5Genesis

5TH GENERATION END-TO-END NETWORK, EXPERIMENTATION, SYSTEM INTEGRATION, AND SHOWCASING

[H2020 - Grant Agreement No. 815178]

Deliverable D3.15

Experiment and Lifecycle Manager (Release A)

Editor	A. Díaz Zayas (UMA)
Contributors	UMA (Universidad de Málaga), TID (Telefónica Investigación y Desarrollo), FOG (Fogus Innovations & Services P.C.)
Version	1.0
Date	October 15th, 2019
Distribution	PUBLIC (PU)

List of Authors

UMA	UNIVERSIDAD DE MÁLAGA
A. Díaz, B. García, P. Merino	
TID	TELEFÓNICA INVESTIGACIÓN Y DESARROLLO
D. Artuñedo	
FOG	FOGUS INNOVATIONS & SERVICES P.C.
D. Tsolkas	

Disclaimer

Copyright

TELEFONICA INVESTIGACION Y DESARROLLO SA	Spain
UNIVERSIDAD DE MALAGA	Spain
UNIVERSITAT POLITECNICA DE VALENCIA	Spain
UNIVERSITY OF SURREY	UK

Version History

Rev. N	Description	Author	Date
1.0	Release of D3.15	Almudena Díaz Zayas	15/10/2019

LIST OF ACRONYMS

Acronym	Meaning
3GPP	Third Generation Partnership Project
5G PPP	5G Infrastructure Public Private Partnership
API	Application programming interface
CPU	Central Processing Unit
CQI	Channel Quality Indicator
C-RAN	Cloud-RAN
CSI	Channel State Information
DUT	Device Under Test
E2E	End To End
EaaS	Experimentation as a Service
EARFCN	Evolved-UTRA Absolute Radio Frequency Number
eMBB	Enhanced Mobile Broadband-5G Generic Service
eNB	eNodeB, evolved NodeB, LTE eq. of base station
ELCM	Experiment Lifecycle Manager
EU	European Union
EPC	Evolved Packet Core
ETL	Extract, Transform, and Load
ETSI	European Telecommunications Standards Institute
EUTRAN	Evolved Universal Terrestrial Access network
FDD	Frequency Division Duplexing
GPS	Global Positioning System
ICCID	Integrated Circuit Card Identifier
ICMP	Internet Control Message protocol
IMEI	International Mobile Station Equipment Identity
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
IOT	Internet of Things
KPI	Key Performance Indicator
LAC	Location Area Code
LTE	Long-Term Evolution
LTE-A	Long-Term Evolution - Advanced
MAC	Medium Access Control
MANO	NFV MANagement and Organisation
MCC	Mobile Country Code
MCS	Mission Critical Services
MCSI	Modulation and Coding Scheme Index
MEC	Mobile Edge Computing
MIMO	Multiple Input Multiple Output
MME	Mobility Management Entity
mMTC	Massive Machine Type Communications-5G Generic Service
MNC	Mobile Network Code

MOCN	Multiple Operator Core Network
MONROE	Measuring Mobile Broadband Networks in Europe.
NFV	Network Function Virtualisation
NGMN	Next generation mobile networks
NMS	Network Managment System
OFDM	Orthogonal Frequency Division Multiplexing
PoC	Proof of concept
PCRF	Policy and Charging Rules Function
PDCP	Packet Data Convergence Protocol (PDCP)
PDSCH	Physical Downlink Shared Channel
PoP	Point of Presence
POSIX	Portable Operating System Interface
P-GW	Packet Data Node Gateway
PLMN	Public Land Mobile Network
PMI	Precoding Matrix Indicator
PNF	Physical Network Functions
PRB	Physical Resource Block
RAN	Radio Access Network
REST	Representational State Transfer
RSCP	Received Signal Code Power
RSRP	Reference Signal Received Power
RSRQ	Reference Signal Received Quality
RSSI	Received Signal Strength Indicator
RTT	Round trip time
SCPI	Standard Commands for Programmable Instruments
SIM	Subscriber Identity Module
SIMO	Single input, multiple output
TAP	Test Automation Platform
UDP	User datagram Protocol
UE	User Equipment
uRLLC	Ultra-Reliable, Low-Latency Communications
YAML	YAML Ain't Markup Language (human readable data serialization language)

Executive Summary

The Experiment Life Cycle Manage, or ELCM, is part of the coordination layer of the 5GENESIS architecture and is responsible of the scheduling and execution of experiments. It handles the life cycle of an experiment from start to end, keeping the experiment in an internal queue until all required resources for the experiment are available, using independent executors to run the experiment and recovering the generated results.

The general codebase of the ELCM will be common to all the platforms of the 5GENESIS Facility, however, the ELCM can be customized by each platform by modifying the contents of the Facility Registry and the ELCM settings (information about this file can be seen in Annex 2 YML Configuration File), modifying the generation of the Platform Specific Configuration created by the Composer.

The ELCM has been developed from the ground up using Python[1], and uses different interfaces for communicating with specific elements of the platforms. Additionally the ELCM exposes an internal web administration interface developed in Flask[2].

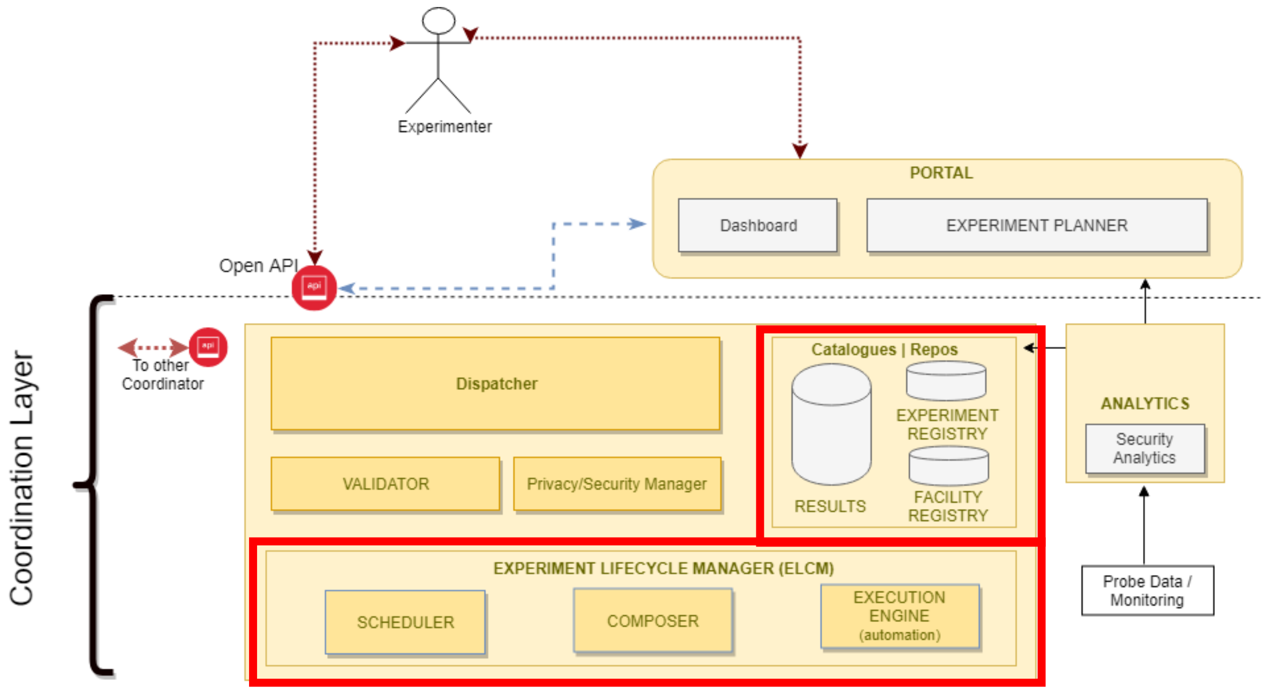


Figure 1 The ELCM component in the Coordination Layer of the 5GENESIS reference architecture

Table of Contents

1. Introduction

1.1. Purpose of the document

This deliverable describes the progress in designing and implementing the Experiment Life Cycle Manager (ELCM) for Release A. The ELCM is the entity that performs the management, orchestration and execution of Experiments in the 5GENESIS Platforms, and has been developed from the ground up during Cycle A. The reader can also find details about the different interfaces of this entity, namely the northbound interface (exposed by the ELCM) and the southbound interfaces (used for controlling the different elements of the 5GENESIS facilities during Release A). The actions planned for the following integration cycles of the project, regarding the ELCM development, are included as well.

It is noted that the structure of the document serves as a placeholder for the next deliverable of Task 3.8, as an effort to avoid repetition and include only the delta (changes and updates) on the ELCM toward the final version of this entity (D3.16, M30).

1.2. Document dependencies

The ELCM design and implementation is based on specifications and requirements described in the first release of the Architecture related deliverables. Table 1 summarizes the relevance towards the deliverables produced by WP2.

Id	Document title	Relevance
D2.2 [3]	5GENESIS Overall Facility Design and Specifications	The 5GENESIS facility architecture is defined in this document. The list of functional components to be deployed in each testbed is defined.
D2.3 [4]	Initial planning of tests and experimentation	This document describes the different components of the coordination layer and defines the sequence of interactions between the components of the facility during an experiment execution.

Table 1: Document dependencies

1.3. Structure of the document

The document is structured as follows:

- Section 1, *Introduction* (the present section)
- Section 2, *ELCM Design*, describes in detail the design principles of the ELCM
- Section 3, *ELCM Implementation*, describes the implementation of the ELCM Release A
- Section 4, *ELCM Northbound and Southbound Interfaces*, presents the interfaces and helpers that can be used by the ELCM during the execution of an experiment.

- Section 5, *Release A Summary and Future Plans*, provides a small summary of the features implementer for Release A, and describes the works planned for the following cycles.

1.4. Target audience

This document provides details about the functionality supported by the ELCM, as well as high-level information regarding its design and implementation. However, specific details that may only be useful while contributing to the codebase of the ELCM are not included.

Therefore, the target audience of this document are the 5GENESIS Platform administrators, who are required to know about the general implementation aspects of the ELCM in order to effectively configure and manage this element.

