



PIACERE

Deliverable D5.3

IaC Execution Manager Prototype – v3

Editor(s):	Josu Díaz de Arcaya
Responsible Partner:	Tecnalia Research & Innovation
Status-Version:	Final-V1.0
Date:	31.05.2023
Distribution level (CO, PU):	PU

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	IaC Execution Manager Prototype
Due Date of Delivery to the EC	31.05.2023

Workpackage responsible for the Deliverable:	WP5 – Package, release and configure IaC
Editor(s):	Josu Díaz de Arcaya (Tecnalia Research & Innovation)
Contributor(s):	Josu Díaz de Arcaya (Tecnalia Research & Innovation)
Reviewer(s):	Giuseppe Celozzi (Ericsson)
Approved by:	All Partners
Recommended/mandatory readers:	WP3, WP4, WP5, WP6, WP7

Abstract:	The main outcomes of the PIACERE IaC Execution Manager Prototype are presented in this deliverable. This deliverable corresponds to Key Result 10.
Keyword List:	Infrastructure as Code, IaC Execution Manager
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	17.05.2023	First draft version	Tecnia
v0.2	24.05.2023	Review	Ericsson
v1.0	29.05.2023	Ready for submission	Tecnia

DRAFT

Table of contents

Terms and abbreviations.....	7
Executive Summary.....	8
1 Introduction	9
1.1 About this deliverable	9
1.2 Document structure	9
2 KR 10- IEM overview	10
2.1 Changes in v3	10
2.1.1 Run time features.....	10
2.1.2 ORM for database management.....	12
2.1.3 Containerization support	14
2.1.4 Support for base64 encoded bundle.....	15
2.1.5 Increased feedback and logging capabilities.....	17
2.1.6 Testing and coverage	18
2.1.7 Support for further Orchestration engines	22
2.2 Functional description and requirements coverage	23
2.3 Main innovations.....	25
3 Overview of preliminary experiments.....	26
3.1 Experiments on AWS.....	26
3.2 Experiments with Docker	26
3.3 Experiments on OpenStack.....	27
4 Lessons learnt and outlook to the future.....	29
5 Conclusions	30
6 References.....	31
APPENDIX: Implementation, delivery, and usage	32
1 Implementation.....	32
1.1 Fitting into overall PIACERE Architecture.....	32
1.2 Technical description	33
1.2.1 Prototype architecture.....	33
1.2.2 Components description	35
1.2.3 Technical specifications.....	35
2 Delivery and usage	37
2.1 Package information	37
2.2 Installation instructions.....	37
2.3 User Manual	41
2.4 Licensing information.....	45
2.5 Download	45

List of tables

TABLE 1: USER REQUIREMENTS ADDRESSED BY THE IEM	24
---	----

List of figures

FIGURE 1 - DEPLOY A PROJECT DEFINED WITH CUCUMBER.....	11
FIGURE 2 - QUERY THE STATUS OF AN EXISTING PROJECT DEFINED IN CUCUMBER.....	11
FIGURE 3 - UNDEPLOY A PROJECT DEFINED IN CUCUMBER.....	11
FIGURE 4 - QUERY THE STATUS OF AN UNDEPLOYED PROJECT DEFINED IN CUCUMBER.....	11
FIGURE 5 - REDEPLOY A PROJECT DEFINED IN CUCUMBER.....	12
FIGURE 6 - TABLE DEFINITION UTILIZING SQLALCHEMY	13
FIGURE 7 - PERSISTENCE ORM-POWERED CLASS UTILIZED BY THE IEM	14
FIGURE 8 - PIACERE PROJECT INCLUDING A DOCKER STAGE.....	15
FIGURE 9 - DOCKER COMPOSE FILE UTILIZED IN THE DEPLOYMENT.	15
FIGURE 10 - DEPLOYMENT ENDPOINT ADAPTED TO A BASE64 ENCODING.	16
FIGURE 11 - SNIPPET FOR DECODING THE BASE64 AND EXTRACTING THE DEPLOYMENT PROJECT.	17
FIGURE 12 - PRINT THE STANDARD OUTPUT OF A COMMAND AND RETURN IT TO THE USER.	17
FIGURE 13 – THE VARIOUS LOGGERS UTILIZED BY THE IEM.	18
FIGURE 14 - THE DEFAULT FORMATTER TO BE UTILIZED BY THE IEM LOGGERS.	18
FIGURE 15 - THE WHOLE LIST OF UNIT TESTS IMPLEMENTED IN THE IEM.....	19
FIGURE 16 - AN EXCERPT OF A UNIT TEST UTILIZING FASTAPI TEST CLIENT.	19
FIGURE 17 - AN EXCERPT OF A UNIT TEST MOCKING THE INTERACTION WITH THE CLOUD PROVIDER.	20
FIGURE 18 - THE VARIOUS INTEGRATION TESTS DEVELOPED TO VALIDATE THE FUNCTIONALITY OF THE IEM.....	21
FIGURE 19 - AN EXCERPT SHOWING AN INTEGRATION TEST WITH AWS.	21
FIGURE 20 - THE COVERAGE SUMMARY OF THE IEM.....	22
FIGURE 21 - AWS CREDENTIAL HANDLING.....	22
FIGURE 22 - AZURE CREDENTIAL HANDLING.	23
FIGURE 23 - OPENSTACK CREDENTIAL HANDLING.	23
FIGURE 24 - VMWARE VSPHERE CREDENTIAL HANDLING.	23
FIGURE 25 - RUNNING THE EXPERIMENT ON AWS.	26
FIGURE 26 - CONFIGURATION FILE FOR AWS.	26
FIGURE 27 - VIRTUAL MACHINE DEPLOYMENT ON AWS.	26
FIGURE 28 - DOCKER COMPOSE FILE FOR A SINGLE WEB SERVER.	27
FIGURE 29 - FRONTPAGE FOR THE NGINX WEB SERVER PROVISIONED.....	27
FIGURE 30 - CONFIGURATION FILE FOR OPENSTACK.	28
FIGURE 31 - VALIDATE THAT THE VIRTUAL MACHINES HAVE BEEN PROVISIONED ON OPENSTACK.....	28
FIGURE 32 - PIACERE RUNTIME WORKFLOW.....	32
FIGURE 34 - IEM PROTOTYPE ARCHITECTURE.	33
FIGURE 35 - IEM INITIATE DEPLOYMENT SEQUENCE DIAGRAM	34
FIGURE 36 - IEM REQUEST THE CURRENT STATUS OF A DEPLOYMENT.....	34
FIGURE 37 - IEM INITIATE UNDEPLOYMENT SEQUENCE DIAGRAM.....	35
FIGURE 38 - ROOT FOLDER FOR THE PIACERE IEM COMPONENT.....	37
FIGURE 39 - CREATE THE VIRTUAL ENVIRONMENT.	38
FIGURE 40 - ACTIVATE THE VIRTUAL ENVIRONMENT.	38
FIGURE 41 - EXCERPT SHOWING THE CONTENT OF THE REQUIREMENTS.TXT FILE FOR DEPENDENCY MANAGEMENT.	39
FIGURE 42 - EXECUTE ALL THE TESTS OF THE IEM WITH THE NOSE TOOL.	40
FIGURE 43 - EXCERPT SHOWING THE DOCKERFILE UTILIZED FOR GENERTING THE CONTAINERIZED IMAGE OF THE IEM.	41
FIGURE 44 - EXCERPT SHOWING THE BUILDING PROCESS FOR THE IEM COMPONENT.	41

FIGURE 45 - OPENAPI SPECIFICATION FOR THE INTERACTION WITH THE IEM COMPONENT. 42
FIGURE 46 - CREDENTIALS TO BE USED BY THE IEM. 43
FIGURE 47 - RUN THE IEM IMAGE. 43
FIGURE 48 - EXECUTE THE IEM WITH THE UVICORN SERVER. 44
FIGURE 49 – EXECUTE THE IEM DIRECTLY. 44
FIGURE 50 - VARIOUS DOCUMENTATION ENDPOINTS AVAILABLE IN THE IEM. 44

DRAFT

Terms and abbreviations

CSP	Cloud Service Provider
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
GA	Grant Agreement to the project
IaC	Infrastructure as Code
IEP	IaC execution platform
IEM	IaC Execution Manager
IOP	IaC Optimization Platform
KPI	Key Performance Indicator
SW	Software
KR	Key Result
REST	Representational State Transfer
API	Application Programming Interface
DevSecOps	Development, Security and Operations
IDE	Integrated Development Environment
ORM	Object Relational Mapper
TDD	Test Driven Development
RDS	Relational Database System

DRAFT

Executive Summary

This manuscript documents the final iteration of the IEM (IaC Execution Manager). We start by providing an overview of the associated Key Result (KR10) and outline the various changes that have taken place during the third year of the project explained in detail, and we move on to correlate each of the changes with the associated requirements. We finalize Section 2 stating the main innovations of the IEM tool. Section 3 provides an overview of the various experiments utilized to validate the appropriateness of the IEM tool within the PIACERE ecosystem. Section 4 provides an in-depth description of the lesson learnt during the project and we finalize the document with the Conclusions in Section 5. The appendix revolves around the implementation and usage of the IEM.

At its current stage, the IEM can fulfil the requirements imposed by the rest of the PIACERE ecosystem. This has been achieved partly due to the successful analysis performed during the first year of the project in terms of analysing the state of the art of the various IaC technologies. At the time of writing this document, three different IaC technologies are supported, and four public and private cloud providers have been tested during the project.

The main innovations of this component are the seamless integration of the various IaC technologies by gluing them together into a continuous execution, the automated solving of common pitfalls, alleviating the burden on the various professionals that take part on the deployment orchestration, extensible architecture, and security. These are explained in further detail in Section 2.

As for the future steps, IaC technologies keep evolving and so does the IEM. In the near future, more efforts need to happen in the field of supporting the self-healing of the whole PIACERE ecosystem.

DRAFT

1 Introduction

This deliverable is the latest iteration of the IaC Execution Manager which previous versions were described in D5.1 [1], and D5.2 [2]. It provides a detailed description of the tool which fulfils an essential role in the overall PIACERE architecture. In this section the content and structure of this deliverable are explained in detail.

1.1 About this deliverable

This document is focused on the IEM prototype, in the iteration corresponding to M30 of the PIACERE project. The design and development of this deliverable has been fully undergone and funded in the scope of the PIACERE project (Horizon 2020 research and innovation Program, under grant agreement no 101000162). The overarching goal of the IEM is to provide a unified approach for the orchestration of the deployment of multilingual IaC technologies for the various scenarios that are held by the PIACERE use cases. In this document, we elaborate on the functionalities developed during the latest iteration of the IEM, which has taken place from month 24 to the current month 30. During this period, the functionalities already developed in previous iterations have been established and further functionalities have been enabled to support the various needs of the PIACERE ecosystem. This document serves as complementary material to the previous deliverables of the IEM component.

1.2 Document structure

The rest of this document is structured as follows. Section 2 provides an in-depth description of the key result (KR) associated with the IEM, which is KR10. In addition, the changes undergone for this third version to be released are explained in detail. Finally, the main innovations proposed by the IEM are showcased. Section 3 covers the development and preliminary experiments that have been performed during this semester. Next, Section 4 explores the lessons learnt and future work for the IEM prototype. Finally, the conclusions of this effort are explained in Section 5.

In addition, detailed information about the implementation, delivery and usage of the IEM prototype are included at the end of this document as an appendix. This appendix represents a complete rewriting of previous iterations of this deliverable, some parts might still be very similar but have been included as part of this document for completeness.

2 KR 10- IEM overview

A general overview of the PIACERE KR10 has been previously introduced in D2.1 [3]. The following definition is extracted from that very same document as it served as the kick off for the development of the IEM component. In this deliverable was stated the goal of the PIACERE's IEM component, which is to establish a unified approach for utilizing various Infrastructure as Code (IaC) technologies from a common interface. The IaC Execution Manager (IEM) plays a crucial role in achieving this overarching goal by overseeing the utilization of IaC code generated in earlier stages of the PIACERE ecosystem. It iteratively executes different IaC technologies to attain the desired project architecture. Moreover, the IEM leverages diverse IaC paradigms, including provisioning heterogeneous infrastructural devices across public and private cloud providers, configuring each infrastructural device to support the project, and operationalizing use case applications within the PIACERE framework. Additionally, the IEM offers a consolidated method for querying deployment information, encompassing which encompass the status of past and present execution and their detailed logs. Furthermore, it enables accessing information about the supported IaC technologies and versions. The IEM exposes its services through a REST API documented in the OpenAPI specification format, which requires implementing the specification for components seeking to utilize the IEM. Secure access to IEM methods is ensured through token-based authentication technologies.

2.1 Changes in v3

This latest iteration is the shortest one, as it only comprises the changes from M24 to M30. Nevertheless, since it is relatively at the end of the PIACERE project, there are significant changes that we detail in this section. First, the Run time features exemplify the various scenarios for which the IEM has been designed, they serve both to document and to validate that the IEM works as expected with the rest of the PIACERE ecosystem. Next, the ORM (Object Relational Mapper) database management provides an abstraction from the underlying persistence, making it possible to integrate with other RDS (Relational Database System) without having to modify the implementation. This subsection is related with REQ82. Next, the Containerization Support provides additional functionalities to the engines already supported by the IEM, making it possible to deploy container-based applications in the provisioned infrastructure. This subsection directly relates to REQ81 (all the requirements are explained in detail in Section 2.2). Next, the support for base64 encoded bundles serves as a change of paradigm in the manner that the PIACERE ecosystem interacts among its various components. This subsection relates to REQ83. Next, the increased feedback and logging capabilities provides increased verbosity and debugging functionalities for both developers and end users. This way, it is possible to better understand what happens under the hood of a particular deployment, which directly relates to REQ83. Next the testing and coverage has attracted attention during this latest iteration since this year the component should be stable to be used by the various PIACERE use cases. In this subsection, all the requirements have been addressed in one way or another (REQ12, REQ81, REQ82, REQ83, REQ84, REQ85, REQ87). Finally, in the support for further orchestration engines section we showcase the implementation of the various external orchestrators supported by the IEM now. This directly addresses requirements REQ83, REQ84, REQ85 and REQ87.

2.1.1 Run time features

To support the required functionalities for the PIACERE run time environment various scenarios have been defined utilizing the Gherkin language. In Figure 1, a fresh project utilizing various IaC (Infrastructure as Code) technologies is defined. The user needs to trigger the deployment from the IDE (Integrated Development Environment), which forwards the call to the IEM via the PRC (PIACERE Runtime Controller). Then, the IEM executes the stages defined in the bundle asynchronously and the user is notified of the outcome of the deployment. This scenario is

related to REQ81, REQ83, REQ84 and REQ87, which are explained in the following section in further detail.

```
6 Scenario: Deploy a fresh project which comprises terraform, ansible, and docker
7 Given a project bundle in the relevant IaC technologies (terraform, ansible, docker-compose), the deployment id,
8   When the user triggers the deployment
9   Then the IEM is invoked
10  And executes the stages of the bundle asynchronously
11  And the user is notified that the deployment has been accepted
```

Figure 1 - Deploy a project defined with cucumber.

In Figure 2 querying the status of an existing project running in the IEM is depicted. In this scenario, the user utilizes the IDE to query the status of an existing deployment, the petition is handed over to the IEM which returns the message to the IDE to be visualized by the user. This scenario is related to REQ55 and REQ82, which are explained in the following section in further detail.

```
16 Scenario: Query the status of a running project
17 Given the deployment id of an already existing project
18   When the user queries the status of the project
19   Then the IEM is invoked
20   And the user is notified of the status
```

Figure 2 - Query the status of an existing project defined in cucumber.

The scenario depicted in Figure 3 the undeployment of an existing project in the IEM. The user triggers the undeployment in the IDE. This action is then passed across the architecture to the IEM which kicks off the undeployment asynchronously. Then, the user is notified about the given action being accepted. This scenario is related to REQ81, REQ83, REQ84 and REQ85, which are explained in the following section in further detail.

```
25 Scenario: Undeploy a project
26 Given the deployment id of an already existing project and the required cloud credentials
27   When the user triggers the undeployment
28   Then the IEM is invoked
29   And tears down the entire deployment asynchronously
30   And the user is notified that the undeployment has been accepted
```

Figure 3 - Undeploy a project defined in cucumber.

The user can query the status of an undeployed project utilizing the IDE, this query is then forwarded to the IEM across the PIACERE architecture, and the response is handed over back to the IDE for the user to check. This scenario, which is depicted in Figure 4, is related to REQ55 and REQ82, which are explained in the following section in further detail.

```
35 Scenario: Query the status of an undeployed project
36 Given the deployment id of an undeployed project
37   When the user queries the status of the project
38   Then the IEM is invoked
39   And the user is notified of the status
```

Figure 4 - Query the status of an undeployed project defined in cucumber.

The user modifies an existing project and kicks off a redeployment. This action is handed over to the PRC which validates that the redeployment is allowed and passed all over to the IEM which then executes the action. This scenario, which is depicted in Figure 5, is related to REQ12, REQ81, REQ83, REQ84 and REQ87, which are explained in the following section in further detail.

```
44 Scenario: Redeploy a project
45 Given a project bundle in the relevant IaC technologies (terraform, ansible, docker-compose), the deployment id,
46   When the user triggers the deployment
47   Then the IEM is invoked
48   And executes the stages of the bundle asynchronously
49   And the user is notified that the deployment has been accepted
```

Figure 5 - Redeploy a project defined in cucumber.

2.1.2 ORM for database management

The IEM utilizes SQLite¹ as its persistence database. SQLite is a library implemented in C, and the most utilized database in the world. In addition, it is bundled as part of the default Python installation which is the programming language utilized for the implementation of the IEM. In previous iterations of this component, the SQLite library was used. However, there are some drawbacks in this approach. For instance, moving away from SQLite would mean a complete reimplementing of the persistence module. Due to this, an ORM tool such as SQLAlchemy² can be beneficial and boost the development productivity.

SQLAlchemy is a Python toolkit that enables developers to utilize Object Relation Mapper strategies. It has a wide support and its adoption in the community is exceptionally large. Some the benefits of switching to an ORM are the following:

- The interaction with the database is simplified as there is no need to utilize SQL statements directly, which relates with the maintenance of the code.
- The abstraction of ORM tools promote the cross-platform compatibility of the code, as switching from one database to another would be transparent and would not require a reimplementing.
- It reduces code duplication and developers have a central location for database schemas and relationships and promotes code reusability.

With regards with the use of SQLAlchemy in the IEM. Figure 6 depicts the implementation of the deployments table in the database. It allows the definition of the primary key, and the type of each of the fields that comprise the table. In this example, the following fields are defined:

- Status_time: the datetime when the deployment was triggered, if no time has been provided the current time is
- Deployment_id: a unique identifier for the deployment that has been triggered. It is a mandatory field.
- Status: a concise field which summarizes the current status of the existing deployment.
- Stdout: the standard output provided after kicking off or tearing down the deployment.
- Stderr: the standard error provided after kicking off or tearing down the deployment.

¹ <https://sqlite.org>

² <https://www.sqlalchemy.org>

```
15 class Deployment(Base):
16     __tablename__ = "deployments"
17
18     id = Column(Integer, primary_key=True)
19     status_time = Column(DateTime, default=datetime.now())
20     deployment_id = Column(String, nullable=False)
21     status = Column(String, nullable=False)
22     stdout = Column(String)
23     stderr = Column(String)
24
25     def __repr__(self):
26         return f"Deployment(id={self.id!r})"
```

Figure 6 - Table definition utilizing SQLAlchemy

In Figure 7, the persistence layer has been reimplemented in order to obtain the aforementioned benefits. It is worth noting that various engines can be instantiated, this is important for testing purposes as an in-memory database can be utilized during CI/CD. At the moment, the following methods are available to be used by the IEM:

- `Insert_deployment`: this method serves the purpose of recording in the database a new state for a deployment that has been passed to the IEM through its REST API.
- `Get_deployment`: this method yields back the last deployment identified by its unique id, which represents the actual state of the given deployment.
- `Get_all_deployments`: this method yields back all the deployments that have taken place in the current environment.
- `Valid_api_key`: this method validates that the key that has been passed to the IEM through its REST API is valid. In addition, it has built-in security so no more than ten validations per second can be done.

```

29  class Persistence:
30  def __init__(self, engine_url="sqlite:///db/iem.db"):
31      self._engine = create_engine(url=engine_url, future=True)
32      Base.metadata.create_all(self._engine)
33
34  def insert_deployment(
35      self, deployment_id: str, status: str, stdout: str, stderr: str
36  ):
37      with Session(self._engine) as session:
38
39          spongebob = Deployment(
40              deployment_id=deployment_id,
41              status=status,
42              stdout=stdout,
43              stderr=stderr,
44          )
45
46          session.add(spongebob)
47
48          session.commit()
49
50  def get_deployment(self, deployment_id: str):
51      with Session(self._engine) as session:
52          stmt = (
53              select(Deployment)
54              .where(Deployment.deployment_id == deployment_id)
55              .order_by(desc(Deployment.id))
56          )
57          return session.scalars(stmt).first()
58
59  def get_all_deployments(self):
60      with Session(self._engine) as session:
61          stmt = select(Deployment).order_by(desc(Deployment.id))
62          for d in session.scalars(stmt):
63              yield d
64
65  @RateLimiter(max_calls=10, period=1)
66  def valid_api_key(self, api_key_query: str):
67      if api_key_query == os.getenv("API_KEY"):
68          return True
69      else:
70          return False

```

Figure 7 - Persistence ORM-powered class utilized by the IEM

In summary, switching to an ORM tool for database management makes sense in this context, as it provides additional flexibility to the IEM. For instance, if a centralized database would be beneficial for a particular use case, the current approach would provide an easy adjustment to such scenario. In addition, development productivity is also promoted as SQL statements do not need to be specifically tailored for each of the use cases.

2.1.3 Containerization support

One of the requirements of this third development iteration of the PIACERE project was to be able to deploy containers in the provisioned and configured infrastructure. For this reason, we have envisioned a manner to support the docker engine within the PIACERE ecosystem, by supporting it in each of the relevant components, including the IEM. Figure 8 clearly depicts that a project including the docker engine is no different than a regular project. The stages are again deployed one by one.

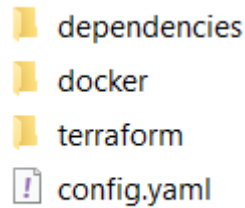


Figure 8 - PIACERE project including a docker stage.

However, deep diving into the project raises a docker compose file as it can be seen in Figure 9. The IEM is able to install all the required dependencies for docker based projects in the freshly provisioned infrastructural devices and kicks off the relevant commands for bringing up the applications. It does so by overloading the already existing ansible engine and triggering the relevant docker commands remotely. The requirement for this to be triggered are the same what the ansible engine would expect, connectivity with the remote infrastructural devices and access privileges over it.

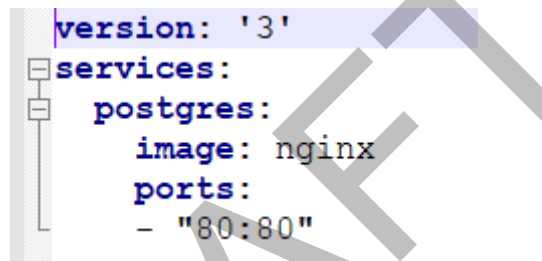


Figure 9 - docker compose file utilized in the deployment.

The result is a seamless integration of container-based applications within the PIACERE ecosystem. This is particularly relevant for the various use cases that participate in the project.

2.1.4 Support for base64 encoded bundle

At the beginning of the project, the handles of the deployment project were passed around the relevant component. However, during this last iteration the approach was simplified and the entire ecosystem. It was agreed among the consortium that the entire folder should be bundled in zip format and encoded utilizing base64. Then each component decodes the relevant information and passed it around the next component in line. Figure 10 depicts the full body of the deployment requests, in which a bundle subsection is displayed at the end of it.

```
{
  "deployment_id": "string",
  "credentials": {
    "aws": {
      "access_key_id": "string",
      "secret_access_key": "string"
    },
    "azure": {
      "arm_client_id": "string",
      "arm_client_secret": "string",
      "arm_subscription_id": "string",
      "arm_tenant_id": "string"
    },
    "openstack": {
      "user_name": "string",
      "password": "string",
      "auth_url": "string",
      "project_name": "string",
      "region_name": "string",
      "domain_name": "string",
      "project_domain_name": "string",
      "user_domain_name": "string"
    },
    "vmware": {
      "user_name": "string",
      "password": "string",
      "server": "string",
      "allow_unverified_ssl": "string"
    },
    "docker": {
      "server": "string",
      "user_name": "string",
      "password": "string"
    }
  },
  "bundle": {
    "base64": "string"
  }
}
```

Figure 10 - Deployment endpoint adapted to a base64 encoding.

In the case of the IEM, the following Figure 11 showcases how the bundle is decoded, and the resulting zip file extracted to the relevant folder.


```
LOGGER.info(f"Decompressing base64 bundle.")
bundle_decode = base64.b64decode(bundle)
ZipFile(BytesIO(bundle_decode)).extractall(repo_path)
```

Figure 11 - Snippet for decoding the base64 and extracting the deployment project.

2.1.5 Increased feedback and logging capabilities

One of the requirements for the IEM is to provide appropriate feedback for the relevant actors, in this case users and developers alike. On one hand, end users utilizing the IEM as part of the entire PIACERE ecosystem are expected to have feedback on the existing deployments via the IDE. On the other hand, developers utilizing the IEM expect constant feedback from the IEM and the status of current deployments. This poses a threat since printing out the logs during the execution and returning them as part of the REST API calls is not straightforward in Python. In the following figure, it can be seen the strategy utilized by the IEM to handle this issue. First, the standard output of the given command is logged to the console. Then, the very same variables are returned to the final user to provide more information about the status of a deployment. The snippet in charge of this functionality is depicted in Figure 12.

```
output = subprocess.run(
    ["terraform", "apply", "-auto-approve"],
    cwd=self._repo_path,
    env=self._env,
    capture_output=True,
)
LOGGER.info(output.stdout.decode("utf-8"))
output.check_returncode()
return "CREATED", output.stdout, output.stderr
```

Figure 12 - Print the standard output of a command and return it to the user.

The above approach guarantees that the users of the whole PIACERE architecture are able to get the appropriate feedback. Similarly, the logging capabilities, which are depicted in Figure 13, for the IEM are defined in the following configuration file. The three different loggers utilized in this environment are the following:

- Root: the root logger is the default logger created when a python module is created. If no explicit logger is created this one is utilized.
- Src: the src logger is the customized logger defined for the IEM.
- Unicorn: this logger controls the way the unicorn environment manages the various logs.

```
[logger_root]
level=INFO
handlers=stream_handler

[logger_src]
level=INFO
handlers=stream_handler
qualname=src
propagate=0

[logger_uvicorn]
level=INFO
handlers=stream_handler
qualname=uvicorn
propagate=0
```

Figure 13 – The various loggers utilized by the IEM.

Figure 14 showcases the formatter that is utilized for all the loggers explained above is displayed. It provides the following fields:

- Asctime: the time in which the action takes place.
- Name: the module in which the IEM logs the information.
- Levelname: the level of the action.
- Message: the user friendly message to understand the incidence.

```
[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Figure 14 - the default formatter to be utilized by the IEM loggers.

This mixed approach fulfils the requirements of users and developers to have the relevant information to debug and understand what happens under the hood of the IEM.

2.1.6 Testing and coverage

To guarantee that the integration with the rest of the components of the PIACERE ecosystem is seamless, we have followed a TDD [4] (Test Driven Development) approach for the development of the IEM. TDD is the practice in which a programmer writes a failing test prior to the implementation of the code. Due to this, the entire functionality of the IEM correlates with a test so that everything works smoothly. In this project, we have implemented two different test types: unit and integration tests. The former is a way of testing the smallest piece of code that can be logically isolated in a system such as function [5], whereas the latter refers to the practice in which various modules or components of a software application are tested as a combined [6]. In Figure 15, the full lists of unit tests utilized for the development and assurance of the quality of the IEM are depicted.

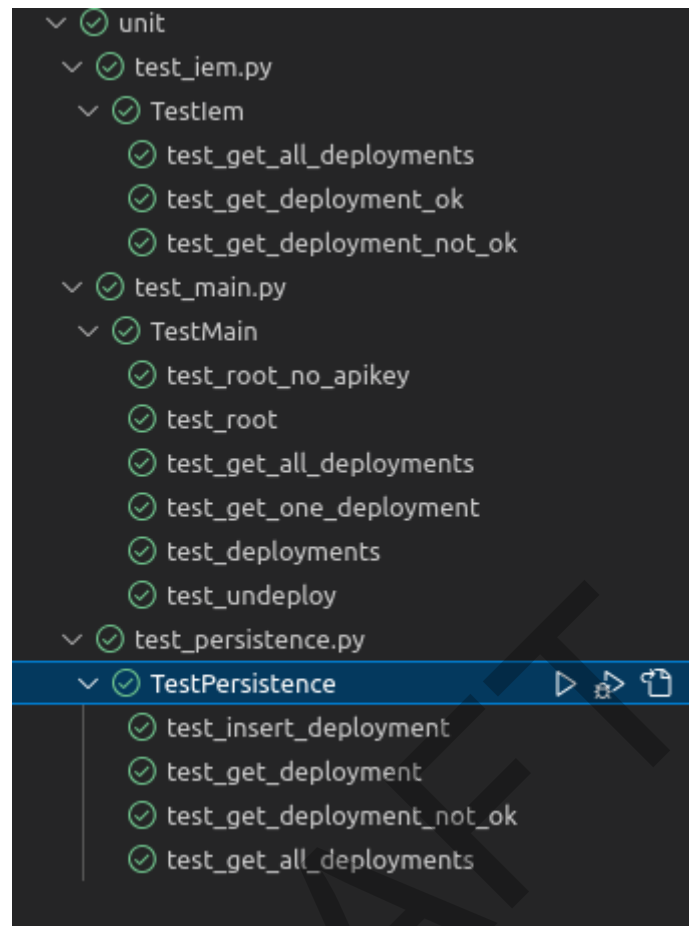


Figure 15 - The whole list of unit tests implemented in the IEM.

There are three main modules that encapsulate the various unit test cases available:

- `Test_iam.py`: this class is devoted to test the basic functionalities offered by the IEM core, without dwelling on the particularities of the REST API.
- `Test_main.py`: this class tests the system from a higher level than the one above. On top of testing the entire functionality, this class also interacts with the REST API with the appropriate credentials.
- `Test_persistence.py`: this class is exclusively in charge of testing the functionality of the persistence layer within the IEM. It guarantees that the interaction with the database is working at every time.

For the implementation of the testing capabilities, we have resorted to the well-known `unittests`³ python framework. An example of a simple test case that validates the use of the API KEY in every single call to the IEM is depicted in Figure 16.

```
def test_root_no_apikey(self):
    response = self.client.get("/")
    assert response.status_code == 403
```

Figure 16 - An excerpt of a unit test utilizing FastAPI test client.

³ <https://docs.python.org/3/library/unittest.html>

In this scenario, the FastAPI testclient⁴ is being used to trigger a call to the REST API without the appropriate credentials, which yields a “403 Forbidden” scenario. However, properly testing the IEM becomes hard since its overarching goal is to orchestrate deployment with external providers, hence bare unit testing is not ideal. However, we have devoted to the mock and patch functionalities of this very same framework. This way, we can validate the entire source code without having to handle the particularities of external cloud providers.

```
@patch("subprocess.run")
def test_deployments(self, mock_run):
    mock_run.return_value = Mock(returncode=0, stdout=b"{}", stderr=b"{}")
    with open("tests/resources/dummy.zip", "rb") as binary_file:
        bundle = base64.b64encode(binary_file.read())
    response = self.client.post(
        f"/deployments/",
        headers={"x-api-key": self._api_key},
        json={
            "deployment_id": str(uuid.uuid4()),
            "credentials": {
                "openstack": {
                    "user_name": "string",
                    "password": "string",
                    "auth_url": "string",
                    "project_name": "string",
                    "region_name": "string",
                    "domain_name": "string",
                    "project_domain_name": "string",
                    "user_domain_name": "string",
                }
            },
            "bundle": {"base64": bundle.decode("utf-8")},
        },
    )
    assert response.status_code == 201
```

Figure 17 - An excerpt of a unit test mocking the interaction with the cloud provider.

Figure 17 patches the functionality provided by subprocess, which is the one that interacts with the external cloud providers. Next, we mock the return of that call so that the whole flow of the IEM for a given deployment is executed. Finally, we validate the appropriate response.

Next, the integration tests are of paramount importance since they essentially validate with real tests that the IEM will work under realistic conditions. Figure 18 shows the integration tests that have been utilized for the IEM are depicted.

⁴ <https://fastapi.tiangolo.com/tutorial/testing/>

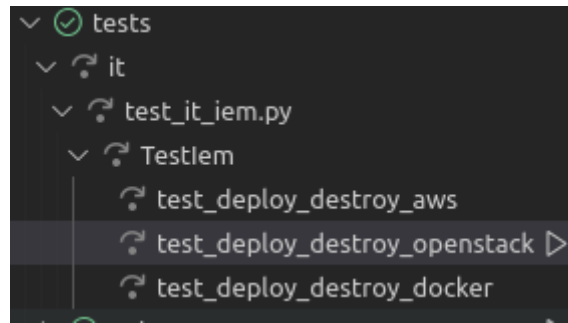


Figure 18 - The various integration tests developed to validate the functionality of the IEM.

These tests deploy realistic projects and test them utilizing various private and public cloud providers. In this case, we have implemented that the IEM works using AWS, OpenStack, and Docker providers. This does not mean that it does not work with other providers, rather it serves as a baseline to exemplify that everything we have been developed with the aforementioned tests is indeed working under these realistic conditions. However, we do not want these integration tests to be executed during the CI/CD pipeline, since they require valid credentials not available in this stage. Due to this, each and every one of these executions requires an explicit definition of an environment variable as depicted in Figure 19 below.

```
@unittest.skipUnless(os.getenv("AWS"), "Define AWS variable to execute")
def test_deploy_destroy_aws(self):

    deployment_id = str(uuid.uuid4())
    a = Iem(
        Credentials(
            aws=Aws(
                access_key_id=os.getenv("AWS_ACCESS_KEY_ID"),
                secret_access_key=os.getenv("AWS_SECRET_ACCESS_KEY"),
            )
        )
    )
    with open("tests/resources/aws.zip", "rb") as binary_file:
        bundle = base64.b64encode(binary_file.read())

    a.deploy(deployment_id=deployment_id, bundle=bundle)

    a.destroy(deployment_id=deployment_id)
```

Figure 19 - An excerpt showing an integration test with AWS.

Finally, the various testing strategies explained in this section yield an excellent 87% coverage of the project, as it is generally accepted that 80% coverage is a good goal [7]. The coverage report is depicted in Figure 20.

Coverage report: 87%
coverage.py v7.2.0, created at 2023-04-04 16:34 +0200

Module	statements	missing	excluded	coverage
main.py	42	3	0	93%
src/core/__init__.py	0	0	0	100%
src/core/engine.py	96	25	0	74%
src/core/iem.py	113	17	0	85%
src/core/persistence.py	41	1	0	98%
src/core/utils.py	52	0	0	100%
Total	344	46	0	87%

coverage.py v7.2.0, created at 2023-04-04 16:34 +0200

Figure 20 - The coverage summary of the IEM.

The testing and coverage have been implemented utilizing the well-respected coverage⁵ framework for python.

2.1.7 Support for further Orchestration engines

The orchestration engines supported by the IEM rely on the providers supported by Terraform⁶. During this project, AWS, OpenStack, Azure, and VMWare vSphere have been tested and utilized for various reasons.

In Figure 21, the AWS provider implemented by Terraform⁷ is showcased. This provider leverages two main variables, the `access_key_id` and the `secret_access_key` which can be generated in the AWS account. This provider has been mainly used for testing purposes throughout the project.

```
"aws": {
  "access_key_id": "string",
  "secret_access_key": "string"
},
```

Figure 21 - AWS Credential handling

Next, the Azure provider implemented by Terraform⁸ is displayed. The following variables: `arm_client_id`, `arm_client_secret`, `arm_subscription_id`, and `arm_tenant_id` is necessary for utilizing the Azure provider. This provider has been mainly used for testing purposes throughout the project. This is depicted in Figure 22.

⁵ <https://coverage.readthedocs.io>

⁶ <https://registry.terraform.io/browse/providers>

⁷ <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

⁸ <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>

```
"azure": {  
  "arm_client_id": "string",  
  "arm_client_secret": "string",  
  "arm_subscription_id": "string",  
  "arm_tenant_id": "string"  
},
```

Figure 22 - Azure credential handling.

The OpenStack provider, displayed in Figure 23, has been of paramount importance in the PIACERE project, as it is utilized by the CSE (Canary Sandbox Environment) [8]. It is a more complex provider than the ones mentioned above as there are more variables involved (user_name, password, auth_url, project_name, region_name, domain_name, project_domain_name, user_domain_name). This provider has been extensively utilized in this project.

```
"openstack": {  
  "user_name": "string",  
  "password": "string",  
  "auth_url": "string",  
  "project_name": "string",  
  "region_name": "string",  
  "domain_name": "string",  
  "project_domain_name": "string",  
  "user_domain_name": "string"  
},
```

Figure 23 - Openstack credential handling.

Finally, the VMware provider, displayed in Figure 24, has been defined in this latest iteration to support one of the PIACERE use cases. It provides access to the VMWare vSphere. At the time of writing this deliverable, it has been successfully tested

```
"vmware": {  
  "user_name": "string",  
  "password": "string",  
  "server": "string",  
  "allow_unverified_ssl": "string"  
},
```

Figure 24 - VMWare vSphere credential handling.

In summary, the IEM has been designed to address various private and public cloud providers. In addition, it has been designed to evolve and be extensible to support current and future providers.

2.2 Functional description and requirements coverage

In the following table, Table 1, a list of the requirements directly addressed by the IEM are showcased. This list has been extracted from WP2 deliverable D2.1 [3]. All the requirements have been satisfied but REQ55, which has been discarded as no other component in the PIACERE ecosystem requires it. However, the IEM can satisfy this requirement seamlessly with its current architecture.

Table 1: User requirements addressed by the IEM

Req ID	Description	Status	Requirement Coverage at M30
REQ12	The IEM shall allow redeployment and reconfiguration, both full and partial, as allowed by the used IaC technology.	Satisfied	The project structure utilized within the PIACERE consortium enables the IEM to automatically detect failing stages of the deployment and trigger an automated redeployment of these stages. In addition, dedicated endpoints are being developed for further granularity in the redeployment.
REQ55	The IEM will log the whole IaC execution run, making metadata and metrics (time it took to run) about the creation of resources available to the rest of the PIACERE components.	Discarded	The IEM includes a built-in database in which it can keep track of the required metrics. Currently, only the information specified in REQ82 is included.
REQ81	IEM should be able to execute IaC generated by ICG for selected IaC languages (e.g., TOSCA / Ansible / Terraform)	Satisfied	ICG generates the code in a manner the IEM can understand its content. This is possible due to the use of configuration files which both the IEM and the ICG can communicate with.
REQ82	IEM shall register the status of past and present executions and enable an appropriate way to query it.	Satisfied	The IEM includes a built-in database in which it records the appropriate metadata that is required by other components.
REQ83	IEM should be able to communicate with the relevant actors (orchestrators, infrastructural elements) in a secure way.	Satisfied	It currently communicates successfully with the required components of the PIACERE ecosystem (e.g., PRC), and with external orchestrators (OpenStack, VMware vSphere).
REQ84	IEM should be able to utilize the required credentials in a secure way.	Satisfied	Credentials are never stored in the IEM, rather they are treated as environment variables and discarded with every new execution.
REQ85	IEM should be able to clean up the resources being allocated.	Satisfied	The clean up method has been already implemented and validated by the PRC, it completely tears down existing deployments freeing up the resources being used.
REQ87	IEM shall work against the production environment and the canary environment.	Satisfied	The IEM can communicate and orchestrate with various public and private cloud providers. The canary environment is implemented leveraging the OpenStack technology which has been fully tested in previous iterations of this component.

2.3 Main innovations

In this section the main innovations regarding the implementation and testing of the IEM are outlined, each represent one relevant contribution towards the goal of the IEM and KR10:

- It integrates various IaC technologies and glue them together seamlessly, this severely reduces the time required for orchestrating the various stages of the projects.
- It automatically handles some connectivity related issues, this way the final user utilizing the PIACERE ecosystem alongside the IEM does not need to be bothered about some common pitfalls of the utilized IaC technologies.
- It lightens the burden on DevOps professionals providing a streamline approach for the deployment orchestration of the utilized technologies.
- It provides an extensible interface so that even IaC technologies not yet considered in the deployment orchestration workflow can be easily integrated into the IEM and PIACERE ecosystem. This is possible due to the use of interfaces that permit future implementations without having to modify the current architecture significantly.
- It offers a secure way of interacting with various public and private cloud providers. Given that it does not store the credentials in the component itself, it would be hard for intruders to grasp any of the secrets of the providers in use.

DRAFT

3 Overview of preliminary experiments

In this section we explain the experiments that have been performed to validate the functionality and suitability of the IEM as part of the PIACERE ecosystem. Firstly, we validate the suitability of the IEM for deploying projects on AWS. Secondly, we validate the IEM and expanded its functionality with the Docker technology, Thirdly, we validate the suitability of the IEM in conjunction with the OpenStack private cloud provider. Finally, we validate the IEM with a common provider in various companies, which is VMWare vSphere.

3.1 Experiments on AWS

To kick off the first experiment and environment variable needs to be defined during its execution. This is preventing the experiment during the CI/CD pipeline where the required credentials are not yet available. To execute the experiment on AWS the command depicted in Figure 25 needs to be triggered, making use the AWS environment variable has been defined.

```
(.venv) josu@MKM0092A:~/prj/piacere/git/iem-api$ AWS=1 nose2 -v tests.it
test_deploy_destroy_aws (tests.it.test_it_iem.TestIem) ... ok
test_deploy_destroy_docker (tests.it.test_it_iem.TestIem) ... skipped Define INTEGRATION variable to execute
test_deploy_destroy_openstack (tests.it.test_it_iem.TestIem) ... skipped Define INTEGRATION variable to execute
```

Figure 25 - Running the experiment on AWS.

To orchestrate a deployment on AWS the credentials depicted in Figure 26 must be fed into the IEM through the REST API. These are the secrets that are to be utilized by AWS to provision the various infrastructural devices.

```
|- - -
input:
  - AWS_ACCESS_KEY_ID
  - AWS_SECRET_ACCESS_KEY
output: ~
engine: terraform
...

```

Figure 26 - Configuration file for AWS.

Once the deployment has finished, we can validate on the AWS web console that the appropriate infrastructural devices have been provisioned according to the design. This scenario is depicted in Figure 27.

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type
<input type="checkbox"/>	hello-terraform	i-022182377bb8d2256	Running	t2.micro

Figure 27 - Virtual Machine deployment on AWS.

This scenario briefly showcases the experiment on AWS, and it is publicly available on GitHub [9].

3.2 Experiments with Docker

To kick off an experiment with docker the appropriate environment variable needs to be already defined during the execution of the experiment. In this particular experiment, the docker compose file displayed in Figure 28 is to be deployed and triggered on the provisioned infrastructural device.

```
version: '3'
services:
  postgres:
    image: nginx
    ports:
      - "80:80"
```

Figure 28 - docker compose file for a single web server.

This experiment kicks off a virtual machine on the cloud provider and install all the requirements for the execution of docker and docker compose. If the experiment has succeeded, the entry page depicted in Figure 29 should be available in the freshly provisioned infrastructural device.

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 29 - Frontpage for the nginx web server provisioned.

This simple experiment successfully validates that the IEM can provide container functionalities on top of the projects to be deployed by the use cases. This experiment is publicly available on GitHub [10].

3.3 Experiments on OpenStack

For this cloud provider, there are various environment variables that need to be fed into the IEM. The functionality provided is like other providers but the syntax in Terraform differs. Fortunately, a previous component of PIACERE (i.e., PRC) provides a unified syntax for the definition of the various infrastructural devices. Figure 30 depicts the necessary inputs that are to be used by the OpenStack provider.

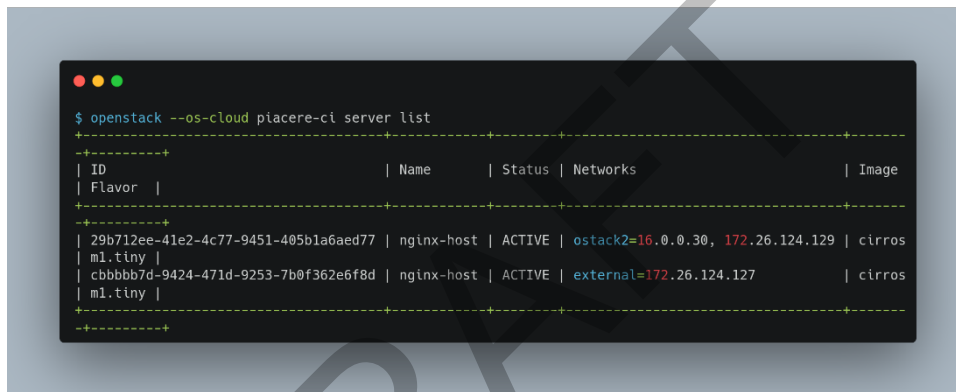
```

|---
engine: terraform
input:
  - OS_USERNAME
  - OS_PASSWORD
  - OS_AUTH_URL
  - OS_PROJECT_NAME
output:
  - instance_server_public_key_user1
  - instance_server_private_key_user1
  - instance_ip_vm1
...

```

Figure 30 - Configuration file for OpenStack.

After executing the project, the OpenStack provider should look as follows, which clearly depicts that the required virtual machines have been provisioned as per shown in Figure 31.



```

$ openstack --os-cloud piacere-ci server list
+-----+
| ID | Name | Status | Networks | Image |
+-----+
| 29b712ee-41e2-4c77-9451-405b1a6aed77 | nginx-host | ACTIVE | ostack2=16.0.0.30, 172.26.124.129 | cirros |
| m1.tiny |
| cbbbbb7d-9424-471d-9253-7b0f362e6f8d | nginx-host | ACTIVE | external=172.26.124.127 | cirros |
| m1.tiny |
+-----+

```

Figure 31 - Validate that the virtual machines have been provisioned on OpenStack.

This Experiment is publicly available on GitHub [11].

4 Lessons learnt and outlook to the future

In this section we provide an overview of the various lesson learnt over these 30 months of work. However, we particularly focus on the last six months of developments as these are the ones that have validated the suitability of the IEM to provide functionality to the various PIACERE use cases.

At the beginning of the project, we were obliged to make an educated guess on which would be the underlying technologies that would best fit the IEM.

- **Lesson learnt #1:** the technologies selected for the implementation were Terraform and Ansible. An extensive state of the art was performed during the first [1] of the project but where we resorted to these two as they are well supported by the community and we as the consortium had previous experience with both. They have proven to be a good choice as they are extensible enough for fulfilling the requirements from the first year without extensive changes in the architecture.

Then, these technologies provide common user clients and libraries, both would have been suitable options for the implementation of the IEM. However, choosing the approach wisely would have an important impact on the extensibility of the IEM.

- **Lesson learnt #2:** having spent significant time during the first year of the project on the existing technologies for the IEM has proven to be time well spent. The IaC ecosystem is continuously changing, hence it is risky to adopt a technology due to the evolving nature of cloud providers. However, both Terraform and Ansible are still well respected, continuously evolving technologies, and have fulfilled the aspiration of the PIACERE project and the IEM.

During the first year of the project, we chose to utilize repository handles that point to the projects to be deployed. This proved to be efficient, but it has some undesired issues.

- **Lesson learnt #3:** during the first year it was decided that repository handles were going to be used for the implementation of the various PIACERE components. This provided an easy manner of communication, and the possibility of rolling back and forth between repository commits if necessary. However, it came with some drawbacks such as that each component was in charge of downloading the full repository every single time it, they needed to interact with it. Due to this, during this last six months it has been decided that the whole folder was going to be passed along the workflow. This has reduced the implementation time and still provides the desired functionality.

In terms of security, the IEM oversees the interaction with the various public and private cloud providers, hence it manages important security information that need to be handled appropriately.

- **Lesson learnt #4:** we chose to never store credentials on the IEM itself as that would be a severe security risk. This has proven to be an excellent decision as there is virtually no risk of security breaches.

In summary, the design of the IEM undergone during the first year of the project has proven to be very resourceful as no major architecture changes had to be done during the subsequent iterations. As for the future work, more efforts need to take place to support an expanded self-healing strategy for the PIACERE project.

5 Conclusions

This document serves as the third and last iteration of the IEM documentation and provides a full overview of this component very focussed on the last six months of development. The IEM aspires to provide a unified interface for the deployment orchestration of the multilingual IaC projects necessary for the PIACERE ecosystem.

There has been major changes over the last six months among which we would like to highlight that: the IEM has become database agnostic by implementing a well respect ORM tool, during this year support for projects utilizing containers have been adopted, the requirements of the tools in PIACERE are less as they do not need to download the whole project very time using git, logging capabilities and feedback to both developers and end users has been increased, a rich ecosystem of public and private cloud providers have been implemented and the whole component has been extensively tested to be used as part of PIACERE.

In terms of the major innovations obtained during this last iteration, it is worth highlighting that the IEM is able to glue together popular IaC technologies that would require extensive tweaking otherwise. In addition, some common pitfalls that tend to be manually solved are now automatically addressed by the IEM. In addition, it provides an extensible secure manner of utilizing IaC technologies by providing an interface for future providers, even those not even implemented at the time of writing this document can be integrated in future iterations. Finally, it aspires to lower the burden on the professionals as they do not be continuously learning various technologies to deploy the projects.

DRAFT

6 References

- [1] PIACERE Consortium, «D5.1 - IaC Execution platform prototype- v1.1,» 2021.
- [2] PIACERE Consortium, «D5.2 - IaC Execution platform prototype- v2,» 2022.
- [3] PIACERE Consortium, “D2.1 PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v1,» 2023.
- [4] S. Hammond and D. Umphress, “Test driven development: the state of the practice,» in *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference*, 2012.
- [5] Smartbear, “What Is Unit Testing?,” [Online]. Available: <https://smartbear.com/learn/automated-testing/what-is-unit-testing/>. [Accessed 17 5 2023].
- [6] R. Awati, “Integration testing or integration and testing (I&T),” [Online]. Available: <https://www.techtarget.com/searchsoftwarequality/definition/integration-testing>. [Accessed 17 5 2023].
- [7] S. Pittet, “What is code coverage?,” [Online]. Available: <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>. [Accessed 17 5 2023].
- [8] PIACERE Consortium, «D5.4 - Canary environment prototype - v1_V1.0».
- [9] PIACERE Consortium, “AWS Experiment,» TecNALIA Research & Innovation, [Online]. Available: <https://git.code.tecnalia.com/piacere/public/the-platform/iem/-/blob/y3/iem-api/tests/resources/aws.zip>. [Accessed 19 5 2023].
- [10] PIACERE Consortium, “Docker Experiment,» TecNALIA Research & Innovation, [Online]. Available: <https://git.code.tecnalia.com/piacere/public/the-platform/iem/-/blob/y3/iem-api/tests/resources/docker.zip>. [Accessed 19 5 2023].
- [11] PIACERE Consortium, “OpenStack Experiment,» TecNALIA Research & Innovation, [Online]. Available: <https://git.code.tecnalia.com/piacere/public/the-platform/iem/-/blob/y3/iem-api/tests/resources/openstack.zip>. [Accessed 19 5 2023].

APPENDIX: Implementation, delivery, and usage

1 Implementation

This section refers to the implementation details of the IEM. First, we provide a description of the IEM fitting into the overall PIACERE Architecture. Then, we provide a technical description, including the prototype architecture, the components description and we finalize with the technical specifications.

1.1 Fitting into overall PIACERE Architecture

The IEM is the component of the PIACERE architecture that receives the IaC code being created on previous stages of the PIACERE workflow by the PRC. The component which interacts the most with the IEM is the Runtime Controller (PRC), as can be seen in the following figure 32. The main interactions are as follows:

- The PRC communicates with the IEM to trigger a deployment. In order to do so, it hands over the following information:
 - The base64 encoded zip bundle that is going to be executed by the IEM.
 - The secrets that are necessary for the execution of the given deployment. These secrets are never persisted by the IEM to guarantee their safety.
- The IEM orchestrates the deployment on the desired public or private cloud provider and saves the necessary information for further queries.

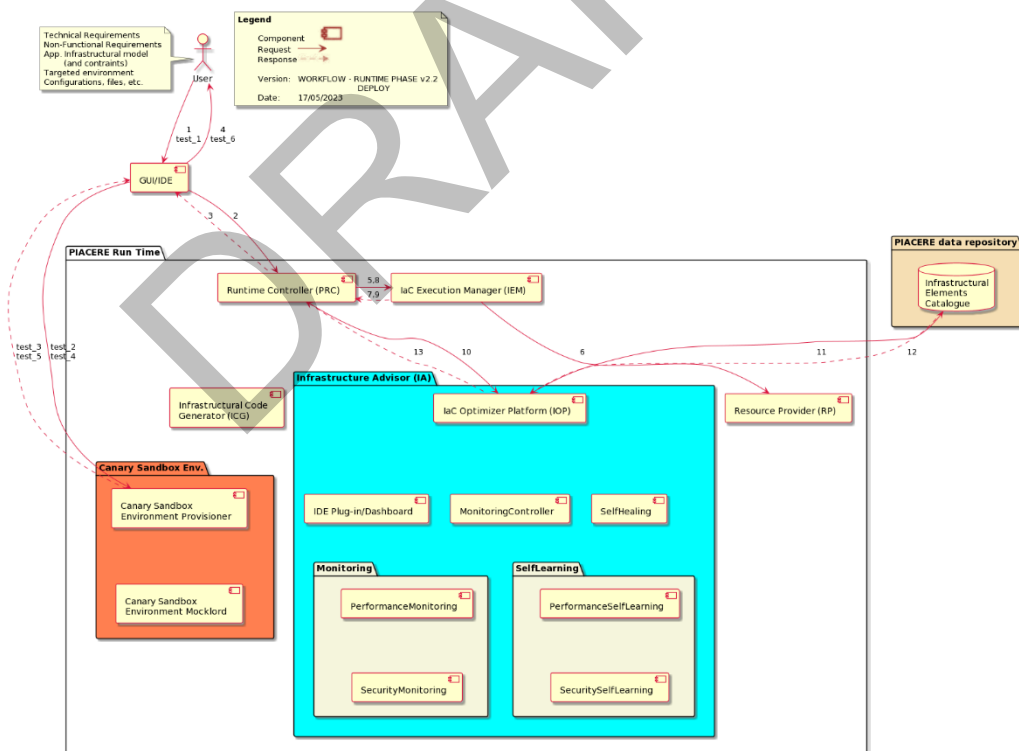


Figure 32 - PIACERE Runtime Workflow

The IEM is yet again pivotal in this endeavour. This is because it provides means to other components in the architecture for triggering specific actions on already running components. This way, the monitoring and self-healing of the architecture can be accomplished.

1.2 Technical description

This section describes the technical details of the IEM component. First, an overview of the prototype architecture is depicted and explained. Then, each of the components is explained in further detail. Finally, a description of the technical specifications is provided.

1.2.1 Prototype architecture

The architecture components that comprise the entirety of the IEM component are depicted in the following figure. This is a REST API that oversees and manages the interaction with the inner functionalities of the IEM. The Core of the system where the business logic resides and can forward the different actions appropriately. The Persistence component which oversees storing the metrics and metadata related with past and present deployments. Finally, the executors understand the IaC code being forwarded to the IEM and orchestrate it against the different public and private cloud providers. All the components described in the following figure 34 are explained in the following section in further detail.

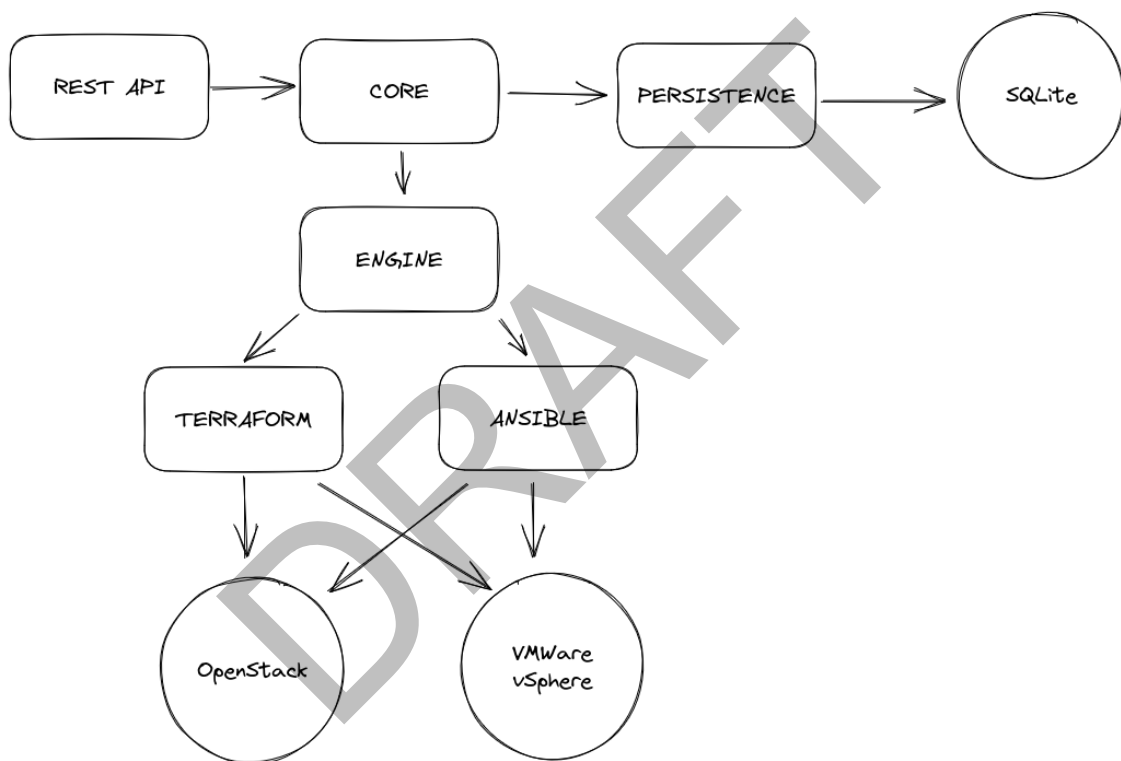


Figure 33 - IEM prototype architecture.

The following image 35 depicts the inner functioning of a deployment within the IEM. The main difference from previous iterations of this diagram is that the IaC Repository has been removed from the main PIACERE workflow. The runtime Controller triggers a deployment and immediately receives a response stating whether the deployment has been accepted. Then, the IEM validates the triggers the fresh deployment using the implemented executors, while storing at every stage the status of that deployment.

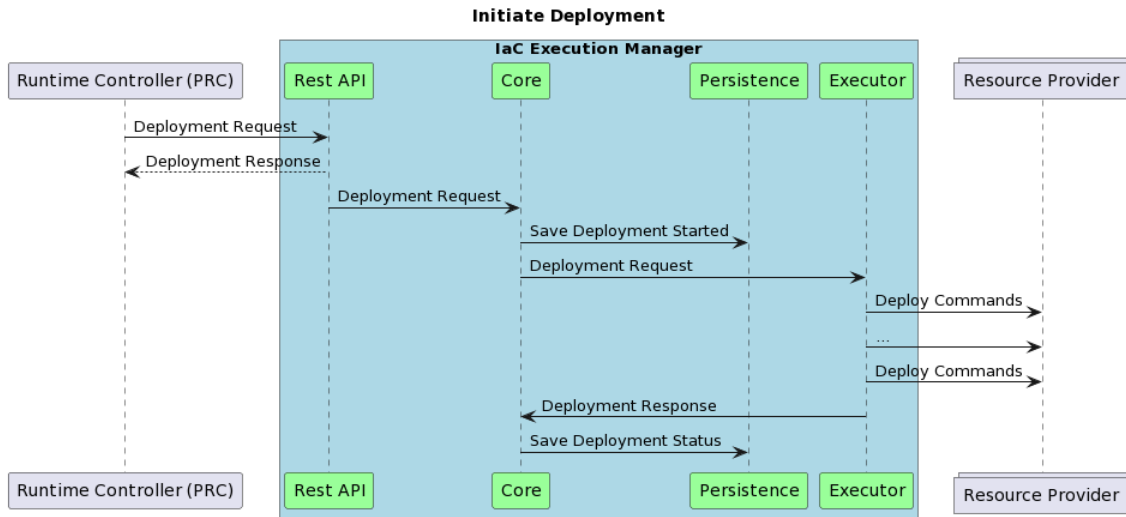


Figure 34 - IEM Initiate Deployment Sequence Diagram

In the following image, the flow in which the PRC queries the status of a given deployment is depicted. The PRC hands over to the request through the IEM’s API specifying the unique identifier of the given deployment. Then, the IEM is able to retrieve this information from the Persistence component.

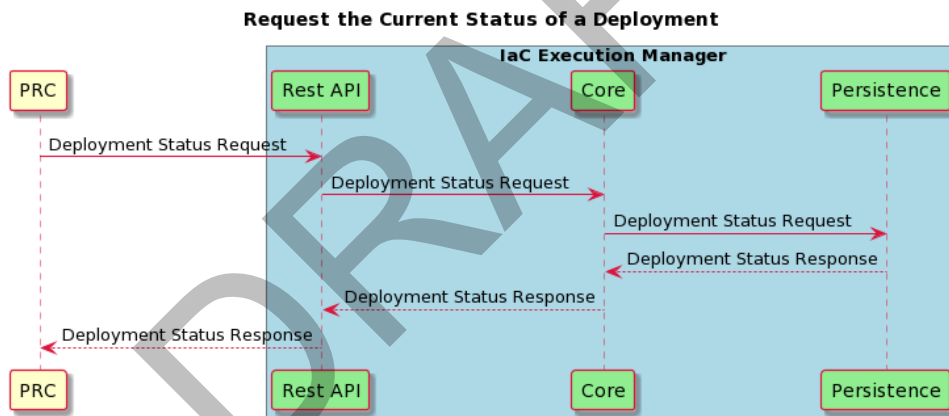


Figure 35 - IEM Request the Current Status of a Deployment

The following figure showcases the sequence diagram that represents the undeployment workflow for the IEM. This scenario mimics the deployment workflow, with the particularity that in this case there is no need to receive the entire infrastructure. Instead, only the unique identifier of the deployment and the required credentials are necessary.

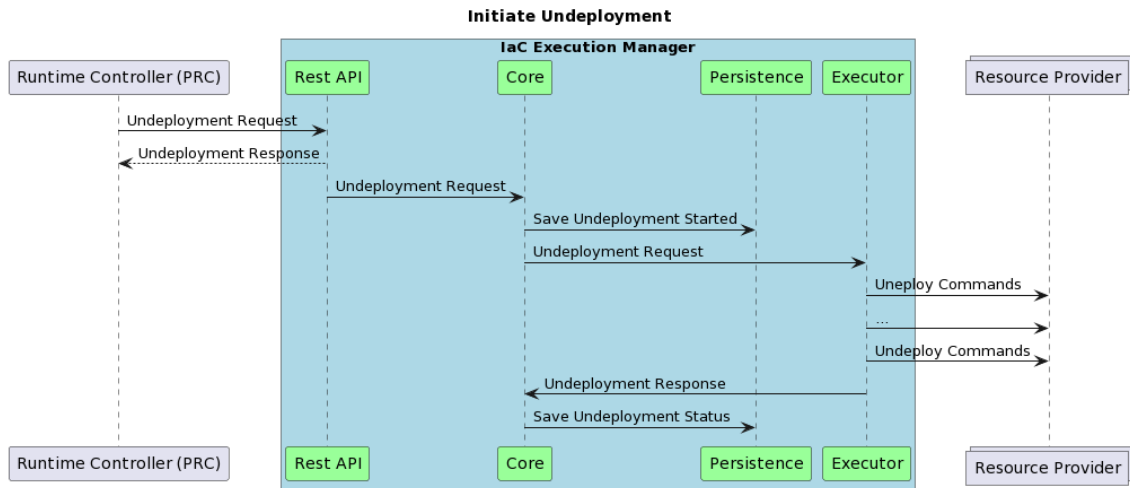


Figure 36 - IEM Initiate Undeployment Sequence Diagram

1.2.2 Components description

The first prototype of the IEM component is comprised of the subcomponents described above. In this section, we take a closer look at each and explain their functionality in further detail.

- **REST API:** this subcomponent is the entry point of all the requests that the IEM need to process. An OpenAPI specification file is provided so the interaction with it becomes as seamless as possible.
- **Core:** this subcomponent contains the business logic of the IEM component. It oversees the flow of the different calls of other PIACERE components appropriately.
- **Persistence:** this subcomponent contains the persistence logic that the IEM component is going to utilize. It is a relational database that will provide the data required for the requests for information by other components. This is information and metadata for past and present executions of the different components. Most of the development for this subcomponent has been undergone during the second year of the project. During this last development iteration, and ORM tool has been utilized to abstract the database technology from the development, which provides the IEM with more flexibility.
- **Executors:** the executors are the subcomponents in charge of the execution of the different technologies that the IEM supports. Two different IaC technologies are supported: i) Ansible for the configuration of the different dependencies that the deployment requires, ii) Terraform for the provisioning of the infrastructural elements required for the deployments to be successfully executed, iii) docker for the application lifecycle management

1.2.3 Technical specifications

The prototype has been developed in the Python programming language, specifically version 3.9.5. It has been selected because python is very proficient at interacting with the currently used IaC technologies (Terraform, Ansible, Docker), and provides an easy manner to add additional technologies in the future.

The input and output interactions of this component are supported by FastAPI, specifically version 0.73.0, which provides a myriad of functionalities for the implementation of REST interfaces, which is the primary way of interacting with the IEM component. In addition, it provides functionalities for providing an OpenAPI implementation that can be used by other

components in the PIACERE infrastructure, as it not only provides an easy way to understand the inputs and outputs required by this component, but also an automatic way to generate the server and client sides if desired. The IEM is served using the [uvicorn](https://www.uvicorn.org/)⁹ ASGI web server, which provides the system with a minimal low-level interface for async development.

The persistence layer is in essence a relation database. This piece of the component has been completed during the second year of the project. The persistence layer has been implemented using SQLite, and it contains a set of relational tables that will information related to past and presents deployments. SQLAlchemy has been introduced to manage the database and to abstract the development from the technological implementation of it.

The IaC code that have been selected to use within the PIACERE framework have been Terraform and Ansible. The former is a well-known technology utilized in the field of infrastructural device provisioning and can interact with a large variety of public and private cloud providers (e.g., AWS, Azure, OpenStack, VMWare vSphere). The latter, on the other hand, is an established tool in industry that is commonly used for the configuration of the infrastructural devices required for the deployment. It has a myriad of modules that can be used for the different nuances that comprise a software project deployment such as dependency management, services configuration, and configuration management. Access to the IEM is hardened with an API key that needs to be fed into the system every single time that an action is required to be taken. Finally, the actual delivery of the component in a containerized manner, with the docker technology.

DRAFT

⁹ <https://www.uvicorn.org/>

2 Delivery and usage

This section offers information on the package itself. First, information about the structure of the repository is provided. Then, instructions on how to install the package are offered. Thirdly, a manual on how to utilize the IEM component is explained in detail. Finally, licensing information and downloading instructions are provided.

2.1 Package information

This section gives an overview on the structure of the IEM component. The root structure for the component is showcased in the following image 38, information about the most relevant files and folders are then explained.

Name	Last commit	Last update
doc	Update kr-10.feature	2 months ago
docs	[DOCS] Another tiny docs README upd...	9 months ago
iem-api	fix tests	3 weeks ago
.gitignore	expose HTTP requests	10 months ago
.gitlab-ci.yml	increase verbosity in testing	3 weeks ago
LICENSE	Add LICENSE	6 months ago
README.md	increase verbosity in testing	3 weeks ago
docker-compose.yml	add persistence layer	1 year ago
openapi.json	self-healing endpoints	1 month ago
sonar-project.properties	change source path in sonar	1 year ago

Figure 37 - Root folder for the PIACERE IEM component.

The main files and folders that can be found in this picture are the following:

- The docs folder contains information such as a gherkin formatted feature files to be able to understand the behaviour of the IEM, and the detailed information about its use.
- The iem-api folder contains the actual source code of this component including files related with the build process such as the Dockerfile.
- The “.gitlab-ci.yml” file oversees the CI/CD pipeline that is triggered every time a modification to the IEM takes place.
- The docker compose file oversees the deployment of this component on the production environment.
- Detailed information on how to interact with the IEM is specified in the openapi.json file.
- The sonar-project.properties file oversees the affairs related with code quality in the given project.

2.2 Installation instructions

The IEM prototype can be found in Tecnalía’s GitLab repository (download instructions at the end). There are a few files that are of paramount importance for getting the IEM prototype up and running. The first one is the “requirements.txt” file, which offers an up-to-date list of the IEM dependencies alongside their specific version. It would be better to install these

requirements in a virtual environment in order not to mess with the local installation of similar packages. There are many ways to start a virtual environment, and specific instructions are out of the scope of this deliverable. Hence, in this document brief instructions towards the installation and use of a tool for creating virtual environments are provided. The following image showcases how to instantiate a virtual environment.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The terminal displays the command `$ python -m venv .venv` in a light blue monospace font.

Figure 38 - Create the virtual environment.

Next, this virtual environment needs to be activated, which can be accomplished with the following command.


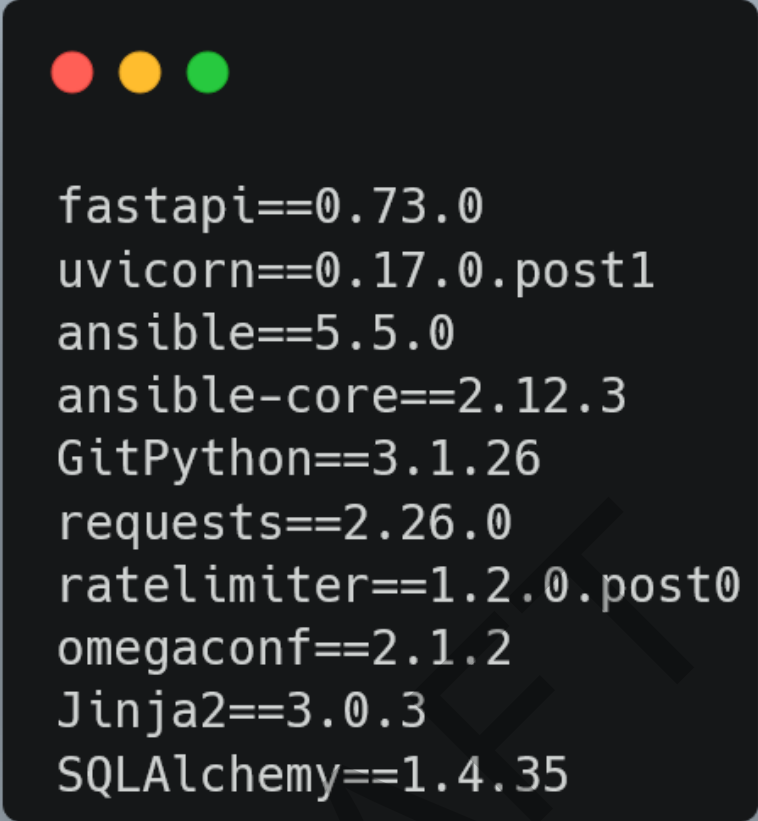
A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The terminal displays the command `$ source .venv/bin/activate` followed by a new prompt `(.venv) $` in a light blue monospace font.

Figure 39 - Activate the virtual environment.

Now that a virtual environment has been created and it is ready to be used, the “requirements.txt” becomes handy. In the following image it can be seen the content of it, which the precise libraries alongside their version that need to be installed to run the IEM prototype. It also showcases how to install these requirements with the python package manager.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The terminal displays the content of a requirements.txt file, listing various Python dependencies with their version constraints.

```
fastapi==0.73.0
uvicorn==0.17.0.post1
ansible==5.5.0
ansible-core==2.12.3
GitPython==3.1.26
requests==2.26.0
ratelimiter==1.2.0.post0
omegaconf==2.1.2
Jinja2==3.0.3
SQLAlchemy==1.4.35
```

Figure 40 - Excerpt showing the content of the requirements.txt file for dependency management.

Unfortunately, not all the requirements can be centralized in this manner since the Terraform client needs to be installed separately. Please refer to the official documentation¹ to get this done. At this stage, all the IEM dependencies should be installed and ready to be used. To make sure that this is in fact the case, the following image shows how to run the different tests that make sure the prototype is properly working.



```
$ nose2 -v
test_deploy_destroy_aws (tests.it.test_it_iem.TestIem) ... skipped Define AWS variable to execute
test_deploy_destroy_docker (tests.it.test_it_iem.TestIem) ... skipped Define INTEGRATION variable to
execute
test_deploy_destroy_openstack (tests.it.test_it_iem.TestIem) ... skipped Define INTEGRATION variable to
execute
test_get_all_deployments (tests.unit.test_persistence.TestPersistence) ... ok
test_get_deployment (tests.unit.test_persistence.TestPersistence) ... ok
test_get_deployment_not_ok (tests.unit.test_persistence.TestPersistence) ... ok
test_insert_deployment (tests.unit.test_persistence.TestPersistence) ... ok
test_get_all_deployments (tests.unit.test_iem.TestIem) ... ok
test_get_deployment_not_ok (tests.unit.test_iem.TestIem) ... ok
test_get_deployment_ok (tests.unit.test_iem.TestIem) ... ok
test_deployments (tests.unit.test_main.TestMain) ... ok
test_get_all_deployments (tests.unit.test_main.TestMain) ... ok
test_get_one_deployment (tests.unit.test_main.TestMain) ... ok
test_root (tests.unit.test_main.TestMain) ... ok
test_root_no_apikey (tests.unit.test_main.TestMain) ... ok
test_undeploy (tests.unit.test_main.TestMain) ... ok

-----
Ran 16 tests in 0.319s

OK (skipped=3)
```

Figure 41 - Execute all the tests of the IEM with the nose tool.

In the image 42 above, we have utilized the nose2¹⁰ tool, but other tools such as coverage¹¹ can be utilized in a similar manner. The IEM component will be deployed containerized with the docker framework. There is a Dockerfile that helps in making this happen. The following excerpt (Figure 43) shows the content of this file. This Dockerfile is based on the official image provided by the terraform team. Given the installation of this tool can be the most time consuming and error prone, we resort to this image in order to achieve better quality and efficiency.

¹⁰ <https://docs.nose2.io/en/latest/>

¹¹ <https://coverage.readthedocs.io/>


```

FROM hashicorp/terraform:1.1.4

COPY requirements.txt /tmp/requirements.txt
RUN apk add py3-pip cargo g++ python3-dev file libffi-dev openssl-dev bash python3=3.9.16-r0 gnupg
RUN pip3 install -r /tmp/requirements.txt
# install docker stack
RUN apk add docker docker-compose

ENV API_KEY=changeme
ENV IEM_HOME=/opt/iem/
ENV DOCKERIZED=true

COPY src/resources/ansible.cfg /etc/ansible/ansible.cfg

# RUN adduser -h ${IEM_HOME} -S -D iem
COPY certs/config ${IEM_HOME}.ssh/config
COPY certs/id_rsa ${IEM_HOME}.ssh/id_rsa
COPY certs/id_rsa.pub ${IEM_HOME}.ssh/id_rsa.pub
RUN adduser -h ${IEM_HOME} -S -D iem && \
  chown -R iem ${IEM_HOME} && \
  chmod 0700 ${IEM_HOME}.ssh && \
  chmod 0644 ${IEM_HOME}.ssh/config && \
  chmod 0600 ${IEM_HOME}.ssh/id_rsa && \
  chmod 0644 ${IEM_HOME}.ssh/id_rsa.pub
USER iem
RUN ansible-galaxy collection install community.general
COPY roles.yml /tmp/roles.yml
RUN ansible-galaxy install -r /tmp/roles.yml

RUN mkdir -p ${IEM_HOME}db && \
  mkdir -p ${IEM_HOME}deployments

COPY src ${IEM_HOME}src
COPY main.py ${IEM_HOME}main.py
COPY logging.ini ${IEM_HOME}logging.ini

ENTRYPOINT ["/usr/bin/env"]
WORKDIR ${IEM_HOME}
CMD /usr/bin/uvicorn main:app --host 0.0.0.0 --log-level info
EXPOSE 8000

```

Figure 42 - Excerpt showing the Dockerfile utilized for generating the containerized image of the IEM.

At this moment, the IEM component can be generated as shown in the following image 44.

```

docker build -t optima-piacere-docker-dev.artifact.tecnalia.com/wp5/iem-api:y3 .

```

Figure 43 - Excerpt showing the building process for the IEM component.

2.3 User Manual

This subsection gives an overview on how the communication with the IEM should take place. In particular, this is detailed in an OpenAPI specification file which different components can adhere to, in order to utilize the different functionalities provided by the IEM.

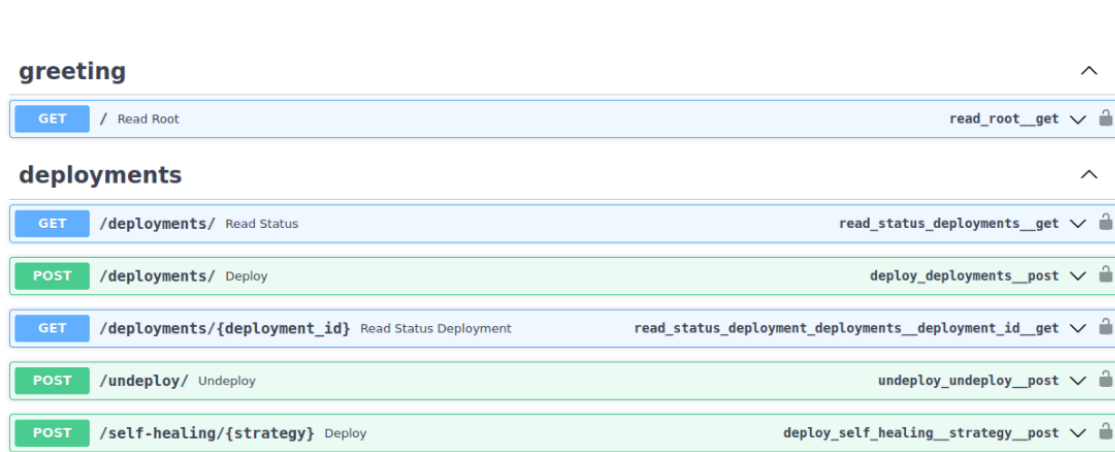


Figure 44 - OpenAPI specification for the interaction with the IEM component.

The Figure 45 above showcases a screenshot of the OpenAPI specification file that resides in the IEM's GitLab repository. For the first prototype, five different endpoints have been defined for the interaction of PIACERE components with the IEM. The details on how these endpoints provide to the components is detailed below:

- GET /deployments/: it provides information about all the deployments that are currently taking place within the PIACERE framework.
- POST /deployments/: it kicks off a deployment, if the deployment has already been started in a previous iteration, it updates the given deployment with the new configuration.
- GET /deployments/{deployment_id}/: it yields detailed information about the status of a given deployment. The deployment to be retrieved should be passed as a path parameter.
- POST /undeploy/: it tears down the deployment specified in the body by the unique identifier.
- POST /self-healing/{strategy}: this endpoint is under heavy development; it provides means for the other PIACERE components in the architecture to execute various self-healing strategy. At the time of writing this document, this endpoint requires further work and is not yet functional.

The following image 46 provides an overview of the various credentials that can be utilized by the IEM.

```
{
  "deployment_id": "string",
  "credentials": {
    "aws": {
      "access_key_id": "string",
      "secret_access_key": "string"
    },
    "azure": {
      "arm_client_id": "string",
      "arm_client_secret": "string",
      "arm_subscription_id": "string",
      "arm_tenant_id": "string"
    },
    "openstack": {
      "user_name": "string",
      "password": "string",
      "auth_url": "string",
      "project_name": "string",
      "region_name": "string",
      "domain_name": "string",
      "project_domain_name": "string",
      "user_domain_name": "string"
    },
    "vmware": {
      "user_name": "string",
      "password": "string",
      "server": "string",
      "allow_unverified_ssl": "string"
    },
    "docker": {
      "server": "string",
      "user_name": "string",
      "password": "string"
    }
  },
  "bundle": {
    "base64": "string"
  }
}
```

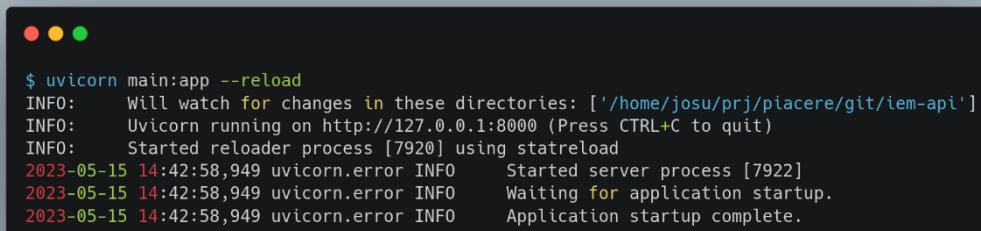
Figure 45 - Credentials to be used by the IEM.

To run an instance of the IEM generated image, the following snippet showcased in the image 47 can be triggered.

```
docker run -p 8000:8000 optima-piacere-dockerdev.artifact.tecnalia.com/wp5/iem-api:y3
```

Figure 46 - Run the IEM image.

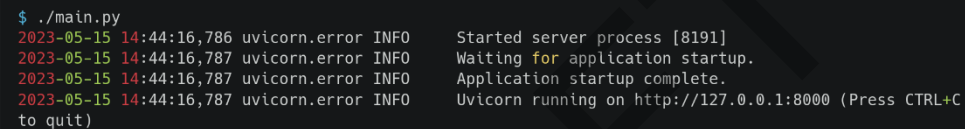
There are other ways of executing the IEM. For instance, the following image depicts how to run the IEM with the uvicorn server, which is the one that serves its functionality in the container image.



```
$ uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['/home/josu/prj/piacere/git/iem-api']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [7920] using statreload
2023-05-15 14:42:58,949 uvicorn.error INFO Started server process [7922]
2023-05-15 14:42:58,949 uvicorn.error INFO Waiting for application startup.
2023-05-15 14:42:58,949 uvicorn.error INFO Application startup complete.
```

Figure 47 - Execute the IEM with the uvicorn server.

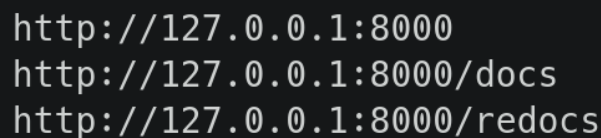
In addition, the main file can also be executed to obtain a similar functionality. This is useful mainly for development and debugging purposes.



```
$ ./main.py
2023-05-15 14:44:16,786 uvicorn.error INFO Started server process [8191]
2023-05-15 14:44:16,787 uvicorn.error INFO Waiting for application startup.
2023-05-15 14:44:16,787 uvicorn.error INFO Application startup complete.
2023-05-15 14:44:16,787 uvicorn.error INFO Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Figure 48 – Execute the IEM directly.

The following image depicts the various endpoints that can be used to check the OpenAPI documentation.



```
http://127.0.0.1:8000
http://127.0.0.1:8000/docs
http://127.0.0.1:8000/redocs
```

Figure 49 - Various documentation endpoints available in the IEM.

2.4 Licensing information

This component is offered under Apache 2.0 license. Detailed information can be found in the GitLab repository.

<https://git.code.tecnalia.com/piacere/public/the-platform/iem/-/blob/y1/LICENSE>

2.5 Download

The source code for the IEM prototype is available Tecnalia's GitLab repository. To get all the necessary files to utilize it, use the following link:

<https://git.code.tecnalia.com/piacere/public/the-platform/iem>

DRAFT