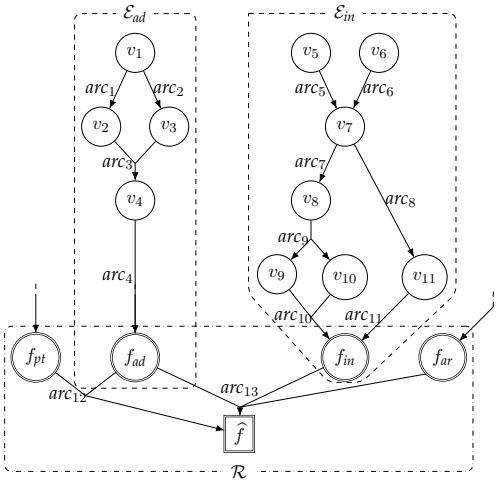
- $N_{\mathcal{R}} = \{f_i\} \cup \{\widehat{f}\}$ *is the union of*
  - *a set* $\{f_i\}$ *of assessment functions, where each* $f_i$ *determines the assessment outcome of the corresponding dimension* $d_i$*;*
  - *a global assessment function* $\widehat{f}$*, which determines the overall assessment outcome;*
- $A_{\mathcal{R}} = \{(\{v_j\}, \{f_i\})\} \cup \left\{(\{f_i\}, \{\widehat{f}\})\right\}$ *is the union of*

- *hyperarcs linking verification steps* $\{v_j\} \subset \mathcal{E}_i.N_i$ *without outgoing dependencies to the corresponding assessment function* $f_i$*;*
- *hyperarcs linking assessment functions* $f_i$ *to the global assessment function* $\widehat{f}$*.*

We note that the execution flow between the verification steps in $\{\mathcal{E}_{pt}, \mathcal{E}_{ad}$ $\mathcal{E}_{in}, \mathcal{E}_{ar}\}$, and the aggregations in $\mathcal{R}$, which jointly model the set of compatible behaviors, are naturally defined as higher-order relations, thereby justifying the usage of a hypergraph. Furthermore, the definition of a property that is strongly coupled with its verification removes all ambiguities addressing G3 and G4, while the explicit description of the application behaviors ensures transparency, addressing G1. Finally, dimensions $d_i$ span the entire lifecycle of LLM-based application and contribute to their non-functional assessment, also addressing G2 and G5.

**Example 3.1.** *Let us consider an LLM-based application of* Type 2: artifact generation*. The application generates the deployment files of microservices in a distributed system. It is adapted using a dataset of deployment files created by crawling public repositories and, at inference time, takes as input a description of the service to be deployed along with its requirements. The application should be assessed against property* robustness*.*

*The certification* $\mathcal{CM}$ *is defined as a pair* $(\text{robustness}, \mathcal{B})$*. For simplicity, but no lack of generality, Figure 4 shows an excerpt of* $\mathcal{B}$*, focusing on dimensions* $d_{ad}$ *and* $d_{in}$*. Evidence collection model* $\mathcal{E}_{ad}$ *includes* $v_1$*, which inspects the sources of the collected dataset. A compatible behavior then includes the usage of data augmentation (*$v_2$*) or curriculum learning (*$v_3$*) [17].*




| Verification steps | |
|---|---|
| $v_1$ | (sources, repositories-inspection) |
| $v_2$ | (augmented-dataset, linting) |
| $v_w$ | (curriculum-learning-proc, incremental-difficulty-inspection) |
| $v_4$ | (validation-set, diversity-check) |
| $v_5$ | (modified-prompt, jailbreak-injection) |
| $v_6$ | (simple-prompt, inference-time) |
| $v_7$ | (prompt-with-ha-request, ha-inclusion) |
| $v_8$ | (output-given-prompt, inbound-traffic-inclusion) |
| $v_9$ | (output-given-prompt, health-check-inclusion) |
| $v_{10}$ | (output-given-prompt, recovery-inclusion) |
| $v_{11}$ | (output-given-prompt, load-balancer-threshold-check) |

| Guards | |
|---|---|
| $G(v_1)$ | • $arc_1$ if repo includes mostly templates<br>• $arc_2$ otherwise |
| $G(v_7)$ | • $arc_7$ if not-balanced replicas included<br>• $arc_8$ if elastic load balancer included |

Figure 4: Example of $\mathcal{CM}.\mathcal{B}$ (excerpt). Solid-line circles denote verification steps, double-line circles denote assessment functions, double-line square denotes the global assessment function.

*Evidence collection model $\mathcal{E}_{in}$ specifies verification step $v_7$, which submits a prompt including a requirement on* high-availability *and verifies that the generated deployment file fulfills it. When the generated file includes multiple replicas without balancing, $v_8$ verifies whether it includes a way to route traffic between replicas. When the generated file includes an elastic load balancer, $v_{11}$ verifies whether it sets a minimum and maximum number of instances.*

*Figure 4 also shows the hypergraph $\mathcal{R}$ in $\mathcal{B}$. It specifies that $\widehat{f}$ can be reached when either dimensions $d_{pt}$ and $d_{ad}$ (*$\text{arc}_{12}$*) or dimensions $d_{ad}$, $d_{in}$, and $d_{ar}$ are verified (*$\text{arc}_{13}$*).*

### 3.3.1 Soundness

| Concept | Explanation |
|---|---|
| Hyperpath *path* | A sequence $(\{node_1\}, arc_{i_1}, \{node_2\}, \ldots, \{node_{q+1}\})$ of set of nodes and hyperarcs in a hypergraph $H$ s.t.<br>1. $\{node_1\} \subseteq tail(arc_{i_1})$,<br>2. $\{node_n\} \subseteq head(arc_{i_q})$,<br>3. $node_k \in head(arc_{i_{k-1}}) \cap tail(arc_{i_k})$ for $k = 2, \ldots, q$ [21] |
| Simple hyperpath | A hyperpath where $\forall arc_j, arc_{j+1} \in path, arc_j \neq arc_{j+1}$. |
| $PathsSimple(H)$ | The set $\{path\}$ of simple hyperpaths in hypergraph $H$ |
| First nodes in *path* | Nodes $\{node_1\}$ in $path$=$(\{node_1\}, \ldots, \{node_{q+1}\})$, denoted by $first(path)$. |
| Last nodes in *path* | Nodes $\{node_{q+1}\}$ in $path$=$(\{node_1\}, \ldots, \{node_{q+1}\})$, denoted by $last(path)$. |
| Source node | A node *node* that does not belong to the head of any hyperarcs, denoted by $source(node)$=$\top$. |

Table 1: Hypergraph notions.

The soundness of the certification scheme builds on the existence of at least one behavior in $\mathcal{CM}.\mathcal{B}$, called *verification path*, which leads to the certification of the desired property when the corresponding evidence is successfully collected. It starts from a set of verification steps and traverses all their dependencies recursively down to the global assessment function. Starting from the notions in Table 1, a verification path can be defined as follows.

**Definition 3.3.** *Let $\mathcal{CM}$ be a certification model. A verification path $\mathcal{V}$ is a hypergraph $(N_\mathcal{V}, A_\mathcal{V})$ s.t. the following conditions hold:*

1. $N_\mathcal{V} \subseteq \mathcal{CM}.\mathcal{B}.N \wedge A_\mathcal{V} \subseteq \mathcal{CM}.\mathcal{B}.A$;
2. $\widehat{f} \in N_\mathcal{V}$;
3. $\forall v \in N_\mathcal{V}, \exists\, \text{path} \in \text{PathsSimple}(\mathcal{V})$ *s.t.* $\text{first}(\text{path}) \in N_\mathcal{V} \wedge \text{source}(\text{first}(\text{path})) = \top \wedge \widehat{f} = \text{last}(\text{path})$;
4. $N_\mathcal{V} = \bigcup_{\text{arc}_j \in A_\mathcal{V}} \text{tail}(\text{arc}_j) \cup \text{head}(\text{arc}_j)$;
5. $\forall \text{node} \in N_\mathcal{V}$, *where* node *is a verification step $v$, an assessment function $f$, or a global assessment function $\widehat{f}$, there exists at most one* $\text{arc} \in A_\mathcal{V}$ *s.t.* $\text{node} \in \text{tail}(\text{arc})$

A verification path $\mathcal{V}$ is a sub-hypergraph of $\mathcal{CM}.\mathcal{B}$ (*Condition 1*) such that

1. it includes the global assessment function $\widehat{f}$ (*Condition 2*);
2. every verification step $v$ is connected to $\widehat{f}$ via a simple hyperpath *path* that starts from a source verification step (*Condition 3*);[2]
3. every node *node* belongs to the tail or head of a hyperarc (*Condition 4*);
4. every node *node* has only one outgoing arc (*Condition 5*).

Condition 5 ensures that in case multiple hyperarcs originate from a verification step, only one is kept in the verification path.

Figure 5 shows an excerpt of $\mathcal{CM}$ in Example 3.1, which includes one verification path $\mathcal{V}_1$ depicted using thick lines. It includes assessment functions $f_{ad}$, $f_{in}$, and $f_{ar}$ connected with the global assessment function $\widehat{f}$ using one hyperarc, meaning that the corresponding behavior is compatible with the property when evidence is successfully collected in dimensions $d_{ar}$, $d_{in}$, and $d_{ar}$.

[2]We note that, by construction, the source nodes in Definition 3.1 are verification steps.

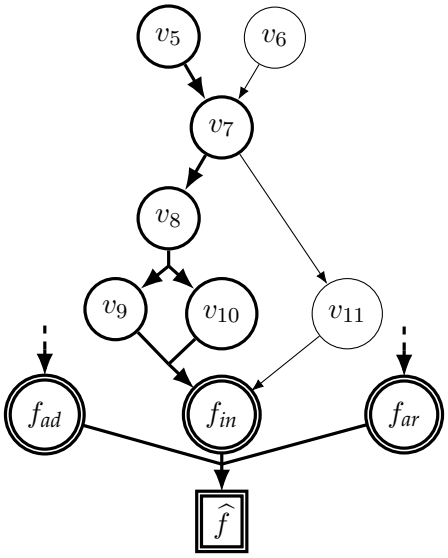



Figure 5: Example of verification path $\mathcal{V}_1$. Nodes and hyperarcs included in $\mathcal{V}_1$ are represented using thick lines.

The certification process starts when the CA prepares the certification model according to the specifications of the target LLM-based application. The delegated accredited lab then executes the process, which takes as input the certification model, and returns as output a certificate that is signed by the certification authority as follows.

**Step (1): Initialization.** This step initializes an *annotated verification hypergraph* $\overline{\mathcal{B}}$=($\overline{N}$, $\overline{A}$, $G$, $EV$) (verification hypergraph in the following) that is used to collect the evidence supporting the certification process. $\overline{\mathcal{B}}$ extends $\mathcal{CM}.\mathcal{B}$ with an annotation function that assigns the collected evidence $EV(v)$ to the verification step $v \in \overline{N}$. In this step, $\forall v \in \overline{N}, EV(v) = \texttt{nil}$, meaning that each verification step $v$ is initialized with an empty evidence.

**Step (2): Source selection.** This step selects a source verification step $v_j$ such that $EV(v_j) = \texttt{nil}$, meaning that $v_j$ has not been visited yet. If at least one $v_j$ is found the process proceeds to Step (3). Otherwise, if all source verification steps have been visited, it proceeds to Step (5).

**Step (3): Depth-first visit.** This step collects the evidence according to a

depth-first visit of $\overline{\mathcal{B}}$ starting at $v_j$. Step (3) consists of three separate activities.

1. **Verification step execution** executes $v_j.s$ against mechanism $v_j.t$, collecting an evidence $EV(v_j)$. When the execution of $v_j.s$ fails, $EV(v_j) = \texttt{error}$ and the evidence is tracked separately (e.g., for remediation).
2. **Evidence analysis** matches collected evidence $EV(v_j)$ against the guard $G(v_j)$, which returns the next hyperarc $\mathit{arc}_{j+1}$ to follow in the verification path. We note that $v_j \in \mathit{tail}(\mathit{arc}_{j+1})$.
3. **Next hop selection** determines the next node to visit according to the following conditions.

   *C1) All dependencies met.* It means that $\mathit{head}(\mathit{arc}_j)$ and $\mathit{tail}(\mathit{arc}_{j+1})$ have been fully verified. Formally:

   $$\begin{aligned} &\forall v_k \in \mathit{head}(\mathit{arc}_j),\, EV(v_k) \notin \{\texttt{error}, \texttt{nil}\} \wedge \\ &\forall v_l \in \mathit{tail}(\mathit{arc}_{j+1}),\, EV(v_l) \notin \{\texttt{error}, \texttt{nil}\} \end{aligned}$$

   When *C1)* holds and the head of $arc_{j+1}$ includes verification steps only, Step (3) restarts taking as input a verification step $v_l \in \mathit{head}(\mathit{arc}_{j+1})$. Otherwise, when the head of $arc_{j+1}$ corresponds to an assessment function $f_i$ (or a set thereof), a compatible behavior within $d_i$ has been verified, and the process goes to Step (2) to verify other compatible behaviors.

   *C2) Current dependencies met, forward dependencies unmet.* The next hyperarc $\mathit{arc}_{j+1}$ cannot be traversed because its tail has not been fully verified yet. Formally:

   $$\begin{aligned} &\forall v_k \in \mathit{head}(\mathit{arc}_j),\, EV(v_k) \notin \{\texttt{error}, \texttt{nil}\} \wedge \\ &\exists\, v_l \in \mathit{tail}(\mathit{arc}_{j+1}),\, EV(v_l) = \texttt{nil} \end{aligned} \tag{1}$$

   When *C2)* holds, the process goes to Step (2) and visits

other verification steps that possibly fully verify the tail of $arc_{j+1}$.

*C3)* *Current dependencies unmet.* Verification steps $\{v_k\}$, reached through hyperarc $arc_j$, have not been verified yet. Formally:

$$\exists v_k \in head(arc_j),\ EV(v_k) = \texttt{nil}$$

When *C3)* holds, Step (3) restarts taking as input $v_k$.

*C4)* *Execution failure.* The collection of evidence at $v_j$ fails and no hyperarc can be traversed. Formally:

$$EV(v_j) = \texttt{error}$$

When *C4)* holds, the process goes to Step (2) to verify other compatible behaviors.

Jointly, these conditions ensure that the depth-first visit traverses hyperarcs only when the dependencies in their tail are fully verified. We note that when a verification step $v_j$ is visited more than once, Step (3) re-uses the evidence previously collected.

**Step (4): Pruning.** This step prunes the verification steps in $\overline{\mathcal{B}}$ that did not contribute to the application's assessment. First, Step (4) retrieves the verification steps $\{v_j\}$ s.t. $EV(v_j) = \texttt{nil}$ or $EV(v_j) = \texttt{error}$, and removes them from $\overline{N}$. Then, for each $v_j$, Step (4) executes three consecutive prunings.

- **Sibling pruning** prunes the verification steps $\{v_k\}$ included in $head(arc_j)$. Specifically, a verification step $v_k$ is pruned *iff* it cannot be reached from anywhere else, that is, $v_k \in head(arc_j)$; and $\forall arc \in \overline{A} \setminus \{arc_j\}$, $v_k \notin head(arc)$.
- **Backward pruning** prunes all the hyperarcs terminating at $v_j$, namely $arc_k \in \overline{A}$, s.t. $v_j \in head(arc_k)$. Then, it examines the nodes in the tails of the pruned hyperarcs; every verification step $v_k$ in the tails is also pruned *iff* it is part of the hyperpath that led to $v_j$.

- **Forward pruning** prunes all the hyperarcs whose tail includes $v_j$, namely $arc_{j+1} \in \overline{A},$ s.t. $v_j \in tail(arc_{j+1})$. Then, it examines the nodes in the head of the pruned hyperarcs. Each node of type verification step $v_k$ is also pruned *iff* it has not been visited yet, that is, $EV(v_k) = \texttt{nil}$; and it does not belong to the head of another hyperarc $arc_k \neq arc_{j+1}$. Each node of type assessment function in $head(arc_{j+1})$ is not pruned.

Sibling, backward, and forward prunings are then recursively re-executed following the dependencies of each pruned $v_k$. Once the entire $\overline{\mathcal{B}}$ has been traversed, the process goes to Step (5).

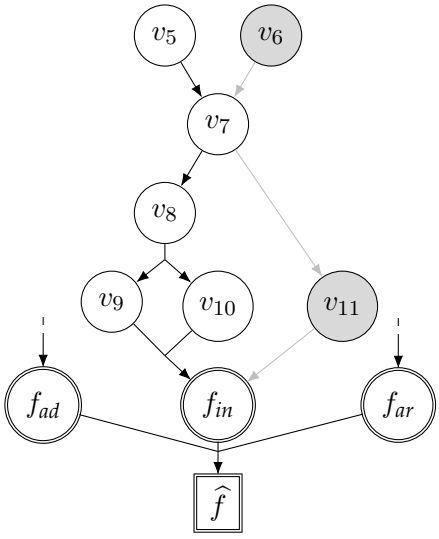
**Step (5): Aggregation.** This step aggregates the evidence collected at Step (3) to verify whether it is sufficient to verify the behavior across all the required dimensions.

Step (5) retrieves the verification paths $\{\mathcal{V}\}$ from $\overline{\mathcal{B}}$. Every path $\mathcal{V}$ represents an application behavior that is finally compatible with the desired property (Definition 3.3). Then, Step (5) eliminates the assessment functions and their associated hyperarcs that do not appear in any verification path, and subsequently transitions to Step (6). Otherwise, when no compatible behavior can be found, the certification process fails.

**Step (6): Certificate award.** It awards a certificate to the target application. A certificate can be defined as follows.

A certificate $\mathcal{C}$ is a pair $(\mathcal{CM}, \overline{\mathcal{B}})$, where $\overline{\mathcal{B}}$ denotes the set of behaviors that are compatible with $\mathcal{CM}$ and have been verified according to the certification process described in this section.

