Serverless Scaling
with Deep Reinforcement Learning

Prof. Danilo Ardagna

POLITECNICO
MILANO 1863

About Us

Federico Mansutti
10683462

Biagio Fabio Schiliro
10933149

Giacomo Ruvolo
10978833

What do we want?

Developing an intelligent autoscaling solution for serverless functions using Deep Reinforcement Learning to overcome the limitations of static threshold-based methods in dynamic workload environments

- **Deploying the system architecture** on a Kubernetes-based infrastructure to support scalability and orchestration
- **Building a Flask-based application** to run on top of the kubernetes infrastructure
- **Integrating a Deep Reinforcement Learning Agent** to manage autoscaling decisions
- **Using JMeter and Prometheus** to evaluate the behaviour of the architecture

System Overview

From a high-level perspective, the system is composed of the following core components:

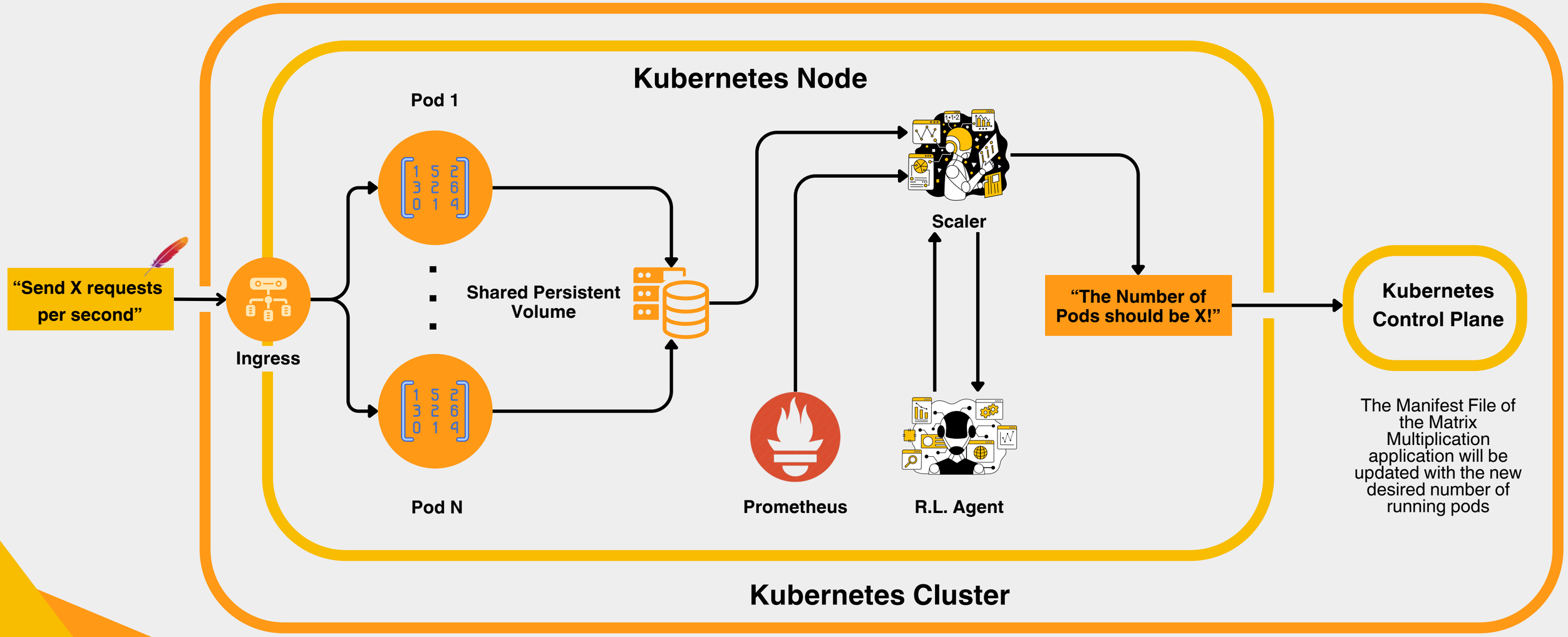
- The **Matrix Multiplication Application,** which serves as the computational core of the system.
  It is deployed across multiple pods and the Reinforcement Learning Agent will dynamically determine the appropriate number of running pods based on the real-time workload demands

- The **Scaler**, which is responsible for analyzing the logs produced by the Matrix Multiplication Application and the metrics collected via Prometheus.
  The scaler itself does not represent the RL agent but, instead, it gathers the necessary data, queries the agent for a decision and, then, updates the Kubernetes manifest to scale the system accordingly

- The **JMeter Load Tester** that is used to generate synthetic workloads to evaluate the system's performance under varying traffic conditions

All the components — the Matrix Multiplication Application, Scaler, RL Agent and JMeter Load Tester — are fully containerized using Docker

System Overview



The Matrix Multiplication

This application performs the multiplication of two user-provided matrices and returns two key metrics, the Service Time and the Response Time.
In particular:

- When a user sends a request, it first passes through the ingress.
  At the moment the ingress forwards the request to a pod, it attaches a timestamp to the request so as to ensure that the response time measurements are not influenced by external network delays.
  Specifically, we measure the effective time from when the request enters the system to when it is returned by the application

- Once the request is received, the application performs the matrix multiplication, calculates the service time and the response time and stores both the metrics in a JSON file within the persistent volume.
  The file is named using the format:

  <Pod_Name>_<Timestamp>.json

The Scaler

The scaler is implemented in Kubernetes as a CronJob, meaning it runs at regular intervals (every minute in our implementation).
During each execution, it analyzes the available metrics and adjusts the number of running pods if necessary (i.e. if the agent returned a changing action).

