



Grant Agreement No.: 731677
Call: H2020-ICT-2016-2017
Topic: ICT-13-2016
Type of action: RIA



FLAME

D3.8: Experimental Methodology for Urban-Scale Media Trials v2

Design Tools, Processes and DevOps Infrastructure

Stephen C. Phillips, Michael Boniface, Nikolay Stanchev, Simon Crowle (IT Innovation); Sebastian Robitzsch, Kay Haensge (InterDigital); Tomas Aliaga (Martel); Aloizio P. Silva (UNIVBRIS); August Betzler (i2CAT)

2019-12-20

When deploying a digital media service on the FLAME platform, you are placing it in a responsive environment that reacts (using behaviours you have defined) in real-time to metrics of interest to you. Considering this behaviour as a continuous cycle of optimization in which your target user experience (UX) is negotiated with the most efficient use of available platform resources. This report describes the design tools, processes and DevOps infrastructure for experimentation of services on a high distribution 5G infrastructure including mobile edge computing. A snapshot of the FLAME documentation available to experimenters is presented. In combination with the first version of this deliverable which focussed more on the theoretical methods, this document provides the practical steps and associated documentation necessary to execute experiments and trials in the FLAME environment.

Work package	WP3
Task	T3.2
Due date	31/10/2018
Submission date	12/01/2019
Deliverable lead	IT Innovation
Version	1.1
Authors	Stephen C. Phillips, Michael Boniface, Nikolay Stanchev, Simon Crowle (IT Innovation); Sebastian Robitzsch, Kay Haensge (InterDigital); Tomas Aliaga (Martel); Aloizio P. Silva (UNIVBRIS); August Betzler (i2CAT)
Reviewers	Marisa Catalan (i2CAT), Marc Godon (VRT)

DISCLAIMER

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731677.

This document reflects only the authors' views and the Commission is not responsible for any use that may be made of the information it contains.

Project co-funded by the European Commission in the H2020 Programme		
Nature of the deliverable:		R
Dissemination Level		
PU	Public, fully open, e.g. web	✓
CL	Classified, information as referred to in Commission Decision 2001/844/EC	
CO	Confidential to FLAME project and Commission Services	

TABLE OF CONTENTS

1	INTRODUCTION	6
2	MEDIA SERVICE DEVELOPERS GUIDE	8
2.1	Authors.....	8
2.2	Introduction	8
2.3	Media Service Nomenclature	9
2.4	Experiment and Trial Pipeline	9
2.5	Practical Steps	14
2.6	Media Service Design and Integration	15
2.7	Using FLAME Platform Services	21
2.8	Trial Preparation	22
3	SANDPIT	27
3.1	Authors.....	27
3.2	Resources	27
3.3	Access.....	27
3.4	Endpoints	28
4	BARCELONA INFRASTRUCTURE	29
4.1	Authors.....	29
4.2	Resources	29
5	UNIVERSITY OF BRISTOL INFRASTRUCTURE.....	30
5.1	Authors.....	30
5.2	Resources	30
5.3	Access to Bristol FLAME Platform	33
6	HOWTO LOG IN TO THE SANDPIT	34
6.1	Authors.....	34
6.2	Account on the server.....	34
6.3	Recommended SSH configuration	34
6.4	Accessing the interactive tutorial	36
6.5	Accessing the platform services.....	37
6.6	Accessing the user equipment (emulated clients).....	38
6.7	More on SSH tunnelling	39
7	FLAME PACKAGING AND WHOAMI	40
7.1	Authors and Reviewers	40

7.2	Prerequisites	40
7.3	Installation of Hypervisors	40
7.4	Create Virtual Instances	40
7.5	Exporting	41
7.6	Deleting	42
7.7	How to Package	42
7.8	Test Systemd Service Integration	43
7.9	Test Systemd Service Function Service Integration	43
8	TOSCA TEMPLATING IN FLAME	44
8.1	Authors.....	44
8.2	General.....	44
8.3	Versions.....	44
8.4	Getting started with TOSCA	44
8.5	Validation	44
8.6	Contributing	44
9	GETTING STARTED WITH TOSCA IN FLAME	45
9.1	The theory of TOSCA.....	45
9.2	FLAME-based TOSCA definitions.....	45
9.3	References	46
10	TOSCA RESOURCE SPECIFICATION	47
10.1	General Information	47
11	TOSCA ALERTS SPECIFICATION	50
11.1	Description	50
11.2	Metadata.....	52
11.3	Policies	52
12	TOSCA BEST PRACTICES.....	58
12.1	General Resource Specification	58
12.2	Alerts Specification	60
13	FLAME CLMC SERVICE DOCUMENTATION	65
13.1	Authors.....	65
13.2	Description	65
13.3	Notes.....	65
13.4	Alerts API Endpoints.....	66
13.5	Graph API Endpoints	68

13.6	CRUD API for service function endpoint configurations	72
14	CONCLUSIONS	77



1 INTRODUCTION

What follows is a snapshot of the documentation available to FLAME experimenters, taken in December 2018. The documentation is held in the project's GitLab repository in hypertext form and is frequently updated in line with the software itself. Many of the technical aspects are also demonstrated in the sandpit interactive tutorial.

For those who have access, the sources for the documentation below may be found at:

1. Media Service Developers Guide: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-experimenter-docs/tree/master>
2. Sandpit: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-experimenter-docs/blob/master/replicas/Sandpit.md>
3. Barcelona: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-experimenter-docs/blob/master/replicas/Barcelona.md>
4. Bristol: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-experimenter-docs/blob/master/replicas/Bristol.md>
5. HOWTO Log in to the Sandpit: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-int-infra/blob/integration/docs/howtos/login-to-infra.md>
6. FLAME Packaging and WhoAml: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-packaging/blob/master/README.md>
7. TOSCA Templating in FLAME: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-tosca/tree/master>
8. Getting Started with TOSCA in FLAME: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-tosca/blob/master/documentation/README.md>
9. TOSCA Resource Specification: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-tosca/blob/master/documentation/resource.md>
10. TOSCA Alerts Specification: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-tosca/blob/master/documentation/alerts.md>
11. TOSCA Best Practices: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-tosca/blob/master/documentation/bestpractices.md>

12. FLAME CLMC Service API: <https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-clmc/blob/master/docs/clmc-service.md>

Please note, that due to the semi-automatic conversion process from the online documentation to this Word document, not all hyperlinks will work.



2 MEDIA SERVICE DEVELOPERS GUIDE

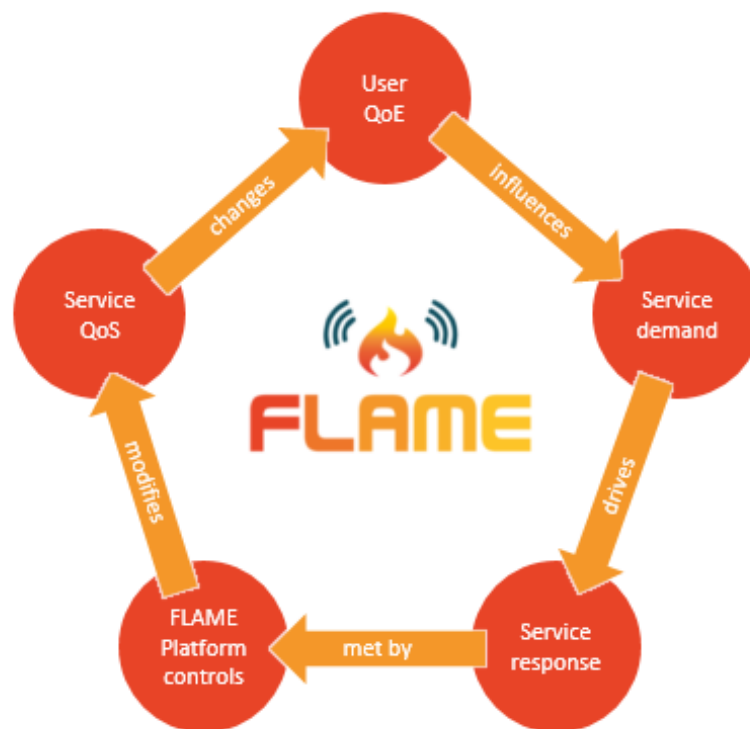
2.1 AUTHORS

Author	Organisation
Stephen C Phillips	IT Innovation
Simon Crowle	IT Innovation
Michael Boniface	IT Innovation
Sebastian Robitzsch	InterDigital Europe

© University of Southampton IT Innovation Centre and other members of the FLAME consortium.

2.2 INTRODUCTION

When deploying a digital media service on the FLAME platform, you are placing it in a responsive environment that reacts (using behaviours you have defined) in real-time to metrics of interest to you. Consider this behaviour as a continuous cycle of optimization in which your target user experience (UX) is negotiated with the most efficient use of available platform resources.



Experimentation Cycle

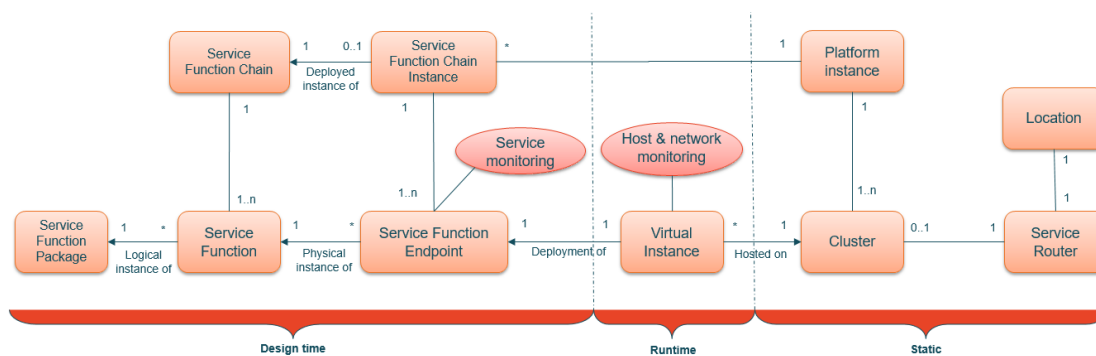
In the figure above we illustrate how user driven engagement with a media service influences the demand for service functions (SF) used to deliver it. This in turn will drive changes in the response characteristics of the SF (for example, resource usage and response time). The FLAME platform continuously monitors and evaluates metrics that reflect this demand, response and resource usage. When pre-defined conditions are met (such as a drop below a response time threshold), the FLAME

platform will intelligently execute control actions to maintain your requested quality of service, and thus sustain high quality user experiences for your clients. This process and the relationship between quality of service (QoS) and quality of experience (QoE) is explored in further detail in [D3.2 "Experimental Methodology for Urban-Scale Media Trials"](#), section 2.5.5 in particular.

Trials of media services from validation partners and from open call 3rd-parties are to validate the acceptance and viability of new interactive media services on an innovative Platform making use of new networking and management technologies and edge computing. The trials generate knowledge for the media service developer, the FLAME Platform operators and developers and the infrastructure operators.

2.3 MEDIA SERVICE NOMENCLATURE

A "media service" is an interactive service used directly by the end users to consume, create and otherwise interact with electronic media. Media services must be integrated with the FLAME Platform to make use of the Platform's capabilities (see [Develop Service Functions](#) below). To do this, some understanding of the way media services are composed and the nomenclature used is necessary.



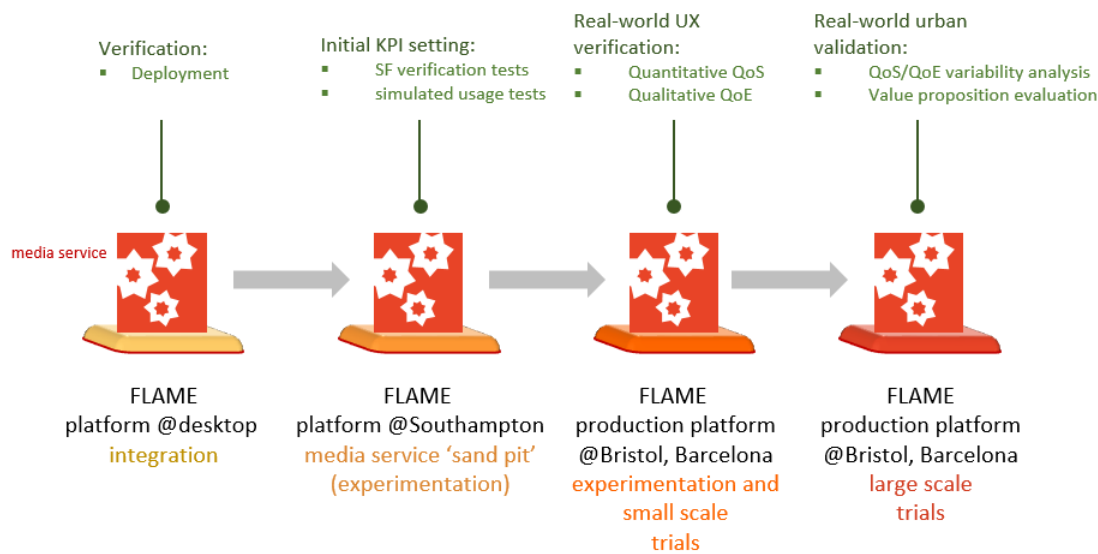
Entity Relationship Diagram

Interpreting the figure above in English: a deployed "media service" maps to a "service function chain instance" in FLAME. Taking a step back, a "service function chain instance" is a particular deployment of a "service function chain" which is described in a TOSCA resource specification document (see [Write TOSCA resource specification](#) below). A "service function chain" is a composition of one or more "service functions" (for example, a web application or a database) and a "service function" logically represents a "service function package" which is commonly a virtual machine image.

When a "service function chain" is deployed by the FLAME orchestrator, a "service function chain instance" is created which is composed of "service function endpoints". They map 1-1 onto actual virtual instances (VMs or containers). Each virtual instances is hosted on a "cluster" (found in the edge nodes of the network and centrally) and each cluster has its own "service router" and is in a "location". A deployment of the FLAME Platform commonly contains multiple clusters and will support multiple "service function chain instances".

2.4 EXPERIMENT AND TRIAL PIPELINE

FLAME provides a pipeline of increasing complexity and realism to support experiments and trials.



Pipeline

An “experiment” is an evaluation of one or more FLAME platform components, deployed in a city and executed under repeatable, controlled conditions during which time any human behaviour related to platform usage is *ideally* emulated using repeatable, machine based methods.

A “trial” is an evaluation of the use of FLAME media applications and services that use the FLAME platform under real-world conditions. Such an evaluation will be conducted with real users in a selected city environment using the FLAME platform operating using a repeatable configuration and deployment pattern. Significant engagement activities are necessary to get these users.

Experiments and trials are both sorts of tests. The word “test” is also applied to various software engineering techniques (e.g. unit test, integration test, etc.) applied in the development of media services. These software tests will be conducted using FLAME-in-a-box and the sandpit deployment in a city.

Engaging your media service with the FLAME platform and methodology will result in the progressive generation of knowledge and value for you and others in your eco-system. As described in [deliverable D3.2](#), section 2.4.3, this process is comprised of essentially three ‘tiers’ of design, development and evaluation: (1) integration, (2) experimentation and (3) user trials. Within each tier your media service is exposed to increasingly demanding tests that evaluate both its and the platform’s capabilities to deliver the user experience you expect for your end-users. FLAME knowledge generation begins almost as soon as you deploy your service for the first time. This first step is marked by an initial packaging of your service and its local deployment in the FLAME-in-a-box environment (see [FLAME-in-a-box](#)). Here you will verify that your packaged SFC, bundled as a collection of virtual machines and a TOSCA specification, deploys, initializes and serves correctly using FLAME's advanced platform services and SDN based network.

Next you will start building knowledge through experimental testing – the outcomes of which are to determine some initial key performance indicators (KPIs) for your service functions as they operate under well-defined workloads. High levels of control can be achieved using a deployment running on the FLAME experimentation platform and using simulation techniques to approximate demand. For example, you may wish to find out how your service theoretically stands up against a sudden surge of thousands of requests. Other more focussed questions that relate to the specifics of your service can

also be synthetically evaluated, such as: what the minimal compute resource requirements are to sustain 'X' number of parallel media transcodes at 'Y' frames per second? The FLAME sandpit has been designed to host larger scale media service deployments and allow you to analyse the many of the behavioural aspects of your service against the simulated behaviour of users (see [Sandpit](#)). When you execute your experiments, you get a near real-time view of how they are performing through use of the CLMC experimentation service (see [Integrate with Telegraf](#) and [Explore Monitoring Data](#)).

At the end of your experimentation it is likely you'll have collected quite a lot of data - sufficient to analyse - and will be starting to draw tentative hypotheses on the factors that impact QoS under specific deployment configurations and (simulated) levels of demand. As a result you may tweak your service function configuration with a view to meeting a particular demand or cost baseline. You can also specify some initial rules that define thresholds to trigger the addition or removal of SF resources in response to user behaviours: this is discussed further in [Define Triggers](#).