2. ELCM Design

The ELCM is divided in 3 main components, as well as several auxiliary elements. These components are:

- The Scheduler is responsible for managing the execution of the experiments on a higher level: An experiment execution is divided in 3 stages (Pre-Run, Run and Post-Run), and the Scheduler keeps track of the execution of each of these stages for multiple experiments in parallel.
- The Execution Engine includes the logic for managing the execution of each experiment stage, by generating an independent Executor. The progress in each Executor is further divided in different Tasks, which are dependent on the test case and the equipment involved in the experiment.
- The Composer is responsible for creating the Platform Specific Configuration of the received experiments. The configuration generated includes the Tasks to be run by the Executors and will depend on the contents of the Facility Registry. The Facility Registry is the entity that defines the expected behavior of the platform when specific test cases and equipment are tested.

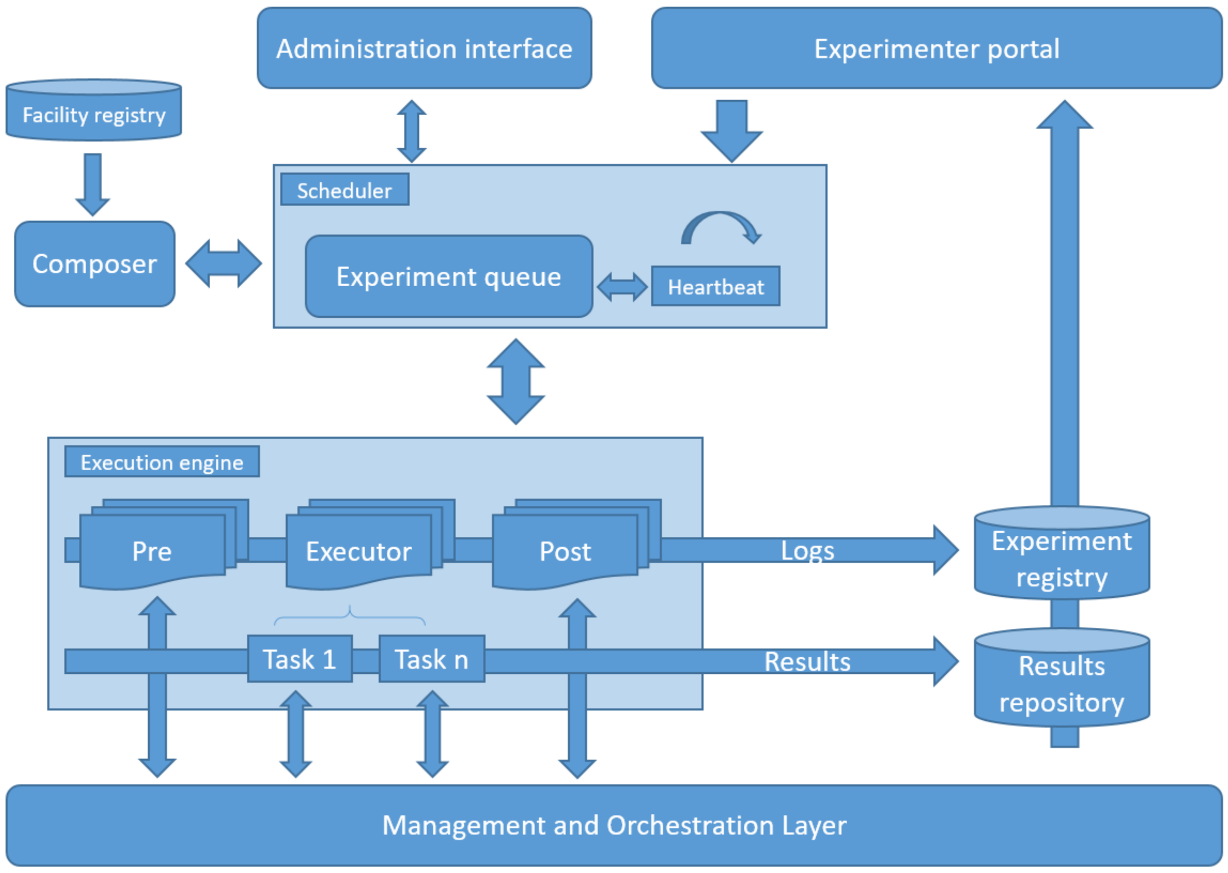


Figure 2 General architecture of the ELCM

The work-flow of the ELCM when an experiment execution is requested is as follows:

- The Scheduler creates a new Experiment Run instance. These objects contain all the information about a particular execution.
- The Scheduler requests the creation of a Platform Specific Configuration to the Composer, using the Experiment Descriptor received on the request.
- The Composer generates this configuration (including the Tasks to execute in each Executor).
- The Scheduler queues the experiment execution, starting from the Pre-Run stage. The execution is then handled by the Pre-Run Executor, which runs on a separate thread and will wait until all resources are available (among other actions).
- When the Pre-Run executor finishes (which means there are available resources), the Scheduler moves the experiment to the Run stage. Again, the real execution of the Tasks is handled by a different thread in parallel.
- The Scheduler moves the execution to the Post-Run stage once the Run stage finishes, and additional Tasks runs on the new Executor.
- When finished, the Scheduler removes the Experiment Run from the queue.

2.1. Scheduler

The Scheduler is the component that manages the execution of the experiments at the Stage level. The stages defined are:

- Pre-Run: This stage includes the experiment registration and configuration, as well as the wait loop until all the resources required by the experiment are available.
- Run: The Run stage is composed by the task required by running the experiment, including the instantiation and decommission of the resources, and the experiment loop.
- Post-Run: This stage is devoted to the management of the final results and any necessary cleanup process.

All the experiments will be kept on an internal Execution queue, where they will transition from one of these stages to the next. When an experiment enters in one of the stages the execution the execution is handled by an independent Executor, which is able to run in parallel alongside any other Executor from another experiment. These Executors will run their specific tasks one after another, until they reach completion. The scheduler will periodically check the status of every Executor from active experiments, triggering the transition to the next stage when an Executor has finished.

Due to the presence of the administration interface the ELCM has been developed as a web application, which are designed to react to external requests. This approach is not compatible with the parallel execution of experiments, which, once started, are expected to run without external intervention. Because of this, once the web application starts, a background thread (known as the Heartbeat) is generated, which periodically checks the status of the Experiment Queue, moving experiments to the next stage when required and removing already finished executions from the queue.

2.2. Execution Engine

The Execution Engine is responsible for performing the specific actions required for the execution of a specific execution Stage. These Stages are, in turn, composed by different Tasks. For every Stage a different Executor will handle the execution of the defined Tasks.

The Executors will run all the tasks one after another, in the order defined by the Composer, however, multiple Executors can run in parallel. Due to this, the ELCM is able to run any number of experiments at the same time, provided that there are enough resources in the platform for all of them.

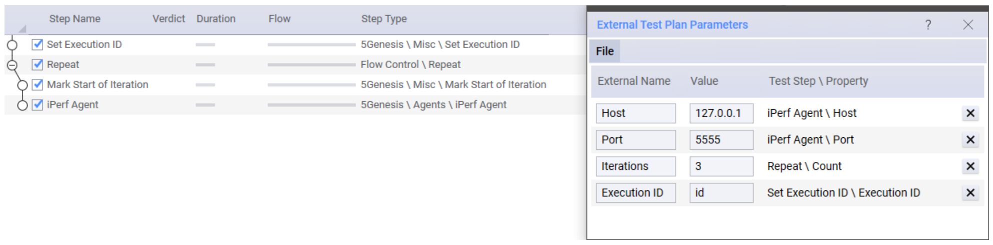
Each of the different Tasks may perform any action on other components of the platform. For example, it's possible to define a Task that calls a shell script for enabling iPerf on a remote machine or another that executes a TAP[1][5] (Test Automation Platform) TestPlan that activates a probe running on a mobile phone.

2.2.1. TAP as execution environment

TAP can be considered as another execution environment, in the same way as shell scripting, but the number of additional features included in TAP (such as the result handling capabilities or the ability to abstract different components of a Platform as Instruments) makes it ideal for certain automation tasks that are in line with the requirements of the 5Genesis Platforms. It is, however, not the only option, and is expected to be used along with other different execution engines.

In order to integrate and make effective use of TAP on the 5Genesis Platforms some considerations need to be taken into account:

- TAP Instruments must be configured beforehand: It's not possible to easily configure the settings outside of the TAP user interface (it's possible to edit the xml files that contain this configuration by hand, however, having to know the names and acceptable values of every possible setting makes this option undesirable).
- In order to retrieve the generated results in a way that allows the Platforms to perform additional analysis and storage, it's important to correctly configure the InfluxDb[6] result listener. For details about this result listener please refer to section 4.5.1 of this deliverable.
- A TAP testplan must be generated for every kind of Task that might be executed from the ELCM using the TAP GUI (again, editing the xml files is possible, but extremely complex). These Testplans may contain External Parameters that can be modified by the ELCM when executing the Testplans.
- In order to support the analysis and visualization of the results, it is necessary to tag the generated results with information about the experiment execution as well as the specific iteration (for test case implementation) in which they were generated. This functionality is supported by the InfluxDb result listener, but two special test steps must be included in the testplan in order to use this feature.



Step Name	Verdict	Duration	Flow	Step Type
Set Execution ID				5Genesis \ Misc \ Set Execution ID
Repeat				Flow Control \ Repeat
Mark Start of Iteration				5Genesis \ Misc \ Mark Start of Iteration
iPerf Agent				5Genesis \ Agents \ iPerf Agent

External Test Plan Parameters

File

External Name	Value	Test Step \ Property
Host	127.0.0.1	iPerf Agent \ Host
Port	5555	iPerf Agent \ Port
Iterations	3	Repeat \ Count
Execution ID	id	Set Execution ID \ Execution ID

Figure 3 Example TestPlan and external parameters for ELCM

Figure 3 shows an example of a testplan that includes all current the requirements for usage on 5Genesis facilities. The first step (“Set Execution ID”) will define the global execution identifier for the results. The actual value can be set using the “ExecutionID” external parameter, which means that the ELCM can modify this parameter with the correct value during the experiment execution.

The second is a default step included in TAP, however, the number of repetitions of the loop has been externalized, so that it can be modified easily with the correct number of iterations.

