helyOS Developer Manual

Release 2.0

Fraunhofer IVI

Apr 23, 2024

CONTENTS

January 22, 2024

(version 2.0)

Fraunhofer IVI Zeunerstrasse 38 01069 Dresden Germany

CHAPTER

ONE

THE HELYOS FRAMEWORK

helyOS is a framework for accelerating the development of yard automation projects.

helyOS has a microservice architecture tailored to applications for managing autonomous vehicle fleets in delimited areas. It provides guidelines for the domain-driven design of applications in yard automation. The software core works as a ready-to-use backend for a control tower and is able to orchestrate microservices and dispatch assignments for complex missions.

1.1 Overview

helyOS is a framework for accelerating the development of yard automation projects.

In the helyOS framework, developers create and tailor autonomous driving applications by adding and combining microservices via a graphic interface or programmatically via the helyOS database. helyOS provides a clean architecture guide where microservices are ascribed to specific domains. Microservices of each domain remain decoupled, which allows teams to change, remove or add code without affecting other parts of the application.

1.1.1 Why you should use helyOS

Product Managers

- The helyOS architecture is scalable and resilient. This means there are fewer problems to resolve in production.
- It reinforces domain driven design. This means: clear responsibility boundaries between teams.

Software & Frontend Developers

- helyOS is reusable, it provides a ready-to-use backend that is configurable via dashboard.
- The helyOS GraphQL API gives full control of the application data to the frontend developer. It is not necessary to change any line of code in the backend to access the data in the frontend.

Motion Algorithm Programmers

- helyOS provides a ready-to-use environment to interact with robot vehicles. Programmers can focus on the development and evaluation of their motion algorithms.
- It is programming language independent, and you can host the algorithms in the computer or in the cloud, e.g., as a paid service.

Automation Engineers

- Complete flexibility for automated vehicles regarding assignment data formats.
- Flexibility regarding the degree of automation of agents.

- Embedded security mechanisms.

1.2 Definitions

1.2.1 Message broker

Message brokers are intermediary computer program modules in telecommunication or computer networks through which software applications communicate by exchanging formally defined messages.

1.2.2 RabbitMQ

It is the helyOS message broker that delivers the assignments to the agents and communicates with the autonomous domain.

1.2.3 Agent

A device that receives assignments. E.g., automated vehicles, cameras, traffic lights. . Agents can be moving devices such as automated vehicles, but also stationary objects, providing data such as cameras, traffic lights. Each agent must have a unique identifier code and be connected as a client to RabbitMQ.

1.2.4 helyOS core

The main component of the system, enclosing the business logicsand the orchestration of services and assignments to the agents. It can also work as the backend for web applications.

1.2.5 Assignment

A task or group of tasks delivered to the agent by the helyOS core via rabbitMQ. The agent must complete the entire assignment without the support of the helyOS core. The agent must report the assignment status as “running”, “completed”, “aborted” or “failed”. helyOS core can of course send a cancel request to the agent for terminating the assignment.

1.2.6 Mission

A mission consists of a group of one or more assignments, delivered to one single agent or to several agents. It usually originates from the client: The client creates a mission and helyOS uses the microservices to decompose the mission into several assignments. The microservices will also define the order of execution of the assignments. In principle, the client does not create assignments. Rather, the client creates missions and the microservices create assignments.

1.2.7 Instant Actions

Identical to the VDA5050 protocol, helyOS core can send instant commands from the user interface to the agents. These commands are independent of any assignment.

1.2.8 Microservices

The approach by which functionalities are implemented using small and independent services. Each one of the small services is provided by oneindependent server, maximizing decoupling and facilitating the development according to specific domains. The word microservices usually refers to "microservice architecture", in this documentation, however, we use the word to define a single-functionality service.

1.2.9 Autonomous domain (also automaton domain)

Corresponds to the robots, cameras, devices and agents connected to rabbitMQ. When the helyOS core dispatches an assignment to any member of the autonomous domain, it expects this assignment to be resolved inside this domain. helyOS core still can cancel assignments.

1.2.10 Yard

A delimited area where agents perform their assignments. It contains map objects.

1.2.11 Map object

The digital representation of any object or map layer inside the yard (space where the agents move): obstacle, parking areas, road, buildings, docking gates. The data format interpretation of these objects is the responsibility of the microservices and/or front-end.

1.2.12 Agent check in

The helyOS framework supports several yards. In real-world applications, the agent needs to authenticate itself every time it enters or switches the yard. This is done by the agent check-in.

1.2.13 Difference between mission and assignment

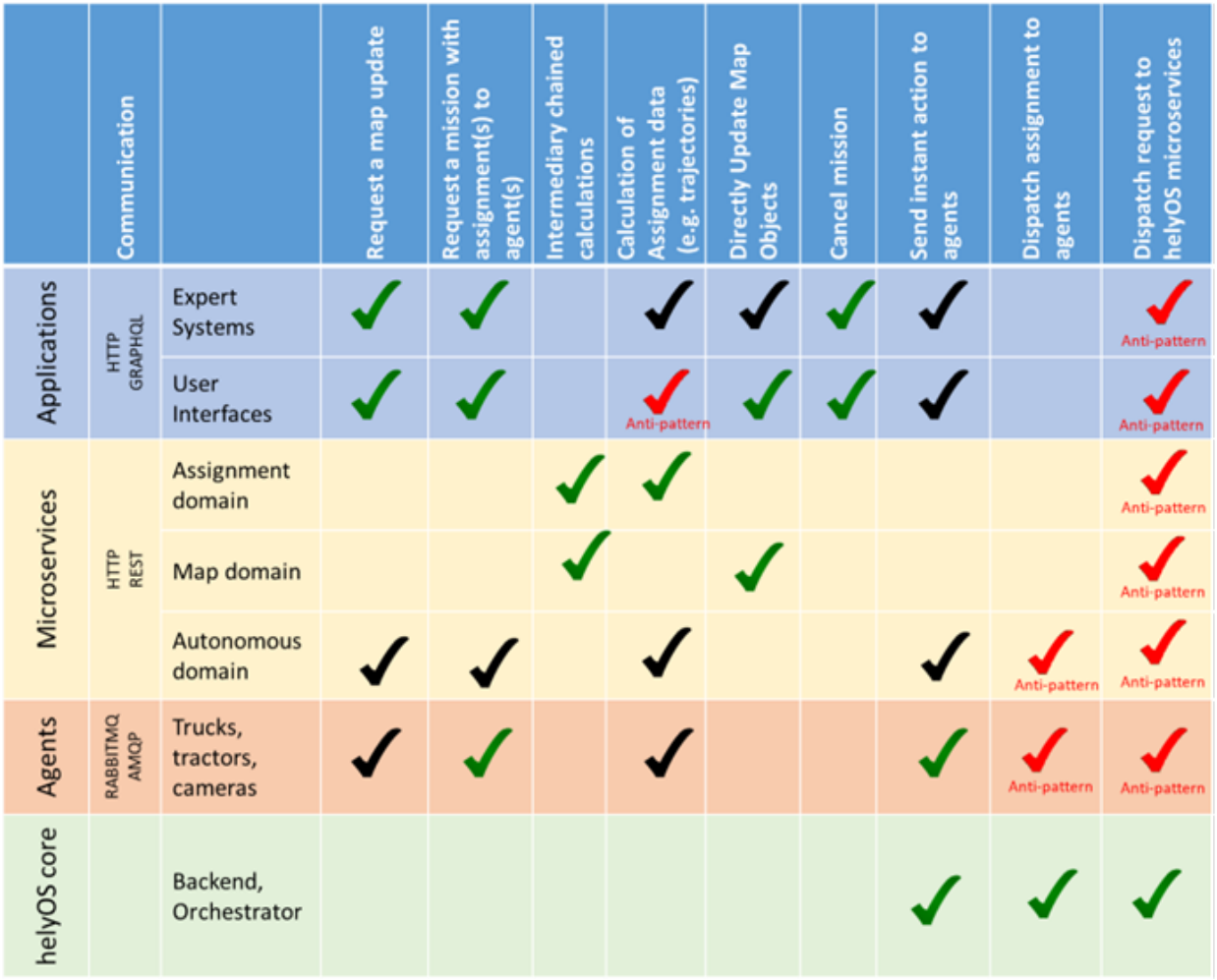
A yard automation application is defined in terms of missions. To complete a mission, agents must perform one or more assignments. helyOS receives the request of a mission and dispatches assignments to one or more agents.

1.2.14 What is possible in helyOS framework?

1.3 How It works

1.3.1 Software Components

The helyOS framework contains three software components:


Communication
Request a map update
Request a mission with assignment(s) to agent(s)
Intermediary chained calculations
Calculation of Assignment data (e.g. trajectories)
Directly Update Map Objects
Cancel mission
Send instant action to agents
Dispatch assignment to agents
Dispatch request to helyOS microservices
Applications
HTTP GRAPHQL
Expert Systems
User Interfaces
Anti-pattern
Microservices
HTTP REST
Assignment domain
Map domain
Autonomous domain
Agents
RABBITMQ AMQP
Trucks, tractors, cameras
helyOS core
Backend, Orchestrator

- **helyOS core**: a flexible backend software that receives mission requests from frontend apps and uses microservices to transform these requests into robot assignments.
- **helyOS Agent SDK**: a Python library used to connect agents (robots and vehicles) to helyOS core via rabbitMQ.
- **helyOS JavaScript SDK**: a JavaScript library used to create frontend apps that communicate with helyOS core.

  Each application built within the helyOS framework is shaped by connecting helyOS with function-specific microservices, e.g., routing, trajectory planning, map update, swarm behavior, etc.

1.3.2 Control Strategy

The helyOS framework embraces the principle of separation of concerns, featuring centralized command but distributed processes. It adopts a microservice architecture where expert subsystems are independently developed according to predefined domains. This approach complies with the use cases found in yard automation and smart farming and it is in line with the most modern practices in system integration.

In contrast to the traditional "leader-follower" approach, helyOS core does not micromanage assignments. It distributes the assignments including all necessary data upfront to the agent, who is part of the autonomous domain of the application. This allows the agent maximum independence while still retaining the possibility of exchanging data with expert systems inside the autonomous domain. This data exchange can be used, for example, in real-time corrections commands to support a running assignment. This approach favors the gradual development of more autonomous agents.

Note: Note that agents can also request missions for themselves from helyOS core. This feature can be exploited when the resolution of a given assignment is not possible inside the autonomous domain, for example, to overcome deadlocks or to emulate a leader-follower approach.

1.3.3 Data Formats

helyOS uses JSON formats. Except for a minimum set of required fields used to control the data flow, developers can freely choose the data structure of the assignment, map and sensor data. In the helyOS framework, most of the data formats are resulted from agreements between user interface programmers, mission planner developers and the robot controller developers.

1.3.4 Communication

helyOS uses the HTTP protocol to communicate with microservices and frontends, and RabbitMQ to communicate with the agents. helyOS uses RabbitMQ and consequently the AMQP protocol. AMQP originated from the financial industry, where it is used for the safe communication of financial data. Its main developers are Cisco, Red Hat, IONA and Twist. This protocol allows both *produce/consume* queue and *publish/subscribe* patterns with advanced routing features. RabbitMQ is also a reference for microservice architecture and the development of remote procedure calls.

1.4 Software Components

1.4.1 helyOS core

The helyOS core is a nodeJS software that works as a ready-to-use backend for control tower software in yard automation. It has five responsibilities:

1. Automated configuration of the rabbitMQ server to be used as message broker for yard automation (requires admin permission).
2. Collection of the yard state (obstacles, map, agent ids and positions, etc.).
3. Provision of an API endpoint for frontend apps for creating missions and accessing the yard state.
4. Orchestration of mission assignments that are sent to the vehicles via rabbitMQ. It is important to stress that the "orchestration" includes the conditional and ordered dispatch of assignments to each agent according to its reported state.
5. Orchestration of microservices used to calculate the assignment data.

   Source Code: available upon request.

1.4.2 helyOS JavaScript SDK

Web apps interact with helyOS via HTTP protocol using the GraphQL language. To accelerate the development, one can optionally use the helyOS JavaScript SDK, which wraps the GraphQL commands in convenient typescript functions. helyOS JavaScript SDK has the following functionalities:

1. create missions in helyOS core
2. cancel missions in helyOS core
3. send instant action commands to the agents via helyOS core
4. get yard and agent live data
5. get mission and assignment data
6. all setting commands available in the helyOS dashboard (requires admin permission)

   Source Code: https://github.com/FraunhoferIVI/helyOS-javascript-sdk

   Documentation: https://fraunhoferivi.github.io/helyOS-javascript-sdk/

1.4.3 helyOS Agent SDK

Agents (vehicles or robots) are connected to helyOS through rabbitMQ. The connection and communication with helyOS can be implemented using the **helyOS-agent-sdk** python package.

The helyOS-agent-sdk includes the following functionalities:

1. connection to rabbitMQ server
2. registration in helyOS yard (check in)
3. report sensors and agent states to helyOS core.
4. receive assignments and instant action commands from helyOS core

5. communicate with other agents and devices in the same RabbitMQ network

   Source Code: https://github.com/FraunhoferIVI/helyOS-agent-sdk

   Documentation: https://fraunhoferivi.github.io/helyOS-agent-sdk/build/html/index.html

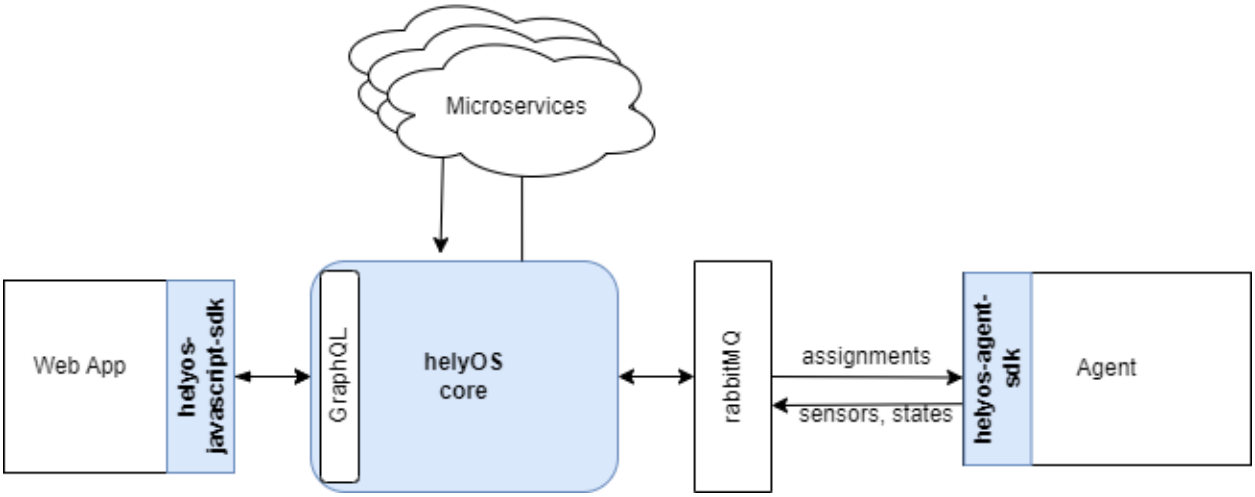


Fig. 1: helyOS framework

1.5 Data Formats

helyOS uses JSON formats. Except fora minimum set of required fields used by helyOS to control the data flow, developers can freely choose the data structure of the assignment, map and sensor data. In the helyOS framework, most of the data formats used in a project will result from agreements between user interface programmers, path planner developers and the agent controller developers.

1.5.1 Yard and map formats

helyOS is very flexible regarding map data structure. The map information is organized by yards, with each yard containing n **map objects**. Each map object has a JSON data field.

The map object data can contain any kind of information; from a simple polygon representing an obstacle to a complete Geo-JSON data structure representing a map layer. The definition of map objects and the alignment with other entities of a project lies within the responsibility of the developers.

- **helyOS yard**
  - id: database id, automatically generated.
  - uid: string that identify the yard.
  - name: name of the yard.
  - lat: latitude map origin.
  - lon: longitude map origin.

- alt: altitude map origin.
- map_data: (optional) JSON field containing relevant rendering information or metadata.
- data_format: (optional) name of the map data format. Example: "Trucktrix-Map"

An agent arriving at an automated yard must perform the check-in for that yard. The check-in procedure registers the agent to the helyOS system. The agent must provide the uid of the respective yard. helyOS returns the map origin as part of the Check-In-Response.