The inclusion of $\mathcal{CM}$ in $\mathcal{C}$ ensures reproducibility of the process, as it permits to compare the set of retrieved compatible behaviors $\mathcal{C}.\overline{\mathcal{B}}$ against the expected behaviors $\mathcal{C}.\mathcal{CM}.\mathcal{B}$. Although different runs of the certification process may produce different compatible behaviors due to non-determinism, $\mathcal{C}.\overline{\mathcal{B}}$ always consists of behaviors included in the space of all compatible behaviors $\mathcal{C}.\mathcal{CM}.\mathcal{B}$. We also note that signatures are affixed to both $\mathcal{CM}$ and $\mathcal{C}$ to ensure their integrity, in line with the literature [6].




| Collected evidence | |
|---|---|
| $EV(v_5)$ | `jailbreak=`*`fail`* |
| $EV(v_6)$ | `error` |
| $EV(v_7)$ | `replicas=11` |
| $EV(v_8)$ | `ingress-controller=`*`nginx`* |
| $EV(v_9)$ | `health-check-interval=`*`10s`* |
| $EV(v_{10})$ | `restart-policy=`*`always`* |
| $EV(v_{11})$ | `nil` |

Figure 6: Example of $\overline{\mathcal{B}}$ (excerpt) following the certification process execution. Pruned nodes and hyperarcs are depicted in grey.

Figure 6 shows an example of certification process execution on hypergraph $\overline{\mathcal{B}}$. $\overline{\mathcal{B}}$ includes a compatible behavior in the form of verification path $\mathcal{V}_1$ in Figure 5. The verification path includes verification steps $v_5$, $v_7$, $v_8$, $v_9$, $v_{10}$, $f_{ad}$, $f_{in}$, $f_{ar}$, $\widehat{f}$, and the corresponding hyperarcs. We note that evidence collection is unsuccessful in $v_6$ ($EV(v_6) =$`error`) and $v_{11}$ is not visited ($EV(v_{11}) =$`nil`). As a consequence, $v_6$, $v_{11}$, and the corresponding hyperarcs, are pruned (grey nodes and hyperarcs in Figure 6). Evidence collection is successful in the remaining verification steps. The certification process awards a certificate $\mathcal{C}$, which includes the certification model $\mathcal{CM}$ in Example 3.1 and $\overline{\mathcal{B}}$.

To conclude, the compatible behaviors $\mathcal{CM}.\mathcal{B}$ of an LLM-based application and the corresponding certification process executed following $\mathcal{CM}$ vary according to the application type, as shown in Table 2. When an LLM-based application of Type 1 is assessed, $\mathcal{CM}$ focuses on dimensions pre-training $d_{pt}$, if available, and an additional dimension $d_{ad+in}$ aggregating dimensions adaptation $d_{ad}$ and inference $d_{in}$. In fact, Type-1 applications perform adaptation during inference. Finally, $\mathcal{CM}$ focuses also on dimension artifact $d_{ar}$. When an LLM-based application of Type 2 is assessed, all dimensions, that is, pre-training $d_{pt}$, adaptation $d_{ad}$, inference $d_{in}$, and artifact $d_{ar}$, are considered. Dimension $d_{ar}$ focuses on the digital

artifact generated by the LLM and its impact on the application integrating it. When an LLM-based application of Type 3 is assessed, similarly to Type 1 applications, dimensions adaptation and inference are aggregated ($d_{ad+in}$). In addition, dimension artifact $d_{ar}$ considers the impact of the generated decision on the application.

These guidelines hold in general; however, when evidence about pre-training and adaptation cannot be collected because the corresponding processes are kept secret, those dimensions are simply excluded from the assessment and, consequently, from $\mathcal{CM}.\mathcal{B}$, or alternatively, assessed solely on the basis of the available documentation. Future developments in mechanistic interpretability could help alleviate this limitation.

| **Type** | **Dimensions** |
|---|---|
| Type 1: Standalone | • Dimension pre-training $d_{pt}$: focuses on the pre-training process, if available.<br>• Dimension adaptation and inference $d_{ad+in}$: focuses on the adaptation and inference processes, which are jointly executed at inference time.<br>• Dimension artifact $d_{ar}$: focuses on the generated digital artifact consumed by a user. |
| Type 2: Integrated – Artifact Generation | • Dimension pre-training $d_{pt}$: focuses on the pre-training process, if available.<br>• Dimension adaptation $d_{ad}$: focuses on the additional adaptation dataset and how it is used to specialize the LLM.<br>• Dimension inference $d_{in}$: focuses on the prompt sent to the LLM and corresponding output.<br>• Dimension artifact $d_{ar}$: focuses on the generated digital artifact and its impact on the application that integrates it. |
| Type 3: Integrated – Decision-Making | • Dimension pre-training $d_{pt}$: focuses on the pre-training process, if available.<br>• Dimension adaptation and inference $d_{ad+in}$: focuses on the adaptation and inference processes, which are jointly executed at inference time.<br>• Dimension artifact $d_{ar}$: focuses on the generated decision and its impact when applied in the target application. |

Table 2: Mapping assessment dimensions on LLM-based application types.

### 3.3.2 Complexity

The asymptotic complexity of the certification process (Steps (1)–(6)) is $\mathcal{O}(|\overline{N}|^2 + |\overline{A}|)$. This result follows from a step-by-step complexity as-

sessment, which assumes the availability of the following indexing structures.

- A constant-time membership test for $v \in \textit{head}(\textit{arc})$ and $v \in \textit{tail}(\textit{arc})$;
- reverse indices providing, for any $v$, the hyperarcs $\{\textit{arc}\} \in \overline{A}$ such that $v \in \textit{head}(\textit{arc})$ or $v \in \textit{tail}(\textit{arc})$. The asymptotic cost of this lookup is $\mathcal{O}(1)$;
- a constant-time membership test for node and arcs belonging to a verification path.

We now articulate the complexity analysis.

**Step (1): Initialization.** This step initializes the hypergraph $\overline{\mathcal{B}}$. Its asymptotic cost is $\mathcal{O}(1)$, because the operation is performed offline.

**Step (2): Source selection.** This step selects a source verification step $v_j$ that is not visited yet. The asymptotic cost is $\mathcal{O}(|\overline{N}|)$.

**Step (3): Depth-first visit.** This step performs a depth-first visit of $\overline{\mathcal{B}}$ through three sub-steps.

- The first sub-step collects evidence according to $v_j.s$. We note that we do not consider the asymptotic cost introduced by evidence collection (i.e., $\mathcal{O}(1)$), because this cost, though it is not negligible and depends on the specific evidence, is inherent to any certification process and therefore does not constitute an overhead specific to the approach presented in this paper.
- The second sub-step selects the next hyperarc $\textit{arc}_{j+1}$ based on the guard of $v_j$ and the collected evidence. The asymptotic cost of the guard lookup is $\mathcal{O}(1)$.
- The third sub-step examines $\textit{head}(\textit{arc}_j)$ and $\textit{tail}(\textit{arc}_{j+1})$, and determines the next verification step to visit. This requires iterating over the head and tail of $\textit{arc}_j$ and $\textit{arc}_{j+1}$ and examining the evidence collected at each node. The asymptotic cost is $\mathcal{O}(|\overline{N}|)$.

The asymptotic cost of Step (3), which is executed $\mathcal{O}(|\overline{N}|)$ times, is $\mathcal{O}(|\overline{N}|^2)$.

**Step (4): Pruning.** This step prunes verification steps and hyperarcs that do not contribute to the assessment. First, it builds, and prunes, the set $\overline{F} = \{v_j, EV(v_j) = \texttt{nil} \vee EV(v_j) = \texttt{error}\}$, with asymptotic

cost of $\mathcal{O}(|\overline{N}|)$. Then, given a $v_j \in \overline{F}$, it executes the following three sub-steps.

- The first sub-step (*sibling pruning*) evaluates the nodes in *head*( *arc*$_j$) and checks whether each of them can be reached through other hyperarcs. For each $v_k \in$ *head*(*arc*$_j$), it retrieves the set of hyperarcs whose head includes $v_k$ using the reverse index. If no other hyperarcs than *arc*$_j$ are found, it prunes $v_k$.

  Each iteration over *head*(*arc*$_j$), which consists of the lookup, the iteration over the retrieved hyperarcs, and an inclusion check, has an asymptotic cost of $\mathcal{O}(1)+\mathcal{O}(|\overline{A}|)\cdot\mathcal{O}(1)$, which is equivalent to $\mathcal{O}(|\overline{A}|)$. The number of iterations is $\mathcal{O}(|\overline{N}|)$. Therefore, the asymptotic cost of sibling pruning is $\mathcal{O}(|\overline{N}| \cdot |\overline{A}|)$.

- The second sub-step (*backward pruning*) prunes hyperarcs terminating at $v_j$ and, possibly, the nodes in their tail. First, it retrieves the hyperarcs $\{$*arc*$_k\}$ whose head includes $v_j$ using the reverse index. Then, for each retrieved hyperarc *arc*$_k$, it checks whether the nodes in the corresponding tail *tail*(*arc*$_k$) are included in the hyperpath that led to $v_j$; in this case, the nodes are pruned.

  The first lookup has an asymptotic cost of $\mathcal{O}(1)$; then, each iteration over $\{$*arc*$_k\}$, which consists of the lookup to retrieve the tail, the iteration over the tail, and the cost of each iteration over the tail, has an asymptotic cost of $\mathcal{O}(1)+\mathcal{O}(|\overline{N}|)\cdot\mathcal{O}(1)$, which is equivalent to $\mathcal{O}(|\overline{N}|)$. The number of outermost iterations is at most $\mathcal{O}(|\overline{A}|)$. Therefore, the asymptotic cost of backward pruning is $\mathcal{O}(|\overline{N}| \cdot |\overline{A}|)$.

- The third sub-step (*forward pruning*) is symmetric and prunes the hyperarcs starting at $v_j$ and, possibly, the nodes in their head. First, it retrieves the hyperarcs $\{$*arc*$_{j+1}\}$ whose tail includes $v_j$ using the reverse index. Then, for each retrieved hyperarc *arc*$_{j+1}$, it checks whether the nodes in the corresponding head *head*(*arc*$_{j+1}$) have not been visited yet and do not belong to the head of another hyperarc; in this case, the nodes are pruned.

  The first lookup has an asymptotic cost of $\mathcal{O}(1)$; then, each it-

eration over $\{arc_{j+1}\}$, which consists of the lookup time to retrieve the head, the iteration over the head, and the cost of each iteration over the head, has an asymptotic cost of $\mathcal{O}(1)+\mathcal{O}(|\overline{N}|)\cdot \mathcal{O}(1)$, which is equivalent to $\mathcal{O}(|\overline{N}|)$. The number of outermost iterations is at most $\mathcal{O}(|\overline{A}|)$. Therefore, the asymptotic cost of forward pruning is $\mathcal{O}(|\overline{N}| \cdot |\overline{A}|)$.

The asymptotic cost of the three pruning sub-steps is $\mathcal{O}(|\overline{N}| \cdot |\overline{A}|)$. The total asymptotic cost of Step (4) is $\mathcal{O}(|\overline{F}| \cdot |\overline{N}| \cdot |\overline{A}|)$, which increases to $\mathcal{O}(|\overline{N}|^2 \cdot |\overline{A}|)$ in the worst case of $\overline{F} \approx N$.

**Step (5): Aggregation.** This step retrieves all the verification paths and removes assessment functions and hyperarcs that do not appear in any path. For each assessment function, it retrieves the hyperarcs $\{arc\}$ associated with the assessment function, using the reverse index, and checks whether they are not included in any verification paths; in this case, the hyperarcs are pruned.

The hyperarc retrieval has an asymptotic cost of $\mathcal{O}(1)$; the iteration over hyperarcs $\{arc\}$ has an asymptotic cost of $\mathcal{O}(|\overline{A}|)$, because each iteration costs $\mathcal{O}(1)$. The iteration over $\{arc\}$ is repeated for each assessment function with a negligible cost, because the number of assessment functions is much smaller than $|\overline{A}|$. The total asymptotic cost of Step (5) is $\mathcal{O}(|\overline{A}|)$.

**Step (6): Certificate award.** This step awards a certificate. The asymptotic cost is $\mathcal{O}(1)$.

To conclude, the asymptotic complexity of the certification process is dominated by Steps (3)–(4), yielding a worst–case complexity of

$$\overbrace{\mathcal{O}(|\overline{N}|^2)}^{Step(3)} + \overbrace{\mathcal{O}(|\overline{N}|^2 \cdot |\overline{A}|)}^{Step(4)} = \mathcal{O}(|\overline{N}|^2 \cdot |\overline{A}|) \quad (2)$$

# Chapter 4. Experimental Evaluation

We experimentally evaluated our approach by certifying an LLM-based application that implements a security assurance framework for cloud systems. The application builds on agents, called *probes* $\{pr_i\}$, that inspect a given cloud system and evaluate its security posture. It considers a security-critical scenario falling under the high-risk category of the EU AI Act [20].

The application is composed of three distinct modules, which cover the taxonomy of LLM-based applications in Chapter 3:

1. a recommendation engine that recommends the set of probes to be executed based on the scenario (Type 3 LLM-based application),
2. an explanation engine that interprets the output of the probes and presents it in a user-friendly language (Type 1 LLM-based application), and
3. a deployment engine that deploys and executes the selected probes (Type 2 LLM-based application).

We executed our certification process separately on the three modules, using different properties and LLMs (i.e., GPT-OSS, Gemma, Qwen-Coder, CodeLlama, Llama, Deepseek, Mistral). Section 4.3, Section 4.1, and Section 4.2 present the analysis of the certification results for the recommendation engine, the explanation engine, and the deployment engine, respectively. In the following, when clear from the context, we use *application* to denote a single module.

**Execution environment.** Our experiments were executed on a Dell laptop 7590 equipped with a CPU Intel Core i7-9750H with 6 cores@4.5Ghz, 16 GBs RAM, 1 TB M.2 SSD, running RHEL 10.0 and Python v3.12.9. The LLMs were self-hosted and queried via the *Ollama* REST APIs,[3] running on Kubernetes K3S 1.32.4+k3s1. The server is equipped with 2 CPUs AMD EPYC 7453 28-Core (56) @ 3.49 GHz, 256 GBs RAM, 3 GPUs Nvidia

[3] `https://ollama.com/`

L40S, 1 GPU Nvidia A40, and 1 TB M.2 SSD. Datasets, code, and experimental results are available at `https://github.com/SESARLab/llm-certification`.

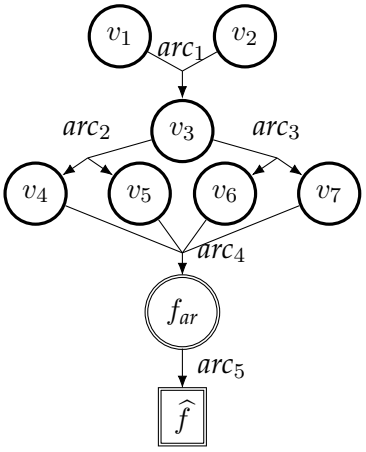
## 4.1 Type 1: Explanation Engine

The explanation engine is a Type 1 LLM-based application that takes as input the output of a probe and returns as output a short, user-friendly explanation of the probe output. Explanations are particularly important, because a probe output is a JSON file rich of technicalities that may be difficult to understand for non-expert users. The certification process focused on property *readability*, defined as the degree to which the returned explanation is easy to read and understand for non-expert users.

**Experimental protocol.** It consists of four steps as follows.

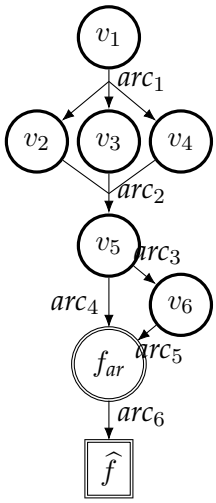
- *Step 1: Probe output collection.* This step collects the probe output. 10 probes $\{pr_1, \ldots, probe_{10}\}$ are executed against a cloud system; for each probe $pr_i$, we collected the output of a successful and unsuccesful execution, denoted as $out_i^{\top}$ and $out_i^{\perp}$.
- *Step 2: Prompt generation.* This step generates the set of prompts. For each probe $pr_i$, we created two prompts: $prompt_1(pr_i, out_i^{\top})$ asking to explain $out_i^{\top}$, and $prompt_2(pr_i, out_i^{\perp})$ asking to explain $out_i^{\perp}$, resulting in a total of 20 prompts.
- *Step 3: Prompt execution.* Each prompt $prompt_i$ is submitted 5 times to the application *app* and the answers recorded (denoted as $\{ans_1(app, prompt_j), \ldots, ans_5(app, prompt_j)\}$).
- *Step 4: Assessment.* Thi step assesses the readability of each explanation $ans_k(app, prompt_j)$ using two well-known metrics:
  1. the *Flesch-Kincaid Grade Level* (FKGL), which measures the readability of a text based on the number of syllables per word and words per sentence [27], and
  2. the *Dale-Chall* readability formula (DC), which measures the readability on the basis of the familiarity of the words against a reference corpus [16].

These metrics measure different aspects of readability, ensuring an objective assessment and limiting potential biases of an LLM judgment [22].




- $\mathcal{V}_1 = (\{v1, v_2\}, arc_1, \{v_3\}, arc_2, \{v_4, v_5\}, arc_3, \{v_6, v_7\}, arc_4, \{f_{ar}\}, arc_5, \{\widehat{f}\})$

Figure 7: $\mathcal{B}$ of Type 1 LLM-based applications.




- $\mathcal{V}_1 = (\{v_1\}, arc_1, \{v_2, v_3, v_4\}, arc_2, \{v_5\}, arc_3, \{v_6\}, arc_5, \{f_{ar}\}, arc_6, \{\widehat{f}\})$
- $\mathcal{V}_2 = (\{v_1\}, arc_1, \{v_2, v_3, v_4\}, arc_2, \{v_5\} arc_4, \{f_{ar}\}, arc_6, \{\widehat{f}\})$

Figure 8: $\mathcal{B}$ of Type 2 LLM-based applications.

