MAchinE Learning for Scalable meTeoROlogy and cliMate

First version of workflow tools published that allows to perform quarterly benchmarks of ML solutions

Markus Abel, Saleh Ashkboos, Tal Ben-Nun, Matthew Chantry, Greta Denisenko & Fabian Emmerich

www.maelstrom-eurohpc.eu

D2.2 First version of workflow tools published that allows to perform quarterly benchmarks of ML solutions

Author(s):	Markus Abel (4cast), Saleh Ashkboos (ETH), Tal Ben-Nun (ETH), Matthew Chantry (ECMWF), Greta Denisenko (4cast), Fabian Emmerich (4cast)
Dissemination Level:	Public
Date:	March 31, 2022
Version:	1.0
Contractual Delivery Date:	31/03/2022
Work Package/ Task:	WP2/ T2.2 T2.3 T2.4 T2.5 T2.6
Document Owner:	4cast
Contributors:	ETH, ECMWF
Status:	Final

MAELSTROM

Machine Learning for Scalable Meteorology and Climate

Research and Innovation Action (RIA)

H2020-JTI-EuroHPC-2019-1: Towards Extreme Scale Technologies and Applications

Project Coordinator:	Dr Peter Dueben (ECMWF)
Project Start Date:	01/04/2021
Project Duration:	36 months

Published by the MAELSTROM Consortium

Contact:
ECMWF, Shinfield Park, Reading, RG2 9AX, United Kingdom
Peter.Dueben@ecmwf.int

The MAELSTROM project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955513. The JU receives support from the European Union's Horizon 2020 research and innovation programme and United Kingdom, Germany, Italy, Luxembourg, Switzerland, Norway.

Contents

List of Figures

List of Tables

1 Executive Summary

Content of this Delivery: status of workflow tools published.

- Overall status: workflow tools are on track, delay in deployment handling
- Workflow tools are complete and published
- Delay in HPC deployment due to security restrictions
- Benchmarking done with Deep500
- Different User Interfaces are developed
- CliMetLab is used for data handling
- During the time of the project, alternatives have emerged with larger development community
- Countermeasure for HPC risks: focus on security and evaluate integration of alternative tools to accelerate development

2 Introduction

2.1 About MAELSTROM

To develop Europe's computer architecture of the future, MAELSTROM will co-design bespoke compute system designs for optimal application performance and energy efficiency, a software framework to optimise usability and training efficiency for machine learning at scale, and large-scale machine learning applications for the domain of weather and climate science.
The MAELSTROM compute system designs will benchmark the applications across a range of computing systems regarding energy consumption, time-to-solution, numerical precision and solution accuracy. Customised compute systems will be designed that are optimised for application needs to strengthen Europe's high-performance computing portfolio and to pull recent hardware developments, driven by general machine learning applications, toward needs of weather and climate applications. The MAELSTROM software framework will enable scientists to apply and compare machine learning tools and libraries efficiently across a wide range of computer systems. A user interface will link application developers with compute system designers, and automated benchmarking and error detection of machine learning solutions will be performed during the development phase. Tools will be published as open source.
The MAELSTROM machine learning applications will cover all important components of the workflow of weather and climate predictions including the processing of observations, the assimilation of observations to generate initial and reference conditions, model simulations, as well as post-processing of model data and the development of forecast products. For each application, benchmark datasets with up to 10 terabytes of data will be published online for training and machine learning tool-developments at the scale of the fastest supercomputers in the world. MAELSTROM machine learning solutions will serve as blueprint for a wide range of machine learning applications on supercomputers in the future.

2.2 Scope of this deliverable

2.2.1 Objectives of this deliverable

Deliverable 2.2 aims at the development of an open-source ML platform that implements the MAELSTROM protocol of Task 2.1. The platform should provide a user-friendly workflow for developing and benchmarking ML solutions. Users will be able to store their ML solutions in the MAELSTROM protocol and reconstruct it at later times to achieve the maximum reproducibility of ML applications. Moreover, ML models created on the platform and stored in the MAELSTROM protocol will be shareable with other users of the platform so they can reproduce the ML solutions. The initial version of the platform due with this Deliverable should provide the first version of ML solutions built on the first version of benchmark datasets delivered by Task 1.2.

2.2.2 Work performed in this deliverable

The Deliverable included the development of three different components:

- Unification and acceleration of W&C data input/output (I/O)
- Benchmarking of deep learning ML solutions
- ML platform

W&C Data I/O

In an effort to simplify, unify and accelerate data I/O for W&C applications, a Python package was developed that provides an interface to the ECMWF cloud which holds the data of the first version of the benchmarking datasets provided by Task 1.2 (Section 3.4). For each of the applications, a plugin has been developed that gives access to the respective data. The implemented caching methods achieve accelerated data I/O.

Bechmarking Deep Learning Models

To allow benchmarking of ML solutions that use deep learning techniques, a framework was developed (Sections 3.2 and 3.5). This framework allows detailed investigation of deep learning models and enables users to create reproducible recipes to create ML models on different datasets and hardware.

ML platform

For the MAELSTROM ML platform, the backend and a preliminary version of the Graphical User Interface (GUI) were developed. The backend was built with several components that allow the development, execution and storage of ML solutions. The GUI enables a user to visualize the execution of their implementation. The steps that produce the ML model can be executed on different hardware architectures such as a desktop machine, cloud computers, or high-performance computer clusters. While the first two are fully supported, the implementation of the latter has suffered from unexpected obstacles that appeared due to the security restrictions by the HPC systems (i.e. Jülich Supercomputing Center, JSC).

2.2.3 Deviations and counter measures

An essential part of the ML platform is to allow users to execute their ML solutions on HPC systems. The implementation of this feature, however, has been delayed due to security restrictions that prevent vulnerability of the systems. In a first attempt, we have tried to use a special interface provided by JSC that allows execution and supervision of jobs. This turned out not to be realizable with the software architecture of the ML platform at the time. As a result, we had to initiate a large-scale reconstruction of the architecture. With the help of the responsible development team at JSC, current investigations are directed to alternative solutions which will allow a more user-friendly access.

The deployment and development on the E4 cluster was easier since the access concept is much more friendly for the executor deployment. Security is still at a very high level, though.

An approach different from JSC, at European project level, is the ICEI-FENIX project[1] that is entitled to develop a standard interface for interactive computing with the usage of Jupyter on the PRACE center. The evaluation of this solution with respect to the security issues will be subject of work within the next months.

[1]www.fenix-ri.eu

3 Workflow tools by Work Package Tasks

3.1 Workflow platform development (Task 2.2)

Any ML workflow can be split into at least three separate steps that reflect individual, isolated instructions: data loading and cleaning, feature engineering, model training and evaluation. These steps represent an entire ML model pipeline. The steps in a pipeline build upon one another and may be executed sequentially or in parallel. Hence, from a mathematical perspective, pipelines can be represented by *directed acyclic graphs* (DAGs).

The Mantik ML tool[2] is an open-source software[3] that allows executing DAGs. Users can define computational steps and plug them together to form a pipeline. Internally, Mantik creates computational graphs that – if triggered by the user – can be executed on any hardware infrastructure such as a local machine, cloud computers, or HPC architectures. In the Mantik framework, each node in a DAG is an isolated, containerized microservice that is responsible for a single, well-defined task. Depending on the type of node, i.e. if a node is a root or a leaf node, it implements one or two interfaces, respectively, that represent the output and/or input of the node. This concept makes pipelines very robust and reproducible.

To operate, Mantik is split into various different components (see Fig. 1):

1. Engine
2. Coordinator
3. Software development kits (SDKs)
4. Bridges
5. Model database

In the following, each component and its responsibility in the Mantik framework will be explained in detail.

3.1.1 Engine

The Engine is the central component of Mantik. It allows users to create individual pipeline steps. Using previously defined steps, users can take these steps and plug them together to form a pipeline. The Engine then builds the computational DAGs representing the user-defined pipelines and triggers their execution upon

[2]www.mantik.ai
[3]www.github.com/mantik-ai/core

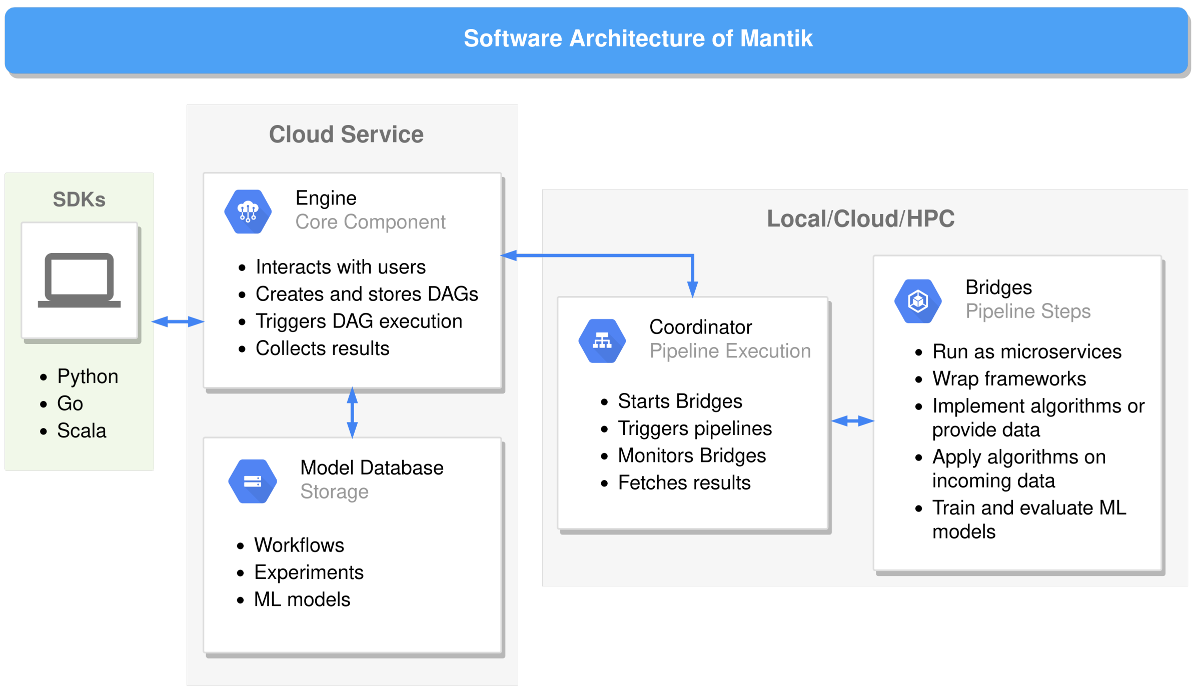


Figure 1: Sketch of the software architecture of the Mantik framework.

user request. Pipeline execution, i.e. direction of the input and output of each step is handled by the Coordinator, which runs on the hardware that was requested by the user. Once a pipeline was executed, the Engine supervises the execution process and, as soon as the pipeline is finished, collects the outcome, which can be of any kind (e.g. data or a trained model), and passes it back to the user.
Due to containerization, the Mantik Engine can be deployed and run on any machine: users may operate their own Engine or connect to a publicly available Engine running in a cloud service such as Amazon Web Services, Google Cloud Computing, or Azure. Users can interact with the Engine via the SDKs that are available in different programming languages. Currently supported are Python, Go, and Scala. These SDKs allow creating and running pipelines as well as fetching the results of a pipeline from the Engine.

3.1.2 Coordinator

The Coordinator is responsible for the execution of DAGs. Due to the containerization concept of Mantik, it is able to execute pipelines on various different hardware architectures. Each pipeline step is spawned as a microservice that is informed about its input and output. I.e., the Coordinator tells each microservice whether it will receive an input, and where it should forward its output to. As soon as the microservice representing the DAG's root node is ready, the Coordinator initiates the pipeline process. If requested by the Engine, the Coordinator has access to all microservices and the information about their state and, as a result, can inform the Engine as soon as a pipeline has succeeded – or failed – and report the outcome.

Currently, the Coordinator supports different hardware architectures. Well supported is the execution of pipelines on local machines with Docker or in a cloud with Kubernetes. For HPC architectures, the Coordinator only offers very preliminary support. Currently, we are working to extend the usability of the Coordinator on HPC by focusing on the hardware architecture of the JUWELS cluster of FZJ. Subsequently, we will transfer this solution to apply it on the E4 hardware.

The aim is to be able to run the Coordinator on login nodes of the cluster. It then can request computing resources and run each microservice on the provided compute nodes. Due to the heavy security restrictions on HPC clusters, it is not yet possible for Engine and Coordinator to communicate with one another and exchange any sort of information. Specifically JSC only allows connecting to the cluster via SSH – and outgoing connections are generally forbidden. As a result, we are only able to manually execute computational graphs created by the Engine via batch scripts (`sbatch`). The Coordinator is also unable to report any pipeline results to the Engine but can only store them locally in a user-specified position on the cluster storage. Thus, our short-term goal is to allow running the Coordinator as well as the Engine inside the cluster. The long-term solution, however, should be to use a special form of authentication that allows for- and backward communication between an Engine outside, and a Coordinator inside the cluster. One possible solution could be to use JSC's solution for their JupyterHub "Jupyter JSC".[4]

3.1.3 SDKs

The SDKs have two main features:

1. Providing an interface to the Engine to set up and execute pipelines.

2. Development of pipeline steps (Bridges).

The Mantik SDKs provide various methods to interact with the Engine. To begin with, they allow connecting to any Engine. This can be a local Engine or one that runs in a cloud service and is exposed to the internet. In addition, the user can put individual pipeline steps (i.e. Bridges) together and let the Engine create and store the DAG reflecting that pipeline. The evaluation of DAGs is lazy, meaning

[4] www.jupyter-jsc.fz-juelich.de

that the execution is only triggered if requested. Upon completion, the result can be requested from the Engine.

3.1.4 Bridges

In the Mantik framework, pipeline steps are Mantik Bridges.[5] Bridges encompass a certain functionality and can be developed in any of the supported programming languages (Python, Go, Scala). To be more precise, they implement a certain (data processing) algorithm and, depending on the type of the Bridge, are able to apply it on a given input and forward the result to a subsequent Bridge or the Engine. To achieve this, Bridges wrap one or several frameworks and allow implementing algorithms using these frameworks. A Python Bridge, for example, may encapsulate frameworks such as NumPy, pandas, scikit-learn, TensorFlow or PyTorch – either individually or as a set.
Bridges have configuration files in which all its meta information are defined – the *MantikHeader*. It describes details such as the Bridge's name, its input and output and what purpose it is suitable for. There exist three types of Bridges:

- DataSet
- Algorithm
- Trainable

A *DataSet Bridge* typically is the root node of a DAG and has no input, but only an output. This means that they are designed to load a certain type of data and provide it for the pipeline. Due to the microservice architecture of pipelines in Mantik, this allows scalable data loading by parallel execution of DataSet Bridges. The output of several Bridges can then be combined and passed to subsequent pipeline steps.
Algorithm Bridges implement algorithms that can serve different purposes. They can be designed to perform data cleaning, filtering, augmentation, or feature engineering. Hence, they have one input, which is data streaming into the Bridge, and one output, which is the outcome of the implemented algorithm. In addition, Algorithm Bridges may also already implement any ML algorithms whose result can be used for any following data processing step.
The third type of Mantik Bridges, the *Trainables*, allow implementing the training and evaluation of ML models with the data produced by prior pipeline steps. They require one input (preprocessed data) and can have up to two outputs. If used to

[5] www.github.com/mantik-ai/bridges

train a model, they provide the trained model and any user-defined metrics that allow analyzing the quality of the model. On the other hand, Trainables (i.e. the models they produced) can be used in pipelines to apply the trained models on the data in order to create the desired outcome.

3.1.5 Model Database

The Mantik AI Database allows to persistently store and reuse bridges and items and reproduce full workflows. The database covers user accounts, MantikItems (including bridges and headers), workflows, including the generated computational graph, and experiments, i.e. single execution of workflows including MantikItem configurations and training results (training statistics, models, meta statistics on resource utilization).
We chose a relational database design in PostgreSQL to mirror relations between objects in Mantik. Files, such as serialized trained models, are not stored by the database itself. Instead, a file-service is connected that stores files and makes them available via file URL. To avoid repeated entries, the database is normalized. Thus semantic data-models differ from the actual tables; here, only the semantic models are described:

- Account: A user account database entry contains the user name, email and a password hash. The combination of name and email must be unique.
- Files: A file entry contains information on its owner, file location in the file service and the file hash to check whether a file is already present in the file service.
- MantikItems: A MantikItem is represented as a combination of its owner, the MantikHeader and used bridge, a reference to the payload file, creation time and its version.
- Workflow: A workflow is a directed acyclic graph (DAG) in which MantikItems are the nodes. Each workflow has an owner, optionally a name and description, and stores the defining Mantikfile.
- Experiment: An experiment refers to one execution of the workflow. MantikItem configurations are stored referring to nodes in the workflow DAG. Training results and meta statistics on resource utilization may also refer to single nodes or the workflow as a whole. Trained models are stored as files in the file-service.

All semantic models can be referenced by ID.

File service

The file service handles storage of larger files that are only referenced in the database by a shared file ID. It is meant to manage e.g. AWS S3 storage buckets and is currently implemented for a local file service. The file service API provides the methods `store_file` and `load_file` for file handling.
The file service is called by the database client; the user is not meant to directly access it.

Architecture

Because Mantik is built to support reproducibility, sharing and mix-and-match of components, a lot of repetitions are to be expected in the AI database. To save on time and have a clearer organization, the database is normalized up to a point where each table represents one immutable object that can be used inside Mantik. This way, repetitions are avoided and IDs have a semantic dimension by guaranteeing uniqueness.

Overlap With Other Projects

The architecture of the Mantik model database has been compared to the MLFlow[6] implementation and is astonishingly similar, in particular with regard to the model format for Python models (Python pickle). As described above, we think it may be very useful to join our ideas with MLFlow. At the current stage, though, this is not yet possible.

3.2 Benchmarking (Task 2.3)

For the benchmarking task, we use and extend Deep500 [1], a modular benchmarking infrastructure for high-performance deep learning. Deep500 factorizes training neural networks at a high level into four levels: operators, network processing, training, and distributed training. This modular definition allows users to easily construct reproducible *recipes* for training ML workloads on different datasets and system. Recipes then generate detailed reports (e.g., logs, violin plots) on specific performance aspects during training, as chosen by the users.
In the MAELSTROM project, we are extending the framework in collaboration with the applications in WP1 to create a simple solution that is compatible with the weather and climate workloads, with one recipe per application. At the same time,