- **helyOS map object**

- yard_id: database id of the yard.
- name: name of the map object.
- **data**: JSON field (defined by the developer).
- type: string to identify the object.
- metadata: (optional) JSON field containing any relevant information.
- data_format: (optional) name of the map data format. Example: "Trucktrix-Map"

1.5.2 Assignment data format

helyOS is agnostic regarding the assignment data format. Nevertheless, the assignment must be enclosed in a JSON structure together with other fields that will help helyOS to route the assignment to the correct agent.

In the helyOS framework, the assignment is originated from a microservice [*]. That is, the developers must create a microservice that produces the assignment data. The microservice's response must have the following data structure:

Listing 1: The microservice response to create assignments.

```
HelyOSMicroserviceResponse {

    request_id?: string;  // auto-generated job id.

    status: "failed" | "pending" | "successful";

    results: AssignmentPlan[]; // array of assignments.

    dispatch_order?: number[][]; //␣
↪order in which the assignments will be dispatched to the agents.

}

AssignmentPlan {
    agent_id?
↪: number; // id of the agent that will receive the assignment.
    agent_uuid?
↪: string; // UUID of the agent that will receive the assignment.
    assignment:␣
↪any; // assignment data, usually defined by the agent vendor.
}
```

Note, in the inset, that the agent assignment data is nested into the field *results*. helyOS core will forward the *assignment* to the agent indicated by *agent_id* or *agent_uuid* . The agent will finally receive this assignment inside the following data structure:

Listing 2: Assignment received by the agent via RabbitMQ message.

```
AgentAssignmentCommand {

    type: string = "assignment_execution";

    uuid: string; // agent UUID.

    body: any␣
↪= **assignment** // the same assignment field in AssignmentPlan.

    metadata:␣
↪AssignmentMetadata; // automatic generated by helyos core.

}


AssignmentMetadata {
    id: number;
    yard_id: number;
    status: string = "to_execute";
    workk_process_
↪id: number; // mission identification in the helyOS database.
    context: any; // data from previous␣
↪assignments belonging the same mission (workk_process_id).
}
```

Check also helyOS and Agents section: *helyOS and Agents*

1.5.3 Agent data format

The agent data is saved in the database and can be updated via the helyOS Dashboard, or user interface, or by the agent itself via RabbitMQ

- **Agent Fields**

– uuid: universal unique identifier.

– available_operations: array of string defining the operations available for the agent.

– geometry: free JSON format defining the vehicle geometry.

– factsheet: JSON field added for compatibility with VDA 5050.

– x, y, z, orientations : x, y and z a numbers to specify the position of the agent. Orientations is a number array with information of the orientation of the first agent part, and of the joint angles for trailers.

– status/state: “not_automatable” | “free” | “ready” | “busy”

– sensors: JSON field containing any data about the agent: temperature, diagnosis data, assignment progress, velocity etc. The **helyOS-native sensor data format** allows the data be visualized in the helyOS dashboard. However, following this specification is

optional; the field sensors can hold any arbitrary data structure. The data format is imposed by the visualization app that the developer choose for reading it.

1.5.4 helyOS-native Sensor Data Format

The sensor data returned from an agent can have any format. The information is published in a RabbitMQ topic, and helyOS forwards the data to user clients via WebSocket. Therefore, the user interface must be aligned with the information and parse the sensor values.

However, if you wish the sensor values to also be visualized on the helyOS admin dashboard, then you must use the following format:

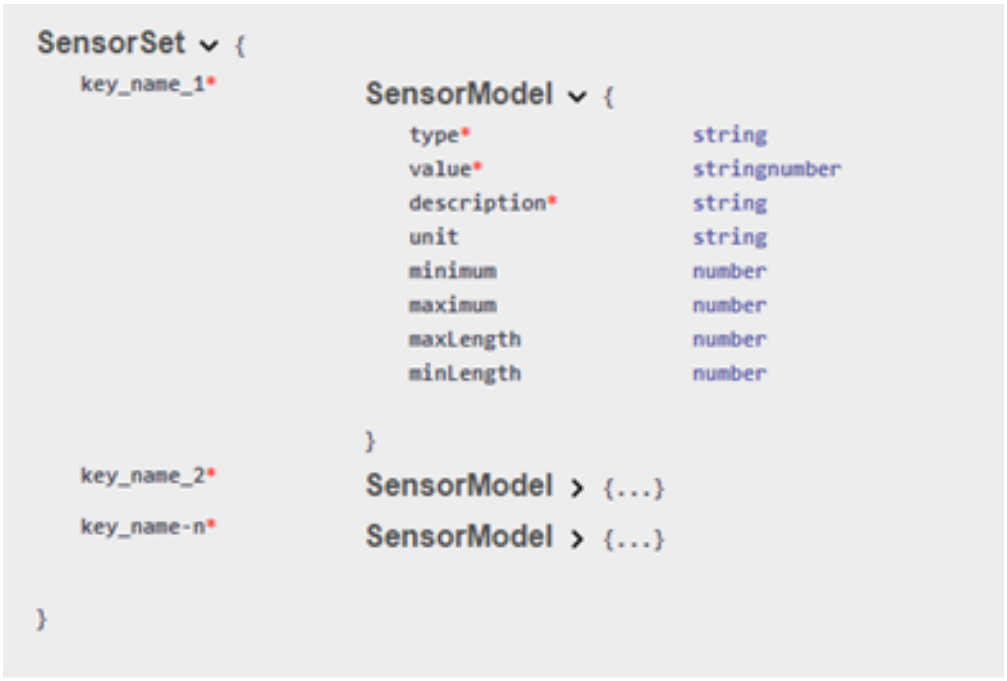


Fig. 2: Sensor data format

1.5.5 Mission request data format

To create a mission, the software developers must insert a row in the table of work processes. They can use the GraphQL language or the helyOS JavaScript SDK. Here again, helyOS does not specify the content of **data**.

```
{
    yardId: number,
    workProcessTypeName: string
    status: string
    agentIds: array of numbers
    waitFreeAgent: boolean
    data: {...}
}
```

The field data will be forwarded to all microservices linked to the mission given by the *workProcessTypeName*.

The follow fields are processed by helyOS core:

- **yardId:** Database id of yard.
- **workProcessTypeName:** One of the mission names previously defined in the helyOS dashboard (Define Missions view).
- **status:** 'draft' | "cancelling" | 'canceled' | 'dispatched' | "preparing resources" | "calculating" | "executing" | "succeeded". When creating, you can only define as 'draft' or "dispatched". When updating, you can only set the status as "cancelling" or "dispatched".
- **agentIds:** A list containing only the database ids of the agents taking part in the mission. This agents will be reserved by helyOS core.
- **waitFreeAgent (optional):** Default is true. It defines if helyOS must wait all agents listed in **agentIds** to report the status free before triggering the mission calculations. Set false if you don't need to reserve the agent and you can pile up assignments in the agent queue. Notice that this may produce assignments calculated with outdated yard data.

CHAPTER

TWO

HELYOS CONFIGURATION

helyOS core is the main component of the system, enclosing the business logics and the orchestration of assignments to the agents. It can also work as the backend for web applications. helyOS can be configured by editing its database or manually using the interactive dashboard.

2.1 Getting Started

Utilizing the Docker image provides the simplest method for running helyOS. T his image can operate locally or be deployed via a cloud provider. For convenience, all components and environment variables should be declared within the docker-compose.yml file

2.1.1 Environment Variables

Database connection

helyOS saves all the data in a PostgreSQL database. The database can be hosted in the same server as helyOS or in a different server.

- PGHOST: hostname for postgres.
- PGPORT: connecting port of the postgres.
- PGDATABASE: database name of your application. A postgres instance can host multiple databases.
- PGUSER: username for postgres database.
- PGPASSWORD: password for postgres.

GraphQL Interface

The GraphQL server is the main entry point for applications or use interfaces. GraphQL requests allow to query and mutate the data in the database. It also controls the authentitcation and authorization of apps in helyOS.

- GQLPORT: server port to query the GraphQL server (defaults to 5000).
- JWT_SECRET: secret key to encrypt the JWT token.

RabbitMQ connection

RabbitMQ is the message broker used by helyOS to communicate with the agents. helyOS core does not only publish and consume messages from the RabbitMQ server, but also creates the necessary accounts, topic exchanges and queues.

- RABBITMQHOST: hostname for rabbitmq server.
- RABBITMQPORT: connecting port for AMQP clients.
- RBMQ_API_PORT: configuration port for REST API for rabbitmq server. It is used to create the rabbitmq accounts.
- RBMQ_SSL= True or False. If True, the AMQP connection to rabbitmq server is encrypted using TSL.
- RBMQ_API_SSL= True or False. If True, the API connection to rabbitmq server encrypted (default = RBMQ_SSL).
- CREATE_RBMQ_ACCOUNTS: True or False. helyOS automatically creates the rabbitmq accounts
- RBMQ_ADMIN_USERNAME: rabbitmq admin username (required if CREATE_RBMQ_ACCOUNTS is True)
- RBMQ_ADMIN_PASSWORD: rabbitmq admin password (required if CREATE_RBMQ_ACCOUNTS is True)
- RBMQ_USERNAME: rabbitmq regular account username (defaults to RBMQ_ADMIN_USERNAME)
- RBMQ_PASSWORD: rabbitmq regular account password (defaults to RBMQ_ADMIN_PASSWORD)

(Optional settings)

- AGENTS_UL_EXCHANGE: exchange topic to send data to agents
- AGENTS_DL_EXCHANGE: exchange topic to receive data to agents
- CHECK_IN_QUEUE: rabbitmq queue name where agents must publish to perform check in.
- AGENT_UPDATE_QUEUE: rabbitmq queue name where high priority messages from agents are published.

helyOS settings

- ENCRYPT (not implemented yet): none | agent | helyos | helyos-agent. RSA encription betwenn helyos core and agent. This is an additional encription to the TSL layer used by the RAbbitMQ (default = none)
- MESSAGE_RATE_LIMIT: maximum burst of number of messages per second that an agent is allowed publish to helyOS. (default = 150)
- MESSAGE_UPDATE_LIMIT: maximum burst of number of database updates per second originated from messages publishing. E.g. status update messages. (default = 20)
- WAIT_AGENT_STATUS_PERIOD: time in seconds that helyOS waits for an agent to change to the required status before triggering a mission. (default = 20)

(Optional settings)

- AGENT_REGISTRATION_TOKEN: It is used to authenticate the agents that were not previoulsy registered in helyOS.
- MOCK_SERVICES: True or False. If True, the services are mocked. It is used only for automated tests purposes. (default = False)
- TLS_REJECT_UNAUTHORIZED: True or False. If True, the TLS connection is rejected if the certificate is not valid. (default = True)
- DEBUG: True or False. If True, the helyos core log is more verbose. (default = False)

2.1.2 Configuration Files

- **helyos_private.key** and **helyos_public.key**: RSA keys in PEM format. They are used to encrypt and sign helyOS messages sent to agent. They are located in */etc/helyos/.ssl_keys/*.
- **ca_certificate.pem**: This is the Certificate Authority (CA) that signed the RabbitMQ server certificate. It is located in */etc/helyos/.ssl_keys/*.
- **microservices.yml**: This file contains the initial configuration of the microservices. This data can be modified later in the dashboard. It is located in */etc/helyos/config/microservices.yml*.
- **missions.yml**: It serves as a blueprint for all available missions for the agents. It instructs the helyos core on how to orchestrate the microservices for each mission. This data can be modified later in the dashboard. It is located in */etc/helyos/config/missions.yml*.

Optional Database Customization

Any file with the extension **.sql* in the folder */etc/helyos/db_initial_data/* will be executed when the database is created. This can be used, in principle, to pre-populate the database with data, but also to create extra tables, views and functions that will be automatically available in the GraphQL server.

2.1.3 Example

Snippet of a *docker-compose.yml*

```
version: '3.5'
services:

    database:
        container_name: helyos_database
        image: postgres:13
        ports:
            - "5432:5432"
        volumes:
            - postgres_data:/var/lib/postgresql/data/
        networks:
            - control-tower-net

    helyos_core:
        image: helyos_core:2
        ports:
            - 5002:5002  # websocket
            - 5000:5000  # GraphQL
            - 8080:8080  # HelyOS Dashboard
        volumes:
            - ./my_folder/yard_map_data.sql:/etc/helyos/db_initial_data/yard_map_data.sql
            - ./my_folder/microservices.yml:/etc/helyos/config/microservices.yml
            - ./my_folder/missions.yml:/etc/helyos/config/missions.yml
            - ./my_folder/helyos_private.key:/etc/helyos/.ssl_keys/helyos_private.key
            - ./my_folder/helyos_public.key:/etc/helyos/.ssl_keys/helyos_public.key
            - ./my_folder/ca_certificate.pem:/etc/helyos/.ssl_keys/ca_certificate.pem

        environment:
```

(continues on next page)

(continued from previous page)

```
            # DATABASE
            - PGUSER=postgres
            - PGDRIVER=QPSQL
            - PGPASSWORD=${PG_PASSWORD}
            - PGHOST=helyos_database
            - PGDATABASE=my_application_db
            - PGPORT=5432
            -
            # RABBITMQ
            - RABBITMQHOST=rabbitmq.server.com
            - RABBITMQPORT=5672
            - RBMQ_API_PORT=15672
            - RBMQ_SSL= False
            - RBMQ_API_SSL= False

            # RBMQ ACCOUNTS
            - CREATE_RBMQ_ACCOUNTS=True #if helyOS creates the rabbitmq accounts
            - RBMQ_ADMIN_USERNAME=helyos_core
            - RBMQ_ADMIN_PASSWORD=${RBMQ_PASSWORD}

            # GRAPHQL
            - GQLPORT=5000
            - JWT_SECRET=${MY_SECRET_KEY}
        networks:
            - control-tower-net

        depends_on:
            - database
```

To run use the command: `docker-compose up`.

2.2 How to Configure helyOS Core as Backend

To set up helyOS as the backend for a yard automation application, the developer needs to perform the following tasks:

1. Connect helyOS to a running rabbitMQ server.
2. Access the helyOS dashboard.
3. Register the frontend applications using the "Register App" tab on the dashboard.
4. Define the yard within the "Yards" tab on the dashboard. A yard is the enclosed area that houses the autonomous agents, such as vehicles or robots.
5. Register the agents (vehicles or robots) in the "Register Agents" dashboard tab.
6. Specify the missions applicable to your application via the dashboard's "Define Missions" menu. Examples of missions. *"drive_from_a_to_b"*, *"seed_field"*, etc.
7. Register microservices in the "Microservices" view. They will be used to process the mission's request data and produce assignment data, e.g. a trajectory path for the vehicle.
8. Create "Mission Recipes", that is, associate each mission to one or more of the registered microservices.

   By using helyOS core as a backend, front-end developers can create the user interface (e.g. Graphana boards, web applications, etc.). They may use either **helyOS JavaScript SDK** (for web applications) or

the **GraphQL** language (for any kind of application) to create missions and access all data from the yard and the automated vehicles.

2.3 Admin Dashboard

The helyOS dashboard is a GUI that helps developers to set up helyOS and to debug the application. The default port to access the dashboard is 8080: http://localhost:8080.

Note: In principle, all helyOS configuration can be also done by directly writing the settings tables in the database via GraphQL interface (or SQL scripts); such an approach would be useful for automating the initial configurations for a deployment.

2.3.1 "Yards" View

In this view, the administrator will register the area where the automated agents (vehicles, robots) are confined, we call it *yard*. The agents must be connected (checked in) to one yard to perform a mission.

- **uid:** It is the responsibility of the administrator to provide an unique identifier to the yard. Not to be confused with the database id, which is automatically ascribed.
- **Yard type:** Any descriptive name chosen by the developer to define the characteristics of the yard (e.g farm_yard, parking_lot, port, etc).
- **Lat, Lon and Alt:** Geographic coordinates of the yard center: latitude, longitude and altitude. (WGS84 reference system)
- **Metadata:** User-defined JSON field containing any metadata relevant for the front-end application, e.g., map zoom level, visible map layers. It is optional.
- **Source:** Each yard has one or several map objects associated with. The source of the map objects can be a direct data input, a map microservice or a simple SQL initialization script. This field is used to specify the name of this source. It is optional.

2.3.2 "Register Agents" View

In this view, the administrator will register the agents (vehicles, robots) and check their status. **Remember that this can also be done by any software or script via graphQL.**

- **uuid:** It is the responsibility of the administrator to provide a unique uuid to the agent.
- **Name or plate:** Some friendly name to identify the agent.
- **Type:** Type of the agent: tractor, car, truck, etc.
- **Public key:** RSA public key, the correspondent private key will be saved in the agent. It is used to send encrypted messages to the agent.
- **accept assignments:** This field specifies wheather the agent can receive assignments.

Note: ATTENTION: Once the agent is connected to helyOS, the next fields may be constantly updated, overwriting any information that you input in the form.

