Comparing Two ML Inference Systems

Jacopo Maragna

November 2023

The following report aims to compare two different ML inference systems both qualitatively and quantitatively.
The experiments were conducted following the **Model-as-a-Service** pattern. Particularly, two different deployment strategies have been tested:

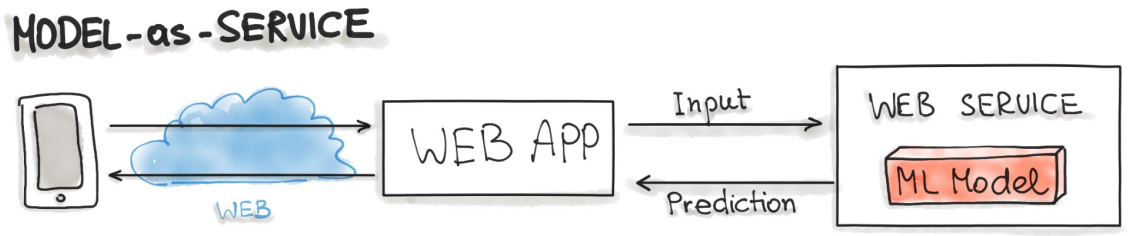
- **Deploying ML Models as Docker Containers**
- **Deploying ML Models as Serverless Functions**

Model Serving

The process of deploying the ML model in a production environment.

Model-as-Service

Model-as-a-Service is a prevalent approach for encapsulating a machine learning model as a standalone service. This involves packaging both the ML model and its interpreter within a dedicated web service, which can be accessed by applications through a REST API or utilized as a gRPC service.



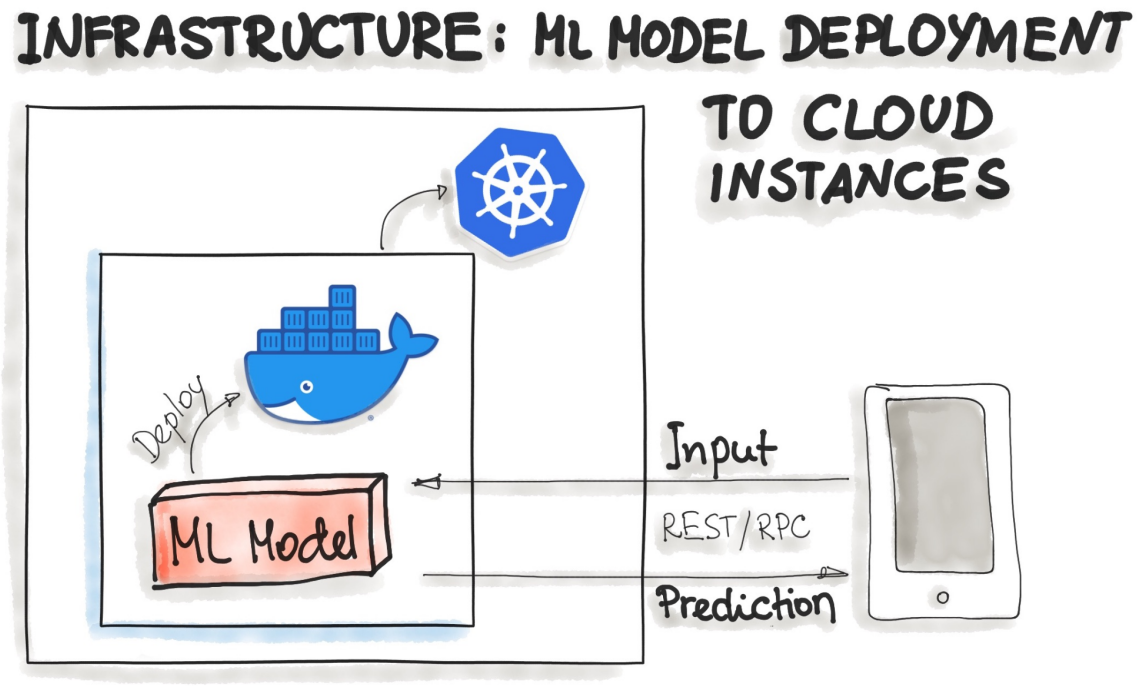
Deployment Strategies

In the following, we discuss common ways for wrapping trained models as deployable services.

Deploying ML Models as Docker Containers

Currently, there is no universally accepted open solution for deploying machine learning models. To address this, ML model inference is typically treated as stateless, lightweight, and idempotent, making containerization the prevailing method for deployment. Docker is the standard containerization technology, suitable for on-premise, cloud, or hybrid deployments. One common approach involves packaging the entire ML technology stack and model prediction code within a Docker container, with orchestration carried out by platforms like Kubernetes or AWS Fargate. This enables the ML model's

functionality, such as prediction, to be accessed via a REST API, often implemented as a Flask application.

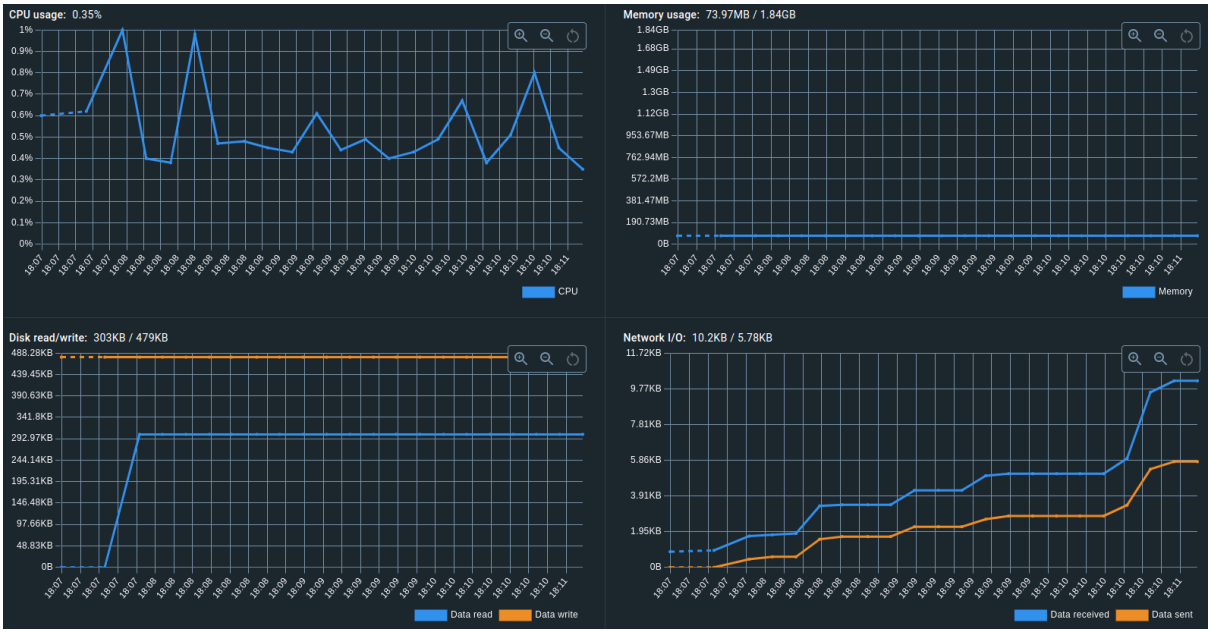


My Implementation

The ML service has been developed using the FastAPI framework and Docker containerization. The model, serialized in the pickle format, is loaded and exposed to provide seamless predictions via a dedicated API endpoint. This integration of FastAPI and Docker ensures a robust and efficient means of delivering machine learning capabilities.

Performance

Experiments were conducted by invoking the endpoint through Postman. The container was running on Docker Desktop with a memory allocation of 1.68GB. The model used for these experiments is an SVC classifier from scikit-learn trained to solve “Banknote Authentication” use cases.
The dashboard displays key performance metrics, including CPU usage, memory utilization, disk read/write activities, and network data. Notably, we observed CPU usage spikes in proximity to inference requests, with a concurrent increase in network data usage as we progressed through the experiments.
It has been observed to have an average execution time of 26 ms.



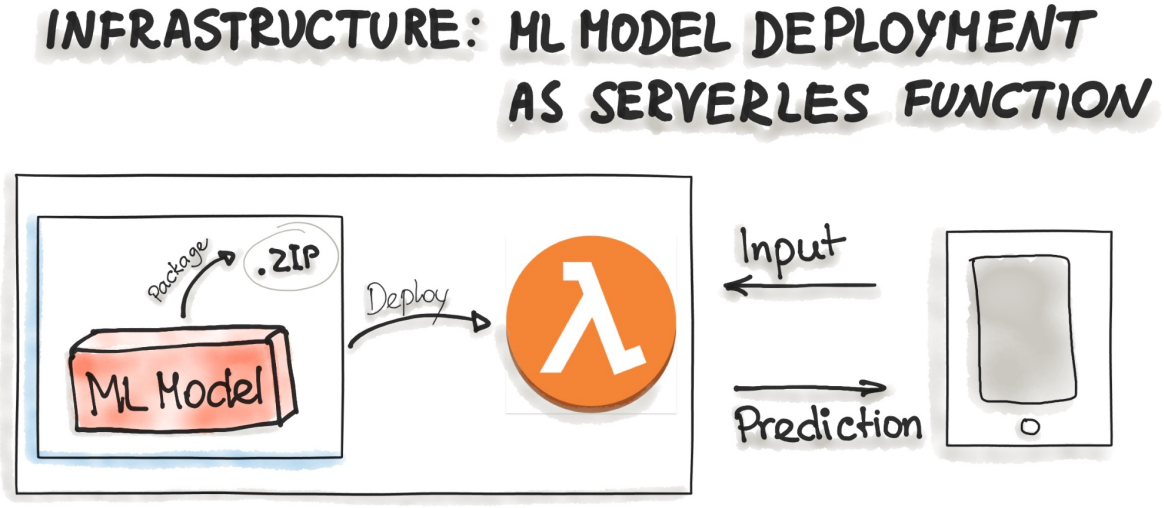
GitHub Repository

https://github.com/jackma-00/ml-inference/tree/main/container

Deploying ML Models as Serverless Functions

Numerous cloud vendors offer machine-learning platforms for model deployment, including Amazon AWS Sagemaker, Google Cloud AI Platform, Azure Machine Learning Studio, and IBM Watson Machine Learning. Additionally, commercial cloud services like AWS Lambda and Google App Engine provide containerization for ML models.

To deploy an ML model as a serverless function, the application code and dependencies are bundled into .zip files, featuring a single entry point function. Major cloud providers like Azure Functions, AWS Lambda, and Google Cloud Functions can manage this function. However, it's essential to be mindful of potential limitations, such as the size of the deployed artifacts.



My Implementation

The ML service has been crafted using an AWS Lambda function in conjunction with Docker containerization. The heart of this architecture is a Lambda handler function, which is invoked with each Lambda execution. To optimize this setup, the entire software stack has been containerized and stored on Amazon Elastic Container Registry (ECR). This strategy allows for a smooth deployment of the Docker image into Lambda, ensuring it is fully equipped to serve predictions.

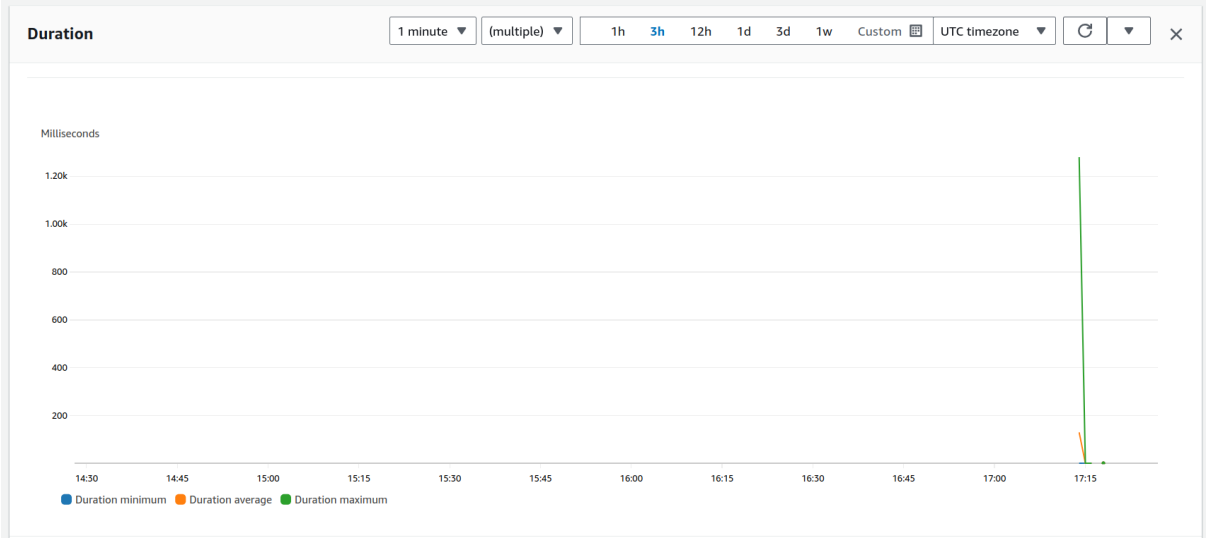
The decision to containerize the image rather than uploading a traditional zip package directly to Lambda was deliberate. This approach sidesteps any limitations on the size of the deployed artifact.

