D9.4 Final Prototype of Benchmarking Toolkit

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.1
PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

In D9.4, the DAPHNE consortium presents the initial prototype of the benchmarking toolkit for the DAPHNE system.

D9.4 Final Prototype of Benchmarking Toolkit			
Type of document	Report	Version	1.1
Dissemination level	CO/PU		
Lead partner	Hasso Plattner Institute		
Author(s) Reviewer(s)	Ilin Tolovski (HPI), Nils Strassenburg (HPI), Philipp Hildebrandt (HPI), Tilmann Rabl (HPI) Aleš Zamuda (University of Maribor) and Andreas Laber (IFAT)		
Contributors	HPI		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	[Draft before review]	Ilin Tolovski (HPI), Nils Strassenburg (HPI), Philipp Hildebrandt (HPI), Tilmann Rabl (HPI)
V1.1	[Draft after review]	Ilin Tolovski (HPI), Nils Strassenburg (HPI), Philipp Hildebrandt (HPI), Tilmann Rabl (HPI)

Abbreviations

Abbreviation	Term
HPI	Hasso Plattner Institute GmbH
WP	Work Package
T	Task
UC	Use Case
BD	Big Data
HPC	High Performance Computing
ML	Machine Learning
UMLAUT	Universal Machine Learning Analysis Utility
IDA	Integrated Data Analysis
HDF	Hierarchical Data Format
HDFS	Hadoop Distributed File System

Table of contents

Executive Summary ... ***3***

1 Introduction ... ***3***

2 UMLAUT – Final System Prototype ... ***4***

2.1 Benchmarking Aspects ... **4**

2.2 System Under Test ... **5**

2.3 Workloads ... **5**

2.4 Benchmarking Metrics ... **6**

2.4.1 Supervised Metrics ... 6

2.4.2 Valued Metrics ... 7

2.4.3 Extensibility ... 8

3 System Usage ... ***8***

3.1 Benchmarking with UMLAUT ... **8**

3.2 Containerization ... **9**

3.2.1 UMLAUT Only ... 9

3.2.2 UMLAUT and DAPHNE ... 9

3.2.3 UMLAUT, DAPHNE, and GPU ... 10

3.2.4 Custom Containers ... 10

3.3 User Interface ... **11**

4 Use Cases ... ***14***

4.1 Microbenchmarking UMLAUT ... **14**

4.2 Benchmarking the IFAT Semiconductors ... **16**

4.3 Benchmarking KAI Material Degradation ... **18**

4.3.1 KAI Material Degradation VW and RDP ... 18

4.3.2 KAI Material Degradation ML ... 19

5 Conclusion ... ***21***

6 References ... ***22***

Executive Summary

This deliverable describes the implementation of the final prototype of the Universal Machine Learning Analysis UTility (UMLAUT) prototype. We describe the improvements and additional features compared to the initial prototype presented in Deliverable 9.3, as well as a summarized system overview. In this deliverable, we use terminology and definitions from the official UMLAUT documentation, Deliverable 9.3, and the official UMLAUT GitHub repository [1, 2, 6].

1 Introduction

In the scope of the DAPHNE project, the focus of Work Package (WP) 9 has been to create an end-to-end benchmarking toolkit for Integrated Data Analysis (IDA) pipelines. For this purpose, we have surveyed the state-of-the-art in high-performance computing (HPC), big data (BD), and machine learning (ML) benchmarks, summarized in Deliverable 9.1 [3, 5]. Based on the survey findings, we defined the specification and data model for the concept of the toolkit in Deliverable 9.2 [4]. We then developed an initial working prototype of the benchmarking toolkit called Universal Machine Learning Analysis UTility (UMLAUT), summarized in the demonstrator Deliverable 9.3 [6].

The initial prototype of UMLAUT is a standalone library that is imported into a modular system such as DAPHNE and used for benchmarking the IDA pipelines. In Deliverables 8.3 and 8.4 [7, 8], it is used to evaluate and monitor IDA pipelines designed by the use case partners. We received the initial feedback from their user experience and incorporated it into our project plan for the final version.

In this deliverable, we present the final prototype of the internal benchmarking and profiling toolkit UMLAUT. The final version consists of improvements in several aspects, such as the integration with DAPHNE, containerization, additional hardware monitoring capabilities, as well as the overall user experience regarding the installation, ease of use, and visualization features.

With the final prototype of UMLAUT, we improve the precision of the measurements by quantifying the overall resource overhead of the toolkit. We present the improvements regarding the DAPHNE integration, enabling the two systems to run as a single, or as standalone containers. Additionally, UMLAUT now monitors the utilization of the graphics processing unit (GPU), allowing us to support a wider range of use cases that make use of hardware accelerators. IDA pipelines written in the DAPHNE DSL language are monitored together with their additional subprocesses, allowing the users to monitor the pipeline's performance at a finer granularity.

In this deliverable, we present a complete system overview of the final prototype of UMLAUT. This includes a comprehensive list of the features and use cases supported by UMLAUT, as well as a detailed insight into the novel features added since Deliverable 9.3.

This deliverable is structured as follows. In Section 2, we present the UMLAUT system prototype. We provide an overview of the benchmarking aspects, system under test, supported workloads, and metrics. In Section 3, we outline the implementation details. In Section 4, we present an overview of the system usage, describing the initial usage steps, the integration

with the DAPHNE container, and the user interface. In Section 5, we present three use cases of using UMLAUT, to microbenchmark a new system environment, a native Python use case from our project partner KAI, and a DAPHNE DSL use case from our project partner IFAT. Finally, in Section 6, we conclude the deliverable.

2 UMLAUT – Final System Prototype

In this section, we introduce the final system prototype of UMLAUT. In Section 2.1, we present the benchmarking aspects of UMLAUT. We present the system under test in our benchmarking scenarios in Section 2.2. In Section 2.3, we present the supported workloads. Finally, we present the complete list of metrics available in UMLAUT in Section 2.4.

2.1 Benchmarking Aspects

In Deliverables 9.2 and 9.3 [4, 6], we specify the benchmarking aspects covered by the benchmarking toolkit. To provide a holistic performance overview of the system, the benchmarking toolkit monitors the performance of individual stages and methods in the pipeline, and also offers an end-to-end performance summary.
UMLAUT monitors all stages or aspects of an IDA pipeline: data preprocessing and cleaning, simulation, computation or model training, and in the case of ML pipelines, inference. To facilitate this, UMLAUT was designed as a modular framework that monitors the runtime of a pipeline at different levels of granularity. The measurements in UMLAUT can be performed for a single statement method, a method comprising several other methods, a pipeline stage, or the complete runtime of the pipeline.

All measurements are collected as parts of a single pipeline run. To achieve this, we define a data model representing single measurements and the complete pipeline runs, shown in Figure 1. Additionally, we define two sets of metrics, supervised, measuring the resource utilization of individual hardware components, and valued metrics, tracking the values of metrics that receive a value as the result of the pipeline outputs.