- **Position:** Fields related to the position of the agent.

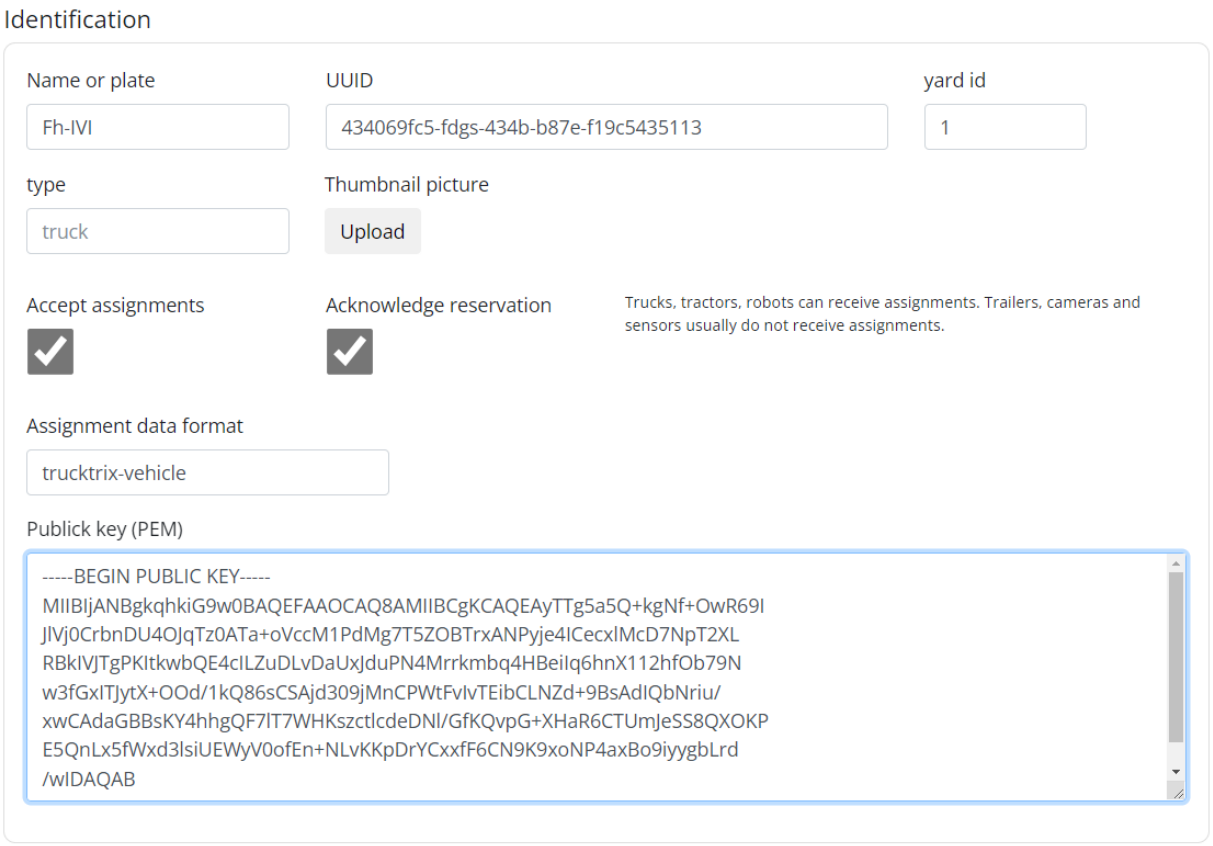


Fig. 1: Register agents view

Spatial Properties

The position parameters should be set according to the chosen reference in your application.

x (unit)	y (unit)	z (unit)	unit
-30000	0	13000	e.g. mm

orientation (mrad)	orientations (mrad)
0.329	0.329

geometry (JSON)

[{"id":1024,"axles":[{"id":0,"position":{"x":1120,"y":0},"steering":
{"axle_type":"forced","max_steer_angle":663,"steering_reference":"remote"},"tire_width":300,"tire_diameter":1044,"wheel_positions":[1160,-1160]},{"id":1,"position":{"x":6535,"y":0},"steering":
{"axle_type":"fixed"},"tire_width":300,"tire_diameter":1044,"wheel_positions":
[1160,1130,-1130,-1160]}],"width":2600,"height":3950,"length":9935,"chassis_position":
{"x":0,"y":0},"ground_clearance":300,"rear_joint_position":{"x":9935,"y":0},"max_front_joint_angle":0}]

State Properties

Status and connection fields are constantly updated by helyOS and agents. You can try to ascribe them a temporary value for debugging purposes.

connection	status
online	free

  - yard_id : The current yard that the agent is checked into.
  - x and y: Spatial coordinates of the agent: x and y.
  - orientation: Angle defining agent orientation.
  - orientations: In case of multi-part vehicles, one can use an array of angles: [1,0.2, . . .]

- **Connection and status:** Connection and work process status.
- **Geometry:** User-defined JSON field to specify the agent geometry information. This field can be overwritten by the agent at any time.

Other Options for Registering Agents

Public key folder
One option to register agents is simply adding their public keys in the folder */agent-pubkeys/* using the following convention for file name: *{uuid}.key*. The agent uuid and public key will be saved in the helyOS database, this is already enough to perform the check in. Other fields can be manually or automatically updated later after the check in.

Agent auto-registration
An agent is also able to register itself in the check-in procedure. For this, the agent should send the auto-registration token in the check-in message. The auto registration token is configured in helyOS core by using the environment variable AGENT_AUTO_REGISTER_TOKEN.

2.3.3 The "Define Missions" View

In this view, the developer will define the missions available for the software application. A mission represents a single task or a group of tasks. These tasks can be related to the calculation of a path, the storing of data, handling of map information, or a combination of all of above.

Note: Each registered mission can be seen as a new feature in the final application.

- **Name:** Name of the mission, that will be later used by the *Client* to trigger this kind of mission. E.g. "park_car", "seed_field", "drive_from_A_to_B".
- **Description:** Text documenting the mission goals and the used microservices.
- **Maximum agents:** Indicates the maximum number of agents handled by this mission.
- **Settings:** User-defined JSON field where the developer can pass fixed parameters to the user application or to all microservices used in this mission. It appends the field "_settings" in the MissionRequest.

The missions trigger one or more microservices. The sequential order of microservices is defined in the Mission Recipes view. That is, the Mission Recipes teach helyOS how to **orchestrate** the microservices to implement the desired mission.

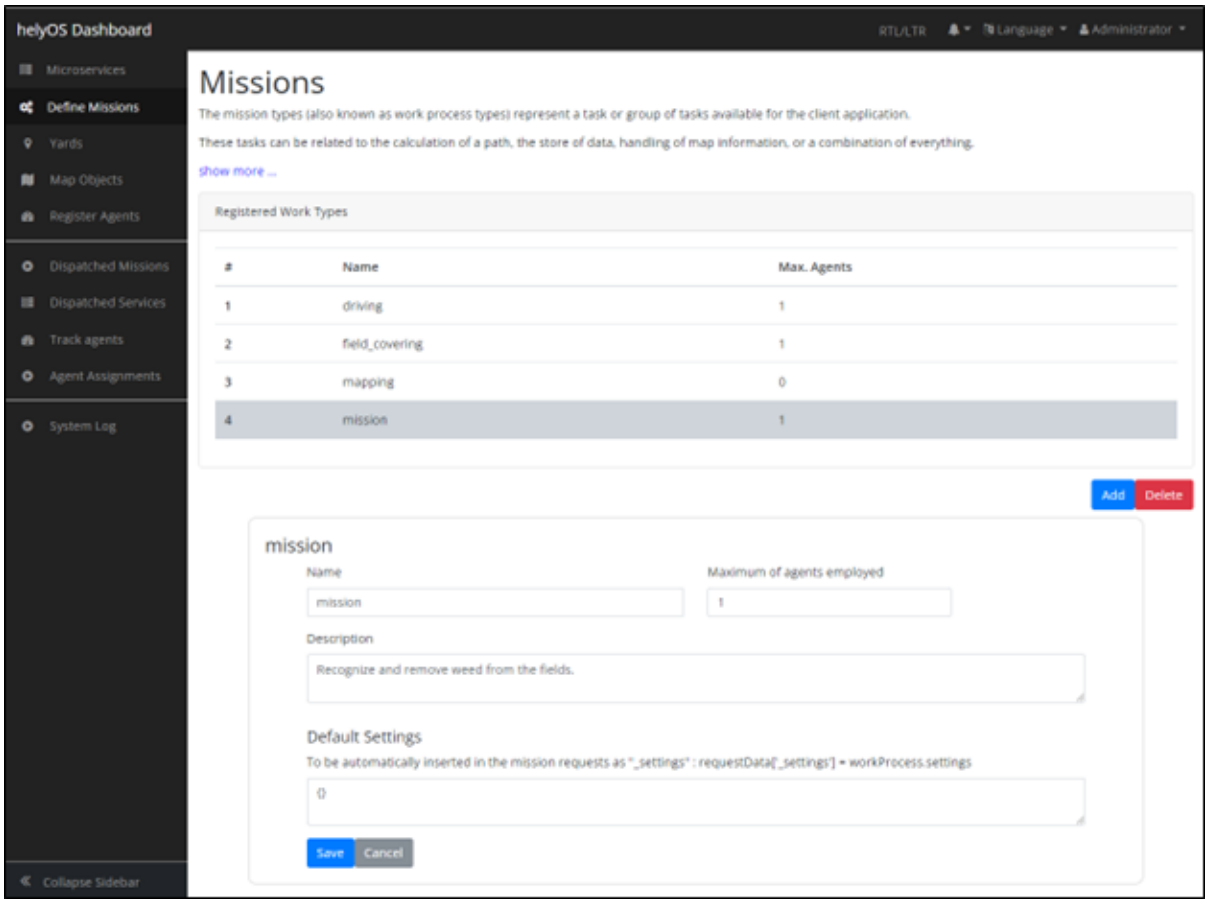


Fig. 2: Define missions view

2.3.4 "Microservices" View

In this view, the developer registers the microservices employed in the missions. Each microservice must belong to one of the three available domains:

Assignment domain:
microservice responses are interpreted as assignment to an agent.

Map domain:
microservice responses are interpreted as updates for the map objects.

Storage domain:
microservice does not respond with relevant data, the request is only used to push data to an external storage server and return the request status (2XX or 4XX).

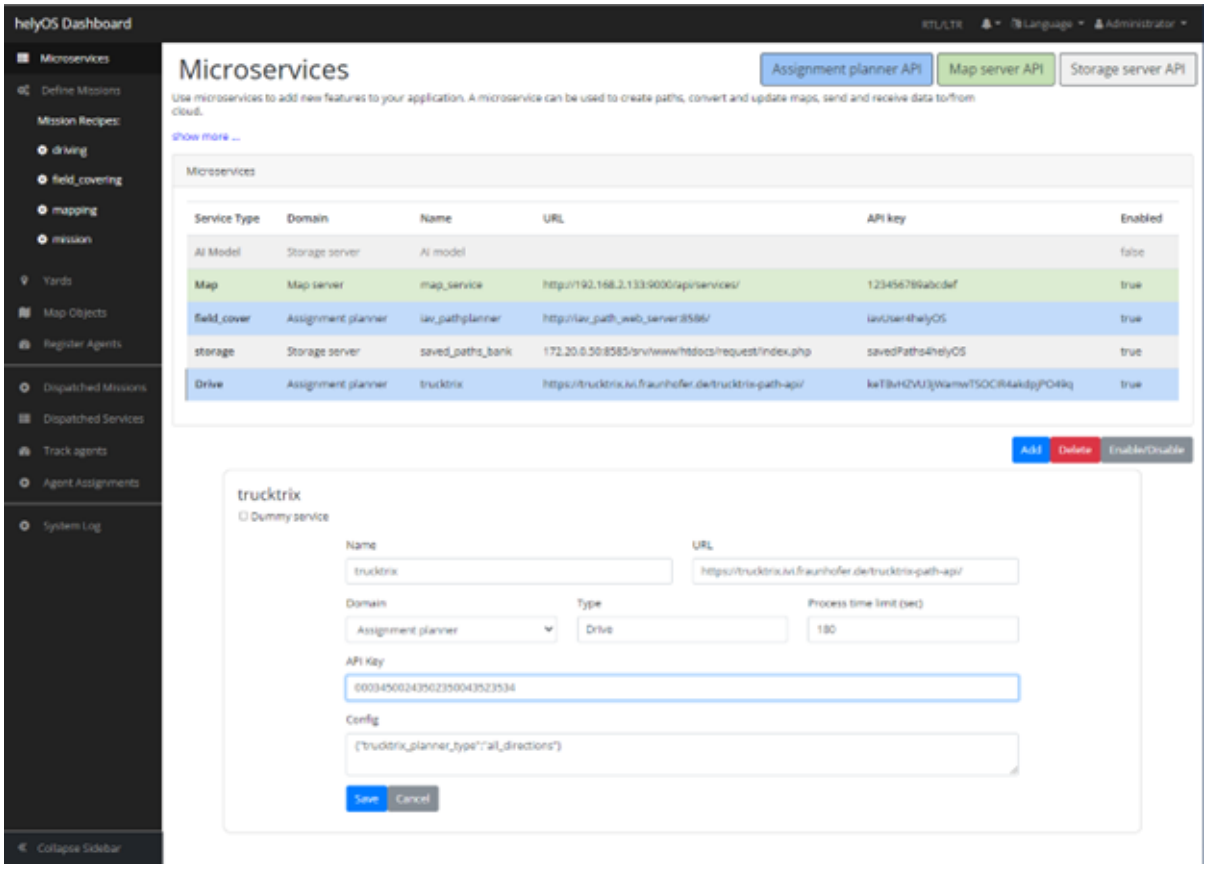


Fig. 3: Microservices view

When registering the microservice the following information is required:

- **Name:** Identify the microservice
- **URL:** Complete URL address, including http or https prefix and the port suffix.
- **Domain:** Choose between *Assignment*, *Map* or *Storage* domain.
- **API key:** Token used to authenticate the request call. It will be added to the request headers under the key *Authorization*.
- **Enable/Disable button:** Enables/disables a microservice.

- **Type:** Any word chosen by the developer to define a class of functionality for the microservice (e.g field_planner, driving_planner). This word is important because it will be used later to define a mission. Many microservices can have the same Type, but only one of them can be enabled at a given time.
- **Process time limit:** Maximum amount of time the system will wait for the microservice result. Not to be confused with the HTTP request timeout, used in the long poll approach. helyOS uses periodic polls spaced by 5 to 10 seconds to get the microservice results.
- **Config:** User-defined JSON field where the developer can pass fixed parameters on to the microservice

```
{
    request*   MissionData
    config       {...}
    context*    HelyOSContext
}
```

Request body sent to microservices. *request* is defined by the software developer according to the application. config is set in the dashboard and context contains the yard state and the response of the previous chained microservice. The yard state contains all the map object and agent ids and positions at the moment of the service request.

The Dummy Service

When a microservice is marked as dummy, helyOS will not send requests to any URL. Instead, helyOS will just copy the mission request data to the result field of the microservice. This is useful in the scenario where the application does not need to perform any calculation in microservices, e.g., if pre-defined assignment or map updates are already stored in the client. For example, if the dummy service was registered in the assignment domain, the *Client* can directly send the assignment data to the agent. If it was registered in the *Map* domain, the request data will be directly used to update the map objects.

2.3.5 "Missions Recipes" View

In this view the developer will decompose the previously registered mission into microservice calls. This is done by adding rows to the "Service Matrix" (click Add button). Each row corresponds to a step in the mission process and is used to orchestrate the microservice calls.

- **Step:** Give a name to your step, using a single word or a letter. Each step within a recipe must be unique.
- **Service Type:** It defines which microservice will be used in the step. The step will call the enabled microservice of the given "Type". The "Type" is defined when the microservices are registered. Note that only one microservice of a given "Type" is enabled.
- **Service Response:** If the microservice called in the step will produce an intermediate result in a chain of microservice calls, the option "intermediate step" should be marked. If the microservice response contains assignment or the map update data ready to be executed, the option "apply step result" should be marked.
- **Request Order:** : The order in which the requests will be dispatched. Note that the microservice responses can return in any order, since the services are asynchronous. If you want to ensure that the order of the microservices responses reflects the order of request dispatches, you must set the "Step Dependencies".
- **Step Dependences:** Define dependencies on other steps (microservices). For instance, if step "C" depends on step "A" and "B", the microservice associated with step "C" will be executed only after the responses of steps "A" and "B" are received. The responses of steps "A" and "B" will be automatically appended in the context of the step "C" request.

