UP-Drive
Automated Urban Parking and Driving

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

Deliverable D3.2

First Development and Integration Cycle of Cloud Infrastructure

Type: R

Status: final

Lead contractor for this deliverable: IBM

Due date of deliverable: 31.12.2016
Actual submission date:

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 3.2: First development and integration cycle of cloud infrastructure. It documents the cloud infrastructure that has been selected and implemented within the project.

Building on D3.1, this deliverable focuses on the Gitlab source code management system, the Swift object store and OpenStack cloud compute infrastructure functionality.

CONTRIBUTING PARTNERS		
Document Manager	**Company/Organisation**	**Name**
Partner 1	IBM	Martin Rufli
Partner 1	IBM	Ralf Kaestner

REVISION TABLE		
Document version	**Date**	**Modified sections - Details**
0	07.12.2016	Initial version
1	09.12.2016	Final version

Table of Contents

1 Introduction 4
2 Source Code Management, Wiki, and Task Tracking System............................ 4
3 Cloud Object Store.. 4
3.1 Requirements 4
3.2 System Selection & Overview 5
3.3 System Access & Interaction.. 5
4 Cloud Compute Stack 5
4.1 Requirements 5
4.2 System Selection & Overview 6
4.3 System Access & Interaction.. 6
5 User Administration & Management 6
6 Information Security Measures 7
6.1 Security of the Hosting Environment 7
6.2 Security of Partner-owned Services 8

1 Introduction

This deliverable corresponds to Task 3.2: First development and integration cycle of cloud infrastructure. It documents the cloud infrastructure that has been selected and implemented within the project. Building on D3.1, this deliverable focuses on the Gitlab source code management system, the Swift object store and OpenStack cloud compute infrastructure functionality.

2 Source Code Management, Wiki, and Task Tracking System

A functional and effective source code management (SCM) system, project Wiki and task/issue tracker forms a basic need of any project spanning multiple parties and collaborators. UP-Drive deliverable 3.1 in detail describes the selection of the SCM, Wiki and issue tracking solutions based on therein specified project requirements. For further information, we refer to this deliverable.

Whilst the selected SCM and documentation facilities have already been tightly integrated into the UP-Drive research and development workflow, IBM continuously maintains user documentation for the project partners. This documentation contains solutions to technical questions as well as best practices, and has itself become part of the project Wiki. For completeness, 0 and Appendix B provide a present snapshot of the documentation for the UP-Drive collaborative development environment and the UP-Drive SCM system, respectively.

3 Cloud Object Store

Cloud object storage is a highly scalable storage architecture which manages data as objects rather than files or blocks. An object is typically composed of the data itself and a variable amount of metadata, and can be referred to by a globally unique identifier. Unlike files in a file system, objects are thereby not organized in a hierarchy, but are logically assigned to containers. At the level of such containers, it is then possible to specify object permissions in the form of access control lists.

3.1 Requirements

The following requirements on the UP-Drive cloud object store have been identified and agreed on by the project partners:

- Support for programmatic access via command line tools as well as from within programs
- Support for GUI-based access via (third-party) GUI tools
- Support for bulk up/downloads as well as that of nested (pseudo-) directory structures
- Scalability to a total storage size of up to 100TB

3.2 System Selection & Overview

Several cloud object store implementations exist which can be used off-the-shelf. For the UP-Drive project, is has been decided that all data shall physically be hosted by IBM Research, at their site in Ruschlikon, Switzerland. This site is particularly experienced with the OpenStack cloud infrastructure-as-a-service platform. Thus, OpenStack Swift, which represents a loosely integrated part of OpenStack, has been selected to implement the UP-Drive cloud object store. A brief overview of OpenStack and its service components is provided in Appendix C.

OpenStack Swift is a scalable, redundant storage system which writes objects and files to multiple disk drives spread throughout file servers in a data centre. The management of objects and storage thereby remains entirely non-transparent to the user: The OpenStack software is responsible for ensuring data replication and integrity across the cluster to achieve the aforementioned properties.

3.3 System Access & Interaction

In addition to the OpenStack Swift service implementation, several client solutions exist which allow the users or other services to interact with the object store through a RESTful API via the encrypted HTTPS protocol. To authenticate the object store service endpoint to the UP-Drive users, we employ a wildcard certificate which is tied to the project's top level domain (https://up-drive.eu). For a more detailed discussion of Swift and the client options considered for UP-Drive, the reader may refer to the Wiki excerpt under Appendix D. It describes the basics of accessing and interacting with the UP-Drive object store and provides installation and setup instructions for suitable client solutions.

4 Cloud Compute Stack

The term cloud computing pertains to the highly scalable provisioning of computational resources, i.e., CPUs, RAM, communication, and disk space. In practice, this is achieved by virtualizing such resources on top of the physical infrastructure which, in this context, is often referred to as bare metal. Modern operating systems provide the tools to virtualize a physical machine's resources without inflicting a noticeable reduction of performance, and thus efficiently allow it to host a varying number of virtual guest machines. To the cloud computing user, provisioning of virtual machines (VMs) once again remains entirely non-transparent which ensures ultimate scalability.

4.1 Requirements

The following requirements on the UP-Drive cloud computing facilities have been identified and agreed on by the project partners:

- Support for programmatic access via command line tools as well as from within programs

- Support for GUI-based access via the Web browser
- Support for SSH access to the VMs
- Scalability to a total of up to 200 physical CPUs and 2TB of physical RAM
- Virtual machine flavours of up to 16 virtual CPUs and 128GB of virtual RAM
- Maximum over-provisioning of factor 2 for CPUs (i.e., not more than 2 virtual CPUs per physical CPU) and factor 1 for RAM (i.e., equivalent amounts of virtual and physical RAM)

4.2 System Selection & Overview

To complement the UP-Drive object store implementation, we have once again selected OpenStack as cloud computing platform for the project. To facilitate a full virtualization of all required system resources, including machines, networking hardware, and disk space, OpenStack offers a suite of services. These services and their roles are further detailed in Appendix C.