Once the scaler accesses the log files produced by the Matrix Multiplication Application, it processes them, moves the files into the **AnalyzedPods** folder and generates a summary file called **podStatus.json**.
Each record in this file will follow a structured format, as illustrated in the image on the right.


```
"check_4": {
    "timestamp": "03.05.2025_14:28:07",
    "nInstances": 2,
    "pressure": 5.125999649153228,
    "avgResponseTime": 5.125999649153228,
    "Threshold": 1.0,
    "queueLengthDominant": 54757.388784426104,
    "utilization": 0.026646669705849987,
    "CPUUsage (Prometheus)": 0.07750482674329473,
    "workload": 9.35,
    "action": "The number of pod will be set to 4!",
    "active": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-mxl9p",
        "matrix-multiply-d847c46c7-kxssf",
        "matrix-multiply-d847c46c7-975pc"
    ],
    "startedPreviousCheck": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-975pc",
        "matrix-multiply-d847c46c7-kxssf"
    ],
    "shutdownPreviousCheck": []
```

The Scaler

Specifically, each element of this record will represent the following:

- **timestamp**: the time at which Check X was performed
- **nInstances**: the number of pods running at the time of the check
- **pressure**, **queueLengthDominant**, **utilization**, **CPU Usage**: performance metrics used for decision-making (refer to the following RL Agent slides for details)
- **Threshold**: the time constraint for the average response time, expressed in seconds
- **workload**: the number of requests per second received by the system


```
"check_4": {
    "timestamp": "03.05.2025_14:28:07",
    "nInstances": 2,
    "pressure": 5.125999649153228,
    "avgResponseTime": 5.125999649153228,
    "Threshold": 1.0,
    "queueLengthDominant": 54757.388784426104,
    "utilization": 0.026646669705849987,
    "CPUUsage (Prometheus)": 0.07750482674329473,
    "workload": 9.35,
    "action": "The number of pod will be set to 4!",
    "active": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-mxl9p",
        "matrix-multiply-d847c46c7-kxssf",
        "matrix-multiply-d847c46c7-975pc"
    ],
    "startedPreviousCheck": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-975pc",
        "matrix-multiply-d847c46c7-kxssf"
    ],
    "shutdownPreviousCheck": []
```

The Scaler

Specifically, each element of this record will represent the following:

- **action**: the number of pods recommended by the RL Agent
- **active**: the list of pods actively running at the time of the check
- **startedPreviousCheck**: the pods that were started after the previous check, based on the RL Agent's recommendation
- **shutdownPreviousCheck**: the pods that were stopped after the previous check, based on the RL Agent's recommendation


```
"check_4": {
    "timestamp": "03.05.2025_14:28:07",
    "nInstances": 2,
    "pressure": 5.125999649153228,
    "avgResponseTime": 5.125999649153228,
    "Threshold": 1.0,
    "queueLengthDominant": 54757.388784426104,
    "utilization": 0.026646669705849987,
    "CPUUsage (Prometheus)": 0.07750482674329473,
    "workload": 9.35,
    "action": "The number of pod will be set to 4!",
    "active": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-mxl9p",
        "matrix-multiply-d847c46c7-kxssf",
        "matrix-multiply-d847c46c7-975pc"
    ],
    "startedPreviousCheck": [
        "matrix-multiply-d847c46c7-v9zz6",
        "matrix-multiply-d847c46c7-975pc",
        "matrix-multiply-d847c46c7-kxssf"
    ],
    "shutdownPreviousCheck": []
```

The RL Agent

The Reinforcement Learning Agent will be invoked via an HTTP POST request that must include the following data:

- The **Number of Instances** that are actually running in the system for our application of interest
- The **Pressure**, which denotes how close the system is to violate the response time constraints

$$P = \frac{ObservedResponseTime}{ResponseTimeThreshold}$$

- The **Queue Length Dominant**, which measures the queue buildup on the dominant component (i.e. the one with the highest pressure)

$$QLD = \frac{ResponseTime - DemandTime_d}{DemandTime_d}$$

The RL Agent

The Reinforcement Learning Agent will be invoked via an HTTP POST request that must include the following data:

- The **Utilization** of the system (i.e. the fraction of time in which each instance is actively processing requests)

$$U = \frac{\sum_{i \in I} ArrivalRate_i \times DemandTime_i}{NumberOfInstances}$$

  Note that we use both the theoretical formula (as shown above) and the actual CPU utilization measured by Prometheus.
  Depending on the objectives, you can choose what should be sent to the agent.

- The **Workload**, that represents the number of requests per second received by the system

The RL Agent

Once the agent receives a request and performs inference to determine the appropriate number of pods to run in the system, it returns a single integer value, referred to as **action** (i.e. the pods to run now).
In order to ensure the robustness of the system, the scaling logic handles the response as follows:

- If **action = -1**, this indicates that an error occurred (e.g. connection failure, malformed response or the agent being unavailable).
  In this case, the number of pods remains unchanged from the previous check

- If **action > 0**, this means the agent successfully returned a valid response and no error occurred.
  The system will then adjust the number of pods according to the new value provided by the agent

The JMeter Load Tester

In order to generate the synthetic workloads, we use the following key components in JMeter:

- The **Concurrency Thread Group**, which defines the number of threads (virtual users) that will send requests.
  This number should be high enough to reproduce the desired workload shape effectively
- The **Exponential Jitter Timer**, which introduces variability in the request inter-arrival times, following an exponential distribution
- The **Throughput Shaping Timer**, which controls the overall shape of the workload over a specified time interval, allowing precise workload modeling
- The **CSV Data Set Config**, which randomizes the matrix input data used in the requests, making the workload behavior more dynamic and closer to the real-world conditions

The JMeter Test Plan, containing all these elements, can be automatically generated following the notebook ***WorkloadAndConfigGenerator.ipynb***

Prometheus / Grafana

While Prometheus is used to collect CPU usage metrics, we expanded its role by integrating a full suite of exporters to monitor the system from multiple perspectives.