Small scale trials form the beginning of the next tier of knowledge generation. Typically, this engagement with real users begins with a small trial group perhaps initially only testing limited aspects of your media experience. The metric collection you used previously through experimentation can also be re-applied here. In addition to this, since you're working with small numbers of participants, you have the opportunity to collect detailed, qualitative data on individual's perceived QoE. This is an excellent time to test your hypothetical model of QoS to QoE. Did real-life demand and service performance align with corresponding levels in your experimentation? How far can you push resources down before UX becomes unacceptable? Are the subjective reports of UX in-line with your expectations? At what point does increasing resources available to your service function(s) no longer result in a cost-effective improvement of UX?

Your small scale trials have led to a deeper understanding of how well your digital media service meets real-world demand. Moreover, through the analysis of your qualitative data and the corresponding changes in QoS metrics, you have generated some knowledge that has empowered you to propose and trial intelligent (FLAME) platform behaviours that can, in theory, sustain a high quality UX at urban scale. Using FLAME, when you evaluate at urban scale you'll see the platform dynamically respond to demand for your service through facilitating behaviours such as net-level indirection, HTTP multi-cast response delivery and efficient routing to dynamic content caches. The knowledge you have created to optimize user experience will be played out and validated at scale and in a real-world, city context.

2.4.1 FLAME-in-a-box

[Full documentation](#)

Once a service function has been developed and packaged some aspects can be tested using FLAME-in-a-box.

FLAME-in-a-Box is a VirtualBox-based mini-FLAME platform which allows the testing of:

1. SFC orchestration templates
2. SF provisioning
3. Basic communication tests of deployed SFEs

All instances that come with the virtual machine image (OVA) are configured to run on a Windows 10 machine with 4 cores and 8GB of RAM.

See [Validate TOSCA Resource Specification and Packaging](#) below for more information on obtaining this component.

2.4.2 Sandpit

IT Innovation provide a server (givry.it-innovation.soton.ac.uk) which has the FLAME platform deployed in virtual machines to emulate a replica with multiple small data centres such as Bristol. Media services should be tested in the sandpit before going to a physical replica. Monitoring data collected by the CLMC may be explored and alerts adjusted. The adaptation of the media service to those alerts may be experimented with.

2.4.3 Small trial

A “small trial” may be around 10 people on site over some short time and might not include all aspects of the experiment (could be a focus on part of the experience).

Your small scale FLAME trials serve to provide a number of important outcomes:

- Validation of the technical deployment and operation of your media service
- Generate observations of real use that will drive usability improvements
- Capture of a selection of QoE data that can be analysed alongside QoS metrics

If well-planned, small scale trials will generate critical data (and later, knowledge) that will significantly improve the quality of your FLAME based media service as well as removing or mitigating risks when you come to trial at urban scale. Journalising your trial will capture important, real-world considerations that will help you ‘iron-out’ the experimentation process (as well as providing important FLAME-based ‘know-how’ to others). Also note some of the limitations of the small-scale trial: limited participant numbers are unlikely to push the performance limits of your service; your findings are most likely indicative and not generalizable; capturing and analysing ‘close’, qualitative observations often demands significant time resources from the experimenter.

The close observations are tractable in small-scale trials and offer you the opportunity to get a richer picture of real user experience of your media service. Observational methods are varied, but well documented – the reader is directed to [well-known literature on this subject](#). It is likely that your users will uncover assumptions that they (and you) have made about the intended experience and make suggestions for improvements. They may also express a wide range of informal, qualitative responses during and after the trial: including descriptions of emotional responses and attitudes related to current and future use. Careful analysis of this data alongside the quantitative QoS metrics captured during your trial can provide guidance to drive improvements – and more focussed, experimental questions.

Small scale user trials are typified by:

- A small participant cohort (5-10 users)
- ‘Horizontal’ or ‘vertical’ evaluation scope
- Naturalistic scenarios
- Observation and reflection process

You should take care to prepare your participants ahead of the trial so that they understand the activity in which they are about to take part. Many users will need a naturalistic story or scenario to contextualise the activity and anticipated interactions with your media service. The most effective method of preparation is a group-based, face-to-face meeting ahead of the trial where the scenario can be presented and discussed. Since your time with these participants will be limited, you should prioritise those aspects of your evaluation most important to you. For example, if you want to understand the overall flow of your users' UX then you may choose to adopt a 'horizontal' evaluation in which each user is superficially exposed to all aspects of the interactive experience. Alternatively, you may wish to consider 'key' aspects of interaction in more depth, in which case a 'vertical' exploration of just one or two aspects of the whole may be the focus of your scenarios.

Part of your small scale trial should make time for reflection on the experience by your participants. This moment might be sited at the end of the trial, or at regular intervals if this is appropriate. At its most basic, you should invite your participants to respond to a simple questionnaire. However, greater insights can be garnered by actively reviewing experiences with your participants – for example, by discussing particular observations you have made and asking further questions to clarify behaviours.

2.4.4 Large trial

A "large trial" would be around 50 people on site, free to use the service as they like.

Primary outcomes for a FLAME urban scale trial are:

- Verified, stable deployment and provision of your media service at scale
- Demonstrable, dynamic management of your media service using FLAME platform behaviours
- Empirically supported knowledge supporting the benefits of the FLAME platform

Successive experimentation and small-scale trials have brought you to a position where the performance of your media service is at least partially understood and the potential impact on UX when running under particular conditions has been hypothesized. Based on this hypothesis, you will have specified FLAME platform behaviours that will respond to changes in KPIs (generated by your service) that seek to optimize UX and platform resources. The data you collect from an urban scale trial will provide evidence (for or against) the efficacy of the integration of your media service and the FLAME platform with respect to creating impact for future media internet technology.

Urban scale trials have a wider reach, in many senses, when compared with the small scale approach described above. This much broader scope implies:

- Qualitative enquiry and analysis methods at scale are not possible
- User participation is open-ended
- The evaluation time-frame is significantly longer
- Data processing and analysis is more challenging

Widening access to your media service to large groups of the public or an organisation has advantages and drawbacks. Certainly, involving large numbers of participants increases the ecological validity of any knowledge you may capture and can also provide evidence that help support a case for later commercial exploitation. Perhaps one of the greatest challenges you will face as an experimenter when

conducting this work is understanding a complex data set that may be generated during unpredictable periods. Why? Unlike small-scale evaluations, you will not have control over when (or indeed which) participants will take part in the trial or have fine-grained control over their experience of it. This is not to say you will have no control: it would be easily possible to adopt [A/B type methods](#) for the purposes of comparison, if your service is appropriately designed.

2.5 PRACTICAL STEPS

We can divide the required work up into practical stages, the tasks required to prepare for:

- Sandpit
- Experiments
- Small trial
- Large trial

Many of the tasks are shared between stages so each one is described below just once but with reference to the stages as necessary.

Sandpit	Experiments	Small trial	Large trial
Do interactive tutorial	Adjust TOSCA spec	Obtain ethical approval (local / EIB)	Publicity
Design for the replicator	Define triggers	Register with DPA	Engage more participants
Develop SF	Define objectives	Sign data sharing agreement	Obtain consent
(Unit) test SF	Define test scenario	Prepare participant info sheet	Anticipate issues with longer schedule
Integrate with Telegraf	Define metrics of interest	Prepare consent form	Prepare for BYOD
Integrate with WHOAMI	Agree schedule	Engage participants	Define triggers
Integrate with FMS	Agree support	Obtain consent	Define objectives
Test integration	Run experiments	Write data management plan	Define test scenario
Package SF	Validate service	Prepare mobile devices	Define metrics of interest
Write TOSCA spec	Validate usability	Adjust TOSCA spec	Agree schedule
Validate TOSCA spec	Collect data for reuse	Define triggers	Agree support
Define triggers	Explore data	Define objectives	Run large trial
Define objectives	Disseminate	Define test scenario	Observe and record
Define test scenario		Define metrics of interest	Debrief online
Define metrics of interest		Agree schedule	Explore data
Agree schedule		Agree support	Disseminate

Agree support		Run small trial	
Use sandpit		Observe and record	
Test integration		Debrief in person	
Explore data		Explore data	
		Disseminate	

The tasks are not in a strict time order, but those above the bold line (e.g. "**Use sandpit**") must be done before the emboldened action and those below are done afterwards.

As many tasks are repeated across stages, the rest of the document groups these tasks into:

- [Media Service Design and Integration](#)
- [Using FLAME Platform Services](#)
- [Trial Preparation](#)

2.6 MEDIA SERVICE DESIGN AND INTEGRATION

A media service must be designed (or adapted) for the FLAME platform and integrated with various platform services.

The adaptation and the integration both depend on what the objectives are for an experiment. You must consider these questions:

- What knowledge you hope to gain through the experiment?
- What knowledge will the FLAME consortium (platform and infrastructure providers) gain through the experiment?
- What are the KPIs for your experiment? What does "success" mean?
- Is it improving the quality of experience (QoE) of the users? Is it reduction in cost? If so, how is that measured?
- What factors contribute to the KPIs? For instance, QoE may be improved by reducing latency. What quality of service (QoS) metrics are important?
- How is the experiment different to deploying the media service in the cloud (such as Amazon)?

The answers to these questions will influence the architecture of the media service (where should different services and data be placed?), the data to be collected through the CLMC and through other routes and the control processes to be implemented. Clearly defining and justifying what is being measured and controlled is core to the contribution of knowledge in the FLAME project. It is insufficient to simply claim "it works".

2.6.1 Do the Interactive Tutorial

The interactive online tutorial lets you do the following steps for real in the sandpit:

- create a TOSCA resource specification;
- submit the TOSCA resource specification to the orchestrator in the sandpit to deploy a pre-built service function package;
- monitor the status of the orchestrator deployment;
- access and explore the Chronograf monitoring data;
- create a TOSCA alerts configuration;
- submit the alerts configuration to the CLMC;
- simulate load on the deployed service and see additional instances deployed.

Before running the tutorial, you must configure your computer to be able to access the tutorial and the FLAME platform services, both of which are hosted on the sandpit server at IT Innovation but are not directly exposed to the Internet. This means setting up two SSH tunnels. Full instructions can be found in the [FLAME sandpit access document](#).

Once the SSH tunnels are in place, the tutorial is accessed via a web browser on:

- <http://localhost:9000/tutorial.html>

These links are also used:

- <http://localhost:9001/orchestrator/sforch> to access the Orchestrator API
- <http://localhost:9001/orchestrator/sfemc> to access the SFEMC API
- <http://localhost:9001/clmc/chronograf> to access the Chronograf web page

Full, step-by-step instructions can be found in the tutorial itself.

2.6.2 Design for the Replicator Environment

FLAME trials exploring the acceptance and viability of new interactive media services on an innovative Platform making use of new networking and management technologies and edge computing. We encourage developers to "think big" and imagine a world where such platforms are widespread but realistically the FLAME testbeds (sandpit and replicas in cities etc) are limited. Therefore, any trial must be designed to fit into and make best use of the environment in which it is to run. This means taking account of what edge computing hardware is available and the physical space itself.

More information about each replica can be found in separate documents:

- [Bristol](#)
- [Barcelona](#)

2.6.3 Develop Service Functions

When developing or adapting service functions for FLAME you need to consider:

- service functions for serving data and service functions for processing data;

- the resource requirements (CPU, disc, memory) and placement of each service function, bearing in mind the [replicator environment](#);
- the communication between service functions;
- integration of service functions with FLAME platform services (see [integration with Telegraf](#) and [integration with WhoAml](#)).

The architecture of your adapted media service should depend on your KPIs. For instance:

- if the latency of user access to data is important then the architecture should place replicas of the relevant data at the edge of the network;
- if the scenario involves many users uploading video content and the KPI is to reduce network traffic then processing elements at the edge of the network could be used to discard low quality or otherwise unwanted video;
- if the scenario is to stream video to many users and the KPI is the cost of the network traffic then HTTP (e.g. MPEG-DASH) should be used for the streaming to take advantage of the platform's coincidental multicast capability.

Information on how the platform technical capabilities relate to scenarios can be found in [D3.1 "FMI Vision, Use cases and Scenarios v1"](#) section 7.

2.6.4 Integrate with Telegraf

To make use of the FLAME platform monitoring and alert functions, each service function must report monitoring data to the CLMC. This reporting is done through a [Telegraf agent](#), part of "[TickStack](#)".

Integration with Telegraf can be done entirely independently from the CLMC, see [Getting started with Telegraf](#). The Telegraf agent can send monitoring data to a variety of destinations. In the CLMC (and TickStack) the data is sent to an [InfluxDB](#) but alternatives such as Kafka can be used in development if useful (see [telegraf output plugins](#)).

There are [many plugins for Telegraf to gather data](#): these are called "input plugins". Experimenters should consider gathering data at all levels:

- from the (virtual) host, such as disc, memory, network and CPU usage;
- from standard plugins for e.g. application containers, databases, etc;
- from their specific software to gather unique metrics important to their application.

Several plugins are automatically configured in a Service Function package. The standard configuration to gather metrics about the (virtual) host is:

```
[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false
  report_active = false
[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs", "devfs"]
[[inputs.diskio]]
```

```

    devices = ["sda1"]
[[inputs.kernel]]
[[inputs.mem]]
[[inputs.processes]]
[[inputs.swap]]
[[inputs.system]]
[[inputs.net]]
[[inputs.netstat]]

```

To get custom data specific to an experiment into Telegraf, you can develop a Telegraf plugin yourself (using Go). Two alternatives are to use the [HTTP listener](#) plugin or the [Logparser](#) plugin.

The HTTP Listener plugin opens a TCP port on the machine and listens for HTTP messages with the data in: thus it is easy to integrate with shell scripts (using curl) or with custom software using standard HTTP APIs.

The Logparser plugin is configured to read a log file (streamed) and search for patterns in the log messages. In this way it extracts pieces of data which are then reformatted to be sent on to Telegraf. This was of integration only loosely couples the experiment software to the monitoring system as all that is necessary is to output additional log messages.

Configuration files for Telegraf plugins should be placed in your service function package in /etc/telegraf.d/.

2.6.5 Integrate with WhoAml

The WhoAml API provides the information required for the endpoint to understand which service function chain it is a part of and other meta data. The API is automatically called when a deployed service function endpoint boots. Much of this data is automatically used by the Telegraf agent to contextualise the monitoring data.

For the media service developer, the WhoAml API is of importance if there are multiple service functions in the service function chain as the information must be used so that one SF can know the FQDN of another SF. The packaging process adds in an automated call to the API so a media service developer does not need to be concerned with that.

The results of the automated API call are placed into environment variables which are included in every user's profile:

- WHOAMI_SFC: a string holding the name of the service function chain, e.g. itinnov-chain
- WHOAMI_SF_CI: a string holding the name of the service function chain instance, e.g. itinnov-chain_1
- WHOAMI_SF: a string holding the name of the packaged service function, e.g. frontend
- WHOAMI_CLUSTER: a string holding a platform-wide unique identifier for the cluster/location name, e.g. 20-sr-cluster1-cluster
- WHOAMI_SF_IDS: a comma-separated list of strings holding the FQDNs which the endpoint serves, e.g. frontend.itinnov-chain.ict-flame.eu, webservice.itinnov-chain.ict-flame.eu

- WHOAMI_SFE: a unique identifier for the endpoint, e.g. `frontend.itinnov-chain.ict-flame.eu`, `webservice.itinnov-chain.ict-flame.eu-172.90.4.64`

For example, in the case of a Service Function Chain comprising a "frontend" (e.g. a web application) and a "backend" (e.g. a database), the frontend would need to know the FQDN of the backend in order to connect to the database. It is the TOSCA resource specification which defines the FQDNs used by the instances of the packaged service functions. If the TOSCA resource specification author decided that the SFs were to be called `webapp.myservice.ict-flame.eu` and `database.myservice.ict-flame.eu` then the web application needs to look at the `WHOAMI_SFIDS` environment variable to get its own FQDN which in this case is just `webapp.myservice.ict-flame.eu`. The web application then needs to extract the domain from its FQDN (`myservice.ict-flame.eu`) and prepend `database` to construct the FQDN of the backend. If the frontend had multiple FQDNs then the appropriate one would need to be selected somehow before performing this procedure.

2.6.6 Integrate with FMS

The FLAME project provide some Foundation Media Services (FMS) which provide functions potentially of use to many experiments. It is not mandatory to use the FMS but they may be useful:

- storage: The storage service allows users to put and retrieve media content through the HTTP protocol.
- quality: The media quality service provides data about media content previously stored in the storage service.
- metabase: The metabase service provides a centralized database of media content and their characteristics (meta-data).
- streaming: The streaming service streams adaptive content to be delivered to the user.
- transcoding: The transcoding service transcodes and transrates video.

2.6.7 Package Service Function

Full documentation

Before deploying into the FLAME platform, a service function must be made into a package. FLAME uses KVM and LXD as the hypervisors on each cluster so a package can be of either sort. The packaging tool supports both. Unless there is a particular need for strong isolation, LXD is recommended.