4.3 System Access & Interaction

Access to the various OpenStack services related to the UP-Drive cloud compute infrastructure will be provided via a Web GUI as well as a command-line client connecting to the service-specific REST endpoints. Thus, virtual machines may for instance be created or deleted from a Web form or by simply issuing terminal commands. At the time of writing this deliverable, IBM is about to author a user documentation which will explain the steps required to work with virtual cloud machines.

5 User Administration & Management

Due to security concerns this section of the deliverable is removed from the public version of the document.

6 Information Security Measures

Due to security concerns this section of the deliverable is removed from the public version of the document.

6.1 Security of the Hosting Environment

Due to security concerns this section of the deliverable is removed from the public version of the document.

6.2 Security of Partner-owned Services

Due to security concerns this section of the deliverable is removed from the public version of the document.

Appendix A GitLab CE

Gitlab CE is an open-source project for hosting and managing Git repositories. It is comparable in functionality to Github, but is hosted on premise, locally. Besides git-based code versioning, it offers additional functionality to support development, testing and deployment; in short devops. This includes a wiki, an issue tracking system, and an automated testing facility (gitlab-ci).

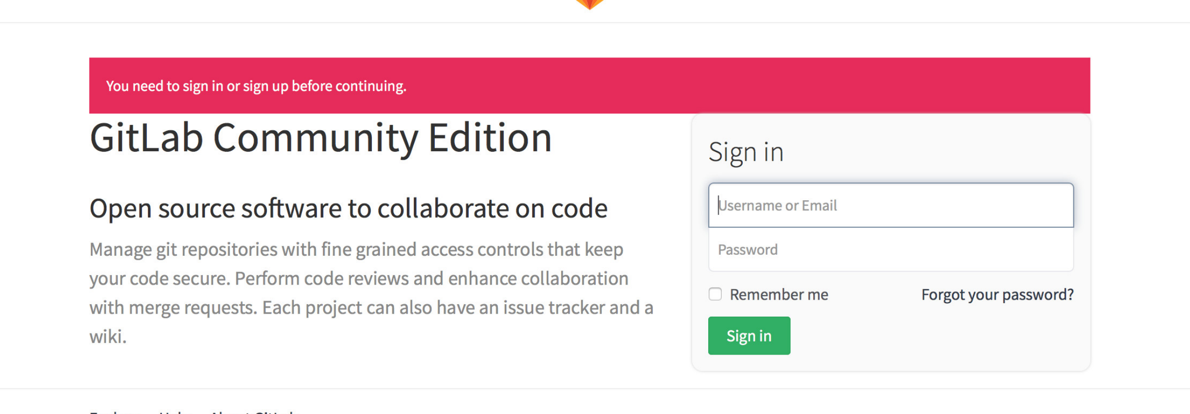


Figure 1: Gitlab CE logon screen.

Below we provide an introduction to the most important functionality of Gitlab CE. For further reading, we refer the user to the official documentation at https://docs.gitlab.com/ce/README.html.

A.1 Accounts

Login to Gitlab is via username and password. Users are assigned rights to read and/or write and collaborate on repositories on a per project or at an aggregate group level as described below.

After logging in users are greeted by an overview screen displaying projects they have access to with recent activity. On the left of the screen the most important functionality of Gitlab is easily navigable in the form of panes.

A.2 Projects

The project pane lists project the user has access to sort by recent activity. On the top right, projects can be full-text searched and selected, which opens a project specific view.

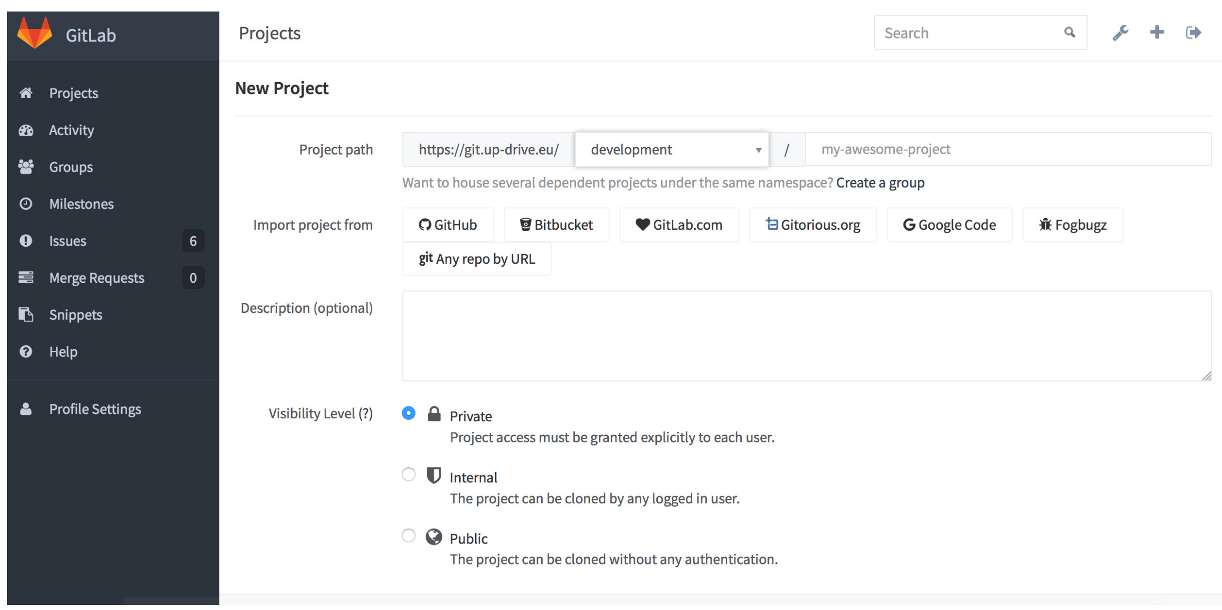


Figure 2: Creation of a new project.

