## **TODO:**

* Power monitor documentation (website), libraries to use it programmatically
* Check bachelor thesis (Di Giampaolo, Pinna)
* Check code automation repository, learn how the code works

# Serverless edge side platforms solutions:

Platforms which enable the execution of functions on a server, close to the workload generator. Automatically (usually) management of aspects like autoscaling, load balancing. We can say these platforms implement something like Kubernetes does.

Proposed solutions:

* **Knative** (<https://knative.dev/>): open-source Kubernetes-based platform designed to simplify the deployment and management of serverless apps. Good choice if a Kubernetes infrastructure already exists.
  + Advantages:
    - Kubernetes-based, very robust;
    - Autoscaling capabilities
    - Portable across cloud providers
  + Disadvantages:
    - Complexity of Kubernetes management
    - Running it adds overhead to the Kubernetes cluster, which can impact performance
* **WasmCloud** (<https://wasmcloud.com/>): WebAssembly platform for building distributed serverless apps that are portable and scalable. **Needs python to be compiled to WebAssembly.**
  + Advantages:
    - Portability, good fit for apps that need to be run on both edge and cloud
    - Security, sandboxed to reduce risks associated to untrusted code
    - Simplified development by abstracting away the infrastructure layer, no need to worry about infrastructure, load balancing, networking
    - Polyglot support
  + Disadvantages:
    - Relatively new
    - Performance limitations due to its design for lightweight serverless tasks
    - Still requires an infrastructure to run, can add some overhead compared to fully managed serverless solutions
* **Faasd** (<https://github.com/openfaas/faasd>): lightweight, single-node distribution of OpenFaas, designed to deploy serverless functions without using Kubernetes. Ideal for environments where Kubernetes would be overkill. It has native support for Python functions offloading, this is very good, since it won’t be mandatory to compile the functions to webassembly, introducing potentially more latency. **Too complex to setup, documentation is ridiculous.**
  + Advantages:
    - Simpler setup and management, no Kubernetes overhead
    - Consumes fewer resources and it’s well-suited for low power environments like edge devices or single-board computers, like RPIs.
    - Open-source, self-hosted, free
    - API-support for REST
  + Disadvantages:
    - Complex event-driven apps will need a more powerful alternative
    - Not suitable for complex production workloads
    - Manual scaling
  + RESOURCES:
    - Faasd on RPI:<https://blog.alexellis.io/faasd-for-lightweight-serverless/>
* **Sledge serverless framework** (<https://github.com/gwsystems/sledge-serverless-framework>): experimental lightweight serverless solution suitable for edge computing. Managed to run an example on Debian. Problem is, when the service is running in the background, the system is very slow. **Only able to offload C/C++ functions (WebAssembly).**
  + Advantages:
    - Specifically designed for edge environments, offering low-latency processing by running functions closer to the user
    - Lightweight deployment, ideal for deployment on edge devices
    - Open-source
  + Disadvantages:
    - Experimental and early-stage
    - Lack of advanced orchestration
* **Faasm** (<https://github.com/faasm/faasm>): high-performance open-source serverless framework that uses Wasm to provide isolated, efficient execution environments for microservices and serverless functions. **Requires too much disk space for its dependencies to be ran as a server on Ubuntu with 25GB disk setup. Moreover, WebAssembly.**
* **TinyFaas** (<https://github.com/OpenFogStack/tinyFaaS>): extremely lightweight FaaS platform designed to minimize overhead and complexity. Unlike many serverless platforms that depend on Kubernetes or other orchestration systems, TinyFaaS is built to run on a single node, making it particularly suitable for low-resource environments like edge devices or lightweight development environments. TinyFaaS is intended for Linux hosts!  
  Important note:the setup will use the computer’s Docker instance to manage containers and will allow anyone in the same network to start Docker containers with arbitrary code.
  + Advantages:
    - Low overhead
    - Ideal for edge and IoT devices
    - Quick setup and deployment
    - Event-driven capabilities
  + Disadvantages:
    - Single node limitation
    - Limited ecosystem
    - Limited event sources and triggers
    - Limited security (see above)
* **Edgeless** (<https://github.com/edgeless-project/edgeless/>): serverless computing platform designed specifically for edge environments. It focuses on reducing latency by processing requests closer to the end-user and minimizing resource consumption.
  + Advantages:
    - Optimized for edge computing
    - Low latency processing
    - Open-source and customizable
  + Disadvantages:
    - Basic orchestration features
    - Limited ecosystem
    - Limited event and language support

## **Local offloading experiment (no RPI involved):**

Trying to offload a simple Python function, all locally, no RPI involved.   
Figured it’s better to test everything on a **VM**, since everything works on Linux systems. Virtualization overhead should not be a problem in the real setup since the RPI is already a Linux machine.

The setup will then be the following:

* **RPI:** sends offloading requests to the offloading framework instance running on the VM, that acts as a server.
* **VM:** on a PC; the VM runs the offloading framework and listens for upcoming requests for offloading. Computes the results and delivers back to the requester.