“Mark Start of Iteration” is a custom step that keeps track of the current iteration number and configures any compatible result listener (in particular the InfluxDb result listener) so that every generated result will be tagged with the correct iteration number. The step will increase automatically this value every time it is executed inside a loop.

The fourth step (iPerf Agent) is provided as an example of the actions that can be performed during an iteration. In this case the step will use the remote iPerf agent detailed in Deliverable D3.5 to start an iPerf client instance that will run for some time. Once this period finishes the step will recover every result generated and publish them so that they are sent to the InfluxDb database, correctly tagged, by the InfluxDb result listener. The IP address and port of the iPerf server have also been defined as external parameters (“Host” and “Port”) so that they can easily be customized by the ELCM or a user.

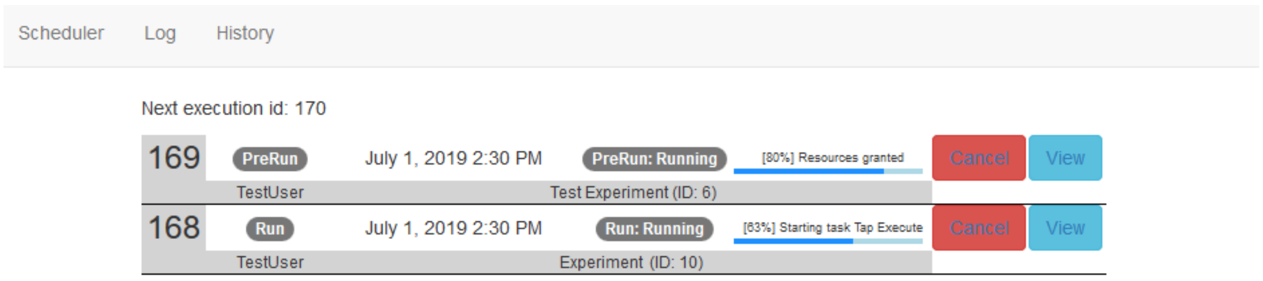
2.3. Composer

The Composer is able to create the Platform Specific Configuration of the experiments, by using the information available in the Facility Registry in conjunction with the Experiment Descriptor received along with the execution request. By using this information the Composer can generate the list of Tasks that are to be executed during the experiment run, as well as any other configuration value needed to support the execution.

2.4. Other components

2.4.1. Administration interface

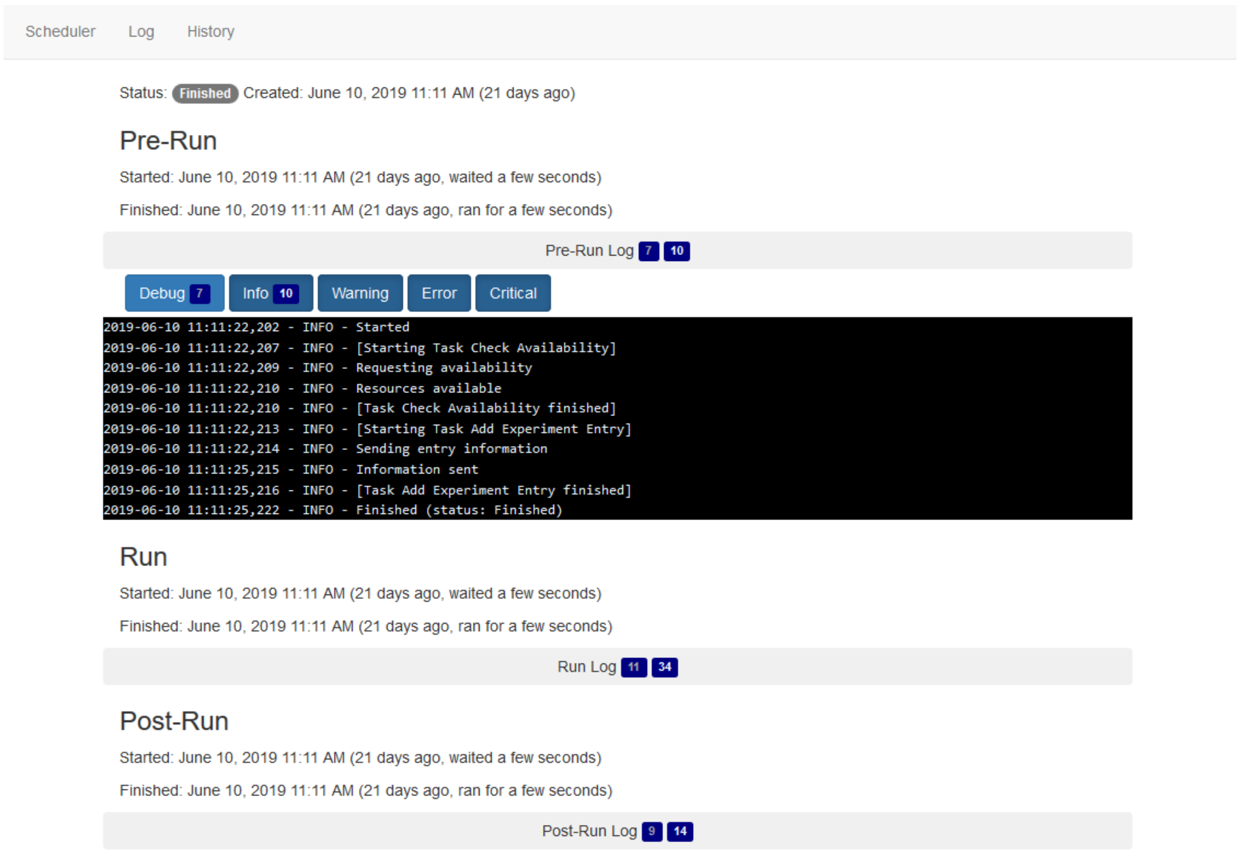
The Administration Interface is a web application developed in Flask[2] that gives a unified interface to platform administrators where they can review the execution status of active experiment run, as well as checking the logs generated by every execution, including previous ones. From this interface it’s also possible to cancel the execution of an experiment.



Next execution id: 170

169	PreRun	July 1, 2019 2:30 PM	PreRun: Running	[80%] Resources granted	
	TestUser	Test Experiment (ID: 6)			
168	Run	July 1, 2019 2:30 PM	Run: Running	[63%] Starting task Tap Execute	
	TestUser	Experiment (ID: 10)			

Figure 4 ELCM Administration Interface



Status: Finished Created: June 10, 2019 11:11 AM (21 days ago)

Pre-Run

Started: June 10, 2019 11:11 AM (21 days ago, waited a few seconds)

Finished: June 10, 2019 11:11 AM (21 days ago, ran for a few seconds)

Pre-Run Log 7 10

```
2019-06-10 11:11:22,202 - INFO - Started
2019-06-10 11:11:22,207 - INFO - [Starting Task Check Availability]
2019-06-10 11:11:22,209 - INFO - Requesting availability
2019-06-10 11:11:22,210 - INFO - Resources available
2019-06-10 11:11:22,210 - INFO - [Task Check Availability finished]
2019-06-10 11:11:22,213 - INFO - [Starting Task Add Experiment Entry]
2019-06-10 11:11:22,214 - INFO - Sending entry information
2019-06-10 11:11:25,215 - INFO - Information sent
2019-06-10 11:11:25,216 - INFO - [Task Add Experiment Entry finished]
2019-06-10 11:11:25,222 - INFO - Finished (status: Finished)
```

Run

Started: June 10, 2019 11:11 AM (21 days ago, waited a few seconds)

Finished: June 10, 2019 11:11 AM (21 days ago, ran for a few seconds)

Run Log 11 34

Post-Run

Started: June 10, 2019 11:11 AM (21 days ago, waited a few seconds)

Finished: June 10, 2019 11:11 AM (21 days ago, ran for a few seconds)

Post-Run Log 9 14

Figure 5 Log viewer

2.4.2. Grafana dashboard generator

The ELCM is able to request the generation of custom dashboards to a running Grafana[7] instance where the experimenter can review the results generated by an experiment. In order to generate the dashboard the ELCM will use the information contained in the Facility Registry, where the platform administrators can include the definition of several Grafana panels, which will be populated with the results generated by the experiment.