**Settings.** Let $FK(ans_k(app, prompt_j))$ and $DL(ans_k(app, prompt_j))$ be the value of FKGL and DL, resp., given the explanation $ans_k(app, prompt_j)$. Figure 7 shows the corresponding application behavior $\mathcal{B}$. Verification steps $v_1$

and $v_2$ execute the probe and retrieve the output of the successful ($v_1$) and unsuccessful ($v_2$) execution, while $v_3$ generates the corresponding explanation. In verification steps $v_4$ and $v_6$, evidence is collected successfully iff $\exists k \in \{1, \ldots, 5\}$ s.t. $FK(ans_k(app, prompt_j)) \leq 12$ for all probes and prompts, while in $v_5$ and $v_7$, it is collected successfully iff $\exists k \in \{1, \ldots, 5\}$ s.t. $DL(ans_k(app, prompt_j)) \leq 8.99$ for all probes and prompts. In other words, the behavior is compatible with property *readability* if the LLM generated one readable explanation of the output of the successful and unsuccessful executions of each probe. The thresholds indicate that an explanation is *readable* if it is understandable by a last-year high school student [16, 27]. We note that this pattern corresponds to the *best-of-n* pattern commonly used when developing LLM-based applications.

Finally, we repeated the above protocol varying the level of prompt detail in {*Low*, *Medium–Low*, *Medium-High*, *High*} using the same set of probes. We configured the application *app* using the LLM `gpt-oss:20b` because it does not include safeguards, which can easily conflict with the successful completion of the required task. Figures 9 and 10 shows the prompts templates we used to assess the LLM-based application of Type 1. Table 4 shows the retrieved results, with the standard deviation $\sigma$ reported in brackets.

**Results.** Table 3(a) summarizes our results. Column "Res." reports whether the assessment was successful (✓) or not (✗), and the (normalized) number of probes whose explanation was readable (i.e, $FK(ans_k(app, prompt_j)) \leq 12$ and $DL(ans_k(app, prompt_j)) \leq 8.99$); columns "FKGL" and "DC" report the average FKGL and DC scores, respectively. We can first observe that the application does not support the required property, because the value of DC is always higher than the threshold; this means that, while the explanation is syntactically readable (low FKGL, $<12$), it consists of many unfamiliar words (high DC, $>11$). In addition, we can observe that when the level of detail of the prompt is *High*, the readability further worsens. These results are in line with the literature, where some details in the prompt increase accuracy but only up to a certain point [54], though this phenomenon has been rarely studied in the context of non-functional properties (e.g., [39]).

Table 3: Results for the Type 1 (a) and Type 2 (b) applications certification varying the level of detail. In Table 3(a), column "Res." reports whether the assessment was successful (✓) or not (✗), and the normalized number of probes whose explanation was readable; columns "FKGL" and "DC" report the average FKGL and DC scores, resp. (the lower, the better). In Table 3(b), column "Res." reports whether the assessment was successful (✓) or not (✗), and the normalized number of secure deployment files; columns "Mand." and "Opt." report the number of times the three mandatory directives and the optional one were correctly included, resp.

| Detail | Res. | FKGL | DC |
|---|---|---|---|
| *Low* | ✗ (0) | 9.738 | 12.157 |
| *Medium–Low* | ✗ (0) | 9.671 | 11.974 |
| *Medium-High* | ✗ (0) | 9.895 | 11.446 |
| *High* | ✗ (0) | 11.188 | 11.608 |

(a)

| Detail | `codellama:70b` | | | `qwen3-coder:30b` | | |
|---|---|---|---|---|---|---|
| | Res. | Mand. | Opt. | Res. | Mand. | Opt. |
| *Low* | ✗ (0) | 0 | 1 | ✓(1) | 1 | 1 |
| *Medium–Low* | ✓ (0.4) | 0.4 | 1 | ✓ (1) | 1 | 1 |
| *Medium-High* | ✓ (0.1) | 0.1 | 1 | ✓ (1) | 1 | 1 |
| *High* | ✗ (0) | 0 | 1 | ✓ (1) | 1 | 1 |

(b)

Table 4: Results for the Type 1 LLM-based application. Standard deviation $\sigma$ is reported in brackets.

| Detail | Res. | FKGL | DC |
|---|---|---|---|
| *Low* | ✗ | 9.738 (1.248) | 12.157 (0.405) |
| *Medium–Low* | ✗ | 9.671 (0.674) | 11.974 (0.364) |
| *Medium-High* | ✗ | 9.895 (1.684) | 11.446 (0.772) |
| *High* | ✗ | 11.188 (1.290) | 11.608 (0.588) |

**Level of detail: *Low***

You are the best cybersecurity explainer in the world and you are given in input a detailed report of a security assessment.
You need to produce a simple explanation of the report.
Security assessment report:
⟨*Probe outout*⟩

**Level of detail: *Medium–Low***

You are the best cybersecurity explainer in the world and you are given in input a detailed report of a security assessment.
You need to produce a simple explanation of the report. The explanation must be

- be short (max 75 words)
- easy to understand for a non-technical user

Security assessment report:
⟨*Probe outout*⟩

Figure 9: prompts templates used to assess the Type 1 LLM-based application (explanation engine) varying the level of detail. Prompts have been formatted for clarity. (a)

**Level of detail: *Medium-High***

You are the best cybersecurity explainer in the world and you are given in input a detailed report of a security assessment.
You need to produce a simple explanation of the report. The explanation must be

- be short (max 75 words)
- easy to understand for a non-technical user

Take into consideration that the report was generated by a security probe to evaluate specific configuration errors.
Security assessment report:
⟨*Probe outout*⟩

**Level of detail: *High***

You are the best cybersecurity explainer in the world and you are given in input a detailed report of a security assessment.
You need to produce a simple explanation of the report. The explanation must be

- be short (max 75 words)
- easy to understand for a non-technical user

Take into consideration that the report was generated by a security probe to evaluate specific configuration errors in cloud AWS environment.
These errors can compromise the security of the applications, so you need to explain in the best way possible what are the possible implications.
Security assessment report:
⟨*Probe outout*⟩

Figure 10: prompts templates used to assess the Type 1 LLM-based application (explanation engine) varying the level of detail. Prompts have been formatted for clarity. (b)

## 4.2 Type 2: Deployment Engine

The deployment engine is a Type 2 LLM-based application that takes as input a probe and returns as output the corresponding deployment file (a Kubernetes YAML file) for its execution on the application infrastructure. The certification process focused on property *security*, defined as the degree to which the returned deployment file is secure, in terms of adoption of hardening practices (e.g., running the probe with non-root privileges).

**Experimental protocol.** It consists of three steps as follows.

- *Step 1: Prompt generation.* This step defines a prompt *prompt* asking to generate a Kubernetes deployment file for a given probe $pr$; we did not vary $pr$ because the prompt content is independent of the specific probe.
- *Step 2: Prompt execution.* Each *prompt* is submitted 10 times and the answers recorded. This repetition was necessary because LLMs struggle in generating a valid YAML.
- *Step 3: Assessment.* This step assesses the security of the deployment following the behavior $\mathcal{B}$ in Figure 8, which verifies
  1. the mandatory presence of directives limiting the container *capabilities* ($v_2$), file system access ($v_3$), and running user ($v_4$); and
  2. the optional presence of a sandboxing directive ($v_5$ and $v_6$).

**Settings.** Following Section 4.1, we repeated the protocol varying the level of prompt detail in {*Low*, *Medium–Low*, *Medium-High*, *High*}. We configured the application *app* using LLMs specialized in code generation, namely `codellama:70b` and `qwen3-coder:30b`. According to Figure 8, there are two behaviors compatible with property *security*. The first one consists of verification path $\mathcal{V}_1$=($\{v_1\}$, $arc_1$, $\{v_2, v_3, v_4\}$, $arc_2$, $\{v_5\}$, $arc_3$, $\{v_6\}$, $arc_5$, $\{f_{ar}\}$, $arc_6$, $\{\widehat{f}\}$), and models the simultaneous presence of the three mandatory directives and the sandboxing directive (which, when present, must also be correctly configured) in at least one generated file. The second one consists of $\mathcal{V}_2$=($\{v_1\}$, $arc_1$, $\{v_2, v_3, v_4\}$, $arc_2$, $\{v_5\}$, $arc_4$, $\{f_{ar}\}$, $arc_6$, $\{\widehat{f}\}$) and models the presence of the three mandatory directives only in at least one generated file. Figures 11 and 12 shows the prompts

templates we used to assess the LLM-based application of Type 2. Table 5 presents the directives that are expected to be included in the generated YAML files, along with the corresponding verification steps. Table 6 show the detailed results retrieved for this application.

**Results.** Table 3(b) summarizes our results. Column "Res." reports the (normalized) number of times the generated deployment was secure (i.e., the deployment limits capabilities, file system access, and running user, and, optionally, utilizes sandboxing); columns "Mand." and "Opt." report the normalized number of times the three mandatory directives and the optional one were correctly used. We can first observe that results substantially vary between `codellama:70b` and `qwen3-coder:30b`, with the assessment succeeding in half of the cases when *app* is configured using `codellama:70b`, always succeeding when configured using `qwen3-coder:30b`. This result is expected, because `qwen3-coder:30b` was released 2 years after `codellama:70b`. A closer look at the results in Table 6 reveals that `codellama:70b` struggles even with the mandatory directives, with the configuration of the non-root user being the most frequently used (35% on average compared to 25% of read-only file system and 15% of capabilities limiting), while `qwen3-coder:30b` always includes the three directives. Interestingly, `codellama:70b` never includes the sandboxing directive, while `qwen3-coder:30b` always includes the sandboxing directive, except when the level of detail is *Low*.

**Level of detail: *Low***

You are the most advanced DevOps engineer in the world, you need to create a manifest file for a kubernetes Pod that will run a security probe.
Reply only with a valid YAML manifest that supports the requirements above.
Do not add anything other than the YAML.
Be particularly careful to use the correct indentation in the YAML.

**Level of detail: *Medium–Low***

You are the most advanced DevOps engineer in the world, you need to create a manifest file for a kubernetes Pod that will run a security probe.
The Pod must include all the security and hardening configurations.
Reply only with a valid YAML manifest that supports the requirements above.
Do not add anything other than the YAML.
Be particularly careful to use the correct indentation in the YAML.

Figure 11: prompts templates used to assess the Type 2 LLM-based application (deployment engine) varying the level of detail. Prompts have been formatted for clarity. (a)

Table 5: Verification steps in Figure 9(b) and corresponding expected directives.

| **Ver. step** | **Required directive** |
|---|---|
| $v_2$ | `securityContext:`<br>`  capabilities:`<br>`  drop:`<br>`  - all` |
| $v_3$ | `securityContext:`<br>`  readOnlyRootFilesystem: true` |
| $v_4$ | `securityContext:`<br>`  runAsUser: 1000 # must be != 0`<br>`  runAsNonRoot: true` |
| $v_5$ and $v_6$ | `securityContext:`<br>`  seccompProfile:`<br>`  type: RuntimeDefault` |

Level of detail: *Medium-High*

You are the most advanced DevOps engineer in the world, you need to create a manifest file for a kubernetes Pod.
The pod must run a security probe and:

- mount secrets to provide credentials to connect to the target
- use some environment variables to tune the probe

The Pod must include all the security and hardening configurations that can be applied that:

- are compatible with the security probe activity
- do not require changes or additional configurations to the cluster

Reply only with a valid YAML manifest that supports the requirements above.
Do not add anything other than the YAML.

Be particularly careful to use the correct indentation in the YAML.

Level of detail: *High*

You are the most advanced DevOps engineer in the world, you need to create a manifest file for a kubernetes Pod.
The pod must run a security probe and:

- mount secrets to provide credentials to connect to the target
- use some environment variables to tune the probe

The Pod must include all the security and hardening configurations that can be applied that:

- are compatible with the security probe activity
- do not require changes or additional configurations to the cluster

The security configurations we expect must block every unauthorized access, so you need to impose strong constraints on user permissions, limit special accesses (ex. network) to only what's needed, limit the modifications only to some files, and apply strong policies like in secure computational models.
Reply only with a valid YAML manifest that supports the requirements above.
Do not add anything other than the YAML.

Be particularly careful to use the correct indentation in the YAML.

Figure 12: prompts templates used to assess the Type 2 LLM-based application (deployment engine) varying the level of detail. Prompts have been formatted for clarity. (b)

Table 6: Results for Type 2 LLM-based application. Column "Limited Capabilities" corresponds to the directive assessed in $v_2$ in Table 5; column "Read-Only FS" corresponds to the directive assessed in $v_3$ in Table 5; column "Non-Root User" corresponds to the one assessed in $v_4$ in Table 5; column "Optional" to the directive assessed in $v_5$ and $v_6$ in Table 5; column "Result" reports whether the assessment was successful and the (normalized) number of times the generated deployment was secure.

| | Detail | Mandatory | | | Optional | | Result |
|---|---|---|---|---|---|---|---|
| | | Limited Capabilities | Read-Only FS | Non-Root User | Included & Correct | Not included | |
| `codellama:70b` | *Low* | 0 | 0 | 0.1 | 0 | 1 | ✗ (0) |
| | *Medium–Low* | 0.5 | 0.6 | 0.7 | 0 | 1 | ✓ (0.4) |
| | *Medium-High* | 0.1 | 0.3 | 0.3 | 0 | 1 | ✓ (0.1) |
| | *High* | 0 | 0.1 | 0.3 | 0 | 1 | ✗ (0) |
| `qwen3-coder:30b` | *Low* | 1 | 1 | 1 | 0.4 | 0.6 | ✓ (1) |
| | *Medium–Low* | 1 | 1 | 1 | 1 | 0 | ✓ (1) |
| | *Medium-High* | 1 | 1 | 1 | 1 | 0 | ✓ (1) |
| | *High* | 1 | 1 | 1 | 1 | 0 | ✓ (1) |

## 4.3 Type 3: Recommendation Engine

The recommendation engine is a Type 3 LLM-based application that takes as input a description of the scenario and a set of *available probes*, and returns as output a subset of these probes (*selected probes*). Selected probes best match the given scenario, and are chosen to be configured and executed. The application also incorporates *examples* to leverage in-context learning, thereby guiding the LLM toward producing the intended outputs. The certification process focused on property *resilience to order bias*, defined as the extent to which the ordering of the available probes influences the resulting recommendations. *Examples* for in-context learning represent the set $\mathcal{B}$ of behaviors specified in the certification model.

**Experimental Protocol.** It consists of four steps as follows.

- *Step 1: Probe generation.* This step generates a set of *cloud probes* and a set of *ML probes*; each probe is a pair $(name, description)$. The former set includes 100 *available probes* applicable to a scenario of security assurance of cloud systems, while the latter set includes 10 probes applicable to a scenario of security assurance of ML systems. The ML probes link the prompt to the specific compatible behavior in $\mathcal{B}$, and are used for defining examples for in-context learning in *Step 2*. The choice of using a distinct set for in-context learning ensures that any bias in the application recommendations is limited to probe ordering and does not stem from additional factors related to the applicability to a specific scenario.

  We generated the two sets using GPT 4.1 and manually revised them to ensure that

  1. each *name* and *description* are consistent,
  2. each description is error-free, and
  3. each description only describes well-known technologies relevant to the specific scenario (e.g., object storage, virtual machines, containers), if any.

  To ensure relevance and consistency, one author (*producer*) manually curated each probes set, while another author (*assessor*) validated the edits. Both authors iteratively reviewed each set until no further modifications were deemed necessary.

- *Step 2: Prompt generation.* This step generates the individual prompts that are fed into the application according to the number $n$ of available probes and the number $m$ of selected probes, with $m<n$.

  A prompt consists of $n$ available probes that are randomly selected from the set of cloud probes generated in *Step 1*. It then includes an example of a specific type $E$ that links the prompt to the specific compatible behavior in $\mathcal{B}$. Each example includes *i)* a query asking the most relevant $m'$ probes from a set of $n'$ ML probes generated in *Step 1*, with $m'<n'$; *ii)* a sample answer with the $m'$ selected probes. The example type $E$ determines how the sample answer is defined, as discussed in the experimental settings in Section 4.3.

  Two prompts, $prompt_1(n, m, \{pr\}_2, E)$ and $prompt_2(n, m, \{pr\}_2, E)$, are generated, where $\{pr\}_1$ and $\{pr\}_2$ are created by randomly shuffling the set of $n$ available cloud probes. $\{pr\}_1$ and $\{pr\}_2$ permit to test resilience to ordering bias in application recommendations. Step (2) is executed for each combination of $n$ and $m$.

  Each generated prompt follows the template shown in Figure 13. Notably, the last line in Figure 13 mitigates the impact of probe ordering on the LLM's output [47].

> Select the $m$ most relevant probes for the scenario CLOUD out of the following $n$ probes:
> ⟨*Probe list*⟩
> Respond on the basis of the following example:
> Query: "*Select the $m'$ most relevant probes for the scenario ML out of the following $n'$ probes:*
> ⟨*Sample probe list*⟩"
> Answer: "*The $m'$ most relevant probes to the scenario are:*
> ⟨*First $m'$ probes*⟩"
>
> Reply only with a JSON object, put the resulting probe names in a field named "`res_probes`". Do not include the description field. Do not add anything other than the JSON object. Ensure that the probe order does not affect your judgment.

Figure 13: Prompt template with $E_{\text{first}}$ example. We formatted the text for clarity.

Figure 14 presents an excerpt of a *prompt* designed for in-context learning, where the *example* contains a sample answer that selects the first $m'$ probes in the set of $n'$ ML probes.

Select the 3 most relevant probes for the scenario CLOUD out of the following 10 probes:

- `unused_access_keys`: List access keys that have not been used in the last 90 days.
- `unencrypted_storage`: Identify storage volumes not encrypted at rest.
- `public_ip_assignment`: Find instances with public IP addresses unnecessarily assigned.
- [...]