# Trying out Faasd:

* **Required dependencies:** docker and containerd running.   
  Download them, make them start. Better to setup the system so that they start on startup (systemctl enable docker, systemctl enable containerd). Check status of these services with systemctl status <service-name>. To use Docker without root privileges (needed to pull images): sudo usermod -aG docker $USER. To apply changes right away: newgrp docker. After this, docker commands should be executable without root privileges. To test Docker: docker run hello-world. To test containerd: sudo ctr version.  
  git, curl, bridge-utils are also needed (**sudo apt install -y git curl bridge-utils**)
* **Installation script:** after cloning faasd repo, run sudo **./hack/install.sh**
* **Faasd services:** these are two services: **faasd** and **faasd-provider**. Both need to be running in order to make faasd work (better to do **systemctl enable faasd** and **systemctl enable faasd-container**, so they too start at system start).  
  Logs can be checked for errors: **sudo journalctl -u faasd**.
* **Faasd start problem:** can happen if faasd is unable to pull the necessary Docker image for the OpenFaaS gateway from GitHub Container Registry (ghcr.io). This is because the installation script is old: need to update ‘**/var/lib/faasd/docker-compose.yaml**’ and under ‘**gateway**’ set ‘**image**’ to ‘**ghcr.io/openfaas/gateway:latest**’. Then restart faasd with **sudo systemctl restart faasd**, and now it should run with no problems. Once you do this the first time, the it always starts with no problems.
* **Faasd usage:** Faasd server runs on localhost at port 8080 ([**http://127.0.0.1:8080**](http://127.0.0.1:8080)). We can login using username ‘**admin**’ and the password stored in **/var/lib/faasd/secrets/basic-auth-password**. By logging in, we are redirected to the Faasd dashboard. We can deploy functions from here or by using the CLI. The CLI can be installed via **curl -sSL https://cli.openfaas.com | sudo sh.** To login to the CLI, we need to do ‘**echo "<password-in-basic-auth-password>" | faas-cli login --username admin --password-stdin’.** 
  + **Create a new python function:** ‘faas-cli new –lang python3 hello-python’. This creates a folder named hello-python/ and two more files: hello-python.yml (deployment config) and handler.py (the function logic).
  + **Edit the function logic**
  + **Build the function:** using faas-cli to build the Docker image of the function: **faas-cli build -f hello-python.yml** (of course, the docker service must be running). First, we login to GHCR via Docker with ‘echo "**<ACCESS-TOKEN>" | docker login ghcr.io -u YOUR\_GITHUB\_USERNAME --password-stdin**’ (token = **ghp\_zhT8u8RRQsD7dtKqiJ8m1qjmSUCjG03lCLqH**, expires on 26th of February 2025). After this the image must be tagged with the correct format: **docker tag hello-python:latest ghcr.io/<github-username> /hello-python:latest**. Then it has to be pushed: **docker push ghcr.io/<github-username>/hello-python:latest.**The yml file must be modified: image: ‘**image: ghcr.io/<github-username>/hello-python:latest’.**
  + **Push the Docker image:** Faasd uses a container registry (ghcr, GitHub Container Registry), so the image must be pushed there.
  + **Deploy the function:** the Gateway URL must be set in hello-python.yml (to 127.0.0.1:8080 for local use).  
    To deploy: faas-cli deploy -f hello-python **PROBLEMS HERE, STOPPED HERE.  
    for reference:** [**https://chatgpt.com/c/674833c9-e2cc-800c-a725-b5e011a17a6d**](https://chatgpt.com/c/674833c9-e2cc-800c-a725-b5e011a17a6d)**,** [**https://chatgpt.com/c/674997fb-dfe4-800c-9c0b-31828031638e**](https://chatgpt.com/c/674997fb-dfe4-800c-9c0b-31828031638e)

# Trying out Edgeless:

EDGELESS is a framework that enables serverless edge computing, which is intended especially for nodes with limited capabilities. An EDGELESS cluster consists of one or more *orchestration domains* which are managed by components called *ε-CON* (controllers).  
An orchestration domain is composed of:

* **Orchestrator (ε-ORC):** manages the scaling of function instances and resources within its domain
* **Balancer (ε-BAL):** plays two main roles
  + Realizes an *inter-domain data plane*, to allow events generated by a function instance in an orchestration domain to be consumed by a function instance in another orchestration domain
  + Configures and manages the resources
  + **NOTE (important):** up to now, the balancer is just a skeleton, no operations are performed by this component
* **Node:** one or more, which can host the execution of function instances of a given workflow in a WebAssembly runtime environment or as a Docker container.

Users can interact with the *ε-CON* to request the creation of *workflows.***A workflow simply specifies how a number of functions and resources should interact with one another to compose the service requested by the user by sending to one another asynchronous events that are akin to function invocations.***Functions* live entirely in the realm of the EDGELESS runtime, while *resources* may interact with the external environment (e.g. handling events in a resource may have side effects, like the update of an entry in an external DB).

Functions are *stateful*: a given function instance is assigned to exactly one workflow, thus the function developer may assume that data will generally remain available across multiple invocations on the same function instance.   
Such state is, however:

* Tied to the specific instance: if multiple instances of the same function exist, there is no consistency guarantee across the multiple states.
* Ephemeral: if a function instance is terminated, there’s no way to save/persist the state.

In EDGELESS, workflow may consist of different ways of ***function compositions***:

* Single function
* Function chain
* Directed acyclic graph of functions
* Arbitrary graph of functions

The bytecode of the WASM function instance (or the name of the Docker container) to be started is provided by the user when requesting the creation of a workflow, which also includes *annotations* to specify the QoS requirements (e.g. the maximum completion time) and workload characteristics (e.g. average invocation rate), as well as other parameters like required HW properties.   
Depending on the combination of such parameters, the *ε-CON* might reject the creation of such workflow, if meeting the requirements is impossible.

EDGELESS has a three-tier hierarchical architecture. A minimal EDGELESS system is composed of a single cluster with one orchestration domain hosting just one node.   
A user can leverage the **CLI** to locally build the WebAssembly bytecode of a function and interact with the *ε-CON* to create workflows. A workflow can be seen as a directed graph of abstractions that performs actions by reacting to events, along with its annotations that define the characteristics and requirements of the application/service to which the workflow is associated. A workflow is identified with an UUID.

