UP-Drive
Automated Urban Parking and Driving

Research and Innovation Action - Horizont 2020
Grant Agreement Number 688652
Start date of the project: 01-01-2016, Duration: 48 months

Deliverable D 5.1

Specification of the Map Frontend and Storage Concept

Type: R (Document, report)

Status: final

Lead contractor for this deliverable: ETHZ

Due date of deliverable: <30.04.2016>
Actual submission date: <29.06.2016>

Coordinator: VW

Project co-funded by the European Commission within HORIZON 2020		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
SEAM	Restricted to partners of the SEAM Cluster (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EXECUTIVE SUMMARY
This deliverable corresponds to task 5.1, 5.2 and 5.3. It describes the hardware and software requirements and specifications for the mapping and localization frontend and storage concepts in the cloud-based backend.

CONTRIBUTING PARTNERS		
	Company/Organisation	**Name**
Document Manager	ETHZ	Mathias Bürki
	ETHZ	Elena Stumm
	ETHZ	Cesar Cadena
	ETHZ	Juan Nieto
	IBM	Martin Rufli
	IBM	Ralf Kästner

REVISION TABLE		
Document version	**Date**	**Modified sections - Details**
1	29.04.2016	Initial version for submission.

1 TABLE OF CONTENTS

2 INTRODUCTION

2.1 Scope of the Document

This deliverable relates to task 5.1, 5.2 and 5.3 and describes the hardware and software requirements and specifications for the mapping and localization frontend as well as the storage and interfacing concepts in the cloud-based backend. In particular it covers the following key aspects:

- Localization frontend: Requirements and specifications for the software module that allows the car to localize precisely during online operation.
- Mapping frontend: Requirements and specifications for how to transmit map-data from the car to the cloud-based backend.
- Mapping backend:
    - Interfaces describing how to interact with the mapping backend, how to issue queries and receive data during operation, as well as during offline data exchange.
    - Applications and functionality available in the mapping backend.
    - Storage concepts for the mapping data.

2.2 General Comments

In this document, we follow a clear distinction between metric mapping/localization and semantic mapping. With metric mapping/localization we refer creating a geometric map that can be used for precise localization of the vehicles e.g. that allows estimating a 6DoF transformation between the vehicle's body-fixed coordinate frame and some well-defined global coordinate frame of reference. With semantic mapping, we refer to the labeling, storage and retrieval of semantically annotated objects, such as zebra crossings, traffic lights, lane markings, etc. that are used to understand and interpret the scenarios on the road on a semantical level.

3 REQUIREMENTS

In this section, we focus on the requirements of the software architecture and infrastructure necessary to achieve the metric localization and mapping, as well as the semantic (offline) mapping tasks specified in work-package 5. A detailed and broader overview of the requirements for WP5 including also the involved sensor suite and calibration can be found in section 4.4 of deliverable D1.1.

The software architecture and infrastructure requirements arise from two aspects, namely the scope and size of the target scenarios on the one side and the use-cases of the localization and mapping system on the other side.

As we aim at deploying multiple automated vehicles in large urban environments, it is crucial to provide a localization and mapping system that scales appropriately to areas of substantial size and that allows multiple vehicles to communicate, cooperate and exchange mapping data during operation.

To illustrate the latter, we enumerate an exemplary list of common and significant use-cases below. Note that this list is not exhaustive but merely serves as a guide to define and justify the requirements.

