Quality Festival 2025
ISBN 978-86-6335-121-9

Tural Muzafarov[1]

Research paper
DOI – 10.24874/QF.25.142

THE IMPLEMENTATION OF ARTIFICIAL INTELLIGENCE (AI) IN MODERN SOFTWARE ECOSYSTEMS

Abstract: *The implementation of Artificial Intelligence (AI) in modern software ecosystems goes beyond model accuracy, requiring the development and coordination of strong, scalable backend infrastructures for smooth integration and deployment. This article outlines the fundamental architectural structures supporting AI-enabled systems, highlighting the crucial function of backend engineering in facilitating the interface between machine learning components and production environments. It rigorously analyses prominent technical difficulties, such as the management of model versioning and deployment lifecycles, reduction of inference latency at scale, and maintenance of data integrity throughout dynamic processing pipelines. In anticipating future advancements, the discussion converges on the paradigm of edge computing as a strategic enabler for decentralised, low-latency AI inference, highlighting its potential to reshape backend infrastructure in distributed intelligent systems.*

Keywords: *model versioning and deployment, inference latency, pipeline reliability, edge computing*

1. Introduction

The very introduction of Artificial Intelligence (AI) into the modern software ecosystem has transitioned projects from isolating model training to creating entire systems that are end-to-end. In thinking about how organizations are using AI models in production, the progression of an AI model has changed from a mere AI model to intelligent systems that can automate or augment work or change experiences for users. This shift necessitates not just good models but also operational pipelines that can efficiently deploy, monitor, and scale AI capabilities continuously. (Rausch & Dustdar, 2019) The operational needs that can be supported differently in AI products cannot be serviced efficiently with old-fashioned software engineering approaches. In AI types of software products, we have data sets, model binaries, and continuous retraining that we do not have in regular applications. Moreover, AI models are quite sensitive to changes in input data and environments, which can make stability a significantly more complex aim than with purely algorithmic software systems. Continuing the trend in the industry, one way of developing systematic processes to achieve AI at scale is through Machine Learning Operations (MLOps) aimed at the systematic integration, deployment, and management of AI assets. Within AI-based applications, backend systems are the bedrock that

[1] Corresponding author: Tural Muzafarov
Email: turalmuzefferli4@gmail.com

facilitates the reliable integration and delivery of machine learning modules. The backend is responsible for orchestrating foundational workflows, such as model serving, versioning, real-time inference handling, data pipeline orchestration, and system monitoring. A robust, reliable backend design ensures that these models are deployed with efficiency, and their lifecycle (their governance, updating, accessing, scaling, etc.) is achieved smoothly. There are also system level challenges that the backend must cater to – for instance, inference latency, asynchronous processing, availability, and fault tolerance while running in production. As AI applications begin to encompass even more real-time distributed use cases (e.g., autonomous vehicles, recommendation engines, intelligent optical character recognition), there is a growing sophistication and complexity around backend infrastructure that will ultimately be a requirement, not an added benefit for AI applications to be sustainably adopted.

2. Main components of the AI systems

The design of an AI system incorporates greater depth than traditional software because it includes data-driven components, such as machine learning models, in a production environment. A viable AI system architecture should be capable of supporting the complete model lifecycle of development, deployment, inference, measurement, monitoring for drift and faults, and maintaining the system's continuous learning without overwhelming stakeholders. This research organised AI architecture components into 5 layers.

2.1. Data Layer

The data layer is the prerequisite for every AI system. The data layer is responsible for capturing, storing, managing, and transforming the data that is used for model training and inference. It typically includes data lakes, real-time data streams, feature stores, and ETL (Extract, Transform, Load) pipelines. Making certain that data is high quality, consistent, and identifiable is essential for the reliability and accuracy of artificial intelligence outputs.

2.2. Model Layer

Machine learning model development, version control, and governance are the primary responsibilities of the model layer. It includes several frameworks and tools that make it easier to manage various model versions, store trained artefacts, and train models. This layer usually consists of experiment tracking systems (like MLflow or Weights , Biases) and model registries, which provide repeatability and traceability across the models' lifetimes (Zaharia, et al., 2018)

2.3. Inference Layer

Models must be deployed in environments that can accommodate future requests when training is complete. The serving and inference layer facilitates the process of putting these models into operation. This layer consists of a number of elements, including REST/gRPC APIs that enable applications to use model functionalities and inference servers (such as TensorFlow Serving and TorchServe). At this layer, lowering inference latency, controlling vertical and horizontal scalability, and guaranteeing safe model access are important considerations.

