Scientific Report

AI Folk – Resource Management for Distributed AI

2023

Contents

1 Introduction

Year 2023 has been very productive for the AI Folk , having completed designing and developing the autonomous driving scenario and having made significant steps towards the completion of the AI Folk methodology.

We have successfully implemented the scenario on the basis of the Flash-mas multi-agent framework, integrating the AI Folk methodology with the existing MAS framework.

The main principle of the AI Folk methodology is the ability of agents to semantically describe machine learning (ML) models, to search for appropriate ML models in the repositories of other agents, to transfer such models and use them for their own goals.

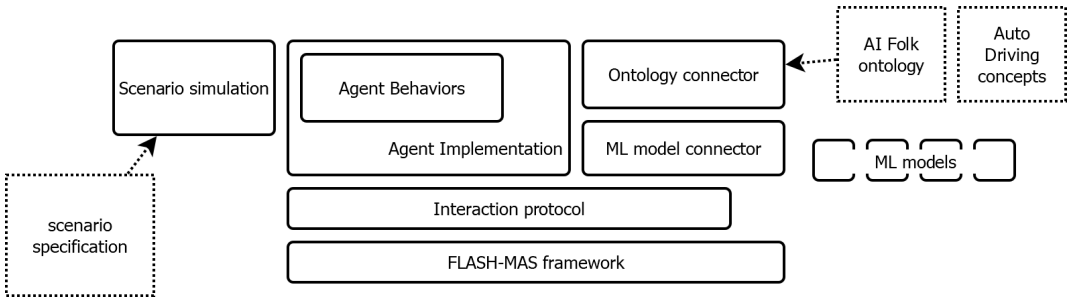

Figure 1: Blocks developed for the AI Folk scenario.

Developing the autonomous driving scenario has encompassed various important steps, some of which have not been anticipated at the time of the proposal:

- development of the scenario itself and focus on elements which are relevant to the AI Folk methodology.
- a survey of available ML models and datasets for autonomous driving, testing and evaluation of the models, and selection of models to be used for the scenario.
- development of ontological concepts and instances needed to semantically describe all elements in the autonomous driving scenario.
- development of methods to interface between the ML models, implemented in the Python programming language, and the multi-agent framework, implemented in Java.
- the creation of new types of entities in the FLASH-MAS framework, dedicated to the AI Folk scenario implementation.
- the integration of the AI Folk ontology, the AI Folk interaction protocol, the execution of ML models, and agent behaviours specific to the AI Folk methodology, in the existing FLASH-MAS framework.
- development of mechanisms for simulating the autonomous driving scenario, integrated with the FLASH-MAS framework.

A structural view of these elements is presented in Figure 1.

This report details the architecture of the AI Folk ontology, details on the AI Folk interaction protocol and the architecture of the experimental AI Folk application, the description of the autonomous driving and disaster response scenarios, and lessons learned so far related to the AI Folk methodology.

1.1 Challenges

There were several unanticipated challenges in implementing the AI Folk methodology and in realizing the experiments.

An important challenge was the fact that most machine learning (ML) models are implemented using Python libraries, whereas the deployment framework, as well as the libraries for working with ontologies, are implemented in Java. That meant developing a mechanism for interaction between the two languages, which is based on a local Python RESTful web server which loads and manages the ML models, and what we call a *driver* which assists the components implemented in Java with accessing the server.

A challenge also related to the ML models was that there is a great deal of variety in how ML models are

evaluated (executed), how input must be transformed in preparation for the model, and how the output of the model must be processed after the model evaluates. This variety was so great that the solution was to implement specific pieces of code for each of the models integrated in our implementation, to deal with each one of these steps – input transformation, model evaluation, and output processing.

The variety of non-agent entities that were necessary in the implementation resulted in the need to implement a new type of entity in FLASH-MAS – the *driver*, an entity which is local to a node and enables agents on that node to interface with local resources, such as the ML Python server, the AI Folk Ontology, and the execution of the scenario.

In order to have correct information about a model, one needs to have information about the dataset on which the model was trained, or, in case this is not available, one needs to evaluate the model on an appropriate dataset. On the dataset, information relevant to the scenario must be gathered. For instance, for the autonomous driving scenario, for dataset we needed to obtain information such as weather conditions, number of cars and pedestrians, and so on; and we needed to evaluate models on the datasets in order to assess their performance (see 5.4).

1.2 Progress on objectives

At the time of this report we have achieved the following progress:

WP1: We have completed creating the AI Folk ontology. The ontology contains scenario - independent concepts and instances, as well as concepts and instances specific to the discussed scenarios.

WP2: The AI Folk protocol has been implemented and integrated into the FLASH-MAS framework. It has been tested in the experiments and it works correctly.

WP3: Work on the methodology is underway and many lessons have been learned during the development of the autonomous driving scenario. What is left to do is to give a finished, applicable aspect to the lessons learned so far.

WP4: An autonomous driving scenario has been researched and all of its aspects implemented, encompassing ontology, simulation of agent deployment, and experiments with the elements of the methodology.

WP5: Work on the disaster response scenario has started, having completed surveying models and datasets for disaster response, and having gathered some elements for the scenario.

WP6: Two papers have been published bearing the project acknowledgement, one in a Q2 journal and one at a prestigious conference with Springer post-proceedings (to be published). Another journal paper is under development.

2 AI Folk Ontology

In the AI Folk communication protocol, agents exchange information between themselves whenever one agent encounters a situation where the input it is receiving from its environment is far different from the input expected by its decision making algorithms in order to produce an accurate decision. The agents will exchange messages consisting in:

- A *qualitative* description of the inputs received over a given window of time, i.e. what *kind* of environment they are perceiving
- Decision making models (usually ML based), which were developed / trained using *data* that is similar *in kind and quality* to the one currently perceived by the querying agent

The purpose of the AI Folk ontology is to introduce the *vocabulary* required to exchange such messages: (i) describe the data in terms of its *qualities* - i.e. provide a **data context description**, and (ii) describe the AI models and their *evaluation* mode, so as to determine if they are suitable for a given *data context*.

Describing data in terms of its *qualities* is by necessity an *application domain* dependent procedure. Consequently, the AI Folk ontology is designed in a modular fashion. It contains a **core** vocabulary defining how concepts such as a *Scenario* and its *TaskCharacterization* are related to a *Model* and its *ModelEvaluation* in an *ApplicationDomain* (see also Figure 2). The core module is then extended by *domain specific* vocabulary (such as the segmentation in autonomous driving one) which introduces concepts and relations that describe *qualities* of data from key perspectives of the application domain (see Section 5.3)

Figure 2 gives a general overview of the AI Folk core concepts and relations. The left side of the concept graph starts from a *Scenario* that is described in terms of *TaskCharacterizations*. Tasks are subdivided from the perspective of a ML algorithm into objectives such as *Classification*, *Regression*, *Segmentation*, *ObjectDetection*. Each such task is *has a domain* to which it applies. The *ApplicationDomain* is also one that was addressed by *Dataset*. The *Dataset* in turn was used during the *evaluation* of a *Model*, obtaining an *EvaluationResult*. In this way, scenario tasks (as perceived by the agent) are linked to models that have been on datasets that have similar content.

Both *TaskCharacterization* and a *Dataset* are further described by one or more *DataContexts*, which is the concept that gets extended in the domain specific ontology module.

To determine the domain specific concepts and relations, the AI Folk ontology uses the *competency question* methodology to determine what the main objects and relations of *relevance* are. For the task of *segmentation in an autonomous driving scenario* one might ask, for example:

- Which are the target classes of the segmentation task? (e.g. drivable lane, opposite lane, traffic

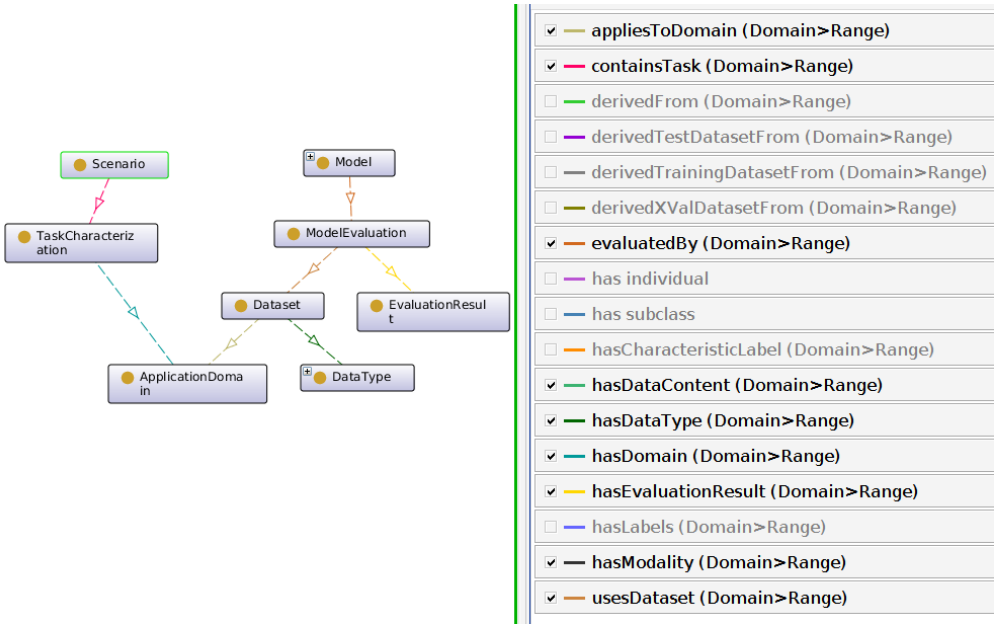

Figure 2: An overview of the main concepts and properties in the AI Folk Core ontology module, showing how Scenarios described in terms of Tasks relate over an ApplicationDomain with ML Models and their Evaluation on Datasets from the same domain

participant, pedestrian, other)
- What are the *illumination conditions* for driving scenes present in the task?
- What are the *weather conditions* for driving scenes present in the task?
- What are the *scene categories* (e.g. urban, parking lot, rural, highway) present in the task?

In Section 5.3 we detail how the AI Folk *DrivingSemanticSegmentation* ontology module addresses these modeling issues.

3 AI Folk Interaction Protocol

The AI Folk protocol implementation mainly follows the specification given in our 2022 report. The protocol is currently implemented in the ML Management Shard. We will give some details on some aspects that have arisen during the implementation and the experiments.

The purpose of the AI Folk interaction protocol is to ensure that agents can search for models which are appropriate to a situation and can transfer information about those models. While initially we intended that search messages contain a SPARQL query that can be applied directly by the local Ontology Driver in order to identify appropriate models, while developing the experiments it resulted that sometimes the query would have to be overly complex to create, given the constraints of working with ontologies. Conversely, it resulted, while implementing situation detection, that situation detection itself required using some machine learning modes (e.g. for determining the weather, or the average number of pedestrians in a given time window), hence there is a finite and determined number of features of the situation that one can handle in a scenario. Given this determination, it resulted that instead of creating a SPARQL query and let the Ontology driver run it, it is easier, from the point of view of the developer implementing the scenario, to iterate through model descriptions and check for the appropriate features.

In conclusion, when an agent needs to search for a model with certain properties, it assembles an RDF graph describing a model with the needed features. It then serializes this description and sends it, in a message, to other agents.

There are several entry points in the AI Folk protocol, all of which are events, since agents in FLASH-MAS are generally event-driven. The first entry point is internal, in the case in which the ML Management Shard detects that the current model in use is not adequate for the current situation of the agent. The shard creates an `AIFOLK_SEARCH` message containing the description of an appropriate model and sends it to other agents.

The second entry point is the receipt of an `AIFOLK_SEARCH` message. In this case the shard looks into the list of models that it contains and checks which model, if any, best match the given criteria. It then sends an `AIFOLK_LISTING` message as a reply.

Upon receiving an `AIFOLK_LISTING` message, the ML Management Shard records it and, if all the replies have been received, or a deadline has passed, it chooses which model to use. For that model, it sends an `AIFOLK_REQUEST` message, which is replied to with an `AIFOLK_TRANSFER` message.

4 AI Folk Deployment

4.1 Entities

In order to create an AI Folk development test-bed and to run experiments with the autonomous driving scenario, there was a need to create or use an infrastructure that:

- supported the deployment of various entities and the management of their lifecycle. Among these entities there are agents, ML models, communication infrastructures, and auxiliary elements such as running the scenario or working with ontologies and ML models;
- enabled communication among the various entities, especially between agents;
- enabled rapid testing of multi-agent scenarios.

In order to satisfy these requirements without creating a development test-bed from scratch, we have decided to use the FLASH-MAS framework – A Fast, Lightweight Agent Shell – developed at UNSTPB in recent years [Olaru et al., 2019]. We have chosen FLASH-MAS over JADE [Bellifemine et al., 1999], a popular MAS deployment framework, due to its flexibility, support for dynamically loaded components, variety and customizability of deployed entities, and good performance.