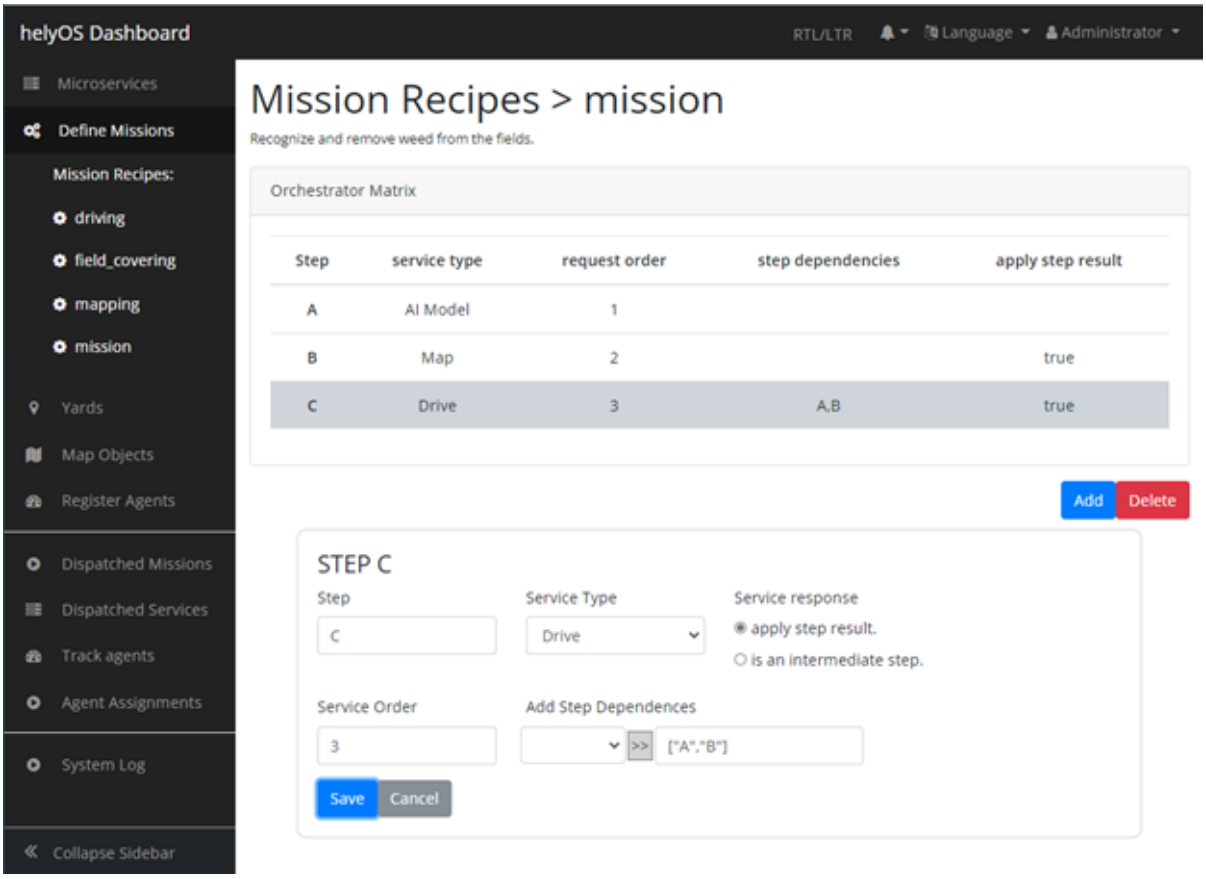


Fig. 4: Mission recipes view

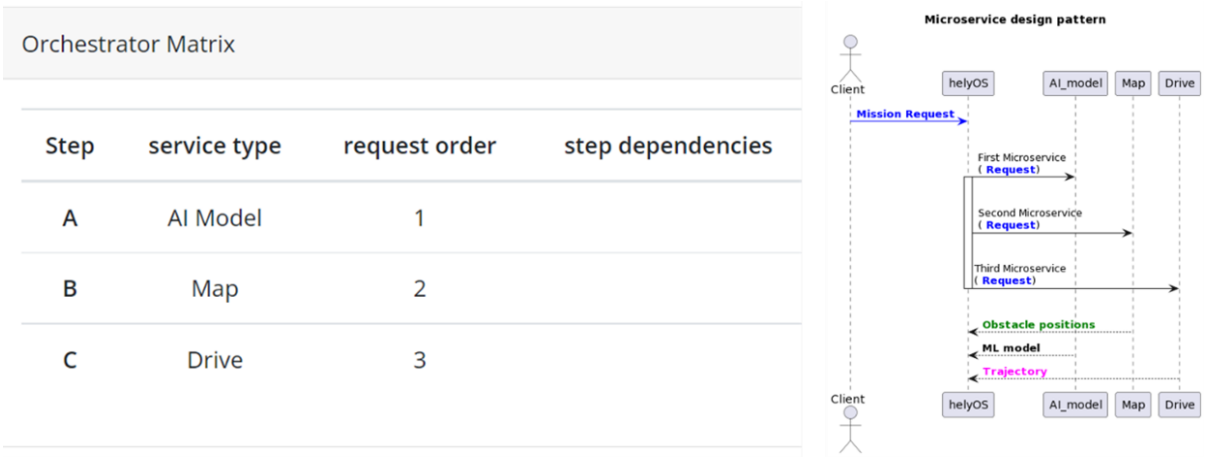


Fig. 5: **Example 1.** No dependencies between steps: All the microservices respond asynchronously.

Orchestrator Matrix

Step	service type	request order	step dependencies
A	AI Model	1	
B	Map	2	A
C	Drive	3	B

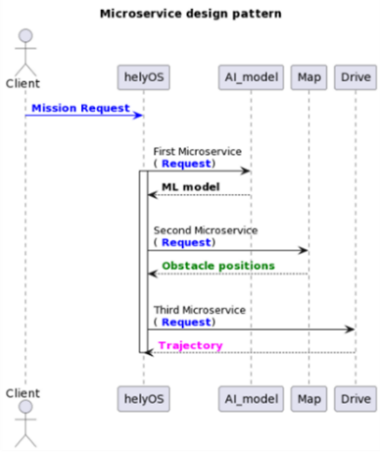


Fig. 6: **Example 2.** Dependencies between steps: Microservices are called and respond sequentially.

CHAPTER

THREE

HELYOS AND CLIENT APPS

helyOS core responds to Postgres database events. This means that the creation, update or deletion of rows in the database trigger actions inside helyOS core. Therefore, the client applications communicate with helyOS core by interacting with the helyOS database. This interaction uses the GraphQL language.

3.1 Application accounts

There are three types of helyOS accounts: the admin, application, and visualization accounts. They are authenticated by token authentication.

Each account type is ascribed to a specific database role in Postgres. Therefore, the permission set will be controlled at the database level. The admin accounts have permission to read and write all tables and execute all DB procedure functions. The application accounts are able to read all the tables but write only on a few of them. The visualization accounts can only read tables.

Further tuning of permissions or creation of new account types must be defined by an outer software layer for a specific software application.

3.2 Communication

The communication between helyOS and user apps uses the GraphQL language. GraphQL is designed to make database APIs flexible and developer friendly. As an alternative to REST, GraphQL lets developers construct requests that pull and change data from multiple tables. GraphQL is the default language to interact with helyOS-based applications and to access the yard state.

GraphQL queries and mutations can be tested using the GraphiQL developer interface:

http://localhost:5000/graphiql

For example, a GraphQL request to create a mission in plain python would be written as:

```
import requests

# Login: request to the Authorization token should come here
...
#

url = "http://helyo_server:5000/graphql"
headers = {"Content-Type": "application/json; charset=utf-8",
        "Authorization":"Bearer eyJhbGciOiJI... "}
```

(continues on next page)

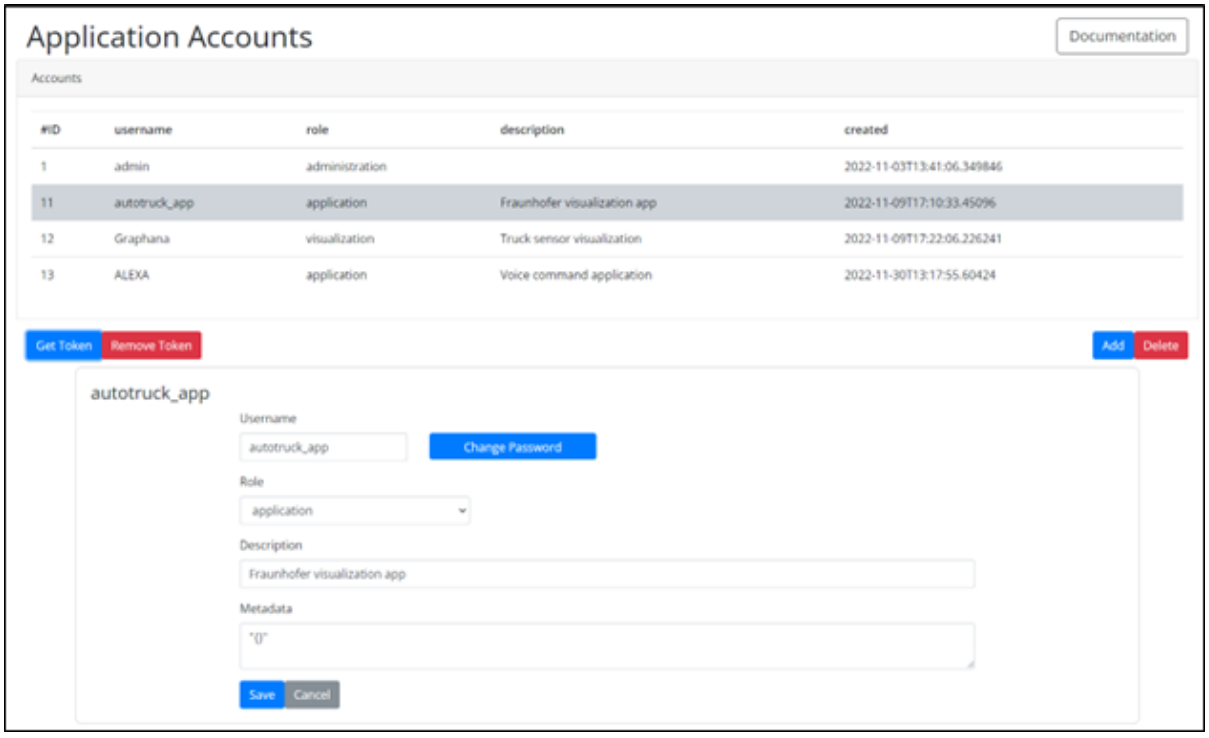


Fig. 1: Application accounts view

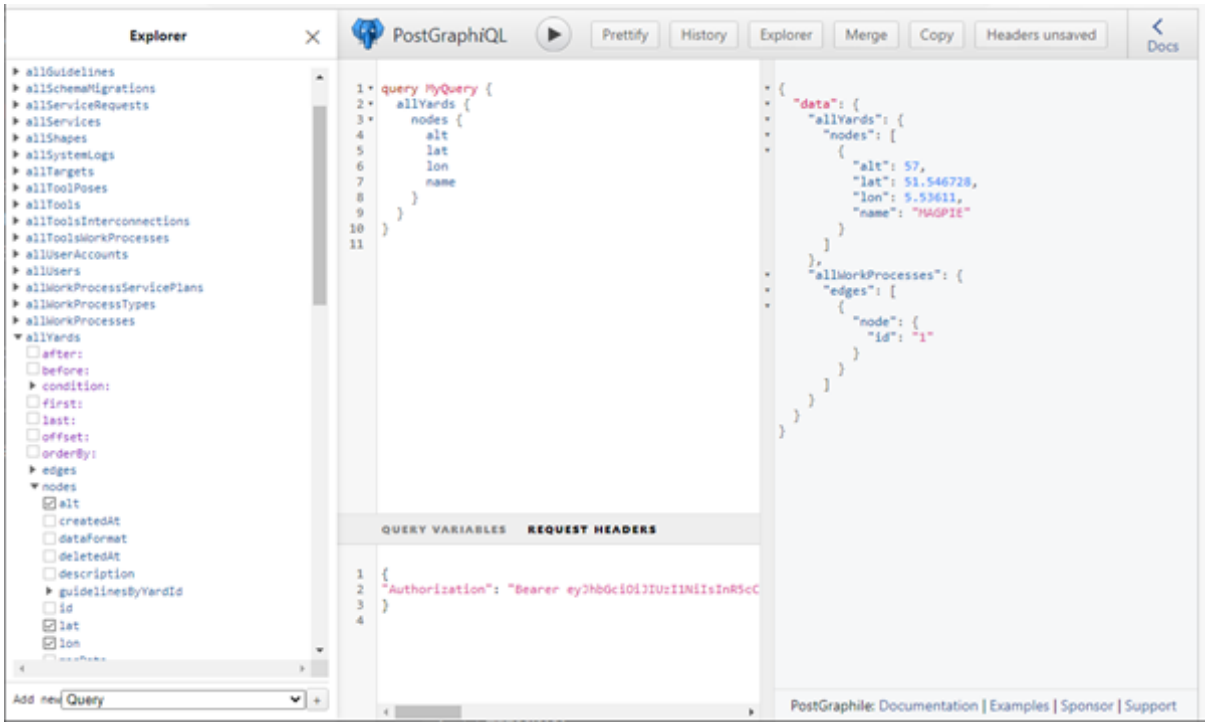


Fig. 2: GraphiQL view

(continued from previous page)

```
body_data = {

"operationName": "createWorkProcess",

"query": """mutation createWorkProcess($postMessage: CreateWorkProcessInput!)
            {
                createWorkProcess(input: $postMessage) {
                    workProcess {id, status }
                }
            }
        """,

"variables": {"postMessage" : {"clientMutationId": "not_used",
                                    "workProcess": {
                                        "status": "draft",
                                        "workProcessTypeName": "driving",
                        "data": "{}",
                                        "agentIds": [1]
                                    }
                                }
                }
}

response = requests.post(url, headers=headers, json=body_data)
```

Using GraphQL python libraries, this can be written more concisely. By using our helyOS- Javascript SDK *helyOS-javascript-sdk <https://github.com/FraunhoferIVI/helyOS-javascript-sdk>*, the above code becomes:

```
import { HelyosServices } from 'helyosjs-sdk';
const helyosService = new HelyosServices('http://localhost', {socketPort:'5002',
                                                              gqlPort:'5000'});
const createNewMission = () =>  helyosService.workProcess.create({
                            status: 'draft',
                            workProcessTypeName: 'driving',
                            agentIds: [1],
                            data: {},
                        });

helyosService.login("username", "password")
.then( response => helyosService.connect())
.then( connected =>  createNewMission());
```

A more advanced example with position tracking:

```
import { HelyosServices } from 'helyosjs-sdk';

const helyosService = new HelyosServices('http://localhost',
                                        {socketPort:'5002', gqlPort:'5000'});

function createNewMission() {
    console.log("==> Creating drive mission...");
    const trucktrixPathPlannerRequest = {   x:-24945.117347564425,
```

(continues on next page)

(continued from previous page)

```
                                        y:12894.566793421798,
                                        anchor:"front",
                                        orientation:1507.1,
                                        orientations:[1507.1],
                                        tool_id:1,
                                        _settings:{},
                                    };

return helyosService.workProcess.create({
                            agentIds: [1],
                            yardId: 1,
                            workProcessTypeName: 'driving',
                            data: trucktrixPathPlannerRequest as any,
                            status: 'dispatched',// status = 'draft' will save the mission
                                                 // but no dispatch it.
    });
}

function trackVehicle() {
console.log("==> Tracking agent position and assignment status...\n");

helyosService.socket.on('new_agent_poses',(updates: any)=>{
const agentData = updates.filter(( agent:any) => agent.agentId === 1);
    console.log(agentData);
});

helyosService.socket.on('change_work_processes',(updates:any)=>{
const wprocessStatus = updates.map((wprocess:any) => wprocess.status);
    console.log(wprocessStatus);
    if (wprocessStatus.includes('succeeded') || wprocessStatus.includes('failed') ) {
        process.exit();
    }
});

}

helyosService.login("username", "password")
.then( response => helyosService.connect())
.then( connected => {
    console.log("==> Connected to helyOS")
    createNewMission()
    .then(() => trackVehicle())
});
```

3.3 Requesting Missions

To trigger a mission in helyOS core, a new instance of the *WorkProcess* must be created. The *WorkProcess* is a generic process entity that can be used to create any kind of mission.