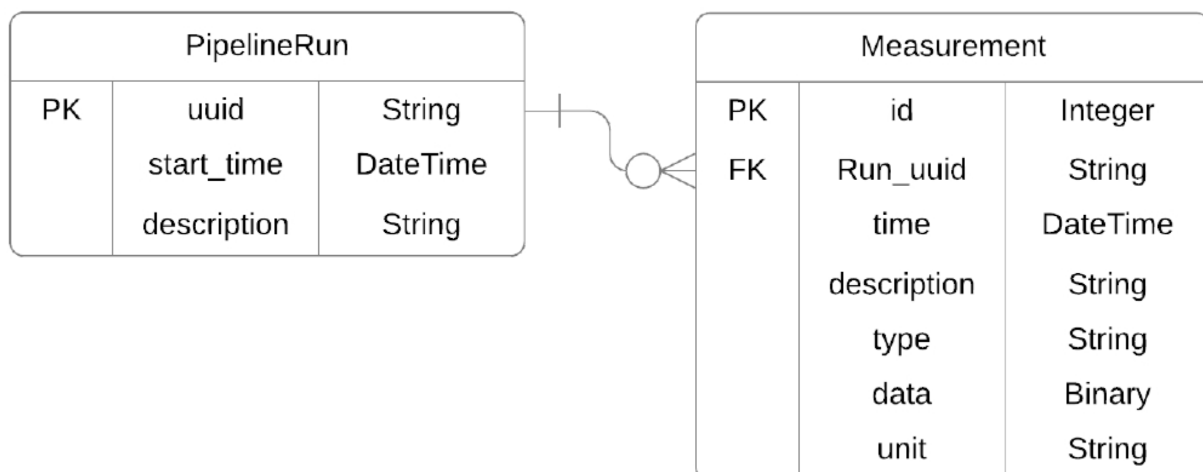

Figure 1: UMLAUT Data Model

2.2 System Under Test

UMLAUT supports monitoring different stages of the pipeline in a modular and independent fashion. The toolkit monitors the system performance at different levels of granularity. These include pipeline monitoring methods comprising a single statement or instruction, computationally intensive pipeline stages, as well as complete pipeline execution. In Figure 2, we show an abstraction of the workflow of the benchmarking toolkit.
The initial UMLAUT prototype presented in Deliverable 9.3 can process pipelines written in Python, DAPHNE DSL, and DaphneLib, monitored via different UMLAUT utilities. When benchmarking DAPHNE DSL pipelines, we tracked the main process executing the pipeline. DAPHNE DSL scripts generate additional processes that need to be monitored to capture the overall system performance. In this version of UMLAUT, we now support the tracking of the subprocesses generated by the DAPHNE DSL pipeline, allowing users to have a complete overview of the system performance.

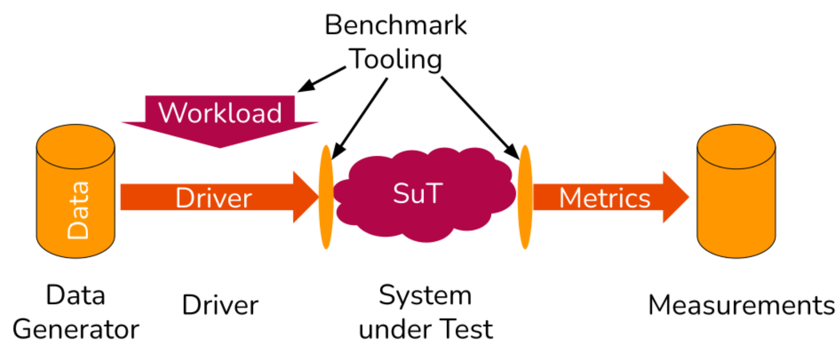

Figure 2: Overview of the Benchmarking Toolkit

2.3 Workloads

In Deliverables 9.2 and 9.3, we have tested UMLAUT with generic IDA pipelines, focusing on covering several stages in a single pipeline. Such workloads included:

- Data cleaning and preparation workloads – part of the data preprocessing stage of an IDA pipeline;
- ML model training – part of the training stage of an IDA pipeline;
- Inference – part of the computation stage of an IDA pipeline.

In this deliverable, we test UMLAUT with two use cases from the project partners in the DAPHNE consortium. In addition to the stages presented in this section, the use cases include domain-specific algorithms that are implemented as external modules and are only invoked within the pipelines. We present the use cases in more detail in Section 4.

2.4 Benchmarking Metrics

In UMLAUT, we collect two types of benchmarking metrics, supervised and valued metrics. In this section, we present their implementation in UMLAUT.

2.4.1 Supervised Metrics

The supervised metrics capture the system performance regarding runtime, throughput, latency, CPU, memory, GPU, and energy consumption. The measurements for all supervised metrics are collected by decorator functions that annotate the methods in the pipeline. In Table 1, we provide a list of all supervised metrics and their implementation.

In order to measure pipelines, e.g., ML pipelines, which often use a GPU to accelerate their execution, we extend the supervised metrics to also include ones concerning the utilization of GPUs. In contrast to metrics like the CPU Metric or Memory Metric, which measure one process, these metrics measure the total utilization of one GPU. This means that for accurate measurements of one pipeline, it should be the only process on the selected GPU.

We extend the set of supervised metrics with four GPU-specific metrics listed in Table 1 and implement them using the NVIDIA Management Library (NVML) [9], a library from Nvidia to access different measurements from their GPUs.. Using these metrics allows an UMLAUT user to track the GPU utilization, GPU memory, GPU time, and GPU power usage.

Table 1: Supervised metrics collected in UMLAUT.

Type	**Description**
TimeMetric()	Measures the time in *seconds* it takes to execute the decorated code using time.perf_counter() [4].
MemoryMetric()	Measures the memory usage of the decorated code in *MB* using psutil memory_info().rss [5].
EnergyMetric()	Measures the energy consumption in *µJoule* of the decorated code using the pyRAPL library [6].
PowerMetric()	Measures the power consumption of the decorated code in *Watt* using the pyRAPL library [6].
LatencyMetric()	Measures the latency of the decorated code in *seconds/entries* using time.perf_counter() [4] to measure time.
ThroughputMetric()	Measures the throughput of the decorated code in *entries/second* using time.perf_counter() [4] to measure time.
CPUMetric()	Measures the CPU usage of the decorated code in *percent* using psutil cpu_percent() [5].
GPUMetric()	Measures the GPU utilization in *percent* using nvmlDeviceGetUtilizationRates().
GPUMemoryMetric()	Measures the GPU memory usage in *MB* using nvmlDeviceGetMemoryInfo().
GPUTimeMetric()	Measures the execution time using CUDA Events. There is usually a small difference between this and time.perf_counter() due to synchronization between GPU and CPU, which matters for quick methods.
GPUPowerMetric()	Measures the power usage of the GPU in *Watt*.