The packaging of a FLAME service must be conducted outside the FLAME platform. As the resulting images are given to the FLAME orchestrator the underlying host environment must be identical to the one used for the clusters of the FLAME platform.

The packaging process comprises 3 steps, all supported by the tool:

- Create a base image for the service function (which includes various supporting functions and services).
- Add your service function code and any special configuration such as Telegraf plugins.
- Export the image.

2.6.8 Write TOSCA resource specification

[Full FLAME TOSCA documentation](#)

[Full FLAME TOSCA resource specification documentation](#)

[TOSCA resource specification examples](#)

FLAME uses two TOSCA files:

1. The resource specification (this section) which defines the resources required by the experiment's service function chain (used by the orchestrator and the CLMC).
2. The alerts specification (below) which defines alerts (which can cause actions) based on values of received monitoring data (used by the CLMC).

The two files must be consistent with each other (the CLMC checks for this).

The resource specification defines what service function should be deployed where and in what initial state (e.g. placed, booted, connected). The state changes to make (boot, shutdown, etc) in response to the alerts are also specified.

2.6.9 Validate TOSCA Resource Specification and Packaging

[Full documentation](#)

Once a service function has been developed and packaged some aspects can be tested using FLAME-in-a-box which also provides experience of the FLAME orchestrator and Service Function Endpoint Management and Control (SFEMC).

FLAME-in-a-Box is a VirtualBox-based mini-FLAME platform which allows the testing of:

1. SFC orchestration templates
2. SF provisioning
3. Basic communication tests of deployed SFEs

All instances that come with the virtual machine image (OVA) are configured to run on a Windows 10 machine with 4 cores and 8GB of RAM.

To obtain FLAME-in-a-box you must:

1. Sign a licence agreement with InterDigital (IDE) by raising an issue.
2. Obtain an account on the sandpit from IT Innovation (see [Use the Sandpit](#)) by raising an issue.
3. Download the OVA file using `scp givry.it-innovation.soton.ac.uk:/var/flame/sandbucket/latest flame-in-a-box.tar.gz` (18GB file)

The downloaded file needs to be unarchived using e.g. `tar xvfz` and inside you will find the OVA file and documentation. The documentation describes how to spin up the necessary virtual machines using

[Oracle VirtualBox](#) and how to use the web interfaces of the orchestrator and SFEMC to deploy a simple service function chain.

2.6.10 Define Triggers

[Full FLAME TOSCA documentation](#)

[Full FLAME TOSCA alerts specification documentation](#)

[TOSCA alerts specification example](#)

As the CLMC makes use of TickStack it includes the [Kapacitor](#) component which is a real-time streaming data processing engine. It can be configured to process stream and batch data from the InfluxDB database (where the monitoring data is stored) and create alerts based on pre-configured thresholds or events in the data. Kapacitor integrates with many other systems and in FLAME we provide a simple way to configure Kapacitor so that it integrates with FLAME's SFEMC so that, based on the monitoring data, service function endpoints may be started and stopped. The alerts specification also makes it easy to send HTTP messages to the experiment's service functions to enable custom actions specific to an experiment. This alerts specification wraps some of the Kapacitor functionality for the common FLAME use cases but the CLMC also provides full access to the underlying Kapacitor API in case it is needed.

Alerts are configured through a YAML-based TOSCA-compliant document according to the TOSCA simple profile. This document is passed to the CLMC service (along with the TOSCA resource specification), which parses and validates the document and checks its consistency with the resource specification. Subsequently, the CLMC service creates and activates the alerts within Kapacitor, then registers the HTTP alert handlers specified in the document.

Three types of alerts are available:

- **threshold** - A threshold event type is an alert in which Kapacitor queries InfluxDB on specific metric in a given period of time by using a query function such as *mean*, *median*, *mode*, etc. The value is then compared against a given threshold. If the result of the comparison operation is true, an alert is triggered.
- **relative** - A relative event type is an alert in which Kapacitor computes the difference between the current aggregated value of a metric and the aggregated value reported a given period of time ago. The difference between the current and the past value is then compared against a given threshold. If the result of the comparison operation is true, an alert is triggered.
- **deadman** - A deadman event type is an alert in which Kapacitor computes the number of reported points in a measurement for a given period of time. This number is then compared to a given threshold value. If less or equal number of points have been reported (in comparison with the threshold value), an alert is triggered.

2.7 USING FLAME PLATFORM SERVICES

FLAME Platform Services are available in FLAME-in-a-box (limited), the Sandpit and at the physical replicas. The APIs are the same in each case, but there is some variety in access methods and endpoint locations:

- [Sandpit](#)
- [Bristol](#)
- [Barcelona](#)

2.7.1 Cross Layer Management and Control

[Full documentation](#)

[CLMC Service API](#)

The CLMC is built on top of TickStack which comprises the [InfluxDB](#) time-series database, the [Chronograf](#) time-series web dashboard, [Kapacitor](#) for processing data in InfluxDB and sending alerts based on the data, and [Neo4J](#) to providing graphs (i.e. nodes and edges) of the system topology. [Telegraf](#) is used for sending data to InfluxDB.

The Neo4J, InfluxDB, Chronograf and Kapacitor APIs are all available to be used and the CLMC wraps some features to make them easier to use and provides some additional functionality.

2.7.2 Orchestrator

The FLAME platform orchestrator has two endpoints:

- `/orchestrator/sforch` to access the Orchestrator web page and API
- `/orchestrator/sfemc` to access the Service Function Endpoint Management and Control (SFEMC) web page and API

The orchestrator endpoint provides a facility to upload a TOSCA resource specification for validation and deployment and provides a view on what service function chains are deployed. The SFEMC endpoint shows the status of the platform's clusters, the service function endpoints and their state (along with the ability to change their state) and the service function endpoint identifiers (FQDNs).

2.8 TRIAL PREPARATION

2.8.1 Obtain Ethical Approval

The University of Southampton, as project coordinator, have applied for and received ethical approval for all the work to be undertaken in the FLAME project (both by core partners and open call 3rd-parties). The application for this "blanket" consent may be found in OwnCloud (under "ethics"). Any deviation from the activities described in the application must be checked with the project's Ethics Board and potentially a further ethics approval application would have to be made.

Note, as the ethics approval application was made to approve all the activities in one go the application document has a few "blanks" in it. Appendix (ii) does not exist: each trial would use a different questionnaire or other data capture method such as a focus group (where you would instead document some seed questions). Appendix (v) is for the case where someone else owns the relationship with the participants. In this case then the trial leader needs to get a simple email from the person providing access to the participants to say that they are happy for them to be used in the study.

Some organisations require researchers to obtain local ethics approval. If this is the case, then the local ethics board may be content to see the application made to the University of Southampton board and to know that it was approved.

2.8.2 Prepare Participant Information Sheet

Part of the ethics approval is the requirement that a specific form of participant information sheet is provided to participants. The information sheet needs to be adapted to a trial by adding in the contact name(s) of the researchers.

The information sheet must be given to participants in a language they are fluent in (to ensure they comprehend the information). Replication sites will provide translations of the document to one of their official languages.

2.8.3 Prepare Participant Consent Form

As for the Participant Information Sheet, the approved Consent Form needs minor adaptation to the trial. Again, a Spanish translation will be provided.

2.8.4 Register with Data Protection Authority

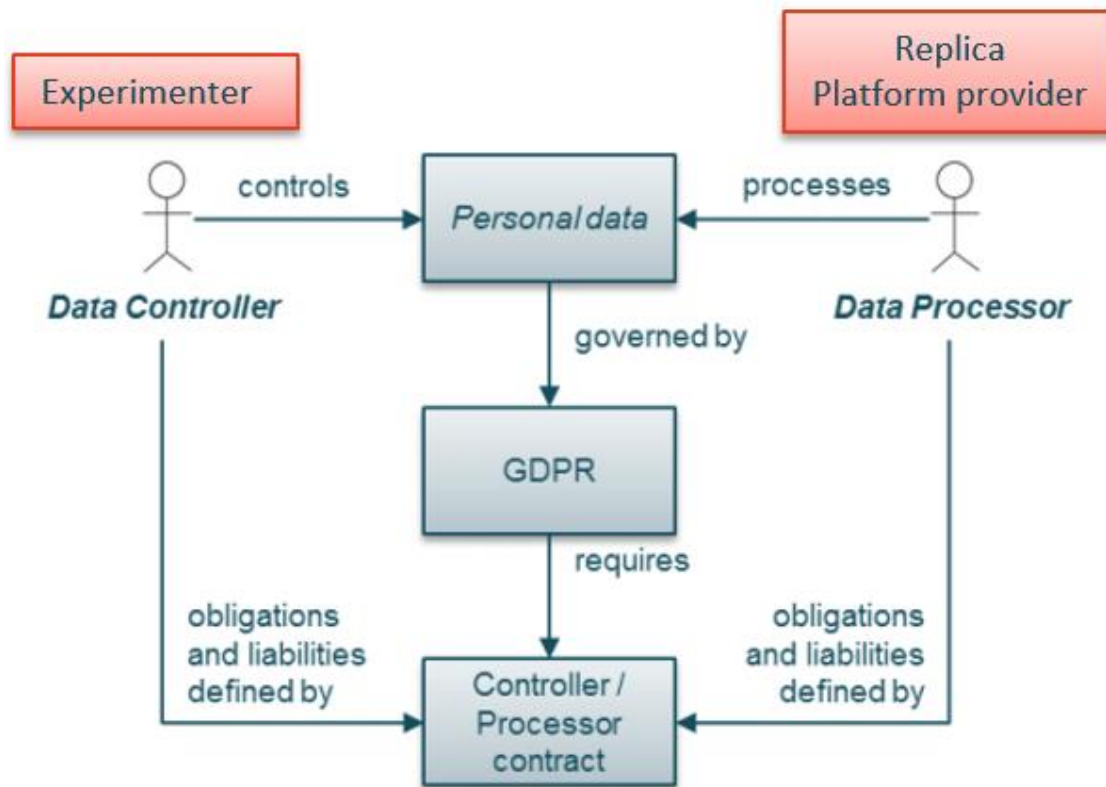
Any company holding personal data must register with their local Data Protection Authority (DPA). You need to check that you are registered appropriately. It is important to note that the purpose for which you hold data is declared in the registration. Your company may well be registered with the DPA (e.g. for managing employee data) but not for research purposes.

Organisations outside of the EC must also register with a DPA within the EC. e.g. Swiss companies may register with the UK or Spanish DPA.

2.8.5 Sign Data Sharing Agreement

The General Data Protection Regulation (GDPR) mandates that a bilateral contract must be in place between any two parties sharing personal data where one is a data controller and one is a data processor.

In a trial, the data controller is the trial leader and, if any personal data is sent to the media services or platform, then the platform host (e.g. University of Bristol or i2CAT) are the data processor.



Data sharing

A standard data sharing contract is being developed and will be provided when ready.

More detail may be found in [D7.1 Data Management Action Plan v1](#)

2.8.6 Engage Participants

Engaging participants is hard and needs to be planned well in advance. Local partners may be able to help with attracting participants to your trials.

- Bristol: Aloizio Pereira da Silva aloizio.eisenmann.dasilva@bristol.ac.uk
- Barcelona: Raul, August, etc barcelona@ict-flame.eu

According to the ethical constraints, only small financial inducements (e.g. shopping vouchers) may be used.

Combining trials from several parties together may be an effective way of increasing the participant engagement and pooling the publicity efforts.

2.8.7 Write Data Management Plan

The data management plan for a trial describes the data which will be captured and specifies how it will be managed and how open it will be made. [D7.1 Data Management Action Plan v1, section 11](#) describes the data management plan which trials must complete. A copy of the plan is part of the Project Plan template and will be reviewed by the 3rd Party Project Manager. Three classes of data management plan are provided: Gold, Silver and Bronze. The distinction being that for Gold all data is open, for Silver some data is open and for Bronze no data is open.

The initial data management plan is a bare-bones version, capturing the essential information. A complete data management plan must be provided at the end of a trial project. Ideally, datasets should be deposited in Zenodo, especially those which are used in publications.

2.8.8 Agree Schedule

Trials at the replicator must be agreed according to local rules. The [FLAME experiment calendar](#) should then be updated to record the activity.

2.8.8.1 Bristol

Contact: Aloizio Pereira da Silva aloizio.eisenmann.dasilva@bristol.ac.uk

Millenium Square in Bristol is privately owned and managed by We The Curious (WTC). It is generally completely open to the public but it is **not** a public space and so additional considerations apply.

There are two categories of tests we regularly encounter:

1. Very small test max 1-3 people: For example, testing connectivity in Millennium Square for no more than a few hours. In this case it is not necessary to pre-arrange with WTC: just arrive at Millenium Square and perform the tests. If there are obstacles in the way due to some other (non-UoB) event, then work around them.
2. More than 3 people or longer term, and a need to e.g. erect a tent: this needs to be agreed with WTC for use of the space. This is the formal method of organising use of the space and should definitely be done for larger groups of people or events that last more than a few hours.

2.8.8.2 Barcelona

Contact: barcelona@ict-flame.eu

Permission for carrying out experiments on street needs to be requested with at least 3 weeks of anticipation.

Additional requirements need to be notified (e.g. table, power plugs, etc.) and availability will be checked by infrastructure providers. More complex setups may require additional planning and thus earlier notification.

- Testbed supports a single experiment at a time
- Operation times of the testbed are 9h – 15h
- During (local) holidays the testbed is not accessible
- Cabinet and lamp posts are “out of reach” to experimenters
- Hardware cannot be modified/upgraded

2.8.9 Disseminate

Experiments and trials offer value to:

- FLAME experimenters
- The FLAME project
- The scientific & technical community
- City (replica) stakeholders

Dissemination should be planned well in advance. Think ahead to what these audiences will be interested to learn, what should be presented, and how. Capture as much as you can (notes; pictures; A/V; interviews): the data captured can also serve to help you understand the trial itself. It is worth considering using professional equipment for any audio or video. You must also make sure that the consent form presented to trial participants includes information on what will be captured for dissemination purposes and the ability for them to opt-out of this aspect should they wish.

3 SANDPIT

The FLAME sandpit is an infrastructure hosted at IT Innovation in Southampton. There is no physical access.

3.1 AUTHORS

Author	Organisation
Stephen C Phillips	IT Innovation

© University of Southampton IT Innovation Centre and other members of the FLAME consortium.

3.2 RESOURCES

The sandpit machine, "givry", has the following specification:

- PowerEdge R440 Server
- 2x Intel Xeon Gold 6150 2.7G, 18C/36T, 10.4GT/s 2UPI, 25M Cache, Turbo, HT (165W)
- 256GB RAM (DDR4-2666)
- 2x 4TB NLSAS disks (all disks now have 12Gbps bus)
- 1x 480GB SSD
- Ubuntu 18.04 (Bionic)

Of course, not all of this is available to the experimenter. Approximately 20 cores spread over 4 clusters is available for service function endpoints.

3.3 ACCESS

[Full documentation](#)

To use the sandpit you must first request an account by raising an issue. An account for your organisation will be set up and named e.g. "atos". For each person who will use the account in your organisation you need to [provide the public part of their SSH key](#).

You then need to access the sandpit using SSH. It is possible to directly SSH to the sandpit machine and execute commands there or set up an SSH tunnel to the machine so that commands can be executed locally. By using an SSH tunnel, clients such as web browsers may be connected to the platform services, to emulated user equipment and to service function endpoints (deployed media services) themselves.

3.3.1 Scheduling

On the whole, one experimenter at a time will be scheduled to use the testbed. Reservations may be seen in [the shared experiment calendar](#).

3.4 ENDPOINTS

Depending on your chosen access method, the endpoints would begin with different domain names and ports (see [sandpit access](#)).

Available endpoints (via command line) are then:

- `/clmc/clmc-service` to access the CLMC API
- `/clmc/chronograf/v1` to access the Chronograf API
- `/clmc/kapacitor` to access the Kapacitor API
- `/clmc/influxdb` to access the InfluxDB API
- `/clmc/neo4j` to access the Neo4J API
- `/repository` to access the image repository

The following endpoints are also available for use through a web browser:

- `/clmc/chronograf` to access the Chronograf web page
- `/clmc/neo4j/browser` to access the Neo4J browser web page
- `/orchestrator/sforch` to access the Orchestrator API
- `/orchestrator/sfemc` to access the SFEMC API

3.4.1 Image Repository

The image repository in the sandpit is implemented using WebDAV.

4 BARCELONA INFRASTRUCTURE

4.1 AUTHORS

Name	Affiliation
August Betzler	i2CAT

© Copyright i2CAT and other members of the FLAME consortium.

