D6.2 External Interface Module

agricore

Deliverable Number	D6.2
Lead Beneficiary	IDE
Authors	IDENER
Work package	WP6
Delivery Date	M34
Dissemination Level	Public

www.agricore-project.eu

The Agricore project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No. 816078

Document Information

Project title	Agent-based support tool for the development of agriculture policies
Project acronym	AGRICORE
Project call	H2020-RUR-04-2018-2019
Grant number	816078
Project duration	1.09.2019-31.8.2023 (48 months)

Version History

Version	Description	Organization	Date
0.1	Deliverable template	IDENER	31-jan-2022
0.2	Structure reworking	IDENER	25-mar-2022
0.3	Content added	IDENER	25-mar-2022
0.4	First content revision	IDENER	28-mar-2022

Executive Summary

This deliverable reports the advances done in the development of the communications system between the AGRICORE modules, according to the specifications and technologies proposed in deliverable D6.1. This document is designed to companion the implementation performed, providing additional information and clarifications about it. Furthermore, the code generated will be available as an Open Source project in the official AGRICORE's repositories for testing and validation.

This document starts with a reminder of the changes that have been made in the AGRICORE architecture and explains the most critical technology that has taken part in these changes. After that, the test done is shown with the explanation of the scenario.

Abbreviations

Abbreviation	Full name
Dapr	Distributed Application Runtime
API	Application Programming Interface
ABM	Agent-Based Modelling
LMM	Land Markets Module
RPC	Remote Procedure Call
CI/CD	Continuous Integration/Continuous Deployment

List of Figures

List of Tables

No table of figures entries found.

Table of Contents

1 Introduction

The objective of the present report is to present a prototype implementation of the communication between modules, based on the revised architecture for AGRICORE as it was introduced on deliverable D6.1. Specifically, to perform a deep dive on the technologies and implementation of a substitute to the originally planned External Interface Module, previously thought to be necessary for the interconnection of the defined modules.

As was briefly discussed in D6.1, the External Interface Module can be replaced with the use of the Distributed Application Runtime (Dapr), a set of APIs that simplify connectivity between microservices. To this end, *Section 1 - Revised AGRICORE architecture* contains a quick summation of the different architecture versions and the reasons behind this paradigm change. Later, *Section 2 - Module Interfacing with Dapr* will contain a deep dive on the selected technologies, ranging from its general description to the specifics of implementation and the solving problems of the specific AGRICORE scenario.

2 Revised AGRICORE Architecture

The architecture devised for the AGRICORE Project has gone through several revisions during the development from what was initially presented in the Grant Agreement, especially relating to the presence and objectives of the External Interface Module. In this Section, a quick summation of the different versions of the architecture will be performed.

2.1 Initial Architecture

The first version of the AGRICORE modular architecture is depicted in Figure 1, which introduced the External Interface Module (labeled D.7), with the following description:

- **D.7 - External Interface Module**, that would serve as a central point to relate the set of external auxiliary modules (D.8, D.9, and D.10) with the Agent-based Simulation Module (D.6), but also as a means to facilitate the incorporation of additional external modules by other researchers.