Node Exporter

It collects hardware and OS-level metrics such as CPU usage, memory consumption, disk I/O and network statistics from the host machine.
It is essential for system-level monitoring

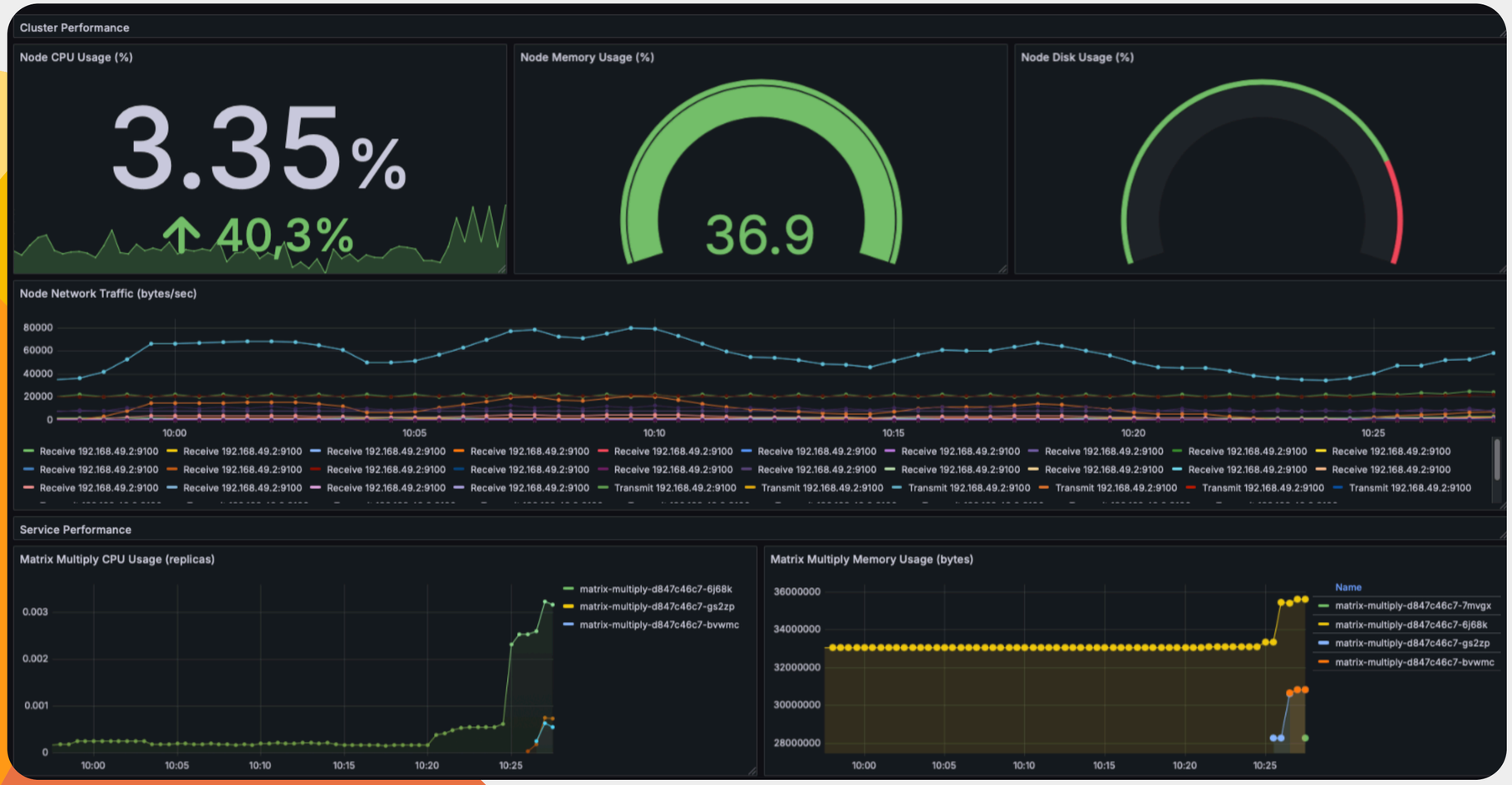
cAdvisor Exporter

It automatically collects real-time metrics on container usage so as to provide detailed insights into the resource usage for each pod

Blackbox Exporter

It performs the endpoint probing via HTTP, HTTPS, TCP, ICMP (ping), etc..., to monitor the service availability and the response times.
It is ideal for tracking the uptime and the reachability of web services

