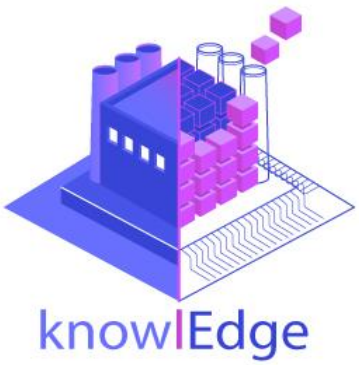
Towards AI powered manufacturing services, processes, and products in an edge-to-cloud-knowlEdge continuum for humans [in-the-loop]



WP6: Advanced Tech Towards Industrial Adoption

EU ID: D6.5 Initial Provisioning and Deployment Management Tool v1.0

Deliverable Lead and Editor: Jannis Warnat, FIT

Contributing Partners: FIT, Nextworks

Date: 2022-06

Dissemination: Public

Status: For EU Approval

Abstract

The purpose of this knowlEdge deliverable, D6.5 “Initial Provisioning and Deployment Management Tool” is to outline the concept for the provisioning and deployment management infrastructure to be developed for the knowledge project.

Grant Agreement: 957331

Document Status

Deliverable Lead	Jannis Warnat, FIT
Internal Reviewer 1	Gabriele Scivoletto, Nextworks
Internal Reviewer 2	Stefan Walter, VTT
Type	Deliverable
Work Package	WP66: Advanced Tech Towards Industrial Adoption
ID	D6.5 Initial Provisioning and Deployment Management Tool
Due Date (Original)	2022-06
Delivery Date	2022-066
Status	V.1.0: Final

History

See Annex A.

Status

This deliverable is subject to final acceptance by the European Commission.

Further Information

www.knowlEdge-project.eu and mailto:info@knowlEdge-project.eu

Disclaimer

Project Partners:

For full details of partners go to www.knowlEdge-project.eu/partners

Executive Summary

This document is deliverable D6.5 “Initial Provisioning and Deployment Management Tool” of the knowlEdge project. It reports the initial work and progress on task T6.4 “Provisioning and Deployment Management” before and until M18.

The document starts by introducing the challenges of software deployments and the benefits of its automation using dedicated software tools. We give a list of requirements for such a tool in the context of the knowlEdge project and a quick overview about the available and evaluated open source deployment frameworks. We explain the choice to employ the Salt and Foreman frameworks and give an overview about their functionalities. Finally, the document lists open questions and challenges that need to be addressed in the progress of the knowlEdge project in the months to come.

Table of Contents

Inhalt

0 Introduction

0.1 knowlEdge Project Overview

The knowlEdge project is funded by the H2020 Framework Programme of the European Commission under Grant Agreement 957331 and conducted from January 2021 until December 2023. The knowlEdge consortium consists of 12 partners from 7 EU countries, and its solution will be tested and evaluated in 3 manufacturing sectors with a total budget of circa 6M€. Further information can be found at www.knowlEdge-project.eu

AI is one of the biggest mega-trends towards the 4th industrial revolution. While these technologies promise business sustainability and product/process quality, it seems that the ever-changing market demands and the lack of skilled humans, in combination with the complexity of technologies, raise an urgent need for new suggestions. Suggestions that will be agile, reusable, distributed, scalable, accountable, secure, standardized and collaborative.

To break the entry barriers for these technologies and unleash their potential, the knowlEdge project will develop a new generation of AI methods, systems and data management infrastructure. This framework will provide means for the secure management of distributed data and the computational infrastructure to execute the needed analytic algorithms and redistribute the knowledge towards a knowledge exchange society. To do so, knowlEdge proposes 6 major innovations in the areas of data management, data analytics and knowledge management: (i) A set of AI services that allow the usage of edge deployments as computational and live data infrastructure, an edge continuous learning execution pipeline; (ii) A digital twin of the shop-floor to test the AI models; (iii) A data management framework deployed from the edge to the cloud ensuring data quality, privacy and confidentiality, building a data safe fog continuum; (iv) Human-AI Collaboration and Domain Knowledge Fusion tools for domain experts to inject their experience into the system to trigger an automatic discovery of knowledge that allows the system to adapt automatically to system changes; (v) A set of standardization mechanisms for the exchange of trained AI-models from one context to another; (vi) A knowledge marketplace platform to distribute and interchange AI trained models.

0.2 Deliverable Purpose and Scope

The purpose of this knowlEdge deliverable, D6.5 “Initial Provisioning and Deployment Management Tool” is to outline the concept for the provisioning and deployment management infrastructure to be developed for the knowledge project in task 6.4 “Provisioning and Deployment Management”

0.3 Target Audience

The D6.5 aims primarily at the partners of the knowlEdge consortium

0.4 Deliverable Context

This manual is based on the project procedures as defined within the knowlEdge Description of Action and Consortium Agreement and where necessary extends them in the operational aspects. However, it is subservient to those documents.

It is one of the cornerstones for achieving the project results, identified as follows:

- **Deliverable:** D6.5 Initial Provisioning and Deployment Management Tool

0.5 Document Status

This document is listed in the Description of Action as public.

0.6 Document Dependencies

This document has no preceding documents. There will be a formal iteration of this document as D6.6 “Final Provisioning and Deployment Management Tool”.

0.7 Glossary and Abbreviations

A definition of common terms related to knowlEdge, as well as a list of abbreviations, is available at www.knowlEdge-project.eu/glossary

0.8 External Annexes and Supporting Documents

External Documents:

- Annexes:
  -
- Supporting Documents:
  -

0.9 Reading Notes

- None

0.10 Document Updates

None

1 Software deployment and Infrastructure as Code

1.1 Introduction

Software deployment is the process of making software available for use in a target environment. This can comprise a selection of the following actions:

- Spin up VMs or containers
- Upgrade / update operating system
- Install and update packages / libraries / dependencies
- Copy files to the target
- Clone code from a code repository (typically Git)
- Build binaries from source code
- Install pre-compiled binaries
- Create configuration files
- Set environment variables
- Provide necessary passwords / keys / secrets
- Run scripts or executables
- Pull Docker images
- Start services
- Check if the software has been started properly

These tasks have historically often been performed in a manual fashion. In the age of many devices and environments and the need to update software often to mitigate security issues and extend functionality there is a clear need to automate this process as much as possible in accordance with the idea and practise of continuous integration / continuous deployment (CI / CD). This enables an agile development and release cycle and allows fixing security issues and functional bugs quickly.

1.2 Infrastructure as code (IaC)

To fully automate the software deployment process, it is necessary to code all deployment steps as a machine readable text file. The steps coded in the file can then be performed automatically. This also allows tracking changes of this file in a code repository. There are multiple open source frameworks available providing IaC functionality:

1.2.1 Puppet

Puppet is an open-core framework written in Ruby [1]. It follows an agent-server architecture where agents and servers communicate via HTTPS using TLS certificates. The desired state of a multitude of target environments is defined in Puppet´s own Ruby-like programming language and stored as files on the server. The agents send some *facts* about themselves

[1] https://puppet.com/docs/puppet/7/puppet_overview.html

to the master and receive the coded desired state they are supposed to be in. This desired state is translated by the Puppet agent into necessary actions to reach the desired state. These actions can then be performed by the agent. Repeating this communication ensures that the desired state on the agents is maintained.

1.2.2 Chef Infra

Chef Infra is somewhat similar to Puppet as it also uses a Ruby-like programming language for the infrastructure code and relies on HTTPS communication between a server and clients[2]. Chef adds a Chef Workstation component that runs locally on the developer´s machine and simplifies configuring and operating the Chef Server. Although the terminology is different, Chef offers a functionality comparable to puppet.

1.2.3 Ansible

Ansible relies on a simpler agentless architecture that uses SSH connections to perform tasks on the target environments[3]. It requires Python to be installed on the targets. The targeted hosts are defined in simple *inventory* INI text files and their desired states are coded in YAML files called *playbooks*. These files can be stored on and executed via Secure Shell (ssh) from any machine, there is no special server component needed.

1.2.4 Salt

Salt uses a server-client architecture (master-minion in Salt terminology) but relies on a different communication strategy than Puppet and Chef [4]. Salt minions communicate with a Salt master using a publish-subscribe pattern. A minion subscribes to a ZeroMQ message queue provided by the Salt master where it asynchronously receives information about its desired state. The salt-minion then performs actions to reach the desired state if necessary. It can also connect to a *Request Server* on the master to send reports or request minion-specific information (e.g. passwords / secrets). The communication is encrypted. In addition, Salt offers an SSH based approach comparable to Ansible if needed for a specific use case.

The Salt communication model makes it particularly attractive for our use case where agents to be managed are typically not accessible from the Internet (as would be required by Puppet, Chef (HTTPS) and Ansible (SSH)). Only the salt master needs to be reachable on port 4505 and 4506 from the minions. So the only connectivity requirement posed on the minions is an outgoing connection to the Internet which is much more easily available compared to an incoming connection.

Salt is supported by VMware which relies on Salt for one of its commercial products[5].

[2] https://docs.chef.io/chef_overview/
[3] https://www.ansible.com/
[4] https://docs.saltproject.io/en/latest/contents.html
[5] https://www.vmware.com/products/vrealize-automation/saltstack-config.html

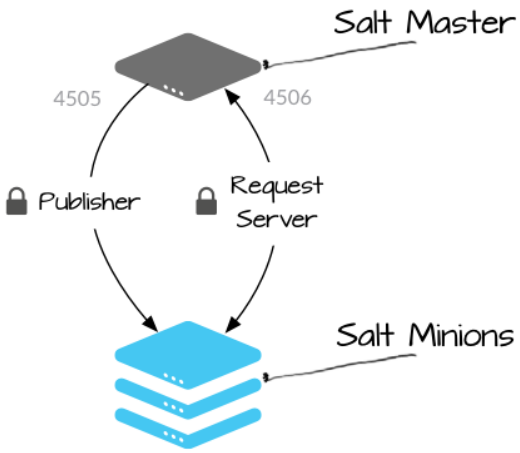


Figure 1: Salt communication[6]

1.2.4.1 Establishing master-minion communication

When the minion connects to the master for the first time, the minion sends its public key to the salt master[6]. This public key needs to be accepted on the Salt master by a dedicated command. The acceptance can be done manually by a system administrator or automated under certain conditions. Before this acceptance is performed, no remote commands can be executed on the minion. After acceptance, the master returns its public key and a rotating AES key that is encrypted using the minion´s public key. All further communication is encrypted using AES keys.

To be sure that the minion is communicating with the correct master, you can provide a fingerprint of the master´s public key in the minion configuration.

1.3 Requirements for the knowlEdge deployment framework

In this context we call the software component that needs to be available on the distributed target environments “deployment agent” while the software component running on a central server is called “deployment master”.

The deployment agent must allow remote control to system administrators via the central deployment manager (running in the cloud) to enable artifact and configuration deployment on distributed target hosts	
Description	A central component (deployment manager) controls target hosts by communicating with a distributed component (deployment agent).
Rationale	This allows system administrators maximal control and oversight over the knowlEdge systems and allows the desired state of the

[6] https://docs.saltproject.io/en/getstarted/system/communication.html

	target environments to be coded as text (“Infrastructure as Code / IaC”)

The deployment manager must register deployment agents running on target hosts after authentication and authorization	
Description	Deployment agents ask the deployment manager to be remotely controlled and provide a key to enable secure communication. The manager can accept or deny the request.
Rationale	Manager and agent need to be sure about the identity of their communication partner.

The deployment manager must provide interfaces to manage the deployment of software artifacts and configurations on target hosts	
Description	System administrators maintain the status of target hosts by providing code and commands to the manager.
Rationale	System administrators should not be bothered with maintaining each target system individually. Maintenance is performed centrally via the deployment manager.

The communication between deployment manager and deployment agents must be authenticated, authorized, and encrypted.	
Description	The communication between manager and agents must be as secure as possible.
Rationale	An attacker gaining control over the communication between manager and agents would have almost full control over the target hosts.

System administrators access to the deployment manager and targeted hosts must be authenticated, authorized, and encrypted.	
Description	The communication between system administrators and the deployment manager must be as secure as possible.
Rationale	A trespasser gaining access to the deployment manager would have powerful control over the target hosts.

The deployment agent must support deployments on target hosts with an up-to-date Linux, Windows, or macOS operating system	
Description	The deployment agent must support a diversity of target hosts with respect to their hard- and software resources.
Rationale	The most common operating systems should be supported. Exotic devices may not be supported.

The deployment manager must provide the status of individual deployments	
Description	A system administrator must be able to see whether his intentions were met on the target hosts.
Rationale	A system administrator should not be burdened with having to connect to each target host individually to check its deployment status.

The deployment manager should integrate with the code repositories and allow automatic deployment as part of a CI/CD pipeline	
Description	The desired state of the target hosts should be coded as text. Changes to the code should trigger an automatic realization of the change on the target hosts.
Rationale	“Infrastructure as Code” in a code repository as part of a CI/CD pipeline allows version control as well as automated testing.

The deployment manager should provide the deployment status and progress in a usable and scalable manner	
Description	The manager should provide a useful (graphical) user interface to allow system administrators an overview over the status of many target hosts.
Rationale	The larger and more diverse the system becomes, the harder it is for a system administrator to keep the overview. The manager should support him or her in keeping it.

The deployment agent should minimize its resources footprint on the target host	
Description	The agent should use as little CPU time, memory, disk space and network bandwidth as possible.
Rationale	The deployment agent should not interfere with target hosts in fulfilling their “actual” function.

2 The Salt framework

Here follows a quick overview over some of the functionalities offered by Salt.

2.1 Salt state files

The desired state of all Salt minions is coded into a file hierarchy of state files (SLS or SaLt State files) in YAML format in a dedicated folder structure on the Salt master. Here is an example of a state file `create-directory.sls` that ensures a certain directory exists on a minion:

```
Create fresh directory for the git clone actions:
  file.directory:
    - name: /home/pi/github/
    - user: pi
    - group: pi
    - mode: 755
    - makedirs: True
```

There could be another SLS file named `clone-from-git.sls` to make sure a certain repository is cloned:

```
Clone some repo:
  git.latest:
    - name: https://github.com/<project>/<repo>.git
    - target: /home/pi/github/<repo>
```

The top of the state file tree is a file called `top.sls` that defines which states are desired for which target minions (in our simple case * meaning all minions). The states are being listed by the names of the files they are defined in:

```
base:
  '*':
    - create-directory
    - clone-from-git
```

We can of course target only a subset of targets with certain states. There is a plethora of possible state functions. Refer to the docs for the details[7]. These states need to be applied to be realized on the Salt minions (see below).

2.2 Salt pillar

The Salt pillar system is defined in a way very similar to the state system. Quote from the docs[8]:

7 https://docs.saltproject.io/en/latest/ref/states/index.html

8 https://docs.saltproject.io/en/getstarted/config/pillar.html

Salt pillar is a system that lets you define secure data that are 'assigned' to one or more minions using targets. Salt pillar data stores values such as ports, file paths, configuration parameters, and passwords.

We can use the Pillar to provide passwords and secrets to some or all minions. E.g., if we wanted to clone a private repository from GitHub we could define a Pillar file like this:

```
git-username: janniswarnat

git-personal-access-token: ghp_<secret-token>
```

And then template the state file like this:

```
Clone some private repo:
  git.latest:
    - name: https://github.com/<project>/<private-repo>.git
    - target: /home/pi/github/<private-repo>
    - https_user: {{ pillar['git-username'] }}
    - https_pass: {{ pillar['git-personal-access-token'] }}
```

Of course, the repository name and target directory can also be templated in a similar fashion.

2.3 Salt Grains

Salt´s Grains interface[9] allows to retrieve information about the minions such as OS type and version, available hard- and software resources and network configuration. Here is an example how to get the cpu model from accepted minions:

```
root@salt-c97877d84-mxqpr:/# salt '*' grains.get cpu_model

docker-salt-minion:

    11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz

raspberrypi:

    ARMv7 Processor rev 3 (v7l)
```

It is possible to use grains information to define states as well as to add customized grains.

2.4 Salt CLI

Commands to the Salt master can be sent via a command line interface[10]. We need to define which hosts we want to target and which command to run. To bring all targeted minions in the desired states defined in the state file structure we run:

```
salt '*' state.highstate
```

To ensure a specific state we run e.g.

```
salt '*' state.apply clone-from-git
```

[9] https://docs.saltproject.io/en/latest/topics/grains/index.html
[10] https://docs.saltproject.io/en/latest/ref/cli/salt.html

It is not only possible to run state functions but also other functions that are not explicitly defined in the state file structure, e.g.:

```
salt '*' test.ping
```

This command will list for each registered minion whether it is responding to the master.

It is also possible to run arbitrary commands on Salt minion, e.g. to list a directory:

```
salt '*' cmd.run 'ls /'
```

2.5 Salt API

All Salt commands can be run on the Salt master using the Salt CLI. But more convenient is using the Salt REST API which offers the same functionality as the CLI[11]. The API is based on CherryPy[12] and by default exposed on port 8000.

There are multiple authentication methods available to receive a token at the `/login` endpoint that can be passed in an `X-Auth-Token` header to subsequent API calls. Here is an example to run the command `test.ping` on all target minions:

```
curl --request POST 'http://salt-master:8000/' \
--header 'Accept: application/x-yaml' \
--header 'X-Auth-
Token: cd8664daa8441093aa47a162ecf491c83a198427' \
--header 'Content-Type: application/json' \
--data-raw '{
    "client": "local",
    "tgt": "*",
    "fun": "test.ping"
}'
```

2.6 Wheel module

An important part of the Salt API is the wheel module[13]. It allows system administrators to add and adapt state and pillar files on the Salt master via the REST API. Of course, the admin has to provide proper credentials and needs to be allowed to use the wheel module. Here is an example HTTP request to change a state file `create-directory.sls`:

```
curl --location --request POST 'http://salt-master:8000' \
--header 'Accept: application/x-yaml' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'X-Auth-Token: 1618975b197a00638603d9d34a8aa5f79b26bd53'
\
```

[11] https://docs.saltproject.io/en/latest/ref/netapi/all/salt.netapi.rest_cherrypy.html
[12] https://docs.cherrypy.dev/en/latest/
[13] https://docs.saltproject.io/en/latest/ref/wheel/all/index.html

```
--data-urlencode 'client=wheel' \
--data-urlencode 'path=create-directory.sls' \
--data-urlencode 'fun=file_roots.write' \
--data-urlencode 'data= Create fresh directory for the git clone
actions:
  file.directory:
    - name: /home/pi/github/
    - user: pi
    - group: pi
    - mode: 755
    - makedirs: True'
```

2.7 Managing arbitrary files on target hosts

A very common deployment use case is to deploy files to a target host. Salt comes with a built-in file server to do this[14]. The files can be stored in a dedicated folder on the Salt master and then assigned to Salt minions via a state file:

```
Deploy nginx configuration file:
  file.managed:
    - name: /etc/nginx/nginx.conf
    - source: salt://nginx/nginx.conf
```

Each time this state is applied, Salt makes sure that the file on the minions matches the one on the master.

2.8 Deployments using Docker

Today many deployment strategies involve artifacts packaged as Docker images. This is also supported by Salt state files.

The availability of a certain Docker image that can be pulled from Docker Hub can be ensured by defining a state file like this[15]:

```
Pull hello-world from Docker Hub:
  docker_image.present:
    - name: hello-world
    - tag: latest
```

It is also possible to configure alternative (private) Docker registries as well as building the image on the target hosts.

To run the container, a state file like this needs to be defined[16]:

[14] https://docs.saltproject.io/en/getstarted/config/files.html
[15] https://docs.saltproject.io/en/latest/ref/states/all/salt.states.docker_image.html
[16] https://docs.saltproject.io/en/latest/ref/states/all/salt.states.docker_container.html

```
Run hello-world:
  docker_container.run:
    - image: hello-world:latest
    - replace: True
    - force: True
```

It is usually more convenient for managing docker deployments to define docker-compose files which is also supported by a dedicated Salt module[17].

2.9 Salt SSH

Salt also offers an agentless usage similar to Ansible[18]. Target hosts need to be defined in a Salt "roster" file at `/etc/salt/roster`, e.g.:

```
Target-host:
  host: target-host
  user: root
```

The Salt master needs an RSA key pair deployed to `/etc/salt/pki/master/ssh` and the target needs to add the master´s public key to its `authorized_keys` file. To enable full functionality, Python needs to be installed on the target. This can be triggered from the master, e.g.:

```
salt-ssh sshd -i -v -l debug -r 'yum -y install epel-release ; yum -y install python3'
```

After Python is installed, the usage of `salt-ssh` is identical to the usage of the Salt CLI, e.g.:

```
salt-ssh '*' test.ping
```

2.10 External Authentication System

The ability to provide salt state and pillar files to the Salt master gives full control over the targeted minions so access needs to be properly authenticated and authorized. There are multiple options available[19]. An obvious choice would be to set up an Azure Active Directory and connect it to Salt via its LDAP external authentication functionality.

Access permissions to targets and state functions can be defined in the configuration file of the Salt master, for example:

```
external_auth:
  ldap:
    test_ldap_user:
        - '*':
```

[17] https://docs.saltproject.io/en/latest/ref/modules/all/salt.modules.dockercompose.html
[18] https://docs.saltproject.io/en/latest/topics/ssh/index.html
[19] https://docs.saltproject.io/en/latest/topics/eauth/index.html

```
    - test.ping
```

This will allow LDAP user `test_ldap_user` to send only the `test.ping` command to all minions.

2.11 Salt installation

2.11.1 Salt master

The Salt master can be deployed as a Docker container in the Azure cloud. An official image is maintained by the Salt community on Docker Hub[20]. The image supports configuration via environment variables, including configuration for the Salt REST API. We will need an extension of this Docker image to use the open source framework Foreman as a GUI for Salt (see below).

The following ports will need to be exposed:

- 4505, 4506: TCP ports for the minion-master communication
- 8000: HTTP port for the Salt REST API
- 9000: HTTP port for the Foreman proxy

In front of the Salt master, a nginx web server instance needs to be set up to perform TLS offloading and route the traffic to the respective port.

Some basic configuration for the Salt master and Salt API can be done via environment variables.

The following configuration artifacts will need to be mounted into the Salt master container:

- RSA key pair for the master
- Some configuration files for the Foreman proxy

Via the Salt API and its *wheel* module[21] we can then add Salt state files and the files for the definition of the Salt *pillar*.

2.11.2 Salt minion

The Salt minion needs to be installed on each device / machine that is supposed to be managed remotely.

The easiest way to install the Salt minion on any well-established operating system is the Salt bootstrap script available on GitHub[22]. The simplest approach would be

```
curl -L https://bootstrap.saltproject.io | sudo sh
```

This installs the correct binaries for all common Unix / Linux derivates as well as Windows using PowerShell. Only minimal configuration is needed in config file `/etc/salt/minion` and that is the domain name of the salt master.

[20] https://hub.docker.com/r/saltstack/salt
[21] https://docs.saltproject.io/en/latest/ref/wheel/all/index.html
[22] https://github.com/saltstack/salt-bootstrap

3 Salt for knowlEdge

The Salt framework can be employed to deploy software on the edge, the fog or in the cloud. Since the cloud will be running on Microsoft Azure, it may be preferable to use native Azure tools and functionality for deployments into the cloud.

In the context of knowlEdge it will be particularly important to allow system administrators to deploy artifacts from the knowlEdge repository (T5.2) and knowlEdge marketplace (T5.3) to target environments. These artifacts will probably be files pulled via a REST API request. This could be done with a Salt state file like this (assuming basic http authentication):

```
Download AI model specification file:
  cmd.run:
    - name: curl -L https://repo.knowledge.eu/<model-id> -u login:
password -o <path-on-target-host>
```

Here is a high level diagram of the architecture of employing Salt in knowlEdge:

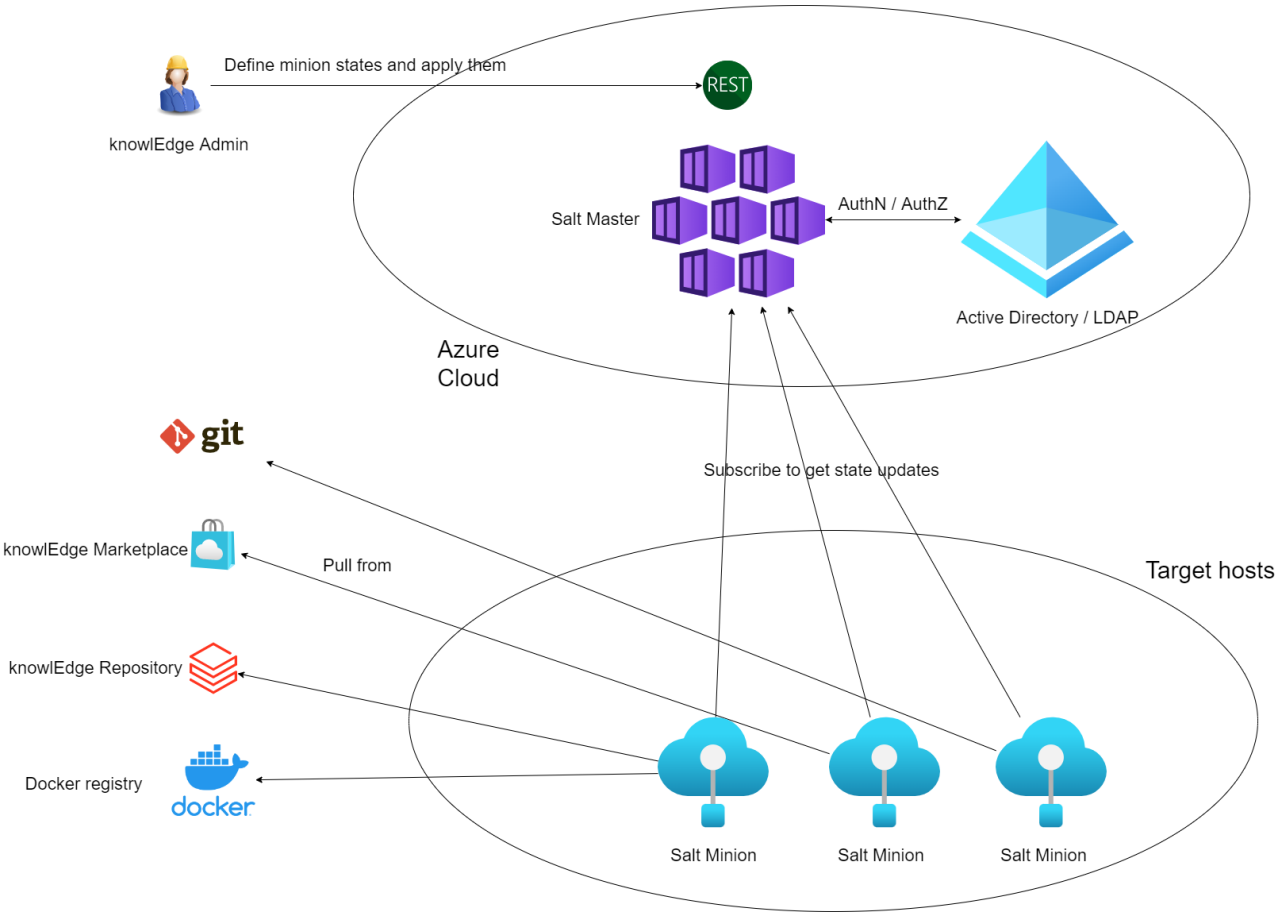


Figure 2: Salt in knowlEdge

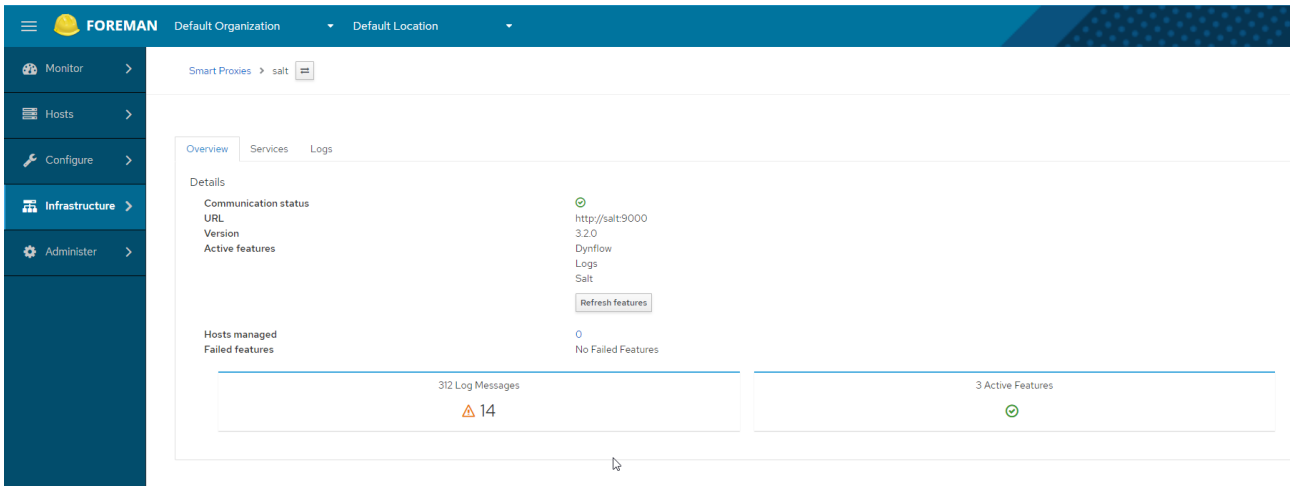
4 The Foreman

Unfortunately, there is no “official” open source graphical user interface (GUI) available for the Salt framework. VMware offers a commercial product called “vRealize Automation SaltStack Config”[23] that includes a GUI. There are some community GUI projects dedicated to Salt but none of these seems to be well maintained or fit to our purpose.

But there is another option that is not dedicated specifically to Salt but supports other IaC frameworks as well. The Foreman[24] is an open source framework for application deployment and server management. With its Salt plugin[25] and Smart Proxy architecture[26] Foreman can act as a GUI for Salt. Foreman is developed using the Ruby on Rails web application framework and supported by an active community. It is supported by Red Hat among other companies[27].

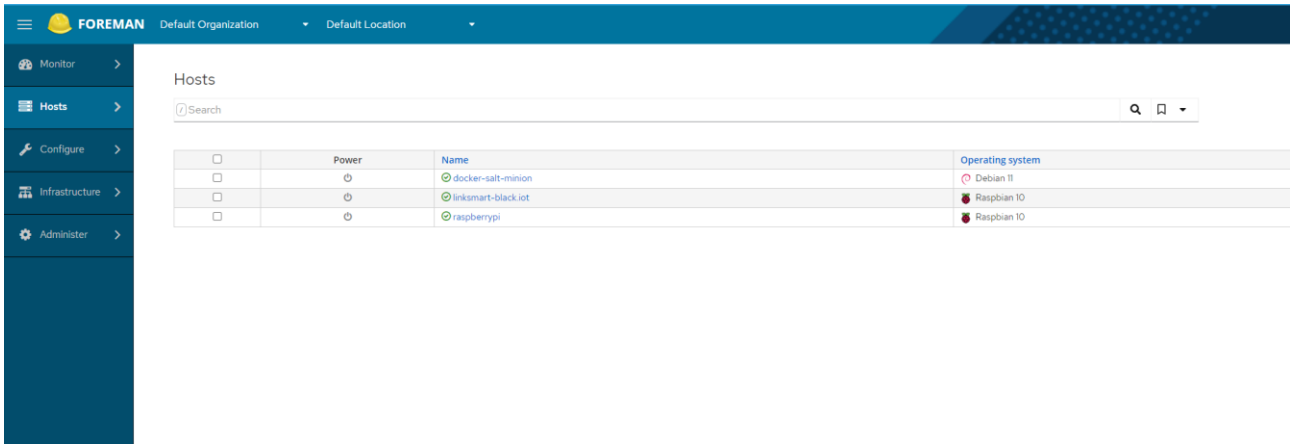
Foreman´s smart proxy has to be installed on the same “machine” as the Salt master so we have to include its installation in the Dockerfile for the Salt master. The proxy exposes a REST API on port 9000 to allow communication with Foreman.

Foreman can also be deployed in a composition of Docker containers[28]. When Foreman is up and running, the Salt smart proxy needs to be registered:



The managed minions then appear in the list of Hosts in Foreman:

[23] https://www.vmware.com/products/vrealize-automation/saltstack-config.html
[24] https://theforeman.org/
[25] https://theforeman.org/plugins/foreman_salt/
[26] https://www.theforeman.org/manuals/3.2/#4.3SmartProxies
[27] https://www.theforeman.org/sponsors.html
[28] https://github.com/theforeman/foreman/blob/develop/developer_docs/containers.asciidoc



Foreman allows minion key management, import of Salt grains as “facts” as well as scheduling remote jobs that trigger actions on the minions via the Salt master. These can be one time jobs or repeated in a regular interval to ensure a certain configuration state of the minions. It is also possible to monitor the progress of running jobs.

The Foreman can provide basic but useful GUI functionalities for Salt but needs further evaluation. If specific requirements for the knowlEdge project appear in the coming months, these could be implemented by Fraunhofer FIT and then returned to the Salt and Foreman open source communities as a contribution.

5 Open challenges

5.1 Internet connectivity of edge devices

Some or all edge devices may have no Internet connectivity available to communicate with the Salt master deployed in the cloud environment. One option could be to deploy a Salt master in the fog and establish SSL tunneled connectivity to edge devices.

5.2 Organizational or security issues

Some edge or fog devices may not be dedicated exclusively to the knowlEdge project but may be part of the daily operations at the pilot site. It is unlikely that pilot partners agree on these devices being remotely controlled by administrators of the knowlEdge project. In this case the configuration and maintenance of the device will stay in the hand of the pilot partner IT and not be performed using the proposed knowlEdge Provisioning and Deployment Management Tool.

6 Conclusion

As a starting point for the work in task 6.4 “Provisioning and Deployment Management” of the knowlEdge project we defined the requirements for the tooling that would support the task. After doing research and experiments with the available deployment and Infrastructure as Code open source frameworks, we identified Salt as a promising candidate to fulfill the requirements. Initial development effort was spent to set up and configure the Salt components in Docker images as well as exploring the offered functionality. Additional effort was spent to set up, configure and evaluate Foreman as a powerful GUI component for Salt.

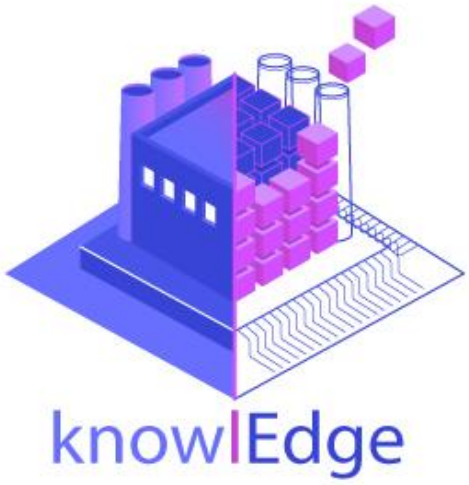
As soon as the knowlEdge cloud infrastructure is available, we can deploy the Salt master in the cloud and configure the first minions in the edge-fog-cloud-continuum to be centrally maintained via the Salt master.

Annex A: History

Document History	
Versions	• 1.0
Contributions	• Jannis Warnat • Gabriele Scivoletto (NXW)

Annex B: References

None



www. knowlEdge-project.eu