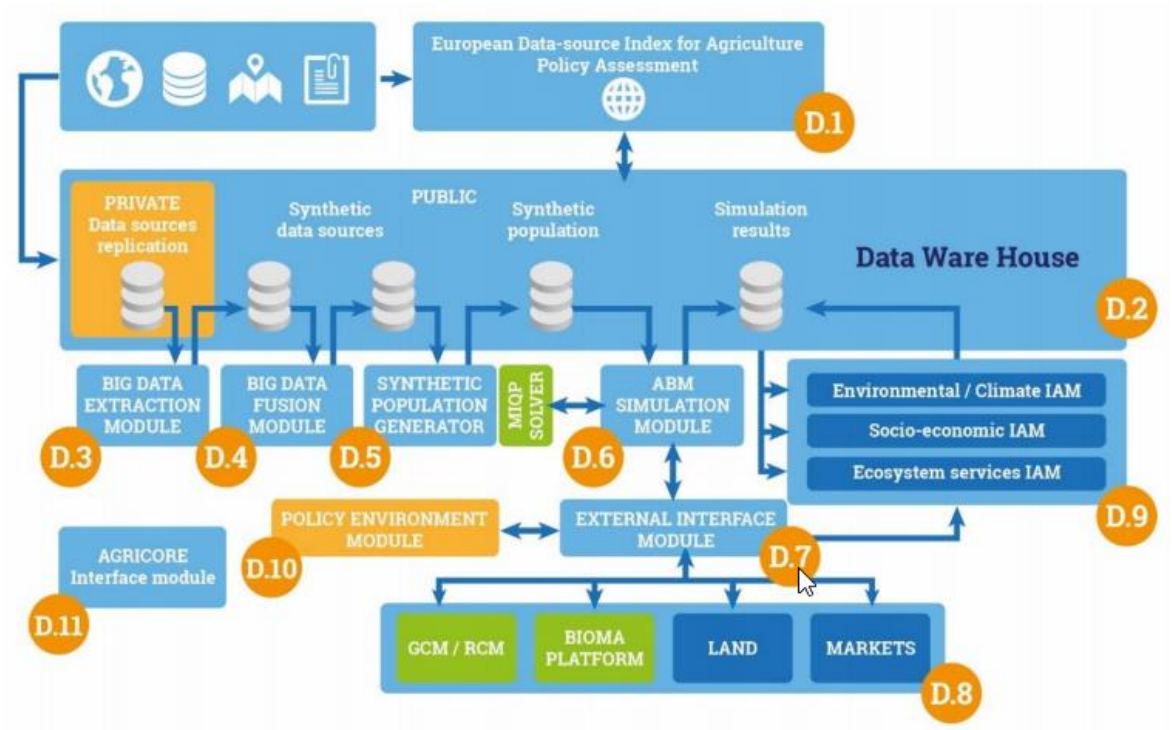

Figure 1 Initial Modular Architecture for AGRICORE

This version of the architecture doesn't need any more discussion and is recapped here for completion's sake.

2.2 Revised architecture

D6.1 presented a new version of the Agricore architecture, attending to the possible optimization brought by introducing Dapr into the workflow. In this way, development is simplified, as a whole module is removed in favour of specific Dapr sidecars for each of the remaining modules that need communication.