**Building EDGELESS:** <https://github.com/edgeless-project/edgeless/blob/main/BUILDING.md>

**Deploying EDGELESS (reference: https://github.com/edgeless-project/edgeless/blob/main/documentation/deploy\_step\_by\_step.md):**in one shell, we first create the default configuration of the **ε-CON**. We provide a configuration file (controller.toml) which specifies the URL exposed by the ε-CON towards the clint and the map of its orchestration domains, identified by their name and URL of the **ε-ORC** (which at this point is not yet deployed). Then, we can deploy the ε-CON. The process will then remain waiting for more action.  
In another shell we can deploy the ε-ORC. Its configuration file contains the domain name and URL exposed by the orchestrator (controller.toml and orchetstrator.toml both have this parameter, this parameter must be the same), the orchestrator baseline configuration, with the strategy that will be used (Random or RoundRobin), and the keep-alive interval before a node is automatically de-registered. When deploying the ε-ORC, the shell outputs all this information.  
In yet another shell, we can now start deploying a **node**. The node configuration file presents some sections:

* [general] section:
  + UUID of the node: must be unique within the orchestration domain.
  + URLs exposed by the node for different purposes:
    - Agent URL: used by the ε-ORC to manage the lifecycle of the functions/resources the node hosts
    - Invocation URL: used by the EDGELESS data plane to consume events addressed to function instances and resources of the node
    - Metrics URL: telemetry data about the node’s runtime.
  + URL of the ε-ORC, to which the node connects
* [wasm\_runtime] section: whether the node accepts WebAssembly function instances
* [container\_runtime] section: whether the node accepts Docker function instances
* [resources] section: name (and if needed, configuration) of the resource providers offrered by the node (if the name is left empty, the resource is not created)
* [user\_node\_capabilities] section:
  + Values of the node capabilities exposed to the ε-ORC
  + Labels assigned to the node, which can be used to force some nodes to be selected as candidates by the ε-ORC

When deploying the node, some of this information is shown in the console. Moreover, the output of the ε-ORC updates, showing that the resource providers have been added and associated with the newly created node, which also offers function execution with given capabilities.

**Building and executing a simple function (provided in the examples):** in a fourth shell (one for the ε-CON, one for the ε-ORC, one for the node), we can start a workflow through the CLI. **A workflow is specified by a JSON file which lists the functions and resources used, and how they interact with each other by creating events on output channels.** The simplest workflow consists of a single function, which doesn’t interact with any other abstraction:  
{

"functions": [

{

"name": "noop",

"class\_specification": {

"id": "noop",

"function\_type": "RUST\_WASM",

"version": "0.1",

"code": "../../functions/noop/noop.wasm",

"outputs": []

},

"output\_mapping": {},

"annotations": {

"init-payload": "nothing interesting"

}

}

],

"resources": [],

"annotations": {}

}

The workflow above creates a function called “noop”, whose bytecode is assumed to be found in the local file “noop.wasm”. To start the workflow, we must first build the noop.wasm function first (https://github.com/edgeless-project/edgeless/blob/main/functions/noop/src/lib.rs).  
**The example function presents the definition of four handlers, which are the basic programming model of EDGELESS functions:**

* **handle\_cast:** called to consume an asynchronousevent intended for this function instance that does not expect a return
* **handle\_call:** similar but the event is synchronous (a return is expected)
* **handle\_init:** called upon function instance creation to initialize its data structures, if any
* **handle\_stop:** called upon function termination to perform clean up procedures, if required

**To build the WebAssembly bytecode:** ./edgeless\_cli function build ../../functions/noop/function.json  
This produces the file “../../functions/noop/noop.wasm” that is needed by the workflow.

The default CLI configuration can be created (with a configuration file that contains only the URL of the ε-CON) and the workflow can be started with UUID=$(./edgeless\_cli workflow start ../../examples/noop/workflow.json)  
This will save into the $UUID environment variable the output of the edgeless\_cli command, which is the UUID of the newly-created workflow.   
The processes in the other terminals will show some action: the ε-CON logs that a new workflow has been started, the ε-ORC logs that a function has spawned of our only node, the node logs that a function instance has been created and initialized. No further action happens because “noop” doesn’t do anything else. The workflow remains active within the ε-CON.

**Listing the active workflows:** ./edgeless\_cli workflow list

**Stopping a workflow:** ./edgeless\_cli workflow stop $UUID

**EDGELESS workflow composition (reference: https://www.youtube.com/watch?v=kc4Ku5p5Nrw):**An EDGELESS workflow is a composition of functions. Let’s assume we have two functions: filter\_in\_range() and moving\_avg(). The first receives as input a number and checks that such input is within a validity range. If this is the case, an event is generated towards the next item, which is the other function, which keeps a window of past values, and creates an output that is the average of the received values, that is then saved to a Redis server.   
If the filter\_in\_range() detects that the value is not ok, it logs the event to an external file.   
We also have a sensor simulator function, needed to test the workflow (this will produce data to be given as input). How can all of this be done in EDGELESS? In the examples we already find the developed functions.

Each function comes with some source code and with a function specification, with an identifier, the function type, version, toml file and the output callbacks (or channels). In this case, the two functions have two channels defined, which are used to call the next function in the chain or log an error. Connecting all the channels is not mandatory.

**The workflow can be composed using a JSON file. The file consists of four sections:**

* **[alias]:** just a name for the workflow
* **[functions]:** list of functions the workflow will be composed of.   
  In each function, we find:
  + An **alias** for the function
  + **Class\_specifications**: copies most of the data from the function specification (identifier, function type, version and callbacks are the same). We also have an additional parameter (*include\_code\_file*) that tells where the bytecode will be found on the local filesystem. This must be available when the workflow is created. In the *outputs* JSON array, we specify the list of output events the function can produce.
  + **Output\_mapping**: this is very important since it’s *where we specify how to combine together functions and resources*. Here we define a mapping of output channels to logical names of functions/resources, local to the workflow.  
    In this example:
    - The sensor simulator should map the output channel to the filter\_in\_range function
    - The filter\_in\_range will map the output to the moving\_avg function and the error to the file log
    - The moving\_avg function will send the output to Redis. The error channel will be left unused.
  + **Annotations:** can be empty or be a JSON object. In the object, we can define the service level objectives and requirements of the function.   
    Notable supported annotations:
    - ***Init\_payload:*** specifies the configuration of the function instances. This data will be used during the call to *handle\_init* to initialize eventual data structures (can be seen as a constructor).
    - ***Resource\_match\_all:*** the function mustbe spawned on a node that hosts all the specified resources, if any.