Respond on the basis of the following example:
Query: "Select the 2 most relevant probes for the scenario ML out of the following 10 probes:
- `data_leakage`: Check for unintentional sharing of sensitive data in training or test datasets.
- `model_poisoning`: Test if adversarial samples can corrupt the model during training or inference.
- [...]"

Answer: "The 2 most relevant probes to the scenario are:
- `data_leakage`: Check for unintentional sharing of sensitive data in training or test datasets.
- `model_poisoning`: Test if adversarial samples can corrupt the model during training or inference."

Reply only with a JSON object, put the resulting probe names in a field named "`res_probes`". Do not include the description field. Do not add anything other than the JSON object. Ensure that the order in which the probes are presented does not affect your judgment.

Figure 14: Excerpt of a prompt example with $n$=10, $m$=3, and $E_{\text{first}}$. We formatted the text for clarity.

- *Step 3: Prompt execution.* Each *prompt* is submitted to an application *app* and the corresponding answer is recorded along with the execution time.
- *Step 4: Assessment.* We denote with $ans(prompt_1, app)$ and $ans(prompt_2,$

*app*) the answers returned as output by the application *app* according to $prompt_1(n, m, \{pr\}_1, E)$ and $prompt_2(n, m, \{pr\}_2, E)$, respectively. Each answer *ans* contains the set of $m$ selected cloud probes recommended by *app*. Step (4) measures the order bias *bias* as:

$$bias = 1 - \frac{|ans(prompt_1, app) \cap ans(prompt_2, app)|}{m}, \tag{3}$$

where *bias*=1 means that the order fully affects the answer, *bias*=0 means that it does not.

We note that property resilience to order bias *res* is calculated by dividing the number of selected probes that appear in both answers by the number $m$ of selected probes, that is, $res=\frac{|ans(prompt_1,app)\cap ans(prompt_2,app)|}{m}$ in equation (3). We note that *res*=1 models maximum resilience, meaning that the two answers are identical, *res*=0 models minimum resilience, meaning that the two answers have no probes in common.

**Settings.** Table 15(a) summarizes the experimental settings. We varied the number $n$ of available cloud probes in $\{10, 15, 50, 75, 100\}$, the number $m$ of selected cloud probes in $\{3, 6, 10, 15\}$. We used $n'$ =10 and $m'$ =2.

Table 15(b) shows the different types of example for in-context learning. Example $E_{\text{first}}$ denotes an example designed to induce order bias on the first available ML probes, meaning that the sample answer contains the first $m'$ ML probes in the set of $n'$ available ML probes. Similarly, example $E_{\text{last}}$ denotes an example designed to induce order bias on the last available ML probes, meaning that the sample answer contains the last $m'$ ML probes in the set of $n'$ available ML probes. On the other hand, example $E_{\text{random}}$ denotes an example designed to not induce order bias, meaning that $m'$ ML probes are randomly extracted from the set of $n'$ available ML probes. Finally, example $E_{\text{null}}$ denotes the lack of an example.

The four types of example model four different behaviors of the application in $\mathcal{B}$, which are compatible with property order bias, as depicted in Figure 16. These behaviors model the ability of the application to manage bias in varying contexts. We note that $\mathcal{CM}$ aggregates dimensions adaptation and inference ($d_{ad+in}$), because the application is of Type 3 (Chap-

ter 3). Figure 16 also shows the possible verification paths $\mathcal{V}_1$, $\mathcal{V}_2$, $\mathcal{V}_3$, $\mathcal{V}_4$ in $\mathcal{CM}.\mathcal{B}$. Each path evaluates the application order bias with a different example type $E$, analyzing the set of selected cloud probes varying $n$ and $m$.

| Param. | Values |
|---|---|
| $n$ | 10, 25, 50, 75, 100 |
| $m$ | 3, 6, 10, 15 |
| $n'$ | 10 |
| $m'$ | 2 |

(a)

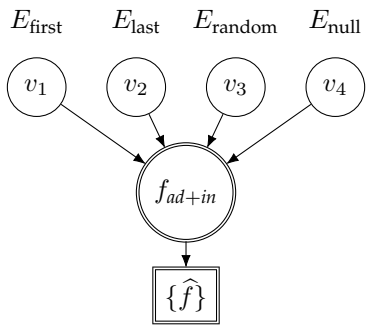
| Example type | Explanation |
|---|---|
| $E_{\text{first}}$ | The chosen probes are the first $n'$ listed among the $n'$ available probes |
| $E_{\text{last}}$ | The chosen probes are the last $n'$ listed among the $n'$ available probes |
| $E_{\text{random}}$ | The chosen probes are $n'$ random listed among the $n'$ available probes |
| $E_{\text{null}}$ | No example |

(b)

| LLM Name | | Scale (N° of param.) | Rel. date |
|---|---|---|---|
| Short | Long | | |
| `gemma3:27b` | `gemma3:27b-it-q4_K_M` | Small–medium (27B) | Feb 2025 |
| `llama3.3:70b` | `llama3.3:70b-instruct-q5_K_M` | Medium–large (70B) | Dec 2024 |
| `deepseek-r1:70b` | `deepseek-r1:70b` | Medium–large (70B) | Jun 2025 |
| `mistral:123b` | `mistral-large:123b-instruct-2411-q5_K_M` | Large (123B) | Nov 2024 |

(c)

Figure 15: Experimental settings.




- $\mathcal{V}_1$=($\{v_1\}$, $arc_1$, $\{f_{ad+in}\}$, $arc_5$, $\{\widehat{f}\}$)
- $\mathcal{V}_2$=($\{v_2\}$, $arc_2$, $\{f_{ad+in}\}$, $arc_5$, $\{\widehat{f}\}$)
- $\mathcal{V}_3$=($\{v_3\}$, $arc_3$, $\{f_{ad+in}\}$, $arc_5$, $\{\widehat{f}\}$)
- $\mathcal{V}_4$=($\{v_4\}$, $arc_4$, $\{f_{ad+in}\}$, $arc_5$, $\{\widehat{f}\}$)

Figure 16: $\mathcal{B}$ of the application.

Table 15(c) lists the LLMs considered in our experiments, including one small-to-medium-sized model (`gemma3:27b`), two medium-sized mod-

els (`llama3.3:70b` and `deepseek-r1:70b`), and one large-scale model (`mistral:123b`).

We chose this setting to maximize coverage, assess order bias across a wide range of combinations, and introduce reasonable variations in model sizes. This approach advances state-of-the-art assessments of order bias, which predominantly fix $n$=2 and $m$=1 [47].

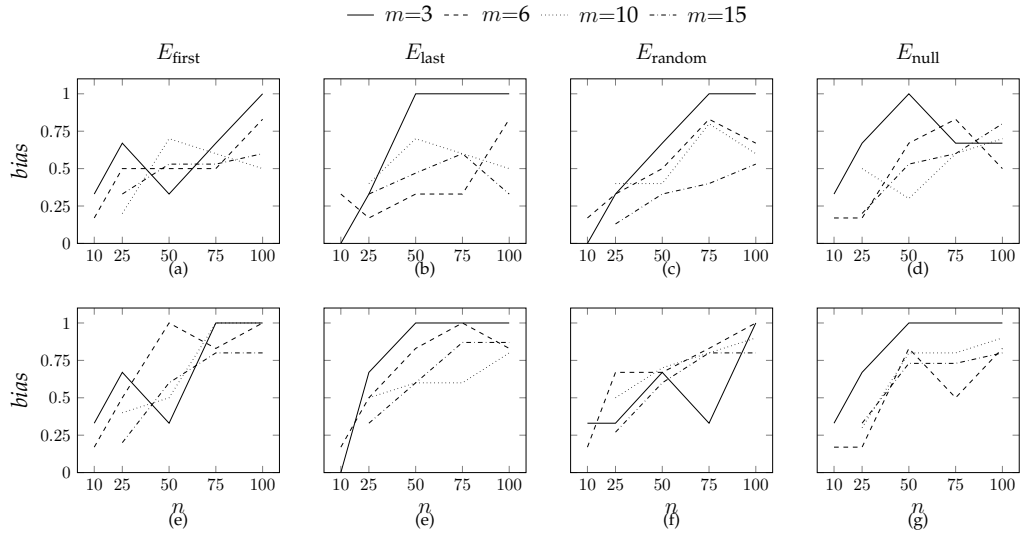



Figure 17: Order bias measured on `gemma3:27b` (a)–(d) and `llama3.3:70b` (e)–(h) varying $n$.

**Results.** Figures 17–18 show our results varying LLMs; Figures 19(a)–(h) show the order bias *bias* varying $n$ and $m$ of the application built using `gemma3:27b`. Figures 20(a)–(h) show the order bias *bias* varying $n$ and $m$ of the application built using `llama3.3:70b`. Figures 21(a)–(h) show the order bias *bias* varying $n$ and $m$ of the application built using `deepseek-r1:70b`. Figures 22(a)–(h) show the order bias *bias* varying $n$ and $m$ of the application built using `mistral:123b`. We note that, when $n \leq m$, no values are plot in Figures 17–18. Figure 17(a)–(d) shows the results of the LLM-based application built on `gemma3:27b`. We can observe that order bias is medium (0.524 on average, with standard deviation $\sigma$=0.254). It increases as $n$ increases (0.298 on average with $n \leq 25$

($\sigma$=0.174), compared to 0.637 on average with $n$>25 ($\sigma$=0.208)), suggesting that the impact of the probe ordering is higher when many probes are available for selection. Finally, Table 7 shows the average *bias* across all settings. We note that the type of example used for in-context learning does not affect the bias. Examples designed to induce order bias (i.e., $E_{\text{first}}$ and $E_{\text{last}}$) result in a bias that is similar (0.521, $\sigma$=0.251) to that observed with random examples or no examples (i.e., $E_{\text{random}}$ and $E_{\text{null}}$) (0.528, $\sigma$=0.259). When $m$ increases, the observed bias tends to decrease (Figures 19 (e)–(h)), which may be due to the reduced number of possible recommendations.

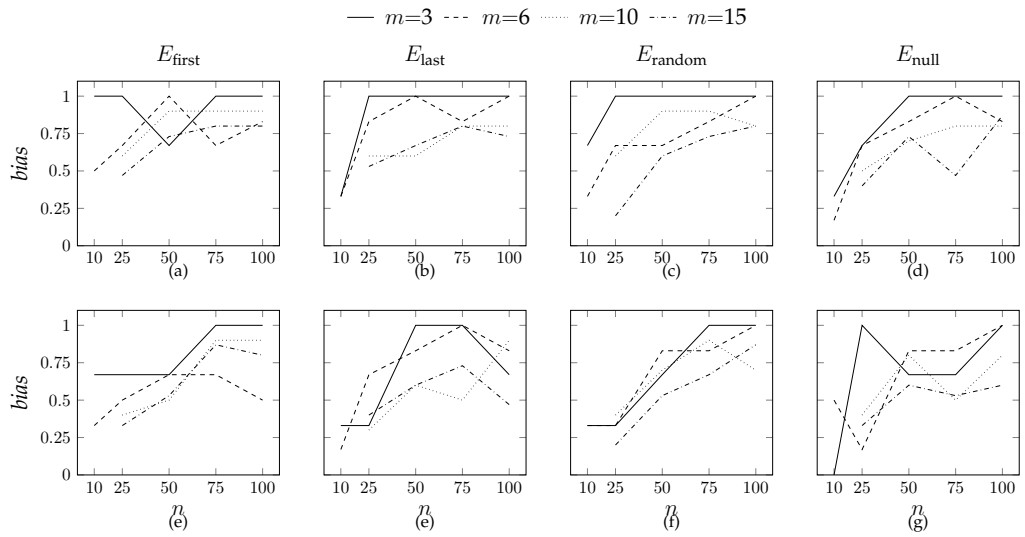



Figure 18: Order bias measured on `deepseek-r1:70b` (a)–(d) and `mistral:123b` (e)–(h) varying $n$.

Figure 17(e)–(h) shows the results of the LLM-based application built on `llama3.3:70b`. The order bias is generally higher (0.661 on average, ($\sigma$=0.279)) compared to `gemma3:27b`, especially as $n$ increases (from 0.362 on average with $n\leq$25 ($\sigma$=0.188) to 0.810 on average with $n$>25 ($\sigma$=0.179)). These results suggest that `llama3.3:70b` is more susceptible to order bias with a larger set of available probes. The type of example used for in-context learning affects the bias, with examples designed to induce order bias resulting in higher bias (0.675 on average with $E_{\text{first}}$ and $E_{\text{last}}$

($\sigma$=0.294) compared to 0.646 on average with $E_{\text{random}}$ and $E_{\text{null}}$ ($\sigma$=0.267)).

When $m$ increases, a decreasing trend in bias similar to `gemma3:27b` is observed (Figures 20(e)–(h)).

Figure 18(a)–(d) shows the results of the LLM-based application built on `deepseek-r1:70b`. The order bias is the highest (0.761 on average ($\sigma$=0.219)), with a noticeable increase when $n>25$ (from 0.586 on average with $n\leq 25$ ($\sigma$=0.249) to 0.848 on average with $n>25$ ($\sigma$=0.137)). The type of example used for in-context learning also significantly influences the bias, with examples designed to induce order bias resulting in higher bias (0.786 on average with $E_{\text{first}}$ and $E_{\text{last}}$ ($\sigma$=0.198), compared to 0.735 on average with $E_{\text{random}}$ and $E_{\text{null}}$ ($\sigma$=0.238)). When $m$ increases, the bias decreases, aligning with trends observed in the other applications (Figures 21(e)–(h)).

Figure 18(e)–(h) shows the results of the LLM-based application built on `mistral:123b`. The order bias is medium (0.640 on average, $\sigma$=0.254), with a strong increase as $n$ increases (from 0.393 on average with $n\leq 25$ ($\sigma$=0.204) to 0.764 on average with $n>25$ ($\sigma$=0.173)). This suggests that `mistral:123b` is affected by the number of available probes. The type of example used for in-context learning marginally affects the bias, with examples designed to induce order bias resulting in slightly higher bias (0.646 on average with $E_{\text{first}}$ and $E_{\text{last}}$ ($\sigma$=0.237), compared to 0.635 on average with $E_{\text{random}}$ and $E_{\text{null}}$ ($\sigma$=0.272)). When $m$ increases, the bias decreases, confirming the trends observed in all applications (Figures 22(e)–(h)).

We also analyzed the execution time of the certification process for each application *app*. The execution time is defined as the sum of the time required to retrieve $\mathit{ans}(\mathit{prompt}_1, \mathit{app})$ and $\mathit{ans}(\mathit{prompt}_2, \mathit{app})$ given $n$, $m$, and $E$. Figure 23 shows the results with $n$=25 to avoid abrupt changes in execution time.

We can observe that, as expected, the execution time increases with $n$ for all applications, that is, the more the available probes the higher the overhead. Execution time exhibits a growing trend with respect to $m$, although the impact is noticeably less significant than the increase observed for $n$. Among the applications, `gemma3:27b` has the lowest execution time (9.671s on average), followed by `mistral:123b` (314.445s on average), `llama3.3:70b` (555.172s on average), and `deepseek-r1:70b`

(903.870s on average). This trend aligns with the parameter size and complexity of the models, where larger models generally require more computational resources and time to process the inputs. Applications built on `mistral:123b` represent a notable exception.

We can finally summarize our results as follows. First, all applications exhibit some degree of order bias, which tends to increase as $n$ increases and decrease as $m$ increases.

Second, the impact of the type of example used for in-context learning varies according to the LLM: from no to moderate impact when using `gemma3:27b` and `mistral:123b`, to high impact when using `llama3.3:70b` and `deepseek-r1:70b`. Finally, the applications where the impact of the type of example is lower are also those where the order bias is lower. Our results show that an LLM with fewer parameters (i.e., `gemma3:27b`) can be less subject to bias than larger LLMs.

We note that the order bias represents the evidence used for certificate awarding. A threshold of maximum tolerated order bias is defined and a certificate is released *iff* the order bias is less that the maximum threshold.[4] In case the threshold is zero, no requirements are imposed on the measured order bias and a certificate is released any time a verification path in $\overline{\mathcal{B}}$ is verified. For instance, the evidence collected by $v_4$ in Figure 16 is the average order bias measured exercising examples of type $E_{\text{null}}$varying $n$ and $m$. In this case, the application built on `mistral:123b` shows `order-bias`=0.640. The awarded certificate contains the certification model $\mathcal{CM}$ and the annotated verification hypergraph $\overline{\mathcal{B}}$ built during the certification process, according to the collected evidence $EV(v)$.

[4]Thresholds are defined using guards $G(v)$ on verification steps.

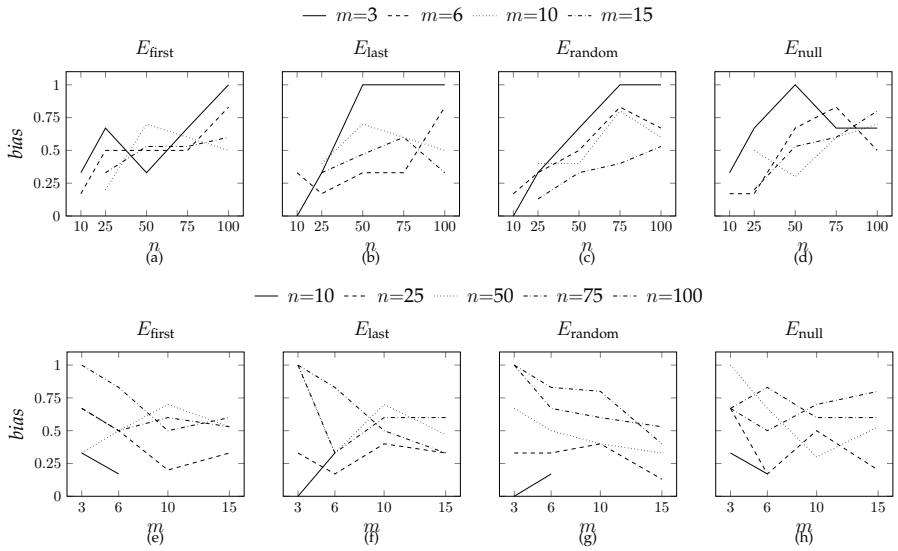



Figure 19: Order bias measured on `gemma3:27b` varying $n$ (a)–(d) and $m$ (e)–(h).