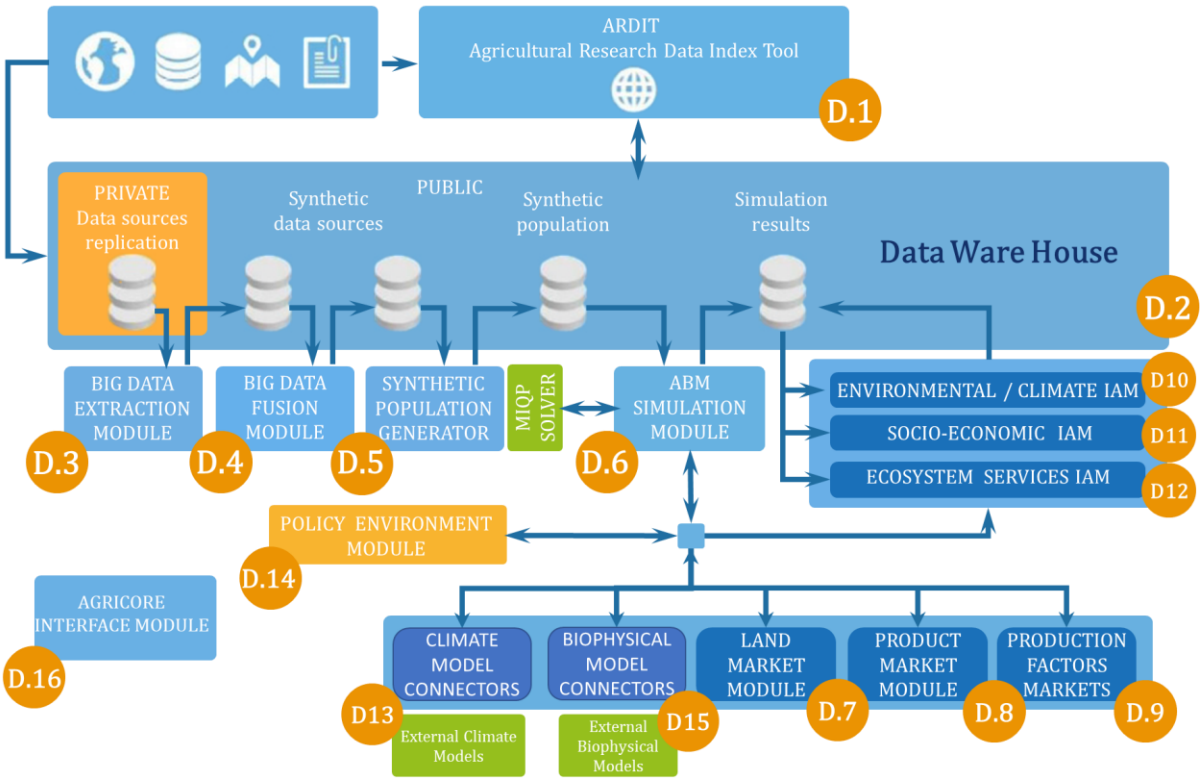

Figure 2 Revised Modular Architecture for AGRICORE

As explained in deliverable D6.1, the External Interface Module is erased from this architecture, in line with a paradigm shift towards a microservices-based system. In this context, using Dapr technologies to facilitate communication between microservices is a much more efficient approach.

3 Module Interfacing with DAPR

3.1 Introduction

Deliverable D6.1 already presented in a very concise way the potential benefits of using Dapr in the AGRICORE project, as well as reasons why it should be adopted. That discussion will not be repeated here, but to lay the foundations to explain the work performed in the present deliverable, an introduction to some specific concepts related to Dapr is needed.

3.2 DAPR summation

Its official documentation defines Dapr as "a portable, event-driven runtime that makes it easy for any developer to build resilient, stateless and stateful applications that run on the cloud and edge and embraces the diversity of languages and developer frameworks".

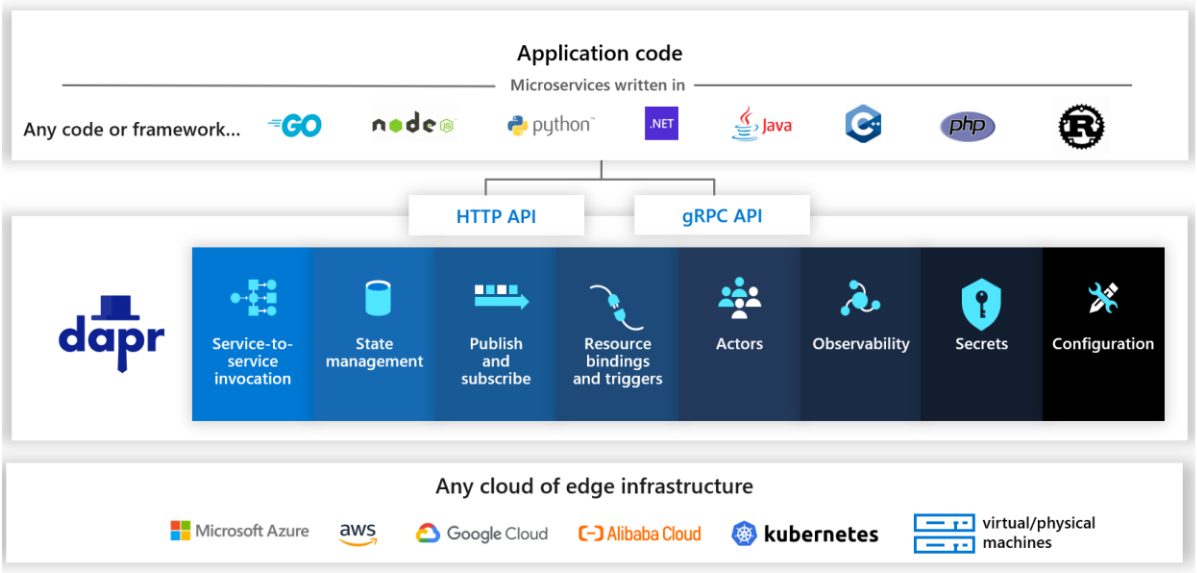

Figure 3 Dapr overview

Dapr consists of several Building Blocks, each of them providing specific functionality for applications to use, making the deployment highly configurable and encouraging a customizable approach. At his point in development, two of these blocks are identified as necessary to achieve the desired implementation of AGRICORE:

- **Service-to-service Invocation**: Resilient service-to-service invocation enables method calls, including retries, on remote services, wherever they are located in the supported hosting environment.
- **Publish and subscribe**: Publishing events and subscribing to topics between services enables event-driven architectures to simplify horizontal scalability and make them resilient to failure. Dapr provides at-least-once message delivery guarantee, message TTL, consumer groups, and other advanced features.