2.4. Orchestration And Automation Layer

Orchestration will be one of the most crucial components since artificial intelligence systems undergo constant change, are frequently updated, retrained as needed, and are based on frequently shifting datasets. Workflows like retraining schedules, model validation, and releases that come after an

A/B testing or canary releases procedure will be automated by this layer.

2.5. Monitoring And Feedback Layer

Yielding the greatest possible performance and sustained dependability depends on continuous monitoring of model behaviour following deployment. This feature tracks several measures like model performance, inference latency, system health, data drift, etc. This also lets one construct feedback loops for retraining the model depending on user engagement, environmental changes, and fresh data. Modern artificial intelligence systems commonly employ monitoring solutions for models such as Prometheus, Evidently AI, and WhyLabs. (Breck, Cai, Nielsen, Salib, & D., 2018)

3. Bridging AI systems and backend infrastructures

The previously mentioned architecture needs to be connected to the corporate system, mobile device, or online service that serves as the main application backend for workable AI-enabled solutions. The integration is required to provide AI the performance, scalability, and accessibility it needs in real-world applications. Backend developers create communications platforms, middleware layers, and APIs that link AI models to application services. The AI output's responsiveness, dependability, and user experience are all impacted by how well this integration works. Furthermore, the backend architecture needs to be carefully planned in order to minimize inference latency, maximize integration quality, and preserve appropriate model performance. Building effective data pipelines to transfer data between components, managing traffic around serverless deployment, container orchestration (like Kubernetes), and load balancing with inference servers are the primary tactics that can be used in this situation. Horizontal scalability and elastic resources are essential in dynamic load scenarios, like some real-time AI-powered systems. Furthermore, to reduce bottlenecks and improve system responsiveness, database optimization is required for model input, output, metadata, and feedback. There are numerous options for expediting the transmission of data to the AI modules, including intelligent caching (what to keep cached and how long), indexing strategies, and distributed database architectures. Complete monitoring of all system metrics (inference throughput, queue time for requests, and backend resources used) should be provided as well, to recognize when performance degradation happens and when there is an abnormality. Security and regulatory considerations must also be factored into back-end design decisions, particularly when AI models operate with sensitive or regulated data. This includes enforcing strict authentication, authorization, and encryption policies around the model providing endpoints. Finally, AI systems with automated model retraining and redeployment pipelines will require CI/CD (Continuous Integration/Continuous Deployment) approaches to ensure system flexibility and continuous improvement capabilities. Altogether, these backend engineering practices form the unseen yet critical foundation that enables AI-driven applications to operate reliably, securely, and at scale in production environments.

4. Data pipeline construction

A data pipeline is the framework used to collect data from the identifiable and accessible sources and transport it to the intended destinations while transforming, optimising and aggregating the data simultaneously. It is a common misconception to understand anything that engages a movement of data as any movement of data equal to a data pipeline.
This definition fails to classify the complexity of the datacentric and transformational process in true data pipelines. Data movement is a part of a data

pipeline, and certainly it is one of the components; however, if you limit a data pipeline to the description of moving some data from one place to another, the intention of a data pipeline is not realised. Moving data from point A to point B does not represent a data pipeline; the transforming component to prepare the data for business use is the key component that differentiates data replication from a data pipeline. The point of data pipelines is to make sense of raw data to generate actionable and meaningful information that can inform and drive a business decision. However, with respect to data pipelines, there will always be validly implemented pipelines that differ broadly in design, intentions, and delivery; nevertheless, the development of an efficient pipeline is based on a system of sequential steps that create a foundation for success. Each data pipeline has three abstract parts.

4.1 Ingestion Layer

The data pipeline process begins at its sources, known as the ingestion points. These points are the access points to the pipeline where the data is accessed and gathered from a number of systems. Data sources can be an extremely varied domain and can include data warehouses, relational databases, web analytics, customer relationship management (CRM) systems, social media tools, and sensors from Internet of Things (IoT) devices. Regardless of its origin, data ingestion, both in batch (in discrete intervals) format and streaming format, is the first step in any data pipeline. The ingestion layer supports many types and formats of data, such as:

Batch Data: Traditionally, data was captured and sent to a data processing queue as batches, commonly from static halves such as databases and logs. Batch processing was a good fit for many use cases, but now as the world changes at a faster pace, making decisions based on processing batch data often results in opportunities being lost before they can even be responded to. By the time the data is pulled, processed, and sorted into indexable or traceable batches, it often is already stale and unable to provide real-time, reflective analysis.