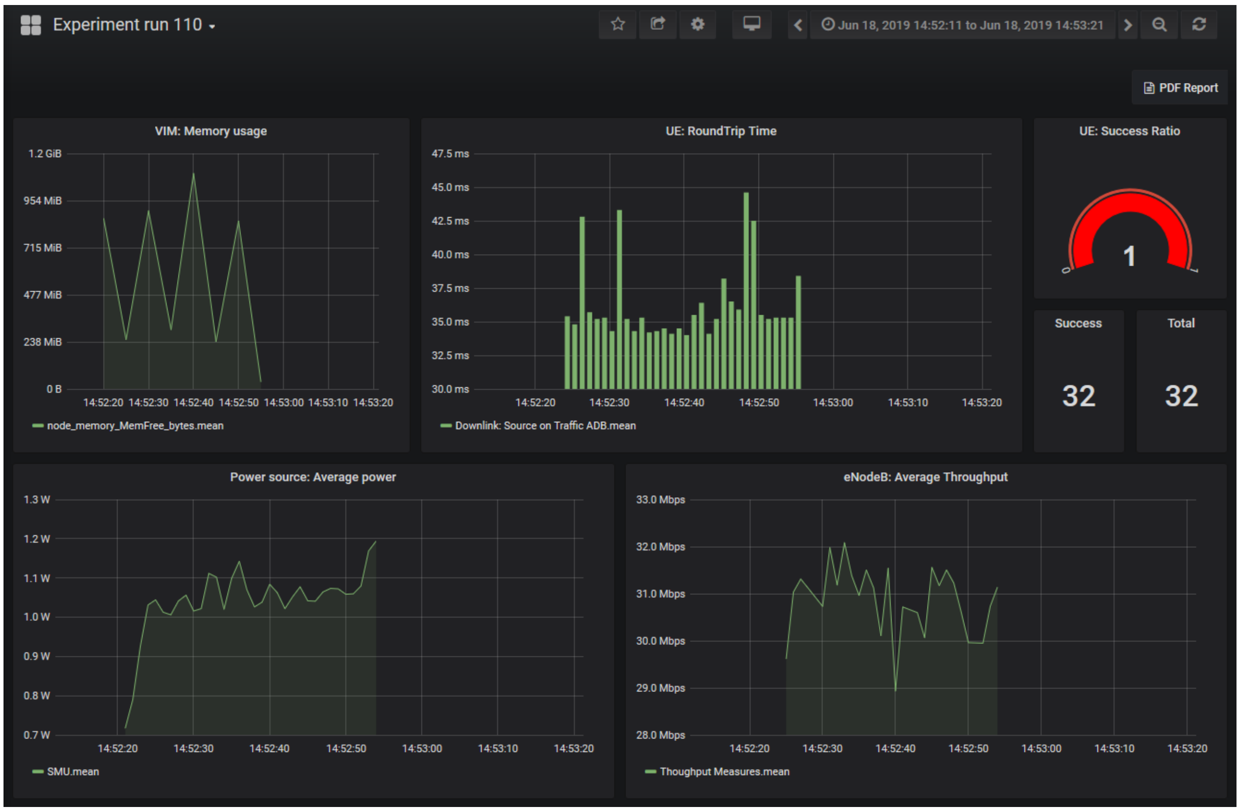

Experiment run 110
Jun 18, 2019 14:52:11 to Jun 18, 2019 14:53:21
PDF Report
VIM: Memory usage
node_memory_MemFree_bytes.mean
UE: RoundTrip Time
Downlink: Source on Traffic ADB.mean
UE: Success Ratio
1
Success
32
Total
32
Power source: Average power
SMU.mean
eNodeB: Average Throughput
Thoughput Measures.mean

Figure 6 RTT experiment dashboard generated by the ELCM

3. ELCM IMPLEMENTATION

The ELCM has been developed in Python[1] 3.7. The web application uses the Flask[2] micro-framework as foundation. Flask is an open-source framework with a large community of developers that provides all the standard functionality required in modern web applications. Other components of the ELCM make use of the standard library of Python and other open-source libraries for additional functionality.

3.1. Experiment life-cycle implementation

This section provides some additional details about the logic and implementation of the experiment life-cycle, from the reception of an execution request until the end of the experiment execution.

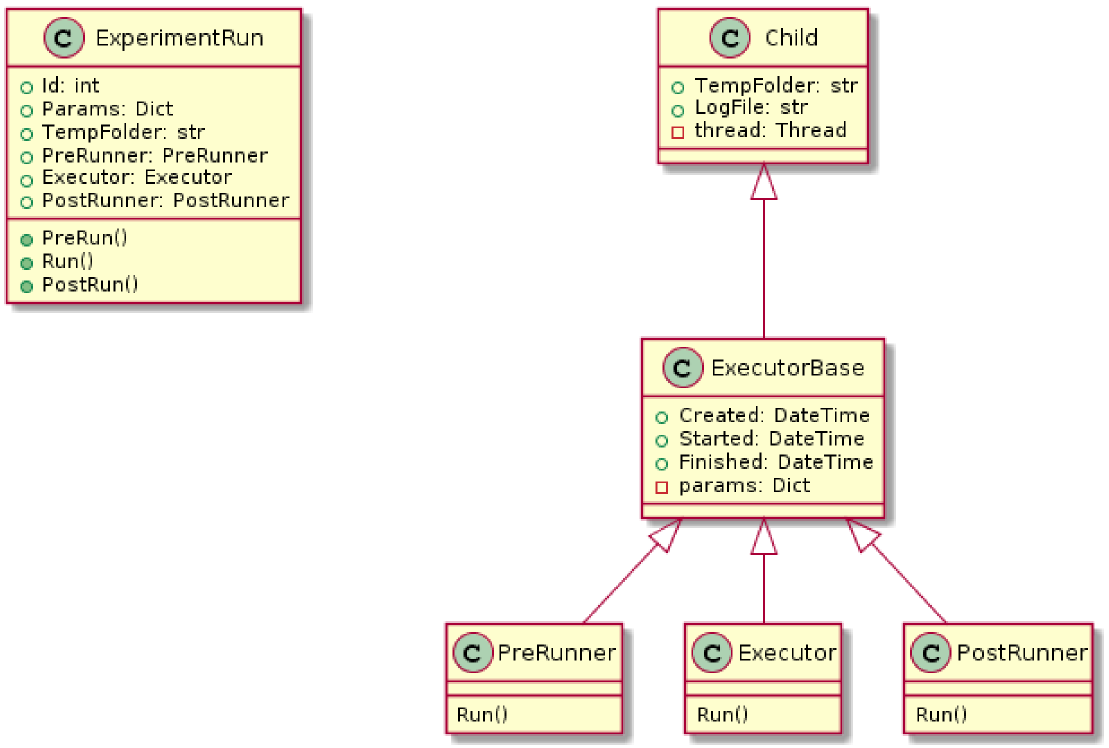


Figure 7 Main classes on the Experiment life-cycle implementation

3.1.1. The ExperimentRun class

The instances of the ExperimentRun class is responsible for storing all the current information about an experiment execution. It also contains references to the PreRunner, Executor, and PostRunner, which are the entities that handle the execution of each independent stage of the experiment.

When an experiment run execution is received, the Scheduler will create a new instance of the ExperimentRun class, identified by a unique *Id* integer value. As part of the parameters required for the creation of the ExperimentRun, is necessary to provide a dictionary that contains all the configuration values and variables required during the execution of the experiments. This dictionary will be known as the *Params* of the experiment, and will be shared with the PreRunner, Executor and PostRunner instances for communication within the different stages.

One important value that is contained in the *Params* dictionary is the Descriptor. This value (the Experiment Descriptor) contains all the information about the experiment execution, including the test cases to run, the equipment to use among others. This descriptor is used as input for the Composer, which will create the list of Task that will run on the different Executors and their configuration values.

The ExperimentRun class provides methods for starting each stage independently, however, the logic for starting the correct stage and to transition from one to the next resides in the Experiment Queue, which is handled by the Scheduler. Likewise, the execution of each stage is delegated to the different Executors.

3.1.2. The ExecutorBase and Child classes

The minimal logic for executing code in parallel threads is contained in the Child class. This class includes all the functionality for managing a separate thread where it's possible to run specific methods, as well as the logic for handling the creation and destruction of temporal folders (so that every thread has its own disk space where they can store intermediate results) and log files.

ExecutorBase, that extends the Child class, provides the extra functionality that is common to all the Executors (PreRunner, Executor and PostRunner). This includes information about when the Executor was created, when it started and finished its execution, and the list of messages that have been generated. These messages are separated from the full logs and provide a fast way for tracking the progress of the execution. For example, a new message will be generated when the Executor starts processing a new Task.

All the executors run a series of Tasks. In the case of the Pre and PostRunner this list is static and common to all experiments, though this might change in the future. The Tasks performed by the Executor is generated by the Composer depending on the test cases and UEs selected in the experiment.

3.1.3. Tasks

In the context of the Experiment Life Cycle Manager, a Task is the minimal action that must be performed in order to run an experiment, and, in general, involve delegating the execution to an external entity. For example, a Task may be used in order to execute a TAP TestPlan that will perform some measurements, or running a script through the command line for configuring some equipment. Tasks may run for as long as needed, but only one Tasks can run on a given executor at a time.

Like in the ExperimentRun class, Tasks receive a dictionary of parameters that further refine their behavior. It's also possible to conditionally run a task depending on the values contained in the parameters dictionary.

Tasks of the following types can be defined by the platform administrators for the execution of Test Cases:

Type	Description	Parameters
Message	This task will include a message in the Executor log, using the selected severity.	*Message*: Text message to add to the log. *Severity*: Severity level (Debug, Info, etc.).
CliExecute	Execute the specified command using the Command Line Interface. The output of the command will be added to the Executor log.	*Parameters*: List of parameters (including the script or executable name) to pass to the command line. *CWD*: (Current Working Directory) Folder where the command will be run.
TapExecute	Executes the specified TAP TestPlan, using the selected external parameters. TAP's output will be added to the Executor log.	*TestPlan*: Path of the TestPlan to run. *Externals*: Dictionary of TAP External Parameters that will be used.

Table 2 Main Task types supported.

3.2. Composer

The Composer is the entity that generates the Platform Specific Configuration (the set of configuration values and Tasks that need to be run in order to perform an experiment). This configuration (an instance of the PlatformConfiguration class) is generated by using the Experiment Descriptor received as part of the execution request and the contents of the Facility Registry. Additionally, the PlatformConfiguration instances also include the list of Grafana panels that will later be used by the Dashboard Generator.

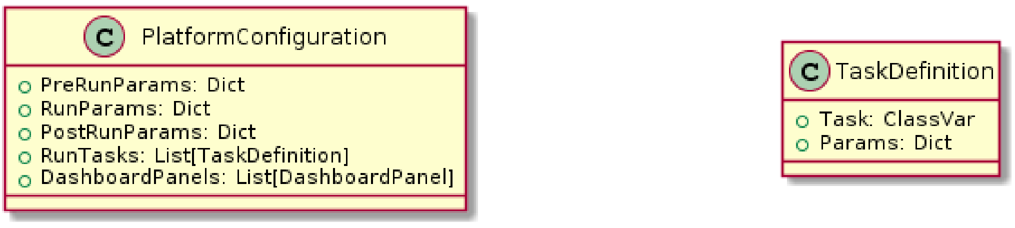


Figure 8 PlatformConfiguration and TaskDefinition classes

3.2.1. Facility Registry (facility.yml)

In the current implementation, the Facility Registry has been implemented as a YAML file that specifies the required configuration values and actions necessary for the execution of experiments in the 5Genesis platform. This file is divided in three main sections: TestCases, UEs and Dashboards. The first two sections define the set of Tasks to run and their configuration values, while the third one is used by the Dashboard Generator and will be detailed on section 2.4.2.