The Publish and subscribe component will receive the main focus, as it has been identified as a critical component for achieving the event-driven nature devised for AGRICORE's modules.

Lastly, let's remember that Dapr uses a sidecar architecture, which exposes an HTTP and gRPC API, either as a container or as a process alongside the application code, keeping the logic separated and allowing access from a multitude of runtimes.

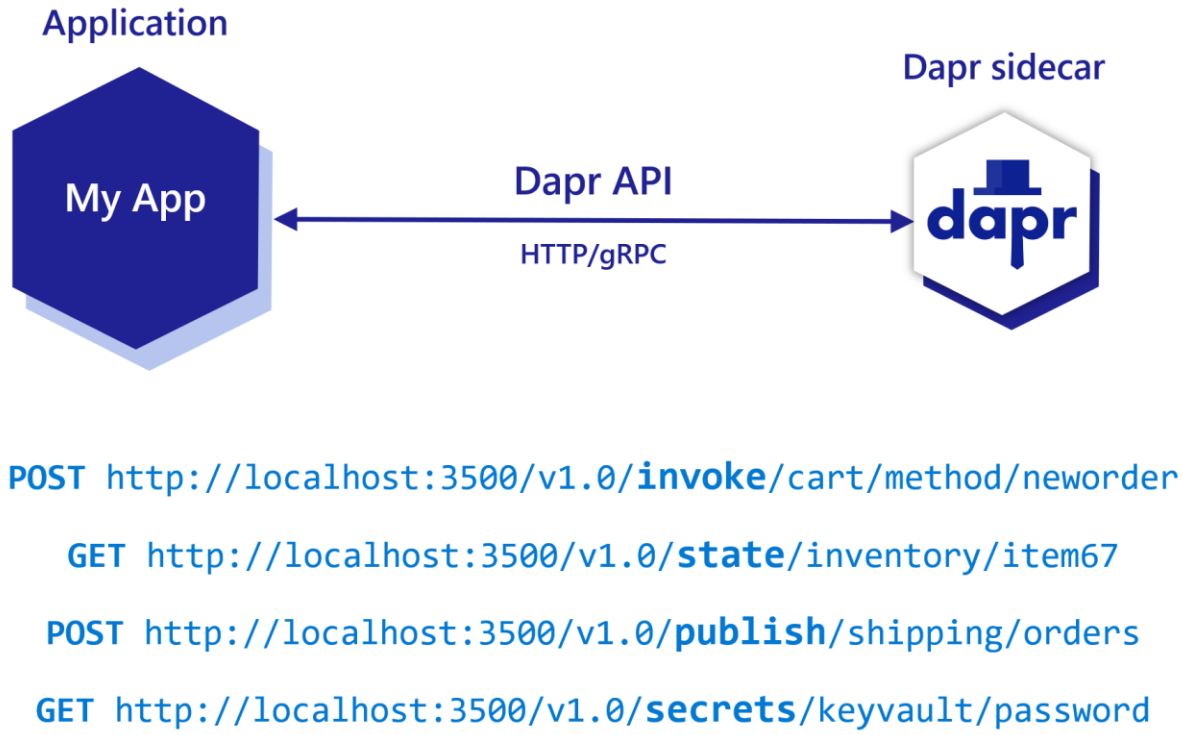

Figure 4 Dapr sidecar overview

4 Communication system implementation

4.1 Introduction

In this section, the details of the communication architecture developed will be presented, alongside a demonstration of the results on a simple scenario, that has the potential of being extended to cover the necessities of the AGRICORE modules.

4.2 Software development

Before continuing, it is worth noting that the current development has been carried out within the parameters set for Software Quality Assurance set in deliverable D6.6. GitLab has been used as the version control tool, and its mechanisms of issues/Merge Requests/Reviews have been used to ensure a quality development pipeline.

Therefore, all developments presented here are accessible for review and testing at the following GitLab repository, under the AGRICORE Open Source project:

https://gitlab.com/agricore/agricore-module-communication

The state of development as of this report's writing is tagged as v1.0.1 and accessible here:

https://gitlab.com/agricore/agricore-module-communication/-/tags/v1.0.1

To take advantage of GitLab's Continous Integration/Continuous Deployment tools, a CI/CD Pipeline has also been configured on branch master, which will build the images and upload them to GitLab's registry, allowing them to be accessible from other repositories. In the future, this will be used to point this project to the corresponding development repositories for all the modules, pull their source code from the registry, and use it to deploy the complete application, eliminating the need of storing source code on this same repository, keeping in line with software development's best practices.