2.4.2 Valued Metrics

The valued metrics capture intrinsic values of the pipeline generated throughout the runtime of a decorated method. Since the values of these metrics are aggregates of measurements executed throughout the method's runtime, we implement them as trackers and insert them into a decorated method. In the final UMLAUT prototype, there are four trackers of valued metrics relevant to the performance of the pipeline: confusion matrix, hyperparameter, time-to-accuracy, and loss tracker. In Table 2, we present all implemented value metrics and their trackers.

Table 2: Valued metrics collected in UMLAUT.

Type	Description
ConfusionMatrixTracker()	Tracks the confusion matrix.
HyperparameterTracker()	Tracks sets of hyperparameters across multiple executions.
TTATracker()	Tracks a list of accuracy values.
LossTracker()	Tracks a list of loss values.

2.4.3 Extensibility

UMLAUT allows extending the set of metrics by implementing decorator methods for supervised metrics or trackers for a valued metric. We described this more in Deliverable 9.3. and in this deliverable, we demonstrate how to extend UMLAUT to support the newly added GPU metrics, as listed in Section 2.4.1.

We followed the steps described in Deliverable 9.3 and implemented new subclasses of the supervised metric. These classes initialize NVML. We then defined the different methods of the new subclasses to use NVML to read and save the hardware measurements.

3 System Usage

UMLAUT can be used as a standalone Python dependency or by choosing one of three docker containers that provide containerized environments for pure Python, DAPHNE, or GPU-accelerated workloads. The description of UMLAUT's standalone installation is described in detail in Deliverable 9.3. In Section 4.1 we present updates on the usage of UMLAUT, how to make use of the containerized setups in Section 4.2 and give an overview of our updated user interface in Section 4.3.

3.1 Benchmarking with UMLAUT

UMLAUT can benchmark Python pipelines and DAPHNE DSL scripts. In this section, we present two use cases: (1) benchmarking the Python pipelines on a method level and (2) end-to-end benchmarking DAPHNE DSL pipelines.

To benchmark one or more methods using UMLAUT, we follow the steps listed below:

We Initialize a Benchmark() object by specifying the name of the output database and provide a corresponding description.

```
bm = umlaut.Benchmark('KAI.db', description="Benchmark custom scripts.")
```

Subsequently, we specify the list of metrics we want to benchmark together with the measurement frequency, given as an interval value.

```
metrics = [umlaut.MemoryMetric('memory', interval=0.1)]
```

We use the benchmark object and the list of metrics to decorate all methods we want to benchmark. Additionally, we can assign a custom name to every method's benchmark using the *name* parameter.

```
@umlaut.BenchmarkSupervisor(metrics, bm, name="training_step")
def train():
```

If we run code that executes the decorated methods, UMLAUT measures the specified metrics for every decorated method and saves the collected results in the specified database file. To access and visualize the collected results we use UMLAUT's Command Line Interface (CLI) presented later in this section.

3.2 Containerization

To limit the installation overhead when using UMLAUT, we provide three docker containers [10]. One UMLAUT-only container with all UMLAUT dependencies installed to run and benchmark Python scripts, a second container that has UMLAUT and DAPHNE installed to benchmark DAPHNE DSL and DaphneLib scripts, and a third container that has UMLAUT and a version of DAPHNE with GPU support installed to benchmark GPU accelerated workloads. Using these containers allows us to easily run and benchmark all Python, DaphneDSL, and DaphneLib pipelines provided by the use case partners. In the following, we give an overview of the three containers and how to use them.

3.2.1 UMLAUT Only

The first container installs UMLAUT with all its dependencies and is designed to benchmark pure Python workloads. To build and run the container the two commands listed below are executed within the root of our GitHub repository.

```
sudo docker build -t umlaut containers/only_umlaut
```

and

```
bash containers/only_umlaut/start.sh
```

3.2.2 UMLAUT and DAPHNE

The second container installs umlaut and builds DAPHNE from source. This container is used to benchmark pipelines using daphne for their execution. To build and run the container the following commands are executed within the End-to-end-ML-System-Benchmark folder.

```
sudo docker build -t umlaut_cpu containers/umlaut_daphne
```

and

```
bash containers/umlaut_daphne/start.sh
```

3.2.3 UMLAUT, DAPHNE, and GPU

The third container installs UMLAUT but builds a DAPHNE version with GPU support. This container is used to benchmark pipelines that are GPU-accelerated. To use the container run the following commands within the End-to-end-ML-System-Benchmark folder.

```
sudo docker build -t umlaut_cuda containers/umlaut_daphne_cuda
```

and

```
bash containers/umlaut_daphne_cuda/start.sh
```

Note that the first command only needs to be executed once, afterwards the container can always be started using only the second command.

3.2.4 Custom Containers

To run custom pipelines, we might have to add custom dependencies or make custom datasets available from within the container.

Custom Dependencies

If a pipeline uses custom dependencies, we have to install these inside one of the UMLAUT containers. To do so, we navigate in End-to-end-ML-System-Benchmark/containers and choose the subfolder of the container we want to modify containing a dockerfile and a start bash script. To install custom dependencies, we add a new line starting with the "RUN" keyword followed by the SHELL command to install the dependency to the dockerfile. For example, to add a Python package, we add the following line before the last line of the dockerfile, starting with the ENTRYPOINT keyword.

```
RUN pip install example_package
```

Custom Datasets

Custom pipelines might access custom datasets. To include them in the container, we modify the docker run command in the *start.sh* bash script.

We add a custom local folder to the docker container by mounting it using *docker run*'s -v option together with the the path to the folder in the users system and the destination path in the container. In Figure 3 we show an example.

```
sudo docker run  -it --hostname daphne-container \
    -e GID=$GID -e TERM=screen-256color -e PATH --gpus all \
    -e USER=$USERNAME -e UID=$UID \
    -v /local/path/to/umlaut/repository:/app/umlaut \
    -v /local/path/in/your/system:/mounted/path/inside/the/container \
    --entrypoint /bin/bash "umlaut_cpu"
```

Figure 3: Customizing a DAPHNE & UMLAUT container.

3.3 User Interface

UMLAUT provides a command-line interface that complements its benchmarking support, allowing users to view and analyze recorded benchmarking results interactively. Compared to the interface described in Deliverable 9.3, we updated UMLAUT's plot interface by replacing MatplotLib with Plotly and added functionality to assign custom names in decorator functions. Each plot represents measurements for a single metric and is displayed using Plotly in an interactive browser window that allows the user to hover over specific values, filter the results or zoom in for more detail.