The *WorkProcess* has the following fields:

- *id*: The unique identifier of the mission.
- *status*: The status of the mission: *draft*, *dispatched*, *executing*, *succeeded*, *failed*, *canceling*, *canceled*.
- *workProcessTypeName*: The type of the mission. The type of the mission is defined by the *WorkProcessType*.
- *data*: The data of the mission. The data of the mission is a JSON object that contains the parameters of the mission.
- *agentIds*: The agents that are assigned to the mission. The agents are defined by the *Agent* type.

To ensure that the mission is executed correctly, the *WorkProcess* must be created with the following status:

- *draft*: The mission is created but not dispatched. This state is useful to save the mission in the database and make modifications before dispatching, or to add it to a *MissionQueue*. To dispatch the mission, the status must be changed to *dispatched*.
- *dispatched*: As soon as a *WorkProcess* is created with the status *dispatched*, or the status of a *WorkProcess* is changed to *dispatched*, helyOS will trigger a series of events to execute the mission. These events include the reservation of the agents, the calls to relevant microservices, and finally the dispatch of assignments to the agents.

3.4 Handling the Mission Execution

While the mission assignments are being actively executed, the status of the *WorkProcess* will be *executing*. If one of the mission assignments fails, the status of the *WorkProcess* will be *failed*. If all the mission assignments succeed, the status of the *WorkProcess* will be *succeeded*.

If the client application wants to cancel the mission, the status of the *WorkProcess* must be changed to *canceling*. This will trigger the canceling of all pending microservice requests, the canceling of all running and pending assignments, and it will signal the release of all agents from the mission. Only after the succession of all these events, the *WorkProcess* will be automatically changed to *canceled*.

Reguarding the state flow of the *WorkProcess*, the client application should not forcefully change the status of the *WorkProcess* to *executing*, *succeeded*, *failed*, or *canceled*. Usually we have the following state flow:

- The *dispatched* is automatically changed to *executing* or *failed*.
- The *executing* is automatically changed to *succeeded* or *failed*.
- The *canceling* is automatically changed to *canceled*.

Note: In principle, you can cancel an individual assignment by changing is status to *canceling*, this will result in a cancel instant action sent directly to the agent. However, this may lead to an inconsitent state of the mission, therefore is recommended to cancel the mission, as described above, rather than cancel individual mission assignments.

CHAPTER

FOUR

HELYOS AND MICROSERVICES

The communication between helyOS and microservices is configured in the helyOS dashboard. Every time helyOS receives a mission request, it makes calls to the relevant microservices. If the microservice results are not immediately available, helyOS will poll the results in subsequent periodic calls. The microservices must serve endpoints according to the helyOS definitions for microservices APIs.

4.1 Missions Request: from App to helyOS core

In order to initiate a mission, the applications are required to add a new entry into the work processes table of helyOS Postgres database. This can be accomplished using either the GraphQL language or its Javascript wrapper, *helyosjs-sdk*. The following example shows how to create a mission using the Javascript SDK.

Listing 1: Example of mission creation using the Javascript SDK. The mission type is "driving" and it employs the agent with id=1.

```
import { HelyosServices } from 'helyosjs-sdk';
import { token } from './token';

const backendUrl = 'http://localhost';
const socketPort = 5002;
const gqlPort = 5000;

const main = async () => {
    const hellosService = new HelyosServices(backendUrl, {socketPort, gqlPort}
↪);
    hellosService.token = token;
    await hellosService.connect()
    hellosService.workProcess.create({
                        yardId: 1,
                        workProcessTypeName: 'driving',
                        status: 'dispatched',
                        agentIds: [1],
                        waitFreeAgent: true,
                        data: { ... } // user-defined JSON field

                    });
}

main();
```

Once the workProcess (mission) is created, the following fields are processed by helyOS core:

- **yardId:** Database id of yard.
- **workProcessTypeName:** One of the mission names previously defined in the helyOS dashboard (Missions Receipe view).
- **status:** 'draft' | "cancelling" | 'canceled' | "dispatched" | "preparing resources" | "calculating" | "executing" | "succeeded". When creating, you can only define as 'draft' or "dispatched". Once the status is set as 'dispatched', the helyOS will prompt the execution of the mission. When updating, you can only set the status as "cancelling" or "dispatched".
- **agentIds:** A list containing only the database ids of the agents taking part in the mission. This agents will be reserved by helyOS core.

The following field is not processed by helyOS core, and it is forwarded to the microservice(s):

- **data:** The mission data, a user-defined JSON field which is specific to the application. This field will be forwarded to the microservices. The microservice will receive the mission data from the client software along with the yard state from helyOS core (helyOSContext). The developer must therefore add here any necessary information that is not present in the yard state. For example, pointing out the agent Id that will receive the assignment, or the ordering that the assignment must be executed.

The following field is optional:

- **waitFreeAgent (optional):** Default is true. It defines whether helyOS must wait for all agents listed in agentIds to report their status as "free" before triggering the mission calculations. Set false if you don't need to reserve the agent and you can pile up assignments in the agent queue. Notice that this may produce assignments calculated with outdated yard context data.

4.2 Service Requests: from helyOS to Microservices

Once the mission is triggered, helyOS will dispatch HTTP POST requests to the related microservices; one mission can trigger the request of one or many microservices. The order in which the microservices are called is pre-configured as mission steps using the mission recipe editor in the *helyOS dashboard*; **each step is associated to one microservice call.**

All these requests contain the field **request** with the mission data, and the field **context** with the yard, agents and mission-related data. As default, the **request** field contains the user-defined **workProcess.data**.

Listing 2: Requesta data send from helyOS to the microservice.

```
HelyOSRequest  {
    request: any;  // mission input data from the application

    context: HelyOSContext;

    config?: any;  // optional configuration data defined in the helyOS␣
↪dashboard.

}
```

The **context** contains all information relevant at the moment of the dispatch, including mission orchestation data, yard data and calculation results from previous steps from other microservices in **context.dependencies**.

Listing 3: Context data autommatically generated by helyOS and sent to the microservice.

```
HelyOSContext  {

    agents: AgentModel[]; // arrray of agents relevant to the mission.

    map: {
                id: number,
                origin: { lat: number, long: number}
                map_objects : MapObjectsModel[]; // array of all map objects␣
↪in the yard.
    };


    orchestation: {
                current_step: string, // name of the current step in the␣
↪mission.
                next_step: string[],  // names of the subsequent steps in the␣
↪mission.
    };

    dependencies: {
                step: string,
                requestUid: string,
                response: any
    }[]; // array of data responses from microservice of previous steps.

}
```

The *AgentModel* and *MapObjectsModel* are defined here: *Models Description*.

4.3 Service Response: from Microservices to helyOS

In general the microservice response is a JSON object with the following structure:

Listing 4: Response data structure as defined in the Assignment planner API.

```
HelyOSMicroserviceResponse {
    request_id?: string;  // generated job id. Can be used to poll results␣
↪from long running jobs.

    status: "failed" | "pending" | "successful";  .

    results: AssignmentPlan[] | MapUpdate | any;

    dispatch_order?: number[][];

    orchestration?: {
              nex_step_request: [step: string]: any; // input data to be sent␣
```

(continues on next page)

(continued from previous page)

```
↪to the next microservice(s).
                }
}
```

- **request_id:** Service generated job id.
- **status** can be "failed" | "pending" | "successful". While "pending" is returned, helyOS will poll the microservice for results using the request_id.
- **results** can be an array of assignments or a map update, depending on the domain where the microservice was registered. If the microservice is perform intermediate calculations, the results can be any other data structure.
- **dispatch_order** is an array of the element indexes of the results array. In case of Assignment planners, the order of the indexes defines the order in which the corresponding assignment in the results array will be dispatched to the agent.
- **orchestration (optional)** is a field designed to transmit data to the subsequent step in the mission. It is utilized when the input data for the following microservice (the field **request**) needs to be different from the initial mission input data (the field **workProcess.data**).

Example of orchestration field.

Supose that a mission is composed of three steps: 1:"plan_plowing", 2:("charging" and "report_external") and 3:"driving".

Orchestrator Matrix

Step	request order	service type	step dependencies	apply step result
plan_plowing	1	planning_paths		
charging	2	go_to_charger	plan_plowing	true
report_external	2	push_stats_to_cloud	plan_plowing	
format_assignment	3	vendor_format	charging,report_external	true

The microservice of the first step will calculate the path and use the orchestation field to request the charging microservice to supply enough energy for the vehicle run the entire path length.

Listing 5: Example of response using the orchestration field.

```
{
  request_id: "1234",
  status: "successful",
  results:  [
    {
      agent_id: 23,
      assignment: {...}
    }
```

(continues on next page)

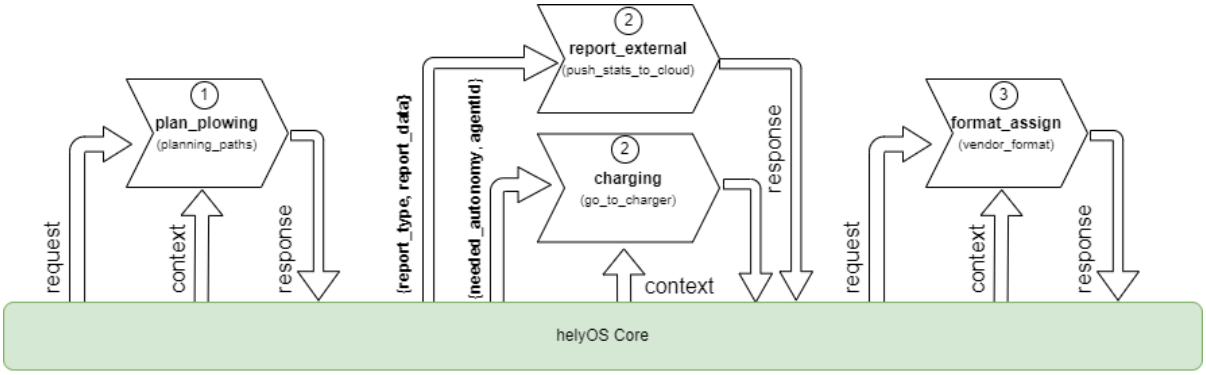
(continued from previous page)

```
    ],

    orchestration:  {
      nex_step_request: {
        "charging": {
            agent_id: 23,
            needed_autonomy: "500 m"
        },
        "report_external": {
            report_type: "charge_report",
            report_data: {...}
        }
      }
    }
  }
```

The microservices of the next step in the mission will receive the following input data in their request field:

- For the "charging" microservice: *request = {agent_id: 23, needed_autonomy: "500 m"}*
- For the "report_external" microservice: *request = { "report_type": "charge_report", report_data: {...}}*



4.3.1 Assignment Creation

Assignments are created by microservices in the *Assignment Planner* domain. A microservice can create one or more assignments per mission, and can define the dispatch order to agents.

Listing 6: Response data to create agent assignments.

```
HelyOSMicroserviceResponse {

    request_id?: string;  // generated job id. Can be used to poll results␣
↪from long running jobs.

    status: "failed" | "pending" | "successful";  .

    results: AssignmentPlan[]; // array of assignments.

    dispatch_order?: number[][]; // order in which the assignments will be␣
↪dispatched to the agents.
```

(continues on next page)

(continued from previous page)

```
    orchestration?: {
              nex_step_request: [step: string]: any;
              }
}


AssignmentPlan {
    agent_id?: number; // id of the agent that will receive the assignment.
    agent_uuid?: string; // UUID of the agent that will receive the assignment.
    assignment: any; // assignment data, usually defined by the agent vendor.
}
```

This microservice response data structure, as defined before, will contains the assignment data in the **results** field.

- **results:** it is an array of assignments where each assignment is ascribed to a agent id.
- **dispatch_order:** When assignments must be executed sequentially, this variable is defined as an array of the element indexes of the results array. The order of the indexes defines the order in which the corresponding assignment will be dispatched to the agent. E.g., [[0], [1,2], [3,4,5]] means that the first assignment will be dispatched first, then the second and third assignments will be dispatched simultaneously, and finally the fourth, fifth and sixth assignments will be dispatched simultaneously.

In the AssignmentPlan, the **assignment** field is a user-defined JSON field that contains the data necessary for the agent to execute the assignment. The agent that will receive the assignment must be identified either by the **agent_id** or by the **agent_uuid** field. The **agent_id** is the database id of the agent, and the **agent_uuid** is the UUID of the agent.

Note:

Note: You cannot send more than one mission at once to a same agent. However, you can SEND SEVERAL ASSIGNMENTS to a same agent! For this, add the assignments into the **results** array with the same **agent_id**.

Use the **dispatch_order** field to let helyOS to sequentially dispatch the assignments to a same agent. Otherwise the assignments will be sent simultaneously; in this case, the agent would need to be smart enough to consume and handle the assignments in the correct order.

4.3.2 Mission Sequence

The following figure illustrates the mission request process from the point of view of the Client application.

1. The client logs on to helyOS and receives an authentication token, which will be used for subsequent requests.
2. The client makes the mission request and helyOS core reserves all agents necessary for that mission.
3. helyOS calls the microservices to calculate the assignment data for the requested mission (which microservices are called and the order in which they are called is pre-configured for each mission type).
4. helyOS receives the assignment data from the microservices and distributes them to the agents using RabbitMQ.
5. When the agents have finished their assignment, they inform helyOS. helyOS may release the agent (reserved = False).

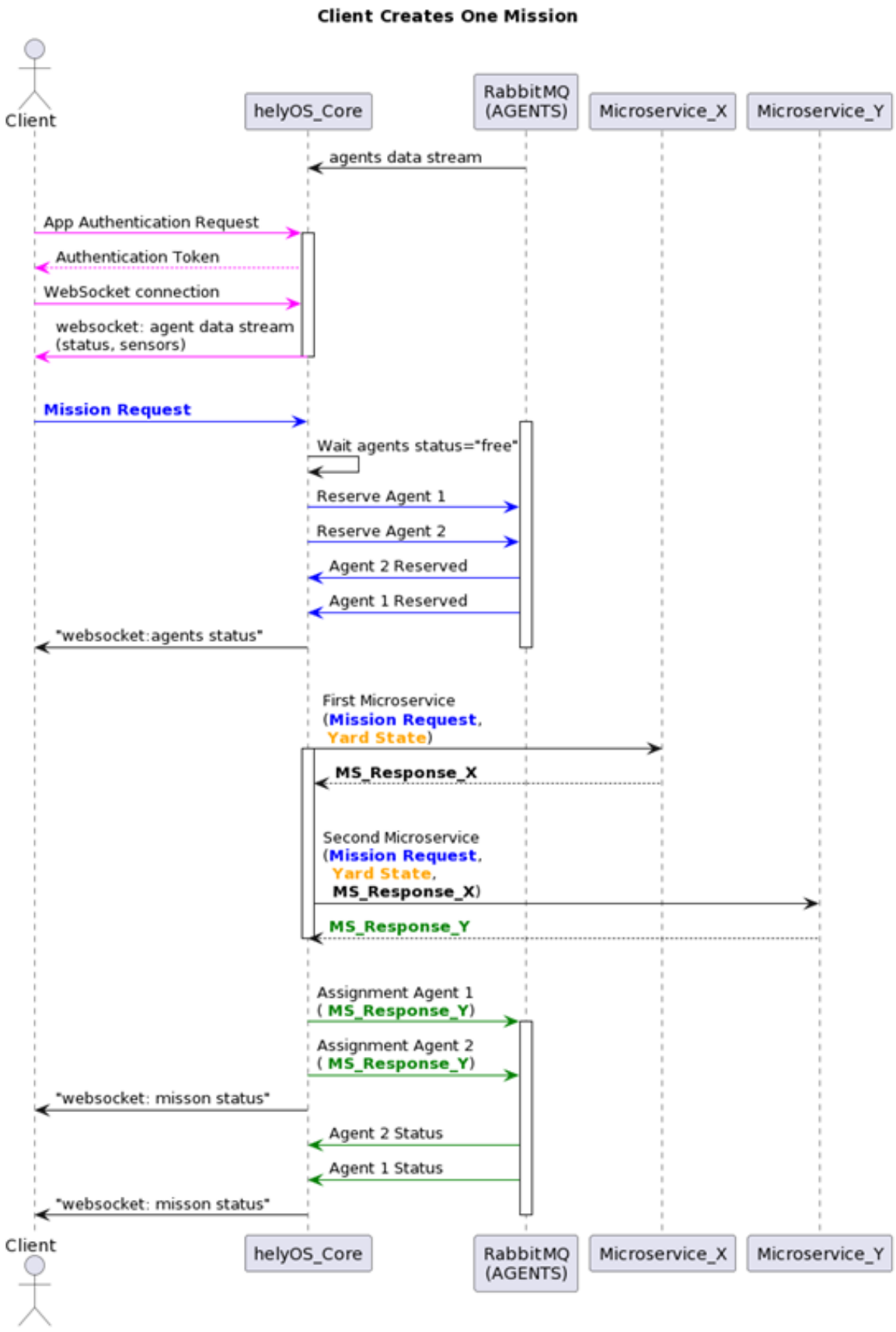


Fig. 1: The process of mission creation from client

4.4 Models Description

Listing 7: Requesta data send from helyOS to the microservice.

```
HelyOSContext  {
    agents: AgentModel[]; // arrray of agents relevant to the mission.

    map: {
          id: number,
          origin: { lat: number, long: number}
          map_objects : MapObjectsModel[]; // array of all map objects in the␣
↪yard.
          };


    orchestation: {
                    current_step: string, // name of the current step in the␣
↪mission.
                    next_step: string,    // name of the next step in the␣
↪mission.
                  };

    dependencies: { step: string,
                    requestUid: string,
                    response: any
                  }[]; // array of data responses from microservice of␣
↪previous steps.

}
```