- Query pose.
  Given (a) camera image(s) or keypoints with feature descriptors, a 6DoF pose of the vehicle wrt. the map reference frame is queried against a metric-map available in the backend.
- Query near-by metric-map data for on-car localization.
  Given a rough estimate of the car's pose in the map, near-by data for localizing on the car (e.g. near-by landmarks) are requested from the backend.
- Query for near-by semantic objects of some kind.
  Given an estimate of the car's pose, semantic objects of some kind (e.g. zebra-crossings) are requested from the backend.
- Query for all semantic objects of a kind.
  Requests all semantic objects of some kind (e.g. zebra-crossings) available in some semantic map in the backend.
- Augment the/a metric-map with online localization/sensor data.
  Transmits data, such as images, keypoints, descriptors, localization estimates, etc. from the car to the mapping backend during an operation and augments a metric-map in the backend with this data on the fly.
- Upload a dataset for archiving.
  After collecting sensor data during an operation, the dataset is uploaded onto the backend and archived.
- Download a metric-map for localization on the car.
  A full metric-map is downloaded from the backend to run localization on the car during a subsequent operation.
- Download an archived dataset for on-desk inspection.
  One or more datasets in the backend are downloaded in order to analyse and inspect them offline.

- Build a metric-map from one or more archived dataset.
  Computes a new metric-map from the sensor data recorded in a given archived dataset.
- Augment a metric-map with one or more archived datasets.
  Adds more sensor data from one or more archived datasets to an already existing metric-map.
- Run Loop-Closure, Bundle-Adjustment, Map-Summarization, etc. on a metric-map.
  Runs various algorithms on an already existing metric-map.
- Query for a GPS estimate given a pose wrt. the map reference frame.
  Given a 6DoF pose of the vehicle wrt. the map reference frame, a (rough) GPS estimate for this pose is calculated.

From this list, we derive the following additional requirements:

- Access of the vehicles to a shared map source during operation. This access is time-critical and only small delays between query and response are acceptable. It may, however, be limited to the exchange of only small amounts of data.
- Offline access to the mapping framework must be possible including the exchange of large amounts of sensor data.
- Various software tools to create and maintain maps for (metric) localization as well as scene interpretation and maintenance and reasoning about semantic data must be available.
- A data archive must be available, allowing to store and retrieve datasets for map building, map augmentation, other forms of post-processing or later inspection and evaluation.

4 FRAMEWORK ARCHITECTURE

4.1 Overview

Given the requirements specified in the previous sections, as well as in section 4.4. of deliverable D1.1, we propose the following cloud-based framework architecture for metric localization and mapping together with semantic mapping and semantic reasoning:

- In order to separate data storage from applications and their service endpoints on the cloud-based backend side and the client side applications on the cars, a multi-layer architecture is envisioned. A conceptual drawing of this architecture can be found in figure 1 below.
- On the data layer, separate blocks for the metric map data and semantic data (object store with databases) are provided. This allows integrating the already existing multiagent-mapping framework from ETHZ [1, 2].
- On the application level, various software modules offer a wide range of functionality such as map building, map curation, localization, semantic labeling as well as handling semantic queries of any kind.
- In order to offer a transparent interface to the client side (the cars), service endpoints are employed implementing a RESTful web-interface to the services available in the mapping framework.

4.2 Data Store Layer

All data which will be entered into and processed by our cloud framework will be maintained in the data store layer. In this layer, we distinguish three different stores. Each of these stores will be designed to efficiently represent and operate on a particular data model. In brief terms, the stores and their fundamental data models will be:

- An object store for unstructured information (flat data model)
- A document store for semi-structured key-value information (associative array model)
- A knowledge store for structured semantic information (RDF or graph data model)

Note that the above data models are not necessarily disjunct. Thus, graph information may also be represented using associative arrays, and the flat data model is generally fundamental to all other models.

[1] Cieslewski, Titus et al. "Map api-scalable decentralized map building for robots." *Robotics and Automation (ICRA), 2015 IEEE International Conference on* 26 May. 2015: 6241-6247.

[2] Dymczyk, Marcin et al. "The gist of maps-summarizing experience for lifelong localization." *Robotics and Automation (ICRA), 2015 IEEE International Conference on* 26 May. 2015: 2767-2773.