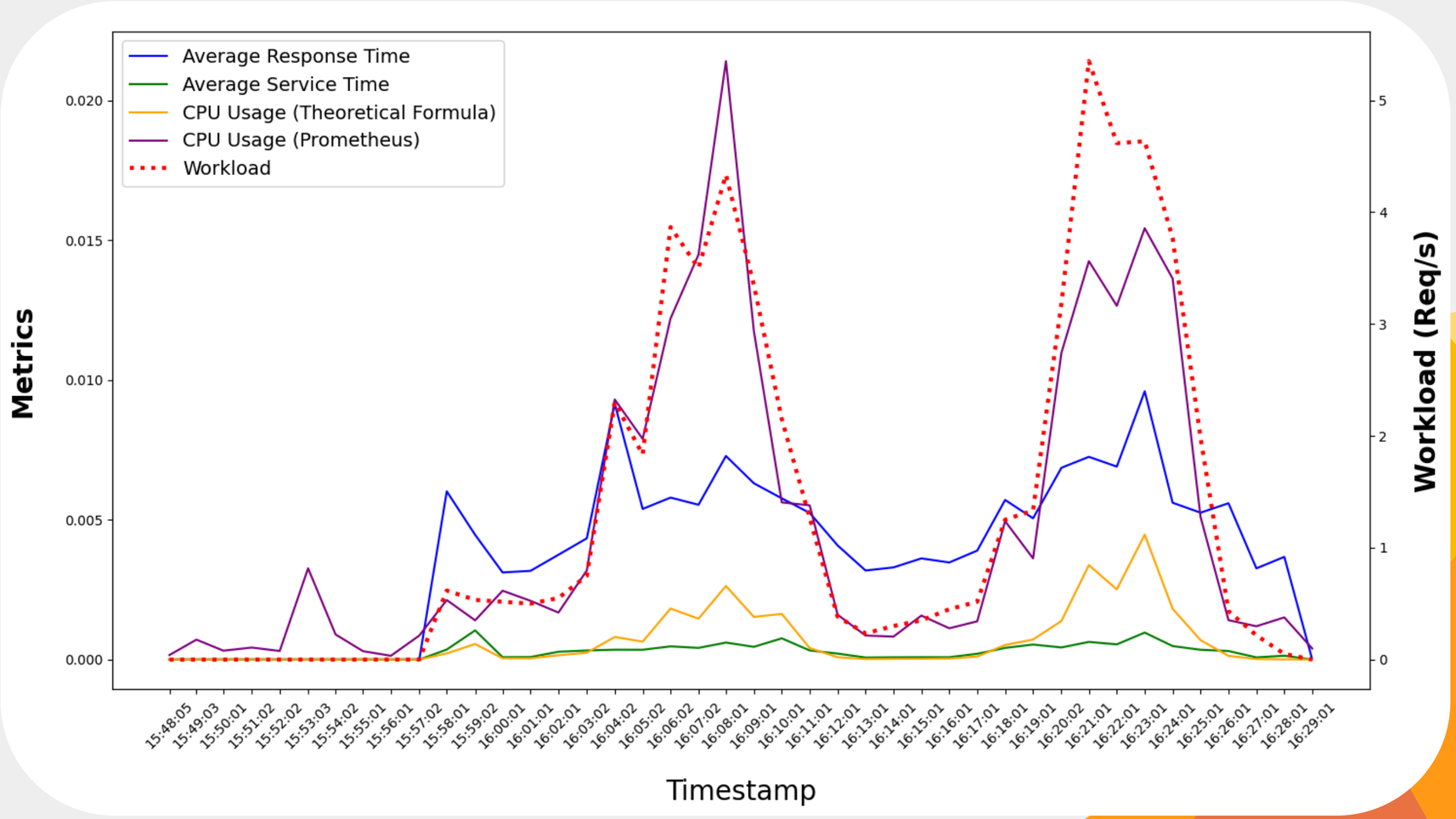
Prometheus / Grafana



All the metrics collected via Prometheus can be visualized through Grafana.
In order to facilitate this, we created a custom dashboard specifically designed to display the key performance indicators of our system, as illustrated in the image

Execution Test (1 Pod)

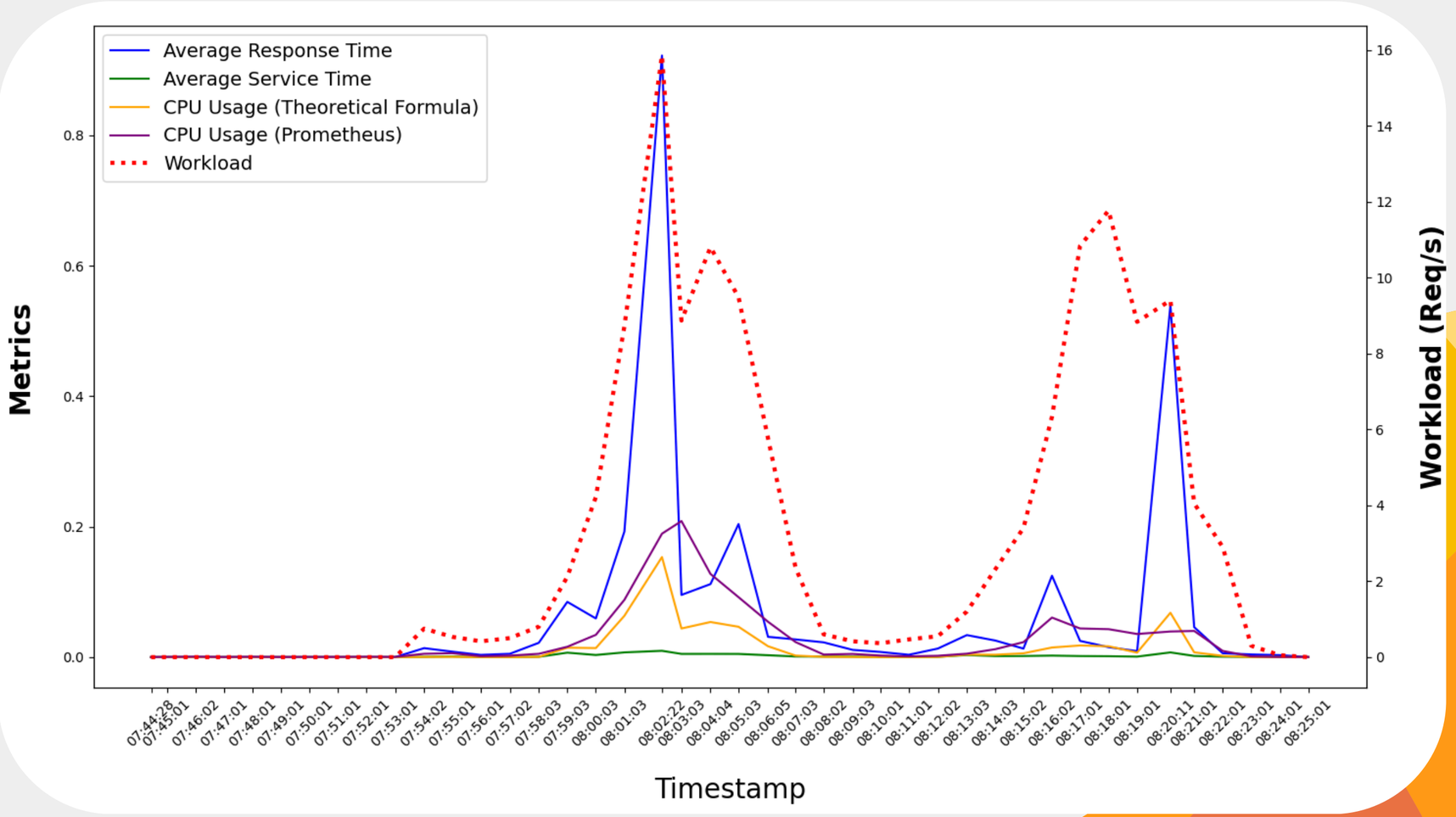
In this test, we simulate a **low-intensity** sinusoidal **workload** by sending **50×50 matrices**.
The sinusoidal pattern is used to mimic real-world traffic fluctuations (e.g. between daytime and nighttime usage)