The TestCases and UEs section follow a similar approach: In both cases they contain a dictionary of identifiers (in the cases of TestCases these correspond to the names of the available test cases, while on the UEs they refer to the names of the different devices in the platform). Each key in these dictionaries contain a list of all the Task definitions that need to be run as part of the experiment. The following is an example of the possible contents of facility.yml

```yaml
UEs:
  GalaxyS7:
    - Order: 1
      Task: Run.Message
      Config:
        Severity: Info
        Message: Using Galaxy S7
    - Order: 2
      Task: Run.TapExecute
      Config:
        TestPlan: C:/5Genesis/startUE.TapPlan
        Externals:
          Device ID: "DEVICE_ID"
    - Order: 10
      Task: Run.TapExecute
      Config:
        TestPlan: C:/5Genesis/stopUE.TapPlan
        Externals:
          Device ID: "DEVICE_ID"

TestCases:
  RTT:
    - Order: 5
      Task: Run.TapExecute
      Config:
        TestPlan: C:/5Genesis/ping.TapPlan
        Externals:
          Experiment ID: "@{ExperimentId}"
  Throughput:
    - Order: 5
      Task: Run.TapExecute
      Config:
        TestPlan: C:/5Genesis/iperf.TapPlan
        Externals:
          Experiment ID: "@{ExperimentId}"
```

Figure 9 Example of the contents of facility.yml

3.2.2. The composition process

The process followed by the Composer in order to generate the Platform Specific Configuration consists in the following:

- As part of the Experiment Descriptor, the Composer will receive a list of Test Cases and UEs that need to be involved in the experiment execution.
- For each of the UEs selected, the Composer will add to a temporary list the information of all the tasks that belong to that particular UE. Following the example on Figure 9, the Composer would add three tasks for a Galaxy S7, with orders 1, 2 and 10.
- For each Test Case, the Composer will add their actions to the same list. Following the same example, it would execute 'iperf.TapPlan' for Throughput test cases and 'ping.TapPlan' for RTT ones.
- The composer will generate the final list of Tasks by sorting the temporary list following the 'Order' values. If multiple tasks share the same order, they will be run

with undetermined precedence. For this reason, it's important to define tasks that configure and initialize the equipment with low order values, measurement tasks in the middle and task that finalize the processes with higher ones.

This process generates the contents of the *RunTasks* list. This list contains all the necessary TaskDefinitions that will be used by the Executor, however, there is a final step before the tasks can be executed: It's possible that the parameters of a Task contains dynamic values, for example '@{ExperimentId}'. These values must be obtained during runtime, and, for this reason, the Executor performs an expansion of all the parameters in a TaskDefinition before the real Task is instantiated and executed.

3.2.3. Grafana dashboard generation

The third main section on the Facility Registry (Dashboards) is devoted to the generation of Grafana[7] dashboards where the experimenters can easily review the results generated by the experiment execution. This section follows a format similar to the TestCases section, where each key corresponds to one of the test cases supported by the platform, but in this case it contains the definition of all the Grafana panels that are to be generated. The values that can be set for each panel are detailed on Table 3.

	Parameter	Type	Description
	Type	String	Panel type. Available values are 'SingleStat' (gauges, numeric values) and 'Graph' (time series graph)
	Name (Optional)	String	Name of the panel, if not set a default name will be generated from the Measurement and Field values
	Measurement	String	Measurement name
	Field	String	Field name
	Unit (Optional)	String	Results unit
	Size	List[int]	Size of the panel (height, width)
	Position	List[int]	Panel position in the dashboard (x, y)
Graph	Lines	Boolean	True to display as line graph, False to display as bars
	Percentage	Boolean	Whether the graph represents a percentage or not
	Interval (Optional)	String	Time interval of the graph. If not set, the default Grafana interval will be used.
SingleStat	Gauge	Boolean	True to display as a gauge, false to display as a single numeric value
	MaxValue (Optional)	Float	Maximum expected value of the gauge, 100 if not set
	MinValue (Optional)	Float	Minimum expected value of the gauge, 0 if not set

Table 3 Available parameters for Grafana panel definition.

By using the information contained in this section, the Dashboard generator will automatically create a JSON description of the complete Dashboard. This description is then sent as payload to the appropriate endpoint on the Grafana Dashboard REST API, in order to trigger the generation of the final dashboard. The Grafana API will reply with the unique URL of the dashboard, which will later be sent to the 5Genesis Portal, where it will be available for the experimenter.

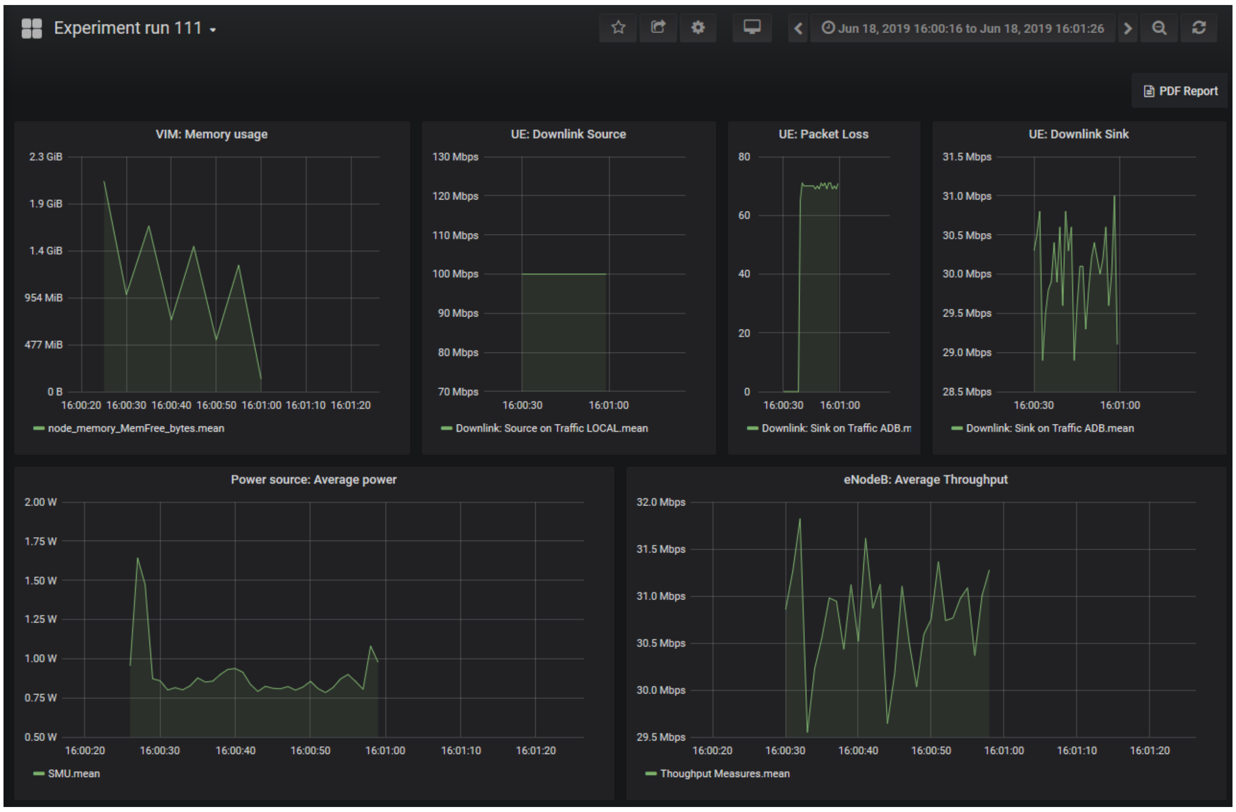

Experiment run 111
Jun 18, 2019 16:00:16 to Jun 18, 2019 16:01:26
PDF Report
VIM: Memory usage
node_memory_MemFree_bytes.mean
UE: Downlink Source
Downlink: Source on Traffic LOCAL.mean
UE: Packet Loss
Downlink: Sink on Traffic ADB.m
UE: Downlink Sink
Downlink: Sink on Traffic ADB.mean
Power source: Average power
SMU.mean
eNodeB: Average Throughput
Thoughput Measures.mean

Figure 10 Throughtput Grafana dashboard

4. ELCM Northbound and Southbound Interfaces

4.1. Portal

Since the Dispatcher element and the Open APIs are not available during Cycle 1, the ELCM does not make use of this element, communicating directly with the Portal in order to execute the experiments. This communication is performed by using a temporary REST API that offers a subset of the functionality that is projected to be available in the Open APIs.

The ELCM exposes a limited API that provides endpoints for starting the execution of experiments, as well as for retrieving the logs generated during the experiment execution. This endpoints are used by the Portal when an experimenter requests the execution of an experiment, and for displaying the generated logs to the user. A short description of the public endpoints exposed by the ELCM can be seen on Table 4 Public endpoints exposed by the ELCM temporary APITable 4, a more complete specification that also includes internal endpoints is available in Annex 1 Temporary ELCM REST API.

Endpoint	Description
/run	Starts a new experiment execution. This endpoint is accessible using the POST method, and requires a valid Experiment Descriptor as payload.
/experiment/<execution_id>/logs	Returns the contents of the logs generated by the specified execution ID in JSON format. This endpoint is available using the GET method.

Table 4 Public endpoints exposed by the ELCM temporary API

Additionally, the ELCM makes use of several endpoints exposed by the Portal, in order to send updated information about the progress of the different experiment executions, so that the user has access to a current view of the status of their experiments.

In the next cycle, this interface will be replaced by a connection with the Dispatcher element, which will handle the communication with the Portal, with other elements in the Platform and among different Platforms.

4.2. Slice manager

The ELCM is able to communicate with the Slice Manager by sending requests to the REST API exposed by this entity. For a detailed description of the Slice Manager please refer to the D3.3 deliverable. Currently, the following actions are supported by the ELCM:

- Create Slice: The ELCM is able to request the creation of a new slice by sending the required information (encoded as a JSON) to the Slice Manager. If this action has been completed successfully the Slice Manager will reply with the Slice ID required in order to perform the actions below.
- Check Slice: The ELCM can request information about the status of a specific Slice by sending the Slice ID to the appropriate endpoint on the Slice Manager. The Slice Manager will respond with a JSON dictionary that contains runtime information about the Slice.

- Check Slice Creation Time: It's possible to obtain the amount of time that the creation of a Slice has taken. This information is included in the response provided by the previous action, but this endpoint is useful for test cases that only measure the Slice Creation Time.
- Delete Slice: The ELCM can request the deletion of a Slice (identified by the Slice ID) to the Slice Manager.

