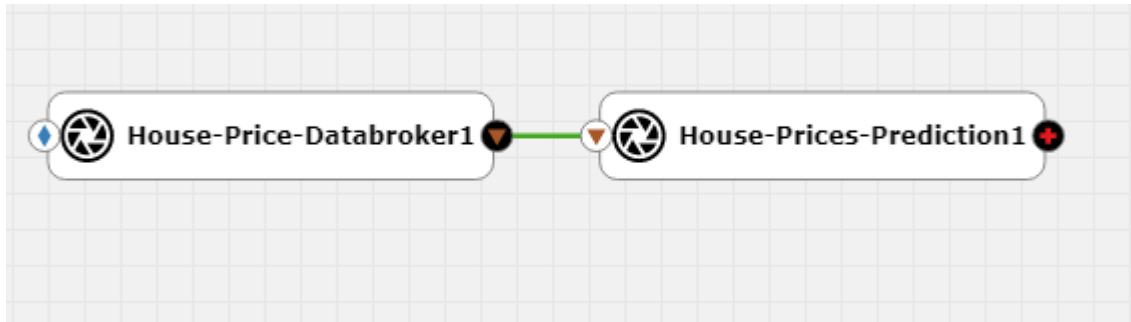


# AI4EU Experiments Container Format



## 1. Introduction

This document specifies the docker container format for tools and models that can be onboarded on the AI4EU Experiments platform so they can be used in the visual composition editor as re-usable, highly interoperable building blocks for AI pipelines.

In short, the container should define its public service methods using protobuf v3 and expose these methods via gRPC. All these technologies are open source and freely available. Here are the respective home pages for documentation and reference:

<https://docs.docker.com/reference/>

<https://developers.google.com/protocol-buffers/docs/overview>

<https://developers.google.com/protocol-buffers/docs/proto3>

<https://www.grpc.io/docs/>

Because the goal is to have re-usable building blocks to compose pipelines, the main reason to choose the above technology stack is to achieve the highest level of interoperability:

- docker is today the defacto standard for serverside software distribution including all dependencies. It is possible to onboard containers for different architectures (x86\_64, GPU, ARM, HPC/Singularity)

- gRPC together with protobuf is a proven specification and implementation for remote procedure calls supporting a broad range of programming languages and it is optimized for performance and high throughput.

**Please note that the tools and models are not limited to deep learning models.** Any AI tool from any AI area like reasoning, semantic web, symbolic AI and of course deep learning can be used for pipelines as long as it exposes a set of public methods via gRPC.

## 2. Define model.proto

The public service methods should be defined in a file called **model.proto**:

- It should be self contained, thus contain the service definitions with all input and output data structures and no imports can be used
- a container can define several rpc methods, but all in one .proto file and in the same service { } block
- package declarations in the context of pipelines can cause scoping issues if a message in different protobufs belongs to different packages

```
//Define the used version of proto
syntax = "proto3";

//Define a message to hold the features input by the client
message Features {
    float MSSubClass      = 1 ;
    float LotArea         = 2 ;
    float YearBuilt       = 3 ;
    float BedroomAbvGr    = 4 ;
    float TotRmsAbvGrd    = 5 ;
}

//Define a message to hold the predicted price
message Prediction {
    float salePrice       = 1 ;
}
```

```
//Define the service
service Predict {
    rpc predict_sale_price(Features) returns (Prediction);
}
```

**Important:** The parser for .proto-files inside AI4EU Experiments is much less flexible than the original protobuf compiler, so here are some rules. If the rules are not followed, it prohibits the model from being usable inside the visual editor AcuCompose.

Rule	Good	Bad
syntax spec must be in double quotes	<code>syntax = "proto3";</code>	<code>syntax = 'proto3';</code>
there must always be a space before an opening curly brace	<code>service Predictor {</code>	<code>service Predictor{</code>
the closing curly brace must be in a separate line	<code>message Empty {</code> <code>}</code>	<code>message Empty {}</code>
there must not be curly braces after a rpc line	<code>rpc predict</code> <code>(AggregateData)</code> <code>returns (Prediction);</code>	<code>rpc predict</code> <code>(AggregateData)</code> <code>returns (Prediction) {}</code>

### 3. Create the gRPC docker container

Based on model.proto, you can generate the necessary gRPC stubs and skeletons for the programming language of your choice using the protobuf compiler **protoc** and the respective protoc-plugins. Then create a short main executable that will read and initialize the model or tool and starts the gRPC server. This executable will be the entrypoint for the docker container.

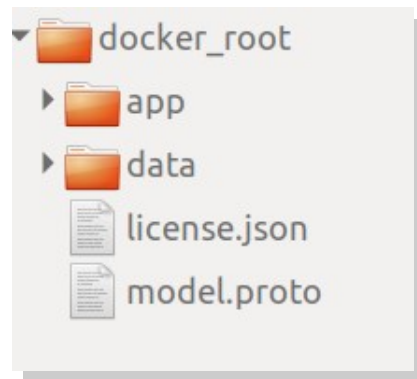
**The gPRC server must listen on port 8061.**

If the model also exposes a **Web-UI** for human interaction, which is optional, it must listen on **port 8062**.

The filetree of the docker container should look like below. In the top level folder of the container should be the files

- model.proto
- license.json

And also the folders for the microservice like app and data, or any supplementary folders:



The license file is not mandatory and can be generated after onboarding with the License Profile Editor in the AI4EU Experiments Web-UI:

<https://docs.acumos.org/en/clio/submodules/license-manager/docs/user-guide-license-profile-editor.html>

There are several detailed tutorials on how to create a dockerized model in this repository: <https://github.com/ai4eu/tutorials>

**Important recommendation:** for security reasons, the application in the container should **not** run as root (which is the default). Instead an unprivileged user should be created that runs the application, here is an example snippet from a Dockerfile:

```
RUN useradd app
USER app
CMD ["java", "-jar", "/app.jar"]
```

This will also allow the docker container to be converted into a Singularity container for HPC deployment.

## 4. Status and Error Codes

The models should use gRPC status codes according to the spec: [https://grpc.github.io/grpc/core/md\\_doc\\_statuscodes.html](https://grpc.github.io/grpc/core/md_doc_statuscodes.html)

For example if no more data is available, the model should return status 5 (NOT\_FOUND) or 11 (OUT\_OF\_RANGE).

## 5. Onboarding

The final step is to onboard the model. There are several ways to onboard a model into AI4EU Experiments but currently the only recommended way is to use **“On-boarding dockerized model URI”**:

1. Upload your docker container to a public registry like Docker Hub
2. Start the onboarding process like in the screenshot below
3. Upload the protobuf file
4. Add license profile

The screenshot shows the 'ON-BOARDING DOCKERIZED MODEL URI' section of the AI4EU Experiments interface. The left sidebar contains navigation links: HOME, MARKETPLACE, MY MODELS, CATALOGS, ON-BOARDING MODEL (highlighted), DESIGN STUDIO BETA, PUBLISH REQUEST, Q AND A, and ML LEARNING PATH. The main content area has four tabs: ON-BOARDING BY COMMAND LINE, ON-BOARDING BY WEB, ON-BOARDING DOCKERIZED MODEL URI (selected), and ON-BOARDING DOCKERIZED MODEL. Below the tabs are three circular icons: 'Create Solution', 'Add Artifacts', and 'Not yet on-boarded'. The 'ON-BOARD DOCKERIZED MODEL URI' section contains a form with the following fields: 'Model Name \*' (text input), 'Host \*' (text input), 'Port \*' (text input), 'Image \*' (text input), and 'Tag' (text input). Below these is a section for 'Upload Protobuf File' with a 'Browse' button and an 'Upload' button. A note indicates 'Supported files type: .proto'. There is a checkbox for 'Add License Profile'. At the bottom are 'On-Board Model' and 'Upload New' buttons. On the right, a sidebar titled 'Instruction for dockerized model URI on-boarding' contains a link 'On-board a dockerized model URI'.

## 6. First Node Parameters (e.g. for Databrokers)

Generally speaking, the orchestrator dispatches the output of the previous node to the following node. A special case is the first node, where obviously no output from the previous node exists. In order to be able to implement a

general orchestrator, the first node must define its services with an Empty message type. Typically this concerns nodes of type Databroker as the usual starting point of a pipeline.

```
syntax = "proto3";
```

```
message Empty {  
}
```

```
message NewsText {  
    string text = 1;  
}
```

```
service NewsDatabroker {  
    rpc pullData(Empty) returns(NewsText);  
}
```

To indicate the end of data, a Databroker should return status code 5 or 11 (see chapter 4)

## 7. Scalability, GPU Support and Training

The potential execution environments range from Minikube on a Laptop over small Kubernetes clusters to big Kubernetes clusters and even HPC and optional GPU acceleration. **It is possible to support all those environments with a single container image** taking into account some recommendations:

- let the model be flexible with memory usage: use more memory only if available
- let the model be scalable if more cpu cores are available (allow for concurrency):  
[https://en.wikipedia.org/wiki/Concurrency\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Concurrency_(computer_science))
- Some AI frameworks like PyTorch or Tensorflow can be used in a way to work with or without GPU with the same code. Here is an example with PyTorch: <https://stackoverflow.com/a/56975325>

- even training is possible, if the model exposes the corresponding methods in the protobuf interface

## 8. Shared Folders for Pipeline Nodes

to be compatible with the upcoming shared folder concept of AI4EU Experiments, please use an environment variable to pass the absolute path of the shared folder to the processes running inside the container: so it should be the path from the inside the container point of view. Let's assume the shared folder is mapped as "/data/shared" into the container, then the container could be started like

```
docker run --env SHARED_FOLDER_PATH=/data/shared ...
```

How this shared folder is actually set up, depends your container runtime and is different for docker, docker-compose or kubernetes. For AI4EU Experiments, the kubernetes deployment client will take care of it.

## 9. Streaming Support for gRPC Services

The “stream” keyword is supported for both RPC input and output.

That means a RPC can have the following forms, assuming Input and Output are Protobuf message types:

- Each RPC call consumes one input message and returns one output message:

```
rpc call(Input) returning (Output);
```

- Each RPC call gets at least one input message, decides by itself when to stop consuming the input stream, after which it returns a single output message:

```
rpc call(stream Input) returning (Output);
```

- Each RPC call gets at one input message and decides by itself how many output messages to produce before closing the stream:

```
rpc call(Input) returning (stream Output);
```

- Each RPC call gets at least one input message, decides by itself how many output messages to produce and when to stop consuming input before closing input and output streams:

```
rpc call(stream Input) returning (stream Output);
```

Special cases of the above are the following, where **Empty** is a message type without any fields.

- RPC is called immediately when orchestration starts and is restarted whenever it closes the output stream:

```
rpc call(Empty) returning (stream Output);
```

This pattern is useful for GUIs, with one RPC call for each type of User Event that should trigger a computation in other components.

This pattern is useful for sensors, with one RPC call for each type of sensor reading that should trigger a computation in other components.

- RPC is consuming messages only:

```
rpc call(stream Input) returning (Empty);
```

This pattern is useful for GUIs, with one RPC call for each type of input to display.

Some notes about how streaming works:

- Connecting “stream” outputs with “non-stream” inputs and vice versa is allowed.
- Orchestrator creates a Queue for each connection between two ports.
- Orchestrator creates a Thread for each RPC call. Essentially orchestration is parallel.
- A RPC is started as soon as there is a message in the input queue, or immediately for the message called **Empty**.
- Cycles among components are possible.

For example a cycle from a GUI that returns User Events as streaming output to a computation component with a simple non-streaming RPC which feeds back into the GUI into a RPC that displays the result.

GUI RPC1 (**Empty** input, stream output) → Computation RPC → GUI RPC2 (stream input)



The Sudoku Tutorial streaming branch contains such a GUI (a webinterface), see <https://github.com/peschue/ai4eu-sudoku/blob/streaming/gui/sudoku-gui.proto> .