*This graph can be generated by passing the podStatus.json file to the function defined in the notebook **visualizeResults.ipynb***

Execution Test (1 Pod)

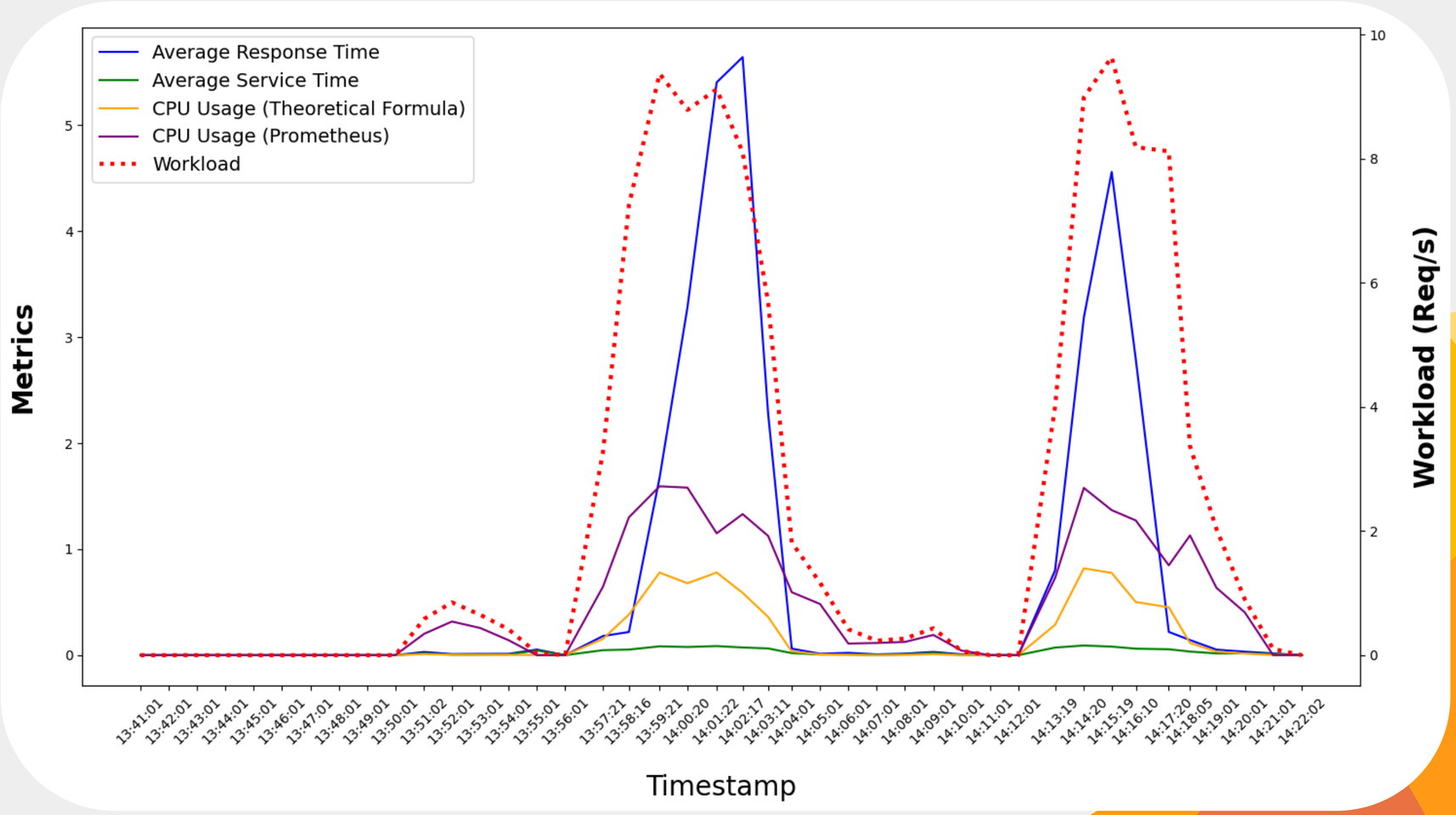
In this test, we simulate a **high-intensity** sinusoidal **workload** by sending **50×50 matrices**.
The sinusoidal pattern is used to mimic real-world traffic fluctuations (e.g. between daytime and nighttime usage)



*This graph can be generated by passing the podStatus.json file to the function defined in the notebook **visualizeResults.ipynb***

Execution Test (1 Pod)