In the following, we show how to execute the *github_example* pipeline and using the file *hello_world.db* (both available in the UMLAUT GitHub repository) to showcase UMLAUT's new CLI and give impressions of UMLAUT's updated plotting functionality.
For the hello_world.db file, we start the CLI as follows.

```
umlaut-cli hello_world.db
```

This command shows a menu in the terminal where we can select a run by its UUID. We navigate the menu by using the ARROW keys, select a run using SPACE, and continue by pressing ENTER. We show an example in Figure 4.

```
? Please select one or more uuids.  (<up>, <down> to move, <space> to select, <a> to toggle, <i> to invert)
 ❯○ 55077e71-c330-424a-b01d-94921a3c666a, 2024-08-13 09:18:33.309111, description: Database for the Github sample measurements
  ○ 871dd01b-6564-4c62-bba2-72f2459587b2, 2024-08-13 09:18:58.224096, description: Database for the Github sample measurements
```

Figure 4: Selecting a pipeline run in umlaut-cli.

Using the ARROW keys and SPACE, we select one or more metrics before continuing by pressing ENTER (see Figure 5).

```
? Please select one or more uuids.  [55077e71-c330-424a-b01d-94921a3c666a]
? Please select measurement types corresponding to uuids.  (<up>, <down> to move, <space> to select, <a> to toggle, <i> to invert)
  Available types for uuid 55077e71-c330-424a-b01d-94921a3c666a
 ❯● cpu
  ● memory
```

Figure 5: Selecting the metrics of interest.

Afterward, we select one or more descriptions using the same navigation methods, as shown in Figure 6. Note that the description of a measurement is usually the name of the method that was benchmarked.

```
? Please select one or more uuids.  [55077e71-c330-424a-b01d-94921a3c666a]
? Please select measurement types corresponding to uuids.  [[1]]
? Please select descriptions corresponding to uuids and types.  (<up>, <down> to move, <space> to select, <a> to toggle, <i> to invert)
  Available descriptions for uuid 55077e71-c330-424a-b01d-94921a3c666a and type cpu
 ❯● bloat cpu
```

Figure 6: Selecting the metric for the decorated method.

After selecting which metrics should be analyzed, the corresponding plots will automatically open up in the standard browser of the system (see Figures 7 and 8).

Metric: CPU usage

CPU usage in %

110 105 100 95 90

0h 0m 0.00s 0h 0m 1.11s 0h 0m 2.31s 0h 0m 3.52s 0h 0m 4.73s

Time elapsed since start of pipeline run

Figure 7: Visualization of the CPU utilization.

Metric: Memory usage

Memory usage in MB

408 407.5 407 406.5

0h 0m 0.00s 0h 0m 1.21s 0h 0m 2.42s 0h 0m 3.62s 0h 0m 4.93s

Seconds elapsed since start of pipeline run

Figure 8: Visualization of the memory consumption.

In Figure 10, to visualize multiple executions in one plot, we select two or more UUIDs or two or more descriptions.

```
? Please select one or more uuids.  done (2 selections)
? Please select measurement types corresponding to uuids.  done (2 selections)
? Please select descriptions corresponding to uuids and types.  done (2 selections)
META_COLUMN_NAMES ['uuid', 'meta_start_time', 'meta_description']
COLUMN_NAMES ['id', 'measurement_datetime', 'measurement_data', 'measurement_unit', 'measured_method_name']
```

Figure 9: Selecting multiple pipelines in umlaut-cli.

The results are shown in one plot and can be distinguished by color as shown in Figure 10.

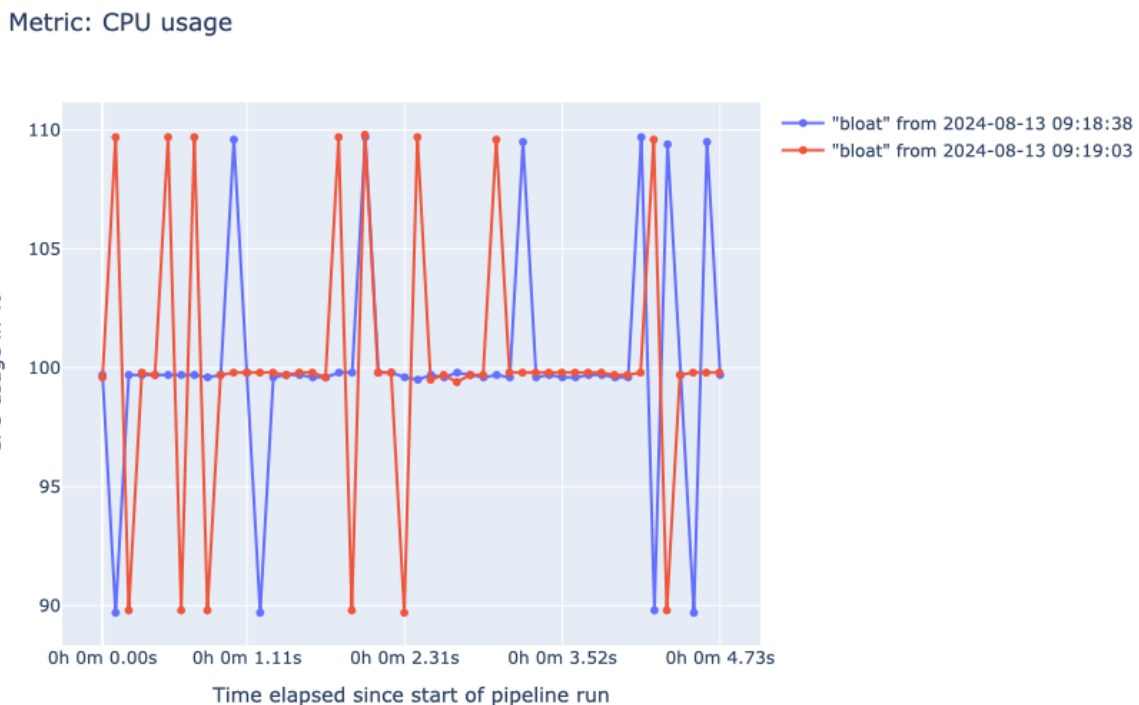

Figure 10: Visualization of the CPU utilization of multiple runs.

As an alternative to manually stepping through the CLI, we can also execute the CLI by providing the relevant information as command line arguments (see Figure 11). We specify the UUIDs using -u, the lists of types with -t, the descriptions using -d, and the plotting backend with -p. This allowed us to automatically run the visualization as the last step in our end-to-end benchmarking script.

```
try:
    subprocess.run(["umlaut-cli", "custom_script.db", "-u", uuid, "-t"] + types + ["-d"] + types + ["-p", "plotly"])
except:
    print("Umlaut could not automatically display the results in your browser.")
    print("This is expected to happen when you are inside a docker container.")
    print("Manually run umlaut-cli from outside the container.")
```

Figure 11: Utilizing umlaut-cli programmatically.