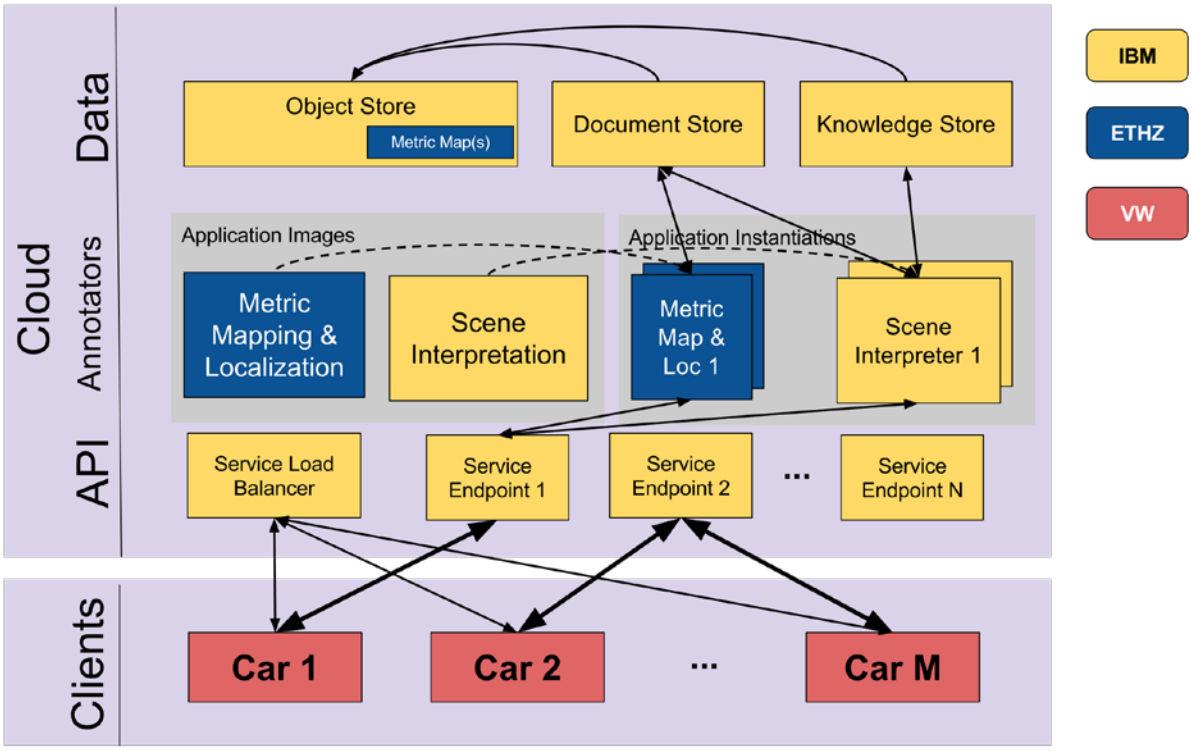

Figure 1: Architectural overview of the planned large-scale mapping and localization framework for UP-Drive.

4.2.1 Object Store (IBM, ETHZ)

As pointed out above, the object store will be the most fundamental component of the data store layer. Here, information can be stored without imposing any assumptions about the nature, format, and meaning. The cloud object store thus adheres to the basic concepts of a file system, decentrally laid out over a cluster of distributed storage devices.

Technologically, we are currently assessing different object store solutions in the focus of scalability, maintainability, and safety. Possible candidate solutions are:

- OpenStack Swift (http://docs.openstack.org/developer/swift)
- IPFS (https://ipfs.io)
- Ceph (http://ceph.com)

In particular, we will utilize the object store to maintain our metric maps in the cloud, such that they can efficiently be accessed and shared by instances of the Mapping & Localization application. Presently, metric maps are the only foreseen type of custom objects whose structure needs to be specified. This specification is given below.

4.2.1.1 Metric Map (ETHZ)

The Metric-Map contains all data necessary to localize the vehicles in 3D space precisely wrt. a common, well-defined coordinate frame of reference using cameras, IMUs and wheel-odometry sensor data.

In particular, it contains raw camera images, 3D points in space, referred to as landmarks, 2D feature descriptors, IMU and wheel-odometry data, pose-graphs representing trajectories of the vehicles and missions corresponding to individual traversals through the mapped area. Most of these elements are indexed and referenced within the Metric-Map by 128 bit Hash-IDs.