4.2 RESOURCES

The Barcelona infrastructure offers both compute and radio access network (RAN) to the experimenters. The compute nodes available to the experimenters can be split in to two locations:

- The Main Data Center (DC) at the i2CAT premises located in "Zona Universitaria" [Main DC \(Google Maps\)](#)
- The cabinet server at the edge, deployed in the street cabinet on Pere IV street where experiments will take place [Street Deployment \(Google Maps\)](#)

The on-street deployment consists of a street cabinet that hosts a compute node and networking equipment. The networking equipment is used to connect with the main DC and also with the 4 lamp posts that are hosting the RAN elements. For the RAN, a Wi-Fi transceiver implementing the IEEE 802.11ac standard is installed in each of the Wi-Fi nodes hanging on the lamp posts. The transceivers support 2x2 MIMO and allow the experimenters to choose 5 GHz, non-DFS channels for the experiments with different channel bandwidths (20/40/80 MHz).

The RAN performance depends on where the STAs are located and how the APs are configured. For single STA connections we observe throughputs of up to 200-250 Mbit/s. This performance can degrade with the number of users and ambient conditions, such as weather conditions (e.g. rain) or obstacles on the street (e.g. trucks).

The performance of the wireless backhaul fluctuates between 180 Mbit/s and 220 Mbit/s, depending on the link and direction of transmission between two lamp posts.

Further, as a particular feature of the Barcelona infrastructure, experimenters will be using wireless backhauling if they choose to include several lamp posts in their experiments. The wireless mesh backhaul is established over directive antennas between pairs of lamp posts, using IEEE 802.11ac transceivers as for the RAN.

The following table indicates the amount of vCPUS, RAM and disk storage (SSD) will be available for the experimenters to be used for their experiments:

Location	vCPUS	RAM	Storage
Main DC	10	32 GB	200 GB
Edge	14	32 GB	200 GB

5 UNIVERSITY OF BRISTOL INFRASTRUCTURE

5.1 AUTHORS

Name	Affiliation
Aloizio P. Silva	University of Bristol

© Smart Internet Lab, University of Bristol and other members of the FLAME consortium.

5.2 RESOURCES

5.2.1 Millennium Square Infrastructure

FLAME infrastructure in Bristol is part of the 5GUK Test Network that is deployed at Bristol City Centre. In particular, at Millennium Square (MS) and We The Curious (WTC) Museum.

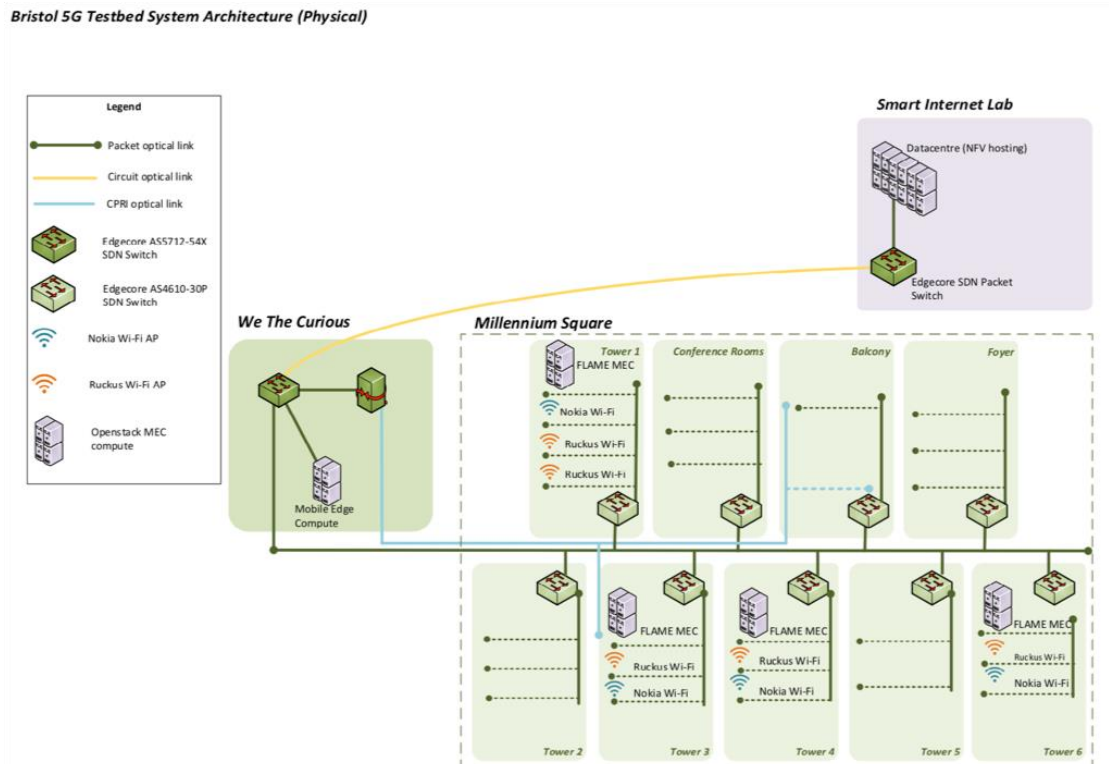
[Millennium Square may be explored in Google Street View](#)

In order to explore and validate the deployment of 5G in an architecture that combines existing technologies and innovations, University of Bristol have deployed a rich testbed comprised of several networking and computing technologies, interconnecting a significant area in the Bristol city centre. This testbed aims to provide a managed platform for the development and testing of new solutions delivering reliable and high-capacity services to several applications and vertical sectors here referred to as FLAME.

The University of Bristol's 5G testbed is a multi-site network connected through a 10km fibre with several active switching nodes, that are depicted in Figure 1.

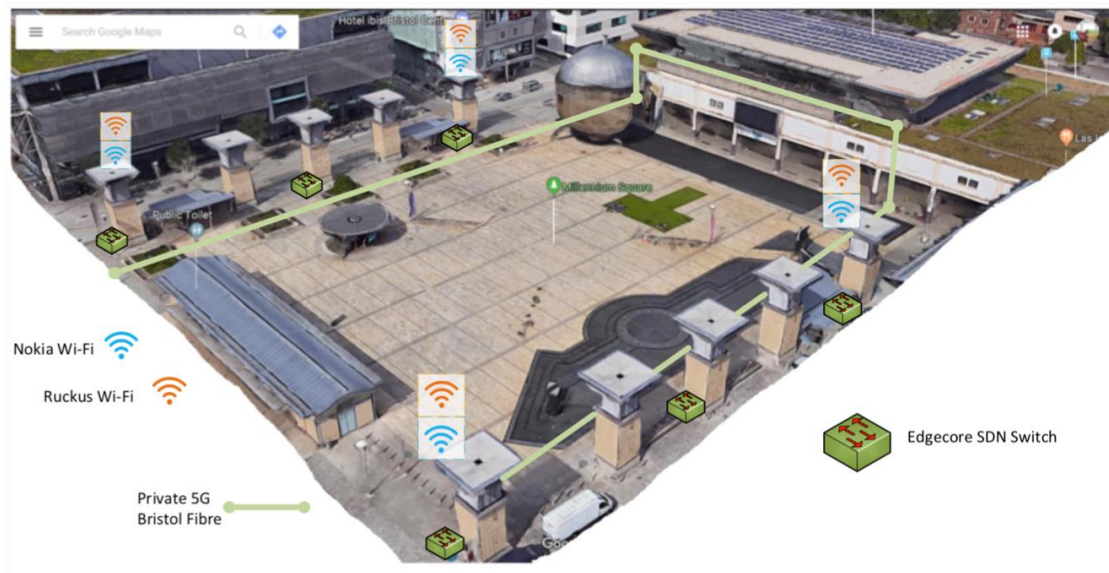
Figure 1 - Bristol High Level Architecture View:

Bristol 5G Testbed System Architecture (Physical)



The core network is located at the Smart Internet Lab at the University of Bristol and an extra edge computing node is available in another central location, known as Watershed. As shown in Figure 2., the access technologies are located in the city centre: Millennium Square for outdoor coverage.

Figure 2 - Distribution of the testbed access technologies:



A summary of the testbed constituent equipment and capabilities is:

- Multi-vendor software-defined networking (SDN) enabled packet switched network

- Corsa switch (Corsa DP2100)

- Edgecore switch (Edgecore AS4610 series & AS5712-54X)
2. SDN enabled optical (Fibre) switched network
 - Polatis Series 6000 Optical Circuit Switch.
 3. Multi-vendor Wi-Fi
 - SDN enabled Ruckus Wi-Fi (T710 and R720)
 - Nokia Wi-Fi (AC400) (Not Available for FLAME)
 4. Mobile Edge Computing node
 - Dell Power Edge R430 Server
 - **Each MEC has available to the experimenter: cores = 15, RAM = 29GB, disk = 900GB**

5.2.2 FLAME Infrastructure

Figure 3 shows the logical FLAME platform architecture deployed at Millennium Square in Bristol. A set of four towers has been allocated to host FLAME Mobile Edge Computing nodes. Each tower has a compute node based on OpenStack Ocata NOVA. Each compute node is connected to an EdgeCore SDN switch that is connected to a single SDN switch located at WTC. The four compute nodes and the edge core SDN switches are connected to the SDN controller based on FloodLight also located at the WTC.

Figure 3 - Bristol FLAME Infrastructure Topology:

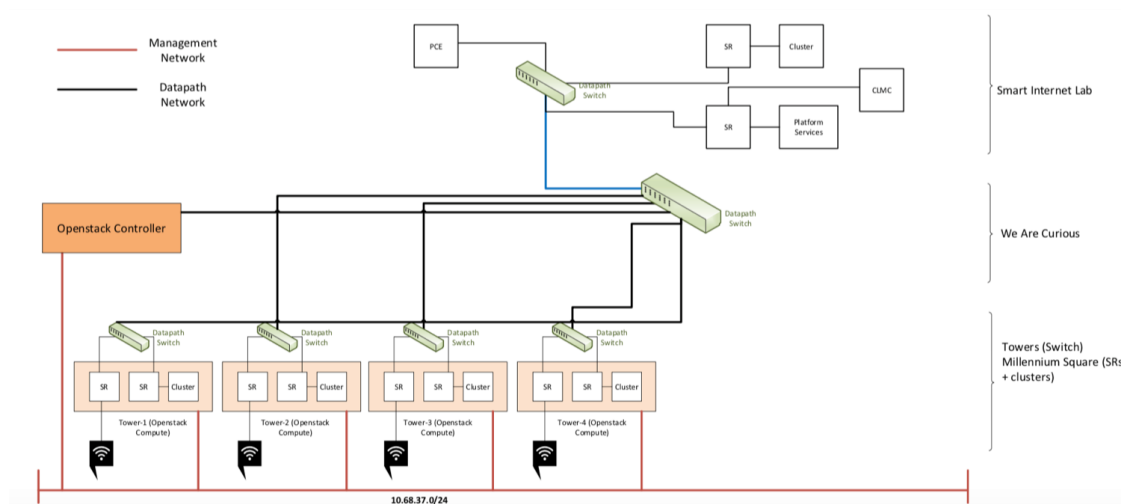



Figure 4 - FLAME MEC node specification:

PowerEdge R430 Server	
	
Components	
1	Intel Xeon E5-2660 v3 2.6GHz, 25M Cache, 9.60GT/s QPI, Turbo, HT, 10C/20T (105W) Max Mem 2133MHz
1	3.5" Chassis with up to 4 Hard Drives
1	Dell EMC 1U Standard Bezel
1	Riser with Two x16 PCIe Gen3 LP slots (x16 PCIe lanes), R430
1	iDRAC Port Card
1	Performance Optimized
1	2400MT/s RDIMMs
4	8GB RDIMM, 2400MT/s, Single Rank, x8 Data Width
1	DIMM Blanks for System with 1 Processor
1	135W Heatsink
1	iDRAC8 Enterprise, integrated Dell Remote Access Controller, Enterprise
2	1.2TB 10K RPM SAS 2.5in Hot-plug Hard Drive, 3.5in HYB CARR
1	PERC H730 Integrated RAID Controller, 1GB Cache
1	Dual, Hot-plug, Redundant Power Supply (1+1), 550W
2	C13 to C14, PDU Style, 10 AMP, 2 Feet (.6m), Power Cord
1	PowerEdge Server FIPS TPM 2.0
1	Asset Tag - ProSupport (Website, barcode, Onboard MacAddress)
1	Intel Ethernet I350 DP 1Gb Server Adapter, Low Profile
1	On-Board LOM
1	ReadyRails Sliding Rails Without Cable Management Arm
1	RAID 1 for H330/H730/H730P (2 HDDs or SSDs)
Software	
1	PowerEdge R430/R530 Motherboard MLK
1	Performance BIOS Settings
1	No Systems Documentation, No OpenManage DVD Kit
Service	
1	Base Warranty
1	3Yr Basic Warranty - Next Business Day - Minimum Warranty
1	5Yr ProSupport and Next Business Day On-Site Service

5.3 ACCESS TO BRISTOL FLAME PLATFORM

Below are the instructions about how to access the Bristol FLAME Platform:

1. Send a request via GitLab asking for Bristol testbed access and assignee should be Aloizio
2. An answer will be returned that the request is in progress
3. The VPN credential to access SIA network will be provided by email
4. The issue in GitLab will be closed after the first access via VPN

6 HOWTO LOG IN TO THE SANDPIT

This document is specific to the FLAME testbed ("sandpit") on IT Innovation's server.

6.1 AUTHORS

Author	Organisation
Stephen C Phillips	IT Innovation

© University of Southampton IT Innovation Centre and other members of the FLAME consortium.

6.2 ACCOUNT ON THE SERVER

Each organisation that requires access to the test infrastructure will have an account created on the server ("givry") using their organisation's short-name, e.g. "atos" (shown as <your-account-name> below). You will be asked for the public SSH keys of individuals at your organisation who need to be able to login to the account. These keys will be added to the account's ~/.ssh/authorized_keys file and they should be relatively new keys of at least 1024 bytes. An individual's key is referred to as <your-private-key-file> below.

GitLab provide [useful instructions on generating key-pairs on different platforms](#). The more complex instructions below generally assume you have a linux system as your local machine or that you have adapted your Windows machine to behave similarly. Adapting a Windows machine may be done using e.g. [the full version of the cmd shell](#) (recommended), [mingw](#), [cygwin](#) or, for Windows 10 users, the [Windows Subsystem for Linux](#). Simple access may be done through an SSH client such as PuTTY.

To log in to givry, from a machine holding your private key:

```
ssh <your-account-name>@givry.it-innovation.soton.ac.uk
```

If your private key is somewhere non-standard you can use:

```
ssh <your-account-name>@givry.it-innovation.soton.ac.uk -i <your-private-key-file>
```

6.3 RECOMMENDED SSH CONFIGURATION

In summary, put the following into your ~/.ssh/config file:

```
Host sandpit
  HostName givry.it-innovation.soton.ac.uk
  User <your-account-name>
  IdentityFile <your-private-key-file>
  ForwardAgent yes

Host sandpit-tunnels
  HostName givry.it-innovation.soton.ac.uk
  User <your-account-name>
  IdentityFile <your-private-key-file>
  LocalForward 9000 tutorial:80
```

```
LocalForward 9001 platform:80
```

```
Host ue20 ue22 ue23 ue24
  ProxyCommand ssh sandpit -W %h:%p
  User ubuntu
  IdentityFile ~/.ssh/flame-ue
  StrictHostKeyChecking no
  UserKnownHostsFile=/dev/null
  ForwardAgent yes
```

You will also then need to copy the flame-ue key to your local computer using scp
 sandpit:~/.ssh/flame-ue ~/.ssh.

With that in place, you can just type:

- `ssh sandpit` to log in to `givry.it-innovation.soton.ac.uk` and the account, hostname and keyfile are automatically used.
- `ssh ue20` (or the other ue machines) to access them directly from your local machine.
- `ssh sandpit-tunnels -nNT &` to create a background job which instantiates two SSH tunnels:
- `localhost:9000` to access the interactive tutorial
- `localhost:9001` to access the platform services

All the detail you need on this and more can be found below.

6.3.1 SSH agent

We recommend using an ssh-agent to avoid the need for entering your private key password.

On linux machines:

```
eval `ssh-agent`
ssh-add
```

On Windows machine you might use PuTTY and Pageant:

- Run Pageant and load in your private key.
- In PuTTY create an ssh connection to `<account>@givry.it-innovation.soton.ac.uk`.
- Make sure "Attempt authentication using Pageant" is ticked in the Connection / SSH / Auth panel.

6.3.2 SSH agent forwarding

You may find it useful to forward your private identity onto givry (and then into the VMs), meaning that you are then able to e.g. checkout git repositories using your private personal identity. To do this, you must be using an ssh-agent.

On linux machines:

```
ssh -A <your-account-name>@givry.it-innovation.soton.ac.uk
```

On Windows, using PuTTY:

- Tick "Allow agent forwarding" in the Connection / SSH / Auth panel.

You can test whether the agent forwarding has worked by typing `echo $SSH_AUTH_SOCK` once logged in. If a socket is shown then you have a link (a socket) back to the ssh-agent on your home machine.

Using SSH agent forwarding is a security risk in that anyone with root access on the destination machine can access the socket and authenticate using your private key.

6.3.3 Identity on givry

An SSH identity will have been created for you on givry: a keypair in `~/ .ssh/id_rsa`. The keypair has no password: do not use it for anything outside of givry.