The UUID was obtained from the .uuid property of the benchmark object, descriptions of various types are saved when initializing the metrics, as shown in Figure 12.

```
metrics = []
types = []
if config["memory"]:
    metrics.append(umlaut.MemoryMetric('memory', interval=config["memoryfreq"]))
    types.append("memory")
if config["gpumemory"]:
    metrics.append(umlaut.GPUMemoryMetric('gpumemory', interval=config["memoryfreq"]))
    types.append("gpumemory")
if config["cpu"]:
    metrics.append(umlaut.CPUMetric('cpu', interval=config["cpufreq"]))
    types.append("cpu")
if config["gpu"]:
    metrics.append(umlaut.GPUMetric('gpu', interval=config["cpufreq"]))
    types.append("gpu")
if config["gpupower"]:
    metrics.append(umlaut.GPUPowerMetric('gpupower', interval=config["cpufreq"]))
    types.append("gpupower")
if config["gputime"]:
    metrics.append(umlaut.GPUTimeMetric('gputime'))
    types.append("gputime")
if config["time"] or len(metrics) == 0:
    metrics.append(umlaut.TimeMetric('time'))
    types.append("time")
```

Figure 12: Defining the metrics of interest in a Python program.

4 Use Cases

In this section, we demonstrate how to use UMLAUT for benchmarking concrete use cases. We start with the use case of pure Python pipelines in Section 4.1 that can be used to give a UMLAUT user an intuition on the overheads that UMLAUT introduces while benchmarking. Afterward, we present how to use UMLAUT to benchmark the Python and DAPHNE DSL implementation of the IFAT Semiconductors pipeline in Section 4.2 and the GPU-accelerated KAI Material Degradation pipeline in Section 4.3. Both pipelines are provided by our use case partners.

4.1 Microbenchmarking UMLAUT

To quantify the resource overhead of incorporating UMLAUT in a new environment, we implemented the three simple benchmarking use cases of sleeping, sorting, and matrix multiplication with UMLAUT. Since all use cases have predictable benchmarking results, users can validate UMLAUT's functionality and give insights on setup-specific benchmarking overheads. As a part of the UMLAUT repository, we provide a list of example pipelines that can be benchmarked. These pipelines can be used to determine the resource overhead posed by UMLAUT. The benchmark sleep script inside the meta_benchmark pipeline can be used as a baseline for all measurements. This pipeline benchmarks an idle Python process that performs so-called sleeping. By benchmarking this process one can observe the amount of resources a system needs to run Python with Umlaut benchmarking running. By default, the results of this 10-second sleep will be displayed using the Python package Plotly in a browser.

Table 3: Baseline resource utilization from UMLAUT.

	Time (s)	Memory (MB)	CPU (%)
Sort 1GB	5.20	1050	190%
Sleep (10s)	10.01	104	100%
Matrix Multiply (10,000 X 10,000)	13.93	2400	1150-2700%

In Table 3, we present an example of a UMLAUT run of the sort, sleep, and matrix multiplication pipelines. We observe a 300 MB overhead in memory consumption and an additional thread utilization used for hosting the UMLAUT process. The exact numbers depend on the system this is executed. Example measurements for this from our system are rendered in Figures 13 and 14.

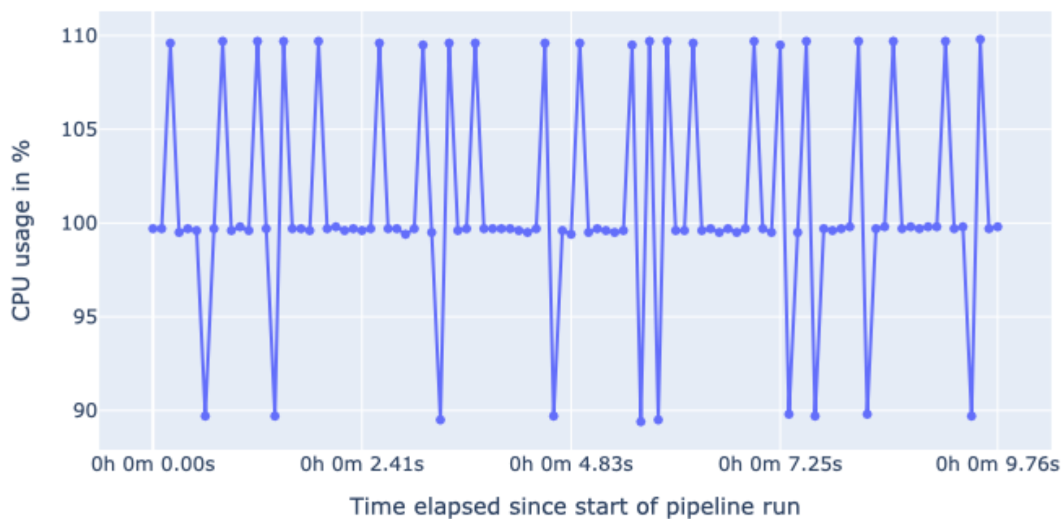

Figure 13: CPU utilization of a single thread by UMLAUT during the sleep stage.

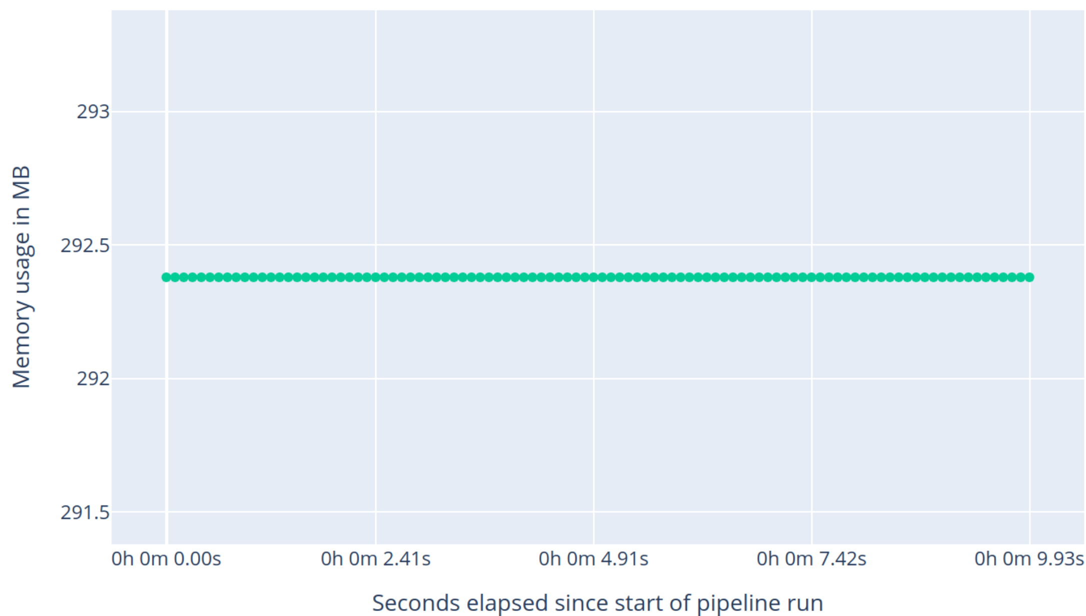

Figure 14: REPLACE WITH MAIN MEMORY CONSUMPTION

4.2 Benchmarking the IFAT Semiconductors

The IFAT Semiconductors pipeline trains a Decision Tree Classifier on sensor measurements. We have access to a Python and a DAPHNE DSL implementation.