Listing 8: Agent and map object models.

```
AgentModel {
    id: number;
    uuid: string;
    name: string;
    agent_type: string;
    agent_class: "vehicle" | "assistant" | "tool" | "charge_station";
    pose: { x: number, y: number, z:number,  orientations: number[] };
    status: "free" | "busy" | "ready" | "not_automatable";
    connection_status: "off-line" | "on-line";
    public_key: string;

    data_format: string;
    resources: any;
    geometry: any
    sensors: any;

    interconnections: {uuid: string, agent_type: string}[];
}

MapObjectsModel {
```

(continues on next page)

(continued from previous page)

```
    id: number;
    name: string;
    type: string;
    data_format: string;
    data: any;
    metadata: any;
}
```

In general the microservice response is a JSON object with the following structure:

Listing 9: Response data structure as defined in the Assignment planner API.

```
HelyOSMicroserviceResponse {
    request_id?: string;  // generated job id. Can be used to poll results␣
↪from long running jobs.

    status: "failed" | "pending" | "successful";  .

    results: AssignmentPlan[] | MapUpdate | any;

    dispatch_order?: number[][];

    orchestration?: {
              nex_step_request: [step: string]: any; // input data to be sent␣
↪to the next microservice(s).
              }
}
```

CHAPTER

FIVE

HELYOS AND AGENTS

Communication between helyOS and agents is conveyed by the RabbitMQ message broker. Most of the messages are sent and received using topic exchanges. The agent identification number (uuid) must be registered in the helyOS database. This can be done via dashboard or via a direct GraphQL request, or, in special cases, automatically during the "check-in" process via a registration token.

5.1 Exchange, Routing-keys and Queues in RabbitMQ

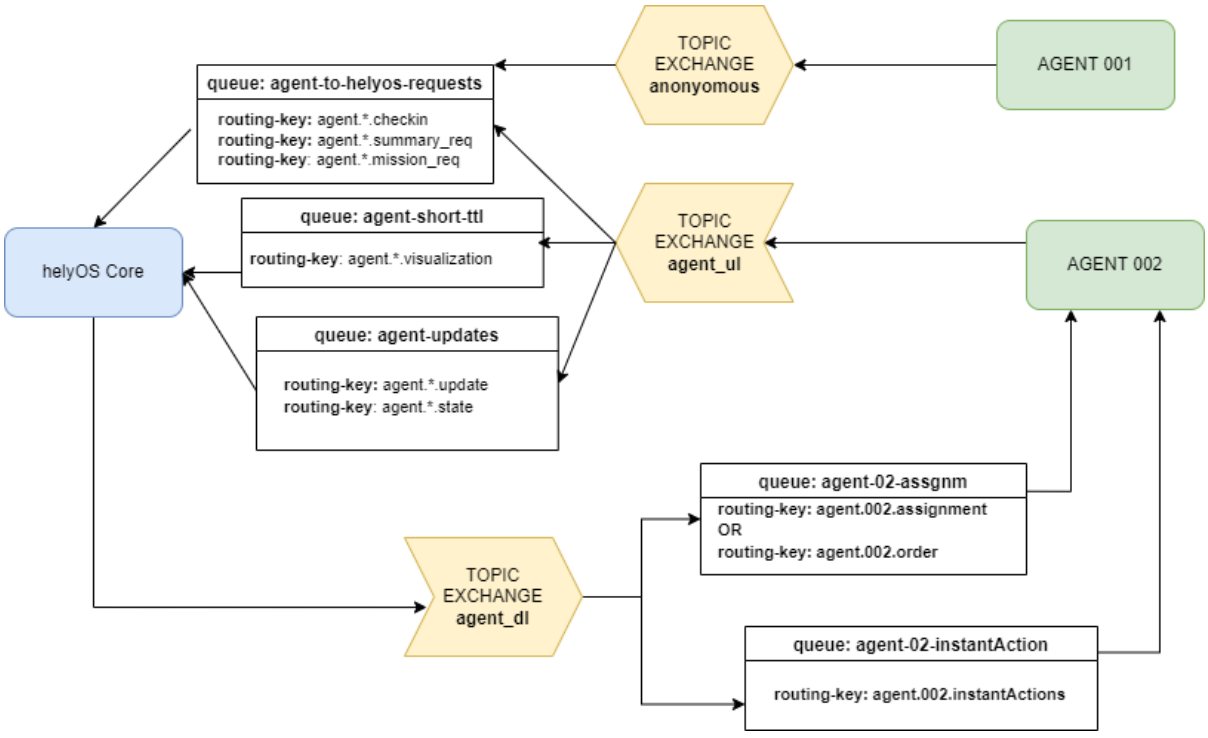


Fig. 1: helyOS and rabbitMQ

Thanks to the RabbitMQ routing features, any RabbitMQ client subscribing to the topic exchange *agent_ul* can get the messages filtered by routing-keys.

The agents will address their messages to the following routing-keys:

- **agent.{uuid}.checkin** : used only for check-in data.
- **agent.{uuid}.update** : messages related to updates of agent properties. e.g., vehicle name, geometry data.

- **agent.{uuid}.visualization** : messages reporting the positioning and sensor data.
- **agent.{uuid}.state** : messages reporting the assignment status and agent state.
- **agent.{uuid}.mission_req** : messages to request missions from helyOS.
- **agent.{uuid}.factsheet** : (included for compatibility with VDA5050) messages to report geometry.

The agents will receive messages from the following routing-keys:

- **agent.{uuid}.assignment** or **agent.{uuid}.order**: receive assignments.
- **agent.{uuid}.instantActions** : receive instant action commands from helyOS core or any other RabbitMQ client.

Routing-keys can be converted to topics for MQTT clients. Check the table below.

All messages exchanged between helyOS and the agents include the following common fields:

- **type:** string, ex: "checkin", "assignment", "cancel", etc..
- **uuid:** string, the identification of the agent the message is about.
- **body:** JSON object.

The additional field **metadata** is present for some messages.

The **body** field will be specific for each message type. The easiest way to communicate to helyOS is to use the agent SDK connector methods: *publish_general_updates*, *publish_states* and *publish_sensors*.

Ref: Documentation and Examples

5.2 Code Examples

You can connect an agent to helyOS via any RabbitMQ or MQTT client using Python, Java, .Net, Ruby, JavaScript, Go, C and C++. Here, we bring some examples of connection using Pyhton clients.

AMQP

```
import pika
from my_agent import sensor_json_str, properties_json_str, states_json_str, mission_
↪request_json_str
from my_callbacks import ia_callback, as_callback

# connect to RabbitMQ
hostname ='rabbitmq.server.de'
username = "134069fc5-fdgs-434b-b87e-f19c5435113"
UPLINK = "xchange_helyos.agents.ul"; DOWNLINK = "xchange_helyos.agents.dl";

credentials = pika.PlainCredentials(username, 'secret_passwd')
sender_validation = pika.BasicProperties(user_id = username)
parameters = pika.ConnectionParameters(hostname,  5672,credentials=credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()

# publish sensors and position - can be performed up to 1000 Hz
channel.basic_publish(UPLINK,"agent.134069fc5-fdgs-434b-b87e-f19c5435113.visualization",␣
```

(continues on next page)

Feature	AMQP (routing-key)	MQTT (topic)	Remarks
Check-in Agents are isolated in yards. Agents request helyOS to be registered into a desired yard.	***agent.{uuid}.checkin***	***agent/{uuid}/checkin***	The agent receives the yard map data and error logs. **AMQP:** Can be executed as anonymous and used to create RabbitMQ accounts on the fly. The response is received following the RPC pattern. **MQTT:** The agent RabbitMQ account must exist. The response is received at a custom topic.
Instant actions Agent receives requests for actions from helyOS core, other agents or external applications.	***agent.{uuid}.instantActions***	***agent/{uuid}/instantActions***	**AMQP:** Publishers are automatically validated upon publication. **MQTT:** Publishers must be validated at receiver side.
Assignments Agent receives assignments from helyOS core.	***agent.{uuid}.assignment*** ***agent.{uuid}.order***	***agent/{uuid}/assignment*** ***agent/{uuid}/order***	**AMQP:** Publishes are automatically validated upon publication. **MQTT:** Publishers must be validated at receiver side.
Agent state Agent notifies helyOS core about its current state and its assignment status.	***agent.{uuid}.state***	***agent/{uuid}/state***	The publication should occur immediately upon the status change and subsequently at regular intervals ranging from 0.2 to 1 Hz.
Agent visualization (optional) Agent broadcasts position and sensor data.	***agent.{uuid}.*** **visualization**	***agent/{uuid}/*****visualization**	Advisable up to 10 Hz. helyOS core will store these data in a fixed sampling rate.
Agent updates (optional) Agents publish to save their properties and position in helyOS core.	***agent.{uuid}.*** **update**	***agent/{uuid}/*****update**	Advisable up to 1 Hz.
Database Requests (optional) Agent can request data from helyOS database and, for some entities, update data.	***agent.{uuid}. database_req***	***agent/{uuid}/database_req***	This feature is intended for retrieving data that is not being published in RabbitMQ. **AMQP:** The response is received following the RPC pattern. **MQTT:** Not implemented. The developer must handle the response received a custom topic.
Mission Requests (optional) For specific scenarios, agent can request a new mission for itself or for other agents.	***agent.{uuid}.*** **mission_req**	***agent/{uuid}/*****mission_req**	**AMQP:** The response is received following the RPC pattern. **MQTT:** The response is received at a custom topic.

(continued from previous page)

```
↪sensor_json_str, sender_validation)

# update propeties as geometry and position - can be performed up to 10 Hz
channel.basic_publish(UPLINK,"agent.134069fc5-fdgs-434b-b87e-f19c5435113.update",␣
↪properties_json_str, sender_validation)

# update agent and assignment status - must be performed immediately when the status␣
↪change. Up to 2 Hz
channel.basic_publish(UPLINK,"agent.134069fc5-fdgs-434b-b87e-f19c5435113.state", states_
↪json_str ,sender_validation)

# request a mission to helyOS
channel.basic_publish(UPLINK,"agent.134069fc5-fdgs-434b-b87e-f19c5435113.mission",␣
↪mission_request_json_str ,sender_validation)

# receive instant actions
channel.queue_declare(queue='ia_queue')
channel.queue_bind('ia_queue', DOWNLINK,"agent.134069fc5-fdgs-434b-b87e-f19c5435113.
↪instantActions")
channel.basic_consume('ia_queue', auto_ack=True, on_message_callback=ia_callback)

# receive order or assignments
channel.queue_declare(queue='as_queue')
channel.queue_bind('as_queue', DOWNLINK, "agent.134069fc5-fdgs-434b-b87e-f19c5435113.
↪assignment") # or ... .order
channel.basic_consume('as_queue', auto_ack=True, on_message_callback=as_callback)

channel.start_consuming()
```

Tapping into the agent's data stream

```
import pika, json

# connect to RabbitMQ
hostname ='rabbitmq.server.de'
username = "assistant-3432-434b-b87e-ds3245323"
UPLINK = "xchange_helyos.agents.ul"

credentials = pika.PlainCredentials(username, 'secret_passwd')
parameters = pika.ConnectionParameters(hostname,  5672,credentials=credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()

def parse_any_helyos_agent_message(raw_str):
    # get message string
    object = json.loads(raw_str)
    message_str = object['message']
    message_signature = object['signature']
    # parse message string
    message = json.loads(message_str)
    print(f"message type: {message['type']}")
    print(f"message uuid: {message['uuid']}")
```

(continues on next page)

(continued from previous page)

```
    print(f"message body: {message['body']}")
    print(f"message metadata: {message.get('metadata', None)}")


# Tapping into the agent's data stream - VISUALIZATION
def tap_visualization_callback(ch, method, properties, raw_str):
    print("visualization data received")
    parse_any_helyos_agent_message(raw_str)

channel.queue_declare(queue='visualization_queue')
channel.queue_bind('visualization_queue', UPLINK, "agent.*.visualization")
channel.basic_consume('visualization_queue', auto_ack=True, on_message_callback=tap_
↪visualization_callback)

# Tapping into the agent's data stream - STATE
def tap_state_callback(ch, method, properties, raw_str):
    print("state data received")
    parse_any_helyos_agent_message(raw_str)

channel.queue_declare(queue='state_queue')
channel.queue_bind('state_queue', UPLINK, "agent.*.state")
channel.basic_consume('state_queue', auto_ack=True, on_message_callback=tap_state_
↪callback)

# Tapping into the agent's data stream - UPDATE
def tap_update_callback(ch, method, properties, raw_str):
    print("update data received")
    parse_any_helyos_agent_message(raw_str)

channel.queue_declare(queue='update_queue')
channel.queue_bind('update_queue', UPLINK, "agent.*.update")
channel.basic_consume('update_queue', auto_ack=True, on_message_callback=tap_update_
↪callback)

channel.start_consuming()
```

MQTT

```
import paho.mqtt.client as mqtt
# connect to RabbitMQ
hostname ='rabbitmq.server.de'
username = "134069fc5-fdgs-434b-b87e-f19c5435113"

client = mqtt.Client()
client.username_pw_set(username, 'secret_passwd')
client.connect(rabbitmq_host, 1886)

# publish sensors and position - can be performed up to 1000 Hz
client.publish("agent/134069fc5-fdgs-434b-b87e-f19c5435113/visualization", sensor_json)

# update propeties as geometry and position - can be performed up to 10 Hz
client.publish("agent/134069fc5-fdgs-434b-b87e-f19c5435113/update", propeties_json)
```

(continues on next page)

(continued from previous page)

```
# update agent and assignment status - must be performed immediately when the status␣
↪change. Up to 2 Hz
client.publish("agent/134069fc5-fdgs-434b-b87e-f19c5435113/state", agent_assign_states_
↪json)

# receive instant actions
client.subscribe("agent/134069fc5-fdgs-434b-b87e-f19c5435113/instantActions")
client.message_callback_add("agent/134069fc5-fdgs-434b-b87e-f19c5435113/instantActions",
↪ia_callback)

# receive order or assignments
client.subscribe("agent/134069fc5-fdgs-434b-b87e-f19c5435113/assignment") # or ../order
client.message_callback_add("agent/134069fc5-fdgs-434b-b87e-f19c5435113/assignment",as_
↪callback)

client.loop_start()
```

These codes can be simplified by using the *helyos-agent-sdk*. See examples also for AMQP and MQTT agents: https://fraunhoferivi.github.io/helyOS-agent-sdk/build/html/examples/index.html

5.3 Check in agent in helyOS

To receive assignments from helyOS, the agent must perform a procedure called "check-in".

In the check-in procedure, the agent will

- Connect to RabbitMQ and send its identification data.
- If the agent is connected as anonymous and possess the helyOS registration token, a new username and password will be automatically created.
- Create a temporary queue to receive the check-in response.

Listing 1: Check-in data sent by the agent to helyOS core. The symbol (?) means optional.

```
CheckinCommandMessage {
    type: "checkin";

    uuid: string;

    body: {
            yard_uid: string;         // yard the agent is checking in.
            status: string;
            pose: { x:number, y:number, z:number, orientations:number[]};
            type?: string;
            name?: string;
            data_format?: string;
```

(continues on next page)

(continued from previous page)

```
            public_key?: string;
            geometry?: AnyDataFormat;
            factsheet?: AnyDataFormat
    }

}
```

- **geometry:** JSON informing the physical geometry data of the vehicle.
- **yard_uid:** Unique identifier of the yard as registered in the dashboard.

helyOS will respond with the following data:

Listing 2: Check-in data sent by helyOS core to agent. The symbol (?) means optional.

```
CheckinResponseMessage {
    type: "checkin";

    uuid: string;

    body: {
            agentId: number;     // agent database id number
            yard_uid: string;   // yard the agent is checking in.
            status: string;
            map: {  uid:string,
                    origin:{lat:number, lon:number, alt:number},
                    map_objects: MapObjects[]
                  };
            rbmq_username: string;
            response_code: string;
            helyos_public_key: string;
            ca_certificate: string;   // RabbitMQ server certificate for SSL connection
            rbmq_password?: string;  // When agent account does not exist in the␣
↪RabbitMQ server.
            password_encrypted? boolean
    }

}
```

- **type** = "check in".
- **map:** JSON with the map information from yard.
- **rbmq_username:** RabbitMQ account to be used by this agent.
- **rbmq_password:** RabbitMQ password, only used for anonymous check-in.
- **password_encrypted:** If true, the rbmq_password field is encrypted with the agent public key.