6.4 ACCESSING THE INTERACTIVE TUTORIAL

The interactive tutorial is hosted in a virtual machine on givry. To access the tutorial in a web browser on your local machine you must make an SSH tunnel through givry to the tutorial machine.

```
ssh -nNT -L 9000:tutorial:80 <your-account-name>@givry.it-innovation.soton.ac.uk &
```

Or, if you have configured SSH as suggested above, you can do:

```
ssh -nNT -L 9000:tutorial:80 sandpit &
```

The `-nNT` options along with the final `&` allow the ssh connection to be put into the background (use e.g. `kill %1` to kill it). The `-L` option sets up the tunnel so that data sent to the local port 9000 (which you can change if you wish) is forwarded to port 80 on the tutorial machine (port 80 is the standard port for web pages). The `tutorial:80` part is in the context of the ssh connection to givry.

To make this easy, add the following to your SSH config (in `~/ .ssh/config`):

```
Host sandpit
  HostName givry.it-innovation.soton.ac.uk
  User <your-account-name>
  IdentityFile <your-private-key-file>
  ForwardAgent yes
```

```
Host sandpit-tunnels
  HostName givry.it-innovation.soton.ac.uk
  User <your-account-name>
  IdentityFile <your-private-key-file>
  LocalForward 9000 tutorial:80
  LocalForward 9001 platform:80
```

You will then be able to just type `ssh sandpit-tunnels` to log in to givry and create the tunnel described above to the tutorial machine and a second tunnel to the platform services (see below). If you want to just create the tunnels and background the process, then use `ssh -nNT sandpit-tunnels &`.

With the recommended tunnel in place, the tutorial is accessible on <http://localhost:9000/tutorial.html>.

6.5 ACCESSING THE PLATFORM SERVICES

Access to platform services is via a proxy on givry in a VM named platform. It provides access to platform services such as the CLMC, orchestrator and image repository.

If you are logged into givry (`ssh sandpit`), the platform services may be accessed through endpoints such as:

- <http://platform/clmc/clmc-service> to access the CLMC API
- <http://platform/clmc/chronograf/v1> to access the Chronograf API
- <http://platform/clmc/kapacitor> to access the Kapacitor API
- <http://platform/clmc/influxdb> to access the InfluxDB API
- <http://platform/clmc/neo4j> to access the Neo4J API
- <http://platform/repository> to access the image repository

Clients such as `curl` or `wget` may be used with APIs under these endpoints, such as:

```
wget http://platform/clmc/kapacitor/v1/tasks
curl http://platform/clmc/influxdb/query?db=mydb --data-urlencode 'q=SELECT * FROM "mymeasurement"'
```

The following endpoints are also available but as they are (primarily) web interfaces it is better to access them via an SSH tunnel (see below):

- <http://platform/clmc/chronograf> to access the Chronograf web page
- <http://platform/clmc/neo4j/browser> to access the Neo4J browser web page
- <http://platform/orchestrator/sforch> to access the Orchestrator API
- <http://platform/orchestrator/sfemc> to access the SFEMC API

To access endpoints via a web browser, first of all an SSH tunnel must be created. To create an SSH tunnel to the platform machine through givry, execute the following command on your local machine:

```
ssh -nNT -L 9001:platform:80 <your-account-name>@givry.it-innovation.soton.ac.uk &
```

Or, if you have configured SSH as suggested above, you can do:

```
ssh -nNT -L 9001:platform:80 sandpit &
```

By far the easiest option however is to use [the SSH configuration described above](#) and then just typing `ssh sandpit-tunnels`.

Once the tunnel is in place, you may use a local client such as a web browser to access the endpoints. Replacing platform in the addresses above with localhost:9001. For instance, on your local machine you can type e.g. `http://localhost:9001/clmc/chronograf` into the address bar of a web browser to access the Chronograf web interface (the 9001 here must match the port number chosen in the SSH tunnel command).

The tunnel also lets you interact with API endpoints from the command line on your local machine. For instance get the list of Kapacitor alerts execute:

```
wget http://localhost:9001/clmc/kapacitor/v1/tasks
```

With the recommended tunnel in place, the API endpoints are:

- `http://localhost:9001/clmc/clmc-service` to access the CLMC API
- `http://localhost:9001/clmc/chronograf/v1` to access the Chronograf API
- `http://localhost:9001/clmc/kapacitor` to access the Kapacitor API
- `http://localhost:9001/clmc/influxdb` to access the InfluxDB API
- `http://localhost:9001/clmc/neo4j` to access the Neo4J API
- `http://localhost:9001/repository` to access the image repository

The web page endpoints are:

- `http://localhost:9001/clmc/chronograf` to access the Chronograf web page
- `http://localhost:9001/clmc/neo4j/browser` to access the Neo4J browser web page
- `http://localhost:9001/orchestrator/sforch` to access the Orchestrator API
- `http://localhost:9001/orchestrator/sfemc` to access the SFEMC API

6.6 ACCESSING THE USER EQUIPMENT (EMULATED CLIENTS)

Experimenters are also able to access "emulated UEs" (emulated user equipment) which take the role of users' mobile phones. They are virtual machines which the experimenter can log into and install client software or load emulation scripts in order to execute tests on their deployed service function chain. The UE machines are named ue20, ue22, ue23 and ue24 (with their numbers matching the clusters they directly connect to).

In a similar way to accessing the platform services, the emulated clients can be accessed via through commands typed into a remote SSH shell or by opening an SSH tunnel and then using a local command line or local clients such as web browsers. The difference is that two SSH hops are required rather than one.

All the UE machines have an "ubuntu" account which has a "flame-ue" key as an `authorized_key`. The flame-ue private and public keys may be found in your `~/ .ssh` folder on givry.

The simple way to log in e.g. ue20 is as follows:

```
ssh <your-account-name>@givry.it-innovation.soton.ac.uk -i <your-private-key-file>
ssh ue20 -i ~/.ssh/flame-ue
```

The ubuntu user on the UEs can become root with `sudo su` and your client software may then be installed.

To make access easier you first need to copy the flame-ue key to your local machine. At this point we assume you have added the Host `sandpit` entry to your local `~/.ssh/config` file. If not then you must specify username, hostname and keyfile as appropriate. From your local machine do:

```
scp sandpit:~/.ssh/flame-ue ~/.ssh
```

Add the following snippet to your `~/.ssh/config` file on your local machine:

```
Host ue20 ue22 ue23 ue24
    ProxyCommand ssh sandpit -W %h:%p
    User ubuntu
    IdentityFile ~/.ssh/flame-ue
    StrictHostKeyChecking no
    UserKnownHostsFile=/dev/null
    ForwardAgent yes
```

This entry (in combination with the one for Host `sandpit`) lets you do e.g. `ssh ue20` and you will log in to ue20 with no further steps. The way it works is that when you execute `ssh ue20` the `ProxyCommand` runs first and makes an SSH connection to `sandpit` and tunnels through there to execute the SSH to ue20. Your local SSH client directly authenticates with ue20 (hence the need for the local copy of the flame-ue key).

With this configuration in place it is then possible to tunnel through the this 2-hop SSH connection to directly access a port on the UE. For instance, to access a service listening on port 8080 deployed on ue20, the following tunnel may be used:

```
ssh -nNT -L 9002:localhost:8080 ue20 &
```

In a web browser on your local machine you can then go to `http://localhost:9002` and the traffic will reach port 8080 on ue20.

6.7 MORE ON SSH TUNNELLING

For more information on SSH tunnelling, we suggest these references:

- <https://blog.trackets.com/2014/05/17/ssh-tunnel-local-and-remote-port-forwarding-explained-with-examples.html>
- <https://superuser.com/questions/96489/an-ssh-tunnel-via-multiple-hops/565167>

7 FLAME PACKAGING AND WHOAMI

7.1 AUTHORS AND REVIEWERS

Name	Email
Sebastian Robitzsch	sebastian.robitzsch@interdigital.com
Tomas Aliaga	tomas.aliaga@martel-innovate.com

© InterDigital and other members of the FLAME consortium.

7.2 PREREQUISITES

The packaging of a FLAME service must be conducted outside the FLAME platform. As the resulting images are given to the FLAME orchestrator the underlying host environment must be identical to the one used for the clusters of the FLAME platform.

Download and install a [Ubuntu 16.04 64bit](#) environment either on a bare-metal machine or as a virtual instance using a hypervisor of your choice. It is recommended to ensure the CPU support for virtualisation is passed up to the virtual packaging instance. The installation script is notifying you when this setting is not available. In case virtualisation cannot be enabled for whatever reason the installation of the base image will take significantly longer.

7.3 INSTALLATION OF HYPERVISORS

FLAME uses LXD or KVM as their hypervisors on each cluster, though focus is shifted towards LXD for now. To mirror this environment into your local packaging instance the `install-hypervisors.sh` script is taking care of that. Simply become root (`sudo su`) and execute the script:

```
~flame-packaging/$ ./install-hypervisors.sh
```

If virtualisation is not available on your system, the script will warn you about that. However, you can safely ignore this warning and it does not prevent you from packaging your service functions. Note, in case of no virtualisation support, it is highly recommended to only package LXD-based containers.

7.4 CREATE VIRTUAL INSTANCES

Within the packaging folder the supported hypervisors are given as sub folders within the necessary scripts can be found to create, export and delete an image.

7.4.1 KVM

The script to create a new KVM-based virtual instance is `kvm/create.sh`. To create a KVM-based image the operating system must be installed first. The script checks whether or not a base image has been already created for the selected Linux flavour. If not, the script creates a new instance of name `base-<LINUX_FLAVOUR>` which is then cloned every time a new service function is packaged using this particular Linux flavour. During installation please ensure you are setting the hostname to base with

username flame (password of your choice). Once the base image has been created the script clones the VM using the name given as the second parameter to `kvm/create.sh`.

7.4.2 LXD

Within `lxd/` the script `create.sh` resides which allows to create a new container based on an existing mint version from the [LXD images repository](#). The script takes two parameters as input, i.e., the Linux flavour and the name of the container. The container name - similar to the VM name - represents the service function.

The list of available container images can be retrieved by executing

```
lxc image alias list images:
```

To pre-filter the images based on the flavour append the flavour name, e.g.:

```
lxc image alias list images:debian
```

Simply choose your preferred Linux version and provide the entire flavour/version/architecture name as the first argument to the `create.sh` script, e.g.:

```
~flame-packaging/lxd/$ ./create.sh debian/9/amd64 <SF_NAME>
```

7.5 EXPORTING

The exporting of a packaged service function shuts down the image and creates a TAR ball from the `qcow2` image. The TAR ball must be either placed on a publicly reachable web server or it must be given to the FLAME platform provider so that it can be placed on the FLAME service function image server.

7.5.1 KVM

To export a KVM image invoke

```
~flame-packaging/kvm/$ ./export-vm.sh <VM_NAME>
```

where `<VM_NAME>` is the name of the virtual instance given when creating the VM with `kvm/create.sh`. If you cannot remember the exact name call `export.sh` without any parameters; the script lists all available images.

7.5.2 LXD

To export an LXD image invoke:

```
~flame-packaging/lxd/$ ./export <C_NAME>
```

where `<C_NAME>` is the name of the container when it was created with `lxd/create.sh`. The resulting TAR ball which is given to the FLAME orchestrator is placed in `/var/local/lxd/`.

7.6 DELETING

To delete an image entirely from the hypervisors and from the disk a dedicated script for each hypervisor is located in the respective folders.

7.6.1 KVM

To clean up a packaged instance from KVM `kvm/delete.sh` is available and takes care of all that.

7.6.2 LXD

To clean up the packaged LXD container `lxd/delete.sh` is available and takes care of all that.

7.7 HOW TO PACKAGE

The packaging of a newly created virtual instance can be done in various ways but always follows the same principles. Either use `ssh/scp` or the CLI of the hypervisor (`virsh console`, `lxc exec`, or `lxc file`).

The packaging toolchain pre-configures the virtual instance with various systemd services which are required to query the platform about "who am I", disables some network interface card related mechanisms to prevent corrupted packets, and configures telegraf responsible for monitoring. The three services are called `flame-whoami`, `flame-ethtool` and `flame-monitoring`.

`flame-ethtool` is started straight after networking becomes available. Once `flame-ethtool` has completed its tasks successfully `flame-whoami` is started which tries to obtain all the relevant whoami values from SFEMC for 600s. If it cannot complete this task in the given time window it will be stopped by systemd. Once it has completed its tasks all configured media-component services will be started.

Any media component which is packaged in an instance must be started as a systemd service after the `flame-whoami` service has finished. An example service file is given in `fs/lib/systemd/system/media-component.service` and looks as follows:

```
[Unit]
Description=flame-myservice
After=flame-monitoring.service
Requires=flame-monitoring.service

[Service]
EnvironmentFile=/etc/profile.d/flame_env.sh
ExecStart=/usr/bin/env bash -c 'echo "invoke your service here, Testing WHOAMI_SFC" : $WHOAMI_SFC"'

[Install]
WantedBy=multi-user.target
```

Simply replace `ExecStart` with your binary/script which should be started and call the file according to your service function, e.g. `frontend.service`. Copy the systemd service file to `/lib/systemd/system` inside the instance (using `scp` or `lxc`)

```
lxc file push my-app.service $LXD_NAME/lib/systemd/system/
```

Then run the following two commands to activate your service at system start-up:

```
~$ lxc exec $LXD_NAME -- systemctl daemon-reload
~$ lxc exec $LXD_NAME -- systemctl enable <SERVICE_FUNCTION>.service
```

where <SERVICE_FUNCTION> stands for the name of the systemd service file.

7.8 TEST SYSTEMD SERVICE INTEGRATION

In order to test the correct start of your packaged service functions the SFEMC whoami unit test can be re-used. To use this unit test please follow the instructions in the [unit-test folder](#). Simply ensure your service function has been packaged as an LXD container and is given as the LXD_SERVICE_FUNCTION argument to the bash scripts. The order of execution would be:

```
~$ ./prepare.sh <LXD_SERVICE_FUNCTION>
~$ ./run.sh <LXD_SERVICE_FUNCTION>
```

After the run.sh script has finished (and reported the unit test has "PASSED") check the status of your systemd service by invoking

```
~$ lxc exec <LXD_SERVICE_FUNCTION> -- systemctl status <SERVICE_FUNCTION>
```

Once all checks are done run

```
~$ ./cleanup.sh <LXD_SERVICE_FUNCTION>
```

which reverts some settings in the SF container (and cleans up the system from other things - more info see the [README.md](#) inside the unit-tests folder).

7.9 TEST SYSTEMD SERVICE FUNCTION SERVICE INTEGRATION

In order to test the correct start of your packaged service functions the SFEMC whoami unit test can be re-used. How to use this unit test please follow the instructions in the unit-test folder [unit-tests/README.md](#).

8 TOSCA TEMPLATING IN FLAME

8.1 AUTHORS

Authors

[Kay Haensge](#)

[Sebastian Robitzsch](#)

[Nikolay Stanchev](#)

[Michael Boniface](#)

Organisation

[InterDigital](#)

[InterDigital](#)

[University of Southampton, IT Innovation Centre](#)

[University of Southampton, IT Innovation Centre](#)

8.2 GENERAL

The repository contains documentation, example scripts and a validator for TOSCA templates, deployable in the FLAME environment.

8.3 VERSIONS

Within this repository we use the versioning system to keep track of changes and reaching certain milestones. An always stable version is on the [master branch](#). However, the latest version of documents, examples etc. is located at the [dev branch](#).

Furthermore, we provide fixed versions in product context. For this please refer to a [tagged version](#).

8.4 GETTING STARTED WITH TOSCA

A [Getting Started Guide](#) and [TOSCA snippets for best practices](#) are available inside the [Documentation](#) folder.

8.5 VALIDATION

To validate written TOSCA templates against the FLAME definition file, we offer a [Validator](#) (which is basically the OpenStack TOSCA-Parser).

8.6 CONTRIBUTING

The content is all open source and we are very happy to accept contributions. Please refer to [CONTRIBUTING.md](#) for details.

9 GETTING STARTED WITH TOSCA IN FLAME

The document will help to understand the way of how services can be deployed in a virtualised environment using template files and policies such as TOSCA (or HOT). Several Use Cases exist and will be described further in detail within this project.

9.1 THE THEORY OF TOSCA

The file descriptors used to specify the deployment of media services are based on [TOSCA YAML](#) and [TOSCA NFV](#) descriptors. The idea is to take advantage of these standard specifications and definitions provided by them. TOSCA offers the possibility to define virtual instance templates and their nodes to be deployed using an Orchestrator. Characteristics of these machines can be defined in the TOSCA template (e.g., nodes, properties, hardware and software requirements, policies, machine status, etc.).