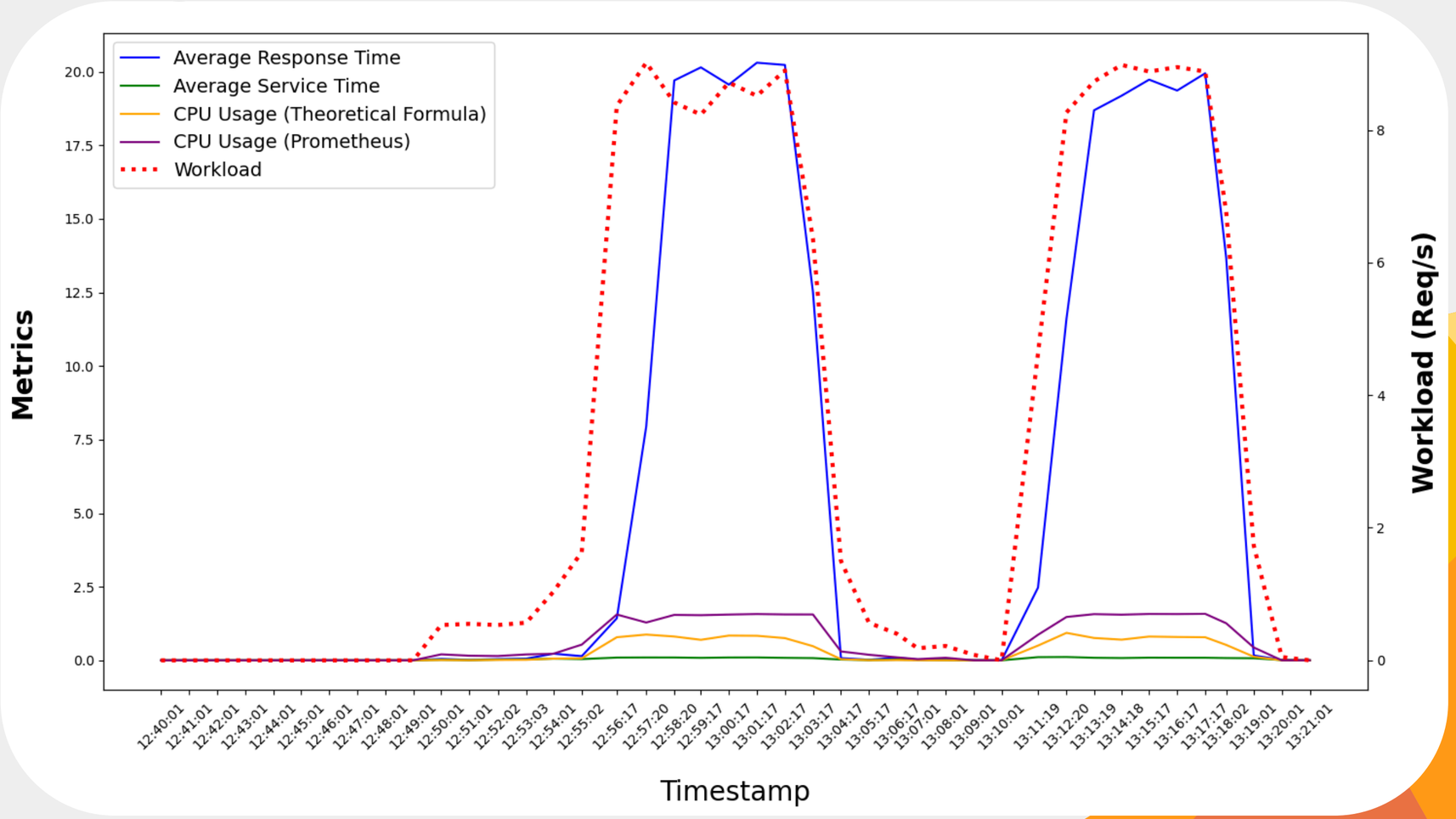
In this test, we simulate a **medium-intensity** sinusoidal **workload** by sending **70×70 matrices**.
The sinusoidal pattern is used to mimic real-world traffic fluctuations (e.g. between daytime and nighttime usage)



*This graph can be generated by passing the podStatus.json file to the function defined in the notebook **visualizeResults.ipynb***

Execution Test (1 Pod)

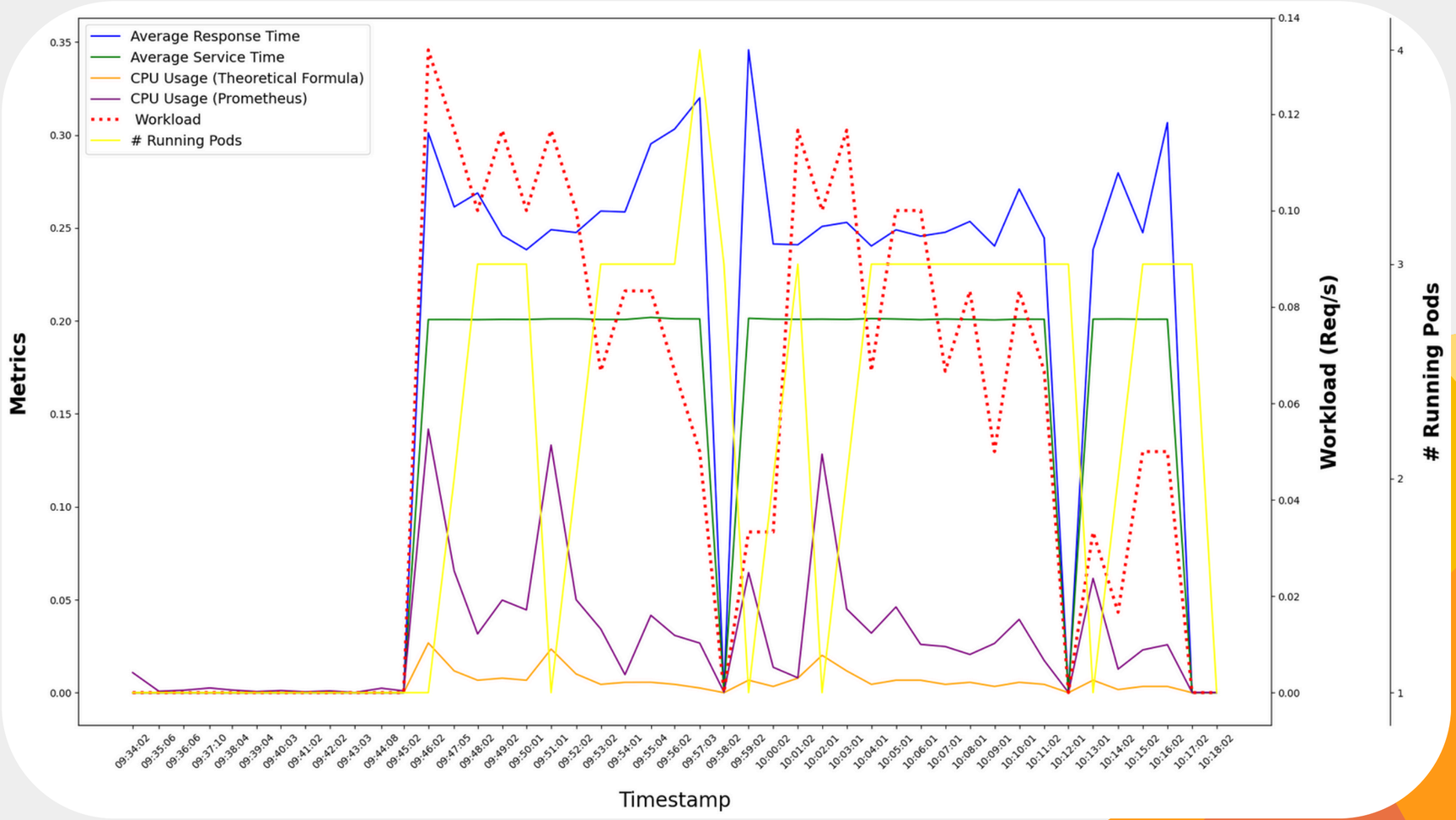
In this test, we simulate a **high-intensity** sinusoidal **workload** by sending **100×100 matrices**.
The sinusoidal pattern is used to mimic real-world traffic fluctuations (e.g. between daytime and nighttime usage)



*This graph can be generated by passing the podStatus.json file to the function defined in the notebook **visualizeResults.ipynb***

Execution Test Suboptimal agent

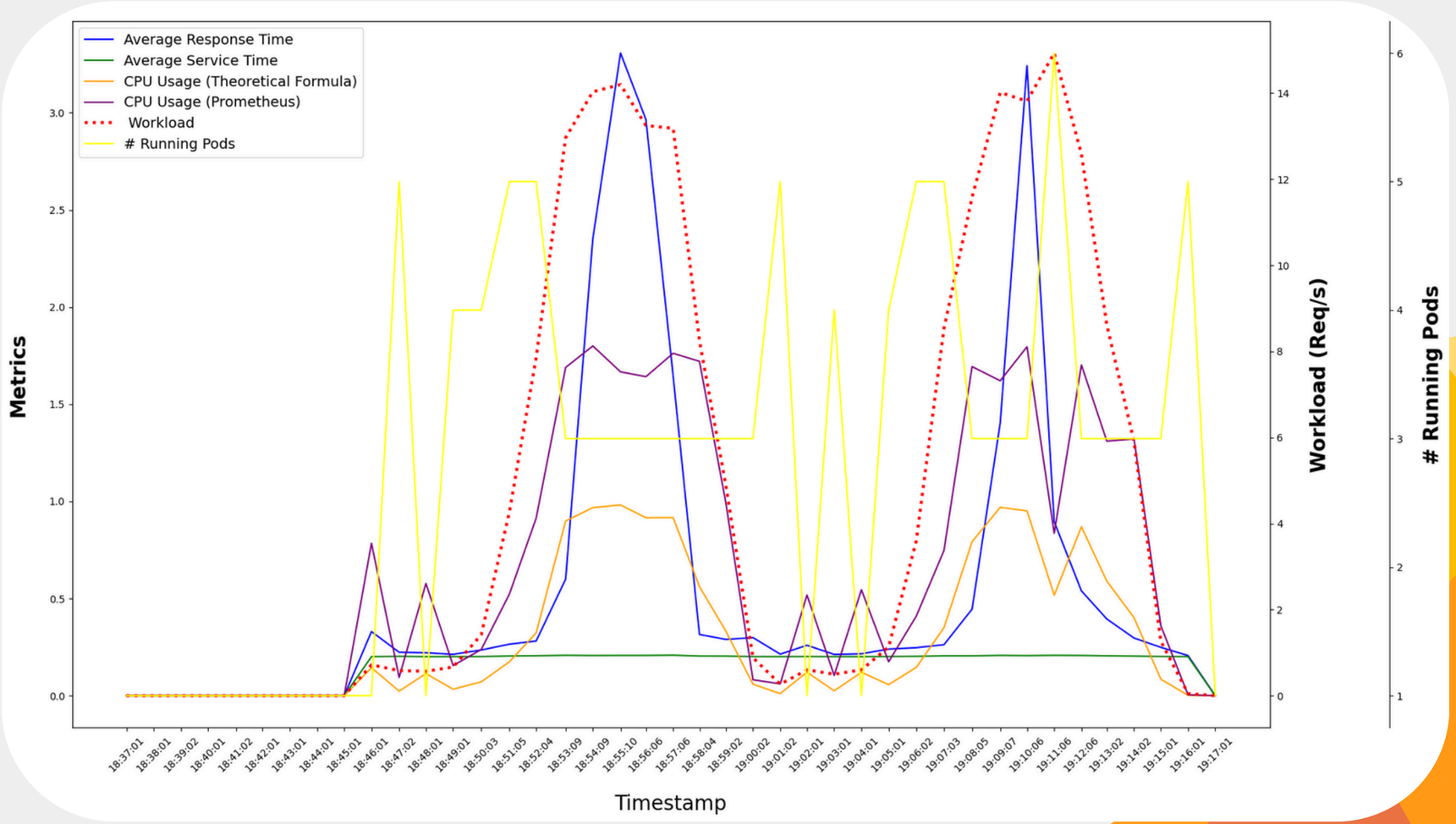
In this test, we simulate a **low-intensity** sinusoidal **workload** by sending **350×350 matrices**.
Unlike the previous tests, where the number of pods was fixed at 1, this time the Reinforcement Learning agent is actively used to dynamically scale the number of pods based on the workload