In this example, we use *init\_payload*:

* + - for the filter\_in\_range we can specify here the min and max values (the range) for the validity interval
    - for the moving\_avg we can specify the number of values that are used to compute the average.
* **[resources]**: we add resources here, for which we do not have to specify the bytecode location, but only the type of resource we want to use.   
  Notable resources:
  + *File-log:* saves log lines to a file
  + *http-egress:* executes HTTP commands on external web servers
  + *http-ingress*: ingests HTTP commands from external clients
  + *redis*: updates a value on an external Redis server

Elements:

* + **Name:** just an alias for the resource.
  + **class\_type:**the type of resource that must be instantiated.
  + **Output\_mapping:** how to map each output channel to the function or resource that will handle it (analogous to the same field in functions).
  + **Configurations:** each resource has some resource-specific configuration parameters.  
    In this example:
    - File-log case: we specify the name of the file where to save the logs
    - Redis case: we specify the URL of the Redis server and the key where we want to save the output

**To deploy the workflow, we need a few things:**

* The **compiled bytecode for each function**. Can be obtained by using the command line utility of EDGELESS: target/debug/edgeless\_cli function build examples/tutorial-01/filter\_in\_range\_function/function.json. This will create the bytecode in Rust wasm for the filter\_in\_range function. The bytecode can then be found in the function’s directory. We need to do the same for the other functions as well.
* **An EDGELESS cluster**: can be created by running edgeless\_inabox, which is a minimal EDGELESS implementation of the core components all together in a single binary (RUST\_LOG=info target/debug/edgeless\_inabox, the needed files can be generated as shown in the repo tutorial)

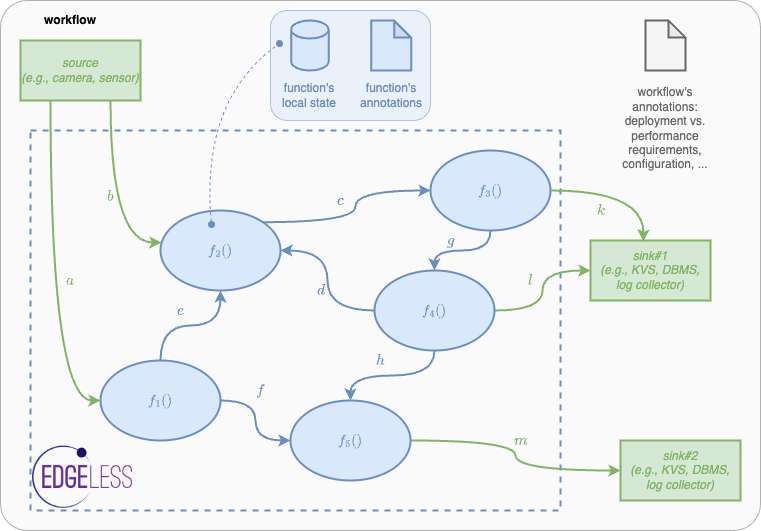
To create the workflow, we can do target/debug/edgeless\_cli workflow start examples/tutorial-01/workflow.json

The workflow gets accepted by the system and the identifier is returned.

We can now see that there is an errors file, that if opened shows the values out of range.  
By accessing the Redis server with the specified key, we’ll see the updating values.

**About setting up and using the workflow in our system:** functions need to be compiled into wasm beforehand and then deployed in a workflow. We will likely need two copies of the functions: the simple functions that can be ran directly on the RPI and the implementation of them that needs to be compliant with EDGELESS.

For the workflow construction point of view, we have two solutions:

* **One Workflow Per Function:** wecreate a workflow for each function we decide it’s offloadable. Then, we dynamically decide based on power consumption measurements on the RPI to offload it or not. In the first case we’ll access the workflow and trigger the function execution, to the retrieve the result. This will result in more calls to the EDGELESS server, of course.
  + Advantages:
    - each function’s offloading is independent, the RPI decides everything
    - simpler setup
  + Disadvantages:
    - Higher overhead (more HTTP calls)
    - Need to manage a potentially large number of workflows (depending on the monitored application)
* **Single workflow with interconnected functions:** we create a single workflow composed of different functions interconnected with each other. We setup for each one of them the possibility to forward the results to the next function in the chain or to send the results back to the RPI, so that execution can be continued locally. We connect through HTTP to trigger the execution of a certain function or group of functions. Each function could receive, apart from its parameters, an additional Boolean parameter which will tell it to forward the results to the next function in the chain or to send them back to the RPI. Another alternative could be passing a vector of names of functions that will be executed on the EDGELESS server: the vector gets passed from function to function and each one of them removes its name from it. The function that receives an empty vector will know that its results will have to be sent back to the RPI.
  + Advantages:
    - Reduced HTTP calls wrt the previous method
    - No need to transfer intermediate results back and forth
  + Disadvantages:
    - Workflow complexity, if there are many interconnected branches
    - Errors in one function might propagate affecting the whole workflow

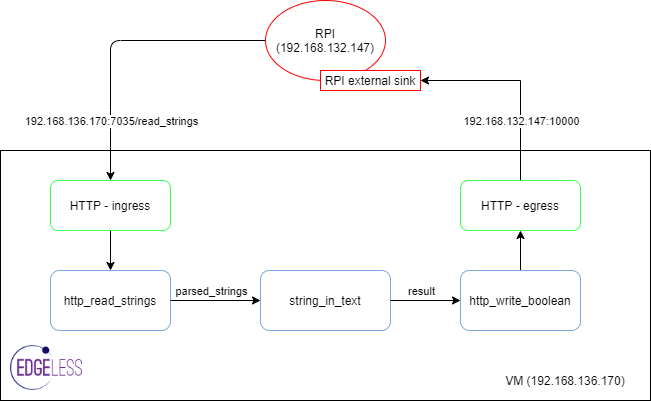
**Workflow with HTTP external source (reference:** <https://github.com/edgeless-project/edgeless/tree/main/examples/simple_workflow_http>**):** the example is composed of a workflow which creates a chain like so:

* HTTP ingress that waits for an external source to POST a message whose body contains an integer
* A function incr that increments by 1 the received number in its payload
* A function double that doubles the received number in the payload
* An HTTP egress that sends the received message to an external sink

We first need to build the functions to get the WASM binaries, and then we can request to the edgeless\_cli the creation of the workflow.

We then need two shells, one to play the role of the external sink, and the other one to play the role of the requester.  
To be able to reach the HTTP-ingress resource from the RPI, we modified *edgeless/node.toml* and its *http-ingress-URL* entry, from http-ingress-URL=”http://127.0.0.1:7035” to http-ingress-URL=”0.0.0.0:7035”, so that requests are accepted from every host. By using the curl command specified in the README file from the RPI, we are now able to reach the server from outside localhost, and the result gets displayed in the sink we opened on the VM.

**Test HTTP workflow:**   
We develop a simple workflow, in which we have 3 components:

* An HTTP ingress resource: receives requests from the RPI, forwards the parsed data to the next function
* String\_in\_text() function: receives JSON data from the HTTP ingress resource, checks if the provided word is contained in the provided text, forwards the Boolean result to the HTTP egress resource
* An HTTP egress resource: receives the result from the function and writes it to an external sink opened on the RPI.

Assuming the RPI has IP address 192.168.132.147, we can first open a sink using Netcat on port 10000 on the RPI, and then make a request towards the EDGELESS cluster running on the VM (which has IP address 192.168.170). The cluster listens for HTTP requests on port 7035. We must specify the name of the host and the parameters: **curl -v -H “Host: RaspberryPI” -H “Content-Type: application/json” http://192.168.136.170:7035/read\_strings -d ‘{“Text”: “This is some sample text”, “Word”: “sample”}’**

The request will be processed by the EDGELESS workflow and a response will be sent on the sink opened on port 10000 of the RPI. The response will contain “true”, as the provided text contains the provided word.  
All the computations have been executed on the EDGELESS cluster.

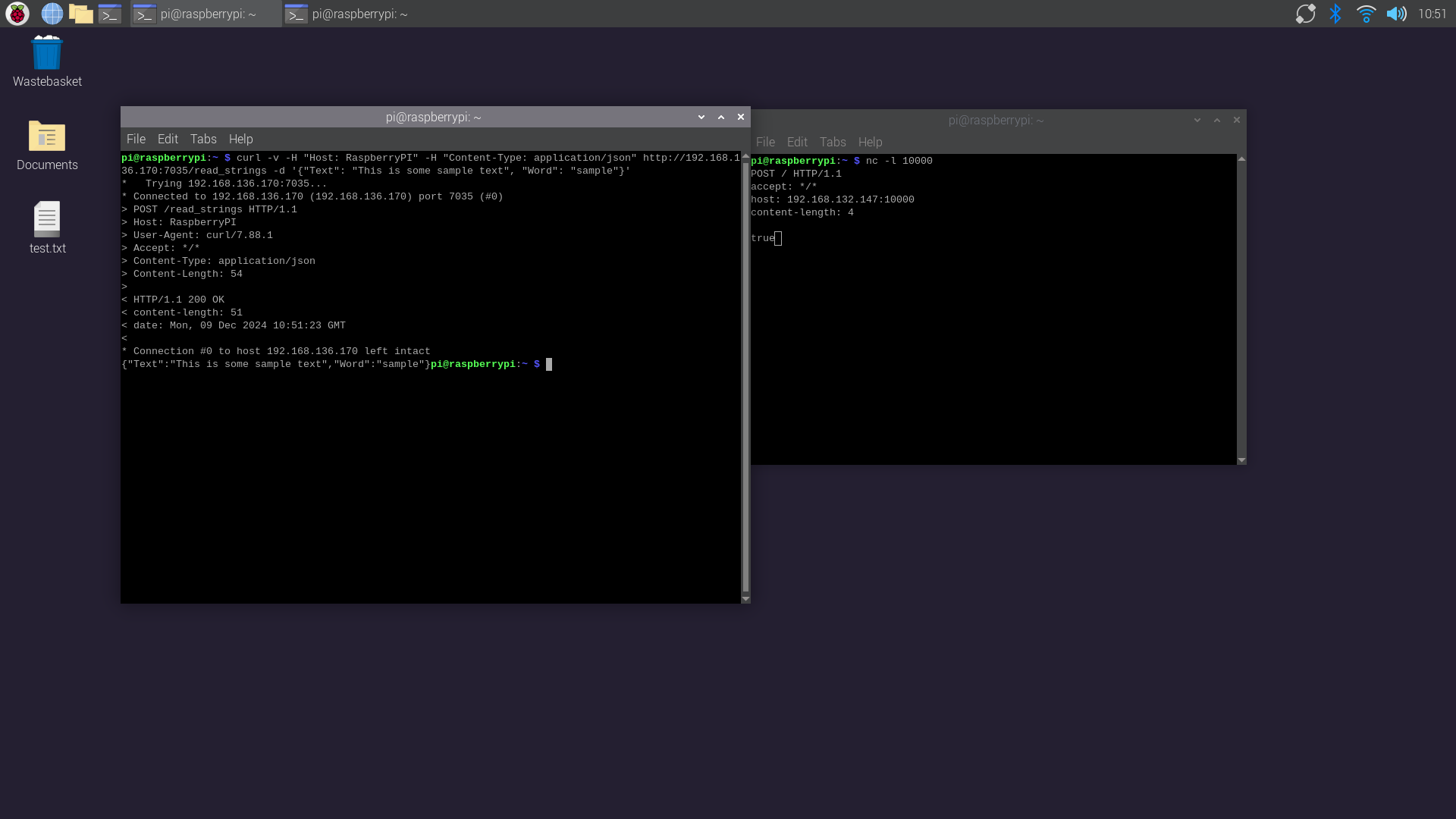
The workflow is composed by means of the *workflow.json* file. The file specifies what the workflow is composed of and the interconnections among the different resources and functions.

Each function is described by its *function.json* file, where the name and outputs can be specified, to setup interconnections.

To run the workflow, some steps need to be taken:

* Start edgeless\_inabox: this will start the EDGELESS cluster on the VM
* Compile the functions in WASM format (can be done using the edgeless\_cli component): **target/debug/edgeless\_cli function build <path\_to\_function\_folder>/function.json**  
  After this, the “version” field in Cargo.lock needs to be updated from 4 to 3.
* Load the workflow into EDGELESS using edgeless\_cli: **target/debug/edgeless\_cli workflow start workflow.json**

We can communicate data to the workflow from the RPI and receive the response back like this:



**Would it be possible to integrate *our* own balancing policy into the Orchestrator in order to take into account power saving objectives of the client?**

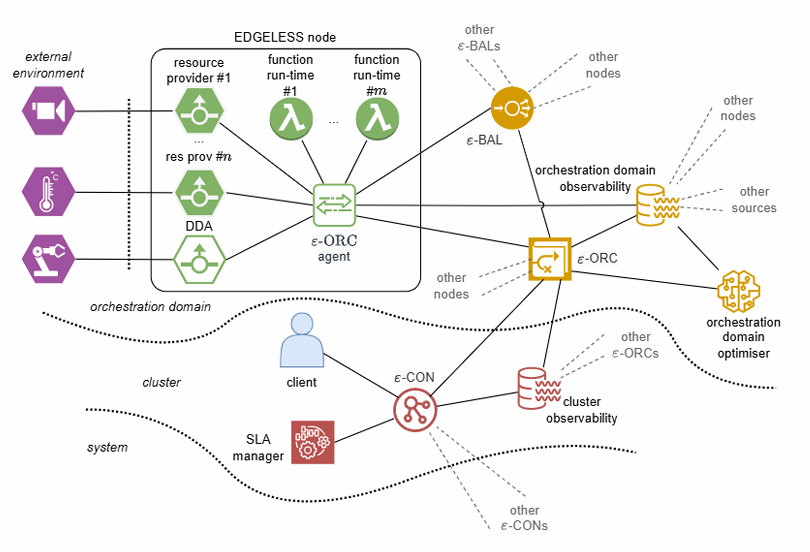
***How can the server implement such a policy? Isn’t the client obliged to send periodic data about the load and power usage? And then how to act? The server just hints the client that it would be better for its power consumption levels to offload to it a certain function?***

Some scenarios in which the server could take the client’s power saving needs into account:

1. **Client power-aware offloading requests:** client sends data about battery level and power along with its offloading requests. The server prioritizes the tasks from clients with low power levels, works well if the server needs to manage multiple clients at once.
2. **Server-based chained offloading**: the client could send an offloading request for a specific function to the server, and include in the request also power data. The server analyzes the request and considers the dependencies, in terms of functions, of the requested function. Moreover, it analyzes the power profile of the client and decides if the request of the client it’s actually ok or if executing more functions than the requested one would be more beneficial to the client than just offloading the single. If this is the case, the server executes the requested function along with the additional ones (of course, the additional functions that can be executed are those that as input will receive data processed by the initially requested function, like in a chain). The server communicates the results and the IDs or names of the functions he autonomously decided to execute, so that the client is aware of which need to be skipped locally. The client can then update its workflow, skipping the already executed functions.   
   To know which function depends on which other function we have the server-side workflow of the program, and at this point needing to connect each function to an HTTP egress resource seems inevitable.  
   This solution would not need the client to send periodic data to the server about its power usage.  
   This setup would require the client’s program to be defined according to this methodology.
   1. Example: the client requests the server to offload func\_A, which produces output data used by func\_B and func\_C. The server checks the power data that was included in the request and determines that offloading all three functions would save power on the client, so it computes all of them. Then the server sends back the results of func\_A, func\_B, func\_C, and a notification too the client that func\_B and func\_C were also executed on the server. The client skips these two functions processing the output directly.

We would like to avoid periodic and predefined communication with the server to send power metrics. Some alternatives to periodic updates:

* Event-driven updates: instead of periodic messaging, the client could send the server its power metrics only when a significant change occurs (e.g. power drops below a certain threshold)
* Power metrics packaged into the offloading request: as described above, the client sends offloading requests to the EDGELESS server and includes some data about its current power status. If requests are frequent enough, the server could be able to reconstruct the power status of the client device.

**New EDGELESS setup to be tested (17/12/2024 meeting, refer to the EDGELESS paper for figures):**we know that an Orchestrator manages a collection of Edge nodes. Let’s assume we are in presence of a single orchestration domain. An EDGELESS node executes an E-ORC agent, that communicates with an E-ORC, orchestrator of a single domain. The E-ORC can, in addition, retrieve metrics from the Orchestration Domain Observability, and some customized orchestration policies can be defined (Orchestration Domain Optimizer), see following picture:   
We could take our RPI and make it operate at the same time as a client and as an EDGELESS node: this is possible, since we just need to spawn a node on it. There will then be one/some other EDGELESS nodes, all referring to the same domain, and so under the control of the same E-ORC.   
So, we have our device (could be a wearable device), represented by the RPI, and it has some needs: it produces data and has to process them, in terms of functions. A workflow will be defined, representing the operations that we want to do on such RPI data. The orchestrator could decide if the single function will be executed on the same node in which the data is produced, so the same RPI, or choose another node, external to the RPI.   
The idea is this: the ORC contains some policies based on which it decides where to execute functions, moreover it has some metrics, coming from the Orchestration Domain Observability, to which (looking at the figure) would participate also the E-ORC agent. The agent could measure what is the load on the device and transfer such data to the observability domain, so that the E-ORC could read such data. If we put the E-ORC agent on the RPI (it is a part of the node, so it should be there), this could measure some metrics like the computational load, and maybe augment its capabilities to provide also the battery level, for example. Other metrics would for example be the latency to reach the other EDGELESS nodes that are under the control of the same E-ORC (just a ping would be needed for this).  
At this point if we have a workflow that defines our application, data will be produced on the RPI but externally of course from the EDGELESS node deployed on the same device. The E-ORC, if able to decide where to relocate the execution of a certain function, could take into account some metrics about the RPI, collected by the E-ORC agent on the local EDGELESS node (e.g., power level), and decide (using such data) to switch policy to aim to (e.g.) minimize the client’s power consumption, and neglect (e.g.) latency and AoI. This could be done or by plugging in another orchestration policy, or by considering a single policy that, taking in input data about (e.g.) power level of the RPI, the load that other edge nodes have and their latency, and change something in this existing policy (e.g., we could have a weight associated to the power consumption, that becomes bigger as the battery level lowers).   
This allows the application to be developed in terms of workflows, and functions can be executed either locally or remotely in a transparent way to the user.  
We can measure if relocating a function is beneficial, via the power monitor.