The Python implementation can be benchmarked by navigating to the ifat_semiconductors folder and running the command:

```
python3 python/ionbeamtuning.py
```

This command runs the Python pipeline with UMLAUT benchmarking. Results will be saved into the database file our_benchmark.db. There are also results in the benchmark.db file that were recorded by the IFAT team for comparison.

The DAPHNE DSL pipeline is intended to run inside our UMLAUT & DAPHNE container. In order to run and benchmark the DSL pipeline, we execute the following command inside the folder /app/umlaut/pipelines/custom_pipeline in the container:

```
python3 run_script.py
-cmd "/app/daphne/bin/daphne daphne/ionbeamtuning.daph" -folder
"/app/pipelines/ifat_semiconductors" -g -gm -gt -gp -t -c -m
```

This uses our end-to-end benchmarking script. By default, results will be saved in the database file custom_script.db which is located in the same folder and can also be accessed outside the container.

By following these steps we obtained the following visualizations for CPU utilization and memory usage. We show the visualizations in Figures 15 and 16, respectively. We can see a sustained memory usage of about 1GB with minor fluctuations during the execution of the pipeline. We observe a CPU utilization of minimum 1000% or 10 cores under full load during the pipeline runtime with peaks of over 8000% some even over 10.000% or 80 and 100 cores under full load respectively.

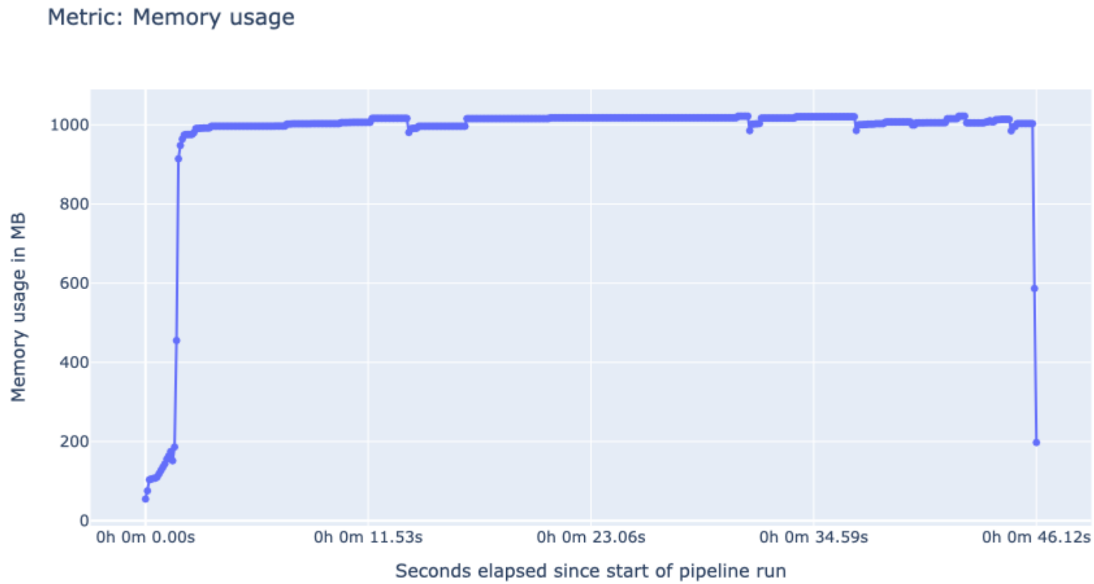

Figure 15: Memory consumption in the IFAT semiconductor analysis.

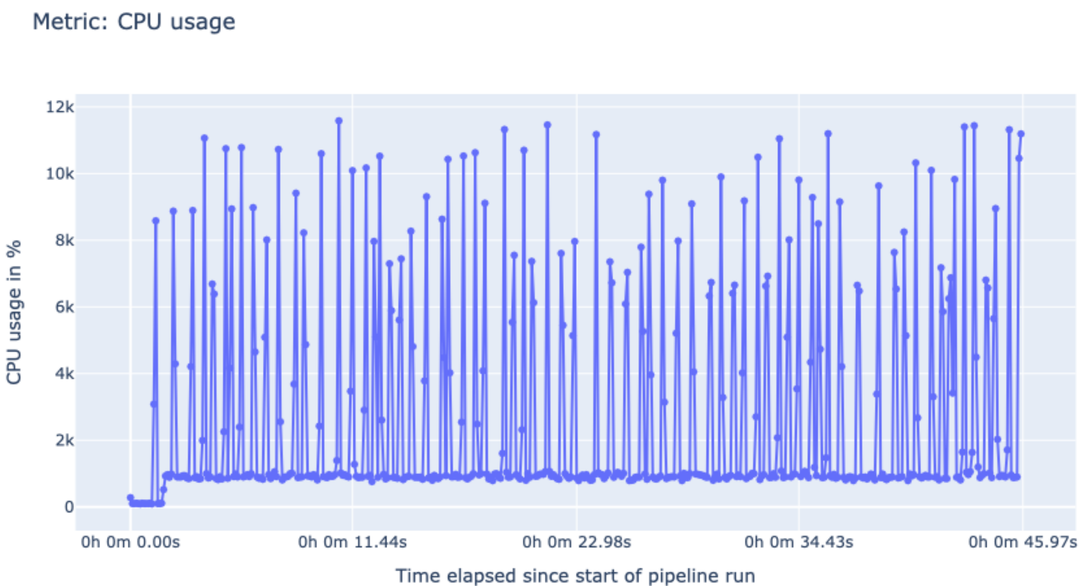

Figure 16: CPU utilization in the IFAT semiconductor analysis.

4.3 Benchmarking KAI Material Degradation

This KAI Material Degradation pipeline runs a use case based on the line simplification problem. It implements the VW[1] and RDP[2] algorithms.

4.3.1 KAI Material Degradation VW and RDP

We could not benchmark the main script of this pipeline due to a missing dataset. However, the different algorithms can be executed directly by running the VW and RDP scripts inside the Python implementation folder. Since these scripts are not structured using multiple methods, we apply end-to-end benchmarking for the Python and DAPHNE DSL pipeline for this use case. Both can be executed inside the DAPHNE & UMLAUT container by running the following command inside the folder /app/umlaut/pipelines/custom_pipeline in the container:

```
python3 run_script.py -cmd "python3 vw-perMeasurand.py"
-folder "/app/pipelines/kai_material-degradation/python-implementation" -g
-gm -gt -gp -t -c -m
```

```
python3 run_script.py -cmd "/app/daphne/bin/daphne vw-perMeasurand.daphne"
-folder "/app/pipelines/kai_material-degradation/dsl-implementation" -g -gm
-gt -gp -t -c -m
```

This uses our end-to-end benchmarking script. By default, both results will be saved in the database file custom_script.db which is located in the same folder and can also be accessed outside the container.