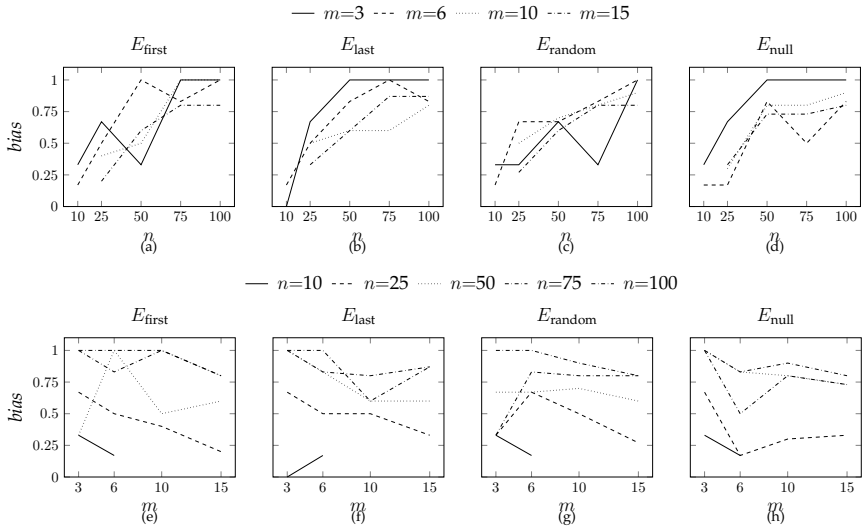



Figure 20: Order bias measured on `llama3.3:70b` varying $n$ (a)–(d) and $m$ (e)–(h).

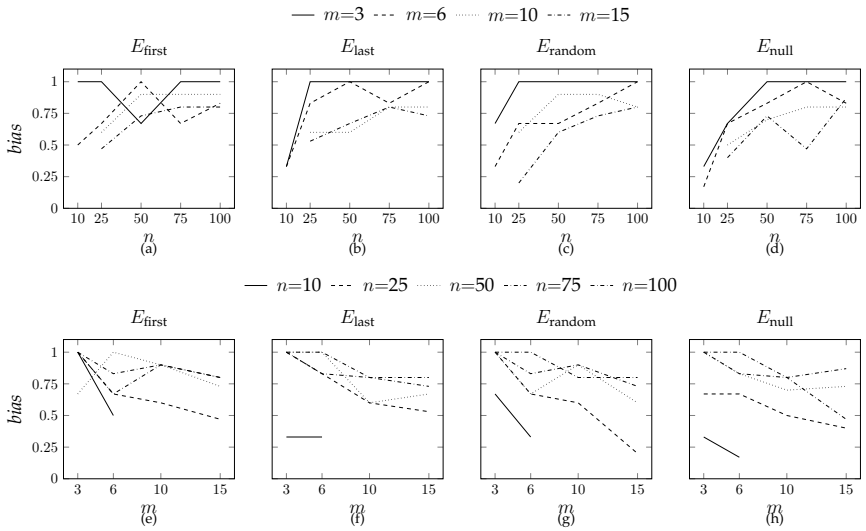



Figure 21: Order bias measured on `deepseek-r1:70b` varying $n$ (a)–(d) and $m$ (e)–(h).

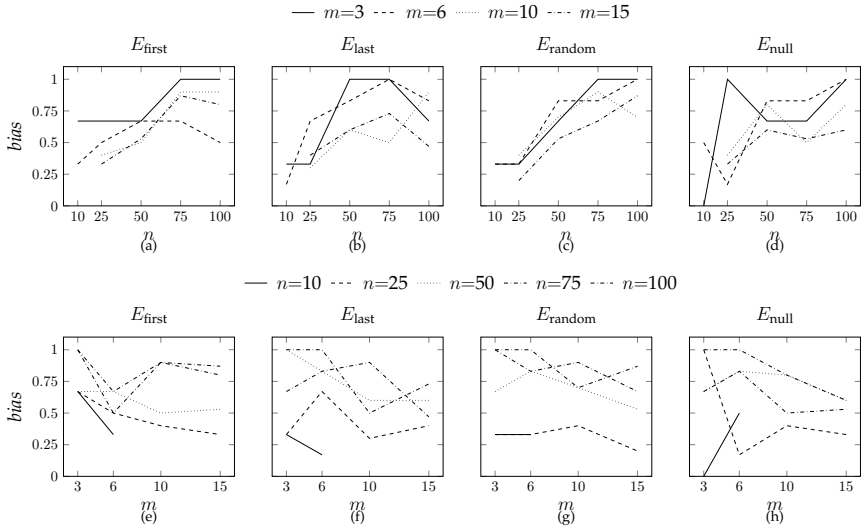



Figure 22: Order bias measured on `mistral:123b` varying $n$ (a)–(d) and $m$ (e)–(h).

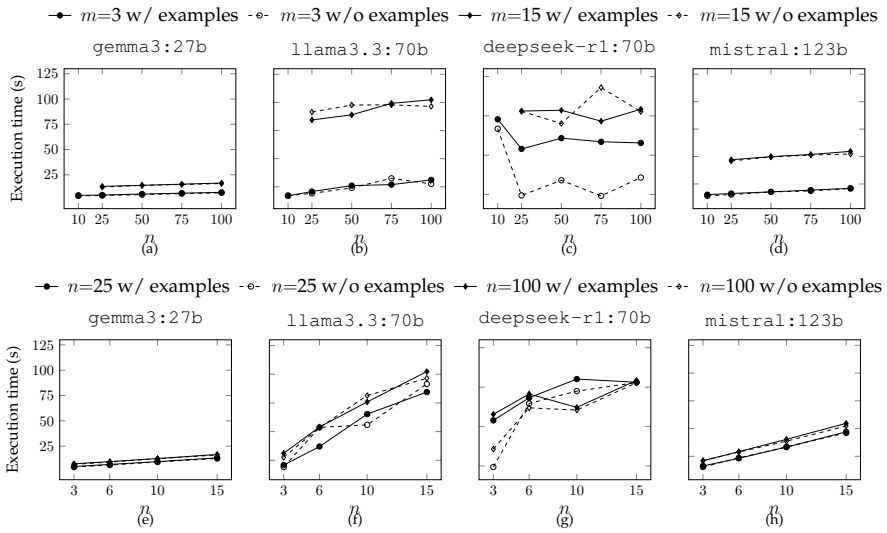



Figure 23: Execution time varying $n$ (a)–(d) and $m$ (e)–(h).

Table 7: Average *bias* for the Type 3 LLM-based application. Standard deviation $\sigma$ is reported in brackets.

| ***app*** | ***bias*** | | | | |
|---|---|---|---|---|---|
| | **Overall** | $E_{\text{first}}$ | $E_{\text{last}}$ | $E_{\text{random}}$ | $E_{\text{null}}$ |
| `gemma3:27b` | 0.524 (0.254) | 0.527 (0.210) | 0.514 (0.293) | 0.505 (0.285) | 0.551 (0.237) |
| `llama3.3:70b` | 0.661 (0.279) | 0.674 (0.302) | 0.676 (0.294) | 0.632 (0.256) | 0.661 (0.285) |
| `deepseek-r1:70b` | 0.761 (0.219) | 0.802 (0.176) | 0.769 (0.222) | 0.761 (0.231) | 0.709 (0.248) |
| `mistral:123b` | 0.640 (0.254) | 0.662 (0.216) | 0.629 (0.263) | 0.646 (0.270) | 0.624 (0.282) |

# Chapter 5. Case Studies

This chapter presents three case studies showing the application of the certification scheme in Chapter 3, each of which focuses on a different aspect of LLM safety and security. The first case study focuses on the implementation of the proposed methodology in the lifecycle of LLMs, examining how they are developed, tested, and deployed in a secure and compliant manner. The second case study focuses on using the proposed methodology to assess the structural integrity of LLMs, examining how their architectural components interact and contribute to their overall behavior. Finally, the third case study focuses on using the proposed methodology to assess the behavioral integrity of LLMs, examining how they respond to different inputs. Each case study is designed to provide insights into the unique challenges and risks associated with LLMs, and to propose novel approaches for addressing these challenges and mitigating these risks. Furthermore we present the experiments conducted to implement and evaluate the case studies presented.

## 5.1 Lifecycle

Classical DevOps pipelines are designed to automate the software development lifecycle, including code integration, testing, deployment, and monitoring. In this context, LLMs are threated only as software artifacts, and the focus is on ensuring that they are developed, tested, and deployed in a consistent and efficient manner. While this approach can be effective, it does not address the unique challenges that LLMs pose, such as the need for continuous monitoring and testing, the potential for model drift, and the need for explainability and interpretability. To address these challenges, MLOps pipelines have been developed, which are designed to manage the entire lifecycle of LLMs. While they are a step forward, they still lack of a full integrated security assurance and certification process. To address this gap, we propose a novel approach called MLCertOps [4], which integrates security assurance and certification into the MLOps pipeline. MLCertOps is designed to ensure that LLMs are developed, tested, and deployed in a secure and compliant manner, while also providing continuous monitoring and testing to ensure that they continue to meet the desired standards over time. In this context we instan-

tiate the proposed hypergraph-based certification approach to define a novel MLCertOps pipeline, and we integrate it into a CI/CD pipeline, providing a comprehensive approach to managing the entire lifecycle of LLMs.

### 5.1.1 Certification Model Definition

MLOps pipelines are commonly developed without taking into consideration security assurance. An inappropriate real-world staging phase can further worsen the situation, bridging non-compliant models directly into final operation environment, whose resolution often involves the manual intervention. To settle the issue, we injected a new assurance phase on each stage of the pipeline: it ensures the compliance at every pipeline stage. The resulting pipeline in Figure 26 is composed of the following stages:

- Split dataset: splits the dataset into training and testing sets.
- Train set evaluation: evaluates the training set for potential issues, such as bias, imbalance, or data quality problems.
- Test set evaluation: evaluates the testing set for potential issues, such as bias, imbalance, or data quality problems.
- Train model: trains the machine learning model using the training set.
- Training evaluation: evaluates the training process and the model's performance.
- Test model: tests the machine learning model using the testing set.
- Testing evaluation: evaluates the testing process and the model's performance.
- Deploy: deploys final software to the operation environment.
- Operation: continuously executes a set of *probes* against the operation environment [7].

As seen before, our approach was formalized as a $\mathcal{CM}$ composed of a pair $(p, \mathcal{B})$, where:

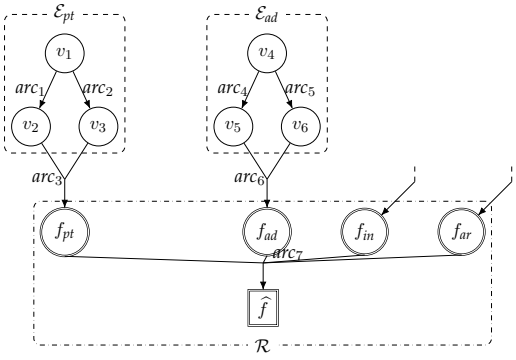
- $p$ is a textual label describing the non-functional property; in this case the MLOps compliance; and
- $\mathcal{B}$ is the hypergraph ($N$, $A$) composed of finite elements that represent the different stages checks of the pipeline.

Figure 24 shows an hypergraph excerpt of the proposed MLCertOps pipeline, focusing on the pre-training and adaptation dimensions. Each node maintains the evidence of the traversal, certifying if a specific check has been passed or not. Finally all the nodes converge into a final node, that represents the evaluation of the sub-hypergraph, and after all the sub-hypergraph evaluations converge to the global evaluation of the pipeline, that is the final output of the hypergraph. For this specific application 4 dimensions were chosen: pre-training $d_{\text{pt}}$, adaptation $d_{\text{ad}}$, inference $d_{\text{in}}$, and artifact $d_{\text{ar}}$. Each dimension represents a critical aspect of a classical LLM lifecycle. In details, the hypergraph is composed of four main nodes and two subgraphs:

- $\mathcal{E}_{\text{pt}}$: this subgraph represents the pre-training evaluation stage of the pipeline, where the training and testing datasets are evaluated for potential issues, such as bias, imbalance, or data quality problems.
- $f_{\text{pt}}$: this node represents the evaluation of the pre-training stage, where the results of the dataset evaluation are analyzed to determine if the dataset is suitable for training.
- $\mathcal{E}_{\text{ad}}$: this subgraph represents the training evaluation stage of the pipeline, where the training process and the model's performance are evaluated.
- $f_{\text{ad}}$: this node represents the evaluation of the training stage, where the results of the training evaluation are analyzed to determine if the model is compliant with the desired performance and security standards.
- $f_{\text{in}}$: this node represents the inference evaluation stage of the pipeline, where the deployed model is monitored for potential issues, such as model drift or performance degradation, through the execution of probes.
- $f_{\text{ar}}$: this node represents the artifact evaluation stage of the pipeline, where the artifacts produced by the LLM are evaluated for potential

issues, such as biases, inconsistencies, or other security concerns.

The hypergraph can be easily extended to model more complex pipelines, by adding more nodes and edges, and by defining more complex evaluation rules for the different stages of the pipeline.




| Verification steps | |
|---|---|
| $v_1$ | (dataset, splitting) |
| $v_2$ | (train dataset, fairness evaluation) |
| $v_3$ | (evaluation dataset, fairness evaluation) |
| $v_4$ | (model, training) |
| $v_5$ | (model, training algorithm evaluation) |
| $v_6$ | (model, testing algorithm evaluation) |

Figure 24: MLCertOps Hypergraph (excerpt)

### 5.1.2 Implementation

After the hypergraph definition, a novel MLCertOps certification process is integrated to collect and evaluate the evidence. The key components of our MLCertOps process include:

- `Moon Cloud`: an assurance platform to assess CI/CD pipelines. It implements *probes* (assurance controls) whose results, called *evidence*, are evaluated by compliance evaluation rules. For instance, a probe can search for vulnerabilities on a given target (e.g., source code) while the evaluation rule filters the vulnerabilities by severity. In case of severity higher than a threshold, the *probe* returns a negative output.
- `Injector`: a set of custom-built programs that provides the assurance capabilities to the DevOps process.
- `Pipeline`: the real target of the probes execution, hosted on a CI/CD provider.

Figure 25 shows the proposed assurance process. First, it first configures the evaluation rule on the Moon Cloud platform (1). Next, the Injector in-

struments the pipeline (2) generating the instrumented pipeline (3). The instrumented pipeline is executed and produces an artifact for every stage that needs to be assessed (4). Finally, the probe(s), triggered by the instrumented pipeline (5), connects to the CI/CD provider, analyzes the necessary artifacts (6), and writes the collected evidence (7). Evidence is then evaluated according to the evaluation rule returning a positive or negative output. The assurance execution report is finally available both on the pipeline stage and on the Moon Cloud dashboard.

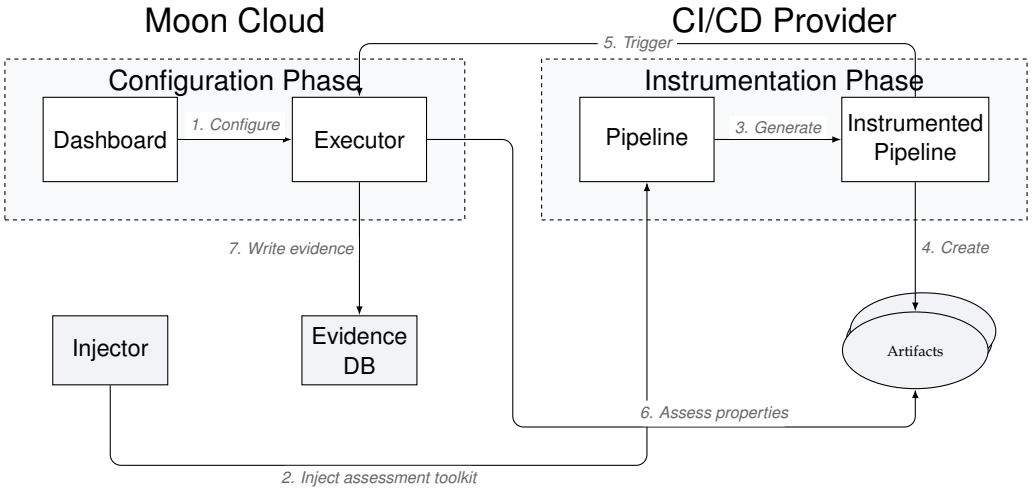



Figure 25: MLCertOps Assurance Process Integration

The proposed MLCertOps pipeline provides a base approach to managing the entire lifecycle of LLMs, more advanced pipelines can be designed by adding more stages, such as data preprocessing, feature engineering, or model selection, and by integrating more advanced assurance techniques, such as explainability and interpretability. By integrating security assurance and certification into the MLOps pipeline, MLCertOps provides a comprehensive approach to managing the entire lifecycle of LLMs, ensuring that they are developed, tested, and deployed in a secure and compliant manner.

### 5.1.3 Experimental Evaluation

In the context of LLMs lifecycle, we implemented the proposed MLCertOps pipeline and integrated it into a CI/CD pipeline. The pipeline is

designed to manage the entire lifecycle of LLMs, ensuring that they are developed, tested, and deployed in a secure and compliant manner. The pipeline is composed of several stages, including data preparation, dataset evaluation and operation evaluation. Each stage is designed to ensure that the model is developed and deployed in a secure and compliant manner, while also providing continuous monitoring and testing to ensure that it continues to meet the desired standards over time.

