MAchinE Learning for Scalable meTeoROlogy and climate

Report on the survey of the workflow, the MAELSTROM protocol and ML requirements

Markus Abel, Fabian Emmerich & Greta Denisenko

www.maelstrom-eurohpc.eu

D2.1 Report on the survey of the workflow, the MAELSTROM protocol and ML requirements

Author(s): Fabian Emmerich (4cast)
Markus Abel (4cast)
Greta Denisenko (4cast)

Dissemination Level: Public

Date: 30/09/2021
Version: 1.0
Contractual Delivery Date: 30/09/2021
Work Package/ Task: WP2/ T2.1
Document Owner: 4cast
Contributors: ETH
Status: Final

MAELSTROM

Machine Learning for Scalable Meteorology and Climate

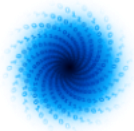
Research and Innovation Action (RIA)
H2020-JTI-EuroHPC-2019-1: Towards Extreme Scale Technologies and Applications

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

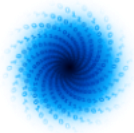
Published by the MAELSTROM Consortium

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

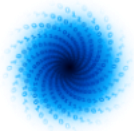
The MAELSTROM project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955513. The JU receives support from the European Union's Horizon 2020 research and innovation programme and United Kingdom, Germany, Italy, Luxembourg, Switzerland,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



Contents

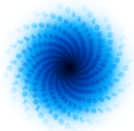


Figures



1 Executive Summary

The architecture of a MAELSTROM platform needs to respect requirements from i) AI tasks, ii) the weather and climate workflow, iii) and from the use cases, i.e., the concrete applications defined in the MAELSTROM work packages. We find that the design of a platform dedicated for AI needs to have a generic form which reflects AI tasks. In addition, a protocol can be designed which is tailor-made such that the enormous amounts of data that are available for weather and climate applications, and modern HPC infrastructures are accessible. This involves the flexibility of the protocol towards parallel data loading and/or storage. Eventually, the requirements collected for the different tasks A1-A6 were taken into account when the architecture and protocol were designed. In the current state, architecture and protocol are focused on a minimal viable product, such that W&C jobs can be run on the Jülich infrastructure.



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 the needs of weather and climate applications.

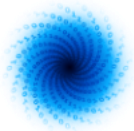
The MAELSTROM software framework will enable scientists to apply and compare machine learning tools and libraries efficiently across a wide range of computer systems. A user interface will link application developers with compute system designers, and automated benchmarking and error detection of machine 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 blueprints 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 2.1 is one of four MAELSTROM deliverables that survey the state-of-the-art in terms of methods, tools and developments in machine learning at the beginning of the project and aim to build additional links between the three work packages that are involved in the MAELSTROM co-design cycle. Deliverable 1.2 is a survey of machine learning methods and tools that are currently used for weather and climate applications. Deliverable 2.1 contains a survey of existing machine learning workflow tools and a summary of the MAELSTROM protocol and machine learning requirements. Deliverables 3.1 and 3.2 provide a systematic analysis of the hardware requirements for the MAELSTROM applications and a roadmap analysis of hardware that will be relevant for machine learning in MAELSTROM.

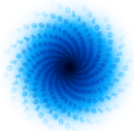


2.2.2 Work performed in this deliverable

The weather and climate (W&C) workflow is one of the main concepts to be defined and clarified in this WP and used in the applications. Typical tasks have been identified and described in Sec. 3. The technical concepts of an engine that enables performant execution of the workflow tasks needs to be defined together with its components. Eventually, the specific tasks are mapped to the architecture such that an efficient execution is possible. This is enabled by communication between components which follows a protocol described in detail in Sec. 4. The top-level description of the MAELSTROM engine is given in a domain specific language, which is also explained in Sec. 4. In Sec. 5, the requirements collected from the different applications are summarized and briefly discussed with respect to the implications on the MAELSTROM engine. The biggest space is given to the protocol, as it forms the basis for all subsequent tasks and is developed up to a state that allows implementation.

2.2.3 Deviations and counter measures

There are no significant deviations from the planned contributions of the deliverable.



3 Survey of the workflow

3.1 Typical ML workflow

A typical ML workflow consists of several steps: data loading, preparation, model training and prediction by the model. Since these steps in the workflow are conceptually independent from one another, the optimal workflow separates each step from its previous and subsequent neighbour and, as a result, ensures efficiency, scalability and reproducibility, and reduces complexity. While this appears to be simple when described at this level of abstraction, each task in the ML workflow has its own complexity that can make the work on each component time-consuming. While the preparation of data (also called *preprocessing*) is often the most expensive task for scientists, the data loading especially for large data sets consumes a substantial time when executing a ML workflow. However, each step of the workflow has its own problems that need to be optimized.

3.1.1 Data loading

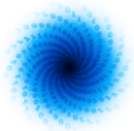
At the beginning of a typical workflow, raw data have to be loaded to perform a preprocessing that enables training a model on them. Aiming for usage of data at petascale, this step is a challenge in the flow. Usually, data are stored as files and located on physical storage either on a local machine or a data cluster. The data have to be loaded into the local memory of the machine where the preprocessing is performed.