Alternatively, by clicking *New Project*, a new window appears (see Figure 2) where a project holding a new git repository can be created. Select the Gitlab group under whose namespace the project is to be created. This choice governs the access rights you grant other registered users of the platform. Further access rights can later be granted at the project level. Add a short but meaningful description of your new project. This information is later displayed at the project's overview page. Finally decide on a visibility level for the new project. By default it is set to *Private*. The private setting ensures that only users who have been explicitly granted access (at the project or group level) can access the project. The *Internal* setting allows all registered UP-Drive users to access the project. The *public* setting makes the project available on the public web.

ATTENTION: Only select Public for repositories that are intended to be available as open source to the public!

After hitting *Create Project*, the project is created and can be pulled and pushed to.

When selecting a project from the list of available projects, one is transferred into a project-specific view -- again composed of a range of functionality in the form of a column of tabs ta the left side of the screen. Clicking *GitLab* at the top left brings you back to the main entry point.

A.2.1 Activity

This pane shows recent activity of you and your collaborators at the project level.

Files

The files pane allows browsing the underlying git repository structure and even modifying existing files or adding new folder and file structures interactively. All changes performed in this view are translated into underlying git commits.

A.2.2 Commits

This pane shows a log of recent commits to the underlying repository - filtered by branch. The *Network* pane provides an intuitive graphical representation of the git branch structure. The *Compare* pane allows comparing different branches and is useful to visualize diffs between them at a file level. The *Branches* tab lists the active branches and allows to interactively creating new branches. Similarly, the *Tags* pane allows for the creation of git tags which are useful to tag releases.

A.2.3 Milestones & Issue

These panes provide a project-specific view at existing milestones and issues. As above, new issues and milestones can be created interactively to track the progress of the project. We describe this functionality in more detail below.

A.2.4 Merge Requests

Merge requests (in Github called pull requests) are a convenient tool for managing branches and ensuring code quality via code review. A typical workflow is for a developer to create a merge request of a feature branch back into the main branch. The developer lacks the privileges to push to the main branch directly, however. She thus creates a merge request which triggers a code review process. In this process the reviewers comment on the code directly on the merge request pane. They may request further changes and the corresponding commits are logged in the conversation. Finally a user with sufficient privileges merges the branch and thereby closes the merge request.

A.3 Wiki

Each project is by default equipped with a wiki. The wiki is a convenient tool for documenting the project beyond the single README.md file that is typically found in the project trunk. The wiki is itself versioned and supports a large range of formatting options in the markdown syntax.

A.4 Settings

The setting pane allows users with sufficient privileges to modify basic project settings, including the renaming of the project, changing its visibility, protecting branches, and archiving or deleting the project once it is no longer active.

A.5 Groups

The groups pane provides an overview of the existing Gitlab groups. Groups are a convenient tool for grouping users with common access requirements. As an end-user, you may be interested in creating per partner, or per task groups to facilitate the access management of your projects.

Via the button *Add Group* a new group can be defined. By clicking on existing groups for which you have sufficient privileges, users can be managed (i.e., added or removed).

A.6 Milestones & Issues

Gitlab milestones are used to mark deadlines for single or cross-project functionality. They typically coincide with code releases. Milestones are assigned a specific due date and are composed of a set of issues or tasks which need to be resolved to mark the milestone complete. Issues on the other hand represent individual aspects that need to be worked on. Typically these take the form of bug fixes, feature additions, or testing requests. Each of them is typically worked on in a separate branch. When finished, a pull request is initiated. When merged, the issue is closed.

Tip: Add references in issue or merge request descriptions or in comments. This will update the issue with info about anything related.

```
To reference an issue: #123¬
To reference a MR: !123¬
To reference a snippet $123¬
```

You can make all the same references in comments or merge requests as you can in an issue description.

The purpose of the WIP: (work in progress) merge request is to encourage early code review, so you can share your work with others. In this way you can reference related work, and hopefully reduce duplicate effort.

You can mention people with @username and ask them to provide feedback in merge requests or issues.

We already encountered Milestones and issues above at the project level and this apparent duplication hence may at a first glance seem a bit confusing. In this pane, milestones and issues are presented at an aggregate level (that is grouped for all projects one has access to). This is particularly useful to get an overview on issues assigned to a specific user, or cross-project issues that need to be resolved before reaching a specific milestone.

A.7 Profile Settings

This pane governs everything related to the user's account settings. Of particular mention is the ability to reset one's password.

Appendix B Git SCM

Git is an open-source version control system that can vastly simplify or infuriatingly complicate your life as a developer. Therefore, we would like to provide some guidelines and best practices for your work with Git.

B.1 Git Command Line Client

If you intend to use Git from the command line, you may find a complete reference in the Pro Git book by Scott Chacon and Ben Straub. The detailed documentation of the Git command line client ranges from the installation all the way to the different Git commands and their implications.

Please note that under Linux, the Git man pages provide a quick reference to the more experienced user.

B.2 Git Conventions for the UP-Drive Project

Please read the best practices first if you are unfamiliar with the Git workflow.

B.2.1 Master Branch

The `master` branch of a Git repository is considered holy. Other developers collaborating with you on a software project will rely on the fact that the `master` branch compiles and contains code modules which have been tested to work. The `master` branch should therefore be moderated carefully and wisely by experienced developers.

B.2.2 Branch Naming Conventions

Code contributions should generally be maintained within dedicated branches other than the `master` branch. To simplify the navigation of these dedicated branches, please comply with the following naming rules:

- Use the snake_case syntax convention for branch names and keep them as short and precise as possible.
- A contribution fixing a known bug in the current `master` branch should be contained in a branch named after `fix/my_nasty_bug`, where`my_nasty_bug` provides a meaningful reference to the bug.
- A requested feature contribution to the current `master` branch should be contained in a branch named after `feature/my_awesome_feature`, where `my_awesome_feature` provides a meaningful reference to the features.
- If you are just diverting your developments from the current `master` branch without any of the above reasons, e.g., for refactoring interfaces, improving performance and stability, or for trying out some new concepts, do so in a branch named after`develop/my_messy_tryouts`.

- Similarly, branches may be created for other purposes using the namespace prefix convention on branch names, e.g., release/1.5.2,experiment/2016-01-01, demo/iros16, or alike.