4.3 Scenario description

For this demonstration of the developed infrastructure, the following scenario is considered, consisting of a possible interaction between the Agent-Based Modelling Simulation (ABM-e) and the Land Markets Module (LMM), when it comes to performing an auction of lands between the farmers. The process of execution will be as follows:

1. The ABM-e generates a list of all the farmers/agents interested in buying or selling lands.
2. The ABM-e publishes a message with this information via Dapr
3. The LMM, which is subscribed to that Topic, receives the message
4. The LMM parses the information and resolves the auction process, using its internal logic.
5. The LMM generates a list of the lands that change hands during the auction.
6. The LMM publishes the results of the auction via Dapr
7. The ABM-e receives the message, as it's subscribed to thisTopic.
8. The ABM-e processes the information and updates its internal data with it.

Thanks to the use of Dapr, the modules can abstract themselves from the details of message communication, such as networking, message queue management, etc.

4.4 Assumptions

For the purposes of this demo, simplified versions of the behaviours and capabilities of the ABM and LMM modules are presented, Market Modules development is specifically an objective of WP5, with Tasks 5.2 explicitly focusing on the Land Market Module, while the development of the ABM Simulation Module is contemplated on WP6, during Task 6.4. Therefore, it's not the objective of the current report to dwell more than necessary on these modules' internal implementation and functionalities.

This is why, when devising the demo scenario for this implementation, the following assumptions have been made:

1. The information flow contemplates a single ABM-e module, assuming that the information of all the agents comes from it. This would not be correct in the real AGRICORE scenario, where each agent present in the simulation would represent its own ABM-e instance, and, therefore, perform its own data management.
2. For the LMM, the logic governing the auction resolution has been ignored, as this only concerns its internal development.
3. Similar to point 2, the processing of auction results by the ABM-e instance has been left blank.
4. For simplicity's sake, the two modules present in the demo comprise a single Python script file. In a real scenario, it makes no difference for this system to execute a code composed of several modules, organized across different folders.

Please take note that, despite this simplification of the modules, the communication architecture and code devices used for the demo are perfectly valid to be applied to the more complex real scenario, and these assumptions do not harm in any way the validity or usefulness of the results presented in this report.

4.5 Environment configuration

As discussed in the previous section, a containerization environment is being used to deploy the modules and tools, with Docker-Compose as the arbiter. To achieve this, a docker-compose.yml file is defined, where the definition of each of the images to be built and included in the container needs to be written, specifying relations between them and other configuration parameters.