3.1.2 Preprocessing

Preprocessing is very time-consuming and can include a large amount of data transformations to arrive at the desired result. Typically, the data are cleaned, clipped, normalized, and/or outliers are removed, and so forth. Especially due to a large amount of noise in data sources, this step can be very exhausting. The data might also be extended with new information derived from the original data. Preprocessing procedures are usually very data-specific, i.e. a certain type of data always requires the same preprocessing to be fed to a certain type of ML model. As a consequence, these techniques must be easily reproducible and extendable. Furthermore, different data manipulation steps also require different frameworks. Additionally, when working with large data sets, the performance of the data processing might also be a bottleneck.

3.1.3 Model training

After preprocessing of the data, they can be used with the specific ML solutions they were designed for. The data are used to train a specific model that employs certain statistical and/or numerical methods. Once a model is trained, its quality can be elaborated using different kinds of metrics. Since the quality of a resulting model can hugely vary on input such as the preprocessed data (also called *features*) and model parameters, it is essential to be able to easily re-train a model with different settings to achieve the best possible quality. Moreover, the duration of the model training process can increase drastically with data size and model complexity.



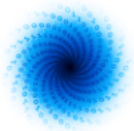
3.1.4 Prediction

The actual aim of a ML workflow is to develop a model to predict a certain behavior or trend from a sample of data. Here, the model quality impacts the accuracy of the predictions. The data that are fed to the model for prediction have to be of the same shape as the data that were used to train it. It is required to use the exact same workflow (data loading and preprocessing) as for training. Hence, it is indispensable that the part of a workflow that retrieved and processed the data of a certain model is easily and with the smallest possible effort reproducible.

3.2 Addressing W&C-specific difficulties

Compared to other domains, W&C ML applications reveal certain problems that dramatically hinder their workflow. Hence, it is essential to address these bottlenecks and, as a result, improve the speed and quality of W&C predictions. Especially the W&C domain uses large data sets up to petascale that slow down the workflow steps due to their mere size.

- **Data loading:** W&C data can possess grids covering the whole globe or regions, e.g. Europe, resulting in large data sizes which slow down the data loading. Thus, the loading process has to be optimized by parallelisation: Due to limited memory sizes, the data should be split into smaller chunks and processed independently on different machines in parallel when working with tera- or petascale data sets. Furthermore, individual data samples (e.g. global maps) are often larger when compared to data samples from other domains resulting in memory limitations that, again, call for parallelisation and the use of smaller chunks.
- **Data formats:** The W&C community uses different data formats than other ML domains (NetCDF, GRIB). These have a different shape compared to conventional data formats and, hence, must be processed in a different manner. However, this can be eliminated by separating data loading from preprocessing and the transformation of the data into a universal data format.
- **Preprocessing:** The duration of data manipulation steps scales with the amount of data and are, thus, very time-consuming in W&C ML workflows. Some common Python frameworks such as pandas especially lack performance when operating on huge datasets. As a consequence, it must be ensured that the preprocessing of data allows usage of the most efficient techniques. Besides using high-performance frameworks such as numpy or xarray, this may also include organizing data using different programming languages. In particular, standard tasks, e.g. inference using a trained model, are realized by compiled Go programs, thus minimizing container size and enhancing execution speed.
- **Model training:** Certain model architectures reach high computational costs when executed on large data sets. With increasing data size, the time for creation of a model increases as well, which is one of the bottlenecks in the W&C ML workflow. This can be addressed by training models on distributed systems that possess a great computational performance, i.e. HPC clusters.
- **Model prediction:** Models in the weather domain are trained often on processed data, like reanalysis data from era5, however when a model is used “in production” current NWP data are used for prediction. For climate models, this does not hold in general.



3.3 Enhancing the W&C ML workflow

3.3.1 Isolating workflow steps

The steps of a ML workflow need not be strictly coupled necessarily. Additionally, scientists may want to use different frameworks such as PyTorch, TensorFlow, scikit-learn, etc., to produce their ML models. This, though, brings complications: the frameworks might be incompatible with each other or HPC systems may not even support using them. This is solved by using isolated environments (i.e. Docker and/or Singularity containers) to run the individual steps of a ML workflow.

3.3.2 Storing workflows

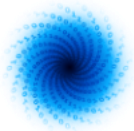
The training of a model is a development cycle. To produce high-quality models, it is necessary to be able to easily re-train a model with slightly tweaked input. Hence, it is essential for scientists to be able to store and reproduce their results any time. This option can be provided by ML tools that allow defining and storing ML pipelines which describe the data loading, preprocessing, model training and prediction using certain data sets.

3.3.3 Supporting shared workflow

It is essential to science that scientists share their knowledge and techniques to achieve the best advancement. This, of course, also applies to the ML domain. Hence, it must be possible for scientists to easily share their ML workflows with the community. Preprocessing of data requires the best domain knowledge when aiming for the optimal models. Scientists can only enhance their models by sharing and discussing their procedures and results with other domain experts. This is achieved by providing tools and platforms to create, store, and execute workflows dedicated for specific data sets. As a consequence, their workflows become transparent and comprehensible for other scientists and can be improved and shared across the community.

3.3.4 Further use of stored workflows

Once workflows are stored to a data base, it is possible to use their outcome as information for other scientists. This can be realized by recommender-tools like Deep500. The integration of such a recommender has not yet been realized, it is under development.



4 Survey of the MAELSTROM protocol

The MAELSTROM protocol reflects the W&C workflow as a specialization of a general ML workflow. Whereas in the domain language one speaks of the workflow, for an implementation the *pipeline* concept is used and consequently, this nomenclature is often used in the following. One can understand both terms as synonyms in most situations. The architecture of a system realizing the workflow and implementing the protocol is shown in Fig. 1.