Note that the agent is using a checkpoint trained on a different workload pattern, so, as result, its performance on this sinusoidal workload is expected to be suboptimal

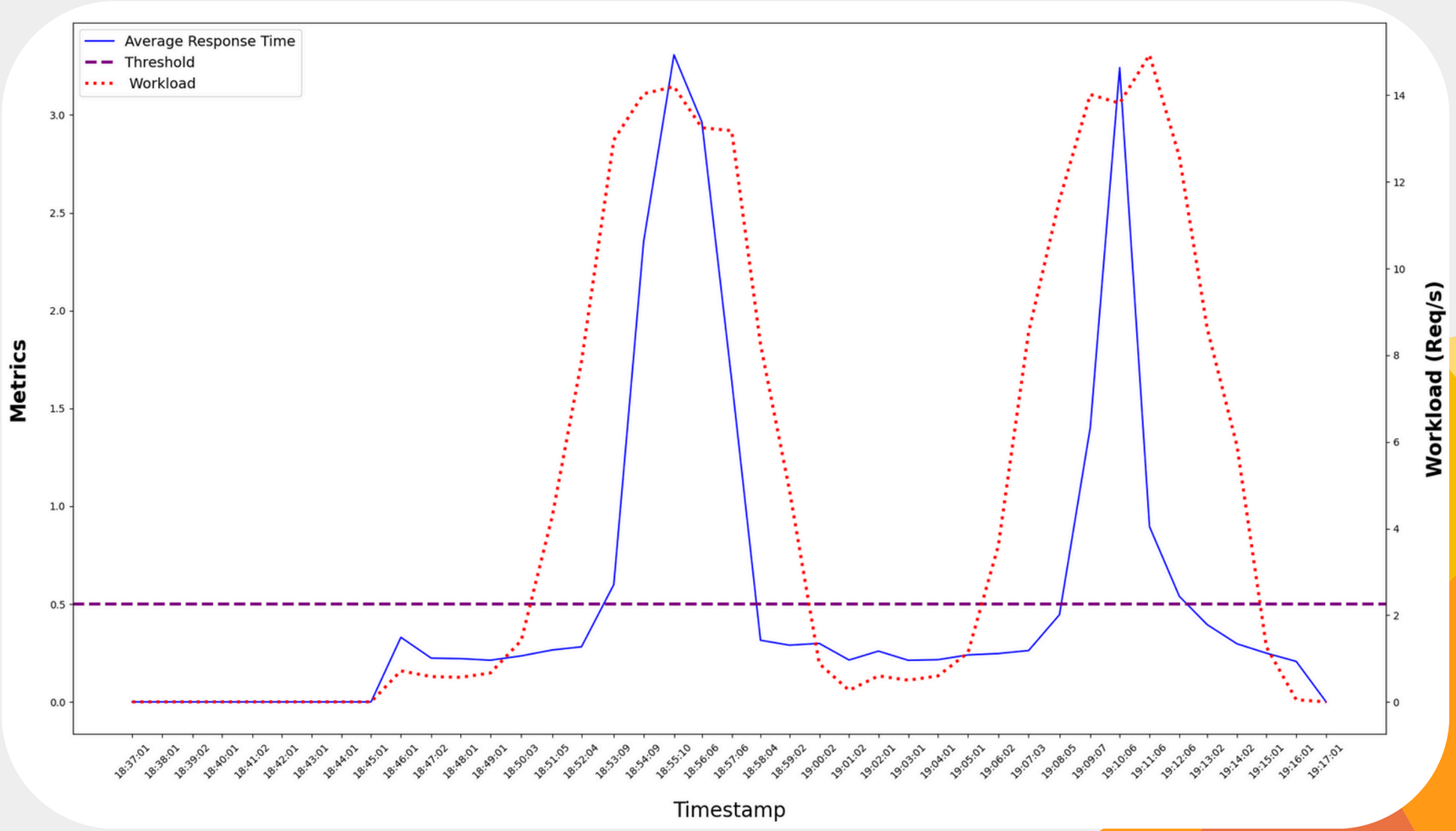
Execution Test Optimal agent

In this test, we simulate a **high-intensity** sinusoidal **workload** by sending **350×350 matrices**.
Unlike the previous tests, where the number of pods was fixed at 1, this time the Reinforcement Learning agent is actively used to dynamically scale the number of pods based on the workload



Execution Test Optimal agent

In this test, we simulate a **high-intensity** sinusoidal **workload** by sending **350×350 matrices**.
The plot shows the response time in relation to the threshold, which represents the upper limit we aim to stay within.
As shown, threshold violations occur only during peak workload intensity and are limited to short durations



ReadMe

For a comprehensive overview of the development and implementation of this project, please refer to the **ReadMe File** available on the following GitHub page:

LINK