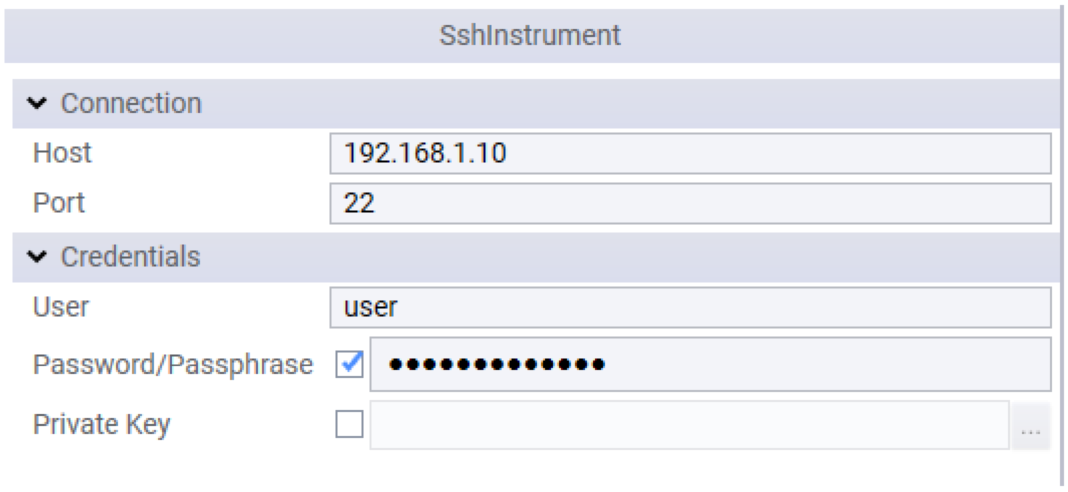
4.3. Network management system (NMS)

Interaction with the different components is performed by using different scripts and TAP[5] Testplans that are executed by the different Tasks defined for each available test cases. For this reason, the NMS does not exist as a separate entity, but is composed by the different options that are available for use to the Platform administrators.

In order to support a vast number of heterogeneous components in a generic way, UMA has developed a TAP plugin that is able to control different elements through SSH.

4.3.1. SSH TAP Plugin

The SSH TAP Plugin is composed by a TAP Instrument that contains all the configuration values of the machine that will be controlled using SSH (Figure 11), and three test steps.


SshInstrument

Connection

Host: 192.168.1.10

Port: 22

Credentials

User: user

Password/Passphrase: ✓ ••••••••••••••

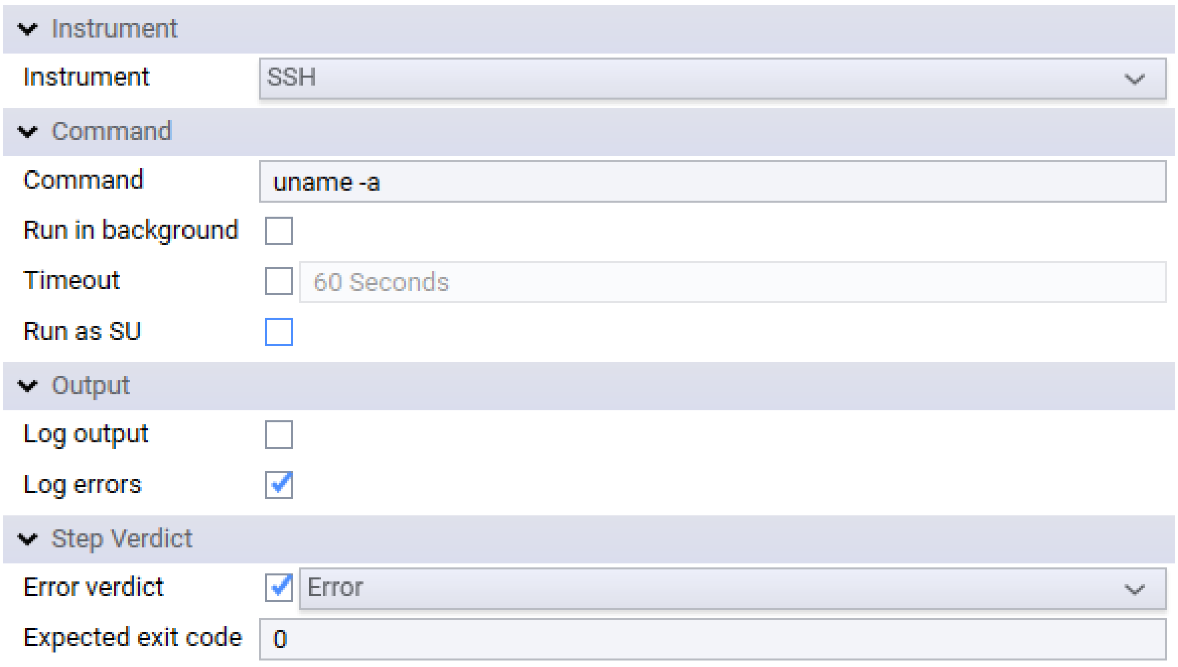
Private Key: ☐

Figure 11 SSH Instrument configuration

The SSH instrument is able to connect using user and password, or using private keys with an optional passphrase.

The following test steps are included in the plugin:

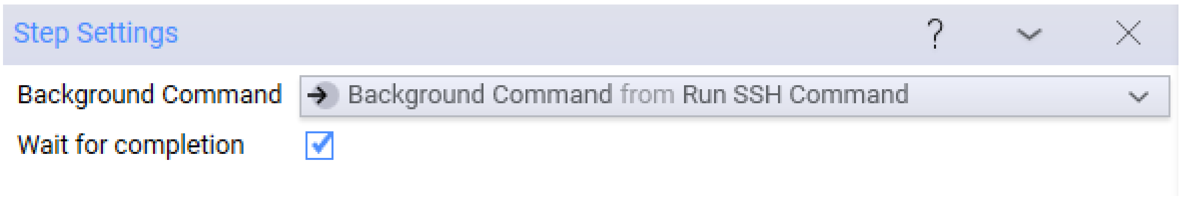
- Run SSH Command: This step is able to execute a command through SSH in the configured instrument. The step can execute this command synchronously (waiting for the command or script to end) or in the background. The step can also be configured to run the command as administrator (if the user has the required privileges in the target machine). The available settings for this step can be seen in Figure 12.



Instrument	
Instrument	SSH
Command	
Command	uname -a
Run in background	☐
Timeout	☐ 60 Seconds
Run as SU	☐
Output	
Log output	☐
Log errors	✓
Step Verdict	
Error verdict	✓ Error
Expected exit code	0

Figure 12 Run SSH Command step settings

- Retrieve Background SSH Command: This step can be used in order to synchronize the test plan execution with an SSH command that was started in the background. This step can be configured so that the command is immediately stopped if it has not been completed, or to continue waiting. If a Timeout value has been configured in the "Run SSH Command" it will be honored.

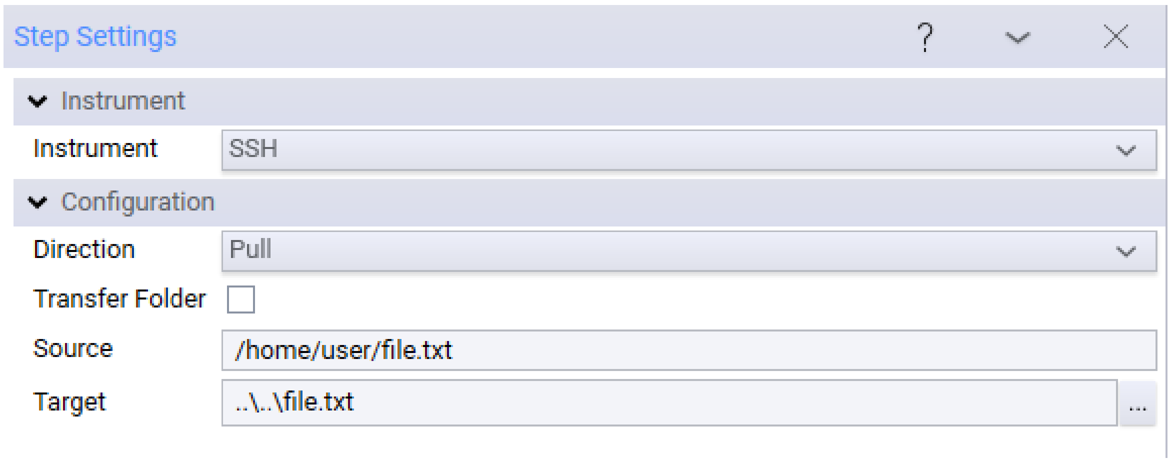


Step Settings

Background Command	Background Command from Run SSH Command
Wait for completion	✓

Figure 13 Retrieve Background SSH Command step settings

- SCP Transfer: This step can be used in order to transfer files and directories from and to the remote machine using the SCP protocol.



Instrument

Instrument	SSH

Configuration

Direction	Pull
Transfer Folder	☐
Source	/home/user/file.txt
Target	..\..\file.txt

Figure 14 SCP Transfer step settings

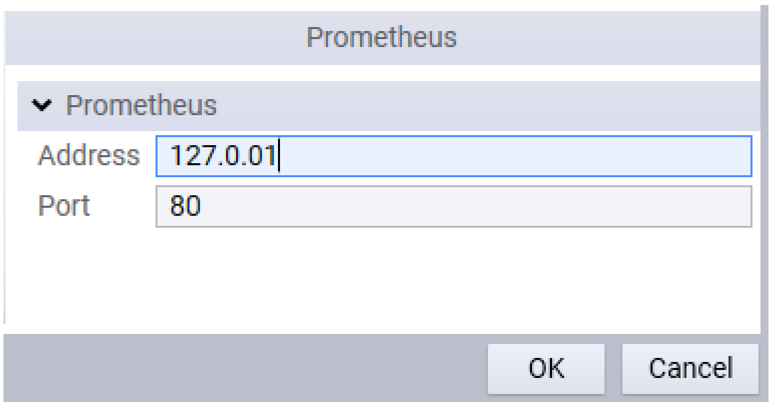
4.4. Monitoring probes

Prometheus[8] is the software selected to record the real-time metrics of the virtualize service deployed in the 5GENESIS Facilities. The Prometheus server will monitor several independent endpoints in real time, providing valuable information about the performance of the Facilities, however, a long term storage of a subset of these measurements, correlated with specific experiment executions is required in order to use this information for extracting the different KPIs defined in the experiments. For this reason, UMA has developed a TAP plugin that can

For this reason, UMA has developed a TAP Plugin that makes use of the Prometheus HTTP API in order to retrieve results from the configured instances based on a customizable PromQL query. The results obtained are published as TAP results, and thus, can be received by all of the configured TAP result listeners for further processing.