The protocol defines the rules, syntax and semantics of communication between components of a system, possibly together with errors. The so-called interfaces between components are described by the protocol, in particular how messages and data are exchanged. The current status of the protocol is described in this document. Future enhancements are indicated, where necessary.

4.1 The Backend - Mantik engine and alternatives

The goal of a platform for ML in W&C is reflected by the platform requirements described in Sec. 5. At the start of the Maelstrom project, at one partner - 4Cast - a Python framework was developed for applications like A6 (ML for the prediction of energy production by renewables). To enhance speed and stability, a type-safe engine was implemented (using the Scala programming language) that orchestrates ML jobs in a microservice architecture. That implies the use of containers, in particular Docker (docs.docker.com) and Singularity for HPC (sylabs.io/docs). Further, the pipeline concept for ML is enforced by offering the typical ML steps: data load, feature engineering, training, deployment, prediction, as explained below by a simple example.

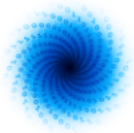
In the meanwhile, other platforms have realized similar concepts. This is summarized as well in Sec. 5. Consequently, it is important to develop a description of a system that realizes the workflow for ML with additional specific features for W&C. This description is formed by the protocol. It essentially allows declaration of the elements of a W&C workflow which then are executed using an engine. Currently, in Maelstrom, the Mantik engine is used, but we want to point out that a replacement by another engine will be possible, as long as the protocol is implemented.

4.2 Example - MNIST

MNIST - i.e. image classification on a handwritten digits dataset - is a widely used introductory example in ML tutorials. As such we use it here to illustrate how to work with Mantik.

The typical development flow is shown here: Declaration of a MantikItem (first listing), implementation of the model interface given by Mantik (second listing), upload of the Item (third listing) and workflow definition in an M-File (fourth listing). The files for two first steps are typically provided by more experienced users from the Mantik community or the Mantik core developers, such that they are available as service for other users and usable without modification. In the third and even more in the fourth step, the actual ML experiment is defined, it is the main part of the regular work being done by any researcher using Mantik.

The model needs to be set up by declaring a method and the data. The item is internally represented by its existing implementation, the bridge, this is a minor implementation detail.

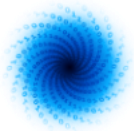


To declare the item, a Mantik header is written, here, we comment after the # sign.

```
name: mnist_linear                              # name of the Mantik Item
bridge: mantik/tf.train                         # service abstracting tensorflow
trainedBridge: mantik/tf.saved_model            # service optimized for inference (compiled Go)

kind: trainable                                          # the type of the item

metaVariables:                                  # list of metavariables. It is detailed in
    - name: batch_size                          # the documentation,
      type: int32
      value: 128
    -
      …

trainingType:                                   # type needed for training this particular
    columns:                                    # method
        image:
            type: tensor
            shape: ["${height}","${width}"]  # metavariables for more dynamic typisation
            componentType: float32
        label: int32

statType:                                       # if training requires: number of epochs
    columns:                                    # and loss factor. This does not appear,
        epoch: int32                            # e.g. for data load
        loss: float32

type:                                           # input type. Must be compatible to the
    input:                                      # tensorflow service. Refers to
        columns:                                # inference (saved model part above).
            image:
                type: tensor
                shape: ["${height}","${width}"]
                componentType: float32
    output:                                     # output type. Must be known to
        columns:                                # Mantik
            label: uint8
            logits:
              type: tensor
              shape: [10]
              componentType: float32
```

The *payload* needs to be sent to the service, once the Mantik header is written. As an example, here is the code for MNIST application:

```
def train(request: TensorFlowTrainRequest, context: TensorFlowContext):
    train_dataset = request.train_dataset()

    # Meta Variables
    batch_size = context.mantikheader.meta_variables.get("batch_size", 128)
    n_epochs = context.mantikheader.meta_variables.get("n_epochs", 5)
    learning_rate = context.mantikheader.meta_variables.get("learning_rate", 0.01)
    width = context.mantikheader.meta_variables.get("width", 28)
    height = context.mantikheader.meta_variables.get("height", 28)

    stats = []
    batches = train_dataset.batch(batch_size)
    iterator = batches.make_initializable_iterator()

    data_x, data_y = iterator.get_next()

    # Model setup
    model = Model(data_x, data_y, learning_rate, width, height)

    sess = context.session
```



```
    sess.run(tf.global_variables_initializer())
    sess.run(tf.local_variables_initializer())
    sess.run(iterator.initializer)

    for epoch in range(n_epochs):
        sess.run(iterator.initializer)

# Training
        try:
            while True:
                _, current_loss = sess.run([model.optimizer, model.loss])
        except tf.errors.OutOfRangeError:
            pass

        print("Epoch ", epoch, " of ", n_epochs, " loss=", current_loss)
        stats.append([epoch, current_loss.item()])

    # Calculating Accuracy
    sess.run(iterator.initializer)
    try:
        while True:
            sess.run([model.accuracy_op])
    except tf.errors.OutOfRangeError:
        pass
    accuracy = sess.run(model.accuracy)

    # Model export
    dir = "trained_model"
    model.export(context.session, dir)
    request.finish_training(Bundle(value=stats), dir)
```

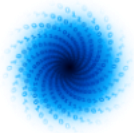
One recognizes the method *train* . In it, model setup, training and model export are called. In terms of software development, Items define an interface for trainable models, dataset items etc. that an Item developer needs to implement. Within these interface implementations, the user is free to implement what they need, however, input and output must fit the declaration of the header file. Why that? In a pipeline, e.g. output of feature engineering must fit input to training. This is automatically checked by the engine and helps bookkeeping the development of proper applications. Mantik has some built- in adapters that convert data to Mantik format and transform between two compatible types. These adapters will be extended by the Mantik core developers as well as the community.