#### 5.1.3.1 Pipeline Integration

Figure 26 shows the realized pipeline fully integrated into GitHub workflows. In details, it shows a pipeline composed of four stages:

- Data preparation: in this stage the dataset is prepared for training, dividing it into training, validation, and testing sets.
- Training Dataset Evaluation: in this stage the training dataset is evaluated for potential issues, such as bias, imbalance, or data quality problems.
- Testing Dataset Evaluation: in this stage the testing dataset is evaluated for potential issues, such as bias, imbalance, or data quality problems.
- Training: in this stage the model is trained on the prepared dataset.
- Training Evaluation: in this stage the training algorithm is evaluated.
- Testing: in this stage the trained model is evaluated on the testing set.
- Testing Evaluation: in this stage the testing results are evaluated for potential issues, such as bias, imbalance, or data quality problems.

The training and testing evaluation stages are instrumented with probes that evaluate the training and testing sets for potential issues, such as bias, imbalance, or data quality problems. The operation stage is also instrumented with probes that continuously monitor the deployed model for potential issues, such as model drift or performance degradation. By integrating security assurance and certification into the MLOps pipeline, MLCertOps provides a comprehensive approach to managing the entire

lifecycle of LLMs, ensuring that they are developed, tested, and deployed in a secure and compliant manner.

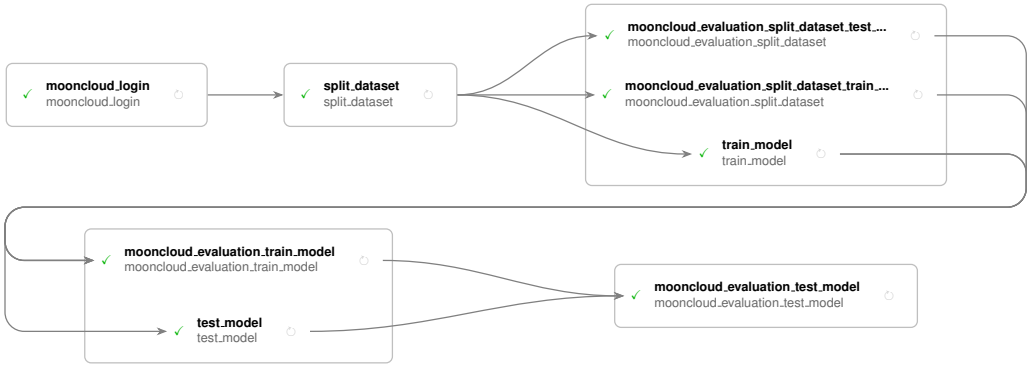



Figure 26: An MLCertOps pipeline

The YAML definition of the pipeline activates every time a new commit is pushed to the repository, ensuring that the model is continuously evaluated and monitored for potential issues. Each stage is defined as a separate job, allowing for parallel execution and efficient resource utilization. Furthermore, each stage is modeled as a dependency of the previous one, ensuring that the pipeline is executed in a logical and sequential manner. Each stage that needs to be evaluated produces a report that is stored as an artifact of the pipeline, that will be accessed by the next stage evaluation. This design allows us to easily integrate into existing and future pipelines, thanks to the modularity and flexibility of the pipeline definition, the end user only have to specify the evaluation rule ID, if the pipeline should stop on non-compliant results, and the credentials to the assessment platform.

On the other part, the injection CI is integrated in through a separate CI definition that is injected into the repository pipeline each time a new commit is pushed.

All the code is available at: `https://github.com/SESARLab/mooncloud_sample-ml-apps` and `https://github.com/SESARLab/Moon-Cloud-CI-Injection`.

#### 5.1.3.2 Pipeline Instantiation

To evaluate the proposed approach we instantiated the MLCertOps pipeline on a sample machine learning application. The pipeline was configured to evaluate the training and testing datasets, as well as the training and testing processes, and to ensure that the model meets the desired performance and security standards. In details the following thresholds were set:

- Dataset splitting: the training and testing datasets should be split in a 50/50 or 60/40 ratio.
- Training dataset fairness: the training dataset should have an ICA normalized result of at least 0.6.
- Testing dataset fairness: the testing dataset should have an ICA normalized result of at least 0.6.
- Model training: the training process should take no more than 15 seconds.
- Model training algorithm evaluation: the training process should use the Adam optimizer and run for no more than 10 epochs.
- Model testing algorithm evaluation: the testing process should shuffle the testing dataset and return normalized results.

#### 5.1.3.3 Results and Discussion

Table 8 shows the results of the instantiated MLCertOps pipeline. The results show that the pipeline was able to successfully evaluate the training and testing datasets, as well as the training and testing processes, and that the model was able to meet the desired performance and security standards.

| Node | Result |
|---|---|
| dataset splitting | {"train_percentage": "50", "test_percentage": "50"} |
| train dataset fairness | {"ICA_normalized_result": "0.7346237"} |
| evaluation dataset fairness | {"ICA_normalized_result": "0.6328961"} |
| model training | {"training_time": "10.23s"} |
| model training algorithm evaluation | {"epochs": "10", "optimizer": "Adam"} |
| model testing algorithm evaluation | {"shuffle": "on", "results_normalization": "true"} |

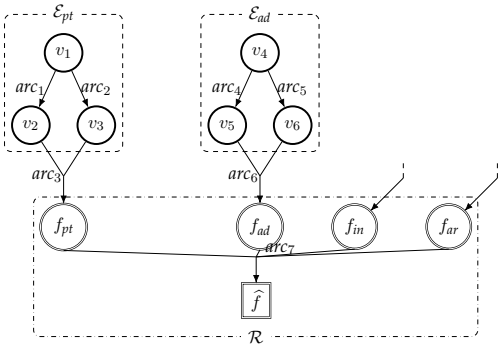
Table 8: MLCertOps Pipeline Results

This approach is intrinsically scalable and adaptable into any GitOps-compliant environment. The pipeline is designed to be flexible and adaptable, allowing for the integration of additional stages or probes as needed to address specific requirements or concerns. The integration into LLMs development pipelines is particularly important, as it can help to mitigate the risks associated with the development and deployment of LLMs, such as misuse, misalignment, mistakes, and structural risks. By continuously evaluating and monitoring the model throughout its lifecycle, we can ensure that it is developed and deployed in a secure and compliant manner, while also providing valuable insights into its behavior and performance. More advance controls can be implemented, such explainability, interpretability, and robustness testing, to further enhance the security and reliability of the model. Furthermore, the process is structured in a fully GitOps-compliant way, allowing for easy integration into existing and future pipelines, and providing a clear and transparent view of the model's development and deployment process. Overall, this approach provides a comprehensive framework for managing the lifecycle of LLMs, ensuring that they are developed and deployed in a secure and compliant manner, while also providing valuable insights into their behavior and performance.

Following the defined $\mathcal{CM}$, the final results can be formalized as a certificate of MLOps compliance composed of the following elements:

- $\mathcal{CM}$: the certification model defined in this section, which includes the property label and the hypergraph representing the evaluation process.
- $\overline{\mathcal{B}}$: the set of behaviors that have been verified according to the certification process, including the evidence collected during the evaluation process, such as the results of the training and testing datasets, the training and testing processes, and the evaluation of the model's performance and security standards.

The evidence collected during the evaluation process can be formalized as a set of nodes in the hypergraph, which represent the different steps of the evaluation process. Figure 27 shows the nodes visited during the evaluation process.




| Verification steps | |
|---|---|
| $EV(v_1)$ | *dataset splitting = success* |
| $EV(v_2)$ | *train dataset fairness evaluation = success* |
| $EV(v_3)$ | *evaluation dataset fairness evaluation = success* |
| $EV(v_4)$ | *model training = success* |
| $EV(v_5)$ | *model training algorithm evaluation = success* |
| $EV(v_6)$ | *model testing algorithm evaluation = success* |

Figure 27: MLOps hypergraph evidence collected during the evaluation process

Note that, in this case, all nodes in the hypergraph were visited during the evaluation process, which indicates that all steps of the evaluation process were completed and all evidence was collected. The results are positive for all the nodes, which indicates that the model is compliant with the desired performance and security standards.

## 5.2 Structural Integrity

Machine learning models are becoming more and more complex, and their behavior can be difficult to understand and predict. This can lead to a range of issues, such as misuse, misalignment, mistakes, and structural risks. To address these issues, we started with some insights from mechanistic interpretability, which is a field that aims to understand the inner workings of machine learning models by analyzing their structure and behavior, and then developed a set of testing techniques that can be used to evaluate the structural properties of machine learning models, such as analyzing the structure of the model, such as its architecture, parameters, and training data, to identify potential issues or risks. Then we proposed a novel approach to assess LLMs structural integrity, leveraging these insights.

### 5.2.1 Base Idea

The base idea of our approach is to analyze the internal structure of LLMs, in particular the activations of individual neurons, to assess the structural integrity of the model. In particular, observing the activations of neurons in response to inputs, we can identify patterns and relationships that can help us understand how the model processes information and how different layers contribute to the final output. Then we can compare the activation distributions to assess whether two versions of a model are structurally integral or not. In details, we followed 4 basic steps:

- Hook registration: For this phase we exploited peculiar callback APIs of state-of-the-art AI frameworks, in particular the forward hooks, which allow us to register a callback function that is executed during the forward pass of the model. This enables us to capture the activations of individual neurons and analyze their behavior in response to different inputs. In detail, when an input is passed through the model, the forward hook is triggered at each layer, allowing us to capture inputs and ouputs of each layer. This information is crucial for understanding how the model processes information and how different layers contribute to the final output. By analyzing the activations of individual neurons, we can gain insights into the internal workings of the model and identify potential issues or risks.

- Querying: To maximize the coverage of our testing, we need to identify a comprehensive set of queries that can be used to evaluate the model's behavior. This involves selecting a diverse range of inputs that can trigger different responses from the model, including edge cases and potential failure modes. By systematically querying the model with a wide variety of inputs, we can gain a better understanding of its behavior and identify any potential issues or risks that may arise in real-world applications. This step is crucial for ensuring that our testing is thorough and effective in evaluating the model's performance and robustness.
- Distribution finding: Once we have collected the activations from the model in response to our queries, we need to analyze this data to find common patterns and identify the distribution of responses. To identify the distribution the mean activation of each layer is calculated.
- Integrity verification: the distribution in the previous step is compared with the respective fine-tuned model's one. The Jensen-Shannon divergence is used to certify whether the model is structurally integral or not. A threshold of 0.1 was defined, if the JS divergence is under that we can consider the model integral, otherwise not. Formally, let $Q^{(l)}$ be the activation distribution obtained from a second model evaluated on the same query set $X$. To quantify the similarity between the reference behavioural profile $P^{(l)}$ and the observed one, we compute the Jensen-Shannon Divergence (JSD):

$$JSD(P^{(l)} \parallel Q^{(l)}) = \frac{1}{2} D_{\mathrm{KL}}(P^{(l)} \parallel M) + \frac{1}{2} D_{\mathrm{KL}}(Q^{(l)} \parallel M) \tag{4}$$

where

$$M = \frac{1}{2}(P^{(l)} + Q^{(l)}) \tag{5}$$

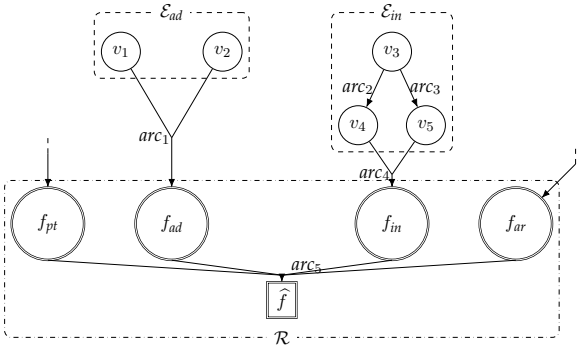
and $D_{\mathrm{KL}}$ denotes the Kullback-Leibler divergence.

### 5.2.2 Certification Model Definition

As seen before, we formalized the structural integrity assessment using the proposed certification scheme in Chapter 3.

The hypergraph is then instantiated, the nodes represent the different models, base and fine-tuned model, the evidence collected during the evaluation, such as the activations of individual neurons, the distribution of responses, and the integrity verification. For this specific application 4 dimensions were chosen: pre-training $d_{\text{pt}}$, adaptation $d_{\text{ad}}$, inference $d_{\text{in}}$, and artifact $d_{\text{ar}}$. Figure 28 shows an hypergraph excerpt which focuses on the adaptation and inference dimensions. Each dimension represents a critical step of LLM internal structure. In details, the hypergraph shows four main nodes and two subgraphs:

- $f_{\text{pt}}$: this node represents the pre-training phase, including the base model, the dataset used for pre-training, and the fine-tuning hyper-parameters
- $\mathcal{E}_{\text{ad}}$: this group of nodes represents the subgraph to produce the base and fine-tuned model
- $f_{\text{ad}}$: this node represents the models that are being evaluated, including the base model and the fine-tuned model
- $\mathcal{E}_{\text{in}}$: this group of nodes represents the subgraph to collect the evidence during the evaluation process, such as the activations of individual neurons and the distribution of responses
- $f_{\text{in}}$: this node represents the integrity verification metrics, such as the Jensen-Shannon divergence, that are used to evaluate the structural integrity of the model.
- $f_{\text{ar}}$: this node represents the artifact evaluation, including LLM responses to the prompts and the expected responses.




| Verification steps | |
|---|---|
| $v_1$ | (base model, download) |
| $v_2$ | (base model, fine-tuning) |
| $v_3$ | (prompts, category dataset creation) |
| $v_4$ | (models, internals evaluation) |
| $v_5$ | (models, JS distance evaluation) |

Figure 28: Structural Hypergraph (excerpt)

### 5.2.3 Experimental Evaluation

LLM internals are a still complex aspect of LLMs, and they are often considered a black-box, due to the lack of interpretability and explainability of their inner workings. However, understanding the internals of LLMs is crucial for evaluating their safety and security, as it allows us to identify potential vulnerabilities and risks that may arise from their architecture and design. In this case study, the focus is on the structural integrity of LLMs, examining how their architectural components interact and contribute to their overall behavior, in particular we focused on evaluating the integrity of the model's internals output distribution, by comparing the output distribution of a fine-tuned model with the output distribution of its base model on a set of prompts.
First we started obtaining the layer-wise output activations of the base model, figure 29 shows the `gpt-oss-20b` model output results, it's possible to notice that the activations are consistently much higher in the middle layers of the model, while the activations in the first and last layers are significantly lower, this pattern is consistent across different prompts, and it suggests that the middle layers of the model are more active and contribute more to the overall output of the model, while the first and last layers are less active and contribute less to the overall output of the model.

Then we started querying the model to define a common pattern in the layers activations, the distribution was assessed by calculating the mean of the activations across the layers on different prompts. Figure 29 shows

the mean of the activations across the layers, on the x-axis the layer number and on the y-axis the mean activation value, we can see that the distribution shows a peculiar pattern relative to `gpt-oss-20b`, where, as before, it's possible to notice the difference between the middle layers and the first and last layers.

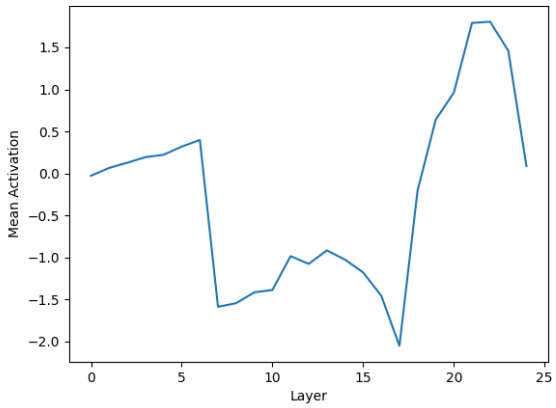



Figure 29: Global distribution of the activations across the layers of the model

#### 5.2.3.1 Fine Tuning

To evaluate the internals integrity of LLMs, we have fine-tuned a base model (`gpt-oss-20b`) through Unsloth Studio, using QLoRA, on a generic prompts dataset(HuggingFaceH4/Multilingual-Thinking). Table 9 shows the used fine-tuning configuration, with the training dataset, the base model, and the training hyperparameters.

The architecture used for the fine-tuning is composed of 1 Nvidia A40 GPU and 3 Nvidia L40S running inside a multi node Kubernetes cluster with multi-GPU configuration. The concurrent multi-GPU training is enabled by the use of DeepSpeed, which is a deep learning optimization library that provides efficient training and inference for large-scale models. In particular DeepSpeed's ZeRO (Zero Redundancy Optimizer) 3 was used. The fine tuning process took around 10 minutes, with a batch size of 1, a learning rate of 2e-4, and a maximum of 30 training steps, with a warmup of 5 steps.

| Hyperparameters | | LoRA | |
|---|---|---|---|
| **Parameter** | **Value** | **Parameter** | **Value** |
| Epochs | 0 | Rank | 8 |
| Batch size | 1 | Alpha | 16 |
| Learning rate | 0.0002 | Dropout | 0 |
| Optimizer | AdamW 8-bit | Variant | lora |
| Max steps | 30 | | |
| Context length | 1024 | | |
| Warmup steps | 5 | | |

Table 9: Unsloth Studio fine-tuning configuration

All the used models and adapters can be found at: `https://huggingface.co/alexdellabruna/gpt-oss-20b-qlora` and `https://huggingface.co/alexdellabruna/gpt-oss-20b-qlora-adapter`.

#### 5.2.3.2 Fine-Tuned Model Internals Evaluation

After that we obtained the layer-wise activation outputs of the fine-tuned model on the same set of prompts. Figure 30 shows the mean of the activations across the layers, we can see that the distribution shows a similar pattern to `gpt-oss-20b`.