The Plugin contains two main components:

- The Prometheus Instrument, that encapsulates all the configuration values required for connecting with a specific Prometheus instance, as well as the basic functionality for sending requests and retrieving results using PromQL.

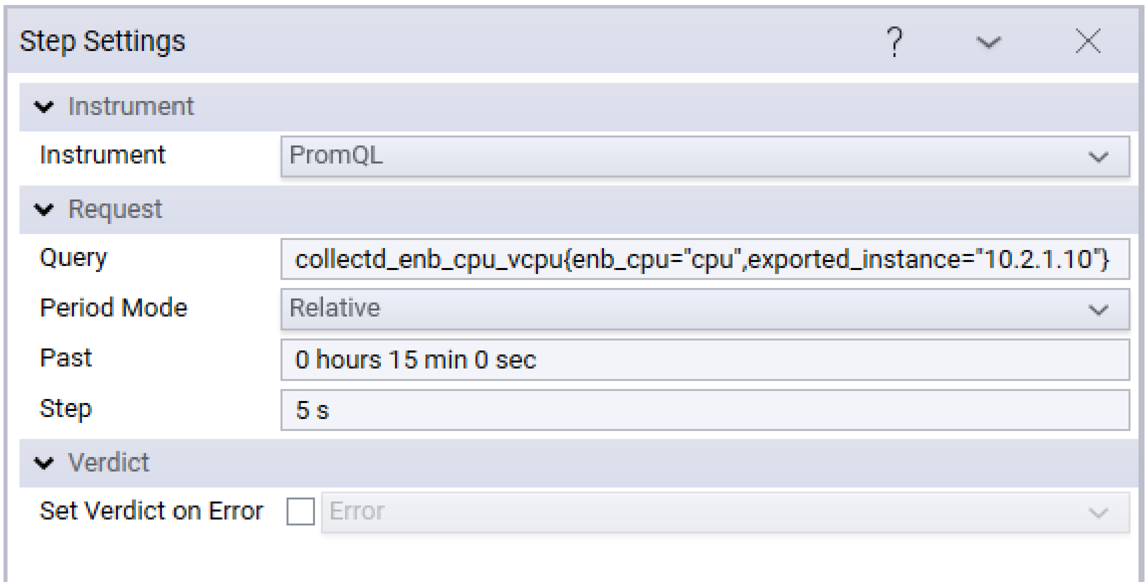


Prometheus

Address	127.0.01
Port	80

Figure 15 Prometheus TAP Instrument settings

- The “Publish Prometheus results” step that provides a way for performing requests to the Prometheus instance available to the end user.



Instrument

Instrument	PromQL

Request

Query	collectd_enb_cpu_vcpu{enb_cpu="cpu",exported_instance="10.2.1.10"}
Period Mode	Relative
Past	0 hours 15 min 0 sec
Step	5 s

Verdict

Set Verdict on Error	☐ Error

Figure 16 Prometheus TAP Step

4.5. Analytics module

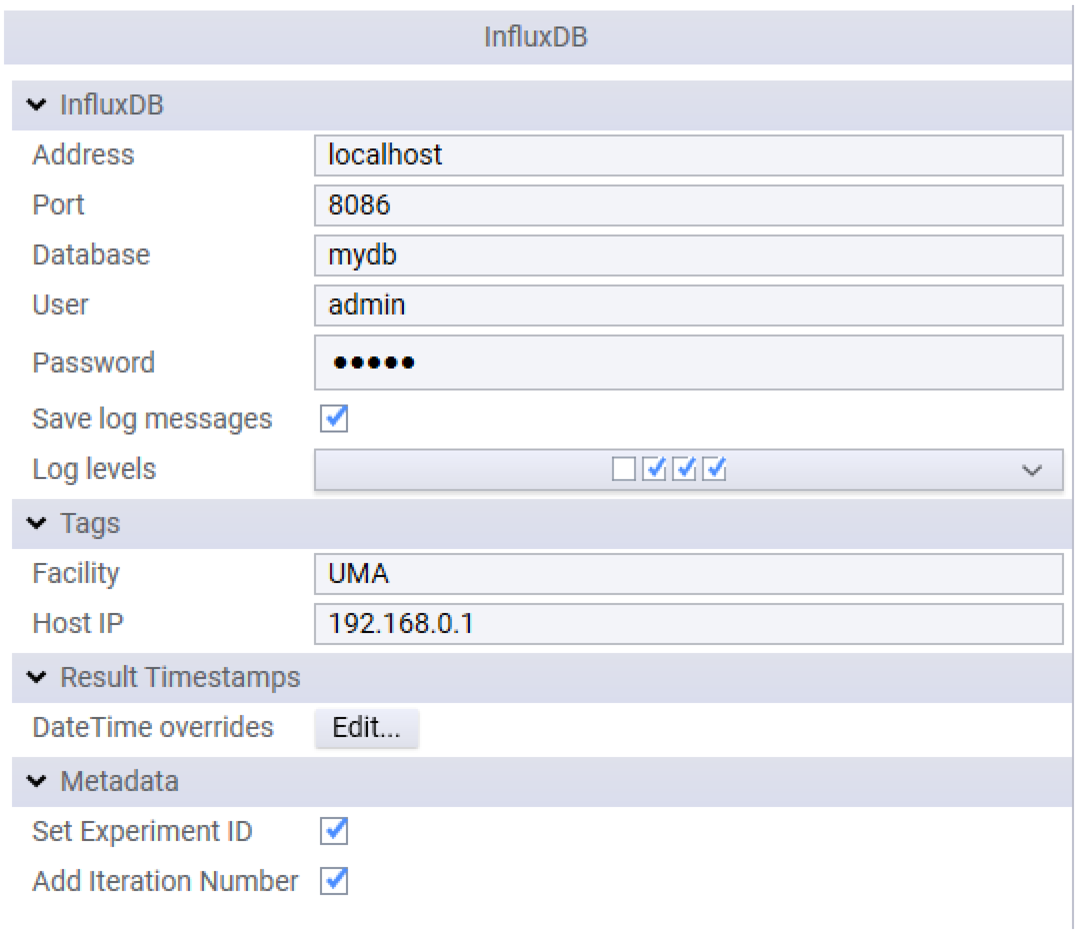
In order to integrate with the Analytics module, it is possible to send all the results generated by a TAP testplan or from tasks inside the ELCM to an InfluxDb[6] instance for later analysis. Additionally, UMA has developed a result listener that is able to organize the results in separated CSV files with additional metadata required by the Analytics module.

4.5.1. InfluxDb Result Listener

The InfluxDb result listener for TAP is the main entry point for the Analytics module. Once this result listener is configured, all the results generated by every TAP testplan will be automatically sent to the InfluxDb instance with additional metadata. The Analytics module can then retrieve these results and extract the available KPIs from them.

Additionally, the InfluxDb Result Listener can send a selection (filtered by severity) of the log messages generated during a test plan execution. This might be useful for debugging or for extracting additional information from the results, correlating these values with the events logged by TAP.

All results sent to InfluxDb must include a valid timestamp that corresponds to the moment when the measurement has been obtained. By default, the result listener will look for a field called "Timestamp" (case ignored) that should contain the POSIX timestamp (the amount of time elapsed since the midnight of January 1, 1970), however, in order to support TAP test steps that do not follow this convention, it is possible to define certain rules for obtaining the timestamp from other fields.



InfluxDB

InfluxDB

Address	localhost
Port	8086
Database	mydb
User	admin
Password	●●●●●
Save log messages	✓
Log levels	☐ ✓ ✓ ✓

Tags

Facility	UMA
Host IP	192.168.0.1

Result Timestamps

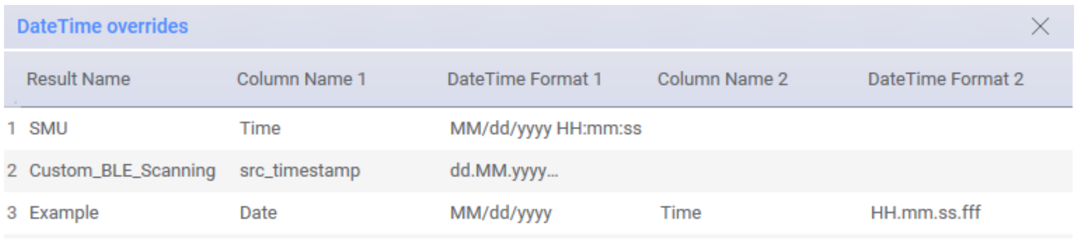
DateTime overrides	Edit...

Metadata

Set Experiment ID	✓
Add Iteration Number	✓

Figure 17 InfluxDb result listener settings

These rules can be defined by editing the "DateTime overrides" table in the result listener settings (Figure 17). This table specifies the name of the result where the rule applies, as well as the names of up to 2 fields (columns) that can be used to extract the timestamp. Additionally, two "Format" columns specify the exact format of the timestamp. Examples of some possible values for this table can be seen in Figure 18.



DateTime overrides

	Result Name	Column Name 1	DateTime Format 1	Column Name 2	DateTime Format 2
1	SMU	Time	MM/dd/yyyy HH:mm:ss		
2	Custom_BLE_Scanning	src_timestamp	dd.MM.yyyy...		
3	Example	Date	MM/dd/yyyy	Time	HH.mm.ss.fff

Figure 18 DateTime overrides

If it's not possible to obtain a valid timestamp, the result listener will display a warning message in the log and ignore the result.

In order to support the analysis and extraction of different KPIs from the results, these have to include certain metadata that can be included automatically by the InfluxDb result listener. This metadata includes a unique ID that corresponds to a specific experiment execution and (if necessary) the iteration in which the result was generated.