The above defined Item can then be added to the Item database with a Mantik Client:

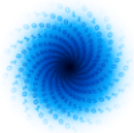
```
import mantik.engine
with mantik.engine.Client("localhost", 8087) as client:
        mnist_item = client.add_artifact(
               <path_to_MantikHeader>,
               named_mantik_id = "<user>/mnist_linear"
               )
```

This is the protocol concerning items.

Eventually, the workflow shall be abstracted completely and the user should not bother with details inside an item - once many items are implemented, for most of the frameworks an implementation exists which can be used. Then, the DSL used in the Mantikfile comes into play: Typical actions are abstracted away and a file with declaration of input, pipeline steps, and output that shall serve the setup of a ML pipeline. A draft which is found in many other ML tools is:



```
"""
Mantikfile
This snippet shows all configuration that is not directly related to pipeline definition.
"""
# Initialize tasks as instance of `MantikItem`, pass additional configuration

get_data = MantikItem(name="mantikai/binary:v1").configure(
    params={"file": "<filename>"}
)

# Define shorthand for output names, valid globally

images, labels = get_data.get_output_reference()

pre_process = MantikItem(name="<user>/pre-processor")

# mark `batch_size` as hyperparameter, to be used below

batch_size = HyperParameter()

# Use the item that was specified above
train = MantikItem(name="<user>/mnist_linear:v1").configure(
    params={"batch_size": batch_size}
)

# Common tasks will be builtins, especially evaluation, SQL like operations, timelag

evaluate = evaluate(metric="rmse")


# Aggregate multiple steps so that both can be used in hyperparameter search

train_and_evaluate = pipeline_component(train, evaluate)


# Set input names so that they can be referenced easily in the pipeline definition

train_and_evaluate.train.set_input_name("data_train")
train_and_evaluate.evaluate.set_input_name("data_test")


# Define hyperparameter search

hyperparameter_search = HyperParameterSearch(
    train_and_evaluate,
    hyperparameters={"batch_size": [64, 128, 256]},
    metric=train_and_evaluate.evaluate.metric,
)


# Deployment (here: save model to a database) can be defined via builtin functions

deploy = save_model()


# Inputs are set in the pipeline so that Items can be reused in multiple pipelines
# Configuration can also be done in the pipeline definition; this is meant to aide the
development of more complex pipelines with repeated use of mantik items

batch_pipeline = MantikPipeline(
    stages={
        "stage1": [
            get_data,
            pre_process.set_input(images),
            dsl.builtins.join(
            inputs=[pre_process.output, labels], reference="join1", how="inner"
            ), # reference arg is for referencing this particular join in later pipeline
steps; join has pandas like arguments
        ],
```



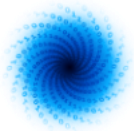
```
        "stage2": [
            dsl.builtins.train_test_split(data=[join1.output], reference="split"),
            hyperparameter_search.set_input(
            data_train=split.output.train, data_test=split.output.test
            ),
        ],
    "stage3": [deploy.set_input(hyperparameter_search.best_model)],
    }
)


# Create a new MantikItem for inference (reference the output of the deploy step)

predict = makeMantikItem(batch_pipeline.stage3.deploy.best_model)


# Define an inference pipeline

predict_pipeline = MantikPipeline(
    stages={
        "stage1": [pre_process.set_input(images)], # Images is defined above to reference
(part of) the output of get_data
        "stage2": [predict.set_input(pre_process.output)],
    },
)

############################################################

# Time lag, used as a builtin function directly in the pipeline
# Suppose get_data returns a timeline; can be imported from somewhere else

dummy_pipeline = MantikPipeline(
    stages={
        "stage1": [
            get_timeline,
            timelag(
                timeline=get_data.output,
                lag_column="time",
                lag="1h",
                reference="timelag1",
            ),
        ]
    }
)
 This example is a draft for the implementation. It becomes clear that a lot of boilerplate
code which is needed for a production system is written in the background.
```

4.3 Items and their Interfaces

The items used by MAELSTROM must realize the W&C workflow described above. In particular, HPC capabilities need to be realized by the abstract description allowed for an item. To be as specific as needed and as general as possible the items realize typical parts of a workflow. Items are consequently specified in the header which declares the item function and provides necessary (software-) infrastructure.

An Item has a container, a payload (user-supplied code to be used), and its header. Clearly, this is programming-language agnostic and depending on the language the concrete interface definition may vary. The most important notation concerns the *kind* of the item. It is (currently) restricted to be one of

dataset, algorithm, trainable, deployment



With these basic item types the W&C workflow can be realized, and at the same time, usage beyond these tasks is not possible. On the most basic level, the Item kind just defines the number of inputs and outputs of said Item. Together with built-in combiners (aggregate multiple inputs to one output or the other way round), a general directed acyclic graph can be expressed in terms of these kinds. Of course, the concept is extensible and new types or subtypes can be added, if needed.

4.4 M-File

At the top level stands the MAELSTROM-*File,* or *Mantikfile, hence M-File.* It is written in a descriptive language which aims to form a so-called Domain-Specific-Language, DSL. We use python-like syntax to ease understanding for the majority of the data scientists and allow for IDE features such as syntax highlighting and code formatting. The M-File serves two purposes:

- The user can develop new workflows or inspect and adapt existing ones (e.g. from the algorithm database).
- The M-File is the single source of truth for the workflow definition and thus guarantees reproducibility. It can easily be version-controlled, e.g. in git.

In the Mantik DSL, the basic engine API is exposed as built-in objects, methods and functions. The syntax allows the initialization of the built-in objects and the call to these objects' methods. All other python syntax (function and class definitions, decorators ...) is explicitly discarded.

Semantically, the ontology of the Mantik engine can be expressed in the Mantik DSL: The main objects are Items and Pipelines (collections of Items with an explicit definition of execution order and data flow). It is assumed that the Items are available in a database and can be referenced by ID or name. Methods for adding Items to this database are available in the Client.

For highly parallelizable hyperparameter optimization we offer the `Hyperparameter` class. An instance of said class can be used for MantikItem configuration and then be varied on engine level, i.e. executed automatically in parallel in different MantikItem instances.

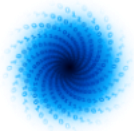
We are currently working on the implementation of built-ins for:

- train test split,
- time lag coordinates,
- model deployment.

4.4.1 DSL specification

The domain specific language is based on the following principles:

- The DSL uses RPython syntax, no "power features" such as functions or decorators.
- There are two fundamental objects: Item and Pipeline.
- Tasks (single execution steps) are initialized as Item objects; configuration can be passed.
- Tasks can be grouped for convenience.
- Typisation is not needed since the information is already included in the underlying Mantik headers. Data transformations are inferred by the engine if necessary.



- Tasks are referenced by the object name, optionally a reference keyword can be passed.
- We provide built-ins for set operations, train test split, time lag and basic evaluation as well as model deployment.
- Pipelines are initialized as Pipeline objects.
- Tasks in a pipeline are grouped in stages; stages can be used for partial execution.
- Task dependencies are declared with `task_b.set_input(task_a.output)` inside the pipeline.
- Hyperparameters can be initialized as `param = Hyperparameter` and used in MantikItem configurations.
- Hyperparameter search is initialized as `hs = HyperParameterSearch(...)`. It can be used in the pipeline like any other task.

4.5 Architecture

Here, the architecture of the system shall be explained. The architecture is microservice-oriented and under development. The workflow has been described above in Sec. 3, here we describe how this is reflected in the components that build the system, cf. Fig. 1. If the user has developed some ML code, this code shall be executed. This is handled by the Mantik engine which plays the counterpart of an operating scheduler in that it takes tasks and executes them in a certain order. The whole architecture is service-based using the Mantik Items, which are container-services. These items serve functionality necessary to execute tasks and can be connected such that execution is most performant. The tasks are sent to the engine by user-side clients. In these clients, top-level code is stored and the corresponding tasks are sent. Since Python is the dominant language for ML, Python code is supported best e.g. by a Jupyter notebook client. However, a developer is free to implement code in any language as long as the services connect with the right input and output types, declared in the protocol. Said this, it has to be made clear that introduction of a new programming language may involve the development of basic containers that speak the Maelstrom protocol. Of course, items with different languages may be mixed.

By the above, implicitly a role concept is introduced: one role is the core developer which helps to develop the backend code for Mantik, e.g. by implementing new services and frameworks in terms of items that are ready to be used, or by enhancing the engine in various ways. This typically requires library programming skills and deeper understanding of containers as microservices. The second role is the community developer who implements custom models or data loading and processing tasks to be executed in Mantik Itemsthe ML user which develops ML applications, which requires know-how in data science and domain knowledge. Thirdly, the W&C user applies available tasks implemented by the community with some straightforward configuration options and mix and match those to a workflow as defined in the M-File (see below).

Eventually, the orchestration of many tasks is managed by the engine.

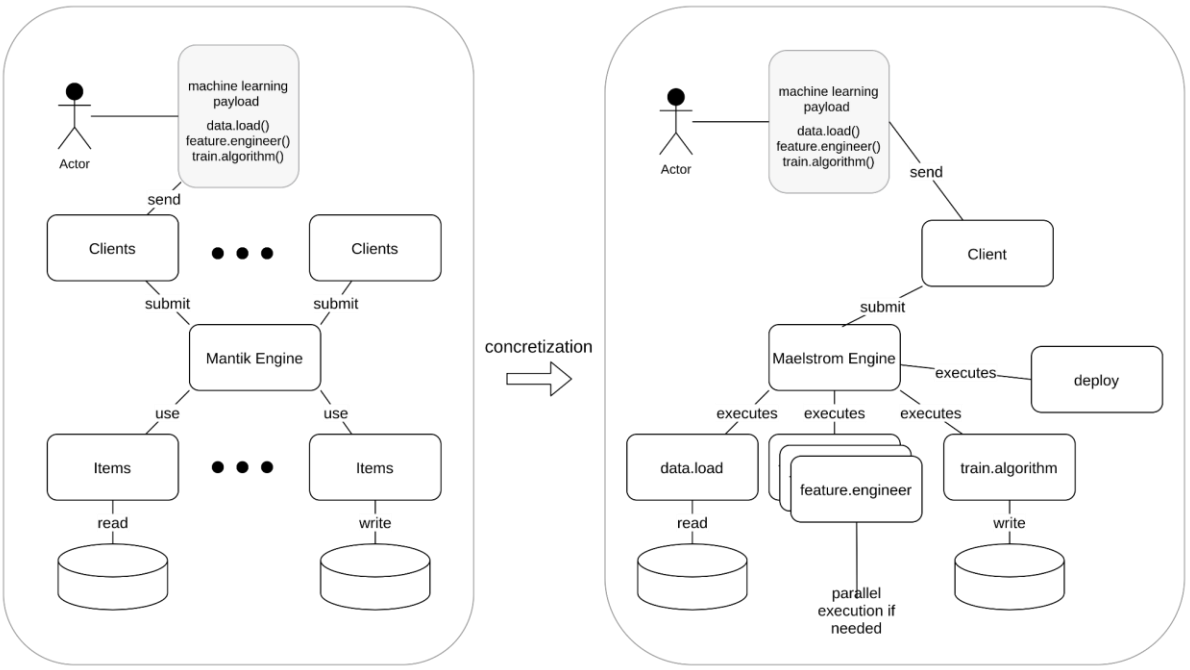
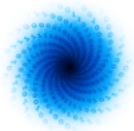


Figure 1: Left: Architecture concept. The engine orchestrates the workflow sent by the ML users (top left actor).The users develop ML code in the language of their choice, at the moment this will be 99% Python code. They send this code as payload to the engine which infers actions on how to execute the code using the protocol. The execution uses "items" - container services that run on the target system. Such containers use infrastructure to read and write data. So far, any pipeline could be executed, as long as it is known to the engine. Right: More concrete view on the components: The realization of Mantik engine in Maelstrom -the Maelstrom Engine- uses a specific implementation, dedicated to HPC execution and the usage of huge data. The items are specified according to the ML flow adapted to W&C. In the figure, only basic workflow steps are indicated. Parallel execution of tasks should be organized by the engine, if possible. It is indicated as multiple containers in the feature engineering step. The concept is flexible enough to be modified during Maelstrom such that more involved services can be added or unsuitable ones can be removed and replaced by the ML engine.

The MAELSTROM protocol now specializes the underlying protocol for general machine learning in that certain *types* are declared - and will be sharpened during further development - which contain exactly the needed functionality, and not more. This restricts the possible use cases, and on the other hand offers some clarity for ML users, since one can only use functionality needed for the concrete W&C application. It shall be noted that, in principle, the development of a script that contains all parts of a workflow at once, is possible and not forbidden. Then, however, the user is deprived of the functionality that will be implemented in the specialized services (e.g. automated detection of parallelization options, cleaning routines, etc.). It shall be noted, too, that the development of the specialized components is subject to community contributions.

From top layer to lowest layer, we have a hierarchy of linked declarations and communication protocols. In the following subsection this rather abstract view is illustrated.



4.6 MNP - Mantik Node Protocol

The communication between components, in particular between engine to Items and Item to Item, is ruled by the MNP (mantik node protocol). Consequently, an Item container has to implement the MNP to enable communication between the services. This protocol builds on protocol buffers (the protobuf library, cf. https://en.wikipedia.org/wiki/Protocol_Buffers), designed for serialization and used mainly for communication and storage of data - properties needed for the items. A full documentation is given in the documentation of the Mantik core (http://mantik.pages.ambrosys.de/core/Mnp.html). In the following, we denote any component as node, which is the abstract representation used by the engine.

It is designed to solve the following requirements.

- Algorithm nodes can transform data. It's not necessary to receive all data in order to start a response.
- Running Items can be reconfigured.
- Only one container is needed for an Item.
- There must be a concept of a single task.
- The lifecycle of an Item can be controlled from the outside.
- The protocol can be tunneled through http proxies.
- Streaming of data, and thus asynchronous computation, is supported.

4.6.1 How MNP works

MNP is meant to be a transport protocol for nodes. It is not specific for W&C flows. Processes that follow the MNP have the following properties:

- A session is initiated by the engine.
- Each session has a set of input and output ports.
- Within a session, data are pushed to input ports and responses read from output ports.
- During session init, it is possible to "wire" output ports to other processes' input ports. Data generated on these output nodes will be automatically forwarded.

By default, a MNP server begins listening / waiting for a session. Once an init-session command is received, a session is initialized. The init-session call must define the number of expected input and output ports. Mantik adds some Init-Configuration (MantikHeader for bridges and URL of payload). If the Init-Session succeeds, a new session is opened, and the corresponding node waits for input data.

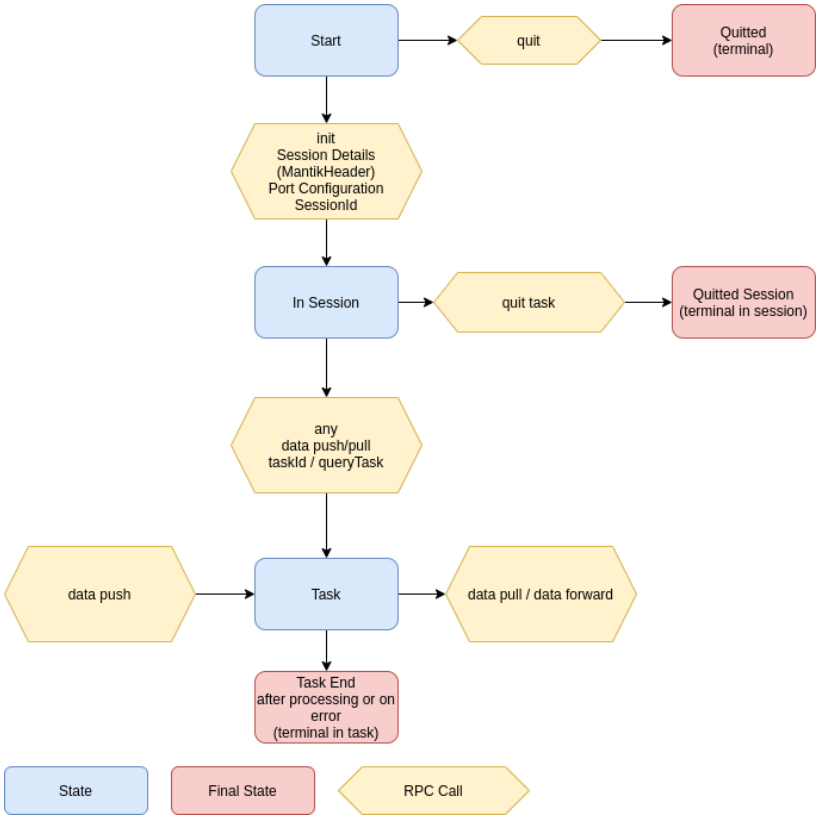


Figure 2: Session description and usage of the MNP
The engine handles sessions as follows: During startup (top tile) the session is initialised. Within the session, any data push or pull, or query Task command for datasets, is created and run. When all tasks are finished, the session is closed.

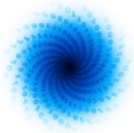
The init-session call may contain forwarding information, such that tasks forward their output to other nodes, saving data roundtrips.

Data are transferred using push-Calls and received using pull-Calls. All push and pull calls contain a task ID. The first pull/push with an unknown taskId creates a Task in which data processing takes place. The Task lives as long as processing is done.

It's also possible to create a task with a Query-Task command. This is necessary for Nodes which only create data and forward it.

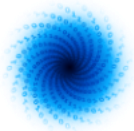
The protocol is parallel and asynchronous:

- Any number of sessions can be created in parallel.
- Any number of tasks can be created in parallel inside a session.
- Session Creation may take some time, this is reflected by a stream-response.
- Push and Pull both use streamed asynchronous data.



The commands of the protocol are about, init, quit, quitSession, push, pull, queryTask which realize the above possibilities.

The MNP defines the low-level communication between engine and Items. It is implemented by the Mantik core developers. For Item development higher level interfaces have been derived (see below); the average user does not get into direct contact with the protocol nor the handling of sessions and tasks shown in figure 2.



5 Survey of machine learning requirements

In this chapter, we briefly summarize the requirements for a ML architecture and their realization.

In the last few years many platforms have emerged, often from scratch. Big cloud providers have discovered AI as a potential opportunity and have developed platforms like Sagemaker (AWS), Google's AI platform with well described AI workflow, IBMs Watson, MLFlow, and many others of similar concept and realization. A detailed discussion of each of the existing solutions would exceed the scope of this document by far, and a publication on a comparison of the underlying architecture or the corresponding requirements has not been found and would likely be outdated already after a couple of months. Of course, this is also caused by the protection of proprietary codes.

There are, however, blogs or explanatory articles on the platforms themselves and some journal issues where requirements for ML platforms are given [1][2][3]. A nice overview is given in [4], where requirements are explicitly described. A recent arXiv publication explicitly presents the different concerns and perspectives of Stakeholders [5]. Basically, all systems reflect the key requirements of users:

1. Easy access for the specialized community, typically through a python/jupyter notebook
2. Automated pipeline building
3. Flexibility in package usage (no vendor lock-in)
4. Reproducibility of the model results
5. Parallelizable
6. Batch training and live prediction go hand in hand
7. Easy and controllable configuration of models

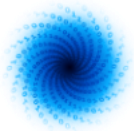
The commercial platforms typically offer workflows and corresponding architectures for a dedicated user audience. None of the platforms is dedicated to the W&C flow with extraordinary data volumes and highly performant hardware with restricted access.

From the MAELSTROM users, a survey has been performed to understand the specific needs with respect to the functionality the W&C workflow requires in addition to the above.
The results can be summed up as follows:

Requirements with respect to model development: Jupyter notebook is favored.

Requirements on versioning ML models were inhomogeneous, such that a first solution consists of storage of the complete environment

Requirements on model handling (chaining, retraining) resulted in a vote for the offered possibilities. Interestingly, parallelization was required to be handled by a platform.



Requirements on the workflow show that any of Data cleaning, Validation/Testing, Hyper-parameter - optimization, Feature engineering, Execution is needed. The typical workflow coincided with the description in Sec. 3.

Requirements on data were quite diverse, however for W&C, clearly, two formats are used (NetCDF and GRIB) and a platform may support those formats optimally.

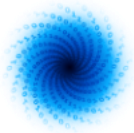
5.1 ML frameworks and software

In the following, a short description of each of the well-established software and tools that the participants primarily rely on in this project is given.

- CliMetLab (https://climetlab.readthedocs.io/en/latest/) is a python package that is used to manage the downloading and loading of climate and meteorological data, for a variety of datasets, dubbed plugins. In the MAELSTROM project, CliMetLab plugins have been created for each of the six applications. Users can access the benchmark set within minimal lines of code.
- Jupyter notebook (https://jupyter.org/) is an open source web application. Jupyter notebooks that can be used for sharing code, equations, visualization and text. In the MAELSTROM project, Jupyter notebooks have been created to explore the benchmark datasets and demonstrate simple machine learning solutions.
- TensorFlow (https://www.tensorflow.org/) is a well-known open-source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs).
- PyTorch (https://pytorch.org/) is an open-source ML library, used for applications such as computer vision and NLP.
- Tensor Core can accelerate large matrix operations, which are the heart of DL.
- CUDA (https://developer.nvidia.com/cuda-zone) is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).
- xarray (http://xarray.pydata.org/en/stable/) is an open-source project and Python package that makes working with labelled multi-dimensional arrays simple, efficient.
- Horovod (https://github.com/horovod/horovod) is an open-source distributed training framework for TensorFlow, Keras, PyTorch, and MXNet.

5.2 Distribution of data and executables

The massive datasets in W&C applications become the main bottleneck to train the model within a reasonable time. Moreover, more advanced and complex architectures have been continuously developed and the number of trainable parameters have been increased significantly to achieve promising accuracy. The current hardware hardly satisfies the memory requirements of such large DL models. In this context, distributed training for deep learning models is necessary. Since it is essential for W&C applications, it is summarized here bya short introduction to the two parallelism strategies – Data parallelism and model parallelism that are explored in the MAELSTROM project.



Data parallelization: In data parallelization strategy as shown in Figure 3, the data are split into N subsets, which corresponds to the number of GPUs. The model is copied onto each GPU, and trained on each subset to obtain the gradients. The gradients on all the GPUs are accumulated and averaged to update the model parameters in the next training iteration.

Model parallelization: Model parallelization strategy splits the model into different parts. The same minibatch samples are copied to all GPUs. Each part of the neural network is trained on the same samples. In this case, the models are not stored in one place, which can effectively conserve memory [6].

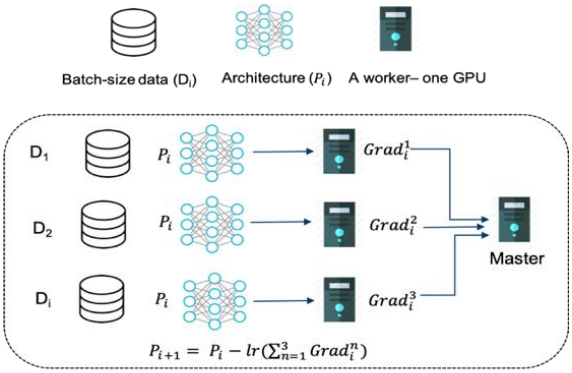
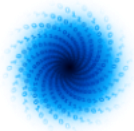


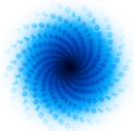
Figure 3: The illustration of data parallelization strategy.



6 Conclusion

In this document, we discussed the state of the art and status of the development of the MAELSTROM platform in depth. The specific requirements of the W&C community have been evaluated and discussed and the plans for the platform that will be developed have been adjusted accordingly. However, the overall workflow for W&C remains similar to the workflow of ML applications in other domains.

The differences emerge in the details, like size of a dataset used to train a ML model, and support of HPC infrastructure. Surprisingly, the actual application of ML does not seem to be the greatest challenge for the individual MAELSTROM applications. Hard restrictions on storage and data handling seem to be most problematic.



7 References

[1] "Architecture for MLOps using TFX, Kubeflow Pipelines, and Cloud Build." https://cloud.google.com/architecture/architecture-for-mlops-using-tfx-kubeflow-pipelines-and-cloud-build (accessed Sept. 14, 2021)

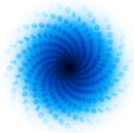
[2] "Deploying Models on AWS SageMaker – Part 1 Architecture." https://mlinproduction.com/sagemaker-architecture (accessed Sept. 14, 2021)

[3] "Productionizing Machine Learning with a Microservices Architecture." https://databricks.com/de/session_na20/productionizing-machine-learning-with-a-microservices-architecture (accessed Sept. 14, 2021)

[4] "Architecting a Machine Learning Pipeline." https://towardsdatascience.com/architecting-a-machine-learning-pipeline-a847f094d1c7 (accessed Sept. 14, 2021)

[5] H. Muccini, K. Vaidhyanathan (2021), "Software Architecture for ML-based Systems: What Exists and What Lies Ahead", arXiv preprint arXiv:2103.07950.

[6] T. Ben-Nun, T. Hoefler (2019), “Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis”, ACM Comput. Surv. 52, 4, Article 65 (September 2019), 43 pages. DOI:https://doi.org/10.1145/3320060



Document History

Version	Author(s)	Date	Changes
0.1	Markus Abel, Fabian Emmerich, Greta Denisenko (4cast)	16/09/2021	Version for review
1.0	Markus Abel, Fabian Emmerich, Greta Denisenko (4cast)	30/09/2021	Refinements after review

Internal Review History

Internal Reviewers	Date	Comments
Peter Dueben (ECMWF)	16/09/2021	Review passed with minor edits
Tal Ben-Nun (ETH)	24/09/2021	Review passed with minor edits

Estimated Effort Contribution per Partner

Partner	Effort
4cast	1 PM
Total	**1 PM**