By following these steps we obtained the following example plots for memory usage, first from the Python pipeline (see Figure 17) and the second from the DSL pipeline (see Figure 18). We can observe a similar behavior for both processes. They first quickly load data into memory before sustaining the same memory usage for the full duration of the pipeline. Interestingly, while the DAPHNE DSL implementation uses about half of the memory of the Python implementation, it is also slower by about a factor of 3.

[1] https://en.wikipedia.org/wiki/Visvalingam–Whyatt_algorithm
[2] https://en.wikipedia.org/wiki/Ramer–Douglas–Peucker_algorithm

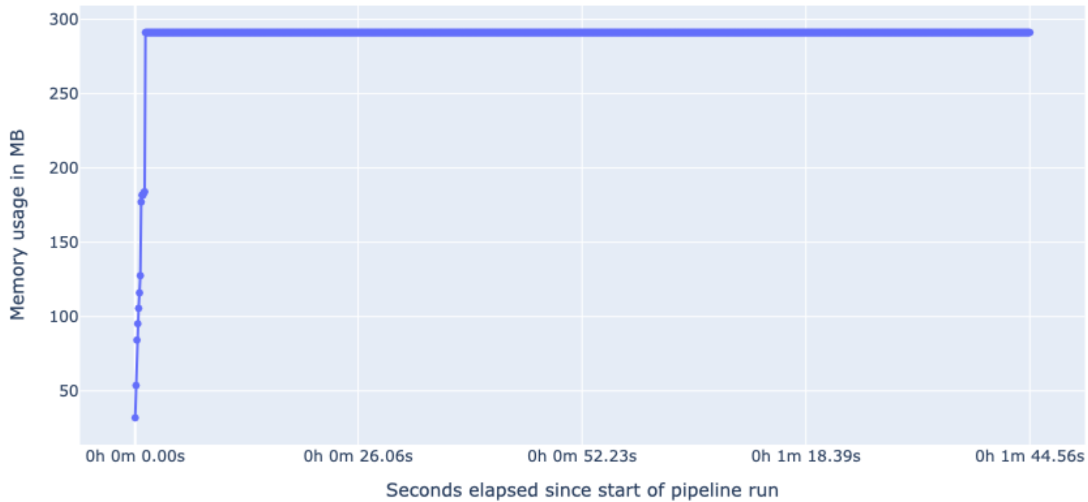

Figure 17: Memory consumption in the Python implementation of the material degradation

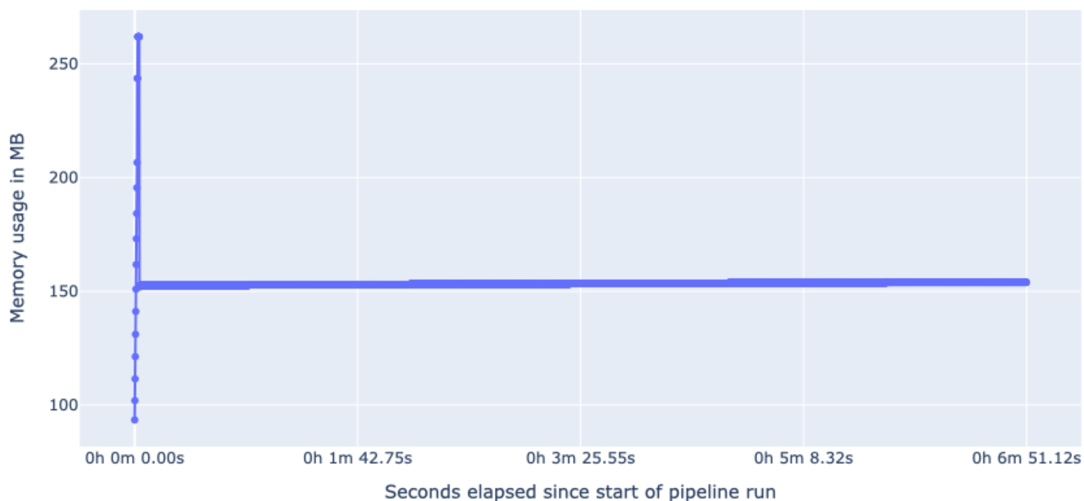

Figure 18: Memory consumption in the DAPHNE DSL implementation of the material degradation use case.

4.3.2 KAI Material Degradation ML

This pipeline trains a Convolutional Neural Network (CNN). It provides a Python implementation and a DAPHNE DSL pipeline in the two different repositories KAI_material-degradation-ml-py and KAI-Material-Degradation-ML-daphne.
The Python pipeline can be benchmarked by navigating to the kai-material-degradation-ml-py folder and running the command:

```
python3 train2-control_stride.py
```

This runs the Python pipeline with UMLAUT benchmarking. We shortened the pipeline by default to two epochs and two folders. Both parameters can be changed in lines 105 and 122 of the script. We also divided it up into different methods to showcase the method-level benchmarking of UMLAUT. Results will be saved into the database file KAI.db.

In Figure 19, we show an example visualization of the GPU power consumption we measured by following these steps.

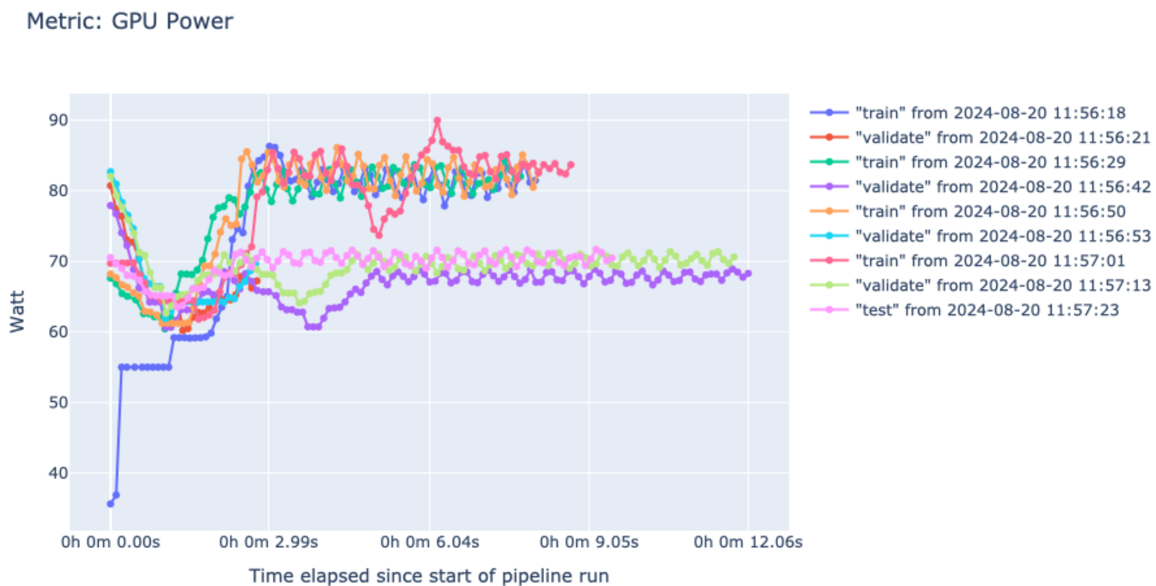

Figure 19: GPU Power consumption in the Python implementation of the material degradation