9.1.1 Introduction to Standard specifications

TOSCA YAML describes a topology template with a graph of node templates, modelling these nodes and establishing relationships among them. It provides a type system of node types describing the blocks for constructing service templates. TOSCA NFV (Network Functions Virtualization) template provides a data model using TOSCA specification. The idea is to define all the topics related with the network virtualization in terms of equipment, protocols, servers and others, available to be located in a data centre or network. Both kind of documents, TOSCA YAML and NFV, need to deploy their functionalities using an Orchestrator. The Orchestrator is in charge of to collect the requirements of the templates, interpret these requirements and to leave ready these requirements. In order to clarify the concepts of deploy specifications, it can be the installation of a database in a virtual machine, start the database and provide the network requirements to be accessed remotely.

9.1.2 Methodology for the definition of the specification language

To capture the requirements of services within a TOSCA-compliant specification, we have created two files. One includes the FLAME-specific definitions of the field-names and its values for the actual TOSCA template. The second file is the template itself which captures the deployment services in possibly different locations. Within the template, a placement policy is applied with triggers to check the status of the overall system against single nodes or a group of nodes. If the triggers can be fired, a specific action towards the SFEMC will be sent to perform according to the template.

Per release, the definitions might be expanded, however the orchestrator and the SFEMC need to understand the meaning of the template definitions.

9.2 FLAME-BASED TOSCA DEFINITIONS

9.2.1 Resource Specification

TOSCA is being used to define required resources on the FLAME platform (such as nodes and their state in certain conditions) as well as to define the alerts ruleset on the CLMC side. The use of such a resource template is further described in the [Resource Description](#) document.

9.2.2 Alerts Specification

The TOSCA alert specification is used to configure alerts within CLMC. Alerts are configured through a YAML-based TOSCA-compliant document according to the TOSCA simple profile. This document is passed to the CLMC service, which parses and validates the document. Subsequently, the CLMC service creates and activates the alerts within Kapacitor, then registers the HTTP alert handlers specified in the document. The specification is compliant with the TOSCA policy template as implemented by the Openstack tosca parser.

The use of such a resource template is further described in the [Alerts Description](#) document.

9.2.3 FAQ and Best Practices

To get a fast insight, we encourage you to have a look into the [FAQ](#) or [Best Practices](#) document.

9.3 REFERENCES

1. [TOSCA Simple Profile YAML](#)
2. [TOSCA NFV](#)
3. [RFC5545](#)

10 TOSCA RESOURCE SPECIFICATION

10.1 GENERAL INFORMATION

In the following, the resource specification fields are described within their specific context.

The [resource definition file](#) includes these types and its definitions which are the base for defining a TOSCA resource specification.

10.1.1 Metadata

Within metadata we specify further information for that particular template. A name of the template, the author(s) as well as the template version have to be specified.

A Media Service is constructed using a **Service Function Chain**. The service function chain defines the relationships between service functions. The service function chain has a unique, but arbitrary string which is chosen by the media service provider and represents the entire chain.

Via `servicefunctionchain` we specify this identification of the deployed service chain.

10.1.2 Nodes

Within TOSCA you can specify your own type of `node_type`. These types have to derive from `tosca.nodes`.

We have defined an own type of SF Endpoint type and applies for every node which can be managed/orchestrated by FLAME's orchestrator or the SFEMC: `eu.ict-flame.nodes.ServiceFunction`.

The node name can be arbitrary. Generally, the node is containing the required capabilities (e.g., required number of CPUS, memory size, etc.) as well properties (type of hypervisor and where to get the image file from). These properties are also enhanced with the FQDN information element. It allows to register the intended *Fully Qualified Domain Names* for that service and its components. This field contains a list of strings, hence multiple FQDNs may be registered.

Note: The FQDNs are associated to that particular FLAME platform. Please make sure, only FQDNs with the managed domain suffix can be registered.

10.1.3 Policies

TOSCA Policies are a type of requirement that govern use or access to resources which can be expressed independently from specific applications (or their resources) and whose fulfillment is not discretely expressed in the application's topology (i.e., via TOSCA Capabilities) `[[1][tosca_yaml]]`.

Such policies consist of a specified type, some properties and trigger(s). However, it is recommended to have only one trigger per policy.

10.1.3.1 Policy types

We have defined several types of policies:

- **Initial Policy:** A special policy type and is defined as `eu.ict-flame.policies.InitialPolicy` and these labeled policies shall be processed only at the beginning of the run-time of the deployment. It determines the first targeted state for the deployed nodes.
- **State Change Policy:** The second policy is defined as `eu.ict-flame.policies.StateChange` and represents the FLAME-specific lifecycle management. Within this type of policies, we allow the change of the lifecycle of a node on a given cluster/location. The policy is active when the condition results in true value from the event source.
- **Internet Access Policy:** This policy is defined as `eu.ict-flame.policies.InternetAccessPolicy` and specifies whether certain FQDNs are permitted (whitelisted), or prohibited (blacklisted).

For the condition within the *StateChange* policy, we specify the following syntax:

```
constraint: <event source>::<event identifier>
```

The `<event source>` is the remote processor for event notification, such as the `clmc`. The `<event identifier>` is the event ID which is specified in the remote processor's script/database processing unit. In case of wrong or unknown `<event source>` or `<event identifier>`, the SFEMC ignores such information elements.

In the case when using the CLMC as data processing unit, the reference has to point to a valid trigger entry on the *alerts specification*.

10.1.3.2 Triggers

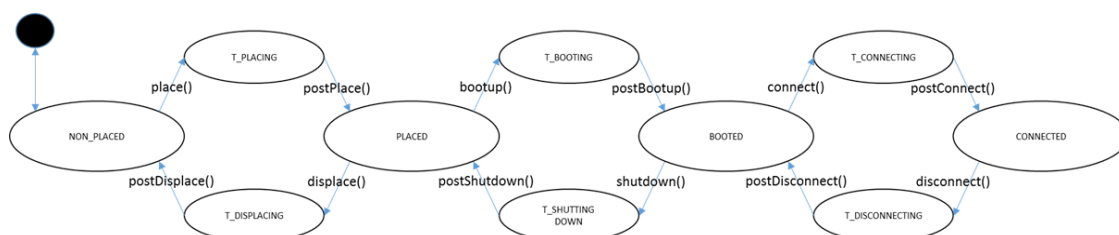
A trigger definition defines the event, condition, and action that is used to "trigger" a policy it is associated with.

These action elements point to the deployed clusters and the targeted SF Endpoint state, using the FQDN as identifier. As example, the trigger is fired and the described application, located in a London cluster, is set to the lifecycle state connected.

```
...
<node_name>:
  <cluster>: <target state value>
  ...
  ...
  ...
```

10.1.3.3 State Types

States are self-defined life-cycle keywords for the work with the State Machine inside the SFEMC.



SFEMC State Machine

In particular, inside the TOSCA template only the following target states are allowed to be set as action inside the triggers:

- `eu.ict-flame.sfe.state.lifecycle.connected`: Push the Endpoint to CONNECTED state
- `eu.ict-flame.sfe.state.lifecycle.booted`: Push the Endpoint to BOOTED state
- `eu.ict-flame.sfe.state.lifecycle.placed`: Push the Endpoint to PLACED state
- `eu.ict-flame.sfe.state.lifecycle.non_placed`: Push the Endpoint to NON_PLACED state

11 TOSCA ALERTS SPECIFICATION

11.1 DESCRIPTION

This document outlines the TOSCA alert specification used to configure alerts within CLMC. Alerts are configured through a YAML-based TOSCA-compliant document according to the TOSCA simple profile. This document is passed to the CLMC service, which parses and validates the document. Subsequently, the CLMC service creates and activates the alerts within Kapacitor, then registers the HTTP alert handlers specified in the document. The specification is compliant with the TOSCA policy template as implemented by the Openstack tosca parser. See an example below on github.com/openstack/tosca-parser.

11.1.1 TOSCA Alerts Specification Document

The TOSCA Alerts Specification Document consists of two main sections - **metadata** and **policies**. Each **policy** contains a number of triggers. A **trigger** is a fully qualified specification for an alert. Full definitions and clarification of the structure of the document is given in the following sections. An example of a valid alert specification document will look like:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: TOSCA Alerts Configuration document

imports:
- flame_clmc_alerts_definitions.yaml

metadata:
  servicefunctionchain: companyA-VR

topology_template:

  policies:
    - high_latency_policy:
      type: eu.ict-flame.policies.Alert
      triggers:
        high_latency:
          description: This event triggers when the mean network latency in a given location exceeds a given threshold (in ms).
          event_type: threshold
          metric: network.latency
          condition:
            threshold: 45
            granularity: 120
            aggregation_method: mean
            resource_type:
              flame_location: watershed
            comparison_operator: gt
          action:
            implementation:
              - flame_sfemc
              - http://companyA.alert-handler.flame.eu/high-latency

    - low_requests_policy:
```

```

type: eu.ict-flame.policies.Alert
triggers:
  low_requests:
    description: |
      This event triggers when the last reported number of requests for a
given service function
      falls behind a given threshold.
    event_type: threshold
    metric: storage.requests
    condition:
      threshold: 5
      granularity: 60
      aggregation_method: last
      resource_type:
        flame_sfp: storage
        flame_sf: storage-users
        location: watershed
      comparison_operator: lt
    action:
      implementation:
        - flame_sfemc
        - http://companyA.alert-handler.flame.eu/low-requests

- requests_diff_policy:
  type: eu.ict-flame.policies.Alert
  triggers:
    increase_in_requests:
      description: |
        This event triggers when the number of requests has increased relative
ve to the number of requests received
        120 seconds ago.
      event_type: relative
      metric: storage.requests
      condition:
        threshold: 100 # requests have increased by at least 100
        granularity: 120
        aggregation_method: mean
        resource_type:
          flame_sfp: storage
          flame_sf: storage-users
          flame_server: watershed
          flame_location: watershed
        comparison_operator: gte
      action:
        implementation:
          - flame_sfemc
    decrease_in_requests:
      description: |
        This event triggers when the number of requests has decreased relative
ve to the number of requests received
        120 seconds ago.
      event_type: relative
      metric: storage.requests
      condition:
        threshold: -100 # requests have decreased by at least 100
        granularity: 120
        aggregation_method: mean

```

```

    resource_type:
      flame_sfp: storage
      flame_sf: storage-users
      flame_location: watershed
    comparison_operator: lte
  action:
    implementation:
      - flame_sfemc

- missing_measurement_policy:
  type: eu.ict-flame.policies.Alert
  triggers:
    missing_storage_measurements:
      description: This event triggers when the number of storage measurements reported falls below the threshold value.
      event_type: deadman
      # deadman trigger instances monitor the whole measurement (storage in this case), so simply put a star for field value
      # to be compliant with the <measurement>.<field> format
      metric: storage.*
      condition:
        threshold: 0 # if requests are less than or equal to 0 (in other words, no measurements are reported)
        granularity: 60 # check for missing data for the last 60 seconds

      resource_type:
        flame_sfp: storage
    action:
      implementation:
        - http://companyA.alert-handler.flame.eu/missing-measurements

```

11.2 METADATA

The **metadata** section specifies the service function chain ID, for which this alerts specification relates to. The format is the following:

```

metadata:
  servicefunctionchain: <sfc_id>

```

11.3 POLICIES

The **policies** section defines a list of policy nodes, each representing a fully qualified configuration for an alert within CLMC. Each policy must be of type eu.ict-flame.policies.Alert. The format is the following:

```

topology_template:
  policies:
    - <policy_identifier>:
      type: eu.ict-flame.policies.Alert
      triggers:
        <event identifier>:
          description: <optional description for the given event trigger>

```

```

event_type: <threshold | relative | deadman>
metric: <measurement>.<field>
condition:
  threshold: <critical value - semantics depend on the event typ
e>
  granularity: <period in seconds - semantic depends on the even
t type>
  aggregation_method: <aggregation function supported by InfluxD
B - e.g. 'mean'>
  resource_type:
    <CLMC Information Model Tag Name>: <CLMC Information Model T
ag Value>
    <CLMC Information Model Tag Name>: <CLMC Information Model T
ag Value>
    ...
  comparison_operator: <logical operator to use for comparison,
e.g. 'gt', 'lt', 'gte', etc.
  action:
    implementation:
      - <flame_sfemc or HTTP Alert Handler URL - receives POST mes
sages from Kapacitor when alerts trigger>
      - <flame_sfemc or HTTP Alert Handler URL - receives POST mes
sages from Kapacitor when alerts trigger>
    ...
  ...

```

11.3.1 Policy definitions

- **policy_identifier** - policy label which **MUST** match with a StateChange policy in the TOSCA resource specification document submitted to the FLAME Orchestrator.
- **event_identifier** - the name of the event that **MUST** match with the *constraint* event name referenced in the TOSCA resource specification document submitted to the FLAME Orchestrator.
- **event_type** - the type of TICK Script template to use to create the alert - more information will be provided about the different options here, but we assume the most common one will be **threshold**. Currently, the other supported types are **relative** and **deadman**.
- **metric** - the metric to query in InfluxDB, must include measurement name and field name in format <measurement>.<field>. The only exception is when a **deadman** event type is used - then the <field> is not used, but the format is still the same for consistency. Therefore, using <measurement>.* will be sufficient.
- **threshold**
 - for **threshold** event type, this is the critical value the queried metric is compared to.
 - for **relative** event type, this is the critical value the difference (between the current aggregated metric value and the past aggregated metric value) is compared to.
 - for **deadman** event type, this is the critical value the number of measurement points (received in InfluxDB) is compared to.
- **granularity** - period in seconds

- for **threshold** event type, this value specifies how often should Kapacitor query InfluxDB to check whether the alert condition is true.
- for **relative** event type, this value specifies how long back in time to compare the current metric value with.
- for **deadman** event type, this value specifies how long the span in time (in which the number of measurement points are checked) is.
- **aggregation_method** - the function to use when querying InfluxDB, e.g. median, mean, etc. This value is only used when the event_type is set to **threshold** or **relative**.
- **resource_type** - provides context for the given event - key-value pairs for the global tags of the CLMC Information Model. This includes any of the following: "flame_sfp", "flame_sf", "flame_server", "flame_location". Keep in mind that **flame_sfc** and **flame_sfci** are also part of the CLMC Information Model. However, filtering on these tags is automatically generated and added to all InfluxDB queries by using the metadata values from the alerts specification. Therefore, including **flame_sfc** and **flame_sfci** in the **resource_type** is considered INVALID. For more information on the global tags, please check the [monitoring documentation](#).
- **comparison_operator** - the logical operator to use for comparison - lt (less than), gt (greater than), lte (less than or equal to), etc.
- **implementation** - a list of the URL entries for alert handlers to which alert data is sent when the event condition is true. If the alert is supposed to be sent to SFEMC, then instead of typing a URL, use **flame_sfemc** - the configurator will generate the correct SFEMC URL.

11.3.2 Event types

- **threshold** - A threshold event type is an alert in which Kapacitor queries InfluxDB on specific metric in a given period of time by using a query function such as *mean*, *median*, *mode*, etc. If the granularity is less than or equal to 60 seconds, then every measurement point is monitored (improving performance), thus, ignoring the aggregation function. This value is then compared against a given threshold. If the result of the comparison operation is true, an alert is triggered. For example:

high_latency:

description: This event triggers when the mean network latency in a given location exceeds a given threshold (in ms).

event_type: threshold

metric: network.latency

condition:

threshold: 45

granularity: 120

aggregation_method: mean

resource_type:

flame_location: watershed

comparison_operator: gt

action:

implementation:

- flame_sfemc

- http://companyA.alert-handler.flame.eu/high-latency

This trigger specification will create an alert task in Kapacitor, which queries the **latency** field in the **network** measurement on location **watershed** every **120** seconds and compares the mean value for the last 120 seconds with the threshold value **45**. If the mean latency exceeds 45 (**gt** operator is used, which stands for **greater than**), an alert is triggered. This alert will be sent through an HTTP POST message to the URLs listed in the **implementation** section.

The currently included InfluxQL functions are:

```
`"count", "mean", "median", "mode", "sum", "first", "last", "max", "min"`
```

The comparison operator mappings are as follows:

```
"lt" : "less than",
"gt" : "greater than",
"lte" : "less than or equal to",
"gte" : "greater than or equal to",
"eq" : "equal",
"neq" : "not equal"
```

- **relative** - A relative event type is an alert in which Kapacitor computes the difference between the current aggregated value of a metric and the aggregated value reported a given period of time ago. The difference between the current and the past value is then compared against a given threshold. If the result of the comparison operation is true, an alert is triggered. For example:

```
decrease_in_requests:
  description: |
    This event triggers when the number of requests has decreased relative to the
    number of requests received
    120 seconds ago.
  event_type: relative
  metric: storage.requests
  condition:
    threshold: -100
    granularity: 120
    aggregation_method: mean
  resource_type:
    flame_sfp: storage
    flame_sf: storage-users
    flame_location: watershed
  comparison_operator: lte
  action:
    implementation:
      - flame_sfemc
```

This trigger specification will create an alert task in Kapacitor, which compares the mean **requests** value reported in measurement **storage** with the mean value received **120** seconds ago. If the difference between the current and the past value is less than or equal to (comparison operator is **lte**) **-100**, an alert is triggered. Simply explained, an alert is triggered if the **requests** current value has decreased by at least 100 relative to the value reported 120 seconds ago. The queried value is contextualised for service function **storage-users** (using service function package **storage**) at location **watershed**. Triggered alerts will be sent through an HTTP POST message to the URLs listed in the **implementation** section.

Note: if X is the current timestamp, the current aggregated value refers to the period $\{X\text{-granularity}; X\}$ while the past aggregated value refers to the period $\{X - 2*\text{granularity}; X - \text{granularity}\}$

- **deadman** - A deadman event type is an alert in which Kapacitor computes the number of reported points in a measurement for a given period of time. This number is then compared to a given threshold value. If less or equal number of points have been reported (in comparison with the threshold value), an alert is triggered.

For example:

```
```yaml
missing_storage_measurements:
 description: This event triggers when the number of storage measurements reported falls below the threshold value.
 event_type: deadman
 metric: storage.*
 condition:
 threshold: 0
 granularity: 60
 resource_type:
 flame_sfp: storage
 action:
 implementation:
 - flame_sfemc
```
```

This trigger specification will create an alert task in Kapacitor, which monitors the number of points reported in measurement **storage** and having tag **sfp** set as **storage**. This value is computed every 60 seconds. If the number of reported points is less than or equal to **0** (no points have been reported for the last 60 seconds), an alert will be triggered. Triggered alerts will be sent through an HTTP POST message to the URLs listed in the **implementation** section.

Notes:

- **metric** only requires the measurement name in this event type and doesn't require a field name.
- the trigger specification still needs to be consistent with the parsing rule for **metric**: `<measurement>.<field>`.
- simply putting a `*` for field is sufficient, e.g. `storage.*`.
- even if you put something else for field value, it will be ignored - only the **measurement** name is used
- **aggregation_method** is not required in this event type, any values provided will be ignored.
- **comparison operator** is not required in this event type, any values provided will be ignored.

11.3.2.1 Alert messages

Every alert handler registered in a trigger *action* --> *implementation* section receives an alert message when the trigger event condition is true. This alert message is sent using an *HTTP POST* request to the

URL of the alert handler. The alert message is generated by Kapacitor and, currently, clmc-service has limited control over it. An alert message follows this format:

```
{
  "message": "TRUE",
  "id": "<trigger_id>",
  "level": "CRITICAL",
  "duration": "<integer for the duration of alert - nanoseconds>",
  "previousLevel": "<the previous level of the alert>",
  "details": "<a context string with info on what triggered the alert>",
  "time": "<timestamp of the alert occurrence>",
  "data": {
    "series": [
      {
        "name": "<measurement name of the alert metric>",
        "tags": {
          "<tag name>": "<tag value>"
        },
        "columns": [
          "<column name>"
        ],
        "values": [
          [
            "<values of each column name>"
          ]
        ]
      }
    ]
  }
}
```

- `message` - currently, this is always set to "TRUE"
- `id` - trigger ID as defined in the alert specification document
- `level` - the level of the alert; currently all alerts that trigger have their level set as CRITICAL
- `previousLevel` - the previous level of the alert
- `details` - a string in the format "db=sfc=sfci=policy=" providing context of the alert
- `duration` - integer, duration of the alert in nanoseconds
- `time` - timestamp of the point that triggered the alert
- `data` - describes the point(s) that triggered the alert
- `data.series.name` - the name of the measurement
- `data.series.tags` - (OPTIONAL) key-value pairs for all measurement tags
- `data.series.columns` - list of column names
- `data.series.values` - list of list of values, each nested list represents a measurement point and the values for each column

12 TOSCA BEST PRACTICES

12.1 GENERAL RESOURCE SPECIFICATION

All elements in `<...>`-brackets have to be set by the TOSCA author. Please keep in mind, the `...` only indicate that there may be more elements within the same level.

12.1.1 Header files

12.1.1.1 Definition and Description

At the top of every TOSCA template, we define the `tosca_definitions_version`. After that a description of the deployed service function chain can be inserted. Via `imports` the TOSCA template becomes a FLAME valid template. This has to be specified, otherwise the Orchestrator cannot parse the given instances or policies.

12.1.1.2 Metadata

Within metadata we specify further information for that particular template. A name of the template, the author(s) as well as the template version have to be specified.

A Media Service is constructed using a **Service Function Chain**. The service function chain defines the relationships between service functions. The service function chain has a unique, but arbitrary string which is chosen by the media service provider and represents the entire chain.

Via `servicefunctionchain` (Service Function Chain) we specify the identification of the deployed service chain.

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0
description: |
  <Please insert a template description>

metadata:
  template_name: <Name the given Deployment scenario>
  template_author: <Add the authors>
  template_version: <Versioning>
  servicefunctionchain: <Specify the servicefunctionchain name>

# Import own definitions of nodes, capabilities and policy syntax.
imports:
  - flame_definitions-<version>.yaml

# Starting the template

topology_template:
  node_templates:
    ...
  policies:
    ...
```

12.1.2 Node Template

Nodes require a unique name (the Fully Qualified Domain Name or “FQDN”) to be identified in groups or in policies. Please make sure, only FQDNs with the managed domain suffix can be registered.

Note: The name of the SF package must not be altered after exporting (see packaging procedures).

```
topology_template:
  node_templates:
    ...
    <node name>:
      type: eu.ict-flame.nodes.ServiceFunction
      capabilities:
        host:
          properties:
            num_cpus: <num_cpus>
            mem_size: <mem_size> MB
            disk_size: <disk_size> GB
      properties:
        hypervisor: <kvm,lxc>
        image_url: <image_url>
        fqdn:
          - <FQDN_x>
          - <FQDN_y>
    ...
```

12.1.3 Policies

Policies are inside of topology_template and are written as a list.

12.1.3.1 Initial Policy

The initial policy is of type eu.ict-flame.policies.InitialPolicy and is executed only once, when the template is deployed.

```
topology_template:
  ...
  ## Policies
  policies:
    - <init policy name>:
      type: eu.ict-flame.policies.InitialPolicy
      description: <Describe the initial policy>
      triggers:
        <trigger_name>:
          condition:
            constraint: initialise
          action:
            <node_name_a>:
              <cluster_location_1>: eu.ict-flame.sfe.state.lifecycle.connected
              <cluster_location_2>: eu.ict-flame.sfe.state.lifecycle.booted
            <node_name_b>:
              <cluster_location_2>: eu.ict-flame.sfe.state.lifecycle.connected
            ...
    ...
```

12.1.3.2 StateMachine Policy

The state machine policy allows placement and state change of particular nodes, based on triggers from external sources. The type is `eu.ict-flame.policies.StateChange`.

Note: It is recommended to use only one trigger per policy.

```
topology_template:
...
  policies:
    - <policy_name>:
      type: eu.ict-flame.policies.StateChange
      description: <Describe the policy>
      triggers:
        check_trigger:
          description: <Describe the trigger>
          condition:
            constraint: clmc::<the remote method to be called, returns only true
/false>
            period: <integer value, unit: seconds>
          action:
            <node_name_a>:
              <cluster_location_1>: eu.ict-flame.sfe.state.lifecycle.connected
              <cluster_location_2>: eu.ict-flame.sfe.state.lifecycle.booted
            <node_name_b>:
              <cluster_location_2>: eu.ict-flame.sfe.state.lifecycle.connected
            ...
```

12.1.3.3 Internet Access Policy

This policy is of type `eu.ict-flame.policies.InternetAccessPolicy`. The method determines whitelisting or blacklisting of given FQDNs.

```
topology_template:
...
  policies:
    - allowed_fqdns:
      type: eu.ict-flame.policies.InternetAccessPolicy
      description: <Describe the policy>
      properties:
        method: <permit|prohibit>
        fqdn:
          - <external_FQDN_a>
          - <external_FQDN_b>
          - <external_FQDN_c>
```

12.2 ALERTS SPECIFICATION

12.2.1 Header files

12.2.1.1 Definition and Description

Similar to the resource specification document, the alert configuration document starts with a definition of the TOSCA version, an optional description and an import statement. Do keep in mind

that the FLAME CLMC validator will currently accept the document only if the TOSCA version used is **tosca_simple_profile_for_nfv_1_0_0**. An import is also required (`flame_clmc_alerts_definitions.yaml`) in order to make the TOSCA document a valid alert configuration document. The description is optional.

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0
```

```
description: Alerts configuration for a simple media service.
```

```
imports:
  - flame_clmc_alerts_definitions.yaml
```

12.2.1.2 Metadata

Again, similar to the resource specification document, the alert configuration has a metadata section, where the author of the document specifies the service function chain the document is written for. The author **MUST** ensure that the **servicefunctionchain** metadata matches the values in the metadata of the resource specification document.

```
metadata:
  servicefunctionchain: <service function chain identifier>
```

12.2.2 Policies

The alert triggers and actions are defined withing a list of TOSCA policies. Each policy must be of type **eu.ict-flame.policies.Alert** for the document to be accepted by the CLMC validator.

```
topology_template:

  policies:
    - <policy identifier>:
      type: eu.ict-flame.policies.Alert
      ...
    - <policy identifier>:
      type: eu.ict-flame.policies.Alert
      ...
    - <policy identifier>:
      type: eu.ict-flame.policies.Alert
      ...
    ...
```

12.2.2.1 Alert Policy

Alert policy - **eu.ict-flame.policies.Alert** - is the only type of policy that is allowed in the alerts configuration document. Each state change policy in the resource specification document **MUST** match with an alert policy from the alerts configuration. In addition, the trigger identifier (alert configuration) for such policy must also match with the constraint label (resource specification). The trigger must also have **flame_sfemc** as its entry in the action section so that an alert is properly sent to SFEMC when needed.

12.2.2.2 Threshold trigger

The threshold type defines the simplest type of trigger which compares a given metric with a given value. However, it has two implementations that depend on the granularity period of the trigger, which is how often a check is made. If the granularity period is less than or equal to 60 seconds, then every

measurement that's reported to CLMC is monitored, therefore, the original frequency of checking will depend on how often the measurements are being reported. For periods greater than 60 seconds, an aggregation method must be supplied (default is **mean**), then every X seconds the reported points are aggregated and the result is compared against the threshold.

topology_template:

```

policies:
  - <policy identifier>:
    type: eu.ict-flame.policies.Alert
    triggers:
      <trigger identifier>:
        description: An example threshold trigger; description is optional.
        event_type: threshold
        metric: <measurement>.<field>
        condition:
          threshold: <threshold value>
          granularity: <check period in seconds>
          aggregation_method: <aggregation method - ignored if granularity per
            iod is <= 60 seconds>
          resource_type:
            <CLMC Information Model Tag Name>: <CLMC Information Model Tag Val
              ue>
            ...
          comparison_operator: <comparison operator - gt, gte, lt, lte, eq, ne
            q>
        action:
          implementation:
            - flame_sfemc
            - <custom HTTP endpoint to POST alert messages to>
            ...

```

12.2.2.3 Relative trigger

The relative type defines a trigger, which compares the current aggregated value of a metric against the value this metric had X seconds ago. The difference between the current value and the past value is then compared against a given threshold. Aggregation method must be supplied for this type of trigger, the default being **mean** again. Keep in mind that this event type is query intensive, therefore, choosing a higher granularity period is sensible, e.g. 60 seconds.

topology_template:

```

policies:
  - <policy identifier>:
    type: eu.ict-flame.policies.Alert
    triggers:
      <trigger identifier>:
        description: An example relative trigger; description is optional.
        event_type: relative
        metric: <measurement>.<field>
        condition:
          # the difference between the current aggregated value and the previo
            us aggregated value is compared against this threshold
          threshold: <threshold value>
          granularity: <check period in seconds, also represents how long back
            in time the current value is compared to>

```

```

        aggregation_method: <aggregation method - required, default is 'mean
'>
        resource_type:
            <CLMC Information Model Tag Name>: <CLMC Information Model Tag Val
ue>
        ...
        comparison_operator: <comparison operator - gt, gte, lt, lte, eq, ne
q>
        action:
            implementation:
                - flame_sfemc
                - <custom HTTP endpoint to POST alert messages to>
            ...

```

12.2.2.4 Deadman trigger

The deadman type defines a trigger, which checks the number of measurement points reported to CLMC and compares the result against a given threshold. An alert will be sent if the number of reported points is less than or equal to the threshold value. A good use case is to check if any points are reported at all per given period of time. This trigger type does not require aggregation method or comparison operator. Since the alert will monitor a measurement instead of a specific field, the metric property can be specified in the format <measurement>.* in order to preserve the <measurement>.<field> format.

```

topology_template:
    policies:
        - <policy identifier>:
            type: eu.ict-flame.policies.Alert
            triggers:
                <trigger identifier>:
                    description: An example relative trigger; description is optional.
                    event_type: deadman
                    metric: <measurement>.*
                    condition:
                        threshold: <threshold value>
                        granularity: <check period in seconds, also represents how long the
span in time (in which the number of measurement points are checked) is >
                    resource_type:
                        <CLMC Information Model Tag Name>: <CLMC Information Model Tag Val
ue>
                    ...
                    action:
                        implementation:
                            - flame_sfemc
                            - <custom HTTP endpoint to POST alert messages to>
                        ...

```

12.2.2.5 Resource type

The resource type section can be used to contextualise the alert for a given service function using the CLMC information model tags. For example, by including **flame_sfp**, only measurements for the given service function package will be taken into account.

```

resource_type:
    flame_sfp: storage # only take into account measurements for the 'storage' serv

```

```
ice function package
```

```
...
```

12.2.2.6 Trigger action

The trigger action section defined the HTTP endpoints where an alert message will be POSTed to. If an alert is supposed to be sent to FLAME SFEMC, then use **flame_sfemc** as entry in the implementation. CLMC will automatically generate the required URL for you. Alternatively, an alert could be configured to send the alert messages to custom HTTP endpoints only, instead of sending to SFEMC, too - simply exclude **flame_sfemc** from the list of entries.

```
action:
```

```
  implementation:
```

```
    - flame_sfemc
```

```
    - <custom HTTP endpoint to POST alert messages to - e.g. http://sfc-A.ict-flame.eu/alerts>
```

```
...
```



13 FLAME CLMC SERVICE DOCUMENTATION

13.1 AUTHORS

Authors

Nikolay Stanchev

Organisation

University of Southampton, IT Innovation Centre

13.2 DESCRIPTION

This document describes the CLMC service and its API endpoints. The CLMC service is implemented using the *Python* web framework called **Pyramid**. It offers different API endpoints such as GraphAPI for calculating round trip time, CRUD API for service function endpoints configuration data and Alerts API for creating and subscribing to alerts in Kapacitor. All source code, tests and configuration files of the service can be found in the **src/service** folder.

13.3 NOTES

- Interacting with *Chronograf* - use `http://<clmc-host>/chronograf`. You will be asked to enter connection details. The only input that you need to edit is the *Connection String* - set it to `http://<clmc-host>:8086` and click the **Add Source** button.
- Interacting with *Kapacitor* - the Kapacitor HTTP API documentation can be found here: <https://docs.influxdata.com/kapacitor/v1.4/working/api/> Notice that all of the URL paths provided by Kapacitor are already namespaced using base path **/kapacitor/v1**. Therefore, no other prefix is required when interacting with the Kapacitor application running on the clmc container, e.g. `http://<clmc-host>/kapacitor/v1/tasks` as described in the Kapacitor API reference.
- Interacting with *InfluxDB* - the InfluxDB HTTP API documentation can be found here: <https://docs.influxdata.com/influxdb/v1.5/tools/api/> In order to interact with the InfluxDB application running on the clmc container, prefix all URL paths in the documentation with **/influxdb**, e.g. `http://<clmc-host>/influxdb/query`
- Interacting with *neo4j* - use `http://<clmc-host>/neo4j/browser/`. This will open the neo4j browser, which lets you interact with the graph using Cypher queries (if necessary).
- Interacting with *clmc-service* - the API endpoints listed in the following sections relate to direct interactions with the clmc-service application server (listening on port 9080). If interacting with the clmc container, all of the listed below URIs must be prefixed with **/clmc-service** so that the nginx reverse proxy server (listening on port 80) can forward to requests to the correct application, e.g. `http://<clmc-host>/clmc-service/alerts?sfc={service function chain id}&sfcid={service function chain instance id}&policy={policy id}&trigger={trigger id}`.

13.4 ALERTS API ENDPOINTS

- **GET /alerts?sfc={service function chain id}&sfc={service function chain instance id}&policy={policy id}&trigger={trigger id}**

This API method can be used to retrieve the generated alert task and alert topic identifiers during the processing of an alerts specification document. These identifiers can then be used to interact with the Kapacitor HTTP API for further configuration or modification of alerts - <https://docs.influxdata.com/kapacitor/v1.4/working/api/>.

- Request:

Expects a URL query string with the request parameters - **sfc**, **sfc**, **policy** and **trigger**. The given parameters must match the values used in the alerts specification document. Otherwise, a wrong ID will be returned.

- Request URL Examples:

/alerts?sfc=MSDemo&sfc=MSDemo-premium&policy=requests_diff&trigger=low_requests

/alerts?sfc=SimpleMediaService&sfc=SimpleMediaService-1&policy=rtt_deviation&trigger=increase_in_rtt

- Response

The response of this request is a JSON-formatted content, which contains the task and topic identifiers, along with the Kapacitor API endpoints to use for configuring the given task, topic and the respective handlers.

Returns a 400 Bad Request if the URL query string parameters are invalid or otherwise incorrect.

- Response Body Example:

```
{
  "task_identifier": "094f23d6e948c78e9fa215528973fb3aeefa5525898626c9ea049dc8e87a7388",
  "topic_identifier": "094f23d6e948c78e9fa215528973fb3aeefa5525898626c9ea049dc8e87a7388",
  "task_api_endpoint": "/kapacitor/v1/tasks/094f23d6e948c78e9fa215528973fb3aeefa5525898626c9ea049dc8e87a7388",
  "topic_api_endpoint": "/kapacitor/v1/alerts/topics/094f23d6e948c78e9fa215528973fb3aeefa5525898626c9ea049dc8e87a7388",
  "topic_handlers_api_endpoint": "/kapacitor/v1/alerts/topics/094f23d6e948c78e9fa215528973fb3aeefa5525898626c9ea049dc8e87a7388/handlers"
}
```

- **POST /alerts**

This API method can be used to send an alert specification document, which is then used by the CLMC service to create alert tasks and subscribe alert handlers to those tasks in Kapacitor. For further information on the alert specification document, please check the [CLMC Alert Specification Documentation](#).

- Request:

Expects two YAML-formatted files in the request - one referenced with ID **alert-spec** representing the TOSCA alert specification document and one referenced with ID **resource-spec** representing the TOSCA resource specification document. The alert specification document is then parsed with the openstack TOSCA parser (<https://github.com/openstack/tosca-parser/tree/master/toscaparser>) and validated against the CLMC alerts specification schema (again check [documentation](#) for more info on this). The TOSCA resource specification document is used only for consistency verification between the two documents - ensuring that they refer to the same service function chain, as well as making sure that there is at least one trigger alert in the alerts specification that relates to a state change policy in the resources specification.

- Example for sending a request with curl:

```
curl -F "alert-spec=@alert-specification.yaml" -F "resource-spec=@resource-specification.yaml" http://localhost:9080/alerts
```

where **alert-specification.yaml** is the path to the alerts specification file and **resource-specification.yaml** is the path to the resource specification file.

- Response:

The response of this request is a JSON-formatted content, which contains the SFC and SFC instance identifiers from the alert specification along with any errors encountered when interacting with Kapacitor.

Returns a 400 Bad Request if the request does not contain a yaml file referenced with ID **resource-spec**.

Returns a 400 Bad Request if the resource specification file is not a TOSCA-compliant valid YAML file.

Returns a 400 Bad Request if the request does not contain a yaml file referenced with ID **alert-spec**.

Returns a 400 Bad Request if the alert specification file is not a valid YAML file.

Returns a 400 Bad Request if the alert specification file cannot be parsed with the TOSCA parser.

Returns a 400 Bad Request if the alert specification file fails validation against the CLMC alerts specification schema.

Returns a 400 Bad Request if there are inconsistencies between the alert specification and resource specification files - e.g. referring to different service function chain and service function chain instance identifier or if there is no alert in the alerts specification related to a given state change policy in the resources specification.

- Response Body Example:

```
{
  "msg": "Alerts specification has been successfully validated and forwarded to Kapacitor",
}
```

```

    "service_function_chain_id": "<sfc_id>",
    "service_function_chain_instance_id": "<sfc_instance_id>"
  }

```

If the CLMC service encounters any errors when creating alerts and handlers in Kapacitor, they will be reported in the response as two lists of error objects. The **triggers_specification_errors** list contains any errors encountered while trying to create the alert tasks; the **triggers_action_errors** list contains any errors encountered while trying to subscribe the HTTP handlers to the created tasks.

```

{
  "msg": "Alerts specification has been successfully validated and forwarded to Kapacitor",
  "service_function_chain_id": "<sfc_id>",
  "service_function_chain_instance_id": "<sfc_instance_id>",
  "triggers_action_errors": [
    {
      "trigger": "<trigger ID the error is related to>",
      "handler": "<handler URL the error is related to>",
      "policy": "<policy ID the error is related to>",
      "error": "<error message returned from Kapacitor>"
    }
  ],
  "triggers_specification_errors": [
    {
      "trigger": "<trigger ID the error is related to>",
      "policy": "<policy ID the error is related to>",
      "error": "<error message returned from Kapacitor>"
    }
  ]
}

```

13.5 GRAPH API ENDPOINTS

- **Assumptions**

- For each service function, there is a field/fields from which the service function response time (service delay) can be derived.
- For each service function, there is a field/fields from which an average estimation of the size of a **request** to this service function can be derived.
- For each service function, there is a field/fields from which an average estimation of the size of a **response** from this service function can be derived.
- All the aforementioned fields reside in a single measurement.

- **POST /graph/temporal?from={timestamp-seconds}&to={timestamp-seconds}**

This API method sends a request to the CLMC service to build a graph related to the time range between the *from* and *to* timestamps (URL query parameters).

- **Request:**

Expects a JSON-formatted request body which declares the service function chain and service function chain instance for which the graph is built. The request should also include the service functions that must be included in the graph along with the measurement name, response time field, request size field and response size field for each service function. The declared fields could be in flux functions across multiple fields.

- Request Body Example:

```
```json
{
 "service_function_chain": "MSDemo",
 "service_function_chain_instance": "MSDemo_1",
 "service_functions": {
 "nginx": {
 "response_time_field": "mean(response_time)",
 "request_size_field": "mean(request_size)",
 "response_size_field": "mean(response_size)",
 "measurement_name": "nginx"
 },
 "minio": {
 "response_time_field": "mean(sum)/mean(count)",
 "request_size_field": "mean(request_size)/mean(count)",
 "response_size_field": "mean(response_size)/mean(count)",
 "measurement_name": "minio_http_requests_duration_seconds"
 }
 }
}
```
```

These parameters are then filled in the following influx query template:

```
```
SELECT {0} AS mean_response_time, {1} AS mean_request_size, {2} AS mean_response_size FROM "{3}"."{4}".$5 WHERE "flame_sfc"='\{6}\' and "flame_sfci"='\{7}\' and time>={8} and time<{9} GROUP BY "flame_sfe", "flame_location", "flame_sf"
```
```

E.g. for the minio service function, the following query will be used to retrieve the data from influx (request url is `*/graph/build?from=1528385420&to=1528385860*`):

```
```
SELECT mean(sum)/mean(count) AS mean_response_time, mean(request_size)/mean(count) AS mean_request_size, mean(response_size)/mean(count) AS mean_response_size FROM "MSDemo"."autogen".minio_http_requests_duration_seconds WHERE "flame_sfc"='MSDemo' and "flame_sfci"='MSDemo_1' and time>=1528385420000000000 and time<1528385860000000000 GROUP BY "flame_sfe", "flame_location", "flame_sf"
```
```

N.B. database name is assumed to be the SFC identifier
N.B. timestamps are converted to nano seconds.

- Response:

The response of this request is a JSON content, which contains all request parameters used to build the graph, along with a request UUID. This request ID can then be used to manage the temporal subgraph that was created in response to this request.

Returns a 400 Bad Request error if the request body is invalid.

Returns a 400 Bad Request error if the request URL parameters are invalid or missing.

- Response Body Example:

```
```json
{
 "database": "MSDemo",
 "retention_policy": "autogen",
 "service_function_chain": "MSDemo",
 "service_function_chain_instance": "MSDemo_1",
 "service_functions": {
 "nginx": {
 "response_time_field": "mean(response_time)",
 "request_size_field": "mean(request_size)",
 "response_size_field": "mean(response_size)",
 "measurement_name": "nginx"
 },
 "minio": {
 "response_time_field": "mean(sum)/mean(count)",
 "request_size_field": "mean(request_size)/mean(count)",
 "response_size_field": "mean(response_size)/mean(count)",
 "measurement_name": "minio_http_requests_duration_seconds"
 }
 },
 "graph": {
 "uuid": "75df6f8d-3829-4fd8-a3e6-b3e917010141",
 "time_range": {
 "from": 1528385420,
 "to": 1528385860
 }
 }
}
```
```

- **DELETE /graph/temporal/{graph_id}**

This API method sends a request to delete the temporal graph associated with a given request UUID (retrieved from the response of a build-graph request). The request UUID must be given in the request URL, e.g. request sent to */graph/temporal/75df6f8d-3829-4fd8-a3e6-b3e917010141*

- Response:

The response of this request is a JSON content, which contains the request UUID and the number of deleted nodes.

Returns a 404 Not Found error if the request UUID is not associated with any nodes in the graph.

- Response Body Example:

```
```json
{
 "uuid": "75df6f8d-3829-4fd8-a3e6-b3e917010141",
 "deleted": 5
}
```
```

- **GET /graph/temporal/{graph_id}/round-trip-time?compute_node={compute_node_id}&endpoint={endpoint_id}**

This API method sends a request to run the Cypher Round-Trip-Time query over a temporal graph associated with a request UUID (retrieved from the response of a build-graph request). The request UUID must be given in the request URL, e.g. request sent to */graph/temporal/75df6f8d-3829-4fd8-a3e6-b3e917010141/round-trip-time?compute_node=DC2&endpoint=minio_1_ep1*

- Response:

The response of this request is a JSON content, which contains the result from the Cypher query including forward latencies, reverse latencies and service function response time along with the calculated round trip time and global tag values for the given service function endpoint.

Returns a 400 Bad Request error if the URL parameters are invalid

Returns a 404 Not Found error if the request UUID and the endpoint ID are not associated with an endpoint node in the graph.

Returns a 404 Not Found error if the compute node ID is not associated with a compute node in the graph.

- Response Body Example:

```
```json
{
 "request_size": 2048,
 "response_size": 104857,
 "bandwidth": 104857600,
 "forward_latencies": [
 22, 11
],
 "total_forward_latency": 33,
 "reverse_latencies": [
 15, 18
],
 "total_reverse_latency": 33,
 "response_time": 15.75,
 "round_trip_time": 81.75,
 "global_tags": {
 "flame_sfe": "minio_1_ep1",
 "flame_sfc": "MSDemo",
 "flame_sfci": "MSDemo_1",
 "flame_sfp": "minio",
 "flame_sf": "minio_1",
 }
}
```

```

 "flame_location": "DC1",
 "flame_server": "DC1"
 }
}
...

```

Here, the *\*forward\_latencies\** and *\*reverse\_latencies\** lists represent the latency experienced at each hop between compute nodes. For example, if the path was DC2-DC3-DC4 and the SF endpoint was hosted on DC4, the response data shows that  $\text{latency}(\text{DC2-DC3}) = 22$ ,  $\text{latency}(\text{DC3-DC4}) = 11$ ,  $\text{latency}(\text{DC4-DC3}) = 15$ ,  $\text{latency}(\text{DC3-DC2}) = 18$ ,  $\text{response\_time}(\text{minio\_1\_ep1}) = 15.75$

N.B. if the endpoint is hosted on the compute node identified in the URL parameter, then there will be no network hops between compute nodes, so the latency lists would be empty, example:

```

```json
{
  "request_size": 2048,
  "response_size": 104857,
  "bandwidth": 104857600,
  "forward_latencies": [],
  "total_forward_latency": 0,
  "reverse_latencies": [],
  "total_reverse_latency": 0,
  "response_time": 3,
  "round_trip_time": 3,
  "global_tags": {
    "flame_sfe": "minio_1_ep1",
    "flame_sfc": "MSDemo",
    "flame_sfci": "MSDemo_1",
    "flame_sfp": "minio",
    "flame_sf": "minio_1",
    "flame_location": "DC1",
    "flame_server": "DC1"
  }
}
...

```

- Generating network measurements

To generate network measurements, which are then used to create the network topology in the Neo4j graph, refer to the `src/service/clmcservice/generate_network_measurements.py` script. An example configuration file is `src/service/resources/GraphAPI/network_config.json`

13.6 CRUD API FOR SERVICE FUNCTION ENDPOINT CONFIGURATIONS

Note: this API is experimental and is not intended to be used at this stage

- **GET /whoami/endpoints**

This API method retrieves all service function endpoint configurations in a JSON format.

– Response:

Returns a JSON-formatted response - a list of JSON objects, each object representing a service function endpoint configuration.

– Response Body Example:

- No service function endpoint configurations found.

```
[]
```

- Multiple service function endpoint configurations found.

```
[
  {
    "location": "location_1",
    "server": "location_1",
    "sfc": "sfc_1",
    "sfc_instance": "sfc_i_1",
    "sf_package": "sf_1",
    "sf": "sf_i_1",
    "sf_endpoint": "sf_endpoint_1"
  },
  {
    "location": "location_2",
    "server": "location_2",
    "sfc": "sfc_2",
    "sfc_instance": "sfc_i_2",
    "sf_package": "sf_2",
    "sf": "sf_i_2",
    "sf_endpoint": "sf_endpoint_2"
  }
]
```

- **GET /whoami/endpoints/instance?sf_endpoint={sf_endpoint_id}**

This API method retrieves the uniquely defined service function endpoint configuration associated with the given URL parameter - sf_endpoint.

– Response:

Returns a JSON-formatted response - a JSON object representing the service function endpoint configuration if it exists.

Returns a 404 Not Found error if there is no service function endpoint configuration associated with the given URL parameter.

Returns a 400 Bad Request error if the url parameter is invalid or missing.

– Response Body Example:

- Request made to /whoami/endpoints/instance?sf_endpoint=sf_endpoint_1

```
{
  "location": "location_1",
  "server": "location_1",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
```

```

    "sf": "sf_i_1",
    "sf_endpoint": "sf_endpoint_1"
  }

```

- **POST /whoami/endpoints**

This API method creates a new service function endpoint configuration.

- Request:

Expects a JSON-formatted request body with the new service function endpoint configuration.

- Request Body Example:

```

{
  "location": "location_1",
  "server": "location_1",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
  "sf": "sf_i_1",
  "sf_endpoint": "sf_endpoint_1"
}

```

- Response

Returns a JSON-formatted response - a JSON object representing the service function endpoint configuration that was created.

Returns a 400 Bad Request error if the request body is invalid.

Returns a 409 Conflict error if there exists another service function endpoint configuration with the same 'sf_endpoint' ID.

- Response Body Example:

```

{
  "location": "location_1",
  "server": "location_1",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
  "sf": "sf_i_1",
  "sf_endpoint": "sf_endpoint_1"
}

```

- **PUT /whoami/endpoints/instance?sf_endpoint={sf_endpoint_id}**

This API method replaces the uniquely defined service function endpoint configuration associated with the given URL parameter - sf_endpoint, with a new service function endpoint configuration given in the request body (JSON format). It can also be used for updating.

- Request:

Expects a JSON-formatted request body with the new service function endpoint configuration.

– Request Body Example:

```
{
  "location": "location_2",
  "server": "location_2",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
  "sf": "sf_i_1",
  "sf_endpoint": "sf_endpoint_1"
}
```

– Response

Returns a JSON-formatted response - a JSON object representing the new service function endpoint configuration that was created (updated).

Returns a 400 Bad Request error if the request body is invalid.

Returns a 400 Bad Request error if the url parameter is invalid.

Returns an 404 Not Found error if there is no service function endpoint configuration associated with the given URL parameter.

Returns a 409 Conflict error if there exists another service function endpoint configuration with the same 'sf_endpoint' ID as the ones in the request body.

– Response Body Example:

- Request made to `/whoami/endpoints/instance?sf_endpoint=sf_endpoint_1`

```
{
  "location": "location_2",
  "server": "location_2",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
  "sf": "sf_i_1",
  "sf_endpoint": "sf_endpoint_1"
}
```

- **DELETE `/whoami/endpoints/instance?sf_endpoint={sf_endpoint_id}`**

This API method deletes the uniquely defined service function endpoint configuration associated with the given URL parameter - `sf_endpoint`.

– Response:

Returns the JSON representation of the deleted object.

Returns an 404 Not Found error if there is no service function endpoint configuration associated with the given URL parameter.

Returns a 400 Bad Request error if the url parameter is invalid.

– Response Body Example:

- Request made to `/whoami/endpoints/instance?sf_endpoint=sf_endpoint_1`

```
{
  "location": "location_1",
  "server": "location_1",
  "sfc": "sfc_1",
  "sfc_instance": "sfc_i_1",
  "sf_package": "sf_1",
  "sf": "sf_i_1",
  "sf_endpoint": "sf_endpoint_1"
}
```



14 CONCLUSIONS

We have presented a snapshot of the extensive documentation available to FLAME's experimenters. The documentation is kept online alongside the source code of the sub-components in FLAME GitLab source control system and as such is frequently updated with improvements and clarifications (from any partner) along with changes to keep the documentation in synch with the developing software.