Performance

The experiments were carried out directly on the AWS console to test the Lambda function. The model used for these experiments is an SVC classifier from scikit-learn trained to solve “Banknote Authentication” use cases. The memory allocated for the Lambda Function was 128 MB.

Duration	Billed duration
1.61 ms	2 ms
Resources configured	**Max memory used**
128 MB	127 MB

It has been noted that the Lambda function exhibits an average execution time of 1.6 milliseconds, with the maximum duration observed at 16 milliseconds, typically occurring during the cold start of the Lambda function.



GitHub Repository

https://github.com/jackma-00/ml-inference/tree/main/serverless

Comparison

These two approaches have their own strengths and weaknesses, and the choice between them depends on your specific use case and requirements.

Qualitative Comparison

Docker Containers

- **Isolation**: Docker containers offer strong isolation, making them suitable for running ML models that require specific libraries and dependencies. You can create custom environments tailored to your model's requirements.
- **Portability**: Containers can be easily moved and run in various environments, both on-premises and in the cloud, which makes it convenient for hybrid setups.
- **Scalability**: Docker containers can be manually or auto-scaled to handle varying workloads.
- **Control**: You have more control over the underlying infrastructure, which can be beneficial for performance optimization and resource utilization.

Serverless Functions

- **Ease of Use**: Serverless functions abstract away infrastructure management, making it easier to deploy and scale ML models without worrying about the underlying infrastructure.
- **Cost-Efficiency**: You typically pay only for the compute resources used during execution, which can be cost-effective for sporadic or bursty workloads.
- **Quick Deployment**: Serverless functions can be deployed quickly and are well-suited for event-driven, low-latency applications.
- **Limited Customization**: You may have limitations in terms of customizing the runtime environment and resource allocation in serverless platforms.

Quantitative Comparison

Scalability

- **Docker Containers**: Scalability is manual or requires container orchestration tools like Kubernetes. You can scale up/down based on your needs.
- **Serverless Functions**: Serverless platforms automatically scale based on the number of incoming requests, providing near-instantaneous scalability.

Latency

- **Docker Containers**: Latency can be consistent and low, but it depends on how you configure the container and the infrastructure.
- **Serverless Functions**: Serverless functions often have slightly higher cold start latency, as there may be a delay in initializing the runtime for a new request. However, in our experiments, we observed that the latency of the Lambda function tends to be lower than that of the container. This is attributed to the RPC (Remote Procedure Call) nature of Lambda, which inherently offers faster response times compared to a traditional API setup.

Cost

- **Docker Containers**: Costs can be more predictable but depend on the resources allocated to the container, which may be underutilized during idle times.

- **Serverless Functions**: Costs are pay-as-you-go, which can be cost-effective for low to moderate workloads but may become less economical for continuous high workloads.

Resource Management

- **Docker Containers**: You have full control over resource allocation and can optimize performance based on your specific requirements.
- **Serverless Functions**: Resource allocation is abstracted, and you have limited control over fine-tuning the runtime environment.

Further improvements

These two analyzed approaches are not only straightforward but also highly effective. In their simplicity, they provide valuable insights into the process of serving an ML model, highlighting pertinent issues and performance considerations. Moving forward, the next steps could involve exploring two open-source solutions, Seldon Core and Kserve, which promise to offer more comprehensive capabilities and advanced features for model deployment and serving.

References

https://ml-ops.org/content/three-levels-of-ml-software#code-deployment-pipelines