[6] www.mlflow.org

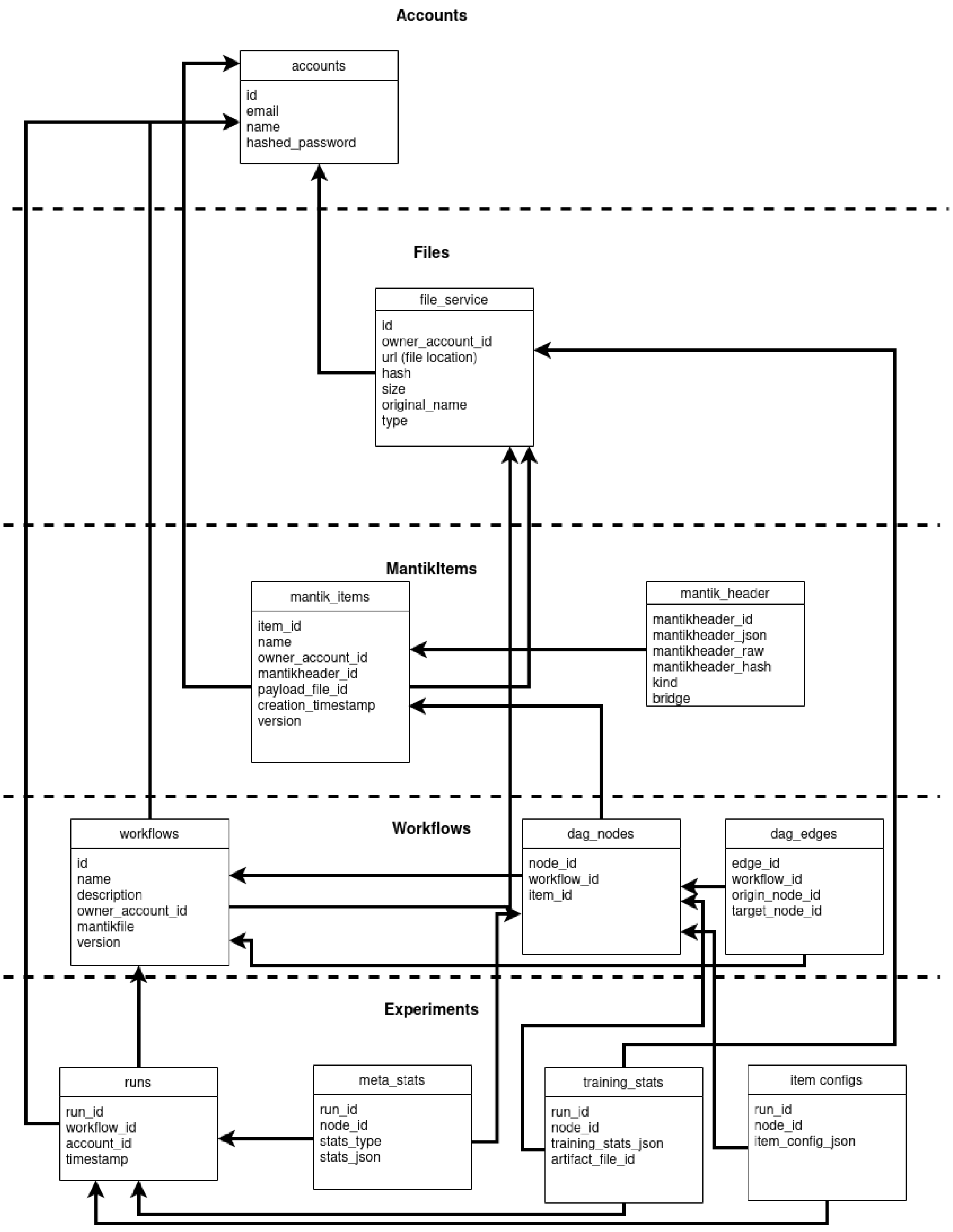


Figure 2: Schematic view of the database tables and their relations. Vertical dashed lines separate semantic data models. Arrows indicate foreign key relations, referencing other tables by ID.

the framework should seamlessly provide the required benchmarking data (e.g., for WP3) on all tested systems. In particular, in the first 12 months of the project we added the following features:

- **Weather Dataset Class**: We define a custom dataset class (the class is named `D500WeatherDataset`) for loading weather datasets, which are large and fragmented w.r.t. the learning tasks. We also define a Sampler class to handle random sampling of weather data and apply necessary data augmentations.
- **PyTorch Model**: We improve PyTorch integration in order to reduce overheads generated by Deep500 during training.
- **Loss function**: We provide loss functions that are commonly used in weather and climate workloads. Specifically, Structural Similarity Index (SSIM) [2], latitude-weighted error, and linear combinations (`MSSIMLoss`) thereof.
- **Scheduler**: We define a hyperparameter scheduler for Cyclical Learning Rates [3] (`CyclicLRScheduler`), which is used by Application 4 in WP1.
- **Measurement**: We implement a composite event hook to measure training, validation, and test loss, while also measuring multiple performance metrics: training time per iteration and per epoch for both training and test phases (`RMSETerminalBarEvent`).
- **Flexibility**: We introduce command-line arguments to Deep500 recipes, such that mutable aspects of the recipe can be modified in hyperparameter searches or by larger systems such as Mantik.

With those features in place, creating recipes for any of the WP1 applications should follow a standard template, easing the work for the ML practitioners and abstracting away low-level details of benchmarking metric collection.

3.3 User Interface (Task 2.4)

The user interface as implemented in the first 12 months of the MAELSTROM project realizes the requirements found up to day. A software interface, in general, denotes the values that can be passed to an application or a method. The workflow tools are based on the Mantik package, which in turn has a variety of methods available to execute ML tasks. For MAELSTROM, we adapted the protocol and published it in the open source Mantik repository on github as an add-on. We have thus developed abstract interfaces for the general usage of the methods which in turn are based on the MAELSTROM protocol. Additionally, to visualize the execution, a Graphical User Interface has been built where the execution of the tasks can be monitored.

In general, we notice that the MLFlow Project has outperformed the Maelstrom development speed in recent years, since it has 71 contributing companies. Consequently, the option to use as much as possible of MLFlow is currently evaluated. In turn, MAELSTROM could contribute the executor flexibility and the HPC usage to MLFlow. Both projects are Open Source, however with different licences (Apache 2.0 vs. GNU AGPL), which needs to be double-checked for compatibility. The following description contains work done within the MAELSTROM project, and has no interference with MLFlow.

As already described above, one particular challenge is the communication to HPC, which typically is highly secure. It cannot be easily bypassed such that the monitoring through an external GUI, e.g. in a browser on a local machine, is not possible at the moment. The current activities aim at a communication which allows at least for a partial display of the executor status.

In the first approach, to allow first applications to be run by the MAELSTROM protocol, the major execution steps are handled by methods that in turn are passed to the Executor. For HPC, the current status is that we strip the execution graph, pass it to the HPC environment and execute it standalone using SLURM. The executor is then started in a Singularity container and a "Coordinator" cares for the parallel execution on multiple nodes. The execution on GPUs is subject of future development.