B.2.3 Merge Requests

If you seek to introduce your contributions, be it a bug fix, feature, or new development of another kind, to the master branch, use merge requests to express your interest to others before they get inevitably confronted with unexpected consequences. In order to file your merge request, you may navigate to the repository containing your contribution in the GitLab Web front-end, call up the entry "Merge Requests" on the sidebar menu, and then file your request by clicking on the button "+ New Merge Request".

Whilst your request is pending, other collaborators may be notified and given sufficient time to review, verify, and test your contributions. When this time has passed, a qualified master branch moderator will accept your request.

B.2.4 Issues and Milestones

To report bugs or request new features, use the 'Issues' or 'Milestones' forms of a project, respectively. Both can be found in the sidebar menu of a project's view in the GitLab Web front-end.

Reporting bugs or requesting features this way (rather than by just telling someone who you think should be held responsible) has the unchallenged advantage of rendering your interests documented. Conversely, the assigned developer can easily keep track of his or her duties by calling up a list of personal to-do items. Also, note that open issues and milestones may be discussed in the Web interface or via mail replies.

B.2.5 Large Files

Larger files, usually exceeding the size of a few Megabytes, do not belong into a Git repository. The reasons for this are manifold, but the most important implication is a noticeable reduction of Git's performance when operating on your local copy of the repository. Recently, the Git developer community has therefore released an extension known as Large File Storage. This extension bypasses the upload of files tagged as being large to the repository itself and instead pushes them to a native filesystem location on the Git server. For version control, so-called text pointers replace the large file inside the actual repository.

Our GitLab server has the Large File Storage feature activated, and we encourage all collaborators to use it where appropriate. For instructions on how to install and use the git-lfs client extension, please consult the official Git Large File Storage homepage.

B.3 Git Best Practices

This section follows T. Guenther, *“Learn Version Control with Git: A step-by-step course for the complete beginner” (2014)*.

B.3.1 Commit Related Changes

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other team members to understand the changes and roll them back if something went wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

B.3.2 Commit Often

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it is easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard both to solve conflicts and to comprehend what happened.

B.3.3 Do not Commit Half-Done Work

You should only commit code when it is completed. This does not mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But do not commit just to have something in the repository before leaving the office at the end of the day. If you are tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git's "Stash" feature instead.

B.3.4 Test Before You Commit

Resist the temptation to commit something that you "think" is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing or sharing your code with others.

B.3.5 Write Good Commit Messages

Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions: What was the motivation for the change? How does it differ from the previous implementation? Use the imperative, present tense ("change", not "changed" or "changes") to be consistent with generated messages from commands like git merge.

B.3.6 Version Control is not a Backup System

Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see "related changes") - you should not just cram in files.

B.3.7 Use Branches

Branching is one of Git's most powerful features - and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas, etc.

B.3.8 Agree on a Workflow

Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, git-flow, etc. Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates' personal preferences. However you choose to work, just make sure to agree on a common workflow that everyone follows.

Appendix C OpenStack

OpenStack is a cloud operating system that allows controlling large pools of compute, storage, and networking resources throughout a data center. All of these components are managed through a dashboard which gives administrators control while empowering their users to provision resources through a web interface.

C.1 Component Overview

OpenStack consists of a collection of open source software, as illustrated in Figure 1, which allows users to perform tasks on the cloud.

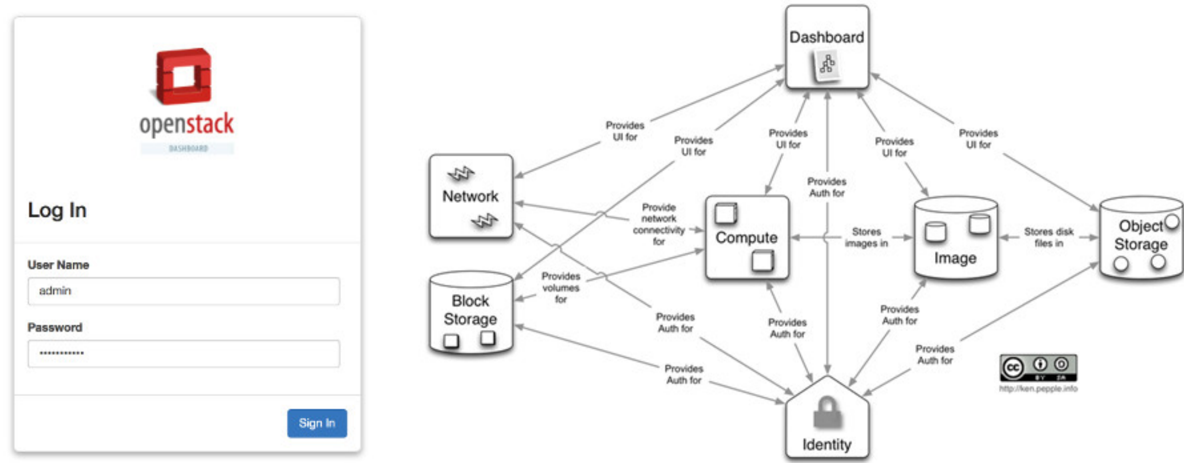


Figure 1: Left: OpenStack login screen. Right: OpenStack component overview and their interaction.

Below we follow the excellent guide at Cloudify and therewith provide an introduction to the most important elements of this collection.

Name	Description
Dashboard (Horizon)	This component provides a web-based portal to interact with all the underlying OpenStack services. Pools of resources can be consumed and managed from this single place, including the creation of virtual machines, configuration of networks and management of volumes.

Name	Description
Compute (Nova)	OpenStack compute is the component which allows the user to create and manage virtual servers using the machine images. It is the brain of the Cloud. OpenStack compute provisions and manages large networks of virtual machines.
Block Storage (Cinder)	This component provides persistent block storage to running instances. The flexible architecture makes creating and managing block storage devices very easy.
Object Storage (Swift)	This component stores and retrieves unstructured data objects through the HTTP based APIs. Further, it is also fault tolerant due to its data replication and scale out architecture.
Networking (Neutron)	It is a pluggable, scalable and API-driven system for managing networks. OpenStack networking is useful for VLAN management, management of IP addresses to different VMs and management of firewalls using these components.
Identity Service (Keystone)	This provides a central directory of users mapped to the OpenStack services. It is used to provide an authentication and authorization service for other OpenStack services.