Let’s spawn the E-ORC agent (i.e. the node) on the RPI, and let’s make it be orchestrated by an E-ORC, and deploy also another node on another device (Ubuntu VM is ok). The E-ORC can be spawned on the VM too. At this point we could compose a workflow of some functions, let’s assume they are three. We start by executing them all on the RPI. After this, we offload the second one on the node executing on the VM.

The author of the project says that the E-ORC is the component that decides on which node a function will be executed, so this should all be possible. He says we aren’t even obliged to modify the code because they already implemented a pattern called “**delegated orchestrator**” (similar to the Operator Pattern in K8s), with which we can have information and provide orchestration commands passing through a proxy realized on Redis. Here (<https://github.com/edgeless-project/cnr-experiments/tree/main/delegated_orc>) we find an example of delegated orchestrator that simply does a load balancing action (if it sees that some nodes have too many functions, it distributes them on nodes that have less, very basic).

**How does the orchestrator decide on which node a function will be executed?**The orchestrator uses the *next()* function (included in *edgeless\_orc/src/orchestration\_logic.rs*). The decision is based on the orchestration strategy defined in the configuration file for the orchestrator. The function returns the UUID of the chosen node, if one is feasible, or None if no suitable nodes are available. **Other strategies could be defined here.**

**Once a node is selected for a function execution, what happens?**There is the *start\_function\_in\_node()* function (*edgeless\_orc/src/orchestrator\_task.rs*) which takes care of attempting to spawn a function instance on a specified worker node. The function is started on the target node using the *start()* method of the node API (). The state of the orchestrator is updated to reflect the active instances.

**Performance and node health metrics collection:**One of the responsibilities of the orchestrator is the *proxy maintenance,* i.e. the optional synchronization of its internal data structures and performance metrics with an external database via a proxy. The database can then be queried by third-party services for monitoring purposes or to implement the delegated orchestrator concept (ref: <https://github.com/edgeless-project/edgeless/blob/main/documentation/orchestrator.md>)

**Edgeless\_orc/src/node\_register.rs:** implementsa *NodeRegister,* that manages node registration within the orchestrator. Node health metrics and performance samples are pushed to a *proxy.* The proxy is defined in *proxy\_redis.rs.*

**Edgeless\_orc/src/proxy\_redis.rs:** defines the *ProxyRedis* struct, which acts as a proxy between the Edgeless orchestrator and a Redis database.

* *Push\_node\_health*: writes a node’s health status (memory, CPU, disk usage, …) to Redis in JSON format under the key *node:health:<node\_id>*. Logs the data to a CSV file if file logging is enables.
* *Push\_performance\_samples*: writes function execution times or other performance metrics to Redis under the key *performance:function\_execution\_time:<function\_id>.* Logs the data to a CSV file if file logging is enabled.
* *Fetch\_node\_health*: fetches node health records from Redis.

**Edgeless\_orc/src/bin/proxy\_cli.rs:** implementation of a CLI for interfacing with the orchestrator’s Redis proxy. Allows the querying, displaying and the management of nodes, resources, performance data and migration intents stored in Redis.

**Edgeless\_node/src/node\_subscriber.rs:** Responsible for sending node-level health and performance data to the orchestrator (via gRPC).Periodically sends health and performance metrics from the node to the orchestrator. We have the *refresh\_task* function that runs continuously sending *Refresh* requests to the node at intervals defined by *subscription\_refresh\_interval\_sec. Refresh* will trigger the node to send updated health and performance data to the orchestrator.  
The health data collection is done via the *get\_health\_status* function: gathers CPU, memory, disk, network usage (using sysinfo) data, as well as GPU load and temperature data. The collected data is packed into *UpdateNodeRequest* and sent to the orchestrator via the node register API. **Other network metrics could be collected here.**

**Edgeless\_node/src/resources/metrics\_collector.rs:** focuses on function and workflow-level performance (execution times) for individual tasks. The metrics tracked here are the per-function execution times, workflow-level timing and exponential weighted moving average (EWMA) for function runtimes. The component tracks the start and end of functions/workflows, and sends direct Redis commands. This mechanism bypasses gRPC and writes directly to Redis.

**Operator pattern in Kubernetes:**The delegated orchestrator pattern in EDGELESS takes inspiration from this pattern. It may be useful to get to know the basics of such pattern.   
The *operator pattern* in Kubernetes is a method of automating the deployment, scaling and management of complex, stateful applications by extending Kubernetes’ capabilities beyond its core resources (like pods, services and deployments).  
How does it work? Operators introduce Custom Resources (CRs), that extend Kubernetes to manage non-native resources (e.g., DBs, message queues, etc.). A custom controller continuously monitors the state of the cluster, comparing the actual state of the resources with the desired state defined in CRs. If any discrepancy is found, the controller takes corrective actions to bring the system back to the desired state (e.g., through redeployment, rescaling, or via service restart). Moreover, users can declare their needs (e.g., “A 3-node Redis cluster”) and the Operator ensures this desired state is maintained automatically.  
How does this relate to EDGELESS’ Delegated Orchestrator? In EDGELESS, the Delegated Orchestrator follows a similar philosophy: it delegates orchestration tasks to the E-ORC, but centralizes the control. A Redis DB is used as message bus (proxy) to facilitate communication among Delegated Orchestrator, E-ORC and nodes.