Check in using python code:

```
def checkin_pseudo_code(username, password):
    # step 1 - connect
    temporary_connection = connect_rabbitmq(rbmq_host, username, password)
    gest_channel = temporary_connection.channel()
```

(continues on next page)

(continued from previous page)

```
    # step 2 - create a queue only to receive the check-in response
    checkin_response_queue = gest_channel.queue_declare(queue="")

    # step 3 - publish the check-in request
    uuid = "y4df7293-5aab-46e2-bf6b"
    publish_in_checkin_exchange_topic(yard_id=1,
                                      uuid: uuid,
                                      routing_key: f"agent-{uuid}-checkin,
                                      status="free",
                                      agent_metadata=data,
                                      reply_to= checkin_response_queue)


    # step 4 - start to consume checkin_response_queue and get the response data
    if username == 'anonymous':
        new_username, new_password, yard_data = listen_checkin_response(checkin_response_
↪queue)
        helyos_connection = connect_rabbitmq(rbmq_host, new_username, new_password)
    else:
        _, _, yard_data = listen_checkin_response(checkin_response_queue)
        helyos_connection = connect_rabbitmq(rbmq_host, username, password)

    return helyos_connection, yard_data
```

The similar code using *helyos-agent-sdk* python package:

```
from helyos_agent_sdk import HelyOSClient, AgentConnector

helyOS_client = HelyOSClient(rbmq_host,rbmq_port, uuid="y4df7293-5aab-46e2-bf6b")
if username!='anonymous':
    helyos_client.connect(username, password)
helyOS_client.perform_checkin(yard_uid='1', agent_data=data, status="free")
helyOS_client.get_checkin_result()

helyos_connection = heylOS_client.connection
```

The *helyOS-agent-sdk* has many other methods to send and receive data from helyOS core in the correct data format. Check the documentation at https://fraunhoferivi.github.io/helyOS-agent-sdk/build/html/index.html.

5.4 Data Flow between helyOS and Agents

Only if the agent's uuid is registered in the helyOS database, the agent can exchange messages with helyOS to report its status and to perform the assignments.

Note that before receiving any assignment, the agent must be reserved for the assignment mission. That is, the agent changes the status from "free" to "ready" (i.e., ready for the mission) upon helyOS *Reserve* request. Once the agent finishes the assignment, the agent will not set its status from "busy" to "free", but to "ready". This is because helyOS may sent him a second assignment belonging to the same mission. For this reason, the agent must wait the "Release" signal from helyOS to set itself "free".

5.5 helyOS Reserves Agent for Mission

Before processing a mission request, helyOS core will reserve the required agent(s). This is done via the routing key, *agent.{uiid}.instantActions*. helyOS requests the agent to be in **"ready"** status (status="ready" and reserved=True). During the assignment, the agent's status changes to **"busy"**. After the assignment is complete, the agent updates its status from **"busy"** to **"ready"**. At this point, helyOS may release the agent, depending on the presence of any further assignments in that mission. The release message is also delivered via instant actions.

The agent reservation is important because:

(i) Mission calculations can require considerable computational power and take several seconds. Therefore, the agent must remain available during this period and not be used by other tasks.

(ii) In some missions, multiple agents may need to perform sequential assignments. In such cases, one agent must be reserved to wait for the completion of assignments from another agent.

(iii) Some missions require unique tools or devices that may not be present at the required agent. Thus, ensuring the readiness of both the agent and its hardware for the specific assignment is important.

(iv) In the interest of security, heavy agents, even those set to automatable mode, should communicate their upcoming assignment visually or soundly to their surroundings. This feature allows anyone nearby to abort the assignment before it starts if deemed necessary.

However, in some scenarios, agents should not be blocked waiting for a mission calculation. Instead, they should either fail the mission if they become suddenly unavailable after the calculation is done, or queue the assignment to be executed later. For those scenarios, the developer must uncheck the option *Acknowledge reservation* on the *Register Agent* tab in the dashboard.

5.6 helyOS Sends Assignment to Agent

As earlier mentioned, the assignments usually originated from the microservices. That is, the microservices translate the requested mission in assignments: *Assignment*. The microservices return the assignments to helyOS core, and helyOS distributes them to the agents. This is done via the routing key *agent.{uiid}.assignments*.

If the option *Acknowledge reservation* is checked, helyOS will send an assignment to the agent **only if the agent status is "ready"**.

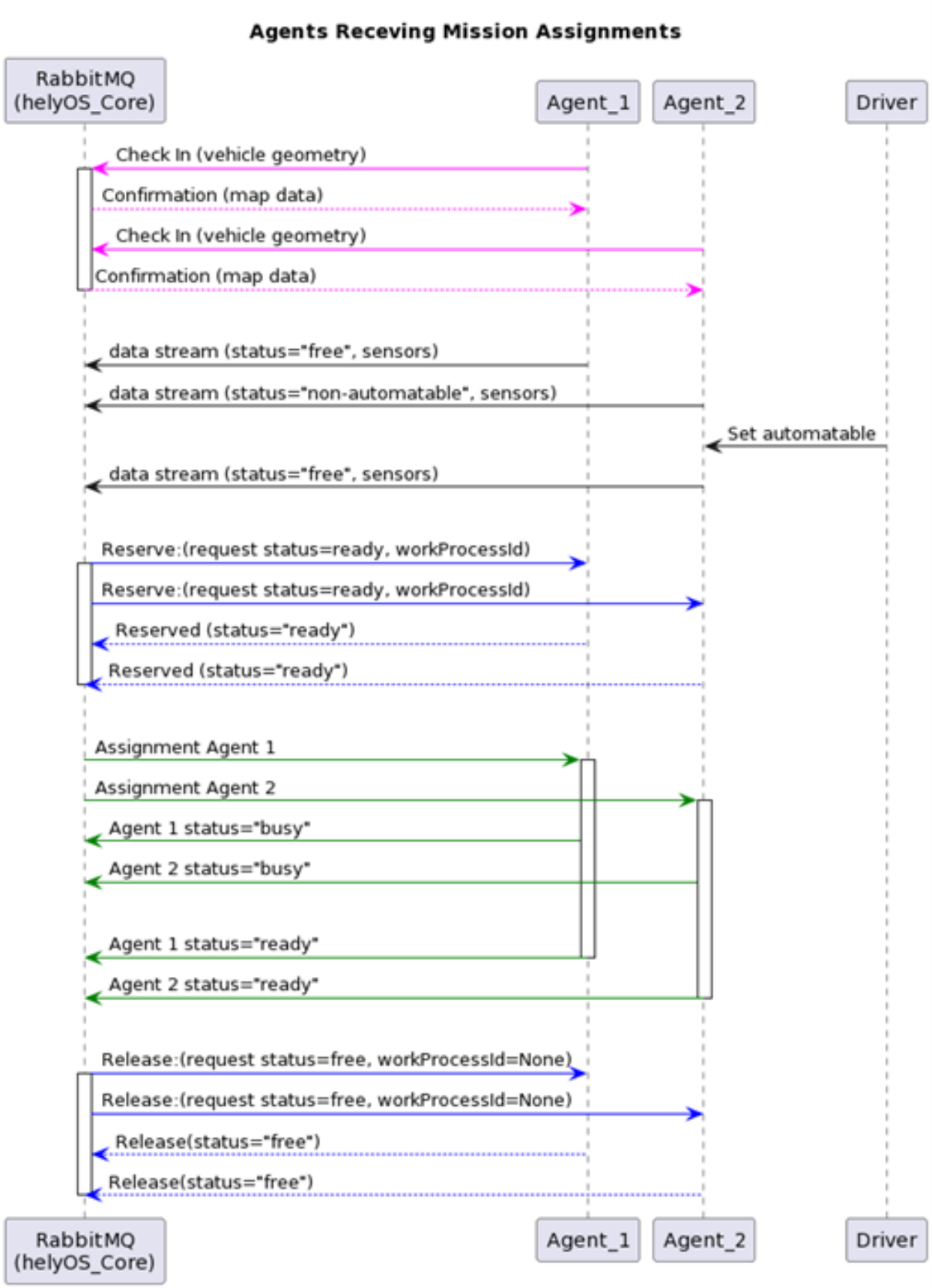


Fig. 2: The process of agents receiving mission assignments

Listing 3: Assignment object data format. The field **metadata** is automatically generated by helyOS core.

```
AssignmentCommandMessage {
    type: "assignment_execution";

    uuid: string;

    body: AnyDataFormat;

    metadata: {
                id: number,             // assignment id.
                workprocess_id: number, // mission id.
                yard_id: number,
                status: string,
                context?: { dependencies: PreviousAssignments[]}
    }

}
```

An easy-to-implement security mechanism is to check the identity of the assignment sender. This is an embedded feature of RabbitMQ. For example, if you want your agent to only execute assignments from helyOS core, you can filter assignments originated from the RabbitMQ account "helyos_core".

5.7 Agent Requests a Mission

In addition to client apps, agents can also request missions from helyOS core. This feature is useful for situations such as the following:

- A smart camera identify a new obstacle and requests a mission to update helyOS map by sending the position of a new obstacle.
- A tractor requests a mission to ask assistance of another agent for executing a task.
- A truck finds itself obstructed by a fixed obstacle, the truck requests a mission from helyOS to calculate a path away from this deadlock situation, or to contact a teleoperated driving service.

CHAPTER

SIX

APPLICATIONS IN YARD AUTOMATION

helyOS core responds to database events. That is, the creation, update or delete of rows in the database tables trigger actions inside the helyOS core. The Client Applications communicate with helyOS core by interacting with the helyOS database using the GraphQL language. The microservices attached to helyOS core will provide the specific features to the application.

6.1 Implementation of a Yard Automation Application

helyOS core is a single NodeJS application serving ports 5000, 5002 and 8080 for the GraphQL connection, the web socket connection and the dashboard GUI respectively. helyOS connects as a client to Postgres and RabbitMQ. All parameters for these connections are passed as environment variables.

Since helyOS is containerized, it is easy to launch a helyOS application in the cloud. Users can either run the container inside a single Linux or Windows computer of a cloud provider, or they can implement helyOS in a serverless approach using available cloud products with horizontal auto scaling.

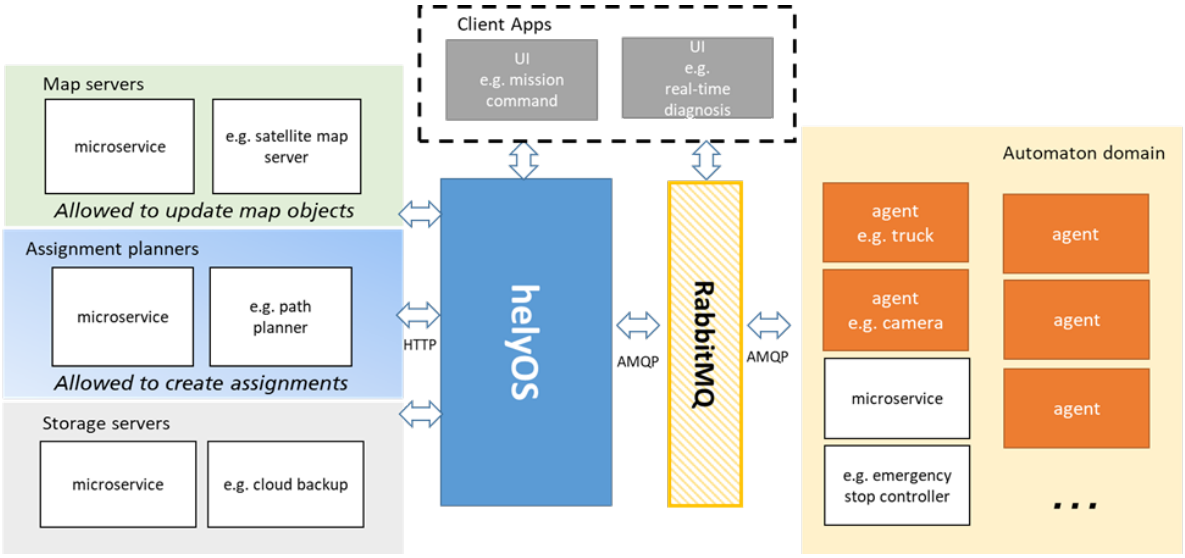


Fig. 1: helyOS yard automation application

6.1.1 What is possible in the helyOS framework?

The helyOS framework gives developers many options to solve yard automation problems. To better use this flexibility, the helyOS framework suggests a template to organize the data flow and the responsibilities of each domain. Following this template will lead to a more robust and maintainable software architecture.

	Communication		Request a map update	Request a mission with assignment(s) to agent(s)	Intermediary chained calculations	Calculation of Assignment data (e.g. trajectories)	Directly Update Map Objects	Cancel mission	Send instant action to agents	Dispatch assignment to agents	Dispatch request to helyOS microservices
Applications	HTTP GRAPHQL	Expert Systems	✓	✓		✓	✓	✓	✓		✓ Anti-pattern
		User Interfaces	✓	✓		✓ Anti-pattern	✓	✓	✓		✓ Anti-pattern
Microservices	HTTP REST	Assignment domain			✓	✓					✓ Anti-pattern
		Map domain			✓		✓				✓ Anti-pattern
		Autonomous domain	✓	✓		✓			✓	✓ Anti-pattern	✓ Anti-pattern
Agents	RABBITMQ AMQP	Trucks, tractors, cameras	✓	✓		✓			✓	✓ Anti-pattern	✓ Anti-pattern
helyOS core		Backend, Orchestrator							✓	✓	✓