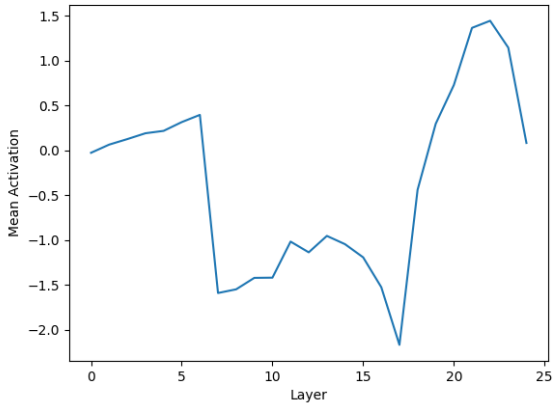



Figure 30: Global distribution of the activations across the layers of the fine-tuned model

All the code is available at: `https://github.com/alexdellabruna/llm-internals-cert`.

#### 5.2.3.3 Results and Discussion

Finally we compared the two distributions and calculated the Jensen-Shannon divergence. Figure 31 shows the comparison between the two distributions, in blue the base model and in orange the fine-tuned model. We can observe that the two distributions are very similar, with the same pattern of the middle layers having higher activations than the first and last layers. The calculated Jensen-Shannon divergence is 0.06, which shows a light modification in the distribution, but it is still relatively low, suggesting that the fine-tuning process did not significantly alter the overall architecture and internal behavior of the model. The threshold for the integrity evaluation was set to 0.1, the calculated divergence is below the threshold, which suggests that the model's internals integrity is preserved after the fine-tuning process and we can consider its internal behavior integral.

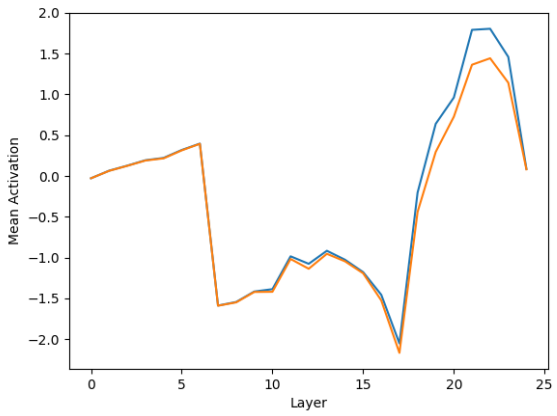
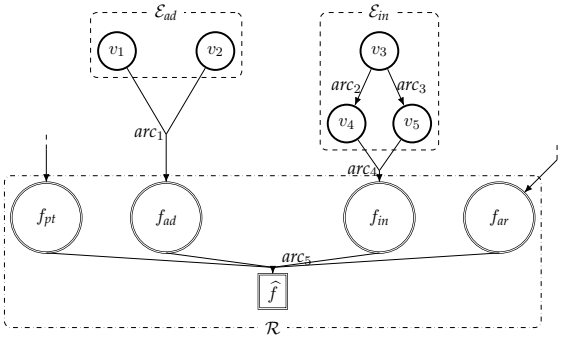



Figure 31: Comparison between the distributions of the base model and the fine-tuned model

Following the defined $\mathcal{CM}$, the final results can be formalized as a certificate of structural integrity composed of the following elements:

- $\mathcal{CM}$: the certification model defined in this section, which includes the property label and the hypergraph representing the evaluation process.
- $\overline{\mathcal{B}}$: the set of behaviors that have been verified according to the certification process, including the evidence collected during the evaluation process, such as the fine-tuning, the internals activation distributions, and the integrity scores.

The evidence collected during the evaluation process can be formalized as a set of nodes in the hypergraph, which represent the different steps of the evaluation process. Figure 32 shows the nodes visited during the evaluation process.




| Verification steps | |
|---|---|
| $EV(v_1)$ | $download = success$ |
| $EV(v_2)$ | $fine-tuning = success$ |
| $EV(v_3)$ | $category\ dataset\ creation = success$ |
| $EV(v_4)$ | $internals\ evaluation = success$ |
| $EV(v_5)$ | $JS\ distance\ evaluation = success$ |

Figure 32: Structural integrity evidence collected during the evaluation process

Note that, in this case, all nodes in the hypergraph were visited during the evaluation process, which indicates that all steps of the evaluation process were completed and all evidence was collected. The results of all nodes is positive and the final integrity score is below the defined threshold, which suggests that the model's internals integrity is preserved after the fine-tuning process and we can consider its internal behavior integral.

## 5.3 Behavioral Integrity

LLM responses from an outside perspective can seem a really complex and unpredictable phenomenon, but from an inside perspective, they are the result of a complex interplay of various factors, such as the model's architecture, training data, and the specific input prompt. To reduce the complexity we can analyze how they are generated and then we can design specific techniques to evaluate their quality and reliability. In this context, we propose a novel framework for LLM certification based on golden datasets.

### 5.3.1 Certification Model Definition

As seen before, we formalized the behavioral integrity assessment using the proposed certification scheme in Chapter 3.

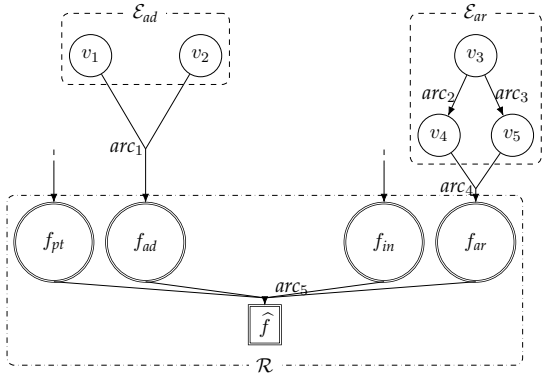
The hypergraph is then instantiated, the nodes represent the different evidence collected during the evaluation process, such as the fine tuning, the golden dataset distributions, and the integrity scores. In details, the hypergraph shows four main nodes and two subgraphs

For this specific application 4 dimensions were chosen: pre-training $d_{\text{pt}}$, adaptation $d_{\text{ad}}$, inference $d_{\text{in}}$, and artifact $d_{\text{ar}}$. Figure 33 shows an hypergraph excerpt which focuses on the adaptation and artifact dimensions. Each dimension represents a critical step of LLM behavior evaluation. In details, the hypergraph shows four main nodes and two subgraphs:

- $f_{\text{pt}}$: this node represents the pre-training phase, including the base model, the dataset used for pre-training, and the fine-tuning hyperparameters
- $\mathcal{E}_{\text{ad}}$: this group of nodes represents the subgraph to produce the base and fine-tuned model
- $f_{\text{ad}}$: this node represents the models that are being evaluated, including the base model and the fine-tuned model
- $f_{\text{in}}$: this node represents the inference phase, including the prompts used for evaluation and the expected responses.
- $\mathcal{E}_{\text{ar}}$: this group of nodes represents the subgraph to collect the evidence during the evaluation process, such as the distribution of re-

sponses to the golden dataset prompts on the base and fine-tuned model

- $f_{\text{ar}}$: this node represents the integrity verification metrics, such as the Jensen-Shannon divergence, that are used to evaluate the behavioral integrity of the model.




| Verification steps | |
|---|---|
| $v_1$ | (base model, download) |
| $v_2$ | (base model, fine-tuning) |
| $v_3$ | (prompts, golden dataset creation) |
| $v_4$ | (models, behavioral evaluation) |
| $v_5$ | (models, JS distance evaluation) |

Figure 33: Behavioral integrity hypergraph (excerpt)

### 5.3.2 Golden Dataset

The evaluation of the behavioral integrity is defined as the consistency between two models on a set of metrics with a given golden dataset. The golden dataset is defined on crucial frontier data points where the behavior must be as similar as possible. If the two models demonstrate congruence in behavior on the golden dataset, it can be concluded that they are integral. In particular, with LLMs we have evaluated a base model and a fine-tuning, obtained through QLoRA application, of the same base model on a generic prompts dataset, and then we have compared their behavior on a golden dataset. To create the golden we selected a subset of prompts that are particularly challenging for the model, and that are representative of the specific use case we want to evaluate.

The Jensen-Shannon (JS) divergence is considered as behavioral metric, both in terms of the entire dataset category and in terms of specific subset. For JS scores a threshold $thr$ is defined in terms of the distance between the measurements, before which the two models $M_1$ and $M_2$ are deemed to be integral. Equation 6 shows a formal definition of the behavioral

integrity, with $G$ being the golden dataset and $G_i$ the part of the golden dataset set with the $i$ category.

$$c_{\text{behavior integ.}}(M_1, M_2, G) =$$
$$\text{let} \begin{cases} js_{\text{tot}} = |JS(M_1, G) - JS(M_2, G)| \\ js_i = |JS(M_1, G_i) - JS(M_2, G_i)| \end{cases} \quad \text{in } \left(js_{\text{tot}} \leq \text{thr}\right) \wedge \bigwedge_{i \in \text{categories}} \left(js_i \leq \text{thr}_i\right) \tag{6}$$

A threshold of 0.1 was defined to decided whether a model can be considered integral or not, if the JS divergence is below 0.1 we can conclude that the model shows an high degree of congruence, so we can consider it integral, otherwise, a JS divergence over 0.1 shows a modification of the behavioral integrity, so we can consider the model not integral.

### 5.3.3 Experimental Evaluation

Behavioral integrity is a critical aspect of LLMs, as it refers to the consistency and reliability of their responses to different inputs. Evaluating the behavioral integrity of LLMs is essential for ensuring that they are safe and secure to use, as it allows us to identify potential issues or risks that may arise from their behavior. In this case study, the focus is on evaluating the behavioral integrity of LLMs by comparing the output distribution of a fine-tuned model with the output distribution of its base model on a set of prompts. By analyzing the differences in the output distributions, we can gain insights into the behavior of the fine-tuned model and identify any potential issues or risks that may arise from its behavior.

#### 5.3.3.1 Golden Dataset

A golden dataset is a carefully curated set of data points that are used as a reference for evaluating the performance of a machine learning model. The golden dataset is created by selecting a subset of data points that are particularly challenging for the model, and that are representative of the specific use case we want to evaluate. By comparing the model's performance on the golden dataset to its performance on other datasets, we can gain insights into its behavior and identify potential issues or risks. In the context of LLMs, when referring to golden datasets, we are referring to a specific set of prompts that are designed to test the model's ability

to generate accurate and relevant responses. In this work, two main categories of prompts were selected for the golden dataset: i) mathematical reasoning prompts, which require the model to perform complex calculations and logical reasoning, and ii) logical reasoning prompts, which require the model to understand and apply logical principles to generate responses. By evaluating the model's performance on these specific categories of prompts, we can gain insights into its ability to handle complex reasoning tasks and identify any potential weaknesses or limitations in its behavior.

#### 5.3.3.2 Fine Tuning

For the fine-tuning we followed the same process described in 5.2.3.1, using the same architecture and configuration.

#### 5.3.3.3 G-Eval

The G-Eval methodology [37] provides a framework for evaluating the performance of natural language generation (NLG) models. It leverages the capabilities of large language models (LLMs) to assess the quality of generated text by comparing it to reference responses. The methodology includes several key components, such as prompt design, response generation, and evaluation metrics, which work together to provide a comprehensive assessment of the model's performance. In particular, G-Eval permits to evaluate the semantic similarity between the model's responses and the expected responses, which is crucial for assessing the behavioral integrity of LLMs. By using G-Eval, we can gain insights into how well the base and fine-tuned model are able to generate responses that are consistent with the expected responses, and identify any potential issues or risks that may arise from its behavior.

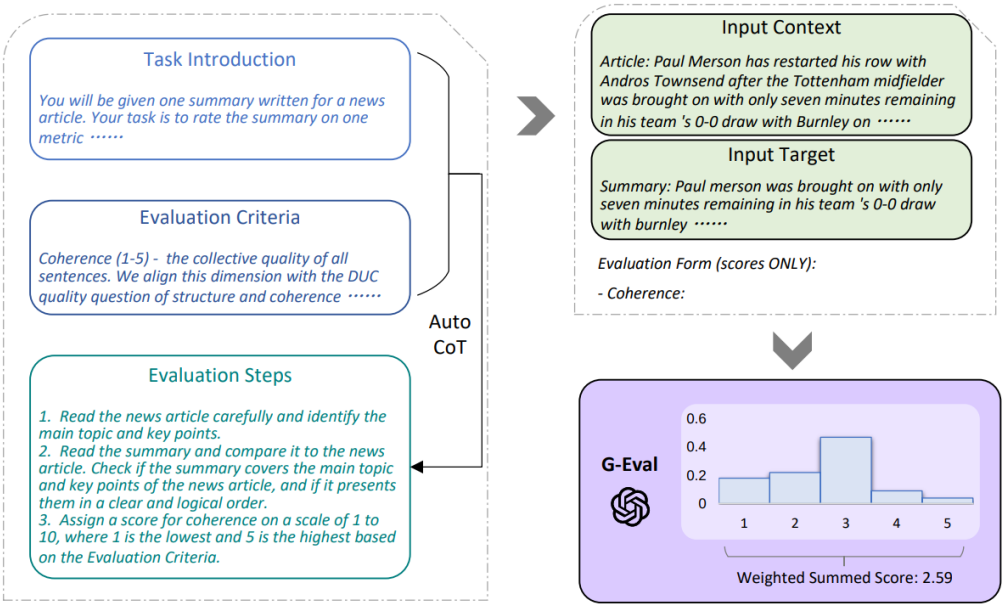



Figure 34: Overview of the G-Eval evaluation process [37]

#### 5.3.3.4 Behavioral Integrity Evaluation

To evaluate the behavioral integrity of the fine-tuned model, we compared its output distribution with that of the base model on the golden dataset. The output distribution was assessed by calculating the similarity between the model's responses to the prompts in the golden dataset and the expected responses. In details the similarity was calculated using the DeepEval library, in particular using the G-Eval methodology with specific prompts to verify the semantic similarity between the model's responses and the expected responses. The results of the evaluation were then compared using the Jensen-Shannon divergence. The threshold for the integrity evaluation was set to 0.1, which means that if the calculated divergence is below this threshold, we can consider the model's behavior to be integral, while if it is above the threshold, we can consider the model's behavior to be non-integral. Figure 35 shows a complete overview of the behavioral integrity evaluation process, starting from the fine-tuning of the model, to the evaluation of its behavior on the golden dataset, and finally to the comparison of the output distributions and the calculation of the Jensen-Shannon divergence. Formally the evaluation

process can be defined as the explosion of the hypergraph in Figure 33, focusing on the nodes $v_4$ and $v_5$, which represent the evaluation of the models on the golden dataset and the evaluation node $f_{\text{Metrics}}$, which represents the calculation of the Jensen-Shannon divergence.

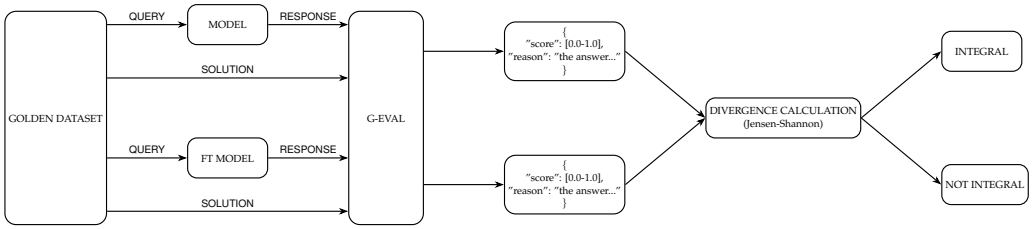
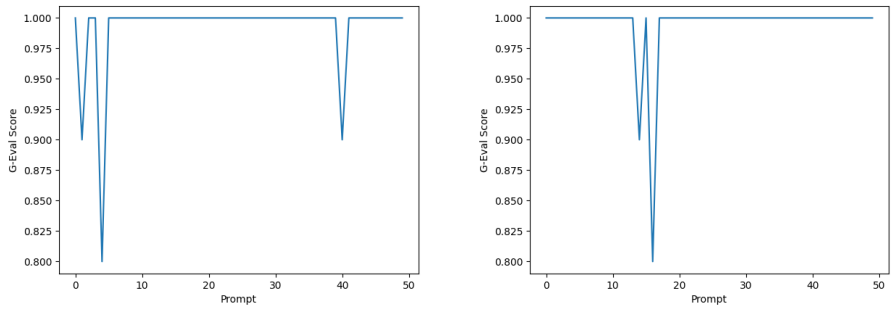



Figure 35: Overview of the behavioral integrity evaluation process

#### 5.3.3.5 Base Model Behavior

We evaluated the behavior of the base model by comparing its output distribution on the golden dataset with the expected responses. The results of this evaluation showed that the base model demonstrates a high level of knowledge and reasoning capabilities on both mathematical and logical reasoning prompts, with a minimum G-Eval score of 0.8 on both mathematical and logical reasoning prompts, which suggests that the model is able to generate responses that are consistent with the expected behavior on these categories of prompts. This indicates that the base model has a good understanding of mathematical and logical concepts, and is able to apply this knowledge to generate accurate and relevant responses to the prompts in the golden dataset.

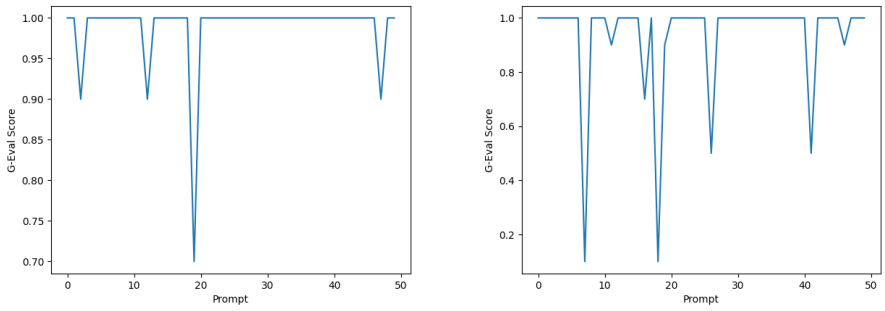



(a) Base model behavior evaluation results on math prompts

(b) Base model behavior evaluation results on logic prompts

#### 5.3.3.6 Fine-Tuned Model Behavior

Then we evaluated the behavior of the fine-tuned model by comparing its output distribution on the golden dataset with the expected responses. The results of this evaluation showed that the fine-tuned model demonstrates a slightly lower level of knowledge and reasoning capabilities on mathematical prompts, with a minimum G-Eval score of 0.7, this suggests that while the fine-tuning process was able to preserve the model's behavioral integrity on mathematical reasoning tasks, it may have introduced some minor changes in its behavior on this category of prompts. On the other hand the fine-tuned model demonstrates a particularly lower level of knowledge and reasoning capabilities on logical prompts, with a minimum G-Eval score of 0.1, this suggests that the fine-tuning process may have introduced significant changes in the model's behavior on logical reasoning tasks, which could be a potential issue or risk to consider when using the fine-tuned model for certain applications.




(a) Fine-tuned model behavior evaluation results on math prompts

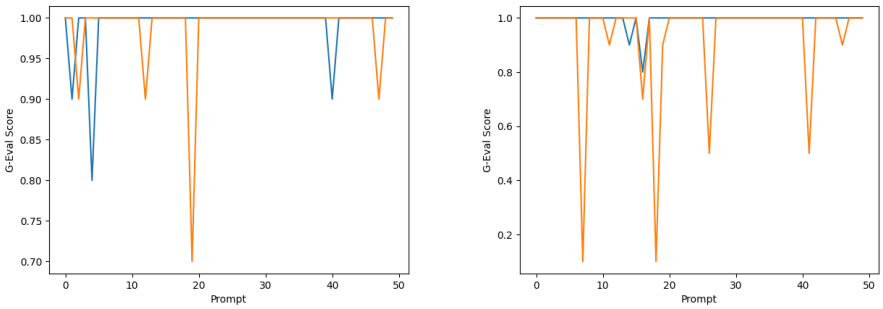
(b) Fine-tuned model behavior evaluation results on logic prompts

All the code is available at: `https://github.com/alexdellabruna/llm-behavior-cert`.

#### 5.3.3.7 Results and Discussion

Figures 38a and 38b show the results of the behavioral integrity evaluation. The blue line shows the base model results while the orange line shows the fine-tuned model results. The x-axis number of the prompt in the category, while the y-axis shows the JS divergence scores. The results

show that the fine-tuned model demonstrates a high level of behavioral integrity on the mathematical reasoning prompts, with a JS divergence score of 0.02, which is below the threshold of 0.1, suggesting that the model's behavior is integral on this category of prompts. On the other hand, the fine-tuned model demonstrates a lower level of behavioral integrity on the logical reasoning prompts, with a JS divergence score of 0.10, which is above the threshold of 0.1, suggesting that the model's behavior is not integral on this category of prompts. These results suggest that while the fine-tuning process was able to preserve the model's behavioral integrity on mathematical reasoning tasks, it may have introduced some changes in its behavior on logical reasoning tasks, which could be a potential issue or risk to consider when using the fine-tuned model for certain applications.




(a) Behavioral integrity evaluation results on math prompts

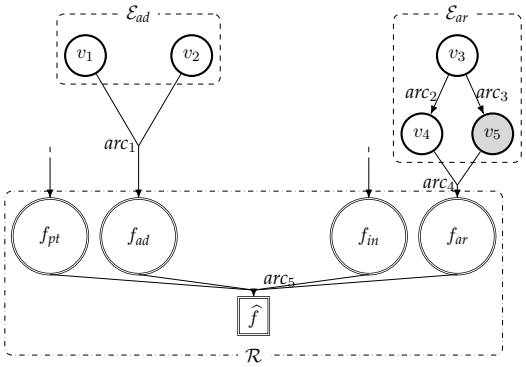
(b) Behavioral integrity evaluation results on logic prompts

Following the defined $\mathcal{CM}$, the final results can be formalized as a certificate of behavioral integrity composed of the following elements:

- $\mathcal{CM}$: the certification model defined in this section, which includes the property label and the hypergraph representing the evaluation process.
- $\overline{\mathcal{B}}$: the set of behaviors that have been verified according to the certification process, including the evidence collected during the evaluation process, such as the fine-tuning, the golden dataset distributions, and the integrity scores.

The evidence collected during the evaluation process can be formalized

as a set of nodes in the hypergraph, which represent the different steps of the evaluation process. Figure 39 shows the nodes visited during the evaluation process.




| Verification steps | |
|---|---|
| $EV(v_1)$ | $download = success$ |
| $EV(v_2)$ | $fine-tuning = success$ |
| $EV(v_3)$ | $golden\ dataset\ creation = success$ |
| $EV(v_4)$ | $behavioral\ evaluation = success$ |
| $EV(v_5)$ | $JS\ distance\ evaluation = fail$ |

Figure 39: Behavioral integrity evidence collected during the evaluation process

Note that, in this case, all nodes in the hypergraph were visited during the evaluation process, which indicates that all steps of the evaluation process were completed and all evidence was collected. The results of all nodes is positive except for the $v_5$ node, which indicates that the Jensen-Shannon divergence score was above the threshold of 0.1, suggesting that the fine-tuned model's behavior is not integral.

# Chapter 6. Conclusions

This thesis explored the concept of safety and security of state-of-the-art LLMs, starting from the definition of the concept of safety and security in the context of LLMs, to the analysis of the potential risks and vulnerabilities associated with their use, to the development of a framework for evaluating and mitigating these risks.

In particular, it introduced, to the best of our knowledge, the first certification scheme for LLM-based applications, which is based on a hypergraph representation of the evaluation process, a definition of compatible behaviors, and a certification methodology. It allows to evaluate the non-functional properties of LLMs in a systematic and comprehensive way, and a wide variety of experiments was conducted to evaluate it, furthermore 3 case studies were conducted to demonstrate the effectiveness of the proposed framework in different scenarios. The case studies covered a wide range of topics, starting from MLOps, to behavioral assessment (black-box approach), and model internals (white-box approach).

## 6.1 Results

The results of the experiments and case studies demonstrate the effectiveness of the proposed framework in the non-functional assessment of LLMs and LLM-based applications. The hypergraph certification scheme was shown to be a powerful tool for assessing safety and security of LLMs, its high modularity permits indeed to easily adapt to different use cases, extending its applicability also over LLMs. The experiments showed that state-of-the-art LLMs are still deeply inaccurate and their use in safe-critical environments should be carefully evaluated, in details:

- Type 1: the explanation engine results showed that state-of-the-art LLMs are still struggling to provide easily understandable explanations, in particular when increasing the level of prompt detail.
- Type 2: the deployment engine results showed that old models are not able to provide secure deployment files, while newer models are much better at providing secure deployment files following hardening best practices, even with highly detailed prompts.

- Type 3: the recommendation engine results showed that state-of-the-art LLMs are still struggling with order bias, and they are not able to provide consistent recommendations, even with highly detailed prompts and examples. In particular, the results showed that the LLMs are not able to consistently select the most appropriate probes for a given scenario, and that they often select probes basing their decision on the order of the probes in the input, rather than on their relevance to the scenario.

The case studies demonstrated that the proposed framework can be applied in a wide variety of scenarios, and that it can provide valuable insights into the safety and security of LLMs, each scenario highlighted different aspects of the safety and security of LLMs, from the MLOps perspective, to the behavioral integrity, and finally to the model internals integrity, providing a comprehensive view of the current landscape of LLM safety and security. In details:

- MLCertOps: the results showed that the proposed approach can effectively integrate security assurance and certification into the MLOps pipeline, providing a comprehensive approach to evaluating and mitigating the risks associated with machine learning models throughout their lifecycle. Furthermore, our approach extended the current paradigm and defined a novel way of defining MLCertOps pipelines.
- Behavioral integrity evaluation: the results showed that the proposed framework can effectively evaluate the behavioral integrity of LLMs, providing valuable insights into their behavioral patterns. The results showed that even a small fine tune (like QLoRA) can significantly modify the behavior of the model.
- Internals integrity evaluation: the results showed that the proposed framework can effectively evaluate the structural integrity of LLMs, providing valuable insights into their architectural components. The results showed that even if a small fine tune (like QLoRA) is applied, the internal activations of the model remain largely unchanged.

## 6.2 Future Work

Future work can build upon the findings of this thesis in several ways. First, the hypergraph certification methodology can be further refined and extended to include weighted edges, which would allow for a more nuanced evaluation of LLMs, taking into account the relative importance of each step in the certification process. Second, the framework can be extended to support the evaluation of LLMs in more complex scenarios, such as multi-agents systems, to further validate its effectiveness and identify any potential limitations. Finally, the framework can be expanded to include semi-automatic generation of certification hypergraphs, which would allow for a more efficient and scalable evaluation of LLMs.

# Bibliography


[1] Muhammad Azeem Akbar et al. "Toward successful DevSecOps in software development organizations: A decision-making framework". In: *Inf. Softw. Technol.* 147 (2022).

[2] Salaheddin Alzubi et al. "Open Deep Search: Democratizing Search with Open-source Reasoning Agents". In: *arXiv preprint arXiv:2503.20201* (2025).

[3] Marco Anisetti, Claudio A. Ardagna, and Nicola Bena. "Multi-Dimensional Certification of Modern Distributed Systems". In: *IEEE Transactions on Services Computing* 16.3 (2023).

[4] Marco Anisetti, Alex Della Bruna, and Giandonato Inverso. "Assurance in DevSecOps Pipelines". In: *ITADATA 2024*. 2024.

[5] Marco Anisetti et al. "Rethinking Certification for Trustworthy Machine-Learning-Based Applications". In: *IEEE Internet Computing* 27.6 (2023).

[6] Claudio A. Ardagna and Nicola Bena. "Non-Functional Certification of Modern Distributed Systems: A Research Manifesto". In: *Proc. of IEEE SSE 2023*. Chicago, IL, USA, July 2023.

[7] Claudio A. Ardagna et al. "Big Data Assurance: An Approach Based on Service-Level Agreements". In: *Big Data* 11 (3 2023).

[8] Yuntao Bai et al. "Constitutional AI: Harmlessness from AI Feedback". In: *arXiv preprint arXiv:2212.08073* (2022).

[9] Christian Banse et al. "EMERALD: Evidence Management for Continuous Certification as a Service in the Cloud". In: *Proc. of CLOSER 2025*. Porto, Portugal, Apr. 2025.

[10] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. *Titans: Learning to Memorize at Test Time*. 2024. arXiv: 2501.00663 [cs.LG]. URL: https://arxiv.org/abs/2501.00663.

[11] Ali Behrouz et al. *Nested Learning: The Illusion of Deep Learning Architectures*. 2025. arXiv: 2512.24695 [cs.LG]. URL: https://arxiv.org/abs/2512.24695.

[12] Nicola Bena et al. "Certifying Accuracy, Privacy, and Robustness of ML-Based Malware Detection". In: *SN Computer Science* 5 (2024).

[13] Federico Bianchi et al. "Safety-Tuned LLaMAs: Lessons From Improving the Safety of Large Language Models that Follow Instructions". In: *arXiv preprint arXiv:2309.07875* (2024).

[14] Dylan Bouchard. "An Actionable Framework for Assessing Bias and Fairness in Large Language Model Use Cases". In: *arXiv preprint: 2407.10853* (2024).

[15] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Proc. of NeurIPS 2020*. Virtual, Dec. 2020.

[16] Jeanne Sternlicht Chall and Edgar Dale. *Readability Revisited: The New Dale-Chall Readability Formula*. Brookline Books, 1995, p. 159.

[17] Xiaoyin Chen et al. "Self-Evolving Curriculum for LLM Reasoning". In: *arXiv preprint arXiv:2505.14970* (2025).

[18] Can Cui et al. "A Survey on Multimodal Large Language Models for Autonomous Driving". In: *Proc. of IEEE/CVF WACVW 2024*. Waikoloa, HI, USA, Jan. 2024.

[19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proc. of NAACL 2019*. Minneapolis, Minnesota, June 2019.

[20] European Parliament and Council. *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence and amending Regulations (EC) No 300/2008, (EU) No 167/2013, (EU) No 168/2013, (EU) 2018/858, (EU) 2018/1139 and (EU) 2019/2144 and Directives 2014/90/EU, (EU) 2016/797 and (EU) 2020/1828 (Artificial Intelligence Act)*. 2024. URL: https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng.

[21] Giorgio Gallo et al. "Directed hypergraphs and applications". In: *Discrete Applied Mathematics* 42.2–3 (1993).

[22] Jiawei Gu et al. "A Survey on LLM-as-a-Judge". In: *arXiv preprint arXiv:2411.15594* (2025).

[23] Dan Hendrycks et al. "Pretrained Transformers Improve Out-of-Distribution Robustness". In: *Proc. of ACL 2020*. Online, July 2020.

[24] Shlomo Hoory et al. "Learning and Evaluating a Differentially Private Pre-trained Language Model". In: *Proc. of EMNLP 2021*. Punta Cana, Dominican Republic, Nov. 2021.

[25] Edward J. Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *Proc. of ICLR 2022*. Virtual, Apr. 2022.

[26] Siwon Kim et al. "ProPILE: Probing Privacy Leakage in Large Language Models". In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023.

[27] J. Peter Kincaid et al. *Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel*. Research Branch Report ADA006655. Naval Technical Training Command, Research Branch, Jan. 1975. URL: https://apps.dtic.mil/sti/pdfs/ADA006655.pdf.

[28] Brian Lester, Rami Al-Rfou, and Noah Constant. "The Power of Scale for Parameter-Efficient Prompt Tuning". In: *Proc. of EMNLP 2021*. Punta Cana, Dominican Republic, Nov. 2021.

[29] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Proc. of NeurIPS 2020*. Virtual, Dec. 2020.

[30] Xiang Lisa Li and Percy Liang. "Prefix-Tuning: Optimizing Continuous Prompts for Generation". In: *Proc. of IJCNLP 2021*. Virtual, Aug. 2021.

[31] Yinheng Li et al. "Large Language Models in Finance: A Survey". In: *Proc. of ACM ICAIf 2023*. Brooklyn, NY, USA, Nov. 2023.

[32] Zhen Li et al. "Leveraging Large Language Models for NLG Evaluation: Advances and Challenges". In: *Proc. of EMNLP 2024*. Miami, FL, USA, Nov. 2024.

[33] Percy Liang et al. "Holistic Evaluation of Language Models". In: *TMLR* (2023).

[34] Chin-Yew Lin. "ROUGE: A package for automatic evaluation of summaries". In: *Proc. of Workshop on Text Summarization Branches Out*. Barcelona, Spain, July 2004.

[35] Sebastian Lins, Stephan Schneider, and Ali Sunyaev. "Trust is Good, Control is Better: Creating Secure Clouds by Continuous Auditing". In: *IEEE Transactions on Cloud Computing* 6.3 (2018).

[36] Xiao Liu et al. "P-Tuning: Prompt Tuning Can Be Comparable to Fine-tuning Across Scales and Tasks". In: *Proc. of ACL 2022*. Dublin, Ireland, May 2022.

[37] Yang Liu et al. *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. 2023. arXiv: 2303.16634 [cs.CL]. URL: https://arxiv.org/abs/2303.16634.

[38] Anton Lozhkov et al. "StarCoder 2 and The Stack v2: The Next Generation". In: *arXiv preprint arXiv:2402.19173* (2024).

[39] Fiammetta Marulli et al. "Understanding Readability of Large Language Models Output: An Empirical Analysis". In: *Procedia Computer Science* 246 (2024).

[40] OpenAI et al. “GPT-4 Technical Report”. In: *arXiv preprint arXiv:2303.08774* (2024).
[41] Kishore Papineni et al. “BLEU: a method for automatic evaluation of machine translation”. In: *Proc. of ACL 2002*. Philadelphia, PA, USA, July 2002.
[42] Baptiste Rozière et al. “Code Llama: Open Foundation Models for Code”. In: *arXiv preprint arXiv:2308.12950* (2024).
[43] Natalie Schluter. “The limits of automatic summarisation according to ROUGE”. In: *Proc. of EACL 2017: Volume 2, Short Papers*. Valencia, Spain, Apr. 2017.
[44] Tobias Schnabel and Jennifer Neville. “Symbolic Prompt Program Search: A Structure-Aware Approach to Efficient Compile-Time Prompt Optimization”. In: *arXiv preprint arXiv:2404.02319* (2024).
[45] Rohin Shah et al. *An Approach to Technical AGI Safety and Security*. 2025. arXiv: 2504.01849 [cs.AI]. URL: https://arxiv.org/abs/2504.01849.
[46] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.
[47] Peiyi Wang et al. “Large Language Models are not Fair Evaluators”. In: *Proc. of ACL 2024*. Bangkok, Thailand, Aug. 2024.
[48] Shenao Wang et al. “Large Language Model Supply Chain: A Research Agenda”. In: *ACM TOSEM* 34.5 (2025).
[49] Yifan Yao et al. “A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly”. In: *High-Confidence Computing* 4.2 (2024).
[50] Asaf Yehudai et al. “Survey on Evaluation of LLM-based Agents”. In: *arXiv preprint arXiv:2503.16416* (2025).
[51] Weizhe Yuan, Graham Neubig, and Pengfei Liu. “BARTScore: Evaluating Generated Text as Text Generation”. In: *Proc. of NeurIPS 2021*. Virtual, Dec. 2021.
[52] Lianmin Zheng et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Proc. of NeurIPS 2023*. New Orleans, LA, USA, Dec. 2023.
[53] Kaijie Zhu et al. “PromptRobust: Towards Evaluating the Robustness of Large Language Models on Adversarial Prompts”. In: *Proc. of LAMPS 2024*. Salt Lake City, UT, USA, Oct. 2024.
[54] Yangtian Zi, Harshitha Menon, and Arjun Guha. “More Than a Score: Probing the Impact of Prompt Specificity on LLM Code Genera-

tion". In: *Proc. of IJCNLP 2025 and AACL 2025*. Mumbai, India, Dec. 2025.