Name	**Description**
Image Service (Glance)	This provides the discovery, registration and delivery services for the disk and server images. It stores and retrieves the virtual machine disk image.
Orchestration (Heat)	This component manages multiple Cloud applications through an OpenStack-native REST API and a CloudFormation-compatible Query API.
Database as a Service (Trove)	Trove is Database as a Service for OpenStack. It's designed to run entirely on OpenStack, with the goal of allowing users to quickly and easily utilize the features of a relational database without the burden of handling complex administrative tasks.

C.2 Getting Started

C.2.1 Dashboard

Web-based interaction with OpenStack is handled via the Horizon Dashboard. Login is via username / password. Once logged in, an overview pane as illustrated in Figure 2 presents itself. The GUI is structure into a column of tabs on the left, via which the various compute, storage, and networking functionality is reached. By default, an overview of used and available compute is presented.

C.2.2 Compute

The overview pane provides at a glance a list of available and used compute infrastructure. The instances pane provides further detail on existing instances in list format. Instances are instantiations of an image. Existing instances can be manipulated (shutdown, restarted, snapshotted, etc.) via the *Actions* button. New instances can be created and launched via the *Launch Instance* button.
The *Volumes* pane lists active volumes - persistent storage that is typically mounted into instances during boot up. The *Images* pane shows a list of all base images that are available and from which instances may be instantiated. It the *Access & Security* pane, for each instance, access rules can be specified.

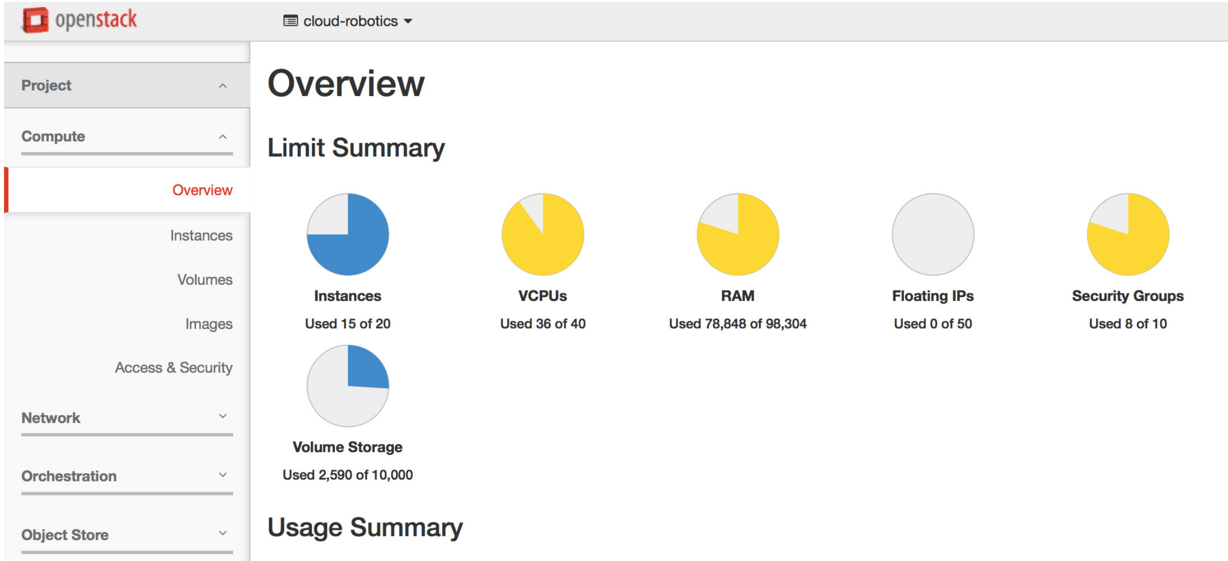


Figure 2: Overview of the OpenStack Dashboard entrance view.

C.2.3 Network

The overview pane provides a GUI representation of the available networks and in particular the interconnectivity between the running instances. Under the *Networks* pane, new networks can be defined to which existing instances can be associated.

C.2.4 Orchestration

Orchestration is beyond the scope of this basic overview. For a more in depth treatment, including examples, you may refer to the linked document.

C.2.5 Object Store

The overview pane lists existing object storage containers. Existing containers can be browsed and new ones generated via the respective buttons. Within these containers files can be up and downloaded and pseudo folders generated either via the Dashboard GUI interface, or programmatically via API. For a more in-depth treatment on the API, please refer to object store document, Appendix D.

Appendix D Swift Object Store

D.1 Overview

D.1.1 Basics

The UP-Drive object store is based on OpenStack Swift, a true cloud storage solution designed for scalability, durability, and availability. Swift is part of the OpenStack cloud computing software. It is entirely operated through the Hypertext Transfer Protocol (HTTP) and thus provided as a Web service with REST interface.

D.1.2 Identity Service

In the UP-Drive implementation, Swift utilizes the OpenStack identity provider called Keystone to securely authenticate users. The following Keystone concepts are relevant to Swift:

- **Project** - A Keystone project (sometimes also called tenant) owns resources (e.g., storage in Swift) and maps to Swift accounts.
- **User** - A Keystone user is a consumer identity (e.g., a person) and maps to Swift users.
- **Role** - A Keystone role represents a user's personality (e.g., project member) and assigns rights and privileges to it. In Swift, a Keystone role maps to a group.

For more information concerning the OpenStack terminology, you may want to consult the OpenStack glossary.

D.1.3 Storage Service

Swift organizes storage into *containers* (sometimes also called buckets). You can think of a container in terms of a top-level directory since containers cannot be nested such as directories in a POSIX filesystem. A Swift project may provide an unbounded number of containers.

A file as you may know it represents an *object* in Swift and is assigned to a container. In absence of the concept of a directory, this assignment is totally flat, i.e., there is no filesystem-like hierarchical ordering of objects beyond the container level. Both Swift containers and objects can be addressed by human-readable names.

To allow for some intuitive organization of the objects within a human-moderated Swift container, it is often helpful to adopt the concept of *pseudo-folders*. As the name suggests, pseudo-folders are not actually implemented in Swift. In fact, from a Swift perspective, there is no difference between an actual file object and an object you may refer to as a folder. Instead, pseudo-folders are simply a conventional concept agreed upon by the users of a container. You can read more about Swift pseudo-folders in the documentation, but the essence is that object names may contain the forward slash character / as a delimiter between the components of a path-like expression.

An important limitation you may not even notice when working with Swift is that the size of a single object is limited (usually to about 6 GiB). In order to support for objects exceeding this size limit, Swift segments them into pieces and stores a so-called *manifest* with references to the individual segments (which again represent Swift objects). Note that the transfer of large objects and the creation of the manifest are transparently handled by Swift clients. There virtually exists no limit on the size of a segmented object (apart from the fact that the manifest may not exceed the size of a single Swift object). Often, object segmentation even results in a performance gain for the client as segments may be transferred in parallel.

D.2 UP-Drive Configuration

IBM manages the UP-Drive object store and is thus responsible for the creation of users, roles, projects, etc. In the following paragraphs, you will find a short wrap-up of the current object store layout and the related identity configuration.

Note: Each UP-Drive consortium member whose access to the object store has been activated should have received an automatically generated e-mail containing its access credentials. If you have not yet received such a mail, but wish to be granted access to the object store, please contact IBM.

D.2.1 Users

UP-Drive object store users are identified by their e-mail addresses and thus have been assigned personal access credentials.

Please make sure to keep your personal credentials confidential at all times. Never share them with others or use them to authenticate against other services. Always store them in a safe location.

D.2.2 Groups

To manage access privileges through Keystone roles, we have assigned users to the following groups:

- `up-drive` - All UP-Drive users are members of this group.
- `PARTNER_ACRONYM` - Any UP-Drive user is also a member of the group named after the acronym of the partner it is affiliated with (i.e., `ibm`,`vw`,`eth`, `cvut`, or `utc`).

D.2.3 Projects

The following Keystone projects have been created:

- `up-drive` - The project shared by all UP-Drive users.
- `FIRST_LAST` - For any UP-Drive user, a personal project has been created, named after its first and last name (e.g., `john_doe`).

D.2.4 Roles

Access privileges for UP-Drive users and groups to projects have been defined through the following Keystone roles:

- up-drive - This role controls access of all users in the up-drivegroup to the up-drive project resources.
- PARTNER_ACRONYM - These roles, one for each partner in UP-Drive, control access of the PARTNER_ACRONYM group to the up-driveproject resources.
- admin - This dedicated role controls administrative access for the admin user to all projects.
- user - This special role grants Swift operator privileges to each UP-Drive user on its personal project and thus allow it to create and delete containers within its personal project and to control access to these containers.

D.2.5 Containers

Several containers have been created with different access rights to ensure safe operation of the UP-Drive object store. The specific per-project configurations are as follows:

- up-drive project
  - shared - A shared container with read/write access granted to all UP-Drive users. In this container, an active project member may create, delete, read, and write objects. Note, however, that other project members will be able to do the same. Thus, objects may unintentionally get modified or lost.
  - shared_segments - A container for object segments with the same access rights as shared.
  - PARTNER_ACRONYM - A per-partner container with read-only access granted to all UP-Drive users and read/write access granted to any user affiliated with that partner. Use this container to safeguard your data against modification or deletion by users belonging to other partners.
  - PARTNER_ACRONYM_segments - A container for object segments with the same access rights as PARTNER_ACRONYM.
- FIRST_LAST project
  - up-drive - An example container in the personal project of each user. Since you have been assigned the role of a Swift operator in this project, you may create/delete your own containers and control access to these containers. You may, for instance, create a container to share data with selected other project partners or individual project members. Use this container (or other containers in your personal project which you may define yourself) as a playground.
  - up-drive_segments - A container for object segments with the same access rights as up-drive.

D.3 Getting Started Guide

In this section, we outline the operation of different client solutions with the UP-Drive object store. Depending on your system requirements and your level of expertise, you may choose to learn about command line access, graphical access, or programmatic access to Swift containers and objects.

This is a brief overview of available Swift client solutions with links to the individual sections.

Client	**Use**	**License**	**Language**	**Supported Platforms**
Swift Command-line Client	Command-line and programmatic access	Apache License	Python	Cross-platform
Cyberduck Graphical Client	Graphical access	GPL	Java	Windows and macOS
Cloudfuse Filesystem Client	Filesystem access	MIT License	C	Linux and macOS
Curl Command-Line Tool and Library	Command-line and programmatic access	MIT/X Derivate License	C	Cross-platform

D.3.1 Swift Command-line Client

The official Swift client is a Python library with a simple command-line tool provided by the OpenStack project. Detailed documentation of the client and its capabilities can be found here.

Installation

Though slightly platform-dependent, a cross-platform installation procedure is provided through the pip) package manager for Python. On Ubuntu Linux, Debian packages are available from the official sources.

To install the Python Swift client using pip, you may have to first install Python according to the procedure outlined for your system. The pippackage manager will then come alongside your Python installation.

Once pip is available on your system, issue the following commands documented by the pip user guide:

```
pip install python-swiftclient

pip install python-keystoneclient
```

Note that we will also require the official Keystone client in order to authenticate the Swift client against the UP-Drive Keystone identity service.

Authentication

To verify that you can authenticate against our Keystone service, you may use the following swift command, filling in your personal credentials:

```
swift --os-auth-url https://identity.up-drive.eu/v3 --os-identity-api-version 3 --os-username YOUR_EMAIL --os-password YOUR_PASSWORD --os-project-name up-drive auth
```

If the authentication succeeds, the client will print export statements for specific environment variables to your terminal. Under Linux and macOS, these statements may be executed by wrapping the above command into aneval statement, i.e., eval $(swift ...). The URL of the Swift endpoint and your private access token will thus be added to the environment, saving you from having to re-enter them with each swift command.

The above commands authenticates you for accessing the shared up-drive project. If you wish to access your private project, you may replace the --os-project-name up-drive arguments by --os-project-name FIRST_LAST, e.g., by --os-project-name john_doe.

Spoiler alarm: Once you have added the storage URL and the access token to your environment using `eval`, the environment variables will take precedence over the command-line arguments supplied to `swift`. Thus, the use of `eval` is not recommended if you wish to switch between different projects.

To keep the `swift` syntax short throughout the following paragraphs, we will further assume that you have exported the authentication variables using the procedure explained above. Note that your authentication token will expire after some time such that you may have to re-authenticate the client.

Automation

In order to semi-automate logon and prevent the user from having to repeatedly re-type user arguments in each swift call, one may create a set of bash scripts (one per container), which can then be sourced (similar to how ROS sources environment variables). A sample bash script (here at the example of the up-drive container) may look as follows. We assume that it is called `up-drive.bash` and located in `/opt/up-drive/up-drive.bash`.

```
#!/bin/bash¬
export OS_AUTH_URL=https://identity.up-drive.eu/v3¬
export OS_IDENTITY_API_VERSION=3¬
export OS_USERNAME=YOUR_EMAIL¬
export OS_PROJECT_NAME=up-drive¬
```

Periodic authentication is then simplified to:

```
source /opt/up-drive/up-drive.bash¬
swift --os-password YOUR_PASSWORD auth¬
```

Container and Object Listings

Assuming that user environment variables have been exported as described above, to list the containers of the project you have been authenticated for, use the command

```
swift list¬
```

This command will intentionally fail with an "Access denied" error for the `up-drive` project, but should neatly list the `up-drive` and `up-drive_segments`containers for your personal project.

-

The command for listing the objects in a container is very similar and looks as follows:

```
swift list CONTAINER
```

When being authenticated for the up-drive project, try to list the shared container or any partner's PARTNER_ACRONYM container. These listings should succeed as you have at least been granted read access to the containers.

Single File Upload, Download, and Deletion

To upload a file object to any container you have been granted write access for, use the command:

```
swift upload CONTAINER PATH_TO_LOCAL_FILE
```

This command will create an object named exactly afterPATH_TO_LOCAL_FILE in CONTAINER which you may verify by the swift list ... command. To use an object name different from PATH_TO_LOCAL_FILE, employ

```
swift upload --object-name OBJECT_NAME CONTAINER PATH_TO_LOCAL_FILE
```

Analogously, download an object from a container by issuing the command:

```
swift download [-o PATH_TO_LOCAL_FILE] CONTAINER OBJECT_NAME
```

Last but not least, delete an object from a container through:

```
swift delete CONTAINER OBJECT_NAME
```

Caution when deleting objects: Object deletion cannot be undone in Swift. So think twice and delete with care!

Directory Upload, Download, and Deletion

With the swift command, you may also operate on entire directories and Swift pseudo-folders instead of on single files and objects.

To upload the contents of a filesystem directory, use:

```
swift upload [--object-name PSEUDO_PATH_PREFIX] CONTAINER
PATH_TO_LOCAL_DIRECTORY
```

With this command, PATH_TO_LOCAL_DIRECTORY is replaced byPSEUDO_PATH_PREFIX to generate the names for individual objects corresponding to the files under PATH_TO_LOCAL_DIRECTORY whilst preserving their relative path pseudo-locations.

To download the contents of a Swift pseudo-folder named PSEUDO_PATH_PREFIX, issue the command:

```
swift download [-D PATH_TO_LOCAL_DIRECTORY] -p PSEUDO_PATH_PREFIX/ [-r] CONTAINER
```

This will download all objects whose names are prefixed byPSEUDO_PATH_PREFIX/, optionally stripping of the pseudo-path prefix and placing the files under PATH_TO_LOCAL_DIRECTORY.

Deletion of all objects sharing a common pseudo-path prefix is not supported directly by swift, but may for instance be achieved as follows in a Linux shell:

```
swift delete CONTAINER $(swift list CONTAINER -p PSEUDO_PATH_PREFIX/)
```

Caution: Please be aware that, once again, deletion in Swift cannot be undone. Since the above command automates the identification of the objects to be deleted, you may unintentionally loose valuable data. We therefore recommend to first generate a listing of the objects to be deleted (either manually or by storing their names in a file), and to then operate swift delete ... from the contents of this listing.

Operations on Large Files

In the Swift perspective, large files exceed the size limit of a single object and must therefore be segmented.

The swift client knows how to handle large file objects. When uploading/downloading a large file to/from CONTAINER, it will automatically and in parallel upload/download the individual segments to/from a container named CONTAINER_segments. Likewise, swift will automatically delete a large object's segments when deleting its manifest.

To support swift operations on large files in the UP-Drive object store, we have created segment containers alongside the containers defined by the initial configuration. Note that the sole purpose of employing such segment containers rather than storing segments in the same container as the manifests of large objects is to hide them away from users in the container listings.

Further Reading

Additional documentation of the swift command can be found under the Swift command-line interface reference.

D.3.2 Cyberduck Graphical Client

The Cyberduck GUI client is the recommended way of accessing the UP-Drive object store for less experienced Swift users. It provides the basic functionality such users would expect from a graphical file browser whilst hiding away most of the operational complexity, and it has been verified to work very reliably with the UP-Drive object store setup.