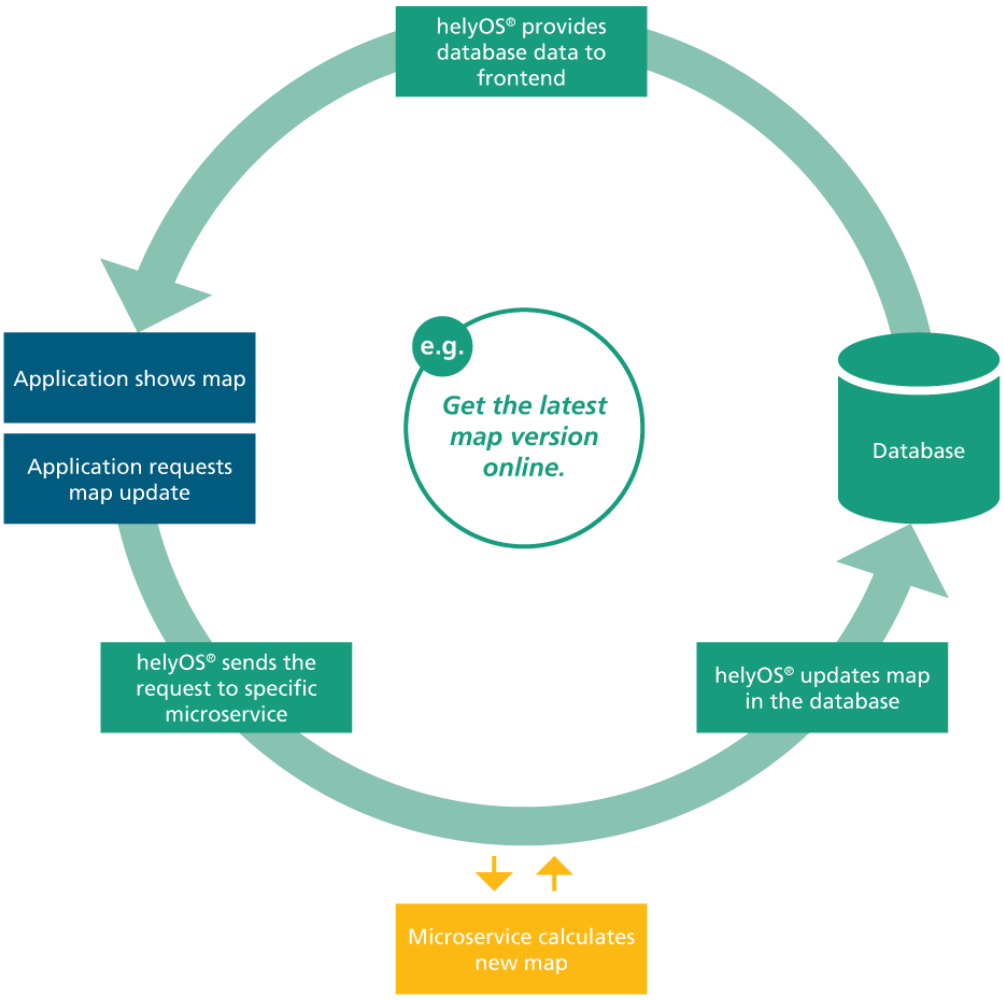
✓ Regular use

✓ Special cases

✓ Anti-pattern

6.2 Examples of missions using helyOS

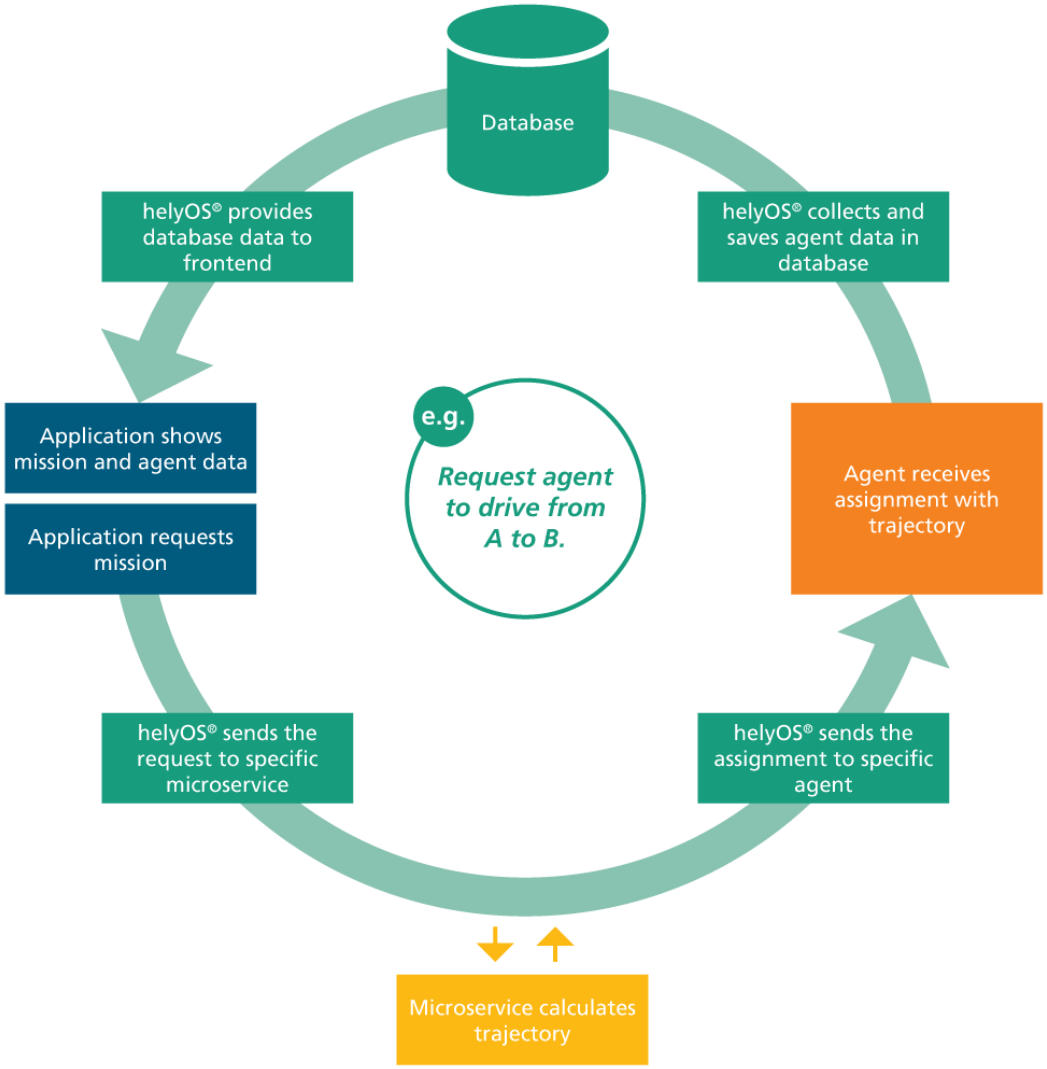
A. Application requests a map update.

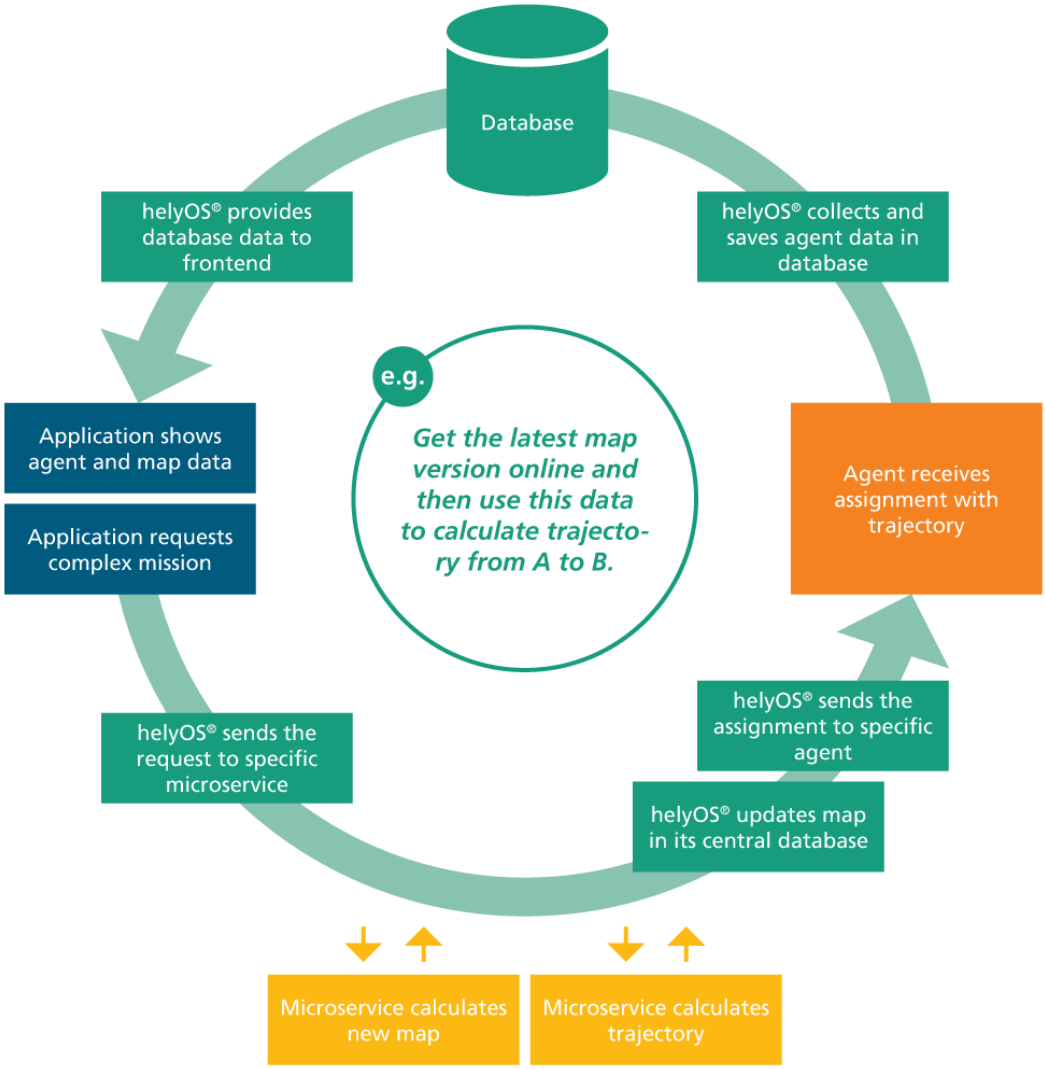


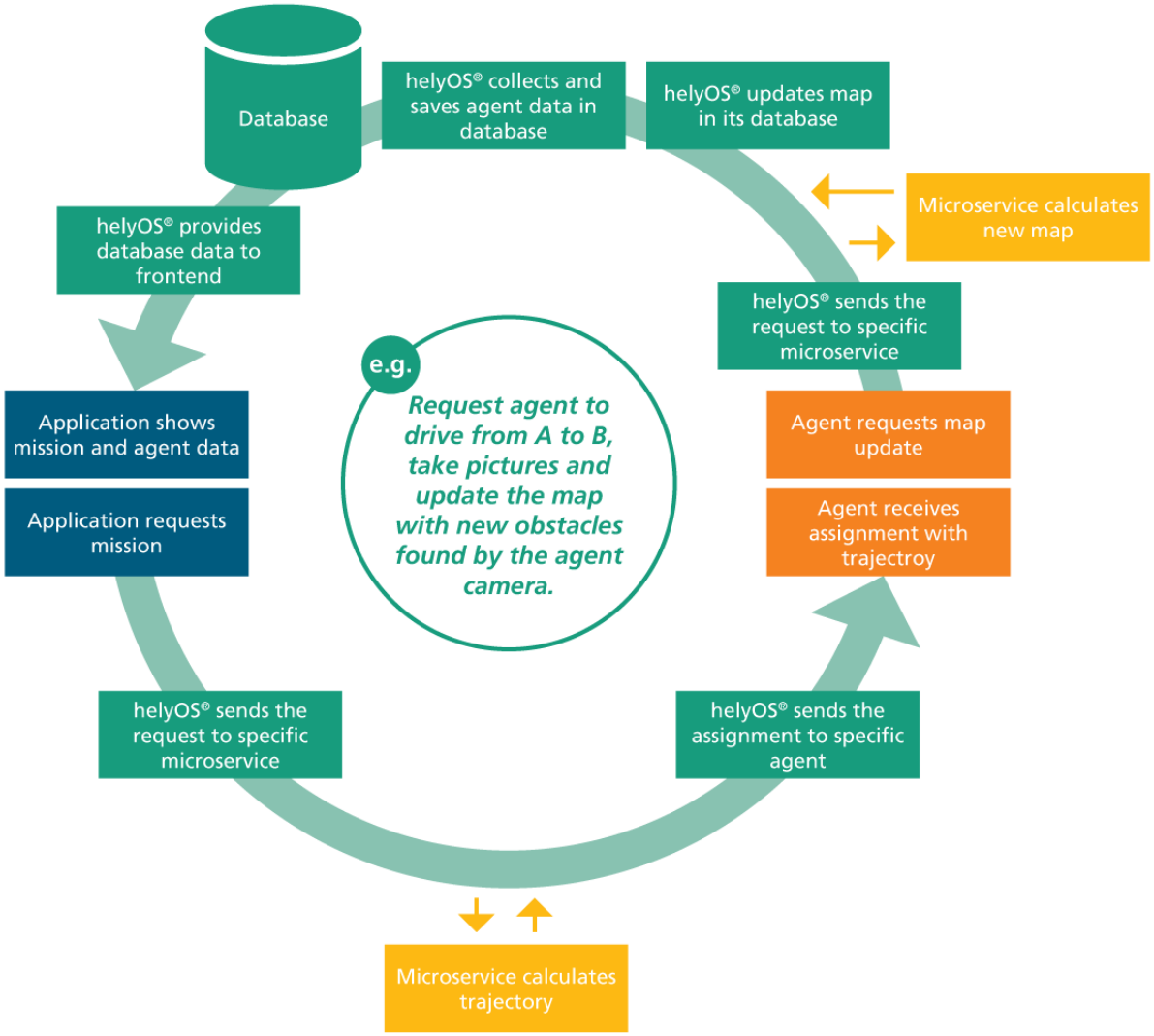
B. Application requests truck to drive from A to B using an online path planner.

C. Application requests truck to drive from A to B using an online path planner but employing the most recent map data in the path calculation.

D. Application requests robot to take pictures and update the map objects.


Database
helyOS® provides database data to frontend
helyOS® collects and saves agent data in database
Application shows mission and agent data
Application requests mission
e.g.
Request agent to drive from A to B.
Agent receives assignment with trajectory
helyOS® sends the request to specific microservice
helyOS® sends the assignment to specific agent
Microservice calculates trajectory


Database
helyOS® provides database data to frontend
helyOS® collects and saves agent data in database
e.g.
Get the latest map version online and then use this data to calculate trajecto-ry from A to B.
Application shows agent and map data
Application requests complex mission
Agent receives assignment with trajectory
helyOS® sends the request to specific microservice
helyOS® sends the assignment to specific agent
helyOS® updates map in its central database
Microservice calculates new map
Microservice calculates trajectory


Database
helyOS® collects and saves agent data in database
helyOS® updates map in its database
Microservice calculates new map
helyOS® sends the request to specific microservice
Agent requests map update
Agent receives assignment with trajectroy
helyOS® sends the assignment to specific agent
Microservice calculates trajectory
helyOS® sends the request to specific microservice
Application requests mission
Application shows mission and agent data
helyOS® provides database data to frontend
e.g.
Request agent to drive from A to B, take pictures and update the map with new obstacles found by the agent camera.

CHAPTER

SEVEN

FAQ

Frequently asked questions.

7.1 Why not code everything in a single backend? Why do I need helyOS?

Because a monolithic solution for real-world yard automation is unwise from the technical and managerial point of view. A yard automation application integrates several technologies and are developed by interdisciplinary teams. The best way to handle such a project is by using microservice architectures.

Please read: *Why you should use helyOS*.

7.2 What are microservices?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are loosely coupled, organized around business capabilities and owned by a small team. The microservice architecture enables an organization to deliver large, complex applications rapidly, reliably and sustainably.

In the helyOS framework, microservices are used to implement path planners, data parsers, map servers, storage services, etc. They are grouped and arranged according to the mission requirements.

7.3 What does it means that "helyOS orchestrates microservices"?

It means that as soon helyOS receives a mission request, helyOS will:

- check the health of the services used by the mission,
- sequentially dispatch the request data to these services,
- periodically pool the server response if it is not immediately available,
- forward the response from one service to the next one (if required),
- deliver the service response to the agents

7.4 What does it means that "helyOS orchestrates assignments"?

It means that as soon helyOS receives a mission request, helyOS will:

- check the availability of the required agents,
- send a signal to reserve the agents for the mission,
- sequentially send the mission assignments to the agents as specified in the mission.
- release the agents if there is no more assignments to be executed in the mission or if the mission was canceled.

7.5 Can helyOS calculate trajectory paths?

No. helyOS connects to a server that calculates paths. helyOS takes care of delivering the calculated paths to the "free" agents. You need only to register the server URL as microservice in the helyOS dashboard.

7.6 Can I send several missions at once to one automated vehicle?

You can create **one** mission with **many** assignments to one automated vehicle.

7.7 What is the difference between mission and assignment?

A yard automation application is defined in terms of its available missions. To complete a mission, agents must perform one or more assignments. helyOS receives the request of a mission and dispatches assignments to one or more agents.

7.8 What is the data format for the agent sensors?

helyOS uses JSON formats. You are free to decide the data structure according to your frontend. If you have no idea, just use the helyOS-native format and you will be safe:

7.9 I want to use an online server for path calculation (or map information) which has its own API. How can I integrate with helyOS?

You need to make a small service to convert from the original API to the helyOS API and register it as a microservice in the dashboard. Since the helyOS API is extremely simple, this can be done with a few lines of code.

[field_id: string]:
“value” : string | number, required
“title” : string, required
“type” : string = “string” or “number”, required
“description”: string,
“unit”: string,
“minimum” : number,
“maximum” : number,
“maxLength”: number,
“minLength”: number

```
sensors = {
  "sensor_set_2": {
    "velocity_01": {
          "title": "velocity",
          "value": 20,
          "type": "number",
          "unit": "km/h",
          "minimum": 0,
          "maximum": 200
     },
     "back_door_status": {
          "title": "Truck door",
          "value": "half-open",
          "type": "string",
          "unit": "km/h",
          "minLength": 5,
          "maxLength": 10
     }
 }
```

Fig. 1: helyOS-native format for agent sensor data

7.10 What is the difference between helyOS and Automation App?

helyOS is a software framework used to facilitate the creation of control tower software for different applications like e.g., agriculture, logistics centers and harbors.

Automation App is proprietary software application used as a frontend tool to prototype projects in logistic centers. Automation App uses helyOS as backend.