Streaming Data: Continuous data that comes from sources like sensors or IoT devices or live feeds from transactions. This data must be processed quickly to provide real-time updates, insights and decision-making, which suits current businesses. Real-time data streaming technologies, including Apache Kafka and Apache Pulsar, are often used for real-time data streaming, which provides low-latency, scalable methods to transfer data throughout the system.

4.2 Transformation and Processing Layer

Subsequent to ingestion, the data is subject to a series of transformations to prepare it for use in business applications. The transformation stages may have a wide variety of processes involved, including, but not limited to, data augmentation, filtering, grouping, aggregation, normalization, sorting, de-duplication, validation, and verification. The aim of these activities is to clean, unify, and otherwise optimize data so it can be analyzed and, in general, support better decision-making. There are 4 data processing levels.

Batch Processing: Batch processing happens when a large set of data is dealt with in unison, as data is stored and can be dealt with at regular intervals. Essentially, data is handled at set intervals and is not instantaneous like real-time data but can be handled with large volumes of data when immediacy is not an issue.

Distributed Processing: When data is being distributed among multiple machines (or servers), we call that distributed processing, which is frequently used when we have large datasets that cannot be stored on a single machine or we want to take advantage of data from multiple devices. Additional benefits include increased fault tolerance, as

processing could continue on other servers if one server fails.

Multi-Processing: Multi-processing is the use of a number of processors in the same physical environment, in contrast to a distributed environment, where the processors are separate systems. One of the possible disadvantages of multi-processing is that a failure of one processor may slow down processing. On the other hand, this way of processing is still relatively safe for sensitive data because processing on a single server is more secure than distributing the processing across separate servers.

Real-Time Processing: Real-time processing is considered when results must be produced immediately, as it is output produced alongside data being processed. The system works very quickly and ignores erroneous records of data and continues processing the next record of data. This is useful for applications that need immediate results but could lead to erroneous data being included in the analysis.

Transaction Processing: Transaction processing, which is a form of processing, is real-time as the purpose is to maintain data accuracy. This distinguished transaction processing from real-time processing, which does not deal with errors and therefore has no purpose in accounting or business environments when it comes to the accuracy of transactions. Transaction processing will stop until all errors are fixed. This may involve features of both hardware and software in the design of the system to effectively help with error correction features and help process resumes after an interruption.

4.3 Destination And Data Sharing Layer

The last phase of a data pipeline deals with the destinations of the data, where the processed data is delivered for analysis and use. Normally, the data is stored in systems like data warehouses or data lakes, where it will be available for analysis and consumed by analytics and data science teams.

Data Warehouse: Some of these have been optimized for storing data with structure, usually in relation-oriented databases. They can support more complex querying and analytical processing and are better suited for business intelligence and reporting. They also offer high performance and scalability. They readily support large amounts of structured related data.

Data Lakes: Data lakes are meant to contain structured, semi-structured, and unstructured data, all part of a flexible and scalable solution. A data lake usually holds raw data in its original format, enabling organizations to ingest and combine vast amounts of data from multiple sources all at once. The data lake approach allows organizations to handle vast amounts of diverse data, enabling advanced analytics, machine learning, exploratory data analysis, and other types of analysis by transforming and processing the data when needed.

In some instances, the pipeline terminates directly in user applications (e.g., data visualization, machine learning, and any number of other systems, such as API endpoints) that consume and otherwise utilize the processed data.

5. Database optimization

Database optimization is essential for AI systems, particularly to minimize latency and improve speed. This involves strategies like indexing, caching, and query optimization to ensure efficient data retrieval and processing.

5.1 Indexing

Indexing involves building data structures that offer fast access to a set or group of records from the set of data or the database, without having to scan every record in a linear fashion. Per a study published in ACM Transactions on Database Systems (TODS),

indexing has been proven to dramatically increase the speed with which queries perform by reducing the time from linear time O(n) to logarithmic time O(log n), or even to constant time in some cases when using hash-based indexing. When executing a query, the optimizer determines the available indexes at the time of execution. If an index exists and it's usable (e.g., if a user filters a result set or sorts a result set on that column), the database will use the index instead of performing a full table scan to yield faster query results and reduce the amount of I/O operations that need to occur.

5.2 Data Partitioning And Sharding

In this study, the primary issue Dividing the database into partitions or shards can distribute the data across multiple servers, reducing the load on any single node and enabling parallel query execution.

5.3 Caching

The use of in-memory caches (e.g., Redis, Memcached) for frequently accessed data can greatly reduce the number of direct reads to the database, lowering query response time. Caching tools like Redis use RAM to store the data; thus, accessing the data is much faster than accessing the data directly from the database.

As memory access latency is in the nanoseconds, whereas disk access (even SSD) is in the microseconds to milliseconds, there is a tremendous advantage to utilizing in-memory.

Common caching strategies include:

Key-Value Pairing: Data are stored in key-value format, enabling O (1) average-time access.

Eviction Policies: Caches have a limited size, so they use policies such as least recently used (LRU) or least frequently used (LFU) to remove stale data.

Cache Invalidation: Ensuring consistency between the cache and database by invalidating or updating cached entries upon data changes.

There are two common caching strategies:

- Write-through cache: Writes data to both the cache and the database simultaneously.
- Write-back cache: Writes data to the cache and updates the database at a later point.

5.4 Connection Pooling

Opening a new database connection requires a lot of resources, including authentication, network configuration, and session launch. In a high-traffic application, requesting a new connection from the database with each request might result in unnecessary delay and a reduction in resources. Connection pooling can alleviate these issues by reusing connections, resulting in:

Reduced Latency: By reducing the need for repeated connection setups, connection pooling reduces response times.

Better Resource Utilization: By limiting the number of active connections, the database server is protected from excessive load, ensuring consistent performance.

Enhanced Scalability: Applications can accommodate a greater number of simultaneous users without suffering from performance decline due to efficient management of connections.

In microservice architectures, the importance of connection pooling is heightened since each service may independently interact with the database. This approach ensures that the system can scale horizontally without overwhelming the database with excessive connection requests.

6. Importance of vector databases in AI Systems

Vector databases are playing a bigger role in modern AI systems, particularly in natural

language processing (NLP), recommendation engines, and computer vision. Vector databases are purpose-built to store, index, and search high-dimensional vector representations (i.e., embeddings) created from machine learning models and to facilitate similarity search and retrieval in complex sequences of operations in AI.

Standard relational databases were not designed to accomplish the unique needs of vector-based queries, specifically finding the nearest neighbors in high-dimensional spaces. Vector databases provide approximate nearest neighbor (ANN) search algorithms such as Hierarchical Navigable Small World (HNSW) graphs, Inverted File Index (IVF), or Product Quantization (PQ), which can make retrieving a vector representation from a vector store significantly more efficient and scalable in the realm of vector stores. The benefits of vector databases in AI:

Semantic Search and Retrieval: Vector databases provide semantic search capabilities using embeddings that record the contextual meaning of inputs. This represents a large improvement over keyword-based searches for large language model (LLM) applications. Document retrieval, augmenting chatbot memory, and knowledge bases are among the applications where improvements in relevance are made with semantic search. (Matthijs, Johnson, & Hervé, 2021)

Scalability: These systems are aimed at serving billions of vector entries while adhering to strict latencies. They allow for real-time applications such as content recommendations, anomaly detection, and personalized search in large ecosystems. (Monigatti, 2023)

Integration with AI Pipelines: Vector databases can be embedded directly into AI workflows that allow models to perform retrieval-augmented generation (RAG), where knowledge is looked up in real-time to produce a more robust output from the model. This allows the model to respond in a more informed, relevant, and current way.

High Performance in High-Dimensional Spaces: Dedicated indexing methods allow vector databases to perform fast approximate nearest neighbor searches that would typically take a significant amount of time in high-dimensional space. This functionality is very useful for use cases like face recognition, image classification, multimodal AI, etc.

Fault Tolerance and Availability: Many production-quality vector databases offer features such as replication, distributed architecture, and horizontal scaling to create data persistence and uptime for enterprise AI environments. (Schwaber-Cohen, 2023)

However, vector databases can have problems of their own, including additional infrastructure costs, complexity of index maintenance, and susceptibility to the ”curse of dimensionality,” which may impact performance if not controlled. But their benefits far outweigh the downsides in most AI environments where large-scale semantic understanding and retrieval is needed.

7. Model deployment and inference

To deploy machine learning (ML) models for real-time inference requires more than training a performant model; it requires a deployment strategy that enables lower latency, scalable, and resilient serving infrastructure. Real-time applications— such as fraud detection systems, recommendation engines, autonomous systems, and conversational agents—often require sub-second latencies, which require highly optimized serving pipelines.

There are many frameworks that exist to help you get your model into production; the industry standards for model deployment include TensorFlow Serving, TorchServe, and ONNX Runtime. These frameworks allow for flexible model versioning, support multiple ML architectures, and provide optimized inference runtime. For example,

TensorFlow Serving provides gRPC and RESTful APIs that make integrating into a production application seamless; TorchServe provides multi-model serving capabilities and custom inference handlers; while ONNX Runtime supports portability for models when exported as an Open Neural Network Exchange (ONNX) model format and optimizations for the execution paths whether deployed in CPU or GPU.

To support performant delivery for model updates or version rollouts, these frameworks offer a hot-swap capability and model canarying that allow new versions to be evaluated on a subset of the traffic before completely deploying them. This substantially reduces the risk of regression in a production environment.

The emergence of container technologies, such as Docker, allows for the packaging of ML models with their dependencies in immutable, portable containers. As a result, we can be assured that it runs consistently regardless of the environment (development, testing, and production). Similarly, orchestrators such as Kubernetes further enable automated deployment, scaling, and management of containerized ML services across cluster nodes. These orchestrators utilize APIs and services that follow a declarative model to automate container deployment, scaling, auto-scaling, replication, and scheduling. For example, Kubernetes supports a declarative model via the YAML files for a Kubernetes service. Kubernetes has the capacity for horizontal pod autoscaling, as well as resource allocation policies on the cluster to support service optimally for various traffic levels.

Deployment systems are often integrated with monitoring (e.g., Prometheus, Grafana) and logging systems (e.g., ELK stack) for observability and maintainability. These systems can allow teams to monitor for inference latency, throughput, and error rate. Deployment systems can, and increasingly do, also include infrastructure for monitoring model performance (e.g., drift, attribution for feature importance, accuracy of the model in the real world), indicating the promise of longer-term success and value after the model is deployed.

Lately, there are more approaches that abstract away managing the infrastructure through serverless ML deployments like AWS SageMaker Inference, Google Cloud Vertex AI, or Azure ML Endpoints. These serverless options allow developers to deploy models while requiring no operational overhead, giving the model auto-scaling and paying for resource consumption as a service. Such serverless options can be advantageous for model deployments with workloads that are very dynamic or bursty.

For edge computing, a lot of the time the models will need to operate with little compute and network capacity available, so techniques for model optimization, such as model quantization, pruning, and knowledge distillation, are beneficial in preparing models before deployment. Popular frameworks that cater to these edge deployments include TensorFlow Lite, ONNX Runtime Mobile, and NVIDIA TensorRT.

Deploying ML models to an online or real-time application is a complex task involving a combination of systems engineering, distributed systems, and software infrastructure. To successfully run real-time AI-driven systems, you need to combine different elements such as serving frameworks, container orchestration, observability tooling, and model optimization strategies for the performance, reproducibility, and scaling features needed.

8. Monitoring, diagnostics and optimization

Consistent monitoring and performance tuning is a fundamental aspect to sustain a scalable backend system that is capable of supporting more users and, at the same time, being highly available. A scalable back-end will support concurrent requests and should

automatically scale to address variability in workloads. Observability refers to a wide range of methods that facilitate ongoing monitoring, structured logging, and anomaly detection.

Monitoring tools such as Prometheus, Grafana, and AWS CloudWatch play a pivotal role in capturing fine-grained system metrics, including: (Amazon Web Services, n.d.)

CPU utilization: Denotes the utilization of compute resources. Continuous use of greater than 80 percent may be triggering scaling or tuning action.

Memory usage (MB/GB): Helps identify memory leaks or inefficient caching strategies.

Request latency (ms): The median latencies and P95/99 latencies are relevant measures. As an example of usage for dramatic workloads, aim for the P95 latency to be less than 200 ms in a high-throughput web service.

Error rates (HTTP 5xx responses / total requests): Rates that are too high may indicate a backend instability or misconfiguration.

Throughput (requests/sec or transactions/sec): Measures how many users/sessions a system can work with at the same time. Rates are also a great indicator for scalability.

This data is illustrated on dashboards (for example, Grafana) that allow you to change the infrastructure before it becomes an issue and see when things started to ”go awry” through time-series view and alerting rules. (Grafana, n.d.)

To validate scalability under load, stress testing and performance benchmarking are routinely conducted using tools like Apache JMeter, Locust, or k6. These tests simulate peak traffic conditions and help establish Service Level Indicators (SLIs) and Service Level Objectives (SLOs). For example:

Target SLO: 99.9 percent of requests should be completed in less than 300 ms.

Baseline Throughput: 10,000 RPS (requests per second) at less than 70 percent CPU utilization.

Additionally, vertical and horizontal autoscaling policies are driven by real-time metrics. For example, Kubernetes’ Horizontal Pod Autoscaler (HPA) can scale pods based on CPU usage (for instance, scale up if the CPU gets above 60 percent).

Error tracking solutions like Sentry enable deep debugging with exceptions that capture stack traces, user sessions, and environmental context surrounding errors. This can help speed up mean time to resolution (MTTR) and regression analysis. In the observability stack, the ELK Stack (Elasticsearch, Logstash, and Kibana) provides scalable log aggregation, searching, and visualization. Logs are commonly formatted in JSON and indexed so they may be retrieved quickly and correlated with metrics.

Distributed tracing (for example, through Jaeger or Open-Telemetry) is also ramping up in microservices-based discussion. With distributed tracing, developers can trace requests through different services to see where latency spikes occur and the dependencies at the service level.

9. Future directions: edge computing

Edge computing is a decentralized computing model, wherein data is computed at the edge, or a smaller location from the data, versus only using central computing in the cloud. This new architecture improves upon cloud computing approaches because it reduces latency, allocation of bandwidth, and responsiveness with intelligent systems, especially when timely data processing and action is necessary.

Typical AI applications, especially those that rely on realtime inferencing, such as autonomous vehicles, industrial robotics, smart (corner) surveillance, and augmented reality applications, require a low decision

latency, a high throughput from unstructured data, and situational contextual awareness. The always-existing latencies of the network and bandwidth issues are inherent challenges for cloud computing and limit establishing decision latency responses.

Edge computing expands AI capabilities by enabling:

Ultra-low latency processing: Critical for time-sensitive tasks like obstacle detection in autonomous vehicles or predictive maintenance in industrial systems.

Bandwidth optimization: Edge nodes preprocess data and transmit only relevant summaries or insights to central systems, reducing the overall network load.

Improved data privacy and security: Since sensitive data (e.g., facial recognition or health metrics) can be processed locally, the risk of exposure is reduced, aligning with privacy regulations like GDPR.

Scalability and fault tolerance: Distributed edge nodes ensure continuous availability even when centralized systems are unreachable or under high load.

References:

Amazon Web Services. (n.d.). Retrieved from https://docs.aws.amazon.com/cloudwatch/

Breck, E., Cai, S., Nielsen, E., Salib, M., & D., S. (2018). The ML test score: A rubric for ML production readiness and technical debt reduction. *2017 IEEE International Conference on Big Data (Big Data)*.

Grafana. (n.d.). Retrieved from https://grafana.com/docs

Monigatti, L. (2023, October 9). From prototype to production: Vector databases in generative AI applications.

Matthijs, D., Johnson, J., & Hervé, J. (2021). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*.

Rausch, T., & Dustdar, S. (2019). Edge Intelligence: The Convergence of Humans, Things, and AI. *2019 IEEE International Conference on Cloud Engineering (IC2E)*.

Schwaber-Cohen, R. (2023, May 3). Retrieved from https://www.pinecone.io/learn/vector-database/

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., . . . Zumar, C. (2018). Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*.

Tural Muzafarov
University of Kragujevac
Kragujevac,
Serbia
turalmuzefferli4@gmail.com
ORCID 0009-0002-7025-4180