Multi-provider capabilities in EnOSlib: driving distributed system experiments on the edge-to-cloud continuum

Baptiste Jonglez[1 [0000−0001−5434−6048]], Matthieu Simonin[2 [0000−0002−9063−0334]], Jolan Philippe[2 [0000−0001−8759−4566]], and Sidi Mohammed Kaddour[1 [0000−0002−2745−4024]]

[1] Nantes Université, Ecole Centrale Nantes, IMT Atlantique, CNRS, Inria, LS2N, UMR 6004, F-44000 Nantes, France
[2] University of Rennes, Inria, CNRS, IRISA, UMR 6074 - Rennes, France
{baptiste.jonglez,matthieu.simonin,jolan.philippe,sidi-mohammed.kaddour}@inria.fr

Abstract. This paper introduces recent advances in EnOSlib, a Python library that aims at facilitating the design and execution of reproducible experiments across distributed computing infrastructures. Originally developed to simplify experimentation on testbeds such as Grid'5000 and Chameleon Cloud, EnOSlib now incorporates support multi-provider deployments, including access to edge resources, as well as advanced services.
Key contributions include integration with Kwollect for fine-grained energy measurements, a planning service for executing timed events, and enhanced network emulation functionalities. These features enable users to model and study complex, realistic scenarios such as latency-sensitive edge-to-cloud applications.
A major new capability is the support for synchronized multi-infrastructure experiments, allowing simultaneous resource reservation and deployment across diverse testbeds. The paper illustrates these capabilities through a distributed video processing use case spanning edge and cloud platforms. This paper is the companion paper of the tutorial presented in DAIS 2025.

Keywords: Experiment-driven research · Performance evaluation · Multiple infrastructure · Distributed computing experimentation library.

1 Introduction

This paper is the companion paper of a tutorial presented in DAIS 2025. The tutorial showcases the EnOSlib [8] library: a largely adopted Python library[3] to ease distributed experimentation on different testbeds [7,9,12,16,18,17,20,21], and which is now also adopted by other frameworks [22,23]. We highlight how experiments for an edge-to-cloud use case can be expressed and run with EnOSlib.

[3] https://discovery.gitlabpages.inria.fr/enoslib/theyuseit.html

EnOSlib targets established research testbeds such as Grid'5000 [6], FIT IoT-LAB [3] and Chameleon Cloud [15], and is also planning to interface with new testbeds like SLICES [2] or FABRIC [5]. The experimenter can leverage EnOSlib to obtain *resources* on these testbeds, such as bare-metal servers, virtual machines, IoT devices, isolated networks... It is then possible to develop and automate complete distributed experiments using these resources. As illustrated in Figure 1, the key steps EnOSlib aims at dealing with are: (i) provisioning resources on which to run experiments, (ii) the management of these resources (e.g. configuration management, software setup), (iii) benchmarking and obtaining metrics from the experiments, and (iv) cleaning and/or destroying used resources.

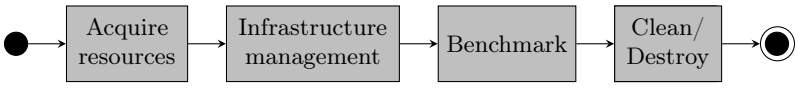

Fig. 1: General experimental workflow with EnOSlib.

Beyond the initial deployment of an application, EnOSlib facilitates the study of distributed systems by providing out-of-the box observability tools (e.g. off-the-shelves monitoring stacks) and variability injection functionalities (e.g. using state-of-art network emulation features). In the following we remind the core abstractions behind EnOSlib: *Provider* and *Service*, as well as an important design choice: *idempotency*.

Provider. At the EnOSlib's core lies the `Provider` abstraction. Providers connect to the various infrastructure to claim resources from. EnOSlib supports two types of resources: compute servers and networks (IPv4 or IPv6). In other words, a provider can be seen as a function that transforms an abstract resource description into a concrete set of `Host`s and `Network`s with the side effect of actually claiming the corresponding resource on the platforms. An example of user script is given in Fig. 2. The `Host` and `Network` object are generic in the sense that a user can interact the same way with those objects regardless of their provenance. This allows *decoupling* between hardware resources specification (different for each platform and each situation to evaluate) and the actual actions to perform during the experiments (common and generic). In practice, that means that an experimentation can be transferred to another testbed at the cost of changing only the provider used, assuming that the experiment doesn't rely on testbed-specific services.

EnOSlib is shipped with providers for Grid'5000 (bare-metal and virtual machines), Chameleon (bare-metal, cloud and edge containers), IoT-LAB (embedded IoT devices), Vagrant (virtual machines on the local machine). The provider abstraction has been extended to optionally let the users get resources on different testbeds *at the same time*: this new feature is presented in Section 2.3.

```
provider_conf = {
  "job_name": "myjobname",
  "resources": {
    "machines": [{
      "roles": ["server"],
      "cluster": "paradoxe",
      "nodes": 1,
      }, {
      "roles": ["client"],
      "cluster": "paradoxe",
      "nodes": 3,
      }
    ],
  }
}
provider = (
  en.G5k.from_dictionary(conf)
)
roles, nets = provider.init()
```

```
provider_conf = {
  "lease_name": "mylease",
  "resources": {
    "machines": [{
      "roles": ["server"],
      "flavour": "compute_skylake",
      "number": 1,
      }, {
      "roles": ["client"],
      "flavour": "compute_skylake",
      "number": 3,
      }
    ],
  }
}
conf = (
  en.CBMConf
    .from_dictionary(provider_conf)
)
provider = en.CBM(conf)
roles, nets = provider.init()
```

Fig. 2: Resources specification and grouping resources into *generic roles* on Grid'5000 (left) and Chameleon (right). The `roles` returned by the provider can be queried to get a group of hosts. For instance `roles["client"]` returns a set of 3 `Hosts` corresponding to 3 (concrete) servers labelled as `client` by the user.

Services. A `Service` is a high-level construction that relies on the resource abstraction to provide reusable "units of behavior". It is instantiated through a Python function call that enforces its configuration as a side effect. Concretely, a `Service` bootstraps classical software stacks on the resources and hides the low-level details of their deployment to the experimenter (although experimenters have to feed the service with specific inputs described in its interface). EnOSlib offers a growing set of `Services` based on user needs. It includes several *Observability* `Services` that can deploy various monitoring/tracing stacks depending on the user needs. *Network emulation* `Services` can be used to get control over bandwidth and latency of network interface. *Software stacks* `Services` are able to deploy complex software stack (e.g. Kubernetes). A recently added `Service` is described in Section 2.2.

Idempotency. Idempotency refers to the property of an operation: applying it multiple times must give the same output as applying it a single time. This property has been popularized by Ansible in the context of configuration management systems. Idempotency is a key property to support repeatability and robustness in experimental workflows. It also facilitates iterative experiment development using Jupyter Notebooks, since a given block of code can be re-executed as many

times as necessary. EnOSlib ensures a first level of idempotency internally, and also relies on Ansible modules to help the experimenter write idempotent code. An example of user script to perform actions is given in Fig. 3.

```python
# On the server: generate a configuration file
with play_on(pattern_hosts="server") as p:
    p.shell("echo 'parameter=value' > /tmp/my_config.conf")

# Copy the config file to all clients
server_host = roles["server"][0].address # get the server IP
with play_on(pattern_hosts="client") as p:
    # Fetch the config file from the server
    p.copy(
        src=f"{server_host}:/tmp/my_config.conf",
        dest="/tmp/my_config.conf",
        remote_src=True
    )

# On each client: run a command using the config file
with play_on(pattern_hosts="client") as p:
    p.command("./my_program --config /tmp/my_config.conf")
```

Fig. 3: Actions on resource described in Fig. 2 using the *generic roles*. Since roles are defined for both resources on Grid'5000 and Chameleon Cloud, triggering actions on distant machines is expressed in the same way. In this example, the server generates a configuration file before copying it on each client. Then, the clients use the copied file as input for a local program. Behind the scenes, the `play_on` invocation generates an Ansible playbook, giving access to all standard Ansible modules (`shell`, `copy`...), encouraging experimenters to write idempotent code.

The paper is organized as follows. First, Section 2 introduces the new features added in EnOSlib since the reference publication [8]. In Section 3, the use case for the tutorial is presented, and the methodology to experiment on it is presented in Section 4. Section 5 and Section 6 respectively present how to deploy the experiment on a multi-site and on a multi-provider context. Concluding remarks are given in Section 7.

2 New EnOSlib features

Building upon the concepts introduced in earlier work on EnOSlib [8], this section presents recent advancements that expand the library features. Notable additions include (i) a support for *metrics collection* with Kwollect [10] (Sect. 2.1); (ii) a *planning service* for mocking events allowing their reproducibility (Sect. 2.2),

and (iii) *multi-provider* functionalities enabling seamless experiments across heterogeneous testbeds (Sect. 2.3). Beyond the core library, EnOSlib can also take advantage of Jupyter notebooks to interactively design, execute, and visualize results, including infrastructure introspection and live data previews (Sect. 2.4).

2.1 Metrics collection with the Kwollect Service

Energy efficiency is becoming more and more critical when evaluating algorithms and distributed systems. While software-based energy measurement tools such as PowerAPI [11], Kepler [4], Scaphandre [1], or PowerJoular [19] have been growing in popularity, they rely on specific vendor functionalities (e.g. Intel RAPL) or estimation models (e.g. regression models based on CPU usage). As a result, they have limited precision and frequency, and they cannot measure the energy impact of some components such as the physical disks or the Power Supplies Units (PSU).

Fine-grained energy measurements at the physical level allow to fully evaluate and compare the energy efficiency of different algorithms on a given piece of hardware; alternatively, the energy consumption of a single algorithm can be measured on different hardware (for example with and without GPU). Most interestingly, high-frequency energy measurements open interesting use-cases such as measuring the energy impact of memory transfers or disk I/O operations.

EnOSlib supports experiments requiring accurate and fine-grained energy measurement through integration with Kwollect [10], a monitoring solution available on the Grid'5000 platform [6]. Kwollect continuously polls physical wattmetres at high frequency and exposes collected data through an API. It also collects power consumption metrics from other physical sources such as Power Distribution Units (PDUs), or the network traffic from network equipment. Beyond Kwollect, other energy monitoring systems could be integrated in EnOSlib as long as they provide an API that EnOSlib can query.

In practice, the experimenter specifies which sections of the experiment should be monitored for energy consumption, and then EnOSlib retrieves energy traces from the Kwollect API using the correct time ranges.

In keeping with the general philosophy of EnOSlib as a library, the rest of the experiment is the responsibility of the experimenter: running the actual workload, and performing data analysis on the energy measurement data. Example code is show in Figure 4 with resulting data show in Figure 5.

2.2 Events planning

Distributed deployments in the wild are subject to various events like the loss of connectivity between communicating processes, nodes failures, additions/removal of resource capabilities, transient resource limitations, ... Distributed systems experiments is also about studying the behaviour of systems in the face of such events. EnOSlib exposes a `Planning` service allowing the experimenter to schedule events at a specific time. This serves two purposes (1) *expressivity*: a user can easily describe a sequence of events and (2) *reproducibility*: the same scenario

```python
# Allocate resources (here, physical servers on G5K)
conf = en.G5kConf() \
    .add_machine(roles=["server"], cluster="nova", nodes=1)
    .add_machine(roles=["server"], cluster="taurus", nodes=1)

# Setup monitoring API
monitor = en.Kwollect(nodes=roles["server"])
monitor.deploy()

# Run a loop of stress tests under the monitor
monitor.start()
duration = 2
time.sleep(duration)
for cpu_cores in [1, 2, 8, 14, 20, 26, 32]:
    en.run_command(f"stress-ng --cpu {cpu_cores} -t {duration}",
                   roles=roles["server"])
time.sleep(duration)
monitor.stop()

# Fetch monitoring data from Kwollect API between start and stop
data = monitor.get_metrics(metrics=["wattmetre_power_watt"])
```

Fig. 4: Example EnOSlib code that collects power measurement from Kwollect during a CPU stress test. The result is shown in Figure 5

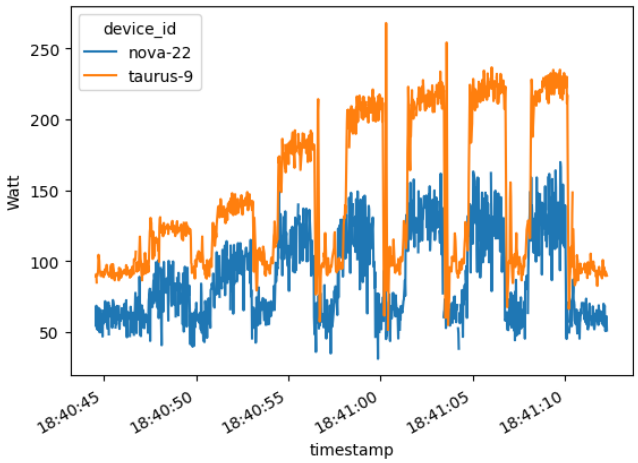

Fig. 5: Power consumption result obtained with EnOSlib using physical wattmetres (two different physical servers during a CPU stress test)

can be replayed. In EnOSlib an event is an action to run at a specific date on a specific set of hosts. Different types of events are currently supported and showcased in Fig. 6: `StartEvent` that are used to start a process given a command line, `KillEvent` that is used to terminate a set of processes (e.g. to simulate process crashes) and a `CGroupEvent` used to schedule `cgroup` changes (e.g. to throttle some resources: CPU, IOs ...). The `Planning` service of EnOSlib is a collection of events and is responsible to run the various actions in a timely manner on a distributed set of Hosts. This service is inspired by MockFog [13] which used such approach to evaluate edge use cases: the EnOSlib `Planning` service makes the approach re-usable for other experiments.

```
1  ps = en.PlanningService()
2
3  ps.add_event(
4    en.StartEvent(
5      date=t1
6      cmd="stress -c 30",
7      host=roles["groupA"],
8      name=f"mysleep"
9    )
10 )
11
12 ps.add_event(
13   en.CGroupEvent(
14     date=t2
15     cpath="cpuset.cpus",
16     value="1-10",
17     host=roles["groupA"],
18     name=f"mysleep"
19   )
20 )
30 ps.add_event(
31   en.CGroupEvent(
32     date=t3
33     cpath="cpuset.cpus",
34     value="0-31",
35     host=roles["groupA"],
36     name=f"mysleep"
37   )
38 )
39
40 # deploy the planning
41 # while monitoring the usage using dstat
42 with en.Dstat(nodes=roles["groupA"]):
43   ps.deploy()
44   # waiting a bit
45   time.sleep(
46     ps.until_end.total_seconds() + 60
47   )
```

Fig. 6: Pseudo code showing the use of the planning service. Line 1 to 38, the planning service is fed with some events: a stress process will be started at `t1`, the available cores to this process identified by its name will be then reduced from time `t2` to `t3`. Finally the user can draw the CPU usage during the execution of the planning thanks to the Dstat monitoring service that tracks some basic system metrics.

2.3 Experiment over multiple infrastructures

Experiments in the Fog/Edge context require heterogeneous resources (e.g. mixing IOT devices and large computing servers) and scalability (e.g. getting lot of the same resources). *Diversity* and *scalability* makes thus some experiments hard to perform on a single testbed. The provider abstraction offered by EnOSlib can

be a solution since a single user script can embed calls to different providers. However this approach falls short when it comes to get the resources over different testbeds *at the same time*. Indeed depending on the platform status, resources might not be available or delayed. EnOSlib offers an elegant way to get resources over different platforms in a synchronized way: the *multi-provider* abstraction. The *multi-provider* abstraction is a new provider that ensures the resources over different platforms are acquired and released at the same time. This releases the user from the burden of writing the synchronization code by focusing on the experimentation logic. A user script is depicted in Fig. 7.

The synchronization algorithm consists in (1) querying each provider for a given time slot; if all providers agree on the time slot, then (2a) the slots are actually reserved; otherwise (2b) another time slot is tested. Testing a time-slot is provider-specific (some infrastructure expose the past and future resource status which can be leveraged to know in advance if a candidate time slot is actually possible). There is an obvious race condition between the step (1) and (2a) or (2b) since the state of the platform might have changed between the two phases. In the general case it is not a real problem especially when dealing with reservation far enough in the future.

```python
# Naive multi-provider code.
# This is not robust.
import enoslib as en

g5k = en.G5k(conf_g5k)
roles_g5k, nets_g5k = g5k.init()

iot = en.Iotlab(iot_conf)
roles_iot, nets_iot = iot.init()

vms = en.Vagrant(vm_conf)
roles_vms, nets_vms = vms.init()
```

```python
# Built-in multi-providers support.
# It ensures synchronized reservations.
import enoslib as en

g5k = en.G5k(conf_g5k)
iot = en.Iotlab(iot_conf)
vagrant = en.Vagrant(vagrant_conf)

roles, networks = en.Providers([
    g5k,
    iot_lab,
    vagrant
])
```

Fig. 7: Pseudo-codes illustrating the multi-provider use case.
On the left the user uses three independent providers to get her resources on Grid'5000, IoT-LAB and local machine (Vagrant virtual machines). However resources may not be available or delayed on Grid'5000 or IoT-LAB due to the current platform availability.
On the right the EnOSlib `Providers` instance can deal with different platforms, get resources in a synchronized way and return them as a regular `Provider`

2.4 Jupyter Integration

EnOSlib integrates with Jupyter notebooks, enabling users to interactively design, execute, and monitor infrastructure and experiments. This integration fa-

cilitates real-time introspection, live data visualization, and step-by-step experiment control, enhancing user experience.

To effectively display complex outputs within Jupyter notebooks, such as tables or plots, appropriate formatting is necessary. This may involve using specific display functions or formatting outputs as HTML or Markdown to ensure correct rendering. Notebook examples demonstrating these capabilities is available online.[4]

3 Edge-to-cloud use-case: distributed video processing

We will consider a common use-case throughout the tutorial: an edge-to-cloud video processing application designed to detect roaming animals[5]. Its high-level architecture is depicted in Figure 8.

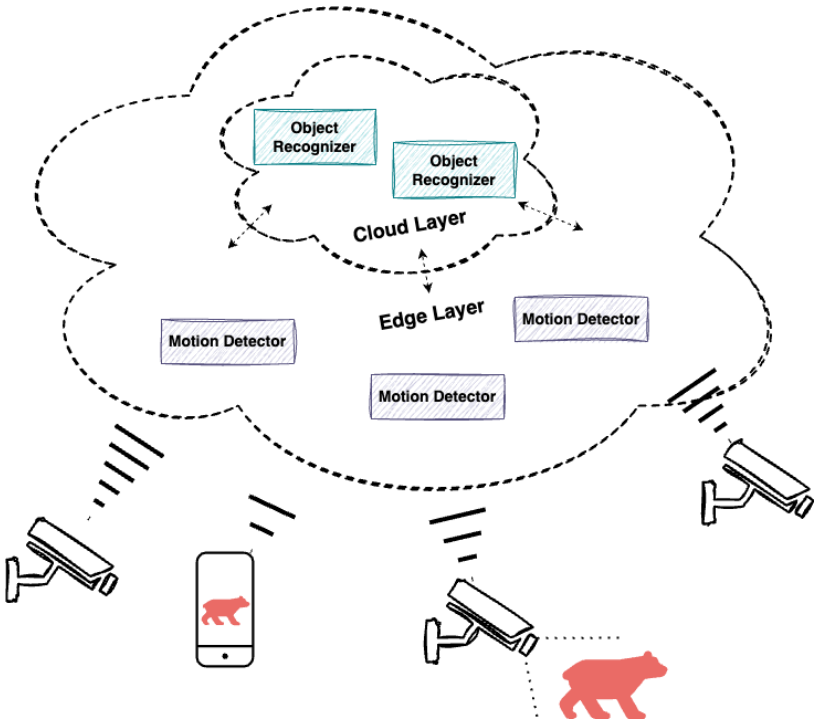

Fig. 8: Edge-to-cloud video processing application

[4] https://discovery.gitlabpages.inria.fr/enoslib/jupyter/
[5] https://gitlab.inria.fr/STACK-RESEARCH-GROUP/software/edge-to-cloud-video-processing

This use-case is made of two main software components:

1. **Motion Detector**: Receives a video feed and continuously tries to detect motion in the images. Whenever a motion event is detected, the corresponding video frames are sent to the Object Recognizer for further analysis. This service typically runs on many small devices that are close to each video source to minimize the amount of data transfers.
2. **Object Recognizer**: Receives video frames with motion events and tries to detect which object or animal is visible in the picture. This service typically has a single instance running in a cloud infrastructure.

The use-case includes a benchmark that consists of injecting a pre-recorded video feed with known parameters (configurable amount of motion over time, with a choice of several animals). It also includes advanced monitoring metrics from both the system and from the application itself (response time, frames per second...)

To deploy this use-case, we will use a simple topology with two Motion Detectors running on small servers such as Nvidia Jetson devices, and one Object Recognizer running on a regular server. This abstract deployment topology is show in Figure 9. We will instantiate different variants of this deployment model during the tutorial.

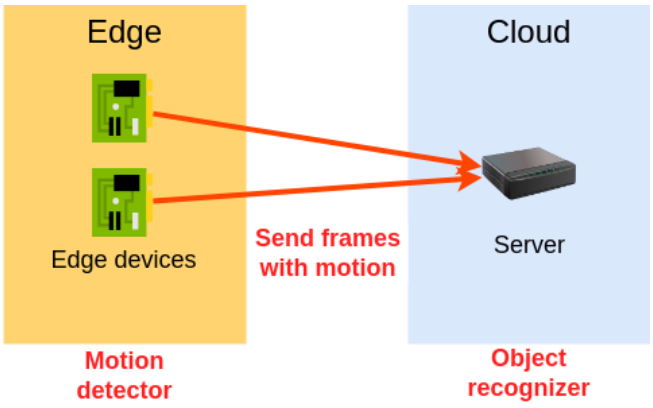

Fig. 9: The abstract deployment model we will use during the tutorial.

4 Methodology: from network emulation to multi-provider experiments

EnOSlib is well suited for experimenting with distributed system software: distributed databases, P2P systems, edge clusters... For this kind of experiments,

we usually want to run the actual software (which rules out simulation) and we want to run it on real or virtualized hardware to obtain realistic system performance. However, we also want a high degree of control on the network, to be able to answer research questions such as: *What is the performance of my system under high network latency?*, or: *How does my distributed system behave when nodes get disconnected?*

In this situation, **network emulation** is a helpful tool: it allows to artificially introduce network issues, such as additional delay or packet loss, while running the real target software on a real testbed.

4.1 Step-by-step workflow

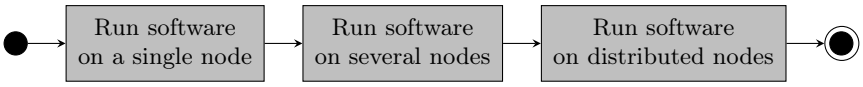

Fig. 10: Workflow for designing a complex distributed system experiment

When designing a complex distributed system experiment, a common workflow is the following:

1. run the target software on a single node to make sure that the deployment process works well (e.g. a virtual machine on the user laptop)
2. run the target software on several nodes connected by a local network, and use network emulation to study the target software under controlled conditions (e.g. run experiments with increasing latency and measure the resulting performance)
3. run the target software on distributed nodes, possibly on simultaneously on multiple platforms (e.g. Grid'5000, Chameleon Cloud, local infrastructure...), to study the target software under real network conditions

For each step of the workflow, the difficulty, cost and complexity of the deployment is increasing. To make sure that later steps work properly, the experimenter can rely on work done in previous steps to make the deployment more robust, building the experimental code in an iterative way.

4.2 Iterating experiments

In experimental research, particularly when assessing system behavior under varying conditions, it is essential to ensure that each test iteration starts from a clean and consistent state. To ease systematic experimentation, it is beneficial to encapsulate the experimental workflow as illustrated in Figure 10 – including resource reservation, setup, benchmark, resource teardown – within an atomic process (e.g. within a dedicated function).

```
import enoslib as en

LATENCIES = ["10ms", "20ms", "40ms", "100ms"]

def reserve_resources():
    my_conf = en.MyProviderConf() \
      .add_machine(roles=["server"], nodes=3) \
      .add_machine(roles=["client"], nodes=1)
    provider = en.MyProvider(my_conf)
    roles, networks = provider.init()
    return provider, roles, networks

def setup_phase(roles):
    # Install all necessary software on all nodes

def benchmark_phase(roles):
    # Run benchmark against the system and collect results
    en.run_command("./benchmark.sh", roles=roles["client"])

def run_experiment_with_emulation(roles, latency):
    # Install and setup all software
    setup_phase(roles)

    # Apply network emulation: all nodes will have the
    # same latency for outgoing packets.
    netem = en.Netem()
    netem.add_constraints(f"delay {latency}", roles["server"],
      symmetric=False)
    netem.deploy()
    netem.validate()

    # Run experiment
    results = benchmark_phase(roles)

    # Save results
    with open(f"results/{latency}/output.csv", "w") as f:
        ...

    # Deconfigure resources before next iteration
    netem.destroy()


# Main program
for latency in LATENCIES:
    # We reserve and release resources for each iteration.
    # This is costly but ensures no side-effect.
    provider, roles, networks = reserve_resources()
    run_experiment_with_emulation(roles, latency)
    provider.destroy()
```

Fig. 11: Typical pattern to iterate experiments with network emulation

This design ensures that each experiment is executed independently and that previous runs do not influence subsequent ones.

Figure 11 illustrates how to iterate over different network latencies using EnOSlib. For each latency value, the experiment reserves resources, sets up the environment (`setup_phase`), applies the specified network latency using `Netem` [6], runs the benchmark (`benchmark_phase`), and finally cleans up the resources using `destroy` functions.

5 Multi-site experiment on Grid'5000

We deploy the use-case from Section 3 on multiple Grid'5000 sites. This is conceptually similar to a multi-provider experiment, but it is a bit simpler on two key aspects: bidirectional network connectivity between the sites is ensured, and we can use the same authentication method to access resources on both sites. Section 6 will extend the experiment to an actual multi-provider context.

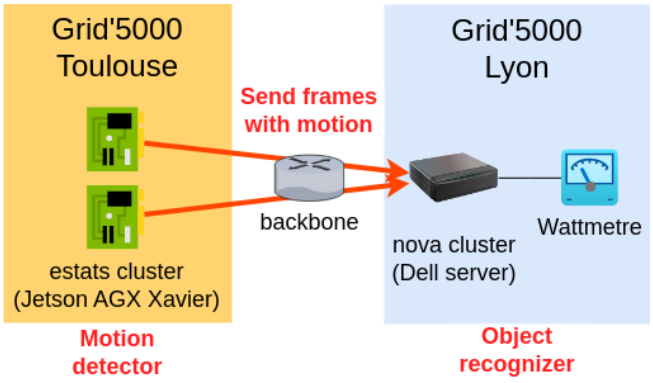

Fig. 12: Multi-site deployment of the use-case on Grid'5000.

We will use two Grid'5000 clusters:

- `estats`[7] in Toulouse, a cluster with 12 Nvidia AGX Xavier devices. These are small but very capable ARM64 devices, similar to a higher-end Raspberry Pi with an integrated GPU.
- `nova`[8] in Lyon, a cluster with very standard `x86_64` Dell servers. Any other cluster would also work, but this one has the advantage of being energy-monitored with physical wattmetres.

[6] https://discovery.gitlabpages.inria.fr/enoslib/tutorials/network_emulation.html
[7] https://www.grid5000.fr/w/Toulouse:Hardware#estats
[8] https://www.grid5000.fr/w/Lyon:Hardware#nova

To deploy the application, we rely on Kubernetes, and specifically the `K3s` Service provided by EnOSlib. It allows to easily deploy K3S clusters on the nodes of the experiment. Anticipating the multi-provider experiment, we deploy two different Kubernetes clusters: first for the edge location, which will run the Motion Detector Application, and then for the cloud location, which will run the Object Recognizer as well as system monitoring services. Figure 13 shows how this is done with EnOSlib.

```
import enoslib as en

def videoprocessing_setup(roles):
    # Deploy two different Kubernetes cluster
    k3s_edge = en.K3s(master=roles["edge"][0], agent=roles["edge"])
    k3s_edge.deploy()
    k3s_cloud = en.K3s(master=roles["cloud"][0], agent=roles["cloud"])
    k3s_cloud.deploy()
    # Copy Kubernetes deployment files to remote nodes.
    # Adapt deployment files to point to service addresses.
    # Run "kubectl apply" commands on master nodes.
```

Fig. 13: Simplified code for deploying two K3S clusters (edge and cloud)

6 Multi-provider experiment on Chameleon Edge and Grid'5000

We extend the experiment: instead of running on multiple sites from the same testbed, we now run an experiment spanning multiple testbeds. We deploy the Motion Detector on Jetson Nano devices from CHI@Edge [14] (CHameleon Infrastructure at Edge), while still deploying the main server in Grid'5000. This deployment is illustrated in Figure 14 while a simplified code for the deployment is show in Figure 15.

One of the challenges in multi-testbed experiments is network interconnection. In our case, the Motion Detector in CHI@Edge needs to be able to send data to the Object Recognizer in Grid'5000. There are several possible ways to handle this challenge:

Native IP connectivity If all testbeds have public IP addresses (e.g. using IPv6), nodes can communicate directly with IP over the public Internet. On Grid'5000, IPv6 access is behind a firewall by default, but it can be opened on-demand

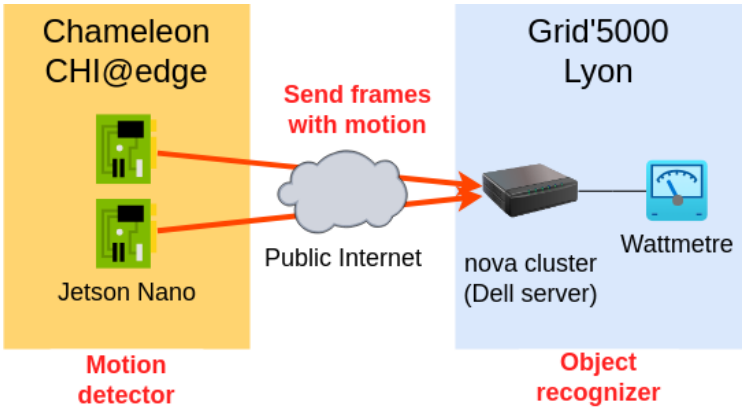

Fig. 14: Multi-provider deployment of the use-case on Chameleon Edge (CHI@EDGE) and Grid'5000.

using the Reconfigurable Firewall service[9]. This is show-cased in another tutorial involving Grid'5000 and FIT IoT-LAB[10].

Native L2 connectivity Some network-oriented testbeds such as FABRIC [5] offer advanced network services, including layer-2 connectivity between testbeds. Since this is somewhat complex to setup and only works if testbeds are already interconnected, this is mostly designed for network-focused experiments that really require layer-2 connectivity.

Tunneling The last resort is often to use tunnels, such as a Wireguard VPN or SSH-based tunneling. This does not give any guarantees on performance, but can provide basic connectivity.

Chameleon Cloud does not support IPv6, while Grid'5000 provides no public IPv4 addresses on nodes. As a result, the experiment relies on tunneling as a last resort.

[9] https://www.grid5000.fr/w/Reconfigurable_Firewall
[10] https://discovery.gitlabpages.inria.fr/enoslib/jupyter/fit_and_g5k/01_networking.html

```
import enoslib as en

chiedge_conf = {
    "walltime": "2:00:00",
    "lease_name": "enoslib-chiedge-lease",
    "resources": {
        "machines": [{
            "roles": ["edge"],
            "device_name": "iot-jetson09",
            "container": {
                "name": "cli-container",
                # This needs to be a ARM64 image
                "image": "debian:12",
            },
        }]
    }
}
g5k_conf = {
    "walltime": "2:00:00",
    "job_name": "enoslib-g5k-job",
    "resources": {
        "machines": [{
            "roles": ["cloud"],
            "cluster": "nova",
            "nodes": 1,
        }]
    }
}
chiedge = en.ChameleonEdge(en.ChameleonEdgeConf.from_dictionary(chiedge_conf))
g5k = en.G5k(en.G5kConf.from_dictionary(g5k_conf))
roles, networks = en.Providers([g5k, chiedge])
```

Fig. 15: Simplified code for multi-provider deployment of the use-case)

7 Conclusion

Since its initial development in 2016, EnOSlib has reached a level of maturity and has been used in many different experiments and projects. At the same time, new features are added regularly to make it easier to perform new kind of distributed experiments.

Among the new features, native multi-provider support is a key enabler for working on the edge-to-cloud continuum. When performing experiments, resources are heterogeneous, are located very far apart, and are often using different infrastructure management software: this makes it difficult to simply obtain simultaneous access on all resources, let alone execute code on all these resources to orchestrate an experiment. The tutorial illustrates how to orchestrate a multi-testbed experiment through a use-case involving communication between edge nodes and cloud nodes. The experiment is deployed on both Grid'5000 in France and Chameleon CHI@Edge in the US.

Acknowledgement

Development of EnOSlib was initially supported by Inria and Orange Labs in the context of the Discovery Open Science initiative. The software is maintained by a core team from Inria on Gitlab[11] and receives contributions from many external contributors: over the years, 34 individual contributors have committed code in the git repository. The authors would like to thank all EnOSlib contributors for their work, as well as all users for their precious feedback to help improve the software.

Some experiments were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

Some experiments were obtained using the Chameleon testbed supported by the National Science Foundation. We thank the Chameleon team for allowing access to the authors to help develop and test EnOSlib on their infrastructure.

References

1. hubblo-org/scaphandre (Apr 2025), https://github.com/hubblo-org/scaphandre, original-date: 2020-10-16T14:10:05Z
2. SLICES (scientific large scale infrastructure for computing/communication experimental studies). https://slices-ri.eu (2025), accessed April 24, 2025
3. Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., Watteyne, T.: FIT IoT-LAB: A large scale open experimental IoT testbed. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT). pp. 459–464 (Dec 2015). https://doi.org/10.1109/WF-IoT.2015.7389098, https://ieeexplore.ieee.org/abstract/document/7389098

[11] https://gitlab.inria.fr/discovery/enoslib

4. Amaral, M., Chen, H., Chiba, T., Nakazawa, R., Choochotkaew, S., Lee, E.K., Eilam, T.: Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In: 2023 IEEE 16th International Conference on Cloud Computing (CLOUD). pp. 69–71 (Jul 2023). https://doi.org/10.1109/CLOUD60044.2023.00017, https://ieeexplore.ieee.org/abstract/document/10254956, iSSN: 2159-6190
5. Baldin, I., Nikolich, A., Griffioen, J., Monga, I.I.S., Wang, K.C., Lehman, T., Ruth, P.: FABRIC: A National-Scale Programmable Experimental Network Infrastructure. IEEE Internet Computing **23**(6), 38–47 (Nov 2019). https://doi.org/10.1109/MIC.2019.2958545, https://ieeexplore.ieee.org/abstract/document/8972790
6. Balouek, D., Amarie, A.C., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Perez, C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding Virtualization Capabilities to the Grid'5000 Testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) Cloud Computing and Services Science. pp. 3–20. Springer International Publishing, Cham (2013). https://doi.org/10.1007/978-3-319-04519-1_1
7. Balouek-Thomert, D., Rodero, I., Parashar, M.: Evaluating policy-driven adaptation on the edge-to-cloud continuum. In: 2021 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC). pp. 11–20 (2021). https://doi.org/10.1109/UrgentHPC54802.2021.00007
8. Cherrueau, R.A., Delavergne, M., Van Kempen, A., Lebre, A., Pertin, D., Balderrama, J.R., Simonet, A., Simonin, M.: Enoslib: A library for experiment-driven research in distributed computing. IEEE Transactions on Parallel and Distributed Systems **33**(6), 1464–1477 (2021). https://doi.org/10.1109/TPDS.2021.3111159
9. Courageux-Sudan, C., Orgerie, A.C., Quinson, M.: Studying the end-to-end performance, energy consumption and carbon footprint of fog applications. In: 2024 IEEE Symposium on Computers and Communications (ISCC). pp. 1–7 (2024). https://doi.org/10.1109/ISCC61673.2024.10733735
10. Delamare, S., Nussbaum, L.: Kwollect: Metrics Collection for Experiments at Scale. In: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). pp. 1–6 (May 2021). https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484540, https://ieeexplore.ieee.org/abstract/document/9484540
11. Fieni, G., Acero, D.R., Rust, P., Rouvoy, R.: PowerAPI: A Python framework for building software-defined power meters. Journal of Open Source Software **9**(98), 6670 (Jun 2024). https://doi.org/10.21105/joss.06670, https://hal.science/hal-04601379, publisher: Open Journals
12. Guilloteau, Q., Bleuzen, J., Poquet, M., Richard, O.: Painless transposition of reproducible distributed environments with nixos compose. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–12 (2022). https://doi.org/10.1109/CLUSTER51413.2022.00051
13. Hasenburg, J., Grambow, M., Bermbach, D.: MockFog 2.0: Automated execution of fog application experiments in the cloud. IEEE Transactions on Cloud Computing **Early Access** (2021)
14. Keahey, K., Anderson, J., Sherman, M., Zhen, Z., Powers, M., Brunkan, I., Cooper, A.: Chameleon@ Edge Community Workshop Report (2021)
15. Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H.S., Hammock, C., Mambretti, J., Barnes, A., Halbah, F., Rocha, A., Stubbs, J.: Lessons Learned from the Chameleon Testbed. pp. 219–233 (2020), https://www.usenix.org/conference/atc20/presentation/keahey

16. Kp, G., Pierre, G., Rouvoy, R.: Studying the energy consumption of stream processing engines in the cloud. In: 2023 IEEE International Conference on Cloud Engineering (IC2E). pp. 99–106 (2023). https://doi.org/10.1109/IC2E59103.2023.00019
17. Lambert, T., Ibrahim, S., Jain, T., Guyon, D.: Stragglers' detection in big data analytic systems: The impact of heartbeat arrival. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 747–751 (2022). https://doi.org/10.1109/CCGrid54584.2022.00084
18. Mokhtari, A., Jonglez, B., Ledoux, T.: Towards digital sustainability: Involving cloud users as key players. In: 2024 IEEE International Conference on Cloud Engineering (IC2E). pp. 126–132 (2024). https://doi.org/10.1109/IC2E61754.2024.00021
19. Noureddine, A.: PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In: 18th International Conference on Intelligent Environments. Biarritz, France (Jun 2022). https://doi.org/10.1109/IE54923.2022.9826760, https://hal.science/hal-03608223
20. Philippe, J., Omond, A., Coullon, H., Prud'Homme, C., Raïs, I.: Fast choreography of cross-devops reconfiguration with ballet: A multi-site openstack case study. In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE Computer Society, Rovaniemi, Finland (mar 2024). https://doi.org/10.1109/SANER60148.2024.00007
21. Rac, S., Sanyal, R., Brorsson, M.: A cloud-edge continuum experimental methodology applied to a 5g core study (2023), https://arxiv.org/abs/2301.11128
22. Rosendo, D., Mattoso, M., Costan, A., Souza, R., Pina, D., Valduriez, P., Antoniu, G.: Provlight: Efficient workflow provenance capture on the edge-to-cloud continuum. In: 2023 IEEE International Conference on Cluster Computing (CLUSTER). pp. 221–233 (2023). https://doi.org/10.1109/CLUSTER52292.2023.00026
23. Rosendo, D., Silva, P., Simonin, M., Costan, A., Antoniu, G.: E2clab: Exploring the computing continuum through repeatable, replicable and reproducible edge-to-cloud experiments. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER). pp. 176–186 (2020). https://doi.org/10.1109/CLUSTER49012.2020.00028