The DAPHNE DSL pipeline is intended to run inside our UMLAUT & DAPHNE with GPU support container. In order to run and benchmark the DSL pipeline the following command is executed inside the folder /app/umlaut/pipelines/custom_pipeline in the container:

```
python3 run_script.py -cmd "/app/daphne/bin/daphne
--cuda cnn-pipeline_framehack.daphne"
-folder "/app/pipelines/KAI_Material-Degradation-ML-daphne" -g -gm -gt -gp
-t -c -m
```

This uses our end-to-end benchmarking script. By default, results will be saved in the database file custom_script.db which is located in the same folder and can also be accessed outside the container. In Figures 20 and 21, we show the measurements for GPU utilization and GPU memory usage. We can observe a GPU memory usage of close to 900MB after the process first starts and loads objects into the GPU memory. We can see multiple small spikes in the GPU usage during the phase of peak memory usage which corresponds to matrix calculations from a relatively small model.

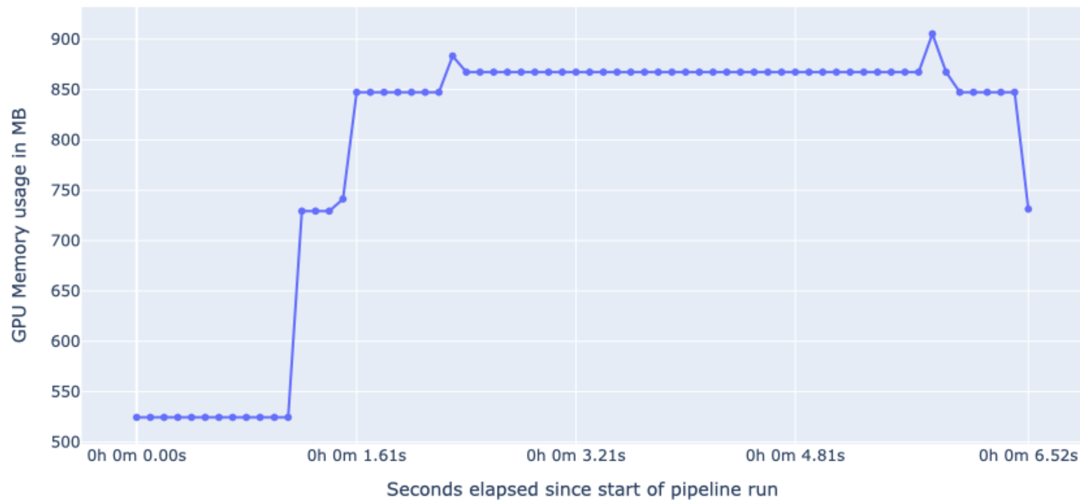

Figure 20: GPU memory consumption in the DAPHNE DSL implementation of the material degradation use case.

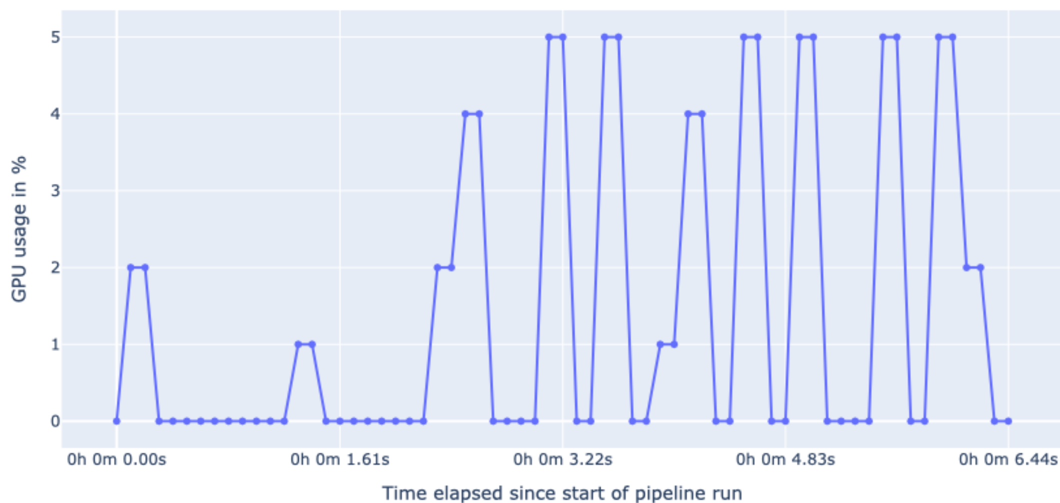

Figure 21: GPU utilization in the DAPHNE DSL implementation of the material degradation use case.

5 Conclusion

In this deliverable, we presented our implementation of the final benchmarking toolkit prototype. We focused on the extensions and improvements compared to the initial prototype implementation presented in Deliverable 9.3. The main improvements include (1) the implementation of predefined and predictable use cases to quantify the resource overhead of incorporating UMLAUT, (2) limiting UMLAUT's installation overhead by providing different DOCKER-based setups, (3) extending UMLAUT to support benchmarking GPU-based metrics, and (4) improving the user interface by switching to Plotly.

6 References

[1] UMLAUT Documentation. https://hpides.github.io/End-to-end-ML-System-Benchmark/index.html, accessed 28.10.2024.
[2] UMLAUT GitHub Repository. https://github.com/hpides/End-to-end-ML-System-Benchmark, accessed 28.10.2024.
[3] Ihde, Nina, et al. "A Survey of Big Data, High Performance Computing, and Machine Learning Benchmarks." Technology Conference on Performance Evaluation and Benchmarking. Springer, Cham, 2021.
[4] DAPHNE Deliverable 9.2: Initial Benchmark Concept and Definition.
[5] DAPHNE Deliverable 9.1: A survey of Benchmarks from DM, HPC, and ML Systems.
[6] DAPHNE Deliverable 9.3. Initial Prototype of the Benchmarking Toolkit.
[7] DAPHNE Deliverable 8.2. Improved Pipelines all Use Case Studies.
[8] DAPHNE Deliverable 8.3. Benchmarking results all use case studies.
[9] NVIDIA NVML. https://developer.nvidia.com/management-library-nvml, accessed 28.10.2024.
[10] Docker. https://www.docker.com/, accessed 28.10.2024.