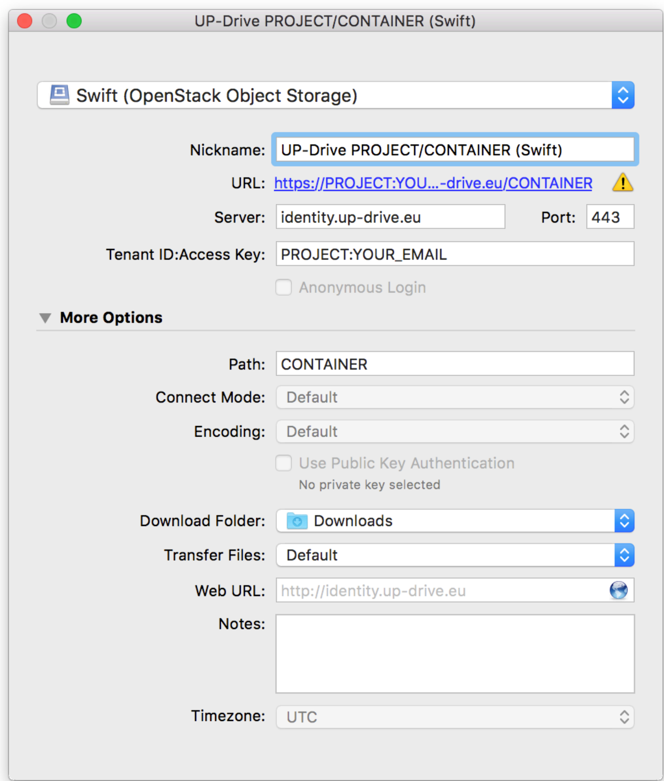
In brevity, Cyberduck neatly supports the following operations:

- Storing of access credentials, including passwords, in bookmarks and the system's keyring
- Operations on container and pseudo-folder contents, mapped to a POSIX-like, virtual directory layout, including creation, (recursive) renaming, and (recursive) deletion
- Parallel transfers, employing a user-defined number of threads
- Large file transfer, using a .file-segments "hidden" pseudo-folder to store the segments
- Continuation of interrupted or paused transfers involving multiple files (see the test and evaluation section for limitations)

As a side note to interested Linux users, we strongly propose to give Cyberduck a shot with Wine and would be happy to see instructions and results included in this section.

Configuration

To connect Cyberduck with the UP-Drive object store, use the configuration scheme depicted below (replacing the capitalized placeholders as explained by the subsequent table) and save it as a bookmark.



The following table highlights the settings values required for accessing the different projects and containers defined by the UP-Drive object store:

Project	Container	Server	Port	Tenant ID:Access Key	Path
up-drive	shared	identity.up-drive.eu	443	up-drive:YOUR_EMAIL	shared
up-drive	PARTNER_ ACRONYM	identity.up-drive.eu	443	up-drive:YOUR_EMAIL	PARTNER_ ACRONYM

Project	Container	Server	Port	Tenant ID:Access Key	Path
FIRST_ LAST	up-drive	identity.up-drive.eu	443	FIRST_LAST:YOUR _EMAIL	up-drive
FIRST_ LAST		identity.up-drive.eu	443	FIRST_LAST:YOUR _EMAIL	

In this table, the first row describes how to connect to the shared container in the shared project. The second row gives the settings for connecting to a partner's container in the shared project, and you will find to have different permissions depending on whether you are affiliated with the partner or not. Using the third row configuration, you will directly connect to the example container in your personal project. And the last row lets you access your personal project at the level of containers, allowing you to create other containers in addition to the exemplary up-drive container, just as you would create a pseudo-folder within a container or nested within another pseudo-folder.

Operations

We found that interacting with the object store through Cyberduck is highly intuitive and follows the concepts of other file browsers such as Finder in macOS or the Explorer in Windows. If you nevertheless and unexpectedly experience any problems with Cyberduck, we will be happy to provide additional operation details in this section.

D.3.3 Cloudfuse Filesystem Client

The Cloudfuse project provides a way of accessing Swift at the level of a remote filesystem. It is a primarily interesting alternative for users of Linux and macOS systems as it requires a Filesystem in Userspace (FUSE)-enabled kernel and the corresponding library.

We are planning to provide further details here once we have thoroughly evaluated and tested the integration of cloudfuse with the UP-Drive object store, but warmly

invite our Linux users to contribute to this section if they decide to venture into this solution.

D.3.4 Curl Command-Line Tool and Library

The widely-used curl command-line tool and its library provide a very low-level, yet portable way of accessing Swift via its REST interface.

We will provide further details here once we have thoroughly evaluated and tested the integration of curl with the UP-Drive object store. You may however find various use-case examples of curl in the documentation of the Swift API.

D.4 Advanced Topics

This section is dedicated to advanced topics related to operating the UP-Drive object store. It will be maintained on the basis of user requests such that each topic remains accessible to other users once it has been documented here.

D.5 Test and Evaluation

To ensure a smooth operation of the UP-Drive object store, we have performed several tests prior to opening up access to the partners. This is a short summary of these tests and our findings:

Test Category	Description	Evaluation	Remarks
File transfer	Up/download of individual files, tested up to 12GB in size	✅	-
Folder transfer	Up/download of nested file and folder structures, tested to a level of 5	✅	-

Test Category	Description	Evaluation	Remarks
	Up/download of a large number of nested (small) files, tested with opencv source repository containing ca 5000 files at 170MB size	✅	-
Load testing	Total bandwidth approaches theoretical limit of 100MBit/s largely independent on the load	✅	-
	Performance unaffected by concurrent up/download of thousands of files, small to large (tested up to 12GB in size)	✅	-
Access	R/W access to up-drive shared, own institution's and personal folders. W access to other institutions' folders	✅	-

Test Category	Description	Evaluation	Remarks
Cyberduck capabilities	Creation of folders and files via right click menu	✅	-
	Folder and file re-naming via right click menu	✅	Caution when renaming folders containing many files. They all will be automatically renamed as well due to the functioning of an object store!
	Folder and file duplication via right click menu	✅	-
	Folder and file moving / duplication via ctrl+c, ctrl+v	❌	Use duplication functionality as described above instead.
	Up/download interruption and resumption of folder structures containing small files	✅	-

Test Category	Description	Evaluation	Remarks
	Up/download interruption and resumption of large files	✗	Should work on segmented files so that up/download resumes with the first not fully transferred segment. Instead, up/download seems to start from scratch.