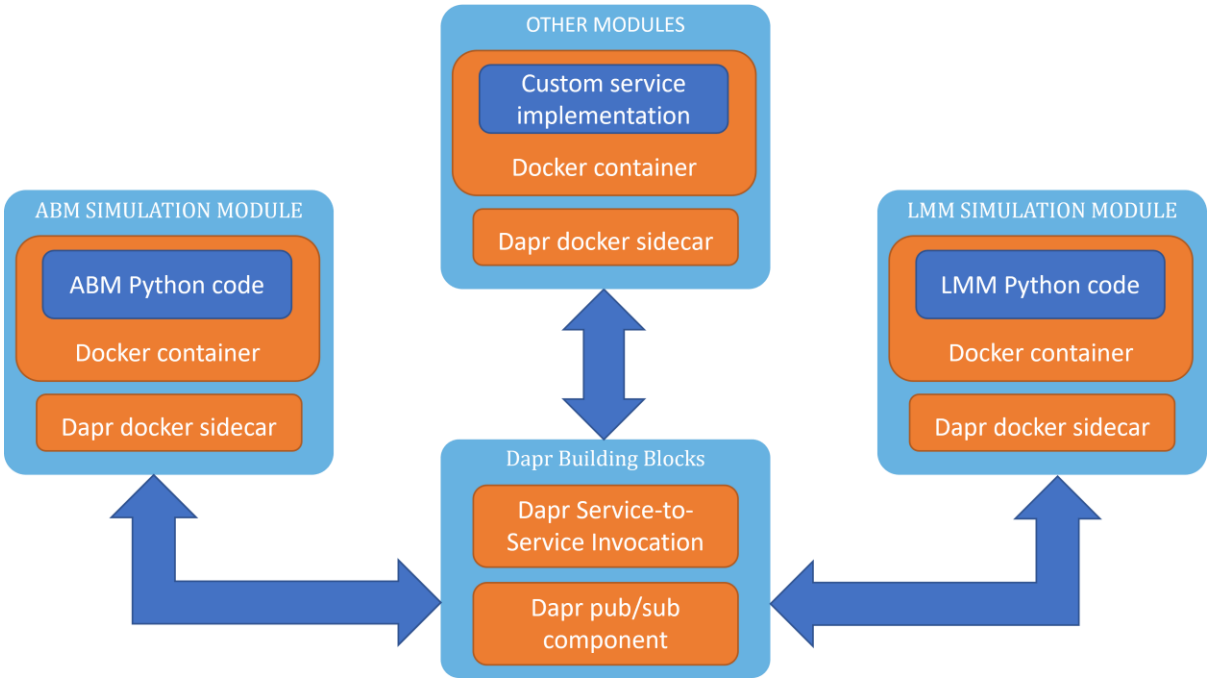

Figure 5 Communcation architecture using Dapr

Let's remember that, as explained in deliverable D6.1, and expanded upon in the previous section of this report, the architecture to enable a service to access the capabilities of the Dapr building blocks is to deploy it on a container alongside which a Dapr Sidecar image will be created, allowing the module to leverage Dapr capabilities by using that module's programming language Dapr SDK, which in the case of the developments for this deliverable has been Python. This schema is represented in the previous figure.

To achieve this, the following configuration in the docker-compose.yml file is used:

```
# Land market module - Python app
  agricore_lmm:
    container_name: agricore_lmm
    # The Docker image is built from local source code
    build: ../image/agricore_lmm
    image: agricore/lmm:0.2

  # LMM - DAPR sidecar
  agricore_lmm_dapr:
    container_name: agricore_lmm_dapr
    image: daprio/daprd:1.6.0
    # The dapr sidecar needs to be tied to its service network
    network_mode: service:agricore_lmm
    # Add dependency to ensure that the related container is already
    # running before starting the sidecar
    depends_on:
      - agricore_lmm
    command:
      - ./daprd
      - '-app-id'
      - agricore_lmm
      - '-app-protocol'
      - 'grpc'
      - '-app-port'
      - '50052'
      - '-components-path'
      - /components
      - '-config'
      - /configuration/agricore-dapr-config.yaml
    # Bind volumes are used to pass the configuration to the container
    volumes:
      - ../dapr/components/:/components
      - ../dapr/configuration/:/configuration
```

Code Block 1 docker-compose config for LMM + Dapr sidecar

With this code snippet, two elements are defined, the Docker image inside of which the module will run, which will be built for execution taking the source code, and its corresponding Dapr sidecar, which uses a precompiled image and will run with the configured parameters to serve the module.

To build the environment in which the modules will run, a Dockerfile is defined alongside it. This file will include instructions about how to build and execute this environment and code. For example, for LMM, the Dockerfile is as follows:

```
# syntax=docker/dockerfile:1
FROM python:3.8-alpine
WORKDIR /src
COPY . .
RUN apk update --no-cache
RUN apk add build-base
RUN apk add linux-headers
RUN pip install --no-cache --upgrade pip setuptools
RUN pip install -r requirements.txt
CMD ["python","lmm.py"]
```

Code Block 2 LMM Dockerfile

This Dockerfile defines that the module will be built over the base of a Python image from the official Docker repository. In particular, the 3.8-alpine version has been selected, meaning that the Python language used will be 3.8, to equal that used to develop the LMM and ABM-e modules, installed on an Alpine Linux distro, chosen for its light footprint. The following lines involve installing the minimum needed dependencies for building and running the code. Finally, the last line indicates how to execute the module.

Alongside the basic Service-to-Service Invocation functionality, the development will be using the Dapr pub/sub building block, to provide Messaging capabilities across services. To enable this, a Message broker needs to be added to this deployment. Dapr allows using several different messaging services, but RabbitMQ has been chosen due to its relative simplicity and lightweight implementation. Thanks to using Dapr, the messaging broker used is transparent to the modules, which allows plugging in and out different components if the need arises, without affecting the source code.