In short, FLASH-MAS, implemented in Java, is able to deploy any persistent entity given it implements a basic `Entity` interface. There are a few standard entities in FLASH-MAS, such as nodes, pylons (embodying communication infrastructures), agents, and shards. *Shards* are sub-agent entities which embody specific agent functionality, such as messaging, or monitoring.

In order to deploy the AI Folk experiments, we needed to build several custom entities, as follows.

Several **drivers** had to be created and designed, for interaction between AI Folk agents and their environment. We call *drivers* persistent entities that enable the relation with node-local non-agent entities (different from pylons, which handle the relation with network-wide infrastructures, e.g. for communication). We have created three pylons for each node in the deployment, to deal with: **scenario execution** – feeding input data to agents and checking their output; **ontology** access and querying (see Section 2); and evaluation of ML models – see Section 4.3.

Several agent shards had to be implemented as well, embodying the functionality needed to implement the AI Folk methodology. The **Scenario shard** receives input from the Scenario driver and feeds it to the agent's event queue. It also intercepts the agent's actions and sends them to the Scenario driver for verification and monitoring. The evaluation of ML models on the input, in order to decide the action, is done by the **ML Pipeline shard**, which uses the currently selected ML models.

The actual implementation of the AI Folk methodology, in terms of evaluation and selection of the appropriate models to use, as well as searching for other, better models to use in the current situation, is done by the **ML selection shard**.

A class-interaction diagram presenting the relations between these entities is presented in Figure 3.

4.2 Deployment

Thanks to the advantages of FLASH-MAS , deployment is easily specified via an XML file, or via the command line, in a very simple and intuitive manner. For instance, for our test scenario, the command line for one node looks something like this:

```
quick.Boot -load_order driver;pylon;agent
-package testing net.xqhs.flash.ml src-experiments aifolk.core aifolk.onto
```

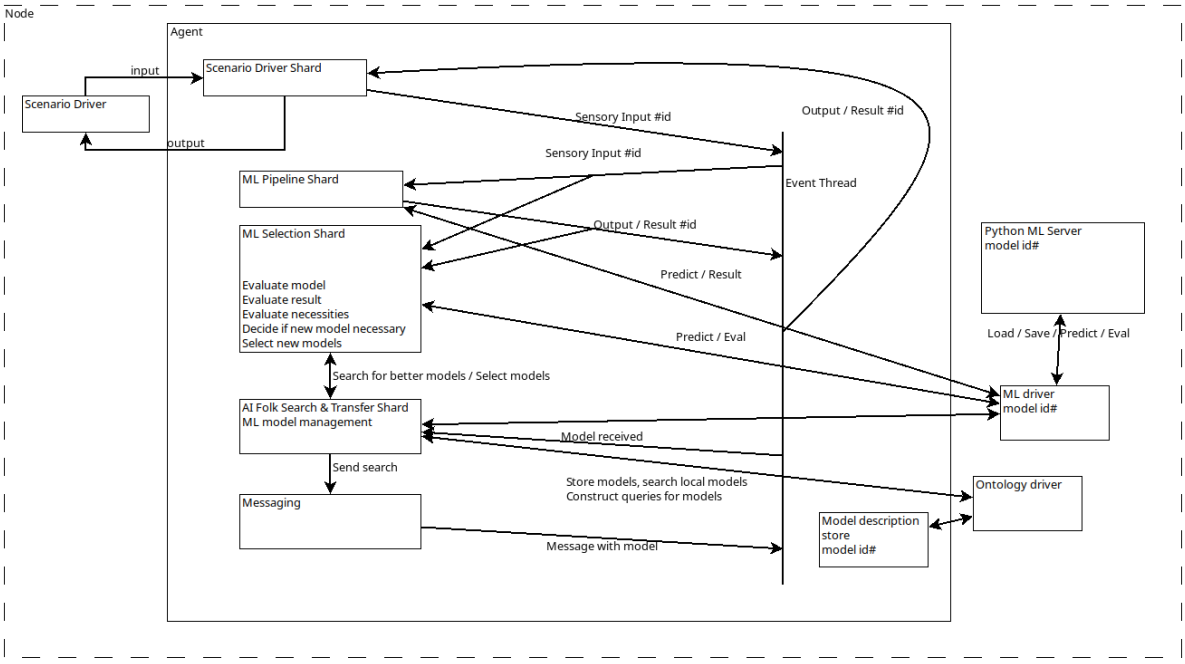

Figure 3: A class-interaction diagram presenting the relations between these entities in an AI Folk deployment

```
-loader agent:composite -node node1
-driver ML:mldriver -driver Scenario:scen script:script -driver Ontology:ont
-pylon local:pylon1
-agent A in-context-of:ML:mldriver in-context-of:Scenario:scen
-shard scenario -shard MLPipeline -shard MLManagement -shard messaging
-shard EchoTesting exit:20
```

In this specification, the important highlights are the three drivers, and the specification of one agent (all agents are specified similarly), which is placed in the context of the three drivers, so it can access their functionality, and which contains functionality for receiving scenario-related events (the scenario shard), for using ML models (the ML pipeline shard), for managing its own ML models, for messaging, and for echoing events at the system output.

4.3 Evaluation of Machine Learning Models

An important challenge in the deployment of AI Folk scenarios was the fact that the Flash-mas framework is implemented in Java, but most machine learning models are implemented in Python, and using dedicated ML libraries such as PyTorch[1] and Tensorflow[2].

Integrating a Python Flask server for neural network predictions into a Java project can offer significant advantages in terms of flexibility, scalability, and ease of deployment. Python, with its rich ecosystem of machine learning libraries like PyTorch and TensorFlow, excels in building and training neural networks. Flask, a lightweight web framework, provides a seamless way to expose these models through RESTful APIs. By employing a Python Flask server, we can encapsulate the complex neural network logic in a language optimized for machine learning, while Java, renowned for its robust enterprise

[1]PyTorch: pytorch.org
[2]Tensorflow: www.tensorflow.org

solutions, can focus on handling other aspects of the project. This separation of concerns facilitates a more modular and maintainable codebase. Additionally, Flask's simplicity and quick development cycles allow for rapid prototyping and testing, enabling efficient iteration of the machine learning components of the project.

For the server, we chose to use a YAML configuration file where we define the initial models and datasets. For each model, we specify its name, the path to the .pth file containing the model weights, the input type processed by the model, the task for which the model is proposed, the dataset used for training, and information related to any required preprocessing. Similarly, for each dataset, we specify its name and the possible labels. To better understand the proposed format, we provide an example of a configuration file.

```
PORT: 5000
MODELS:
  -
    name: "MobileNetV2"
    path: "models/mobilenetv2.pth"
    transform: "datasets.transform.ImageNetTransform"
    cuda: false
    input_space: RGB
    task: "classification"
    dataset: "ImageNet"
  -
    name: "ResNet18"
    path: "models/resnet18.pth"
    cuda: true
    input_space: RGB
    transform: "datasets.transform.ImageNetTransform"
    task: "classification"
    dataset: "Custom Dataset"
DATASETS:
  -
    name: "Custom Dataset"
    class_names: ['apple','atm card','cat','banana','bangle','battery','bottle','broom',
                  'bulb','calender','camera']
```

For each service, the route is defined using the `@app.route` decorator with the path `/predict`. It specifies that the route can only handle HTTP POST requests (`methods=['POST']`). The services provided by this server are as follows:

- `/add_model` – The `add_model` service is a Flask route defined in the ML server. This service is designed to handle HTTP POST requests for adding machine learning models to the server. Here's a breakdown of how it works:
  1. Request Parameters
     The service expects the following parameters to be included in the POST request form data:
     - `model_name`: The name of the model.
     - `model_file`: The path to the model file.
     - `model_config`: A JSON-formatted string containing additional configuration for the model.
  2. Functionality
     Upon receiving a request, the service extracts the relevant information from the form data. It then attempts to load the model using the `get_model` function, which utilizes PyTorch to load the model from the specified file path. The loaded model is associated with its configuration, including the model name, file path, CUDA availability, task type, and dataset information. The loaded model is added to a global dictionary `models`, where model names

serve as keys.

3. Response
   The service returns a JSON response:
   - If the model is successfully added, it returns a success message indicating that the model has been added.
   - If there is an error (e.g., missing model name or model file), it returns an error message along with a 400 Bad Request status.
4. Example POST Request Data

```
{
  "model_name": "example_model",
  "model_file": "/path/to/example_model.pth",
  "model_config": "{\"cuda\": true, \"task\": \"classification\",
                   \"dataset\": \"mnist\"}"
}
```

5. Example Success Response

```
{
  "message": "Model 'example_model' has been successfully added."
}
```

   This service facilitates the dynamic addition of machine learning models to the server, allowing users to extend the set of available models without modifying the code.

- `/add_dataset`
  1. Request Parameters
     The service expects the following parameters to be included in the POST request form data:
     - `dataset_name`: The name of the dataset.
     - `dataset_classes`: A JSON-formatted string containing the class names for the dataset.
  2. Functionality
     Upon receiving a request, the service extracts the relevant information from the form data. It then checks if the dataset already exists in the global `datasets` dictionary. If it does not exist, the service adds the dataset with its class names to `datasets`.
  3. Response
     The service returns a JSON response:
     - If the dataset is successfully added, it returns a success message indicating that the dataset has been added.
     - If there is an error (e.g., the dataset already exists or missing dataset name or class names), it returns an error message along with a 404 Not Found status.
  4. Example POST Request Data

```
{
  "dataset_name": "example_dataset",
  "dataset_classes": "[\"class1\", \"class2\", \"class3\"]"
}
```

  5. Example Success Response

```
{
  "message": "Dataset 'example_dataset' has been successfully added."
}
```

     This service facilitates the dynamic addition of datasets to the server, allowing users to extend the set of available datasets without modifying the code.

- `/predict`
  1. Request Parameters
     The service expects the following parameters to be included in the POST request form data:
     - `model_name`: The name of the machine learning model.
     - `input_data`: Base64-encoded input data for prediction.
  2. Functionality

Upon receiving a request, the service extracts the relevant information from the form data. It checks if the specified model exists in the global `models` dictionary. If the model exists, it performs the following steps:
   - Decodes the Base64-encoded input data.
   - Utilizes the PyTorch model and associated transformations to make a prediction.
   - Constructs a response containing the prediction results and additional information like class names, task type, and dataset.

3. Response
   The service returns a JSON response:
   - If the model exists and the prediction is successful, it returns a response containing the prediction results and related information.
   - If there is an error (e.g., the specified model does not exist), it returns an error message along with a 404 Not Found status.

4. Example POST Request Data

```
{
  "model_name": "example_model",
  "input_data": "base64_encoded_data"
}
```

5. Example Success Response

```
{
  "prediction": [0.85, 0.15, 0.0],
  "class_names": ["class1", "class2", "class3"],
  "task": "classification",
  "dataset": "example_dataset"
}
```

   This service facilitates making predictions using a specified machine learning model, providing flexibility for different models and tasks.

- `/get_models`
  1. Functionality
     Upon receiving a request, the service reads the model configuration file specified by the constant `MODEL_CONFIG_FILE`. It then constructs a response containing information about available models by iterating through the models defined in the configuration.
  2. Response
     The service returns a JSON response:
     - If the model configuration file is successfully loaded, it returns a response containing a dictionary of available models and their respective configurations.
  3. Example Success Response

```
{
  "models": {
    "example_model1": {
      "name": "example_model1",
      "path": "/path/to/example_model1.pth",
      "cuda": true,
      "task": "classification",
      "dataset": "mnist"
    },
    "example_model2": {
      "name": "example_model2",
      "path": "/path/to/example_model2.pth",
      "cuda": false,
      "task": "regression",
      "dataset": "cifar10"
    },
    ...
```

```
  }
}
```

This service provides information about the available machine learning models based on the configuration file.

- `/export_model`
  1. Request Parameters
     The service expects the following parameters to be included in the POST request form data:
     - `model_name`: The name of the machine learning model to export.
     - `export_directory_path`: The path to the directory where the model will be exported.
  2. Functionality
     Upon receiving a request, the service checks if the specified model exists in the global `models` dictionary. If the model exists, it performs the following steps:
     - Reads the model configuration file specified by the constant `MODEL_CONFIG_FILE`.
     - Identifies the configuration for the specified model.
     - Creates a new configuration file containing only the information for the specified model.
     - Saves the new configuration file in the export directory.
     - Copies the model file to the export directory.
  3. Response The service returns a JSON response:
     - If the model exists and the export is successful, it returns a response containing a success message and the destination directory.
     - If there is an error (e.g., the specified model does not exist), it returns an error message along with a 404 Not Found status.
  4. Example POST Request Data

     ```
     {
       "model_name": "example_model",
       "export_directory_path": "/path/to/export_directory"
     }
     ```

  5. Example Success Response

     ```
     {
       "message": "Model 'example_model' has been successfully exported.",
       "destination": "/path/to/export_directory"
     }
     ```

     This service facilitates exporting a machine learning model along with its configuration to a specified directory.

5 Autonomous Driving Scenario

For the current stage of the project, an autonomous driving scenario has been developed. The purpose of the scenario is to have a hypothetical driver which is put in various situations that can prove the need for the cooperation with other agents in order to have better models, trained on more suitable datasets, in order to obtain the best results available. The scenario was made for object segmentation, one of the most important tasks regarding autonomous driving, but the idea can be extended regarding any other autonomous driving component - localization, perception, prediction, planning or motion control and any algorithm inside a specific component, for example object detection or depth estimation, considering the perception component.

This section describes the scenario used for this project, the datasets and the models that were used considering the current scenario, the corresponding ontology that was developed in order to have a

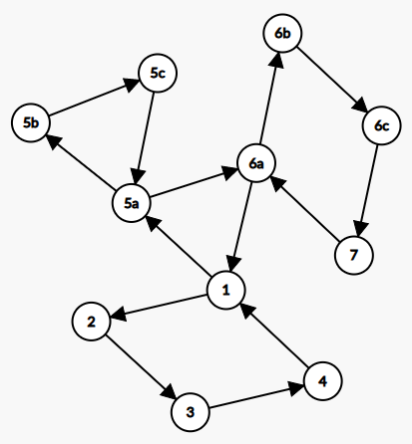

Figure 4: Autonomous driving scenario

symbolic modeling for the current scenario, the experiments that were made for this task and the analysis of the results.

5.1 Experimental Scenario

The experimental scenario contains 7 cities with different light conditions, population density and road types. At the beginning of the scenario, a car starts in the city labeled with number 1. The initial model that is used by the car is trained on Cityscapes. The segmentation network can vary too - the discussion will consider the best dataset for a city but also the possible switch between different models. The car will move through the day into various cities and different light conditions. The experimental scenario can be seen in Figure 4, which considers a variation between the nodes regarding the change of the dataset. The scenario considers only the semantic segmentation, but can be used for any other task.

Initially, the driver will go from city 1 to city 2, then city 3, then city 4. All of them are during the day. However, for city 2 the best training dataset is ADE 20k, for city 3 the best training dataset is pascal VOC and for city 4 CamVid has the best results. In city 1, suddenly a lot of people will appear and the car will go through a university campus. A new model could be used when this sudden change appears, then the car can switch to the previous model. City 2 will have a lot of unmarked roads, for which we could use another model and city 3 will have some unusual crossroads and another model will be used for that.

After visiting the fourth city, the car returns to city 1, then goes to city 5, which is best represented by Pascal VOC. However The light conditions will change and the dusk comes (node 5b in the figure). The best dataset for this condition is BDD100k. The night comes (5c) and the best model for the night is SUN RGB-D. The weather in 5c will be very bad, a storm will come and another segmentation model can be used for the bad weather.

The next day (the driver is back to the node 5a with VOC) the car moves to another city, city number 6 and the light change again from day to dusk and night (nodes 6a, 6b and 6c). However the best models considering this city are KITTI for day, SUN RGB-D for the dusk and BDD100k for the night. (here we can also vary the network for city 6). The car goes to another city, city number 7, which is best represented by Sun RGB-D. The car leaves this city and arrives in city 6 in dawn conditions (similar to dusk, represented by SUN RGB-D) then the day comes (node 6a) and then the car goes back to city 1.

Table 1: Semantic segmentation datasets (part 1)

Name	ADE20k	Cityscapes	Pascal VOC	Stanford	CamVid
created	2016	2016	2012	2017	2008
domain	general	driving	general	indoor	driving
tasks	object detection, semantic segmentation	detection, segmentation	classification, detection, segmentation	detection, segmentation, depth	object detection, semantic segmentation
no. classes	150	30	20	24	32
illumination	generally day	daytime	generally day	generally day	daytime
weather	generally sunny	good/ medium weather	generally sunny	indoor/ artificial light	mostly generally sunny

Table 2: Semantic segmentation datasets (part 2)

Name	KITTI	BDD100k	Vistas	SUN	Synthia
created	2013	2020	2017	2017	2016
domain	driving	driving	driving	mostly indoor	virtual city
tasks	multiple tasks (detection segmentation etc)	detection, segmentation	detection, segmentation	detection, segmentation, depth	semantic segmentation, detection
no. classes	11	19	124	37	13
illumination	daytime	day, dusk, night	various day time	generally artificial light/ day	daytime
weather	generally sunny	sunny, rain, overcast, snow	various day times	not appliable	sunny, overcast

5.2 Datasets and Models

For the current scenario, in order to be more realistic, some datasets and models were used. The analyzed datasets are:

- ADE20k [Zhou et al., 2017], Cityscapes [Cordts et al., 2016], Pascal VOC [Everingham et al., 2010], Stanford [Armeni et al., 2017], CamVid [Brostow et al., 2009], KITTI [Geiger et al., 2013], BDD100k [Yu et al., 2020], Vistas [Neuhold et al., 2017] as well as Sun RGB-D [Xiao et al., 2010] and Synthia [Ros et al., 2016];
- for the segmentation network there have been used DeepLabv3 [Yurtkulu et al., 2019] and YoloV8 [Jocher et al., 2023], which are state-of-the art segmentation models.

The most important information regarding the datasets can be found in table 1 and table 2. Besides the most relevant properties described in the tables, each dataset was evaluated according to a number of questions regarding the average number of cars or pedestrians per image, average percentage for the cars or the pedestrians in an image, the percentage of the weather conditions and the illumination conditions and also considering the number of different intersections in an image. These questions could be further used in order to evaluate the benefit of using one dataset against another.

5.3 Autonomous Driving Ontology

Remember from Section 2 that the AI Folk ontology is developed in a modular way, with the core vocabulary being extended by a domain specific one. In the driving scene segmentation scenario for autonomous driving a vocabulary to describe a *DataContext* specific to such an application domain is defined.

Figure 5 shows how the *DataContext* concept from the *core* AI Folk ontology is extended by a *DrivingSceneContext* specific one. The driving scene context is characterized by an *environment condition* (which can refer to the illumination condition or to the weather condition) and a *driving scene category*

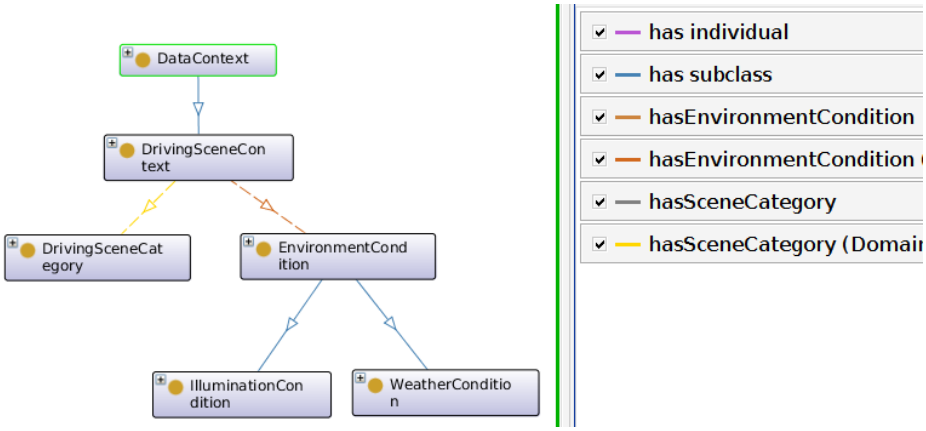

Figure 5: The concepts that characterize a *DrivingSceneContext* in the Segmentation for Autonomous Driving application domain

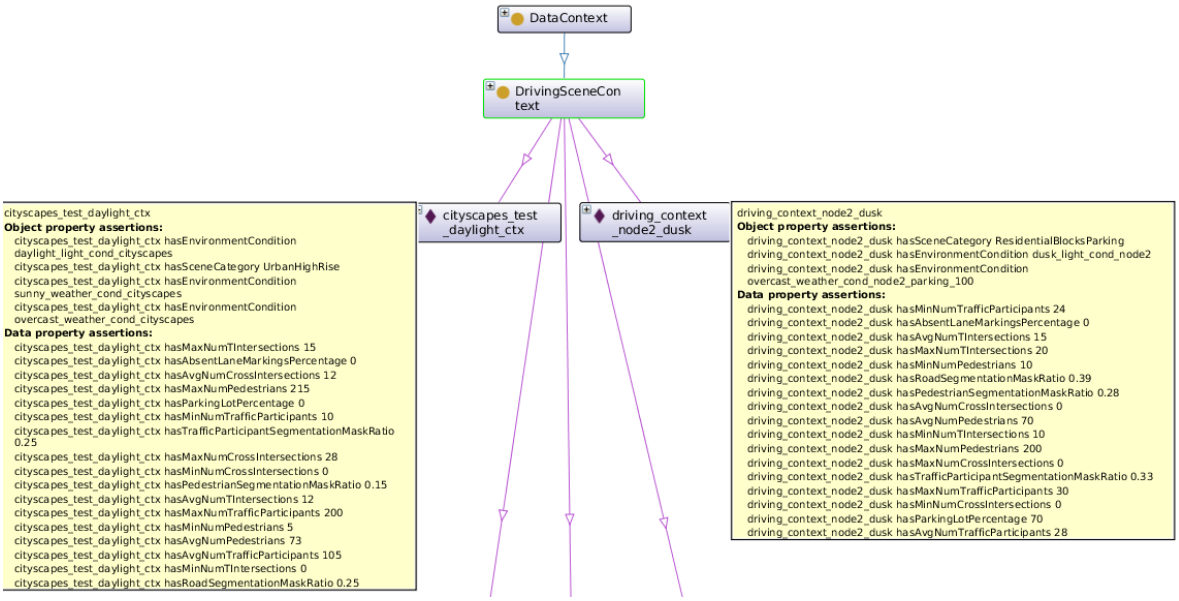

Figure 6: View of instances of the DrivingSceneContext concept. The *cityscapes_test_daylight_ctx* individual identifies a data context from the Cityscapes dataset, while the *driving_context_node2_dusk* identifies an individual representing a data context from the scene encountered by an agent during runtime.

(e.g. a city, rural, highway or parking area).

Furtermore, the *competency question* based ontology development methodology explained in Section 2, gives rise to *instances* of the *DrivingSceneContext* which have *DataProperties* as exemplified in Figure 6.

In Figure 6 it is important to note the extensive set of *data property assertions* which were identified through the *competency question* methodology and which are deemed likely to influence the performance of a segmentation model. Specifically, the data properties describe of some key situations and objects that are likely to be encountered in an autonomous driving scenario. They relate to aspects such as:

- The minimum, average and maximum number of pedestrians and other traffic participants
- The minimum, average and maximum number of cross or T-shaped intersections
- The average size ratio between a pedestrian or traffic participant and the whole image resolution (i.e. how large do pedestrians or traffic participants appear in an image)

The values to the properties are obtained using approaches as those described in Section 5.4. Depending on the values of these properties an agent can determine whether the current ML-based decision making model that it is currently using is still appropriate for the *data context* it is perceiving from its environment. If the model is no longer suitable from the *semantic* perspective described by the ontology, the agent will use the messaging protocol described in Section 3 (see also Figure 7).

5.4 Experiments and Results

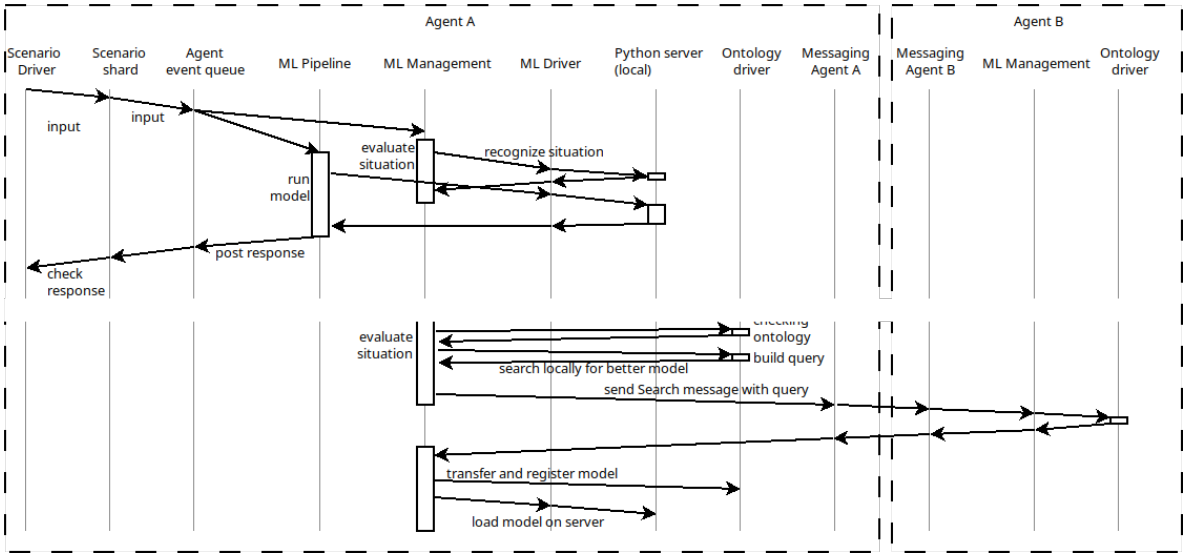

Figure 7: Interaction diagram between the various components in the implementation, when an input is received, and when the situation requires a more adequate model.

Together with implementing the elements of the AI Folk project we have also developed experiments testing essential functionality in the project. The following functionality was tested:

- interaction between agents by means of the AI Folk protocol.
- functionality of the Ontology Driver for storing model descriptions and building the description of searched models.
- ability of agents to use the ML Pipeline in order to obtain output from the received input
- ability of agents to decide which model is more appropriate for a given situation.
- ability of agents to recognize situation in terms of a given set of parameters (e.g. weather, number of pedestrians).
- ability to select the model to use in the ML Pipeline at runtime.

The concrete scenario that was researched was the changing of the currently used ML model for segmentation, based on the appropriateness of the model in the context of a large number of pedestrians close to the autonomous vehicle. We have made this choice based on the comparison between the average percentage taken by the pedestrians in the images on which the model was trained.

Given our observations – presented in Figure 8, one can see that, in the case where more pedestrians appear in the scene, YOLO detects more pedestrians in both datasets, so it is more adequate for use

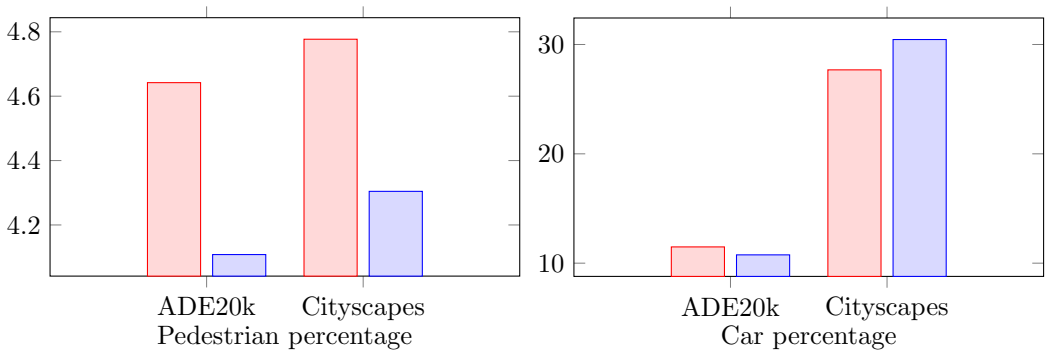

Figure 8: Comparison of average surface percentage of pedestrians and cars identified by YOLO (in red) and Deeplab (in blue) in the ADE20k and Cityscapes datasets.

in this case.

The tested scenario was the following: agent A is currently using the Deeplab model for segmentation. At a certain point, the number of pedestrians identified in a recent window of 5 images becomes considerably greater than the average of pedestrians in the dataset on which the model was trained. In this case, agent A looks for another model to use, if any is available. It will ask agents B and C for appropriate models. Agent B recommends the use of YOLO, which is more appropriate for the case of more pedestrians. Agent A downloads the model and switches to using it, at least for the period in which the number of pedestrians is high. A detailed view of the interaction between agents, drivers and sub-agent components is presented in Figure 7.

6 Disaster Response Scenario

In order to emphasize the flexibility of the AI Folk methodology we have devised a scenario in the domain of disaster response, identifying resources that we can use in implementing a proof-of-concept in this domain.

In order to create the scenario, we have use the following principles: agents are embedded in robotic devices with medium-sized computational capabilities; ML models are pre-trained and focused on specific functionality; agents are able to communicate with other nearby agents in order to exchange model parameters and experience information.

In the scenario there are three main types of agents – UAVs (Unmanned Aerial Vehicles) for reconnaissance, UAVs for fire management, and UGVs (Unmanned Ground Vehicles). The scenario is as follows:

After a major earthquake the buildings in the old center of the city are the most affected. Several UAVs and UGVs are deployed. Reconnaissance UAVs are tasked with assessing damage evaluation; Fire management UAVs are tasked with monitoring the evolution of fires; and UGVs are tasked with locating human survivors and victims using sight and sound.

UAV 003 is a reconnaissance UAV – it employs ML models that assess damage to buildings based on visual imaging, and models that perform image segmentation, recognizing building elements, cars, and people. While assessing a damaged building, the segmentation model identify human shapes, but it cannot discerne if the humans are suffering or if they are deceased.

Using a simple decision algorithm, the drone decides that, instead of calling an UGV, specialized in

(a) (b) (c)

Figure 9: Sample images from the AIDER dataset [Kyrkou, 2020] (a and b) and from the D-Fire dataset [DFi, 2023] (c).

locating human survivors and victims, it can attempt to assess this itself. It contacts a UGV nearby and it transfers a decision model that allows it to classify the state of human victims. Having acquired the model, it identifies two survivors in the damaged building. It then calls for rescue teams, as it is now certain they are needed in this case.

Later on, UAV 003 needs to return to base in order to recharge. However, on the way there is an area with several smoke columns from a fire below. THe UAV is unable to navigate through smoke, and it does not have enough energy to go around the smoke, but nearby there are several UAVs specialized in navigating through smoke. On request by UAV 003, one of them transfers it a control model allowing it to recognize the places with thinner smoke and fly though there.

We have already started a survey for identifying appropriate ML models and datasets for use in the disaster response scenario.

AIDER (Aerial Image Dataset for Emergency Response applications) [Kyrkou, 2020]: The dataset construction involved manually collecting all images for four disaster events, namely Fire/Smoke, Flood, Collapsed Building/Rubble, and Traffic Accidents, as well as one class for the Normal case. The dataset includes around 500 images for each disaster class and over 4000 images for the normal class. This makes it an imbalanced classification problem. A few relevant images can be seen in Figure 9.

D-Fire [DFi, 2023] is an image dataset of fire and smoke occurrences designed for machine learning and object detection algorithms with more than 21,000 images, of which 1164 contain only fire, 5867 contain only smoke, 4658 contain fire and smoke, and 9838 contain no fire and no smoke. All images were annotated according to the YOLO format (normalized coordinates between 0 and 1). However, the authors provide the yolo2pixel function that converts coordinates in YOLO format to coordinates in pixels.

YOLO is a very popular framework for detection, segmentation and tracking [Jiang et al., 2022]. We have successfully integrated it in our AI Folk experiments with the autonomous driving scenario and we can use it in the disaster response scenario to detect people or other elements in the environment.

7 AI Folk Methodology

The end goal of our project is to develop a methodology that allows the integration of machine learning model searching and transfer into a multi-agent deployment in any scenario. Since the beginning, we knew that in deploying the methodology for a given scenario, some parts of the implementation will remain invariant and some parts will need to be tailored to the necessities of the scenario. While implementing the experiments with the autonomous drive scenario we have learned several lessons, which we will need to verify and extend while implementing the disaster response scenario.

Table 3: Invariant and scenario-specific elements in deploying the AI Folk methodology in a given scenario.

Component	Invariant	Scenario-specific
AI Folk Ontology	Core concepts related to data contexts, datasets, functions, models, model evaluation, task characterizations, neural networks and optimizers, etc.	Specific concepts. E.g., for the autonomous driving scenario, the driving scene, environment conditions Instances of specific tasks, datasets and models that can be used in autonomous driving scenario
AI Folk Interaction Protocol	The protocol and its integration with the FLASH-MAS framework are invariant.	A difference appears in the case in which models are not transferred, but obtained through other means (e.g. through functions in the libraries). In this case the `REQUEST` and `TRANSFER` primitives are no longer necessary.
FLASH-MAS entities for AI Folk	Developed entities are scenario-invariant.	Situation recognition code is scenario-specific.
ML Python server	The implementation of the server is invariant, but the libraries that must be installed on the host machine and which the server imports are scenario-specific.	The libraries and the drivers of each individual model are scenario specific.
Scripts for situation recognition		The scripts are specific to model and to the given scenario, as they evaluate scenario-specific parameters.

Currently, the main outline of the AI Folk methodology, when implementing it for a given scenario, appears to be the following:

- gather all the ML models to be used in the scenario;
- for each model, create instances in the ontology, describing the model and the dataset(s) on which it was trained;
- create scripts that evaluate specific parameters on input data and on used datasets;
- implement code for input transformation, evaluation, output processing, and library import, for each model in the scenario;
- decide on the appropriate manner of transferring models between agents, depending on model size and network capabilities;
- integrate in agents decision code for selection of the models, depending on scenario-specific parameters for models;
- deploy agents according to the scenario requirements.

References

[DFi, 2023] (2023). D-Fire: an image dataset for fire and smoke detection. online.

[Armeni et al., 2017] Armeni, I., Sax, A., Zamir, A. R., and Savarese, S. (2017). Joint 2D-3D-Semantic Data for Indoor Scene Understanding. *ArXiv e-prints*.

[Bellifemine et al., 1999] Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE - a FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, pages 97–108. Citeseer.

[Brostow et al., 2009] Brostow, G. J., Fauqueur, J., and Cipolla, R. (2009). Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, 30(2):88–97.

[Cordts et al., 2016] Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., and Schiele, B. (2016). The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223.

[Everingham et al., 2010] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88:303–338.

[Geiger et al., 2013] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237.

[Jiang et al., 2022] Jiang, P., Ergu, D., Liu, F., Cai, Y., and Ma, B. (2022). A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073.

[Jocher et al., 2023] Jocher, G., Chaurasia, A., and Qiu, J. (2023). YOLO by Ultralytics.

[Kyrkou, 2020] Kyrkou, C. (2020). AIDER (aerial image dataset for emergency response applications). online.

[Neuhold et al., 2017] Neuhold, G., Ollmann, T., Rota Bulo, S., and Kontschieder, P. (2017). The mapillary vistas dataset for semantic understanding of street scenes. In *Proceedings of the IEEE international conference on computer vision*, pages 4990–4999.

[Olaru et al., 2019] Olaru, A., Sorici, A., and Florea, A. M. (2019). A flexible and lightweight agent deployment architecture. In *2019 22nd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 28-30 May 2019*, pages 251–258. IEEE.

[Ros et al., 2016] Ros, G., Sellart, L., Materzynska, J., Vazquez, D., and Lopez, A. M. (2016). The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243.

[Xiao et al., 2010] Xiao, J., Hays, J., Ehinger, K. A., Oliva, A., and Torralba, A. (2010). Sun database: Large-scale scene recognition from abbey to zoo. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 3485–3492. IEEE.

[Yu et al., 2020] Yu, F., Chen, H., Wang, X., Xian, W., Chen, Y., Liu, F., Madhavan, V., and Darrell, T. (2020). Bdd100k: A diverse driving dataset for heterogeneous multitask learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2636–2645.

[Yurtkulu et al., 2019] Yurtkulu, S. C., Şahin, Y. H., and Unal, G. (2019). Semantic segmentation with extended deeplabv3 architecture. In *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4. IEEE.

[Zhou et al., 2017] Zhou, B., Zhao, H., Puig, X., Fidler, S., Barriuso, A., and Torralba, A. (2017). Scene parsing through ade20k dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 633–641.

8 Executive report

We have continued the development of the AI Folk project with the integration of the components in a multi-agent system deployment framework and with the implementation of a scenario in the domain of autonomous driving. We have also made significant progress in the design and development of the disaster response scenario.

The AI Folk ontology has been implemented, providing the core concepts related to machine learning (ML) models, datasets, tasks, and data context characterization. Based on the core ontology, an ontology specific to the autonomous driving scenario has been implemented.

We have overcome a number of challenges and we have integrated the components of the project, allowing us to carry out experiments in the autonomous driving scenario. These components are a modified version of the FLASH-MAS multi-agent framework, a connector to the AI Folk ontology, a means to run experiments according to scenarios, and a mechanism to interface Java implementations with the code necessary to run ML models implemented using Python libraries. We have created components implementing the AI Folk interaction protocol.

For the autonomous driving scenario, we have surveyed models that can be used in such a scenario, focusing on the task of image segmentation, and we have extracted features of those models which can be used to separate between models in various scenario-specific situations. We have selected some models which we have integrated with the existing implementation, describing them, and implementing adequate input transformation and output processing.

We have created a demonstrative scenario in which an agent is using a given ML model, is recognizing its situation, and when the situation requires it, it contacts other agents in search of a more appropriate model. After receiving information about a more appropriate model, it loads that model and it starts using it, seamlessly with respect to other agent components. In the demonstrative scenario, we have focused on the number of pedestrians detected in a recent time window and on considering which models are more appropriate in the detection and segmentation of pedestrians in the input.

Ww have devised a primary, skeleton version of the AI Folk methodology. A such, we have designed the disaster response scenario and we have started applying the methodology to this scenario: we have made a primary survey of models which can be used in the scenario, with the immediate next steps of describing those models semantically, integrating those models with the existing implementation, and developing the necessary scenario-specific situation recognition functionality, so that deployment of the scenario can begin.

Project coordinator

Andrei Olaru