A detailed list of Metric-Map's elements and how they are referenced can the found in the table below.

Missions	A single dataset recorded during a "run" through the mapped environment can be added to the metric-map as a mission. A map can consist of multiple missions, referred to as a multi-mission-, or multi-session map.	`mission_id`
Visual Frame	A visual frame corresponds to a single camera image.	`visual_frame_id`
Keypoint	Refers to an x,y location on a camera image.	`index`
Binary Descriptor	Feature-descriptor for a keypoint. Currently either BRISK or Freak.	`index`
Visual NFrame	Collection of multiple Visual Frames recorded simultaneously (multi-camera rig).	`visual_nframe_id`
Vertex	Consists of a reference to a Visual NFrame, a 6DoF pose wrt. the mission baseframe, IMU data	`vertex_id`
Edge	Connects two vertices along a mission.	`edge_id`
Landmark	Consists of a 3D point expressed wrt. to a near-by vertex, and a list of observations (VisualNFrameId, frame index, keypoint index)	`global_landmark_id, store_landmark_id`
Map		`map_id`

At the time of writing this document, the Metric-Map is stored on disk as a list of .table and .yaml files grouped in a folder. It can be serialized and deserialized to/from memory using Google's protobuf framework.

The storage concept (file format, etc.) may be subject to change in the future.

4.2.2 Document Store (IBM)

Another component of the data store layer is the document store. Here, we consider a document to represent semi-structured information complying to the associative array data model. That is, a document will most generically be composed of a set of key-value pairs.

The important aspect of the document store is that it allows for efficient retrieval of information based on key-value queries. To these ends, the document store will be responsible for creating suitable index representations for a set of documents sharing the same keys and value types.

The distributed document store solutions currently under assessment are:

- ElasticSearch+Lucene (https://www.elastic.co)
- MongoDB (https://www.mongodb.com)
- Apache Cassandra (http://cassandra.apache.org)

4.2.3 Knowledge Store (IBM)

Last but not least, we will enable our data store layer to maintain and retrieve structured, semantic information, i.e., ontological knowledge descriptions. Whilst such descriptions may equivalently be mapped to a key-value store, the RDF or graph data models are much better suited to perform efficient queries (reasoning) over the represented knowledge. The knowledge store will thus add another level of structure assumptions and specialization to our data store layer.

According to any of the assumed standard data models for knowledge representation, i.e. RDF or graph, semantic knowledge will be composed of a set of triplets of the general form subject-predicate-object, e.g., "Stop sign - is a - traffic sign." These triplets can, for instance, be mapped to a directed graph whose connectivity encodes them as directed edges from subjects to objects, and the predicates becomes properties of these edges. In a decentralized solution, the graph is then split into multiple subgraphs, distributed over cluster nodes.

So far, we have assessed the following distributed knowledge store candidates:

- OrientDB (http://orientdb.com/orientdb)
- Apache Jena (https://jena.apache.org)

4.3 Application Layer

The application layer sits between the data store layer and the API layer, and consists in computational modules. In a distributed, virtualized framework, it will be important to distinguish between the application images, i.e., the algorithmic recipes, and their runtime instantiations. Depending on the client demand of an application's particular service, a virtually unlimited number of instances may thus be spawned from the same image.

In our UP-Drive framework, we will employ the Docker standard (https://www.docker.com) in order to represent and manage application images and their instantiations (containers).

4.3.1 Metric Localization and Mapping

The Metric Mapping & Localization application offers a list of functionality related to the creation and curation of a metric map as well as the metric localization. It is deployed as software package on the IBM cloud infrastructure and based on the multiagent-mapping framework developed at ETHZ.

The cars interact with the Metric Mapping & Localization application through a web-interface (see Service Endpoints) allowing uploading sensor data, formulating specific metric-map queries during online operation and receiving metric-map data.

The Metric Mapping & Localization application further allows interaction with available metric-maps (see Data Storage) through a console interface and through the execution of scripted jobs.

Functionality
• Visual(-Inertial) Odometry • Place Recognition • Metric localization • Bundle Adjustment • (Dense Reconstruction) • Map Summarization
Data Input
• Camera images • IMU data • Wheel odometry data • GPS data
Data Output
• 6DoF transformations between car body frame and metric map fixed frame of reference. • Arbitrary metric map data (features, landmarks, vertices, …)

Interface
Protobuf serialization/deserialization of metric map data (see Data Storage). RESTful web interface, implementing a Request-Reply pattern (see Service Endpoints).
Use Cases
• Load images, IMU and wheel odometry data of a recorded dataset, create a metric map of it (run visual-odometry estimation, Loop-Closure, Bundle Adjustment). • Augment an existing metric map with a new dataset (run metric localization/Loop-Closure, add new data structures (vertices, landmarks, …) to the map, run Bundle Adjustment). • Query 6DoF pose wrt. a global coordinate frame given a (set of) input image(s) or a set of 2D features. • Query local submap for onboard localization given a (set of) input image(s) or a set of 2D features. • Query 6DoF pose of a pose-graph vertex wrt. a global coordinate frame or any other well-defined frame of reference. • Query for a set of spatially nearby landmarks given a rough pose estimate. • Query for a set of likely co-observable landmarks given a rough pose estimate and a set of recently observed landmarks.

4.3.2 Scene Interpretation

In the scope of this deliverable, the scene interpretation application has been considered in terms of its requirements. The introductory diagram describing our cloud framework should therefore be thought of to represent this application in terms of a functional placeholder. Its details will instead be described in the upcoming UP-Drive deliverable D6.1: “Software specification and architecture for scene understanding”.

Here, we have foreseen the following requirements of the scene interpretation application:

- Access to information contained in a metric map: This can either be achieved through the Mapping & Localization API or through the more general documents API, assuming that all metric map information will be represented redundantly in the document store.
- Access to key-value information and semantic information: Retrieval and storage can be realized through the documents API and the knowledge API.

4.4 Client API Layer

4.4.1 Service Load Balancer

The service load balancer is the common entry gate for all clients in order to establish the communication with a service endpoint instance. Alternatively, a client may directly contact a

particular service endpoint instance, in which case the automated load balancing mechanism will be intentionally circumnavigated.

Once contacted by a client, the service load balancer will dispatch all further client communication to the most available service endpoint instance using the HTTP response status code 303 (see other). Means of determining the availability of a service endpoint instance need to be defined, but may involve simple measures such as the number of connected clients.

4.4.2 Service Endpoints

Service endpoint instances provide a REST (Representational State Transfer) interface to clients. Calls made to their API will generally be proxied into the cloud application and data layer of our framework. Direct client access of these layers is conceptually permitted, a design decision which we justify as follows:

- Whilst we will commit to an endpoint API at a relatively early stage of the UP-drive project, the annotator and storage API may be revised and adapted frequently. As long as clients remain restricted to exclusively communicate through the high-level endpoint API, such changes remain transparent and the clients are largely unaffected by ongoing developments.

- The component design of our planned framework entails redundancy of the stored information due to licensing and optimization constraints. Thus, consistency can only be assured by proxying API calls through a singular service endpoint. Note that this principle does not necessarily jeopardize scalability as we may run several of such endpoint instances, each serving a limited number of clients only.

In the following sections, we will sketch a specification draft of our endpoint API in terms of the different service calls we plan to provide. For formal simplicity, this draft consists of tables of the following format:

Method	The calling method, e.g., HTTP POST/PUT/GET/DELETE/...
URI	The resource identifier scheme of the call, usually of the form: http(s)://host/collection/[element_id/][?param=value]
Summary	A comprehensive summary of this API call.
Arguments	A brief description of the calling arguments: ● **element_id:** `<type>` (mandatory arguments in bold face) ● param: `<type>` (optional arguments in normal face)

	• **data:** <content-type> (data supplied with POST/PUT requests)
Response	A formal description of the call response in JSON notation: • <type> (primitive type, e.g., <string>, <boolean>, etc.) • {"member": <type>, ...} (struct) • [<type>] (array)

Calls will be grouped according to the functional component of the framework they contribute to. Each such group is represented by a unique top-level collection in the REST API: objects, maps, scenes, etc.

4.4.2.1 Object Store

To store or retrieve objects to or from the cloud framework, we plan to provide respective endpoint calls. These calls handle objects as files, referred to using a unique object identifier. The identifier format thereby depends on the storage solution implemented in the framework's data store layer and may, for instance, simply represent a hash value computed from an object's content.

Method	HTTP POST
URI	http(s)://host/objects
Summary	Store object in the cloud.
Arguments	**data**: `<content-type>` (file object of arbitrary type)

Response	`{`
	`  “success”: <bool>,`
	`  “message”: <string>,`
	`  “object_id”: <id>`
	`}`

Method	HTTP GET
URI	http(s)://host/objects/<object_id>
Summary	Retrieve object from the cloud.
Arguments	**object_id**: `<id>`
Response	`<content-type>` (file object of stored type)

Note that deleting or modifying a cloud object through the API is permitted as it would violate the append-only assumption behind our framework, putting its consistency at risk.

4.4.2.2 Document Store

Pieces of key-value information will enter the cloud framework through the documents API. We will generally assume JSON-encoded formatting of documents and thus adhere to a widely used Web standard for representing key-value information in a structured manner.

Often, it will be useful to search documents stored in the cloud by the content they provide. To allow for efficient searching, documents will be associated with indexes. Literally, a document thus becomes an element of an index collection which has been created to accelerate the search for a set of documents sharing the same key-value semantics.

We are planning to support indexing of the following value types:

- strings and numbers
- timestamps
- geospatial coordinates and polygons

Method	HTTP POST
URI	http(s)://host/documents
Summary	Create a document index in the cloud.
Arguments	**index_id:** <id> data: <application/json> (JSON-encoded index creation parameters) Index creation parameters may comprise: • index settings (sharding, replication) • mappings (field type and format information)

Response	{
	“success”: <bool>,
	“message”: <string>,
	“index_id”: <id>
	}

Method	HTTP POST
URI	http(s)://host/documents/<index_id>
Summary	Store a document in the cloud.
Arguments	**index_id:** <id> **data:** <application/json> (JSON-encoded document)
Response	{ “success”: <bool>, “message”: <string>,

	“index_id”: <id>, “document_id”: <id> }

Method	HTTP GET
URI	http(s)://host/documents/<index_id>/<document_id>
Summary	Retrieve a document from the cloud.
Arguments	**index_id:** <id> **document_id:** <id>
Response	<application/json> (JSON-encoded document)

Method	HTTP GET
URI	http(s)://host/documents/<index_id>/search
Summary	Search the cloud document store.
Arguments	**index_id:** <id> **data:** <application/json> (JSON-encoded search)
Response	<application/json> (JSON-encoded search results)

4.4.2.3 Knowledge Store

Semantic knowledge will be maintained in the knowledge store and exposed to the clients through the knowledge API. To represent semantic knowledge, we will again adhere to an open Web standard known as XML-encoded RDF (Resource Description Framework). Note that this standard follows a graph-based data model in which semantic information is stored in the form of subject-predicate-object triplets.

A mandatory operation over semantic knowledge is semantic querying. We will therefore provide an endpoint service call in the knowledge API which will allow clients to issue SPARQL (SPARQL Protocol and RDF Query Language) queries.

Method	HTTP POST
URI	http(s)://host/knowledge
Summary	Store an RDF document in the cloud.
Arguments	**data:** <application/rdf+xml> (XML-encoded RDF document)
Response	{ “success”: <bool>, “message”: <string>, “document_id”: <id> }

Method	HTTP GET
URI	http(s)://host/knowledge/<document_id>
Summary	Retrieve an RDF document from the cloud.
Arguments	**document_id:** <id>
Response	<application/rdf+xml> (XML-encoded RDF document)

Method	HTTP GET
URI	http(s)://host/knowledge/sparql?query=<encoded_query>
Summary	Issue a SPARQL query.
Arguments	**encoded_query:** <string> (URL-encoded SPARQL query)
Response	<application/sparql-results+xml> (XML-encoded query results)

Note that the above call interface will be compliant with the W3C recommendation for SPARQL queries. See also http://www.w3.org/TR/rdf-sparql-protocol.

4.4.2.4 Metric Mapping and Localization

The following list enumerates queries and responses illustrating how to interact with the metric map backend used for localization. The list is not exhaustive and may be adapted and extended to meet the needs within the projects at any time.

Method	HTTP GET
URI	http(s)://host/maps
Summary	Get all available maps.
Arguments	None
Response	`[<id>]`

Method	HTTP GET
URI	http(s)://host/maps/<map_id>/missions
Summary	Get all missions of a map.
Arguments	**map_id**: `<id>`
Response	`[<id>]`

An analogous pattern is used to retrieve vertex ids of a mission, landmark ids of a vertex, landmark ids of a mission, or map, etc.

Method	HTTP GET
URI	http(s)://host/maps/<map_id>/missions/<mission_id>/vertices/<vertex_id>/frames/<frame_index>/keypoints

Summary	Get keypoints of a visual frame.
Arguments	**map_id**: `<id>` **mission_id**: `<id>` **vertex_id**: `<id>` **frame_index**: `<int>`
Response	`[[<float>, <float>]]`(array of keypoints [x, y])

Method	HTTP GET
URI	http(s)://host/maps/<map_id>/protobuf
Summary	Get a full map as a protobuf object.
Arguments	**map_id**: `<id>`
Response	`[<int>]`

Method	HTTP GET
URI	http(s)://host/maps/<map_id>/landmarks_with_descriptors
Summary	Get (all) landmarks with descriptors from the map. If a *pose* is specified, only landmarks seen from vertices within given *radius* around this *pose* and deviating in yaw-angle less than *yaw_angle_deviation* are returned.
Arguments	**map_id**: `<id>` pose: `<application/json>` radius: `<float>` yaw_angle_deviation: `<float>`
Response	`<application/json>` (landmarks x,y,z wrt. global frame of reference, and one or more descriptors associated with each landmark.)

Method	HTTP GET
URI	http(s)://host/maps/<map_id>/pose_to_gps
Summary	Given a pose wrt. the map reference frame, a (rough) GPS pose is estimated.
Arguments	**map_id**: `<id>` **pose:** `<application/json>`

Response	`{`
	`“success”: <bool>,`
	`“lat”: <float>,`
	`“long”: <float>,`
	`“heading”, <float>,`
	`… additional available GPS data.`
	`}`

Method	HTTP POST
URI	http(s)://host/maps/<map_id>/actions/<action_tag>
Summary	Run actions on a map.
Arguments	**map_id**: `<id>` **action_tag**: `<”bundle_adjust”|”summarize”|”loop_close”|...>` **data**: `<application/json>` (action parameters, t.b.d.)

Response	`{`
	`“success”: <bool>,`
	`“message”: <string>`
	`}`

Method	HTTP POST

URI	http(s)://host/maps/<map_id>/actions/localize
Summary	Query the map for a global localization pose estimate.
Arguments	**map_id**: `<id>` **data**: `<application/json>` (images, or keypoints with descriptors, potentially a rough guess of the vehicle's pose.)

Response	`{`
	`"success": <bool>,`
	`"p_W_I": [<float>, <float>, <float>],`
	`"q_W_I": [<float>, <float>, <float>, <float>]`
	`}`

4.5 Client Applications

4.5.1 Online Localization Module

This software module is deployed on the car(s) and allows localizing the car precisely wrt. a well-defined coordinate frame of reference. For that, camera images, as well as IMU and wheel-odometry data are fetched from the DDS gateway and localization queries are issued against a metric-map either available online in the backend or downloaded from the backend prior to departing for the current operation. The connection to the designated service endpoint on the backend is established through a RESTful web-interface.

Sensor and/or online localization data may be transmitted back to the backend on the fly.

Functionality
• Online localization of the car.
Data Input
• Camera images • IMU data • Wheel odometry data • (GPS)
Data Output

• 6DoF transformations between car body frame and metric map fixed frame of reference. • Sensor and/or online localization data (images, localization estimates, landmark-keypoint matches, keypoints, descriptors, etc.).
Interface
RESTful web interface.

4.5.2 Dataset Uploader

This software module is deployed on the car(s) and allows uploading a collected dataset (.dat file) to the backend after finishing an operation for post-processing or archiving. The connection to the designated service endpoint on the backend is established through the RESTful web-interface described above, utilizing the objects API.

Functionality
• Uploading of a dataset (.dat file).
Data Input
• .dat file with all available sensor data as well as all computational and log output of all modules in operation on the car during the recording.
Interface
RESTful web interface.

5 USE-CASE ANALYSIS

In order to demonstrate the usage of our proposed framework, we describe some exemplary use-cases from section 3 in detail.

- **Query global pose:**

  Assuming the vehicle has no prior knowledge about its pose in the world, we aim at querying the metric-map for a (rough) estimate of the 6DoF pose between the vehicle's frame of reference and some well-defined global frame of reference.

  For this, the Online Localization Module fetches the latest image(s) from the DDS message bus and issues a query to the cloud-based backend. Only a Map ID and the host-name/IP of the Service Load Balancer are needed as prior knowledge which are both assumed to be known in any case.

  An HTTP GET web query is issued with the following URL
  http(s)://host/maps/<map_id>/actions/localize
  and the image(s) contained in the data parameter.

  The corresponding service endpoint in the backend processes the query by delegating it to the Metric Localization & Mapping application.

  The result, namely as 6DoF pose estimate wrt. a well-defined global coordinate frame, is returned to the vehicle as a response to the web query in a JSON message.

- **Query near-by metric-map data for on-car localization:**

  Assuming there is rough estimate of the vehicle's global pose, close-by landmarks with descriptors, or any other metric mapping data, ought to be fetched in order to run local localization on the vehicle in the near future. In addition to that, we again assume the Map ID as well as the host-name/IP of the Service Load Balancer to be known a priori.

  The following HTTP GET web-query is issued:
  http(s)://host/maps/<map_id>/landmarks_with_descriptors
  with the optional parameters pose and potentially radius and yaw_angle_deviation specified.

  As a result, a JSON message is returned containing the 3D position of the near-by landmarks wrt. the global frame of reference together with one or more feature descriptor for each landmark.

- **Upload a dataset for archiving:**

  In this use case, we intend to upload a dataset file in the .dat format for later retrieval from the cloud object store. The retrieval task assumes that we will be able to resolve the object identifier of the cloud object based on characteristic information. This information may involve, but does not have to be limited to, a description of the acquisition time and location of the dataset.

  First, an HTTP POST call with the dataset file will be issued to:
  http(s)://host/objects
  This will store the dataset file in the object store and return the object identifier, e.g., a hash value.

  Then, another HTTP POST call with a JSON document containing the returned object identifier, the time, and the location information will be directed to
  http(s)://host/documents/datasets
  This will store and index the particular document describing the uploaded dataset in the datasets document index for later retrieval through the documents search API.

6 CONCLUSIONS

In this document, we have evaluated the requirements for the localization and mapping framework, derived concrete specifications thereof, and presented an adequate infrastructure and software architecture framework. The proposed set-up guarantees scalability, online and offline accessibility and transparency, allowing a conceptual separation between client modules located on the vehicle and mapping applications and data in the cloud-based backend. Through a RESTful web-interface, a state-of-the-art protocol to issue queries and to exchange data is employed.

The working principle is further illustrated by a detailed analysis of some key use-cases.