3.3.1 Command Line Interface

There are roughly two groups of Command Line Interfaces (CLI): for the backend user who manages the operation of the workflow tools (using the modified Mantik engine), and the ML developer who is using the MAELSTROM protocol to run machine learning tasks based on the protocol developed.

There are no additional steps needed for communication, however, the URL of the MAELSTROM engine needs to be known. Once the Engine is running, the communication with it is handled using the following syntax. The programming languages used are Python for ML on the HPC infrastructure and in general Scala, which is managing the backend with the Mantik Engine.

You can issue basic operations without interacting with Python or Scala.

The tool is called with the command `mantik` and communicates to a running Mantik Engine via the mnp (Mantik/MAELSTROM Node Protocol). This protocol is based on gRpc (Google Remote Procedure Call - grpc.io) which is an excellent basis for efficient passing of messages.

Basic Usage

command execution `mantik <command-name>` executes the given command.
help For a help text, call `mantik help` or `mantik <command> -help`
flags Most commands have a set of flags which help to modify their behaviour.
In the following, we describe the subcommands and their usage.

Item Management

An Item is a container with a specific set of packages and a payload, i.e. the actual code to be executed.
show items Items are shown by Mantik Items
a single item Details on a specific item are shown, including Mantik header, by `mantik item <name>`
item extraction Extract an item into a directory for inspection or local build with `mantik extract -o <directory> <name>`
build item from directory A directory is packed into a Mantik item with `mantik add -n <name> <directory>`
tag items Items are tagged with a new name calling `mantik tag <name> <new-name>`

Log into a remote registry

You can log into a remote registry (e.g. Mantik Hub) using
login `mantik login`
logout `mantik logout`

Transfer items to and from a remote Registry

You can download and upload Mantik Items to a remote registry (e.g. Mantik Hub). In order to do so, you have to be logged in (cf. above).
download `mantik pull <name>`
upload `mantik push <name>`

Deploy Algorithms or Pipelines

It is possible to deploy items on your local Mantik engine, using Docker or Kubernetes.
deployment `mantik deploy <name>`

3.3.2 Model Database Interface

The user interface to the database is provided by Python data classes for the aforementioned semantic data models alongside an SQL client. The client is capable of handling the uploading logic based on semantic data models. Some internal logic, such as testing for uniqueness of uploaded files, is implemented in the database itself as an SQL function.

Connection

The client is initialized with information on database connection:

- `db_user` database user name
- `password` database user password
- `dbname` database name
- `port` database connection port
- `host` database host

Semantic data models

Dataclasses are provided that abstract information into semantic data models and mediate between user interface and database client.

- `MantikAccountDBEntry`: User name, email, password hash and (optional) user ID.
- `MantikFileDBEntry`: File information (including file hash), owner ID, (optional) file URL referencing the file service.
- `MantikItemDBWrapper`: MantikItem information (name, owner, version), MantikHeader, payload (associated files).
- `MantikWorkflowDBWrapper`: Workflow information, DAG nodes, DAG edges.
- `MantikExperimentDBWrapper`: Run information, training statistics, meta statistics, MantikItem configurations.

Add Mantik entities to the database

Mantik entities refers to the semantic data models wrapped into above mentioned data classes. By default, each database insert returns the (generated) ID of the main table each entity refers to.

- `client.add_user(MantikAccountDBEntry)`: Add a user, with a constraint on uniqueness of user name and email. Returns user ID.
- `client.add_file(MantikFileDBEntry)`: Add a file to the database and file service if file hash is not present in the table. Returns file ID.
- `client.add_item(MantikItemDBWrapper)`: Add a MantikItem, MantikHeader and payload if not already present. Returns item ID.
- `client.add_workflow(MantikWorkflowDBWrapper)`: Add a workflow and its DAG representation. Returns workflow ID.
- `client.add_experiment(MantikExperimentDBWrapper)`: Add an experiment (run), statistics, generated files and MantikItem configurations. Returns run ID.

Report

A Python function to retrieve all data of a single experiment and create a Markdown report is available as an extension of the client as `ai_client.extras.report.generate_report(client, experiment_id) -> str`. An example report is shown below:

Report on experiment

User

Name: User Name

Email: user@mail.domain

ID: 5a256fb9-8f35-417e-a273-5f7ba2ee4a22

Bridges

Bridge ID	Name	Owner	Header	Creation Date	Payload	Version

Get the bridges with

Items

Item ID	Name	Owner	Header	Creation Date	Payload	Version
02e39f0a-b8ed-4857-a962-db4a188d9073	my_awesome_ml_notebook	27080a18-c928-4ac2-bcf4-3829927a87ed	2c5911cd-d379-424d-ae69-dfb1afb9dd68	2022-03-10 19:16:54.762248+00:00	cc5e63ae-487a-4935-a543-23040bc3b637	latest

Get the items with

`curl file:///home/user/ai-database/.file_service/282e64fc-7c84-418d-9db6-d9331702cf63`.

Model

The trained model is available from

`curl file:///file_service/ai-database/.file_service/cbbc66c8-e043-4e0e-849a-45cc1812465c`.

The training stats are

Run ID	Node ID	Stats JSON	aritfact_file_id
b96e777f-9c94-4220-acd0-01bd3c5eeaf1	None	score=0.9393433500278241	69b28d21-7856-4acf-a73f-bf4a7199fdc4

Workflow

ID	Name	Description	Owner	Creation Date	Mantikfile	Version
76ab2f2d-045a-4f9a-b4dd-7cfd9ad4c40f	My awesome workflow	Just an awesome workflow	5a256fb9-8f35-417e-a273-5f7ba2ee4a22	2022-03-10 20:16:54.776992		1.0

The workflow has been used in experiment runs

b96e777f-9c94-4220-acd0-01bd3c5eeaf1

Run

Run Info

ID	Workflow ID	Owner	Execution date
b96e777f-9c94-4220-acd0-01bd3c5eeaf1	76ab2f2d-045a-4f9a-b4dd-7cfd9ad4c40f	5a256fb9-8f35-417e-a273-5f7ba2ee4a22	2022-03-10 20:16:54.807292

Run training stats

Run ID	Node ID	Stats JSON	aritfact_file_id
b96e777f-9c94-4220-acd0-01bd3c5eeaf1	None	score=0.9393433500278241	69b28d21-7856-4acf-a73f-bf4a7199fdc4

Run meta stats

Run ID	Node ID	Stats Type	Stats

b96e777f-9c94-4220-acd0-01bd3c5eeaf1	None	Runtime	start=1646939810.7822099
			end=1646939810.8224137
			duration=0.04020380973815918

3.3.3 HPC Interface

For the HPC interface, several possibilities have been investigated: the UNICORE (i.e. pyunicore) client, using slurm directly on the infrastructure, and eventually the executor interface we have specifically developed within MAELSTROM.

UNICORE[7] is a framework to enable federated access to computing resources: "UNICORE provides tools, services and RESTful APIs for integrating HPC compute and data resources into federated environments, in a secure and transparent fashion."

Ideally, it should be used as ground to build upon. We have spent at least a person month on UNICORE usage. It turns out to be a very useful tool, that, however, can only realize batch execution on HPC hardware. Since the workflow is aimed at feedback from the job executed we searched for alternatives. The basic insufficiency was that UNICORE delivers information only about a single job, and we aim at the details of each subunit, e.g. if hyperparameters are searched. That will open the way for optimization and a more efficient and energy-saving usage of the valuable HPC ressources. UNICORE basically uses the available scheduling and queuing mechanisms of the infrastructure one aims at. For JSC and E4 the job execution is handled by Slurm.

Other common schedulers on HPC are Torque, Slurm, LSF, SGE and LoadLeveler. A good comparison of features is found at the Slurm website[8]. So, to directly execute a batch job, we can directly use Slurm instead going through UNICORE. That still does not solve the issue of communicating the job status back to the Engine, but there are workarounds within Slurm, e.g. through emails.

For MAELSTROM, we investigated together with the team at JSC the possibility to enable backward communication using the secure shell. The cooperation turned out to be very helpful in clarifying the available options. In Juelich, for usage of Jupyter notebooks, ssh has been enabled to communicate with a local Jupyter notebook, however, such that no security breach can occur (or rather the probability of security breach is minimized). Consequently, within the MAELSTROM project, a redesign of the execution has been developed such that a HPC executor, or Coordinator, executes one task/model on HPC such that it lives on the entrance node and distributes work to other nodes of the cluster.

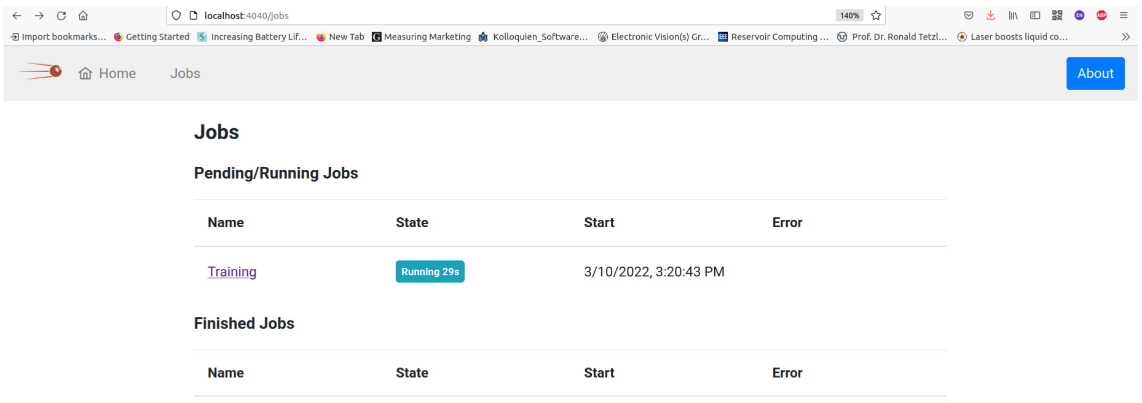
Currently, a Python interface is implemented, described above in the SDK section. We implemented an example based on the Application A6 such that other users are able to use it.

[7] www.unicore.eu

[8] https://slurm.schedmd.com/rosetta.html

3.3.4 GUI for Executor and Job Status

The Engine executes many jobs, which in turn consist of subunits. This is represented by the execution graph. The graphical user interface (GUI) shows all jobs submitted to MAELSTROM/Mantik in real time such that the status of the job execution can be followed live, cf. Fig. 3. In the background, logging is used to track any change of the status.



Jobs

Pending/Running Jobs

Name	State	Start	Error
Training	Running 29s	3/10/2022, 3:20:43 PM	

Finished Jobs

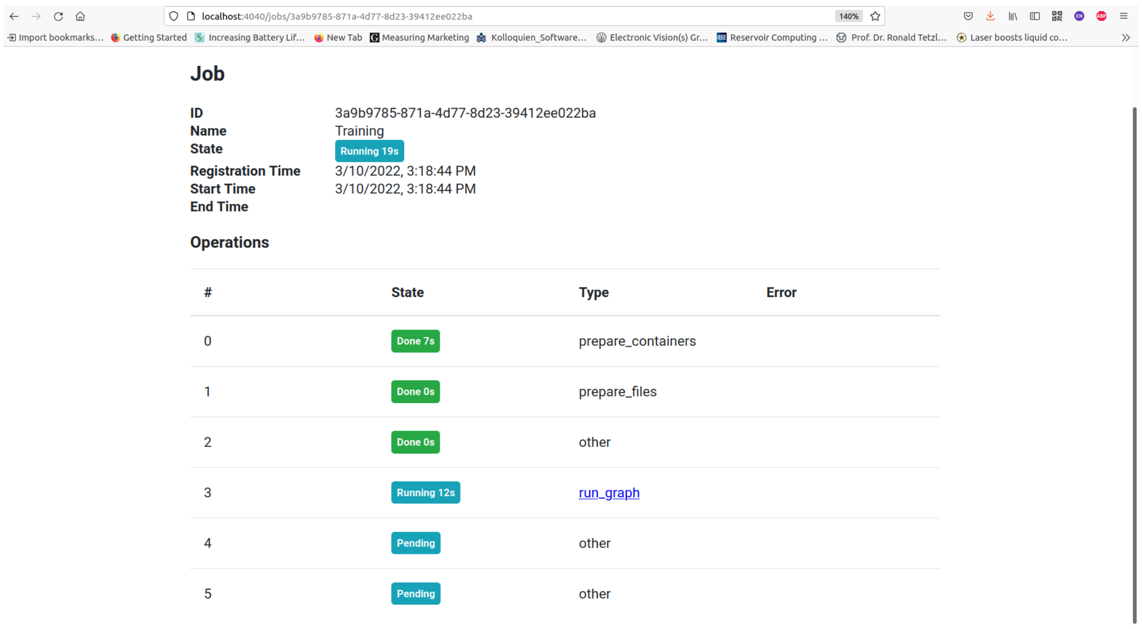
Name	State	Start	Error

Figure 3: Image of the GUI to the Engine. The jobs that are currently executed and the already finished jobs are shown.

To track the status of a job with all its dependencies and subgraphs the corresponding panel shows the current job status, cf. Fig. 4.

Finally, the execution graph itself may become very complex, and visualization may help to understand possible optimization strategies. It can be viewed in a separate panel, cf. Fig. 5.

The GUI, in general will be extended step by step to allow visualization of the various methods executed by a job in form of an execution graph.



localhost:4040/jobs/3a9b9785-871a-4d77-8d23-39412ee022ba

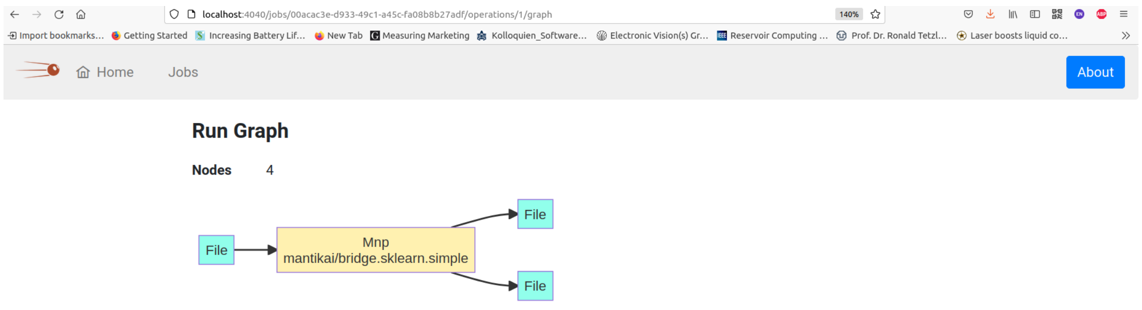
Job

ID	3a9b9785-871a-4d77-8d23-39412ee022ba
Name	Training
State	Running 19s
Registration Time	3/10/2022, 3:18:44 PM
Start Time	3/10/2022, 3:18:44 PM
End Time	

Operations

#	State	Type	Error
0	Done 7s	prepare_containers	
1	Done 0s	prepare_files	
2	Done 0s	other	
3	Running 12s	run_graph	
4	Pending	other	
5	Pending	other	

Figure 4: Image of the GUI panel for the job status. The current status is displayed.



localhost:4040/jobs/00acac3e-d933-49c1-a45c-fa08b8b27adf/operations/1/graph

Run Graph

Nodes 4

Figure 5: Image of the GUI panel for the the execution graph of a job. The example is a simple job with only 4 nodes.

3.4 Data input/output acceleration for W&C (Task 2.5)

To unify data input/output (I/O) throughout MAELSTROM, the first set of the benchmark datasets for the MAELSTROM domain applications which were published in

Deliverable 1.1 are already using the same Python package, CliMetLab[9], for data ingestion. CliMetLab aims to simplify the access and use of meteorological and climate datasets. Within the framework of CliMetLab the MAELSTROM applications have developed plugins which can be installed via the python package manger pip. Once installed, each of the datasets can be downloaded and loaded into memory with equivalent commands for each dataset. In doing this the MAELSTROM applications have standardised the access to each of the applications, lowering the barrier not only to use but also to adopting tools that accelerate I/O. This standardisation will greatly simplify the adoption of the data tools developed in MAELSTROM. As the definitions of the datasets are done within the CliMetLab plugins, the data access within the python code is trivial and done via a small number of lines of code with minimal boilerplate coding. For example, in Application 3:

```python
cmlds = cml.load_dataset("maelstrom-radiation",subset="tier-1")
dataset = cmlds.to_xarray()
```

If the benchmark datasets evolve, increasing in size or complexity, this can be captured with version control and dataset labels (e.g. the subset label used above). To minimise both local storage and data requests to the main dataset storage system, CliMetLab is in the process of implementing caching and mirroring of remote data in a local location. This would enable multiple users on the same system to access the dataset without duplication.

By the end of the MAELSTROM project, the application datasets will have developed and expanded to the point where most, if not all, of the MAELSTROM datasets will not fit into the system memory. In this case training and inference times can be limited by the speed at which examples can be read from disk. This would be highly undesirable, as it means computational resources will be under-utilised. The extent to which data loading will be the computational bottleneck will depend on a number of features including the hardware used, the computational demands on the hardware (e.g. number of concurrent users trying to load data on HPC systems), and crucially the data storage choices. This last point includes both the file format and the layout of the examples within the files themselves. The file choices are normally inherited from whichever system produced the training and inference data. For example, in Applications 3 and 5, the data originates from model forecasts. The MAELSTROM benchmark datasets cover a range of different applications, which cover different data access patterns. For optimal throughput of data, the examples should be written in compact chunks within the files. In the case of application 5, this is broadly true without any data manipulation. Coherent

[9]https://climetlab.readthedocs.io/

chunks of latitude and longitude for a small number of fields are the required data. For Application 3, which requires vertical columns of data for many atmospheric fields, these are natively not stored coherently.

The first of the benchmarking tasks, completed in a parallel deliverable, will begin to capture the differences across the applications, highlighting which of the applications have significant time spent in I/O, either as an initial step before training or streaming data during training. Once the benchmarking has been assessed, those applications with significant I/O overhead can be examined to highlight problematic elements. By using CliMetLab, assessing these applications will be a standardised procedure. Broadly there will be three aspects for optimisation of I/O. The first will capture the hardware resources being dedicated to reading data. The second will be software optimisation. The third will be data format and structure on disk. For the first part, Mantik will enable running and storing the configuration of experiments across hardware systems. This will allow the compute resources for I/O to be optimised. For the second part, our applications shared dependency on CliMetLab means optimisation of all applications can be carried out together. For example changing the use of the underlying xarray and Dask tools that are used under the hood of CliMetLab. Therefore, optimisations here will be easy to adopt across the applications and provide guidance to future creators of CliMetLab plugin datasets. For the third part of optimisation, rewriting and restructuring data, again CliMetLab helps to create a smooth path. CliMetLab plugins point towards a number of backend file types, e.g. NetCDF, GRIB, Zarr and TensorFlow records; with minimal changes required for the plugin developer and no pain for the plugin user. This enables the exploration of the file types and data chunking within those types, while making no changes to the wider coding environment for benchmarking. This has already been explored for Application 3 which has interfaces to both NetCDF TensorFlow records versions of the dataset. The former provides the user with complete metadata, in the latter the data has been restructured to bring all variables together in examples to be used for efficient training and inference.

This deliverable marks an intermediate step in the task to accelerate data input and output. While progress has been made there is still more work ahead to fully optimise I/O. Examining the benchmarking results will be a key next step in creating action points across our benchmark applications.

3.5 Deployment and infrastructure (Task 2.6)

With Deep500 extended to support weather and climate workloads (Section 3.2), we use Application 4 (A4) as a case study for deploying ML training. To that end, we created a recipe for one of the training workloads in A4 and benchmarked the process on different cluster infrastructures available to MAELSTROM. The results were fully reproduced across three systems with the same script.

We reproduce results for uncertainty quantification on ensemble prediction systems [4] with the MAELSTROM ENS10 dataset. The model used is a U-Net variant and contains specialized data augmentations for ensemble prediction. Below, we highlight elements of the recipe. The full recipe is available online[10].

```python
# Fixed Components
def fixed_components_gen(cfg: UQDataClass):
    return {
        "model": model,
        "model_kwargs": {
            "in_channels": len(cfg.parameters) * len(cfg.time_steps) * 2,
            "out_channels": 1,
        },
        "dataset": loader.CallableD500WeatherDataset(cfg),
        "epochs": cfg.epochs,
    }


# Mutable Components
def mutable_components_gen(cfg: UQDataClass):
    return {
        "batch_size": cfg.batch_size,
        "executor": executor(),
        "executor_kwargs": dict(device=d5.GPUDevice()),
        "train_sampler": loader.WeatherShuffleSampler,
        "train_sampler_kwargs": dict(seed=cfg.seed, args=cfg),
        "validation_sampler": loader.WeatherShuffleSampler,
        "validation_sampler_kwargs": dict(seed=cfg.seed, args=cfg),
        "optimizer": d5fw.AdamOptimizer,
        "optimizer_kwargs": dict(learning_rate=1e-2),
        "events": [
            RMSETerminalBarEvent(
                loader.CallableD500WeatherDataset(cfg).test_set,
                loader.WeatherShuffleSampler,
                batch_size=cfg.batch_size),
            CyclicLRScheduler(
                per_epoch=True,
```

[10] https://github.com/spcl/deep-weather/tree/master/Uncertainty_Quantification/Deep500_recipe

```
33                  base_lr=cfg.base_lr,
34                  max_lr=cfg.max_lr,
35                  step_size_up=(len(loader.CallableD500WeatherDataset(cfg)) //
36                                cfg.batch_size) // 2,
37                  step_size_down=None,
38                  mode="triangular2",
39                  gamma=1.0,
40                  scale_fn=None,
41                  scale_mode="cycle",
42                  cycle_momentum=False,
43              ),
44          ],
45      }
```

A Deep500 recipe is defined by three elements: **fixed** components (that will not change, see lines 1–11), **mutable** components (which may be tuned by different hyperparameter searches, lines 14–45), and **success metrics** (which would qualify a competitor to be ranked). Notice that this recipe is constructed from functions, which are customized by the `cfg` parameter. This parameter is defined by command-line arguments to allow for easy tuning on larger infrastructures, such as Mantik.

The fixed components in our case define the benchmark itself, which is the basic model, input/output channels, and the ENS10 dataset. In the mutable components, aspects such as the deep learning framework (`executor`), the batch size, optimizer, and hyperparameter tuners can be adapted and further tuned. The mutable components rely on our newly developed weather-specific modules for augmentation and training schedules, described in Section 3.2. Since the weather prediction workload is a regression problem, we do not define any success criteria as metrics, and thus every workload qualifies to appear in rankings.

Using our recipe, we ran training on local ETH compute resources, the JSC JUWELS Booster supercomputer, and the E4 cluster. The automatically-generated results by the recipe provide a breakdown of the training runtime, both per-epoch and per-batch. Table 1 shows our results on JSC-booster machine. We ran all experiments for six epochs with batch size 2 (same parameters as [4]).

Experiment Number	Total Training Time	Average Time per Epoch	Average Time per Iteration	Final Validation Loss	GPU energy consumption
1	4,451s	740s	1.14s	0.121	157.03Wh
2	5,062s	847s	1.08s	0.123	157.61Wh
3	5,021s	836s	1.07s	0.123	135.66Wh

Table 1: Training results of Deep500 recipe on JUWELS-Booster supercomputer.

4 Conclusion

The current deliverable describes the development status of the W&C workflow tools. Whereas the development of the core component - the full MAELSTROM workflow tool and protocol - has not reached alpha stage as planned, the reasons and obstacles encountered are mentioned above.

We can recognize, however, that the major problems can be solved, and a way how to solve them is identified. At the current stage, we are confident to reach usage within MAELSTROM in the following months.

The status will be presented on the PASC conference, and the feedback will be used to reach beta stage for distribution to the wider community during the summer of 2022.

References

[1] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 66–77, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[2] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4), 2004.

[3] Leslie N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

[4] Peter Grönquist, Chengyuan Yao, Tal Ben-Nun, Nikoli Dryden, Peter Dueben, Shigang Li, and Torsten Hoefler. Deep learning for post-processing ensemble weather forecasts. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194):20200092, 2021.

Document History

Version	Author(s)	Date	Changes
0.1	Markus Abel, Fabian Emmerich, Greta Denisenko (4cast)	11/03/2022	Chapters 3.1 and 3.3
0.2	Saleh Ashkboos, Tal Ben-Nun (ETH)	12/03/2022	Chapters 3.2 and 3.5
0.3	Mathhew Chantry (ECMWF)	14/03/2022	Chapter 3.4
0.4	Fabian Emmerich (4cast)	16/03/2022	Chapter 2.2
1.0	Markus Abel, Fabian Emmerich, Greta Denisenko (4cast)	30/03/2022	Final version

Internal Review History

Internal Reviewers	Date	Comments
Tal Ben-Nun (ETH)	20/03/2022	Review passed with minor edits
Daniele Gregori (E4)	29/03/2022	Review passed with minor edits

Estimated Effort Contribution per Partner

Partner	Effort
4cast	2 PM
ETH	1 PM
ECMWF	0.5 PM
Total	3.5 PM