To include the RabbitMQ image, the following code is used in the docker-compose.yml file. Once again, a precompiled image running on Alpine has been selected.

```
rabbitmq:
    container_name: agricore_rabbitmq
    hostname: agricore_rabbitmq
    image: rabbitmq:3-management-alpine
```

Code Block 3 docker-compose Message Broker configuration

To activate the component for use, a configuration file is included in Dapr's components folder. The name field assigned here will be used across the code to access the message stream.

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: agricore_pubsub
  namespace: default
spec:
  type: pubsub.rabbitmq
  version: v1
  metadata:
  - name: host
    value: "amqp://agricore_rabbitmq:5672"
  - name: durable
    value: "false"
  - name: deletedWhenUnused
    value: "false"
  - name: autoAck
    value: "false"
  - name: reconnectWait
    value: "0"
  - name: concurrency
    value: parallel
```

Code Block 4 Pubsub component configuration

Complete configuration files can be found in the official GitLab repository.

4.6 Demo execution

To execute the presented demo navigate to the "docker" folder of the project structure and execute the following command, having installed the corresponding Docker dependencies for the host machine's Operating System beforehand (i.e. Docker Desktop on Windows systems):

```
docker-compose up
```

Code Block 5 Running the project

This will start building the containers, according to the configurations explained in the previous section. the Python modules are set to perform the information exchange described in the scenario setup. Using the Python Dapr SDK, the modules can easily access the functionality of the two Dapr Components configured, service-to-service invocation and messaging via RabbitMQ.

To expose a function from a service to be directly callable by another, the following code is used:

```
# Create and expose invocable method

@app.method(name='getSimulationUpdate')
def getSimulationUpdate(request: InvokeMethodRequest) -> InvokeMethodResponse:
    print(request.metadata, flush=True)
    print(request.text(), flush=True)
    return InvokeMethodResponse(str(status), "text/plain; charset=UTF-8")
```

Code Block 6 Expose an invocable method

To subscribe to a messaging topic, therefore receiving all messages posted to it by other modules, the following code is used (note the use of the previously configured pubsub_name):

```
# Message subscription

@app.subscribe(pubsub_name='agricore_pubsub', topic='LAND_AUCTION_RESULTS')
def processAuctionResults(event: v1.Event) -> None:
    auction_data = json.loads(event.Data())
    print(f'ABM received Auction results for', len(auction_data), 'lands that
changed ownership. ', flush=True)
    updateLandData(auction_data)
```

Code Block 7 Message topic subcription

To publish a message on a given topic, use the following code:

```
# Message publication

def publishAuctionResults(auctionResults):
    with DaprClient() as d:
        resp = d.publish_event(
            pubsub_name='agricore_pubsub',
            topic_name='LAND_AUCTION_RESULTS',
            data=json.dumps(auctionResults),
            data_content_type='application/json'
        )
```

Code Block 8 Message publication

Once the project is deployed, the modules perform the message publication and processing of results as expected.

```
LMM Starting
LMM received data  for 2 farmers. Starting auction process
```

Figure 6 LMM and ABM simplified output

```
ABM Starting
ABM received Auction results for 2 lands that changed ownership.
```

Complete source code for this demo can be found in the official GitLab repository

5 Conclusions

Deliverable D6.1 theorized that utilizing technologies such as Dapr for interfacing microservices could be a suitable replacement for a dedicated intercommunications module in AGRICORE. After studying the technologies and implementing a demonstration of the concept, this statement is believed to be true. Dapr provides a flexible and transparent enough work environment in which to set the intermodular communication, and the work presented in this report set the foundation for the deployment of AGRICORE in the future.

However, not everything is automatic in this approach, a minimal knowledge of the framework and the different SDK needed to leverage Dapr in the multitude of programming languages used across AGRICORE is needed. The demo performed here is a good starting point, but it needs to be complemented by offering support to the partners that request it in adopting these technologies.

For preparing this report, the following deliverables have been taken into consideration:

Deliverable Number	Deliverable Title	Lead beneficiary	Type	Dissemination Level	Due date
D6.1	AGRICORE architecture and interfaces	IDE	Report	Public	M23
D6.6	Software Quality Assurance measures for AGRICORE	AAT	Report	Public	M15