- The Execution ID can be specified by using the “Set Execution ID” test step. This step only has to be run once, normally at the very beginning of a test plan, before any result has been generated.
  This ID is used in order to easily obtain all the results generated by a single experiment execution (and only those).
- The Iteration number can be set by using the “Mark Start of Iteration” step. This step has to run once for every iteration, so it’s normally the first step that is included inside the loop.
  The iteration number if necessary for obtaining KPIs on Test Cases where a set of actions must be repeated for a certain number of times. This value can be used to separate the results generated by the different iterations.

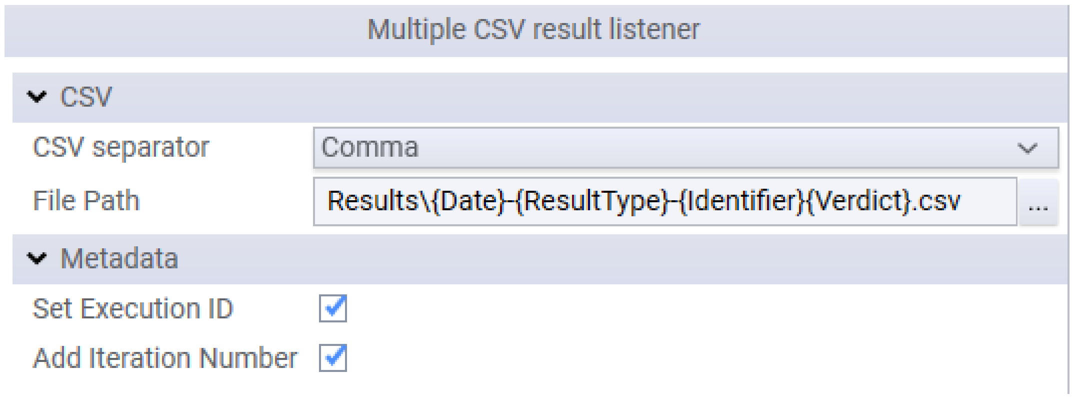
An example of a testplan where these steps are used can be seen in section 2.2.1.

4.5.2. InfluxDb Helper class

The ELCM includes the implementation of a helper class (“InfluxDb”) that is able to send arbitrary results to an InfluxDb instance. This helper can be used to send information about the performance of the ELCM or to extract KPIs from actions that are performed by the ELCM without the need of using TAP Plugins. For example, the ELCM automatically logs information about the Slice Creation Time KPI when a new slice is deployed during the execution of an experiment.

4.5.3. Multi-CSV Result Listener

In addition to the InfluxDb result listener, UMA has developed a custom CSV result listener. This result listener will save all generated results in different files based on the specific type of result, as well as include the execution ID and iteration number metadata supported by the InfluxDb result listener. This gives Platform operators an additional storage for their results, which can be easily retrieved and processed from the CSV files without the need of accessing the InfluxDb database.


Multiple CSV result listener

CSV

CSV separator	Comma
File Path	Results\{Date}-{ResultType}-{Identifier}{Verdict}.csv

Metadata

Set Execution ID	✓
Add Iteration Number	✓

Figure 19 Multi-CSV Result Listener settings

5. Release A Summary and Future Plans

All the features detailed in the previous sections, as well as the specification and design detailed in Section 2 has been performed as part of Task T3.8 during Cycle 1. This includes:

- The general design of the ELCM separated on the Scheduler, Composer and Execution Engine, in Section 2.
- The separation of Experiment Executions in three different stages composed by independent tasks (Section 2.2), and the composition process (Sections 2.3 and 3.2).
- The implementation of these design principles, along with the administration interface of the ELCM, in Section 3.
- The development of several interfaces for communicating with other components of the 5Genesis Platforms, and the creation of different TAP plugins outside of the ELCM that provide the necessary means for defining the different Tasks required during an Experiment Execution (Section 4).

Among others.

The development of the ELCM will continue in order to address any possible issue detected during the deployment and integration of this component in the different 5GENESIS Platforms, as well as for including any functionality required for the implementation of the different Test Cases available for experimentation.

5.1. Scheduling based on used resources

The current implementation of the ELCM does not take into account the resources required for each experiment. It is currently possible to initiate the execution of several experiments that will run in parallel while accessing the same resources (for example, the same UE or measurement probes). The ELCM will be updated so that different experiment executions must wait until all the required resources are released and are available for use, avoiding the generation of incorrect results or other operational issues.

5.2. Dispatcher and Open APIs integration

At the start of the development process it was decided that for Phase 1 the ELCM and the Portal would communicate directly, without the use of an independent Dispatcher. This decision was taken in order to improve the development rate of these two components (by removing the component in the middle we were able to focus on the functionality of each app, reducing the overhead of their communication to a minimum), and also in order to obtain more information about the possible requirements of the Open APIs described in Deliverable D3.7.

The direct API described on section 4.1 has been used as a base for the definition of the Open APIs that will be exposed by the Dispatcher component. Once this element is completed, the ELCM (and Portal) will be updated, removing the existing direct API and using the Dispatcher as the communication interface with the Portal.

5.3. Privacy and Security manager integration

Even though the Portal is able to manage the existence of multiple users, keeping the information about available experiments and results privately, the current implementation of the ELCM does not perform any check regarding the identity of the user that requests the execution of an experiment or access to experiment logs through the exposed REST API.

This management will be performed by the Privacy and Security manager, however, development of this component has not started during Phase 1. Once this element becomes available, the ELCM (and Portal) will be updated for using the functionality provided.

6. REFERENCES

[1] Welcome to Python.org [Online], https://www.python.org/, Retrieved 09/2019
[2] Flask [Online], https://palletsprojects.com/p/flask/, Retrieved 09/2019
[3] 5GENESIS Consortium, D2.2 Initial overall facility design and specifications: https://5genesis.eu/wp-content/uploads/2018/12/5GENESIS_D2.2_v1.0.pdf
[4] 5GENESIS Consortium, D2.3 Initial planning of tests and experimentation: https://5genesis.eu/wp-content/uploads/2019/02/5GENESIS_D2.3_v1.0.pdf
[5] Test Automation Platform (TAP) [Online], https://www.keysight.com/en/pc-2873415/test-automation-platform-tap, Retrieved 09/2019
[6] InfluxDB: Purpose-Built Open Source Time Series Database [Online], https://www.influxdata.com/, Retrieved 09/2019
[7] Grafana Labs [Online], https://grafana.com/, Retrieved 09/2019
[8] Prometheus - Monitoring system & time series database [Online], https://prometheus.io/, Retrieved 09/2019

ANNEX 1 TEMPORARY ELCM REST API

API endpoints:

Endpoint	Method	Payload	Response
/run	POST	Experiment Descriptor	{ *ExecutionId*:<int> *Success*:<bool> *Message*:<str> }
		Notes: Creates a new experiment execution using the data received in the payload, and returns the execution id or some information in case of error.	
/experiment/ <int:id>/json	GET	None	{ *Coarse*:<str: Coarse execution status (PreRun, Run, PostRun)> *Status*:<str: Current executor status (Init, Waiting, Running, Cancelled, Errored, Finished)> *PerCent*:<int> *Messages*:<List[str]: Collection of all the messages generated during the execution> }
		Notes: Internal endpoint. Used for updating the information displayed on the index view	
/experiment/ <int:id>/logs	GET	None	{ *Status*: <str: ['Success', 'Not Found'] *PreRun*:<LogInfo> *Executor*:<LogInfo> *PostRun*:<PostRun> }
		Notes: Returns the logs of the three executors of the selected experiment run, along with their severity levels (see LogInfo).	
/experiment/ nextExperimentId	GET	None	{ *NextId*:<int> }

		Notes: Internal. Used to detect new executions and update views.

Data model:

- **Experiment descriptor:**

{

Id: <int>

Name:<str>

User: <**User**>

Executions:<List[int]: Ids of the previous executions of this experiment>

Platform:<str: Platform name (from Portal configuration)>

TestCases:<List[str]: Names of the test cases to run>

UEs:<Dict[Dict[str, object]]: The keys of the dictionary are the names of the UEs to use, the dictionary in the values include extra information about the UE (currently the OS)>

Slice:<str> [Unused]

NSD:<str: File name of the NSD file>

VNF_Locations:<List[**VNF**]> [Unused]

}

- **User:**

{

Id:<int>

UserName:<str>

Email:<str>

Organization:<str>

Experiments:<List[int]: Ids of the experiments registered by the user>

}

- **VNF:**

{

Id:<int>

User:<int: Owner id>

Name:<str>

Description:<str>

VNFD:<str: file name>

Image:<str: file name>

```
        Location:<str: ‘Edge’ or ‘Data Network’>
}
```

- **LogInfo:**

```
{
        Count: { Debug: <int>, Info: <int>, Warning: <int>, Error: <int>, Critical: <int> }
        Log: <List[Tuple[str, str]]: List of pairs (<str:Severity>, <str:Message>)>
}
```

ANNEX 2 YML CONFIGURATION FILE (CONFIG.YML)

TempFolder: *Folder that will be used for storing temporary files.*

Logging:

Folder: *Folder where the logs will be stored.*

AppLevel: *Minimum severity level to display on the application log.*

LogLevel: *Minimum severity level to record on the files.*

Dispatcher: *Location where the Dispatcher (Portal) can be reached (sections 4.1 and 5.2).*

Host:

Port:

SliceManager: *Location where the Slice Manager can be reached (section 4.2).*

Host:

Port:

Flask: *Flask specific configuration (section 2.4.1).*

SECRET_KEY: *Unique secret key used for encrypting certain information*

Tap: *TAP specific configuration (section 2.2.1).*

Exe: *TAP executable name.*

Folder: *TAP installation folder.*

Results: *TAP results folder.*

EnsureClosed: *Whether or not to perform additional checks on the processes spawned by TAP*

Grafana: *Configuration values for the Grafana dashboard generation (section 2.4.2).*

Enabled:

Host:

Port:

Bearer: *Grafana API key*

ReportGenerator: *Location of the PDF report generation*

InfluxDb: InfluxDb instance configuration values (section 4.5.2).

Host:

Port:

User:

Password:

Database:

Metadata: *Additional information about the platform and metadata for InfluxDb results*

HostIp:

Facility: