INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Architecture for scalable deployment of AI models

João Vitor Nogueira da Costa - a42545

Dissertation presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:
Prof. Rui Pedro Lopes
Prof. José Rufino

Bragança
October, 2025

Architecture for scalable deployment of AI models

João Vitor Nogueira da Costa - a42545

Dissertation presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Rui Pedro Lopes

Prof. José Rufino

Bragança

October, 2025

Dedication

Primeiramente quero agradecer aos meus orientadores, professor Rui Pedro Sanches de Castro Lopes e professor José Carlos Rufino Amaro pela ajuda e orientação durante todo o processo. Agradeço a todos os amigos e colegas que me acompanharam ao longo destes anos e que realmente tornaram este percurso divertido e relativamente fácil. Por último, mas não menos importante, um grande obrigado à Maria, que me apoiou incondicionalmente em todos os momentos. Por fim, agradeço à minha família, que sempre acreditou em mim e me apoiou em todas as minhas decisões.

Recognition of the use of Artificial Intelligence

I acknowledge the use of ChatGPT4 (OpenAI, https://chatgpt.com/) and Claude (Antropic, https://claude.ai) for translation and text improvement both semantically and syntactically.

Abstract

This work presents a modular architecture for the scalable deployment of Artificial Intelligence (AI) models that combines Infrastructure-as-Code, container orchestration, and automated observability-driven control loops.

The system provisions compute resources on on-premises Proxmox environments using Terraform, applies post-provision configuration with Ansible, orchestrates containerized services through Docker Swarm, serves Machine Learning (ML) models via TorchServe, and stores and visualizes operational metrics using InfluxDB and Grafana.

The final design closes an autonomous feedback loop in which Grafana alerts trigger a backend that executes Terraform actions to add or remove worker nodes; newly created machines are configured and joined to the cluster automatically by Ansible.

The prototype was validated with two pretrained image classification models (ResNet-18, DenseNet-161), demonstrating functional correctness (idempotent provisioning, service replication, load balancing, and failover) and performance benefits under load when elastic scaling is enabled.

While the approach proved portable between Amazon Web Services (AWS) and Proxmox and effective for medium scale workloads, the evaluation surfaced practical constraints—most notably Virtual Machine (VM) provisioning latency and a five-minute alert resolution delay—that limit responsiveness to short bursts.

The architecture meets its primary objectives of scalable, automated model serving with minimal operator intervention, and outlines opportunities for reducing reaction time (e.g., container level scaling before VM creation) and enhancing scheduling sophistication.

Keywords: Infrastructure as Code (IaC); Terraform; AWS; Docker; Auto-scaling; ML provisioning.

Resumo

Este trabalho apresenta uma arquitetura modular para a implementação escalável de modelos de AI, que combina Infraestrutura-como-Código, orquestração de contentores e ciclos de controlo automatizados baseados em observabilidade.

O sistema aprovisiona recursos computacionais em ambientes Proxmox locais utilizando o Terraform, aplica configurações pós-aprovisionamento com o Ansible, orquestra serviços contentorizados através do Docker Swarm, disponibiliza modelos de ML via TorchServe, e armazena e visualiza métricas operacionais utilizando o InfluxDB e o Grafana.

O design final fecha um ciclo de feedback autónomo em que alertas do Grafana desencadeiam um backend que executa ações do Terraform para adicionar ou remover nós de trabalho; as máquinas recém-criadas são configuradas e integradas automaticamente no cluster pelo Ansible.

O protótipo foi validado com dois modelos de classificação de imagens pré-treinados (ResNet-18, DenseNet-161), demonstrando correção funcional (aprovisionamento idempotente, replicação de serviços, balanceamento de carga e tolerância a falhas) e benefícios de desempenho sob carga quando a escalabilidade elástica está ativada.

Embora a abordagem tenha provado ser portátil entre AWS e Proxmox e eficaz para cargas de trabalho de média escala, a avaliação revelou limitações práticas — nomeadamente a latência no aprovisionamento de VM e um atraso de cinco minutos na resolução de alertas — que reduzem a capacidade de resposta a picos de curta duração.

A arquitetura cumpre os seus principais objetivos de disponibilização escalável e automatizada de modelos, com intervenção mínima do operador, e identifica oportunidades

para reduzir o tempo de reação (por exemplo, escalabilidade ao nível de contentores antes da criação de VM) e aumentar a sofisticação do agendamento.

Palavras-chave: IaC; Terraform; AWS; Docker; Escalabilidade automática; Aprovisionamento de ML.

Contents

List of Figures

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

AWS Amazon Web Services.

CI/CD Continuous Integration / Continuous Deployment.

CPU Central Processing Unit.

ESTiG Escola Superior de Tecnologia e Gestão.

GPU Graphics Processing Unit.

HCL HashiCorp Configuration Language.

HPC High-Performance Computing.

HTTP HyperText Transfer Protocol.

IaC Infrastructure as Code.

IPB Instituto Politécnico de Bragança.

MaaS Model as a Service.

ML Machine Learning.

MLOps Machine Learning Operations.

REST Representational State Transfer.

SaaS Software-as-a-Service.

SOA Service-Oriented Architecture.

SSH Secure Shell.

TFX TensorFlow Extended.

VM Virtual Machine.

VPC Virtual Private Cloud.

Chapter 1

Introduction

The deployment of AI models in production environments has become a cornerstone of modern technological innovation. As AI systems grow in complexity and application scope, their operational success depends not only on the accuracy of the models themselves but also on the efficiency, scalability, and resilience of the deployment architecture.

Delivering AI services at scale demands infrastructures that can respond dynamically to fluctuations in demand, integrate seamlessly into heterogeneous environments, and maintain performance under sustained workloads. This work addresses these requirements through the conception and validation of a modular, automated, and self-regulating deployment platform capable of supporting AI-driven services across diverse operational contexts.

1.1 Context

The increasing adoption of AI in domains such as healthcare, finance, manufacturing, and digital services has placed significant emphasis on the ability to serve models reliably to large and distributed user bases. In such contexts, scalability is not a luxury but a fundamental necessity: diagnostic tools must deliver real-time predictions to medical professionals, recommendation engines must process millions of concurrent requests, and predictive maintenance systems must analyse streams of industrial sensor data without

interruption.

Achieving these capabilities requires the orchestration of multiple technological layers. Containerization provides a portable and consistent environment for model execution, while orchestration platforms distribute workloads across computing resources to optimize performance and ensure availability. IaC enables automated, reproducible provisioning of virtualized resources, reducing human error and accelerating deployment cycles. Furthermore, robust monitoring systems provide the observability needed to maintain operational efficiency, detect anomalies, and trigger automated responses to performance degradation or infrastructure stress.

This technological ecosystem must also address challenges such as cold start latency, the complexities of model version management, and the fine-tuning of alerting mechanisms to avoid over- or under-scaling. Security considerations, regulatory compliance, and cost control further influence the design and operation of AI deployment pipelines, making the integration of these elements into a cohesive architecture a complex but essential task.

1.2 Objectives and Contributions

The primary objective of this work is to design, implement, and evaluate an architecture that enables the scalable, reliable, and cost-efficient deployment of AI models.

The system should automate the provisioning and configuration of infrastructure resources, leveraging IaC to ensure reproducibility and maintainability. It should support containerized model serving, with built-in mechanisms for load balancing, fault tolerance, and elastic scaling in response to fluctuating computational demand.

Continuous monitoring and alerting should form a core part of the design, allowing the system to observe performance metrics in real time and initiate automated scaling actions when thresholds are reached.

In addition, the architecture design should make it portable across cloud-based, on-premises, and hybrid environments, ensuring adaptability to various operational constraints.

All this should be pursued with an emphasis on reducing operational costs by scaling resources down during periods of low demand, while maintaining the security, resilience, and compliance necessary for deployment in sensitive or regulated contexts.

1.3 Structure of the document

The remainder of this document is organized as follows.

- Chapter 2: presents the theoretical and technological background underpinning scalable AI deployment, including the principles, techniques, and tools relevant to building robust deployment infrastructures;
- Chapter 3: outlines the functional and non-functional requirements for the proposed system, followed by a detailed description of the planned architecture and its core components, including automation frameworks, orchestration platforms, model serving solutions, and monitoring tools;
- Chapter 4: describes the implementation phase in detail, tracing the evolution from an initial prototype to a fully automated, self-scaling deployment platform, along with the practical considerations and technical decisions made throughout the process;
- Chapter 5: documents the validation stage, in which the system is evaluated through functional testing, performance and load analysis, fault tolerance assessment, and monitoring effectiveness;
- Chapter 6: concludes this report with a synthesis of findings, an assessment of the system's strengths and limitations, and recommendations for future enhancements.

Chapter 2

State-Of-The-Art

The deployment of AI models at scale is a multidisciplinary challenge that combines principles of software architecture, distributed systems, and Machine Learning Operations (MLOps). The ultimate objective is to transform trained models from isolated research artifacts into operational components capable of delivering consistent, low-latency, and high-availability services to end-users or integrated systems. This chapter reviews the theoretical and technological foundations that inform the design of scalable AI deployment infrastructures, with a focus on deployment workflows, enabling tools, architectural strategies, and scalability mechanisms.

2.1 Deployment of AI Models

As AI continues to evolve, it is becoming an essential part of many real-world applications, ranging from chatbots and recommendation engines to autonomous vehicles and predictive analytics [1]. However, one of the key challenges in making AI useful at a large scale lies not just in developing powerful models, but in deploying them efficiently so they can serve many users or systems simultaneously. This challenge is addressed by what is known as architecture for scalable deployment of AI models [2], [3].

At its core, this type of architecture involves building the technical infrastructure needed to make AI models available in a reliable, flexible, and efficient way. It ensures

that models can handle large volumes of requests, adapt to changing demand, and operate smoothly in a variety of environments, whether running in the cloud, on-premises, or on edge devices. To achieve this, engineers use tools and systems like containerization technologies such as Docker, orchestration platforms like Kubernetes, and specialized model-serving frameworks such as TensorFlow Serving or TorchServe. These components are often combined with cloud services from providers like AWS or Google Cloud, along with Application Programming Interfaces (APIs), load balancers, and automated deployment pipelines that enable continuous integration and delivery [4].

This architecture is used whenever AI models are deployed into real-world production environments. It plays a critical role in industries like technology, healthcare, retail, and finance, where models must be accessed by large numbers of users or connected systems. For instance, a recommendation algorithm in an e-commerce platform may need to serve millions of users at once, or a diagnostic model in a hospital system may be called upon by doctors working across various departments or even different locations. In such cases, it is essential that the deployment architecture can scale up during peak usage, and scale down when demand decreases, ensuring both performance and cost-efficiency [5].

The importance of scalable deployment cannot be overstated. It directly affects how responsive, reliable, and cost-effective AI systems are in practice. A well-architected deployment ensures that users experience fast and accurate responses from AI models, even under heavy load. It also helps organizations reduce infrastructure costs by allocating computing resources dynamically [6]. Moreover, with the right deployment strategies in place, teams can update or improve models without causing service interruptions, which is especially important in high-stakes environments like healthcare or financial services. Additionally, scalable architectures enable AI systems to be deployed globally, bringing low-latency access to users across different regions [7].

Ultimately, the architecture for scalable deployment of AI models is a vital enabler of modern AI. It bridges the gap between model development and real-world impact, allowing AI to function as a dependable part of digital infrastructure at scale [8].

2.1.1 Key Aspects

Model Export

After an AI model has been trained, it does not immediately leap into action; it must first undergo a crucial transition known as *model export*, which acts as a bridge between the controlled environment of training and the unpredictable world of production. Initially, the model exists within the tight confines of the training framework. These environments are optimized for flexibility and experimentation, but not for the efficiency and portability required in deployment. This is where *model export* comes in: it transforms the model into a framework-agnostic format like ONNX or TorchScript, enabling it to operate outside its original ecosystem. Such exported models are essential for deploying AI to lightweight or constrained systems, from mobile phones and IoT devices to real-time industrial controllers [9].

This act of exporting a model does more than simply relocate it; it lays the foundation for modular and scalable inference. A model trained in Python can now be executed in C++ or even embedded in edge devices—systems that often lack the capacity to host full training libraries. Export formats also support important optimization strategies such as quantization and pruning, making inference not only possible but efficient under strict resource limitations. These capabilities have real-world implications [8]. In healthcare, exported models are integrated into diagnostic tools that must make instantaneous decisions; in manufacturing, they guide robotic systems or quality inspection pipelines that cannot afford latency. In addition, exported models can be versioned and encrypted, enhancing traceability and security, an increasingly vital requirement in regulated industries [10].

Model Integration

But export is only the beginning. Once a model has been prepared for deployment, it must be integrated into the broader operational fabric of the application or organization it serves. A model, no matter how sophisticated, is powerless if it lives in isolation. Integration connects the model to databases, data pipelines, APIs, logging systems, and

user interfaces, transforming it into an active agent within a larger digital ecosystem. This process is often carried out using a microservices architecture, where the model becomes a self-contained service accessible via Representational State Transfer (REST) or gRPC APIs. Frameworks like FastAPI and Flask are used to build these services, while API gateways and service meshes like Istio ensure traffic is routed efficiently and securely [11].

Integration is also where AI begins to touch real people. In e-commerce, for example, a recommendation model isn't just another component—it must interact with real-time user behavior to personalize shopping experiences [10]. In healthcare, the model might feed directly into hospital information systems, offering doctors diagnostic suggestions in the middle of patient care. Such integrations are not just technically challenging—they must also adhere to strict regulations, from HIPAA in the U.S. to GDPR in Europe. And because real-world systems are never static, integration is not a one-off effort. It must account for ongoing maintenance, updates, and retraining. This is where MLOps enters the picture, blending DevOps principles with ML-specific workflows to ensure lifecycle continuity, reliability, and repeatability [2], [4].

Model Scalability

As these AI systems are deployed and integrated, they must also be scalable. A model that works well in testing, but collapses under real-world demand is of little use. Scalability ensures that AI systems can handle increasing loads—whether that means more users, more data, or more complex tasks—without suffering delays or breakdowns [12]. *Horizontal scaling* is the most common approach: more instances of the model service are deployed across different machines or containers, orchestrated by platforms like Kubernetes. *Vertical scaling*, involving upgrades to individual machines, can supplement this approach when needed. Effective scalability hinges on intelligent resource allocation—distributing Central Processing Unit (CPU), Graphics Processing Unit (GPU), and memory loads so that performance remains smooth and cost-effective [7], [13].

Load balancing, request batching, and asynchronous processing all help sustain throughput under pressure. Lightweight or compressed versions of models are often used in high-load scenarios, and in some advanced systems, models are dynamically loaded and cached based on usage patterns. Without such strategies, AI systems risk becoming bottlenecks rather than enablers. In mission-critical applications, from fraud detection systems that must evaluate thousands of transactions per second to conversational agents managing simultaneous dialogues, scalability is not optional; it is essential for business viability [7].

Model Monitoring

However, even a scalable system is not complete unless it is also monitored. Unlike traditional software, AI systems are inherently probabilistic and sensitive to changes in their input environment. Data drifts. Behavior evolves. A model that performs well today may begin to falter tomorrow. Monitoring provides the eyes and ears of the deployed system, tracking performance metrics such as latency, resource usage, and—most crucially—prediction accuracy and reliability. Tools like Prometheus and Grafana offer visibility into infrastructure health, while specialized platforms such as Arize AI and Fiddler dive into model-specific metrics like data distribution, confidence scores, and fairness indicators [14].

Monitoring also supports accountability and compliance. In high-stakes sectors like finance or healthcare, every prediction may need to be recorded, explained, and audited. Real-time explainability tools can provide insight into why a model made a particular decision, supporting transparency and trust. And as the environment shifts—perhaps a new fraud tactic emerges, or user behavior changes—monitoring can detect these shifts early, triggering automated retraining pipelines that refresh the model with new data. This closes the feedback loop, allowing AI systems not only to adapt but to evolve continuously [1].

Taken together, model export, integration, scalability, and monitoring are not isolated stages—they form an interconnected narrative in the life of any deployed AI system.

Each stage sets the stage for the next: exporting enables integration, integration demands scalability, and scalability is incomplete without vigilant monitoring. This story of deployment is not just about getting a model into production; it's about ensuring that the model continues to thrive, adapt, and deliver value in the ever-changing real world.

2.1.2 Common Techniques

AI model deployment encompasses a range of techniques that address different operational needs, performance requirements, and risk management strategies. The most prevalent methods include: i) *batch processing*, where models analyze large volumes of data at scheduled intervals; ii) *real-time processing*, which provides immediate predictions for time-sensitive applications; and iii) *edge deployment*, which moves computation closer to data sources to reduce latency and enhance privacy. Additionally, iterative improvement and reliability are supported by techniques such as *A/B testing*, used to compare model versions and optimize outcomes through controlled experiments, and *canary releases*, which introduce new models gradually to mitigate potential failures in production. Together, these approaches form the foundation of modern AI deployment practices, balancing scalability, responsiveness, and safety across diverse environments.

Batch Processing

Batch processing is a deployment strategy in which data is collected, grouped, and processed in large chunks or "batches" at predetermined intervals, making it especially effective in scenarios where immediate responses are not required. In this approach, the model receives all input data at once, processes it, and returns outputs after the computation is complete—unlike real-time processing, where data is handled as it arrives. This strategy is commonly used in data-heavy applications such as credit scoring, offline recommendation engines, image annotation pipelines, and healthcare diagnostics. For instance, a medical institution might analyze a large set of X-ray images overnight to generate diagnostic results for physicians to review the next day, while marketing platforms may segment

customers based on recent activity and periodically generate targeted campaigns [2].

The major advantage of batch processing is efficiency. When operating at scale, it allows for optimized use of computational resources by grouping operations, minimizing I/O overhead, and enabling parallel processing. Tools like Apache Spark, AWS SageMaker Batch Transform, and Hadoop are commonly employed for such workloads. Batch jobs can also be scheduled during off-peak hours to reduce costs on cloud platforms and avoid overloading resources used during business hours [14].

However, batch processing has limitations. It is unsuitable for use cases requiring low latency or real-time decision-making. Furthermore, it introduces inherent delay due to the time gap between data generation and processing. Therefore, modern systems often employ hybrid architectures, using batch processing for long-term analysis and real-time processing for immediate actions. This dual approach enables businesses to benefit from both efficiency and responsiveness, depending on their operational requirements [15].

Real-time Processing

Real-time processing, also known as online inference, is a technique where AI models provide predictions almost instantaneously as new data becomes available. Unlike batch processing, which accumulates and processes data over time, real-time processing handles each data point individually and produces immediate outputs. This approach is critical in domains that require fast or instant decisions, such as autonomous driving, cybersecurity, financial trading, and customer service chatbots.

In real-time processing systems, latency is a key concern. These systems must be engineered to minimize the time between receiving an input and generating an output—often targeting response times in milliseconds. Achieving such low latency requires not only optimized model architectures but also carefully designed infrastructure. This includes using lightweight models, deploying inference on GPUs or TPUs, and employing efficient networking protocols. Technologies such as NVIDIA TensorRT, ONNX Runtime, or FastAPI-based inference servers are frequently utilized for real-time deployment [16].

Real-time AI is widely used in various industries. In transportation, real-time models

predict train or bus delays and suggest optimal routes to commuters. In manufacturing, edge devices equipped with real-time models detect defects on production lines and trigger corrective actions without human intervention. In finance, fraud detection systems continuously analyze transaction data to flag anomalies the moment they occur. These applications showcase how real-time AI can enhance operational efficiency, improve user experiences, and increase safety [2].

Despite its advantages, real-time processing can be complex and resource-intensive to implement. It requires robust system architecture, constant monitoring, and often high infrastructure costs due to the need for always-on services. Additionally, real-time models must be fault-tolerant and secure, especially when deployed in mission-critical applications. Nonetheless, as latency-sensitive applications become more common—especially with the rise of IoT and autonomous systems—real-time AI processing is becoming a fundamental capability for modern intelligent systems [4].

Edge Deployment

Edge deployment refers to the practice of placing AI models on devices or servers that are physically close to where data is generated, such as embedded systems, smartphones, gateways, or local servers. This strategy drastically reduces latency and bandwidth usage by performing computation locally. It's especially useful for applications that demand fast response times and strong data privacy, as it avoids transmitting data to and from centralized cloud environments.

One key advantage of edge deployment is its ability to function in real-time without needing a constant internet connection. In industrial settings, edge devices can analyze sensor data on-site to detect equipment failures and trigger predictive maintenance. Similarly, smart surveillance systems can process video feeds locally to identify threats or unusual activities instantly, without waiting for cloud-based analysis. This leads to faster, more autonomous decision-making [2].

Privacy is another strong benefit of edge deployment. Since data remains within the local environment, sensitive information—such as facial recognition inputs or patient

health records—is processed securely, minimizing exposure risks. This is vital in sectors like healthcare, where regulations like HIPAA in the U.S. require stringent data handling. Technologies supporting this paradigm include platforms such as NVIDIA Jetson, TensorFlow Lite, and Intel OpenVINO, along with tools like AWS Greengrass and Azure IoT Edge that manage updates and orchestration. As the IoT ecosystem expands, edge deployment is becoming a dominant model for delivering fast, efficient, and privacy-conscious AI [10].

A/B Testing

A/B testing is a controlled experimental method where multiple versions of an AI model are deployed to different user segments to evaluate which performs better. The technique is crucial when small, data-driven improvements are needed, and it allows teams to make informed decisions by comparing performance metrics like accuracy, conversion rates, or user engagement between the model variants [8].

In deployment, A/B testing helps mitigate risk when introducing new models [2]. For instance, before rolling out a new recommendation algorithm to all users, a small percentage may be exposed to the new version while the rest continue using the existing one [17]. If the new model performs better on key indicators, it can be promoted; if not, the team can revert to the original without major consequences. This protects production systems from potential regressions caused by untested updates [5].

Technically, A/B testing involves infrastructure for traffic routing, monitoring, and statistical evaluation. Solutions such as LaunchDarkly, Optimizely, and built-in tools within Google TensorFlow Extended (TFX) or Azure ML make it easier to integrate A/B experiments into deployment pipelines. One ongoing challenge is ensuring the test is unbiased and statistically valid, especially in systems with low traffic or high variability. Nonetheless, A/B testing remains a cornerstone of AI experimentation and iteration, particularly for user-facing systems and business-critical applications [14].

Canary Releases

Canary releases are a model deployment technique where a new version is introduced to a small portion of users or traffic before being scaled to the entire system. This approach functions as a safety mechanism—much like canaries once used in coal mines—to detect problems early and prevent widespread impact if issues arise with the new model version.

This strategy is particularly useful in sensitive or high-risk applications. For example, a financial institution might initially deploy a new credit scoring model to a small percentage of applicants to compare its decisions with the current model. If errors, inconsistencies, or biases are identified, the rollout can be paused or reversed before the model affects the broader user base. This provides a critical safety net in production environments.

Implementing canary releases requires precise traffic control, observability, and rollback capabilities. Tools like Kubernetes, Istio, and Spinnaker enable granular traffic routing and performance monitoring, allowing teams to incrementally increase usage if the model behaves as expected. Unlike A/B testing, which is aimed at performance optimization, canary releases focus on minimizing risk and ensuring system stability. Often, organizations use both approaches—first validating performance through A/B testing, then deploying via canary release to safeguard reliability. Together, they offer a robust framework for secure and resilient AI deployment [8].

2.1.3 Popular Tools

In modern ML and MLOps ecosystems, several tools have emerged as industry standards for model development, deployment, and infrastructure management.

TensorFlow

TensorFlow is an open-source deep learning framework developed by the Google Brain team, widely recognized for its flexibility, scalability, and support for production-ready deployment. It provides a comprehensive ecosystem for building, training, and deploying ML models, supporting both low-level operations for advanced users and high-level APIs

like Keras for quick prototyping. TensorFlow supports a wide range of use cases, from natural language processing and computer vision to time-series forecasting and reinforcement learning [14].

One of TensorFlow's key strengths lies in its deployment capabilities. TensorFlow Serving is a dedicated model serving system optimized for TensorFlow models, allowing developers to deploy trained models in production environments with low latency and high throughput. TensorFlow also supports exporting models into lightweight formats via TensorFlow Lite, enabling deployment on mobile, embedded, and IoT devices. Furthermore, TensorFlow.js allows models to run directly in web browsers, opening doors for client-side AI applications [2].

TensorFlow is widely used in both industry and research. Large-scale platforms like Google Translate, Gmail's Smart Compose, and Google Photos' image recognition features are powered by TensorFlow models. In healthcare, TensorFlow has been employed to build models that detect diabetic retinopathy and lung cancer from medical images. Its extensive support for distributed training and GPU acceleration makes it ideal for organizations dealing with vast datasets and requiring high-performance inference [10].

The framework continues to evolve, with recent improvements focusing on performance optimization (XLA compiler), interoperability (support for ONNX), and tooling (TensorBoard for visualization and TFX for MLOps). Its active community, rich documentation, and integration with platforms like Google Cloud AI make it a dominant force in the ML landscape. Despite the emergence of competing libraries, TensorFlow remains a go-to tool for both academic researchers and enterprise AI engineers due to its versatility and production readiness [7].

Docker

Docker allows developers to package applications—including ML models—with all their dependencies into standardized units called containers. These containers are lightweight, portable, and consistent across different environments, which solves the infamous "it works

on my machine" problem that plagues traditional software deployment. For AI applications, Docker enables model developers to encapsulate not just the trained model, but also preprocessing scripts, libraries, and environment configurations into a reproducible format [8].

The role of Docker in AI model deployment is particularly prominent in microservices-based architectures. Each model or service can be containerized independently, scaled horizontally, and deployed as part of a larger orchestration framework. This is especially useful for serving multiple models simultaneously in a production environment or when models need to be tested across different hardware or software setups. Containers also improve security and maintainability by isolating each component. Docker has become the backbone of many MLOps pipelines. It allows for seamless integration into Continuous Integration / Continuous Deployment (CI/CD) workflows, where models can be automatically built, tested, and deployed through containers. In industries like finance, where model versioning, auditability, and rollback are critical, Docker provides the necessary infrastructure for reliable and traceable deployment. Moreover, because containers are OS-agnostic, they can be run on any cloud provider or on-premises servers without modification [12].

The real power of Docker is often unlocked when combined with orchestration tools like Kubernetes. While Docker manages individual containers, Kubernetes handles the lifecycle of container clusters, offering capabilities such as load balancing, failover, and auto-scaling. As AI adoption grows across sectors, Docker continues to play a pivotal role in ensuring that model deployment is fast, consistent, and scalable—no matter where or how the model is executed [18].

Kubernetes

Kubernetes is an open-source platform developed by Google for automating the deployment, scaling, and management of containerized applications. In the context of AI, Kubernetes has emerged as a critical component of scalable, production-grade ML infrastructure. It orchestrates the deployment of containers (often Docker-based), manages resource

allocation, scales workloads up or down based on demand, and provides high availability and fault tolerance [18].

For AI workloads, Kubernetes simplifies the challenge of deploying multiple models across distributed infrastructures. It allows data scientists and ML engineers to deploy model-serving endpoints, batch processing jobs, or training clusters with minimal manual configuration. Kubernetes can be configured to run GPU workloads, making it suitable for training deep learning models or serving inference tasks that require accelerated computation. Moreover, features like pod autoscaling and rolling updates ensure that deployments remain both efficient and reliable [12].

Kubernetes also supports advanced deployment strategies such as A/B testing, blue-green deployments, and canary releases, which are essential for safely deploying new AI models in production. Tools like Kubeflow—built specifically for Kubernetes—extend its capabilities to ML by providing components for model training, hyperparameter tuning, pipeline management, and model serving. This makes Kubernetes not just an orchestration platform, but an MLOps backbone [13].

Organizations like Spotify, Shopify, and Tesla use Kubernetes to manage large-scale AI infrastructure, ensuring resilience and performance in mission-critical systems. In edge computing, lightweight versions of Kubernetes (like K3s or KubeEdge) allow similar orchestration capabilities on edge clusters. As AI deployments become increasingly complex, Kubernetes offers a unifying layer that abstracts infrastructure management and allows teams to focus on delivering intelligent applications reliably and at scale [7].

AWS SageMaker

AWS SageMaker is Amazon's fully managed ML service that provides a complete pipeline for building, training, and deploying ML models at scale. Designed to simplify the end-to-end ML workflow, SageMaker supports model development using built-in algorithms, Jupyter notebooks for custom training, automated model tuning, and one-click deployment. It is a powerful platform for enterprises that want to leverage cloud infrastructure without managing the underlying hardware or software [14].

It is particularly useful for deploying AI models in real-time or batch environments. The platform provides several deployment options, including real-time endpoints for low-latency inference, batch transform for processing large datasets, and SageMaker Neo for deploying optimized models to edge devices. Additionally, it supports multi-model endpoints, allowing users to deploy multiple models to the same endpoint and dynamically route inference requests, thereby reducing cost and complexity [8].

A key strength of SageMaker is its integration with the broader AWS ecosystem. It connects seamlessly with data sources like S3, logging services like CloudWatch, and orchestration tools like AWS Step Functions. Security is handled through IAM policies, Virtual Private Cloud (VPC) configurations, and encryption, which is compliant with enterprise and regulatory standards. SageMaker Pipelines and Model Monitor further enhance the platform by automating workflows and monitoring deployed models for data drift, latency, and bias [14].

SageMaker has been widely adopted in sectors such as healthcare, finance, manufacturing, and retail. Companies use it to develop recommendation systems, demand forecasting models, predictive maintenance tools, and computer vision applications. With features like AutoPilot (automated ML), built-in explainability, and GPU-based training, SageMaker continues to evolve as a comprehensive, enterprise-ready solution for end-to-end ML deployment in the cloud [17].

FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints. It is specifically designed for speed and developer productivity, making it an excellent choice for deploying ML models as web services. Built on top of Starlette for web routing and Pydantic for data validation, FastAPI offers asynchronous request handling and automatic generation of interactive API documentation using OpenAPI and Swagger.

For AI deployments, FastAPI allows data scientists and ML engineers to expose their models via RESTful endpoints in just a few lines of code. The framework is particularly

well-suited for lightweight inference tasks where minimal latency is required. For example, a sentiment analysis model or a language classifier can be wrapped in a FastAPI service and deployed to respond to HyperText Transfer Protocol (HTTP) requests from web or mobile applications in real-time [8].

One of FastAPI's biggest advantages is its support for asynchronous operations and high concurrency, which allows it to handle thousands of requests per second. This makes it an ideal choice for serving models in production environments where performance and responsiveness are crucial. FastAPI is also fully compatible with tools like Docker and Kubernetes, making it easy to containerize and scale deployments across cloud or on-premises infrastructure.

FastAPI has gained popularity in both startup environments and large enterprises due to its ease of use, speed, and clean syntax. It is commonly used in combination with model-serving libraries like ONNX Runtime, TensorFlow Serving, or custom PyTorch inference scripts. Its flexibility makes it suitable for building anything from simple proof-of-concept APIs to robust, large-scale AI services. As the demand for API-first AI systems grows, FastAPI is increasingly seen as the go-to solution for model deployment in Python-centric workflows.

PyTorchServe

PyTorchServe is an open-source model serving framework developed by AWS and Meta for deploying PyTorch models in production. It provides a flexible and efficient way to host, scale, and manage deep learning inference services. Designed to integrate seamlessly with the PyTorch ecosystem, PyTorchServe abstracts the complexity of serving models, offering built-in components for inference handling, metrics collection, and model version management [8].

PyTorchServe supports REST and gRPC inference APIs, enabling applications to interact with deployed models through standard protocols. It allows developers to define custom handlers for preprocessing and postprocessing, making it highly adaptable for tasks like image classification, text generation, and recommendation systems. The

framework also supports multi-model serving, dynamic loading, and batching, optimizing resource usage in production environments [14].

A key advantage of PyTorchServe is its native integration with monitoring tools such as Prometheus and model logging through metrics endpoints. It can be easily containerized using Docker and orchestrated via Kubernetes, aligning with modern MLOps workflows. PyTorchServe is increasingly adopted by organizations that build their AI pipelines around PyTorch due to its simplicity, scalability, and strong community backing. Its ongoing development ensures compatibility with the latest PyTorch versions and evolving production standards [7].

Technologies Comparison

Table 2.1 provides a comparison of several widely adopted technologies used throughout the AI model deployment lifecycle. Each tool contributes to a different aspect of the operationalization of ML systems, from model export and integration to scalability and monitoring. TensorFlow and PyTorchServe focus primarily on framework-specific model serving, providing dedicated mechanisms for exporting and hosting trained models. Docker and Kubernetes serve as the backbone for containerization and orchestration, enabling scalable, reliable, and portable deployments. AWS SageMaker offers a fully managed environment that simplifies the entire ML workflow, while FastAPI provides a lightweight, high-performance option for exposing inference endpoints as RESTful services. Together, these technologies represent complementary layers within the modern MLOps stack, facilitating efficient model delivery and lifecycle management.

2.1.4 Approaches

AI model deployment can be achieved through several distinct approaches, each offering trade-offs in terms of scalability, control, latency, cost, and data governance. The most common paradigms include: i) *cloud deployment*, where models are hosted on managed cloud infrastructure for high scalability and ease of integration; ii) *on-premises*

Technology	Model Export	Integration	Scalability	Monitoring
TensorFlow	✓ Supports `SavedModel` and `TF Lite` formats.	✓ Integrates with TF Serving or TensorFlow.js.	✗ Limited native scaling; relies on orchestration.	✗ Basic via TensorBoard; advanced requires external tools.
Docker	✗ Not used for export, but packages exported models.	✓ Essential for service encapsulation.	✓ Enables scalable deployments with orchestrators.	✗ Requires integration with Prometheus or similar tools.
Kubernetes	✗ Not directly involved in export.	✓ Manages deployed, containerized model services.	✓ Supports auto-scaling and high availability.	✓ Integrates with Prometheus, Grafana, etc.
AWS SageMaker	✓ Supports ONNX, TensorFlow, and others.	✓ Tightly integrated with AWS infrastructure (S3, Lambda).	✓ Auto-scaling and multi-model endpoints supported.	✓ Built-in tools for drift, latency, fairness (e.g., Clarify).
PyTorchServe	✓ Supports serialized PyTorch `.pt` and TorchScript models.	✓ Integrates with Docker, Kubernetes, and custom Python handlers.	✓ Handles multi-model serving and batch inference.	✓ Exports Prometheus metrics for monitoring and logging.
FastAPI	✗ Not a model export tool.	✓ Exposes models via RESTful endpoints.	✗ Requires Docker/Kubernetes for scaling.	✗ Monitoring depends on external services.

Table 2.1: Comparison of Common Technologies in AI Model Deployment

deployment, which emphasizes security, privacy, and low-latency inference within local infrastructure; and, iii) *hybrid deployment*, which strategically combines cloud and local resources to balance performance and flexibility. Additionally, the emergence of *Model-as-a-Service (MaaS)* has introduced a consumption-based model that enables access to pre-trained or customizable AI models through APIs, lowering the entry barrier for organizations adopting AI. Each of these approaches addresses different operational needs and constraints, as discussed in the following subsections.

Cloud Deployment

Cloud deployment refers to hosting and executing AI models on remote servers managed by cloud service providers like AWS, Google Cloud Platform (GCP), Microsoft Azure, or IBM Cloud. This approach offers high computational capacity, elasticity, and ease of

access, making it ideal for organizations that want to rapidly scale AI workloads without managing physical infrastructure. Models trained locally or in the cloud can be deployed as APIs, batch jobs, or real-time endpoints using managed services [7].

One of the main benefits of cloud deployment is scalability. Cloud platforms allow automatic scaling based on incoming traffic, which is critical for services with variable loads such as customer support chatbots, recommendation engines, or fraud detection systems. They also provide access to High-Performance Computing (HPC) resources, including GPUs and TPUs, which enable fast inference for large deep learning models. Additionally, these platforms offer integrations with storage, logging, monitoring, and DevOps tools, forming a complete ecosystem for end-to-end AI operations [3].

Cloud deployment is especially useful for enterprise applications where security, compliance, and service-level agreements (SLAs) are crucial. For example, banks use cloud services for credit risk modeling and real-time transaction monitoring. In healthcare, cloud deployment supports AI-driven diagnostics and population health analysis at scale. Cloud-based AI systems are also commonly employed in Software-as-a-Service (SaaS) products, such as CRM platforms that offer AI-powered recommendations or predictions [19].

However, cloud deployment has its limitations. Latency can be a major issue for time-sensitive applications, as data must travel to the cloud and back. Privacy is another concern, particularly when dealing with sensitive data in sectors like healthcare or defense. Additionally, high-volume inference can incur substantial costs due to cloud computing and bandwidth usage. To address these concerns, many organizations combine cloud with edge or on-premises solutions, resulting in hybrid deployment strategies that balance performance, cost, and security [3].

On-Premises Deployment

On-premises deployment involves hosting AI models on local servers, private data centers, or specialized industrial equipment within the organization's physical infrastructure.

This deployment model gives organizations full control over their data, models, and infrastructure, making it the preferred choice for sectors where data privacy, security, or regulatory compliance is critical. Industries such as healthcare, manufacturing, finance, and government often favor on-premises deployment for these reasons [10].

One major advantage of on-premises deployment is low latency. Since data does not need to be transmitted over the internet, inference can happen faster and more reliably. This is particularly beneficial in industrial settings, such as semiconductor manufacturing, where AI models must make split-second decisions to detect faults or optimize production. Similarly, in hospitals, on-premises deployment ensures that diagnostic models have immediate access to local imaging equipment and patient records, reducing time-to-diagnosis [3].

Security and compliance are also driving factors. Organizations in regulated industries may face restrictions that prevent data from being stored or processed outside national borders. On-premises deployment ensures compliance with these regulations by keeping all data local. Moreover, it offers a more stable cost structure, as cloud service costs can scale unpredictably with usage, especially in high-frequency inference scenarios [20].

However, on-premises deployment presents challenges in terms of infrastructure management and scalability. It requires significant upfront investment in hardware, software, and technical expertise. Maintenance, upgrades, and hardware failures are the organization's responsibility. Furthermore, integrating with external APIs or updating models can be more cumbersome compared to cloud-based solutions. Despite these challenges, on-premises deployment remains an essential option for organizations where performance, privacy, and compliance are non-negotiable [21].

Hybrid Deployment

Hybrid deployment is a strategy that combines cloud and on-premises (or edge) resources to execute AI workloads. This approach seeks to capitalize on the strengths of both deployment types—leveraging the scalability and flexibility of the cloud with the speed and control of local infrastructure. Hybrid deployments are increasingly popular in scenarios

that demand both real-time processing and centralized model training or orchestration [8].

A common hybrid architecture involves training large models in the cloud using powerful GPUs or TPUs, then exporting those models and deploying them locally for inference. For example, in smart city infrastructure, AI models may be trained in the cloud using aggregated traffic data, and then deployed to roadside edge devices for real-time vehicle detection and routing decisions. This reduces latency and minimizes bandwidth usage, while still enabling continuous improvement via cloud-based retraining [10].

Hybrid deployment is also useful for organizations with geographically distributed operations. In retail, edge devices in stores can run local models for inventory management or customer analytics, while a centralized cloud system coordinates model updates and collects aggregated metrics. This allows businesses to operate autonomously at the local level while maintaining global oversight and optimization.

Nevertheless, hybrid deployment introduces complexity in terms of system design, synchronization, and model management. Organizations must implement robust version control, consistent monitoring across environments, and secure communication channels between edge and cloud components. MLOps practices and tools become crucial in hybrid systems to automate testing, validation, and deployment across heterogeneous infrastructures. As AI adoption grows in both cloud-native and operational technology environments, hybrid deployment is emerging as a strategic imperative [10], [18].

Model-as-a-Service (MaaS)

Model-as-a-Service (MaaS) is a cloud-based paradigm in which pre-trained or customizable AI models are offered as services through APIs. It allows users to access sophisticated AI functionality, such as natural language processing, image recognition, speech synthesis, or recommendation systems, without needing to build, train, or host models themselves. MaaS platforms abstract away the infrastructure and model complexity, enabling rapid integration into applications [20].

Popular examples of MaaS include OpenAI's GPT-based models, Hugging Face Inference API, Google Cloud AI, IBM Watson, and Azure Cognitive Services. These platforms

support a wide range of AI capabilities, from language translation and text summarization to facial recognition and sentiment analysis. For instance, a developer building a customer service chatbot can use a MaaS provider's language model to understand user intent without needing to train a model from scratch [14].

MaaS is especially valuable for startups, small teams, and businesses without in-house AI expertise. It significantly reduces the time, cost, and expertise required to adopt AI, enabling developers to focus on core business functionality. Furthermore, MaaS platforms often handle scalability, security, compliance, and monitoring as part of their service-level agreements, which is beneficial for applications with fluctuating demand or strict uptime requirements.

However, MaaS also introduces certain trade-offs. Dependency on third-party APIs can limit customization and control. Privacy and latency are concerns when user data must be sent to external servers. Moreover, pricing models—usually based on per-request billing—can become expensive at scale. To address these challenges, some advanced MaaS platforms are now integrating with edge computing (e.g., MaaS on MEC nodes), allowing inference to happen locally while the provider manages model updates and retraining in the cloud. This new evolution of MaaS blends the convenience of cloud services with the responsiveness and privacy of local deployment [3].

2.2 Scalable Architectures

Scalable architecture refers to the design principles and strategies used to build systems that can efficiently handle increasing amounts of work, users, or data without compromising performance or reliability. It ensures that as demand grows, the system can grow with it, whether by adding more resources (vertical scaling) or distributing the load across multiple systems (horizontal scaling). In today's digital landscape, where applications must support millions of users and vast data volumes, scalable architecture is crucial for maintaining responsiveness, availability, and cost-effectiveness [22]–[25].

2.2.1 Scalability

Scalability is a crucial property of a software system, defining its ability to handle an increasing amount of work through the addition of resources [23]. Essentially, it refers to the capability of a software system to accommodate growth in various dimensions of its operations. This includes the number of users or simultaneous requests, the amount of data to be processed and managed, the value derived from data analysis, and the ability to maintain a stable and consistent response time under growing loads [22]. For systems such as SaaS applications, scalability means maintaining consistent performance even as workload increases, and being able to incrementally add resources as needed [23]. Importantly, the foundations of scalability must be established from the beginning during the software's architectural design.

Approaches to Scalability

Large-scale systems mainly use two tactics for scalability: Vertical Scalability (Scale Up) and Horizontal Scalability (Scale Out).

Vertical Scalability involves running the application on a machine with a better configuration, which includes more computing power, greater memory, higher disk bandwidth, and increased disk space [23]. It usually requires deploying higher-capacity hardware so that a single node can deliver greater throughput. To take full advantage of this, the software components must efficiently utilize the additional hardware resources, such as using multi-threaded servers to leverage multi-core CPUs [22]. However, vertical scalability has limitations at internet scale, as a single machine's resources cannot be increased indefinitely, and the cost of such upgrades often grows disproportionately [24]. From a software architecture perspective, vertical scalability also implies breaking down functionality into individual components that can be used separately to meet new or changing requirements without needing to redesign the entire architecture [26].

With Horizontal Scalability, an application runs in a distributed manner across multiple machines with similar configurations [23], [24]. It involves replicating processing and

data storage nodes, enabling requests to be spread across these replicas to boost the system's total capacity. This model relies on replicating processing components so that the application load can be evenly distributed, requiring effective load balancing strategies. Economically, it is advantageous because it avoids costly system replacements and allows infrastructure costs to grow proportionally with demand [22]. Horizontal scalability allows service providers to deploy a new service using a small set of building blocks (processing, networking, and storage), and simply add more blocks as demand increases—without needing expensive redesigns. It also allows an architecture to extend its capabilities by forming efficient connections with other software architectures, creating synergy among them [24].

Other scalable design principles for application servers include Divide-and-Conquer strategies (partitioning tasks into smaller functions), Asynchronicity (work based on resource availability), Encapsulation (well-encapsulated components), Concurrency (parallel tasks), and Frugality (cost efficiency).

Load Balancing

Load balancing is an essential component of Horizontal Scalability. It ensures that the application load is distributed evenly across the replicated processing components [22]. Stateless services are critical for this, as they allow load balancing policies to distribute individual requests in isolation [23]. In high-availability architectures, such as those used in e-commerce, load balancing is achieved by having multiple hosts at each layer of the system. One example of a load balancing mechanism is the Amazon Elastic Container Service (ECS) Autoscaling Group, which ensures continuous server availability and can launch new instances in response to high request volumes. This contributes significantly to system reliability and fault tolerance [27].

Microservices Architecture

The Microservices architecture has become increasingly popular, leading many companies to migrate from monolithic applications. Microservices support independent development

and deployment, which can help systems to more effectively address problems.

In this context, an emerging concept is the "Modular Monolith", which aims to combine the fast development cycles of a monolith with the scalability, security, and fault tolerance of microservices. This approach organizes systems into loosely coupled modules, each with clearly defined boundaries and explicit dependencies, promoting independence and isolation. Unlike microservices, a modular monolith typically uses a single shared database. It can also serve as an intermediate step before fully migrating to microservices.

A notable framework for modular monoliths is Google's Service Weaver, which allows developers to write applications as modular monoliths and deploy them as a set of microservices. It offers flexible scalability, enabling applications to scale either horizontally or vertically, whether implemented as a monolith or as distributed microservices. However, migrating to microservices can also introduce challenges, such as more complex cross-binary changes and increased difficulty in managing API modifications [28].

Service-Oriented Architecture (SOA) can also be used to decouple large software systems, offering excellent scalability. Many scalable systems adopt SOA, such as Amazon DynamoDB. The general trend in software development is the shift from monolithic applications to microservices. Tools like Ansible, an automation platform, are designed to work with complex environments and architectures, including distributed systems and microservices [23].

2.2.2 Key Tools and Technologies

To build and manage scalable architectures, it is essential to use a robust set of tools and technologies that support automation, load distribution, resource management, and effective monitoring.

AWS Cloud Platform

Cloud computing platforms are the foundation of many modern scalable architectures, offering on-demand services that eliminate the need to acquire and maintain physical infrastructure. AWS is one of the most widely used public cloud providers, known for its scalability and accessibility. It provides a wide range of services that support the creation of reliable and highly available infrastructure solutions [27].

Key AWS services for scalability include Elastic Compute Cloud (EC2), which allows users to select server specifications and provides computational power via virtual or physical machines with preconfigured OS images. EC2 instances can be launched and configured as needed, making them fundamental to infrastructure flexibility [29].

The Auto scaling Group feature enables automatic scaling both horizontally and vertically in response to heavy loads. For instance, it can automatically launch more powerful instances (e.g., t3.large instead of t3.medium) when server capacity is insufficient, ensuring continuous server availability and responsiveness to high request volumes [27].

Elastic Container Service (ECS) supports the deployment and orchestration of Docker containers, while Elastic Kubernetes Service (EKS) offers a managed Kubernetes solution on AWS for running Kubernetes clusters. DynamoDB is Amazon's fast, reliable, and cost-effective NoSQL database designed for internet-scale applications, with automated provisioning and a single-tier scalability architecture where all nodes are equal and interconnected in a ring structure [30].

Other AWS tools include S3 buckets for encrypted software backups and media storage, CloudWatch for real-time CPU usage monitoring on active hosts, and SNS for instant notifications about service anomalies or server issues. Identity and Access Management (IAM) enables secure user and permission management [23].

Containerization

Containerization and orchestration are key to building scalable and efficient architectures, particularly in microservices and cloud-native environments [31].

Docker provides a lightweight, standalone executable package that includes everything needed to run an application—code, libraries, dependencies, and runtime. It ensures applications run consistently across different environments and solves runtime issues across various operating systems. Docker abstracts away the host system, simplifies component recovery, and supports easy scaling of applications [30].

Kubernetes is an open-source container orchestration platform that automates deployment, scaling, and management of containerized applications. It ensures the desired number of application instances are running at all times, starting or stopping them as necessary. While Kubernetes can be complex, it is manageable with automation tools.

Kubernetes emphasizes scalability and high availability, enabling systems to continue functioning even if nodes fail by transferring workloads to other nodes within seconds without human intervention. It integrates with CI/CD tools like Jenkins and Ansible for automating cluster and application management. Managed Kubernetes offerings include Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS) [32].

Docker Swarm is another container orchestration tool that manages a cluster of Docker hosts as a single virtual system. It supports load balancing and service discovery for big data applications based on microservices **\cite {alghawli_resilient_2024}**.

Orchestration

Terraform provides a declarative IaC approach for defining and managing infrastructure. It supports the creation, modification, and versioning of infrastructure securely and efficiently. With support for over 1700 providers—including AWS, Azure, and GCP—it is cloud-agnostic and also supports on-premises platforms like VMware [33].

Terraform offers automation and efficiency by accelerating cloud adoption, facilitating collaboration, and enhancing scalability. It reduces infrastructure provisioning time and minimizes human error risk. Integration into CI/CD pipelines (e.g., with Jenkins) enables infrastructure management automation and supports microservices orchestration and container management [34]

Terraform integrates deeply with AWS, allowing efficient provisioning and resource

management. It is cloud-agnostic, meaning it can manage resources across multiple cloud providers such as AWS, Azure, and Google Cloud Platform (GCP). Studies indicate that Terraform significantly reduces provisioning time compared to manual AWS management [35].

Ansible is an open-source software platform for automating IT infrastructure management, application deployment, and system configuration. Using a push-based architecture, a control node connects to managed nodes (via Secure Shell (SSH) for Linux, WinRM for Windows) to execute YAML-based playbooks [30].

Ansible is designed for scalability and efficiency, particularly in complex distributed systems and microservices environments. It excels at configuration management, application deployment, and service orchestration. Automation with Ansible improves both operational efficiency and system scalability [36].

Ansible also integrates with cloud providers like AWS, Azure, and GCP for dynamic resource provisioning and autoscaling. It supports Docker and Kubernetes integration to automate container and cluster management. Through idempotency, it ensures repeated task execution produces consistent results, avoiding duplicates or disruptions. The Ansible Automation Platform centralizes execution, supports GitOps, configuration management, and continuous delivery, and includes Automation Mesh, which simplifies scaling across multiple platforms and sites with fault tolerance and redundancy [30].

Databases

Databases are a critical component of scalability, with the choice of database technology having a direct impact on performance and architectural scalability. SaaS applications may use traditional or modified relational databases, while NoSQL databases are also common [23].

Amazon DynamoDB is a highly scalable, fast, reliable, and cost-effective NoSQL database service designed for internet-scale applications. It includes automated provisioning and database management.

PostgreSQL is mentioned in Kubernetes cluster configurations as a relational database

solution. MongoDB can run in containers for scalable deployments. MySQL is cited in the context of running applications like WordPress [31].

To support scalability, it's essential to separate system functions and avoid direct access to the database, allowing each component to be scaled independently. This also enables stateless server development, facilitating parallel development by different teams [23].

Observability and Monitoring

Observability and monitoring are crucial to ensuring the functionality and responsiveness of scalable systems [27].

Continuous monitoring tools and practices are vital for maintaining efficiency and resolving issues promptly during the operational phase (Day 2) of the software lifecycle. Real-time vulnerability detection and risk assessment mechanisms allow clusters to collect periodic data and automatically adjust configurations based on detected risks and applied controls [30].

Data analytics services, such as the TSTEM platform, leverage the Elastic Stack (Elasticsearch, Logstash, Kibana) for centralized logging and problem detection. Elasticsearch acts as the log repository, Logstash processes and sends logs, and Kibana provides a web interface for visualization.

Real-time threat detection is another key feature, with platforms like TSTEM capable of processing compute-intensive data pipelines to detect, collect, and share Cyber Threat Intelligence (CTI) from multiple online sources. AI and ML play a major role in producing faster and more accurate threat telemetry, especially as data becomes larger and more diverse [33].

Real-time notifications (e.g., via AWS SNS) are important for keeping stakeholders informed of abnormal service behavior or server issues. Tools like Ansible can be used to monitor progress, performance, and address problems in real time, offering full visibility into automation workflows. Systems can also send alerts via email, such as in the event of a failed Jenkins job [37].

2.3 Summary

The deployment of AI models marks a critical phase in the ML life cycle, bridging model development with real-world applications. Over the past decade, this area has matured rapidly, driven by the need for scalable, reliable, and efficient AI systems. Modern deployment extends beyond simply placing a trained model into production it emphasizes scalability, resilience, maintainability, and continuous monitoring to ensure lasting performance. Various deployment techniques serve distinct operational needs: batch processing suits tasks like fraud detection or recommendation updates, while real-time (or online) deployment powers latency-sensitive applications such as autonomous systems and personalized healthcare. Edge deployment has become increasingly important for reducing latency, preserving privacy, and saving bandwidth by processing data locally. Progressive roll out strategies such as A/B testing and canary releases allow organizations to validate models incrementally, minimizing risks in production.

A rich ecosystem of tools and architectures supports these deployment methods. Frameworks such as TensorFlow facilitate model training and serving, while Docker and Kubernetes provide portability, orchestration, and fault tolerance for distributed systems. Cloud-native services like AWS SageMaker simplify end-to-end workflows, and lightweight frameworks such as FastAPI enable efficient, high-performance APIs integration. Deployment architectures vary, from scalable cloud environments to secure on-premises setups and hybrid models that blend both advantages. Managed AI services (Model as a Service (MaaS)) further streamline operations by delivering AI capabilities on demand without infrastructure overhead. As scalability remains a central challenge, technologies like load balancing, microservices, and orchestration ensure adaptability and resilience under fluctuating workloads. Supported by robust infrastructure, databases, and monitoring tools, AI deployment has evolved into a continuous process of integration, observation, and optimization forming a cornerstone of sustainable and trustworthy AI systems.

Chapter 3

Requirements and Architecture

This chapter defines the technical requirements of the target architecture, together with the corresponding technical choices. The goal is to construct a self-regulating and responsive infrastructure capable of provisioning compute resources, deploying containerized AI models, and maintaining continuous observability and operational efficiency. The infrastructure combines IaC, container orchestration, automation, and monitoring systems to form a unified and scalable AI service platform.

3.1 Requirements

This section outlines the functional and non-functional requirements necessary for building a scalable, self-regulating infrastructure capable of deploying and managing AI models. These requirements guided the design, implementation, and evaluation of the system to ensure it meets its goals. For each requirement, a brief description of the tools and technologies selected for its fulfillment is provided.

3.1.1 Functional Requirements

Infrastructure Automation: Terraform serves as the foundational automation tool, managing infrastructure lifecycle events such as resource creation, configuration, and teardown. By using the HashiCorp Configuration Language (HCL), the system defines VM specifications, network parameters, and resource constraints declaratively. When an increase in demand is detected, Terraform can be programmatically instructed to provision two or more VMs automatically, ensuring minimal human intervention.

Virtualization Management: Proxmox VE, an open-source server virtualization environment, is responsible for the instantiation and lifecycle management of VMs. It provides APIs and CLI tools that Terraform can use to create VMs from predefined templates. This enables consistent machine creation, reproducible development/test environments, and horizontal scaling across compute clusters.

Automated Configuration Management: Ansible playbooks automate the setup of system dependencies and ensure consistency across all VMs. These playbooks configure Docker, install dependencies such as NVIDIA drivers for GPU-accelerated inference, and add the machine to the Docker Swarm cluster. Each machine is treated as an idempotent target, reducing configuration drift.

Model Deployment and Execution: AI models, trained offline using frameworks such as PyTorch, are containerized and exposed via HTTP APIs or socket-based RPC endpoints within Docker containers. These containers can be easily replicated and deployed to multiple nodes, ensuring fault tolerance and load distribution.

Load Balancing and Service Discovery: Docker Swarm offers built-in service discovery and load balancing. It abstracts container endpoints into services and intelligently routes requests based on service health and availability. As new containers are deployed or old ones fail, Swarm redistributes the traffic with zero downtime.

Real-Time Monitoring and Metrics Logging: InfluxDB collects high-frequency metrics related to CPU utilization, memory usage, container status, and disk I/O. Grafana visualizes this data using dashboards, enabling engineers to monitor infrastructure health

at a glance. These dashboards are also used for performance tuning and capacity planning.

Dynamic Scaling: Grafana's alerting system defines upper and lower CPU usage thresholds (e.g., 80% and 40%). When the upper threshold is breached consistently over a specified period, an alert is sent to a listener service that triggers Terraform to scale-out the cluster by creating new VMs. Conversely, when usage falls below the lower threshold for a sustained period, the system automatically initiates a controlled scale-in process using the Terraform destroy facilities.

Web-Based Alert Handler: A lightweight HTTP server acts as a webhook listener for Grafana alerts. On receiving an alert, it parses the payload, determines whether it's a scale-out or scale-in event, and executes the corresponding Terraform or Ansible commands. This creates an autonomous feedback loop for infrastructure elasticity.

3.1.2 Non-Functional Requirements

Scalability: The architecture must scale horizontally by adding or removing worker nodes VMs based on computational load. The system should support deploying new AI model replicas without downtime, maintaining consistent latency for inference requests during high-traffic periods.

Security: SSH key-based authentication is enforced across all VMs. Communication between containers can be encrypted using Docker Swarm's built-in mutual TLS encryption. Firewalls and network segmentation limit exposure to external traffic, and only essential ports are exposed. Secrets such as model API keys or database tokens are stored in environment variables or external secrets managers.

Resilience and Redundancy: Services deployed in Docker Swarm are replicated across multiple nodes. If one node fails, containers are rescheduled to other healthy nodes. The manager nodes maintain quorum, and system state is consistently replicated to prevent split-brain scenarios.

Maintainability: IaC tools such as Terraform and Ansible provide modular and reusable configurations. Components are broken into logical roles or modules (e.g., VM

creation, Docker installation, model deployment), enabling easier maintenance, updates, and testing.

Observability: Logging (using journald or syslog) and metrics (via InfluxDB) offer complete observability of system state. Grafana dashboards expose both high-level metrics (e.g., total requests per minute) and low-level ones (e.g., disk I/O per container).

Portability: By using containerized deployments, the entire AI inference stack can run on any infrastructure—cloud-based, on-premises, or hybrid—provided it supports Docker and the required system dependencies.

Cost Optimization: Scaling down resources during low-demand periods reduces hardware and energy costs. Idle resources are programmatically destroyed to prevent unnecessary spending, particularly relevant in cloud environments where VM uptime directly correlates with billing.

Compliance and Auditability: The system logs all critical changes such as VM provisioning, container deployment, and access attempts. These logs are timestamped and stored centrally, providing traceability for compliance audits and incident investigations.

3.2 System Architecture

As illustrated in Figure 3.1, the proposed solution is structured around the following main tools: Terraform, Ansible, Proxmox, Docker Swarm, PyTorch Serve, Grafana, and InfluxDB. Further details are next provided on each one and their interactions.

Terraform

IaC is essential for ensuring reproducibility, version control, and scalability in infrastructure management. With IaC, infrastructure configurations are defined as code, allowing consistent deployments across different environments and facilitating collaboration among team members.

Terraform was chosen as the IaC tool for this project due to its broad support for various providers and its ability to manage both cloud and on-premises resources in a

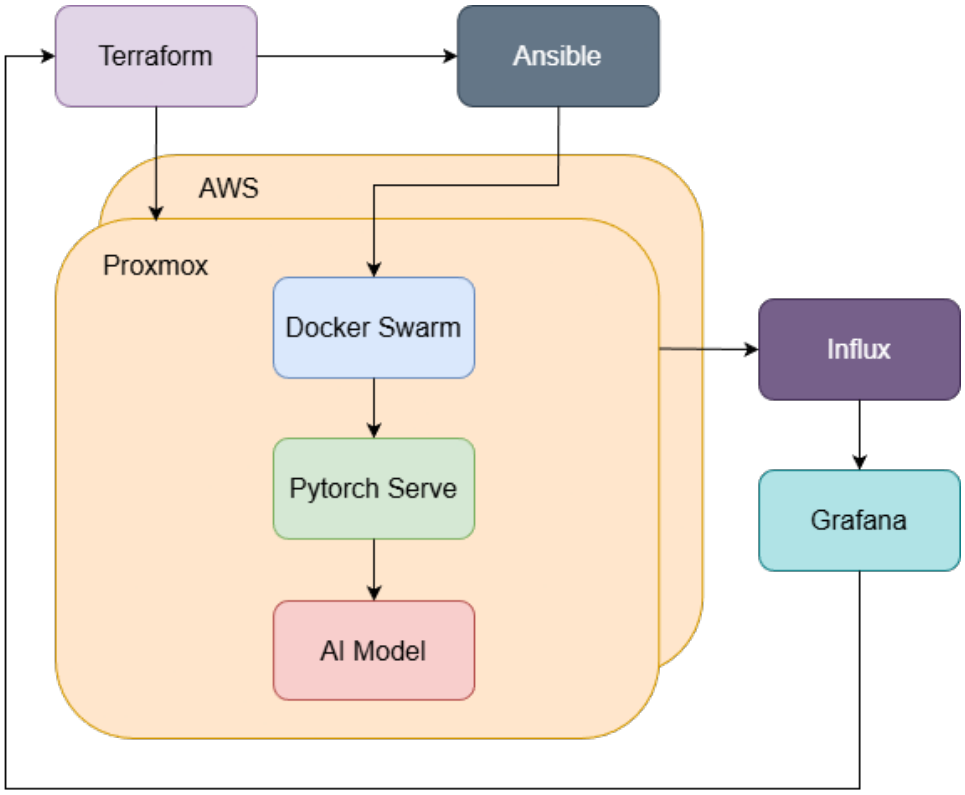

Figure 3.1: High-level System Architecture

declarative way. Terraform is used to provision infrastructure components such as VMs, networking configurations, and storage, automating the deployment process and reducing manual errors.

Proxmox

The system architecture is designed to support container orchestration, model serving, monitoring, and scalability within a virtualized environment managed by Proxmox. The architecture follows a modular approach, where each component plays a specific role in the overall workflow from infrastructure provisioning to application deployment and performance monitoring.

Ansible

Ansible was selected for configuration management and post-provisioning tasks due to its simplicity, agentless architecture, and strong community support. After the infrastructure is provisioned, Ansible is used to automate the configuration of systems, install necessary packages, apply security settings, and deploy services.

Its declarative YAML-based playbooks make it easy to define desired system states and ensure consistency across multiple machines. In this project, Ansible plays a key role in preparing the VMs created by Terraform, ensuring they are ready to host container orchestration tools and application components.

Because it does not require agents running on the managed nodes, Ansible simplifies the management of VMs, especially in a local lab environment like the one used here. This makes it an ideal choice for automating repetitive setup tasks, reducing configuration drift, and supporting scalable infrastructure management workflows.

Docker Swarm

Docker Swarm was chosen as the container orchestration tool primarily due to its simplicity and native integration with Docker. While Kubernetes is a powerful and widely adopted orchestration platform, it has a steeper learning curve and requires more complex setup and management factors that can be limiting in an educational or resource-constrained environment.

Docker Swarm offers a lightweight alternative that is easier to configure and manage, especially for small to medium-scale deployments. It allows users to turn a group of Docker hosts into a single, virtualized cluster where containers can be deployed, distributed, and scaled efficiently.

In this project, Docker Swarm is used to orchestrate services across the VMs provisioned in the local Proxmox environment. It enables service replication, automatic load balancing, and simple scaling operations (e.g., increasing the number of replicas for a service) using straightforward Docker CLI commands. This approach facilitates learning

core concepts of container orchestration while still providing practical tools to manage multi-container applications in a distributed setup.

PyTorch Serve containers

Its main function is the serving AI of models due to its native integration with the PyTorch ecosystem, which ensures seamless deployment of models developed using the same framework. This compatibility simplifies the workflow from training to production, avoiding the need for complex conversions or external tools.

One of the key advantages of PyTorch Serve is its ability to host multiple models simultaneously, making it suitable for projects that require serving different models within the same environment. It exposes models through RESTful APIs, allowing applications to send inference requests over HTTP, which facilitates integration with front-end interfaces and other services.

In addition, PyTorch Serve supports features such as automatic batching, model versioning, logging, and metrics, making it well-suited for production environments. When deployed in container orchestration platforms like Docker Swarm, it can scale horizontally by replicating model servers across multiple nodes, ensuring high availability and performance under load.

Grafana & InfluxDB

Grafana and InfluxDB were chosen as the monitoring and alerting solution for this work due to their seamless integration, real-time data visualization capabilities, and flexibility in handling time-series data. InfluxDB is used to store metrics collected from various services, while Grafana provides a customizable dashboard interface to visualize this data in an intuitive and accessible way.

One of the key reasons for selecting this stack was its support for creating alerting rules based on metric thresholds or patterns. Grafana allows users to define alert conditions directly from dashboards, enabling proactive monitoring of system performance, resource usage, or service health.

To automate incident response and improve observability, Grafana alerts are configured to trigger webhooks if specific conditions are met. These webhooks send real-time notifications to external systems such as messaging platforms (e.g., Discord, Slack) or trigger automated actions like scaling services or restarting containers.

This integration creates a feedback loop between the monitoring layer and the orchestration or notification systems, helping maintain system reliability and enabling quicker reaction to anomalies or performance degradation.
This feedback-driven architecture ensures adaptability and high availability in variable-load environments.

3.3 Anticipated Challenges

The current architecture, while allowing for functional and scalable deployments, still needs improvements to cope with several important challenges:

- Cold Start Latency: It may take considerable time (2–5 minutes) to provision a new VM, configure it, and deploy containers. During high traffic, this may lead to degraded performance before new nodes are ready.

- Model Lifecycle Management: As model versions increase, managing their lifecycle (e.g., deployment, rollback, A/B testing) becomes more complex. Integration with model registries (like MLflow) may be required in the future.

- Alert Configuration Complexity: Poorly tuned thresholds may result in over/under-scaling, leading to either resource exhaustion or degraded performance. Adaptive alerting based on predictive analytics may mitigate this.

- Security Gaps: While basic security measures are in place, improvements such as TLS encryption between services, container image signing, and tighter API access controls are still needed.

- Infrastructure Drift: Configuration changes made outside of Terraform (e.g., direct changes in Proxmox) may cause infrastructure state drift, leading to inconsistencies that are difficult to detect or reconcile.
- Resource Fragmentation: Docker Swarm does not offer the sophisticated scheduling features of Kubernetes, such as GPU-aware placement or affinity rules. This may lead to inefficient resource usage in heterogeneous environments.
- Integration Overhead: Adding support for tools like Vault, ArgoCD, or Kubernetes would improve capabilities but significantly increase the system's complexity and maintenance burden.

Addressing these limitations in the future is essential for scaling the system into a high-availability production-grade environment.

3.4 Summary

This chapter presented the comprehensive requirements and architecture for a scalable, self-regulating infrastructure designed to support the deployment and operation of AI models. The system combines IaC (Terraform), virtualization (Proxmox), configuration management (Ansible), container orchestration (Docker Swarm), model serving (PyTorch Serve), and observability tools (Grafana and InfluxDB) into a cohesive platform.

Functional requirements such as infrastructure automation, dynamic scaling, real-time monitoring, and autonomous alert handling were addressed to ensure responsiveness and minimal human intervention. Non-functional requirements—including scalability, security, maintainability, portability, and compliance—were carefully considered to support production-readiness and long-term operability.

The architecture section detailed the rationale behind the chosen tools, emphasizing ease of integration, educational accessibility, and low operational overhead. The consolidated technical architecture demonstrated how these tools interact within a control-loop paradigm to achieve elasticity, observability, and resilience.

Finally, several remaining challenges were acknowledged, including provisioning latency, model life cycle complexity, alert tuning, and infrastructure drift. These highlight key areas for future improvement as the system matures toward full-scale production deployment.

Chapter 4

Implementation

This chapter presents details on the practical development of the system, starting with the creation of a basic functional prototype and gradually evolving into a robust, automated, and scalable deployment of the architecture. It describes the implementation of the main system components, such as container orchestration, model serving, infrastructure provisioning, and monitoring solutions.

4.1 Baseline Implementation

The development process began with the creation of a minimal but functional prototype aimed at validating the core components of the system. This initial phase focused on container orchestration using Docker Swarm and on deploying ML models via TorchServe. The objective was to establish a working environment that could later be enhanced with automation, monitoring capabilities, and scaling mechanisms. In this stage, the emphasis was on ensuring that each component could operate reliably in isolation before integrating them into a larger architecture.

4.1.1 Technology Research and Initial Setup

Before any implementation, an extensive technology evaluation was carried out to determine the most suitable tools for the main components of the architectre (container orchestration, model deployment, infrastructure automation, and system monitoring). Compatibility, community support, ease of integration, and long-term maintainability were the main criteria for tool selection, dictating the choice of the tools already presented in the previous chapter.

4.1.2 Container Orchestration with Docker Swarm

Docker Swarm was thus chosen for orchestration due to its simplicity, ease of setup, and seamless integration with Docker-based workflows.

The initial cluster Docker Swarm was deliberately kept simple, consisting of one manager node and two worker nodes. The manager node was in charge of orchestration tasks such as scheduling and maintaining service state, while the worker nodes focused on executing the deployed services.

Setting up Docker Swarm involved installing Docker on all nodes, initializing the swarm using the docker swarm init command on the manager, and then adding the worker nodes to the cluster via a secure join token generated by the manager. This structure provided a controlled environment for experimentation while laying the groundwork for future scalability.

4.1.3 Model Serving With TorchServe

Once the Docker Swarm cluster was functional, TorchServe was integrated as the platform for model serving.

Two pretrained deep learning models were selected to test the system's deployment and inference capabilities: ResNet-18, a lightweight convolutional neural network optimized for fast inference and suitable for real-time applications, and DenseNet-161, a more

computationally intensive architecture designed for high accuracy in complex image classification tasks. Each model was packaged into a .mar file using the torch-model-archiver tool and stored in a designated model-store directory, which was then mounted into the Docker containers running TorchServe. This allowed the models to be loaded automatically upon service startup.

During early deployment attempts, an HTTP 400 error occurred, which was traced to an authentication mismatch. Since TorchServe does not require authentication by default, the problem was solved by disabling any authentication-related settings in the deployment configuration. With this issue addressed, the models could be served through REST API endpoints, making them accessible for inference requests across the cluster.

4.2 Towards a Scalable Implementation

After the successful deployment of the prototype, the focus of the work turned toward scalability and automation. Thus, the next phase involved designing a process for infrastructure provisioning that could be reproduced quickly and reliably. This was based on Terraform, the infrastructure-as-code tool that was selected due to its declarative configuration style and broad provider support.

4.2.1 Cloud Infrastructure Provisioning With Terraform

Initially, Terraform was configured to provision resources on AWS. This was done as a quick proof-of-concept, before the deployment of the Proxmox cluster that would be used o sustain the next stages of this research.

The AWS infrastructure included a VPC with subnets and routing tables, Elastic Compute Cloud (EC2) instances created from specific Amazon Machine Images (AMIs), Secure Shell (SSH) key pairs for secure access, and security groups that allowed traffic for Docker, SSH, and HTTP. This automated provisioning approach enabled rapid deployment of the system in a production-grade cloud environment, ensuring consistency between deployments and reducing manual configuration errors. Figure 4.1 illustrates the

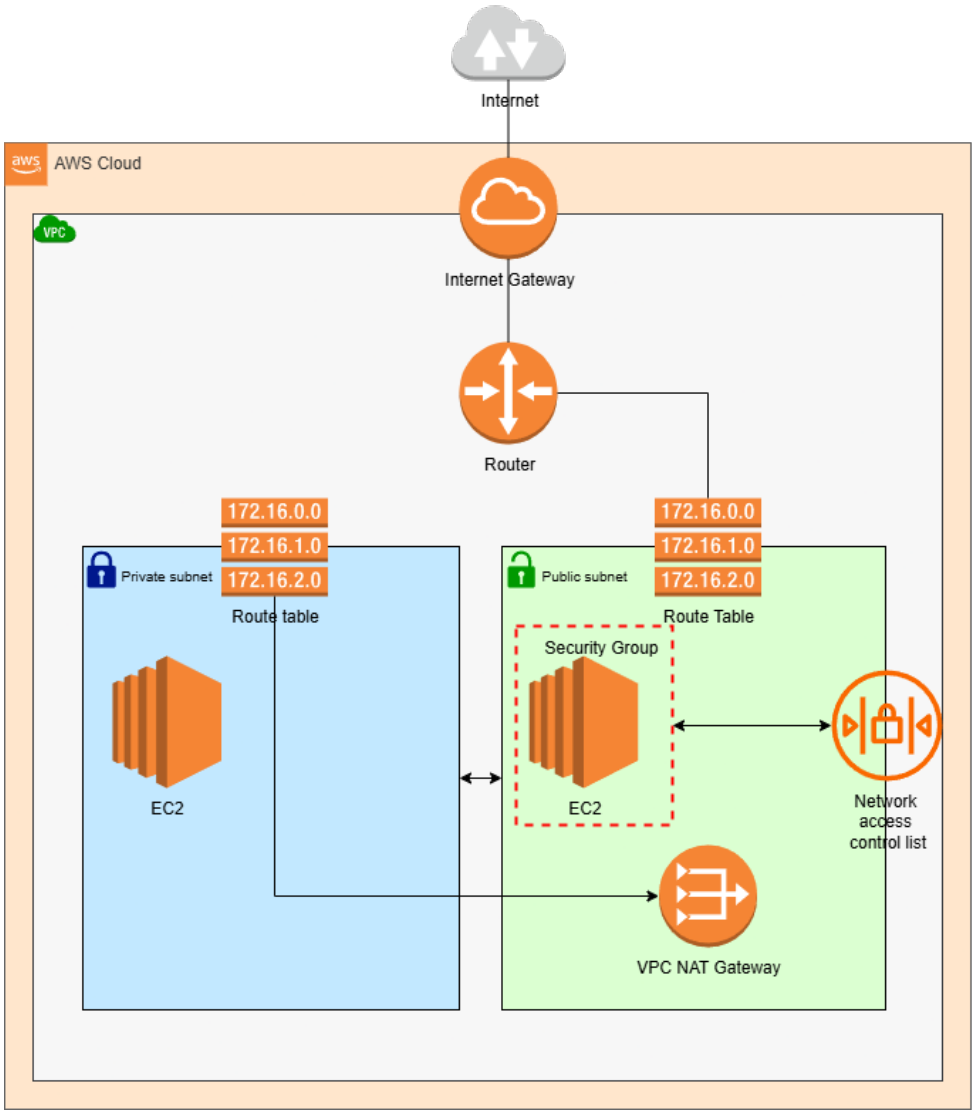

Figure 4.1: Terraform provisioning on AWS (Example)

Terraform provisioning setup on AWS, showing the automated creation and configuration of these key resources.

4.2.2 Migration to an On-Premises Proxmox Cluster

While AWS provided a powerful test bed, continuous cloud usage would have incurred in significant costs. Thus, development moved to an on-premises Proxmox virtualization cluster, where a nested Proxmox cluster was created to support this work. This transition

provided full administrative control over the infrastructure, accelerating the development of this work.

However, Terraform's Proxmox support, being community maintained, lacked some advanced capabilities—particularly post-provision configuration management. To overcome these limitations, Ansible was incorporated into the workflow: Terraform handled the creation of VMs, while Ansible took care of software installation and system setup. This involved installing Docker and Docker Swarm, generating and distributing swarm join tokens, and deploying TorchServe with the required models. This two-step approach allowed for flexible, reproducible, and maintainable infrastructure deployment.

4.2.3 Monitoring and Alerting with InfluxDB and Grafana

Once the system was stable in the on-premises Proxmox, monitoring capabilities were added using InfluxDB for time series data storage and Grafana for visualization as shown in figure 4.2. Both tools were deployed via Docker containers within the swarm, ensuring consistency and simplifying upgrades. Grafana dashboards were configured to display real-time system metrics such as CPU usage, memory consumption, and network throughput. Additionally, alerting rules were set up to trigger webhook notifications when specific thresholds were exceeded. These alerts were designed to integrate directly with the scaling mechanism, allowing the system to react to increased load automatically.

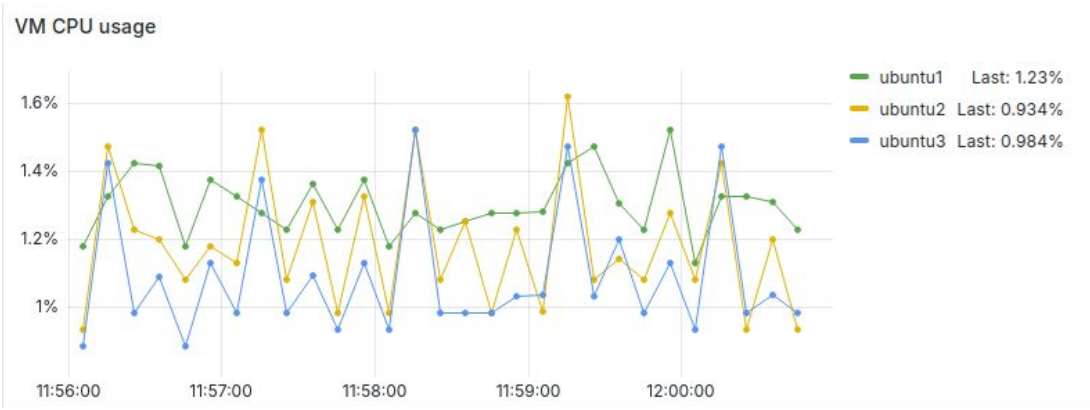

Figure 4.2: Grafana Monitoring CPU

4.2.4 Dynamic Auto-Scaling Mechanism

A key objective of the final architecture was to enable dynamic scaling based on system demand. The auto-scaling process shown in figure 4.3 began with Grafana detecting high resource utilization, which triggered a webhook alert. This alert was sent to a backend service responsible for initiating Terraform scripts that provisioned additional VMs in Proxmox. Once these machines were created, Ansible automatically configured them, installed the required software, and joined them to the Docker Swarm cluster as worker nodes, thereby increasing the available processing capacity as shown in figure 4.4.

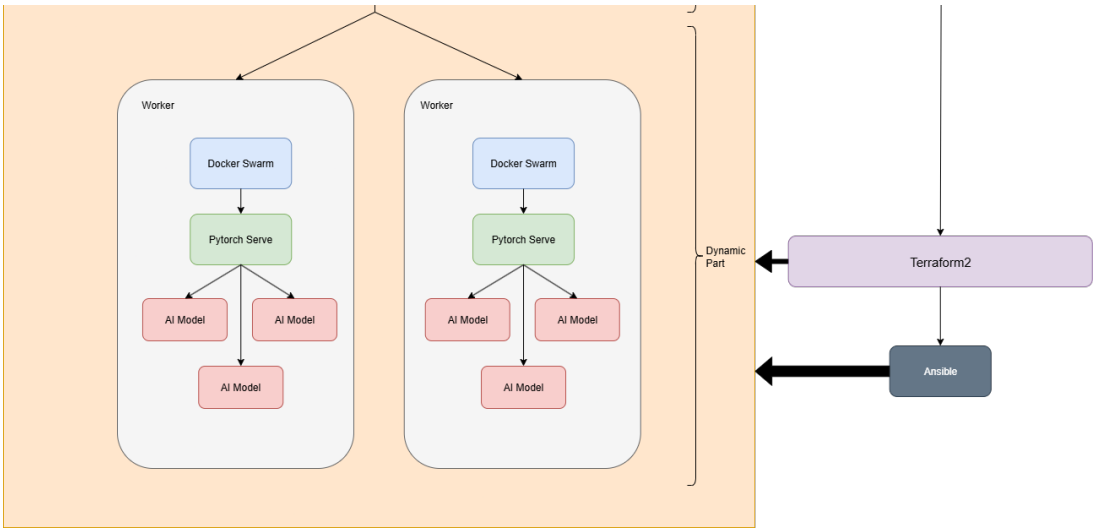

Figure 4.3: Dynamic Infrastructure

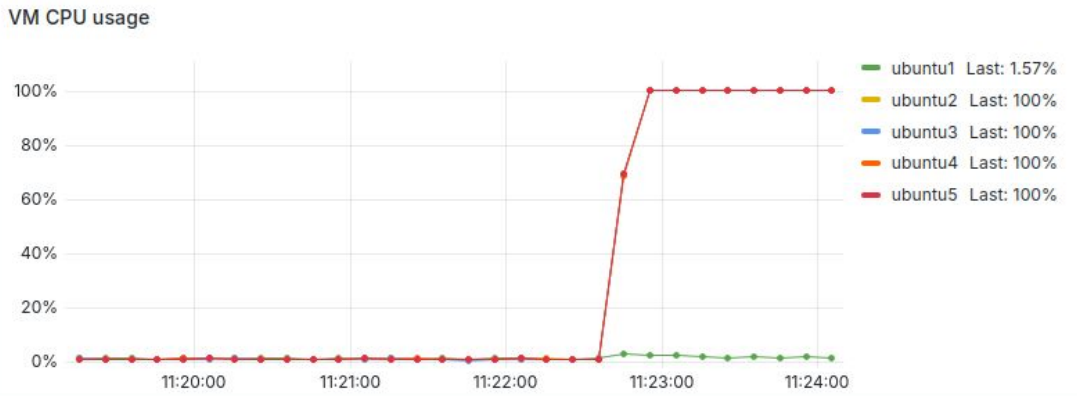

Figure 4.4: Grafana Monitoring Dynamic

While the mechanism worked reliably, some operational challenges emerged. Certain VMs occasionally failed to join the swarm due to misconfigured SSH keys or mismatched join tokens. These issues were mitigated through improved token and key management with Ansible, as well as retry and health check mechanisms to verify successful node integration before considering the scaling event complete.

4.3 Summary

The resulting architecture is a fully automated, scalable AI deployment system built on Docker Swarm and TorchServe. Infrastructure provisioning is handled through a combined Terraform and Ansible workflow, while system monitoring and scaling decisions are managed through InfluxDB, Grafana, and a custom backend API. This design allows the system to adapt dynamically to changing workloads, scaling resources up or down as required while maintaining service reliability. The combination of automation, observability, and orchestration ensures that the system is both flexible for future enhancements and robust enough for real-world operational demands.

Chapter 5

Experiments

This chapter presents a set of tests, and their results, conducted to validate the functionalities implemented and characterize the performance of the architecture deployed. The validation demonstrates that the implemented system fulfills requirements and achieves deployment and scalability objectives.

5.1 Overview and Methodology

The validation process evaluated the system across four dimensions: functional correctness, performance and scalability under load, monitoring effectiveness, and fault tolerance mechanisms. Testing was conducted in the nested Proxmox testbed, with the following characteristics:

- Virtual Hardware per nested Node VM: 2 vCores , 2 GB of RAM, 20 GB of vDisk, Ubuntu 22.04 LTS;
- Cluster Topology: 1 Docker Swarm manager node, 2/4/6 worker nodes;
- Models Deployed: ResNet-18 (lightweight) and DenseNet-161 (computationally intensive); both pretrained models were hosted in TorchServe containers to represent different computational requirements.

The nested Proxmox testbed consisted of a single VM, with enough resources to host the nested VMs used to support the Docker Swram cluster. The physical Proxmox server that hosted the nested Proxmox testbed had 2 x AMD Zen1 7351 CPUs, which although enough to support running this configuration, may have limited its performance, due to their age (and also to the nested nature of the testbed).

5.2 Functional Validation

5.2.1 Infrastructure Provisioning and Management

Terraform and Proxmox automation was validated by creating baseline configurations with specific resource allocations, network configurations, and storage parameters. VMs were successfully provisioned with an average time of 2 minutes 30 seconds per machine. Resource specifications matched Terraform declarations exactly with no drift. Reapplication of configurations produced expected "no changes" output, confirming idempotent behavior.

Template-based VM creation using golden Ubuntu 22.04 LTS images significantly reduced provisioning time compared to full OS installation. Network interfaces were properly configured with VMs receiving assigned static IPs and responding to ping within 30 seconds of boot. Cloud-init scripts executed successfully in all cases.

5.2.2 Configuration Management and Model Deployment

Ansible playbooks containing system updates, Docker Engine installation, Swarm initialization, and TorchServe deployment were executed and re-executed to verify idempotency. Both ResNet-18 and DenseNet-161 models produced predictions matching expected outputs for test images, confirming correct packaging and loading. Container startup time remained consistent across deployments.

5.2.3 Load Balancing and Orchestration

Docker Swarm's capabilities were validated using a standardized workload of 3000 requests with 100 concurrent requests, repeated five times per configuration for statistical reliability. When comparing two-worker against four-worker configurations, the two-worker cluster completed the workload in 2511.32 seconds, while the four-worker cluster achieved 1211.91 seconds a 51.7% reduction that demonstrates near-linear scaling efficiency with minimal overhead.

Request distribution analysis confirmed Swarm's load balancer successfully routes traffic to all healthy replicas in near-equal proportions. When one worker was forcibly shut down, Swarm detected the failure and routed traffic to surviving nodes with minimal request failures once failover completed.

5.2.4 Monitoring and Alerting Integration

The monitoring infrastructure (InfluxDB, Grafana, Telegraf) achieved over 99.9% metric collection reliability during 72-hour continuous operation. Grafana alert rules configured with CPU thresholds (80% scale-out, 40% scale-in) fired correctly with no false positives. The webhook listener successfully received and processed all alerts, executing appropriate automation scripts without failures.

5.3 Performance and Load Testing

5.3.1 Comparative Performance Analysis

Performance testing employed the 3000-request workload at 100 concurrency, tested five times per configuration. Four configurations were evaluated – see Table 5.1.

The fixed four-worker configuration achieved 1211.91 seconds with 51.7% reduction over baseline, closely approximating theoretical 2× improvement. The six-worker configuration further improved performance to 705.28 seconds, representing a 71.9% reduction and demonstrating continued near-linear scaling with 3× resources yielding 3.56×

Table 5.1: Performance Comparison Across Cluster Configurations

Configuration	Total Time	Improvement	Resource Efficiency
2 Workers (Baseline)	2511.32 s	—	Baseline
Dynamic Scaling	1922.78 s	23.4% reduction	Variable resources, 1.31× throughput
4 Workers (Fixed)	1211.91 s	51.7% reduction	2× resources, 2.07× throughput
6 Workers (Fixed)	705.28 s	71.9% reduction	3× resources, 3.56× throughput

throughput. The minimal efficiency loss across configurations is attributable to minor orchestration overhead.

As show in Figure 5.1 the dynamic scaling configuration achieved 1922.78 seconds (23.4% improvement over baseline). This slower performance versus fixed four-worker setup must be interpreted within the dynamic scaling design philosophy. The system begins with two workers, then provisions additional capacity when CPU crosses 80% threshold. However, provisioning time (approximately 2 minutes 30 seconds for complete VM provisioning and Ansible configuration) means additional capacity isn't instantaneous. During provisioning, the system continues with two-worker capacity, creating temporary performance penalty.

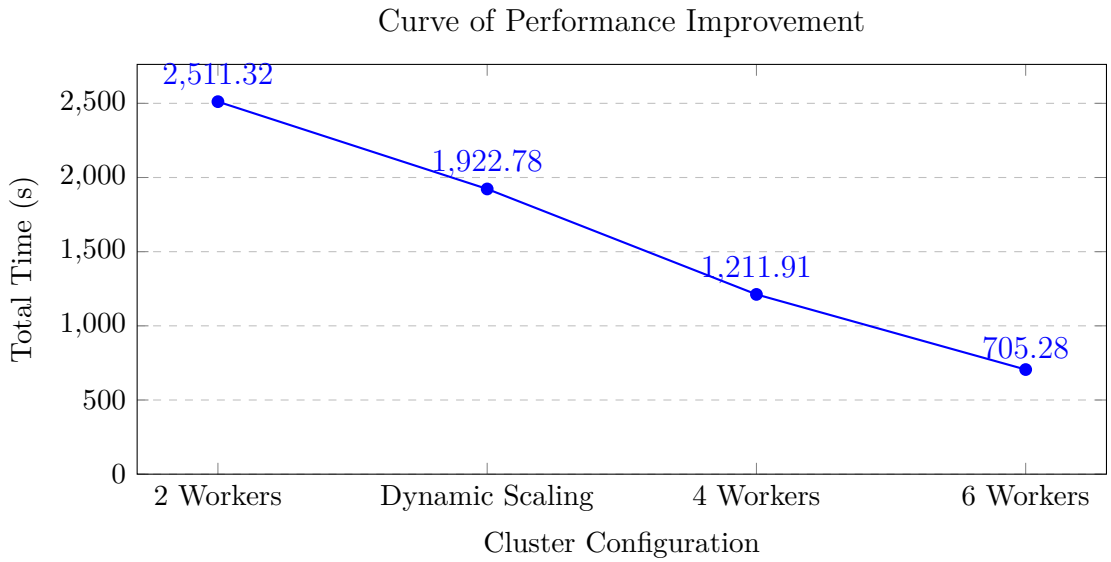

Figure 5.1: Improvement curve

5.3.2 Dynamic Scaling Effectiveness

The 58.6% performance gap between dynamic scaling and fixed four-worker configuration represents provisioning latency cost. However, this misses dynamic scaling's crucial advantage: resource efficiency during variable load. Fixed four-worker configurations consume resources continuously regardless of demand. Dynamic configurations operate with two workers during low-demand periods, provisioning additional capacity only when needed.

For workloads with variable demand patterns, dynamic scaling provides significant efficiency advantages. Consider a 24-hour period with 6 hours peak demand (requiring 4 workers) and 18 hours off-peak (requiring 2 workers). Fixed configuration consumes 96 worker-hours (4×24), while dynamic consumes approximately 54 worker-hours ($2 \times 18 + 4 \times 6$) a 43.8% reduction in resource consumption while maintaining peak load capability.

5.3.3 Resource Utilization Analysis

CPU emerged as the primary bottleneck limiting throughput, with profiling revealing CPU time dominated by image preprocessing, neural network forward passes, and post-processing. Memory remained stable and never approached saturation, indicating it's not a bottleneck. Network throughput remained well below interface limits, and disk I/O stayed minimal during steady-state. CPU capacity is the sole bottleneck, with substantial headroom in other resource dimensions.

This validates CPU-based scaling decisions and confirms horizontal scaling directly addresses the primary constraint. Headroom in other resources suggests infrastructure could support more intensive workloads without proportional increases in memory or network capacity.

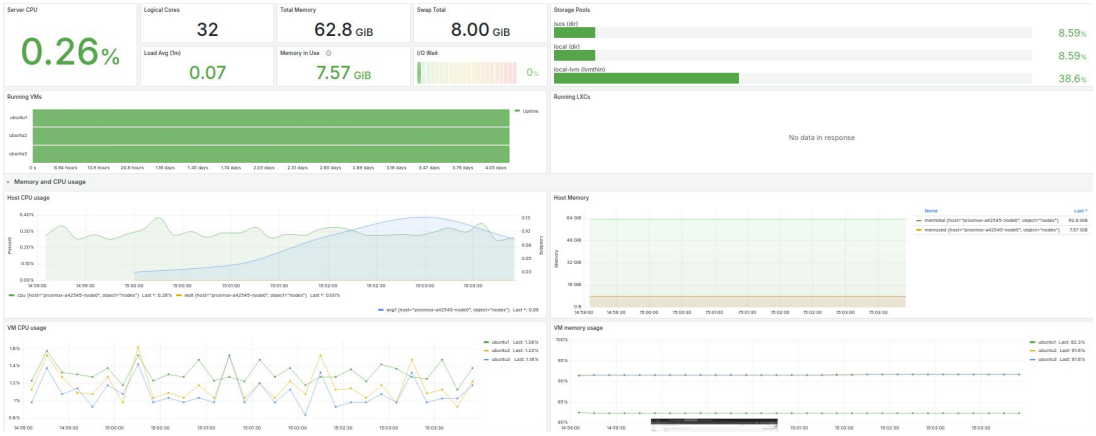

Figure 5.2: Grafana Dashboard

5.4 Monitoring Effectiveness and System Resilience

5.4.1 Monitoring System Validation

The monitoring subsystem provided operational visibility with high reliability. A single Grafana dashboard (Figure 5.2), connected directly to the Proxmox environment, delivered comprehensive visibility into the infrastructure. From this dashboard, it was possible to track resource utilization across all VMs, observe which instances were currently running, and monitor the creation or deletion of machines as they occurred.

The observability setup proved valuable for troubleshooting during development. For example, the Grafana metrics helped identify abnormal resource usage patterns, allowing correlation with logs to pinpoint inefficient processes or performance bottlenecks. Network and system-level metrics further enabled quick detection of issues affecting latency or workload distribution.

5.4.2 Fault Tolerance Assessment

Worker Node Failures: Swarm detected node failures based on missed heartbeats. During detection windows, a small percentage of requests timed out. Once detected, Swarm stopped routing to failed nodes, and remaining workers absorbed the load. Swarm automatically rescheduled lost replicas, resulting in brief service disruption and minimal

request failures.

Container Failures: Swarm detected health check failures and automatically restarted containers. During restart, requests were distributed among the remaining healthy replicas, with no failures due to effective load balancing around unhealthy containers.

Network Partitions: Partitioned nodes were marked unreachable after missed heartbeats. Containers continued running but could not receive work, prompting Swarm to reschedule workloads to reachable nodes. Brief latency increases occurred during rescheduling, with minimal request failures. When partitions resolved, nodes automatically rejoined without manual intervention.

5.5 Key Findings and Lessons Learned

5.5.1 Technical Achievements

The validation demonstrates:

- Self-regulating AI model deployment is achievable using open-source tools and modest infrastructure;
- Fully automated feedback loops (metrics → alerts → infrastructure changes) operate reliably;
- Cloud-agnostic architecture enables flexible deployment across environments;
- Docker Swarm delivers production-grade resilience for medium-scale workloads;
- Comprehensive observability proves essential for operations and development productivity.

Grafana dashboards served not only as operational tools but also communication aids, making system behavior immediately understandable to technical and non-technical audiences.

5.5.2 Architectural Trade-offs

Docker Swarm prioritized simplicity and rapid development over advanced capabilities. Swarm's straightforward setup enabled quick prototyping but revealed limitations including lack of GPU-aware scheduling, limited traffic management granularity, and simpler autoscaling compared to sophisticated platforms.

Scaling entire VMs provides strong isolation but introduces measurable latency (approximately 2 minutes 30 seconds for complete provisioning and configuration). Container-level scaling responds in seconds but requires spare capacity on existing nodes. The optimal strategy is likely hybrid: fast container scaling for short-duration spikes, followed by VM scaling for sustained increases.

Using separate tools (Terraform for provisioning, Ansible for configuration) introduced integration complexity but provided flexibility. The 2-tool approach proved workable for this work's scale, but managing handoffs consumed development effort.

5.5.3 Alert Threshold Tuning

Determining appropriate thresholds (80% CPU scale-out, 40% scale-in) required iterative experimentation. Initial values caused delayed responses or excessive oscillation. Final values balance responsiveness and stability but are workload-dependent.

Key lesson: threshold tuning must account for scaling latency. Given multi-minute provisioning times, alerts must fire while demand rises rather than after saturation occurs. Future work could implement adaptive thresholds adjusting based on observed provisioning times, workload patterns, or time-of-day forecasts.

5.5.4 Development Process Insights

Investing in comprehensive monitoring early proved invaluable. Grafana dashboards enabled rapid debugging including identifying why scaling wasn't triggering, diagnosing node joining problems, and understanding performance characteristics. For future projects, establishing observability infrastructure should be among first tasks.

Terraform and Ansible's declarative, idempotent design made the system robust to partial failures. When scaling events were interrupted, just re-running automation converged to desired state without manual cleanup or duplicate resource risks.

The phased development approach, starting with manually created clusters, then Terraform-based, then Ansible-configured, then adding monitoring, and finally enabling auto-scaling, validated each component before adding complexity. This prevented "big bang integration" scenarios where many components fail simultaneously.

5.5.5 Limitations and Future Directions

While the system meets its core functional requirements, several initial goals were deferred or simplified due to time and infrastructure constraints.

GPU-accelerated inference was not implemented, primarily due to infrastructure complexity and orchestration limitations. The current deployment operates on CPU-only nodes, which restricts throughput and scalability for compute-intensive deep learning models.

The current scaling logic relies on a single metric (CPU) with a fixed threshold. Although functional, this approach is limited in adaptability. More advanced strategies such as latency-based scaling, predictive scaling using time-series forecasting, or multi-metric composite policies could significantly improve responsiveness, efficiency, and cost-effectiveness under dynamic workloads.

From a security perspective, only partial measures have been implemented, mainly focused on ensuring image integrity and secure access control. Specifically:

- **Image and Dependency Integrity:** Container images are signed and stored in trusted registries to mitigate the risk of tampered or malicious images.
- **Access Control and Authentication:** Centralized authentication and role-based access control (RBAC) are implemented to manage user permissions within the orchestration environment.

Further security enhancements are planned for future iterations, including:

- **Transport Security:** Implementing TLS encryption for all service-to-service and API communications to prevent data interception.
- **Monitoring and Intrusion Detection:** Deploying real-time security monitoring, anomaly detection, and log auditing to detect and respond to threats.
- **Audit Logging and Compliance:** Enabling comprehensive audit trails and log retention policies to support traceability and compliance requirements.

Future iterations could also implement fast container-level scaling prior to VM-level scaling, deploy multi-manager clusters from the outset for realistic fail over testing, and evaluate performance across a broader range of model types beyond image classification. Enhanced automation and integration with secure CI/CD pipelines could further streamline deployment and improve maintainability.

5.6 Summary

The experimental validation demonstrated that the proposed architecture successfully achieved automated deployment, monitoring, and scaling of deep learning services using open-source technologies. Tests confirmed that the system met all core functional requirements, with infrastructure provisioning, configuration management, and model deployment executing reliably across multiple environments. Terraform and Ansible automation ensured consistent provisioning and idempotent configuration, while Docker Swarm provided a straightforward yet effective orchestration layer for managing workloads.

Performance and scalability evaluations highlighted the system's ability to efficiently handle increased load through horizontal scaling. The four-worker and six-worker configurations achieved near-linear performance improvements compared to the two-worker

baseline, reducing total execution times by 51.7% and 71.9%, respectively. Dynamic scaling demonstrated resource efficiency advantages during variable workloads, maintaining acceptable response times while minimizing idle capacity. Despite the provisioning delay introduced by VM creation, the results validate that automated scaling can be achieved effectively within the given infrastructure constraints.

Monitoring and observability played a central role in ensuring system transparency and operational stability. A single Grafana dashboard connected to the Proxmox environment provided real-time visibility into system metrics, enabling effective analysis of CPU utilization, active VMs, and scaling events. This monitoring setup not only supported troubleshooting but also validated the feedback loop between performance metrics and automation triggers. Fault-tolerance tests further confirmed the system's resilience, as Swarm successfully recovered from worker and container failures with minimal request disruption.

From a security standpoint, only foundational measures have been implemented, specifically image integrity verification and access control mechanisms through trusted registries and role-based authentication. These represent a solid baseline for an academic environment; however, production deployments would require stronger protections. Future enhancements will include the addition of TLS encryption for service communications, real-time intrusion detection, and comprehensive audit logging to ensure end-to-end security and compliance.

Overall, the experiments validated the system's ability to autonomously provision, configure, monitor, and scale ML services in a controlled test environment. The architecture demonstrated strong reliability, adaptability, and observability, though limited by CPU-only operation and simplified scaling logic. Future work will focus on improving security, reducing provisioning latency, supporting GPU-accelerated workloads, and refining adaptive scaling policies to enhance responsiveness and efficiency under dynamic workloads.

Chapter 6

Conclusion

This dissertation presented the conception, development, and validation of an autonomous and scalable framework for deploying AI models. The work started from a practical challenge: how to operationalize AI services in a way that could adapt dynamically to changing workloads without continuous human intervention. Through the integration of Infrastructure-as-Code, container orchestration, and observability-driven automation, an end-to-end system was built using exclusively open-source technologies. The implemented prototype demonstrated that it is possible to achieve a self-regulating infrastructure capable of provisioning resources, deploying containerized services, and adjusting its operation automatically according to real-time metrics. The successful deployment and testing across both AWS and Proxmox environments confirmed not only the feasibility of the proposed approach but also its portability across cloud and on-premises infrastructures.

Throughout the development, several architectural decisions shaped the system's performance and usability. The choice of Docker Swarm, for example, allowed for a simpler and faster implementation process, enabling the validation of the main concepts within the available time and resource constraints. This decision, however, came with trade-offs in terms of scheduling sophistication and fine-grained resource control when compared to Kubernetes. Likewise, the use of a VM-based scaling mechanism ensured resource isolation and predictability but introduced additional latency during provisioning. These

results illustrate the balance that had to be achieved between implementation complexity, operational efficiency, and maintainability. Within the context of this project, this balance proved effective for educational and medium-scale production settings, while also suggesting that more advanced scaling strategies would be needed for latency-sensitive or large-scale environments.

A key outcome of this work was the emphasis placed on observability as a central component of the system's design. Monitoring and alerting were not merely supporting elements but essential tools for understanding, validating, and improving the architecture throughout its development. This experience highlighted that observability is fundamental for autonomous systems, as it enables both transparency and trust in automated decision-making processes. Integrating observability with automated remediation mechanisms provided valuable insights into how infrastructures can evolve towards greater autonomy, while also revealing the importance of adaptive thresholds and context-aware alerting policies.

Beyond the technical achievements, this dissertation contributes to the broader field of ML Operations by demonstrating that robust, production-oriented AI deployment can be achieved without resorting to proprietary or overly complex solutions. The implemented architecture shows that open-source tools, when properly integrated, can deliver autonomy, scalability, and resilience even under constrained conditions. This is particularly relevant for institutions or organizations seeking sovereignty, flexibility, and cost-effectiveness in their AI operations. Nevertheless, the limitations identified—such as the absence of GPU-aware scheduling, limited responsiveness to transient loads, and the need for stronger security mechanisms—indicate clear paths for future improvement.

The work developed in this dissertation establishes a solid foundation for future research and development, with natural extensions including the integration of container-level autoscaling, predictive scaling based on workload analytics, and more comprehensive security measures such as end-to-end encryption, audit logging, and intrusion detection. These enhancements would allow the system to evolve into a production-ready platform suitable for more demanding environments.

Bibliography

[1] Y. Tian, Z. Zhang, Y. Yang, *et al.*, "An Edge-Cloud Collaboration Framework for Generative AI Service Provision With Synergetic Big Cloud Model and Small Edge Models," *IEEE Network*, vol. 38, no. 5, pp. 37–46, Sep. 2024, ISSN: 0890-8044, 1558-156X. DOI: 10.1109/MNET.2024.3420755. [Online]. Available: https://ieeexplore.ieee.org/document/10577141/ (visited on 08/12/2025).

[2] M. N. Vivekananda, P. Ashok Shidlyali, and V. V. Malgi, "Generative AI Meets Large Language Models: Evolution, Challenges, and Future Directions," in *2025 Global Conference in Emerging Technology (GINOTECH)*, PUNE, India: IEEE, May 2025, pp. 1–6, ISBN: 9798331507756. DOI: 10.1109/GINOTECH63460.2025.11076892. [Online]. Available: https://ieeexplore.ieee.org/document/11076892/ (visited on 08/12/2025).

[3] Q. Zheng, J. Wang, and Y. Shen, "Overview and Trend of Large-scale Model Deployment Mode," in *2024 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, Toronto, ON, Canada: IEEE, Jun. 2024, pp. 1–6, ISBN: 9798350364262. DOI: 10.1109/BMSB62888.2024.10608242. [Online]. Available: https://ieeexplore.ieee.org/document/10608242/ (visited on 08/12/2025).

[4] S. Jhingran, N. Bansal, R. Chaturvedi, A. Singh, and Y. Arora, "Decentralized Generative AI Model Deployment Using Microservices," in *2024 International Conference on Artificial Intelligence and Quantum Computation-Based Sensor Application (ICAIQSA)*, Nagpur, India: IEEE, Dec. 2024, pp. 1–5. DOI: 10.1109/

icaiqsa64000.2024.10882424. [Online]. Available: https://ieeexplore.ieee.org/document/10882424/ (visited on 07/19/2025).

[5] C. Sisimayi and M. V. Visaya, "Deployment of AI Models and a Mapper Algorithm to Enhance Clinical Decision-Making," in *2024 International Conference on Sustainable Technology and Engineering (i-COSTE)*, Perth, Australia: IEEE, Dec. 2024, pp. 1–6. DOI: 10.1109/i-coste63786.2024.11025118. [Online]. Available: https://ieeexplore.ieee.org/document/11025118/ (visited on 07/19/2025).

[6] L. Wang, X. Ren, C. Zhao, F. Zhao, and S. Yang, "MPDM: A Multi-Paradigm Deployment Model for Large-Scale Edge-Cloud Intelligence," *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8773–8785, May 2023, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 2327-4662, 2372-2541. DOI: 10.1109/jiot.2022.3232582. [Online]. Available: https://ieeexplore.ieee.org/document/10000402/ (visited on 07/19/2025).

[7] A. Christidis, S. Moschoyiannis, C.-H. Hsu, and R. Davies, "Enabling Serverless Deployment of Large-Scale AI Workloads," *IEEE Access*, vol. 8, pp. 70 150–70 161, 2020, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 2169-3536. DOI: 10.1109/access.2020.2985282. [Online]. Available: https://ieeexplore.ieee.org/document/9055400/ (visited on 07/19/2025).

[8] A. J. Vasquez, K. Drake, and J. Bramante, "AI/ML Deployment at the Edge for Run-by-Run and Real-Time Analysis," in *2024 International Symposium on Semiconductor Manufacturing (ISSM)*, Tokyo, Japan: IEEE, Dec. 2024, pp. 1–4. DOI: 10.1109/issm64832.2024.10875005. [Online]. Available: https://ieeexplore.ieee.org/document/10875005/ (visited on 07/19/2025).

[9] A. Christidis, S. Moschoyiannis, C.-H. Hsu, and R. Davies, "Enabling Serverless Deployment of Large-Scale AI Workloads," *IEEE Access*, vol. 8, pp. 70 150–70 161, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2985282. [Online]. Available: https://ieeexplore.ieee.org/document/9055400/ (visited on 08/12/2025).

[10] Y. Tian, Z. Zhang, Y. Yang, *et al.*, "An Edge-Cloud Collaboration Framework for Generative AI Service Provision With Synergetic Big Cloud Model and Small Edge Models," *IEEE Network*, vol. 38, no. 5, pp. 37–46, Sep. 2024, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 0890-8044, 1558-156X. DOI: 10.1109/mnet.2024.3420755. [Online]. Available: https://ieeexplore.ieee.org/document/10577141/ (visited on 07/19/2025).

[11] C. Sisimayi and M. V. Visaya, "Deployment of AI Models and a Mapper Algorithm to Enhance Clinical Decision-Making," in *2024 International Conference on Sustainable Technology and Engineering (i-COSTE)*, Perth, Australia: IEEE, Dec. 2024, pp. 1–6, ISBN: 9798331517335. DOI: 10.1109/i-COSTE63786.2024.11025118. [Online]. Available: https://ieeexplore.ieee.org/document/11025118/ (visited on 08/12/2025).

[12] S. Jhingran, N. Bansal, R. Chaturvedi, A. Singh, and Y. Arora, "Decentralized Generative AI Model Deployment Using Microservices," in *2024 International Conference on Artificial Intelligence and Quantum Computation-Based Sensor Application (ICAIQSA)*, Nagpur, India: IEEE, Dec. 2024, pp. 1–5, ISBN: 9798331517953. DOI: 10.1109/ICAIQSA64000.2024.10882424. [Online]. Available: https://ieeexplore.ieee.org/document/10882424/ (visited on 08/12/2025).

[13] M. Chahoud, H. Sami, A. Mourad, H. Otrok, J. Bentahar, and M. Guizani, "Towards On-Demand Deployment of Multiple Clients and Heterogeneous Models in Federated Learning," in *2023 International Wireless Communications and Mobile Computing (IWCMC)*, Marrakesh, Morocco: IEEE, Jun. 2023, pp. 1556–1561, ISBN: 9798350333398. DOI: 10.1109/IWCMC58020.2023.10182555. [Online]. Available: https://ieeexplore.ieee.org/document/10182555/ (visited on 08/12/2025).

[14] Y. Wang, W. Song, Y. Yang, C. Mahmoudi, S. Shekhar, and K. P. Birman, "Dash: A Low Code Development Platform for AI Applications in Industry," in *2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, New York, NY, USA: IEEE, Oct. 2023, pp. 0072–0081, ISBN:

9798350304138. DOI: 10.1109/UEMCON59035.2023.10316092. [Online]. Available: https://ieeexplore.ieee.org/document/10316092/ (visited on 08/12/2025).

[15] Y. Liang, P. Yang, Y. He, and F. Lyu, "Resource-Efficient Generative AI Model Deployment in Mobile Edge Networks," in *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, Cape Town, South Africa: IEEE, Dec. 2024, pp. 2647–2652, ISBN: 9798350351255. DOI: 10.1109/GLOBECOM52923.2024.10901571. [Online]. Available: https://ieeexplore.ieee.org/document/10901571/ (visited on 08/12/2025).

[16] A. J. Vasquez, K. Drake, and J. Bramante, "AI/ML Deployment at the Edge for Run-by-Run and Real-Time Analysis," in *2024 International Symposium on Semiconductor Manufacturing (ISSM)*, Tokyo, Japan: IEEE, Dec. 2024, pp. 1–4, ISBN: 9798331510589. DOI: 10.1109/ISSM64832.2024.10875005. [Online]. Available: https://ieeexplore.ieee.org/document/10875005/ (visited on 08/12/2025).

[17] L. Wang, X. Ren, C. Zhao, F. Zhao, and S. Yang, "MPDM: A Multi-Paradigm Deployment Model for Large-Scale Edge-Cloud Intelligence," *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8773–8785, May 2023, ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2022.3232582. [Online]. Available: https://ieeexplore.ieee.org/document/10000402/ (visited on 08/12/2025).

[18] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – Vulnerability Analysis," en, *Computer Communications*, vol. 122, pp. 30–43, Jun. 2018, ISSN: 01403664. DOI: 10.1016/j.comcom.2018.03.011. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0140366417300956 (visited on 08/12/2025).

[19] M. De Benedictis and A. Lioy, "Integrity verification of Docker containers for a lightweight cloud environment," en, *Future Generation Computer Systems*, vol. 97, pp. 236–246, Aug. 2019, ISSN: 0167739X. DOI: 10.1016/j.future.2019.02.026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18327201 (visited on 08/12/2025).

[20] P. Poczekajło, R. Suszyński, and A. Antosz, "The use of modern methods of virtualization and shared network functions to optimize the resources used," en, *Procedia Computer Science*, vol. 225, pp. 2773–2780, 2023, ISSN: 18770509. DOI: 10.1016/j.procs.2023.10.269. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050923014278 (visited on 08/12/2025).

[21] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," en, *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020, ISSN: 18770509. DOI: 10.1016/j.procs.2020.04.152. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050920311315 (visited on 08/12/2025).

[22] I. Gorton and V. Teja Rayavarapu, "Foundations of Scalable Software Architectures," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 233–236. DOI: 10.1109/icsa-c54293.2022.00052. [Online]. Available: https://ieeexplore.ieee.org/document/9779797/ (visited on 07/21/2025).

[23] W.-T. Tsai, Y. Huang, X. Bai, and J. Gao, "Scalable Architectures for SaaS," in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, Shenzhen, TBD, China: IEEE, Apr. 2012, pp. 112–117. DOI: 10.1109/isorcw.2012.44. [Online]. Available: http://ieeexplore.ieee.org/document/6196111/ (visited on 07/21/2025).

[24] M. Arango and B. Kaponig, "Ultra-scalable architectures for Telecommunications and Web 2.0 services," in *2009 13th International Conference on Intelligence in Next Generation Networks*, Bordeaux, France: IEEE, Oct. 2009, pp. 1–7. DOI: 10.1109/icin.2009.5357107. [Online]. Available: http://ieeexplore.ieee.org/document/5357107/ (visited on 07/21/2025).

[25] S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform: An Introduction to Infrastructure Automation, Integration, and Scalability using Terraform*, en. Berkeley, CA: Apress, 2024, ISBN: 9798868800733 9798868800740. DOI: 10.1007/979-8-

8688-0074-0. [Online]. Available: https://link.springer.com/10.1007/979-8-8688-0074-0 (visited on 05/07/2025).

[26] M. Fayad, H. Hamza, and H. Sanchez, "Towards scalable and adaptable software architectures," in *IRI -2005 IEEE International Conference on Information Reuse and Integration, Conf, 2005.*, Las Vegas, NV, USA: IEEE, pp. 102–107. DOI: 10.1109/iri-05.2005.1506457. [Online]. Available: http://ieeexplore.ieee.org/document/1506457/ (visited on 07/21/2025).

[27] A. S. A. Alghawli and T. Radivilova, "Resilient cloud cluster with DevSecOps security model, automates a data analysis, vulnerability search and risk calculation," en, *Alexandria Engineering Journal*, vol. 107, pp. 136–149, Nov. 2024, Publisher: Elsevier BV, ISSN: 1110-0168. DOI: 10.1016/j.aej.2024.07.036. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1110016824007567 (visited on 07/29/2025).

[28] R. Su and X. Li, *Modular Monolith: Is This the Trend in Software Architecture?* Version Number: 1, 2024. DOI: 10.48550/ARXIV.2401.11867. [Online]. Available: https://arxiv.org/abs/2401.11867 (visited on 08/11/2025).

[29] S. Bailuguttu, A. S. Chavan, O. Pal, K. Sannakavalappa, and D. Chakrabarti, "Comparing performance of bastion host on cloud using Amazon web services vs terraform," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 30, no. 3, p. 1722, Jun. 2023, ISSN: 2502-4760, 2502-4752. DOI: 10.11591/ijeecs.v30.i3.pp1722-1728. [Online]. Available: https://ijeecs.iaescore.com/index.php/IJEECS/article/view/29400 (visited on 06/05/2024).

[30] L. Berton, *Practical Ansible Automation Handbook*, eng. Delhi: Orange Education PVT Ltd, 2023, ISBN: 978-93-88590-89-1.

[31] A. Klos, M. Rosenbaum, and W. Schiffmann, "Scalable and Highly Available Multi-Objective Neural Architecture Search in Bare Metal Kubernetes Cluster," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Portland, OR, USA: IEEE, Jun. 2021, pp. 605–610. DOI: 10.1109/ipdpsw52791.

2021.00094. [Online]. Available: https://ieeexplore.ieee.org/document/9460405/ (visited on 07/21/2025).

[32] N. Singh, A. Singh, and V. Rawat, "Deploying Jenkins, Ansible and Kubernetes to Automate Continuous Integration and Continuous Deployment Pipeline," in *2022 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, Delhi, India: IEEE, Dec. 2022, pp. 1–5, ISBN: 9798350332247. DOI: 10.1109/SOLI57430.2022.10294378. [Online]. Available: https://ieeexplore.ieee.org/document/10294378/ (visited on 08/11/2025).

[33] P. Balasubramanian, S. Nazari, D. K. Kholgh, A. Mahmoodi, J. Seby, and P. Kostakos, "A cognitive platform for collecting cyber threat intelligence and real-time detection using cloud computing," en, *Decision Analytics Journal*, vol. 14, p. 100 545, Mar. 2025, ISSN: 27726622. DOI: 10.1016/j.dajour.2025.100545. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2772662225000013 (visited on 08/12/2025).

[34] A. Mehdi and R. Walia, "Terraform: Streamlining Infrastructure Deployment and Management Through Infrastructure as Code," in *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, Greater Noida, India: IEEE, Nov. 2023, pp. 851–856, ISBN: 9798350306118. DOI: 10.1109/ICCCIS60361.2023.10425616. [Online]. Available: https://ieeexplore.ieee.org/document/10425616/ (visited on 08/12/2025).

[35] A. Mehdi and R. Walia, "Terraform: Streamlining Infrastructure Deployment and Management Through Infrastructure as Code," in *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, Greater Noida, India: IEEE, Nov. 2023, pp. 851–856, ISBN: 9798350306118. DOI: 10.1109/ICCCIS60361.2023.10425616. [Online]. Available: https://ieeexplore.ieee.org/document/10425616/ (visited on 06/04/2024).

[36] S. Bhatia and C. Gabhane, *Reverse engineering with Terraform: an introduction to infrastructure automation, integration, and scalability using Terraform*, eng. Berkeley, CA: Apress L. P, 2024, ISBN: 9798868800733 9798868800740.

[37] A. Cepuc, R. Botez, O. Craciun, I.-A. Ivanciu, and V. Dobrota, "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes," in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, Bucharest, Romania: IEEE, Dec. 2020, pp. 1–6, ISBN: 978-0-7381-1265-7. DOI: 10.1109/RoEduNet51892.2020.9324857. [Online]. Available: https://ieeexplore.ieee.org/document/9324857/ (visited on 08/11/2025).