**Delegated orchestration (ref: https://github.com/edgeless-project/edgeless/blob/main/documentation/local\_orchestration.md):**The feature requires an external Redis in-memory DB.   
This Redis DB is used to:

* Mirror the internal data structures of the E-ORC: updated periodically by the E-ORC and read by the delegated orchestrator to take its decisions
* Receive orchestration intents: once the delegated orchestrator has taken a decision, it informs the E-ORC by updating the DB with its *intents*, which will be promptly enforced, if possible.

Note: the DB is flushed by the E-ORC when it starts.  
The Redis proxy can be enabled via the specified section in the *orchestrator.toml,* in which the URL of the Redis DB must be specified.

The E-ORC’s internal status is serialized to Redis by means of some entries. The ones we might be interested about are the following (**more keys about network metrics could be defined here**):

* *nodes:capabilities:UUID*: JSON object representing the capabilities of the node having that UUID (serialization of the *NodeCapabilities* struct)
* *node:health:UUID*: JSON object representing the health status of the node having that UUID (serialization of the *NodeHealthStatus* struct)
* *performance:function\_execution\_time:UUID*: list of function execution times of the function with the given physical UUID (serialization of the *NodePerformanceSamples* struct)

Currently, only one type of intent is available, which allows the delegated orchestrator to migrate one function instance from its current node to another (if possible, otherwise the request is ignored).

How to migrate? Assume we want to migrate the function that has the logical identifier FID to the node with identifier NODE. The delegated orchestrator needs to updated two keys in the Redis DB:

1. Must set the *intent:migrate:FID* key to NODE
2. Must append the key to the list *intents*.

Multiple intents can be submitted at once, and will be processed in FIFO order.

**Setup of a delegated orchestration EDGELESS system:** the system as described in the last meeting has some criticalities. We want to setup a system that uses two nodes: one on the RPI, and one on the VM. The VM also hosts the orchestrator, that orchestrates both nodes. We want to use the *delegated orchestrator* pattern, to send custom migration intents to the orchestrator, based on network metrics and computational load of the nodes. Latency is a key metric in this setup: the ORC is able to compute the latencies from himself to the two nodes because the nodes register to him, and he for sure knows their addresses, and with just a *ping* operation this gets done.

**ASK THESE POINTS:**

* Where do we spawn the delegated orchestrator? **The VM should be OK**
* If the ORC needs to compute the latencies between himself and the two nodes, what about the node on the VM? It is hosted ono the same machine as the ORC, so the latency will always be much lower to reach that node from the ORC. **Tricks to manually augment latency here could be used.**
* **A consideration:** an EDGELESS workflow starts, gets executed, and then ends. As a sample application me will most likely need one that does a continuous analysis of something, like accelerometric data. An option could be that to read continuously from a file (or continuously generate) data from an accelerometer dataset, already available, and process such data in batch. We could analyze the samples, correct them, extract features, classify them, show an output (these already might be the functions of the workflow).

***The development process involves some steps:***

1. DEFINE SYSTEM REQUIREMENTS
   1. Identifying the metrics to be considered in the analysis
      1. Power metrics: power usage of the nodes (written to Redis)
      2. Latencies: time taken for the orchestrator to communicate with each node
   2. Decision-making criteria
      1. Defining how power metrics and latencies will be weighted (e.g., nodes with lower power consumption might be prioritized, nodes with higher latency might be deprioritized)
   3. Expected I/O
      1. Input
         1. Power metrics (written by nodes)
         2. Latency values (calculated by the ORC) **Done**
      2. Output
         1. Best node to execute a function
2. REDIS DATA STRUCTURE DESIGN
   1. Defining keys and values
      1. Nodes write their power metrics to Redis
      2. ORC stores latency to each node **Done**
         1. key structure: latency:<FROM>:<TO> (e.g., latency:ORC:RPI\_node)
3. IMPLEMENTATION OF THE METRICS COLLECTION MECHANISM
   1. Writing nodes’ power metrics to Redis. Here, potentially we won’t need to modify the EDGELESS node’s code, but we could create a Rust script that runs on the VM and RPI at the same time as the nodes, collects power usage data, and periodically sends updated at the Redis server. We can pretend this code is already part of the Node’s.
      1. **How can we measure the VM node’s power metrics?**
      2. **How can we measure the RPI node’s power metrics?**
   2. The ORC periodically sends *ping* requests to nodes to measure the latency, and push such data to Redis too. Potentially, also for this step a custom new script could be designed, and pretend that it’s part of the ORC. **Done**
4. DELEGATED ORCHESTRATOR IMPLEMENTATION
   1. Query Redis for metrics
   2. Combine metrics
   3. Decision logic
   4. Push intents?

**Meeting 14/01/2025: what to do next?**The proposed setup is good. Ok to test this simple workflow in which the RPI generates data which is some sample random accelerometric data, we extract some features and classify the results.  
**TODO:** formalize the setup. What does this mean? Try to make it as realistic as possible (e.g., for now, the classification result is pushed on Redis, while in reality it would likely be needed back again on the RPI, so do like this). Spawn the other node on another VM, so that we do not have everything in one place (my pc).  
**TODO:** implement the delegated orchestrator. For now, it’s ok to make it decide based upon only the latency measurements, using a simple threshold. Then, we need to implement more sophisticated decision strategies, one of which could be that to base ourselves also on a historical latency data, and predict based on such. **TODO:** test the workflow, gather some numbers

**Building EDGELESS on RPI:** 1GB swap file is too little to make the build process finish without complications, and for this reason we enlarge the file to 4GB with the following commands. These will create a larger swap file, ignoring the dphys-swapfile system limits, which limit its dimension to 2GB:

1. Disable and remove current swap:
   1. *sudo dphys-swapfile swapoff*
   2. *sudo rm /var/swap*
2. Create new swap file manually:
   1. *sudo dd if=/dev/zero of=/var/swap bs=1M count 4096* (for a 4GB swap file)
3. Set permissions:
   1. *sudo chmod 600 /var/swap*
4. Format the swap file:
   1. *sudo mkswap /var/swap*
5. Enable the new swap:
   1. *sudo swapon /var/swap*
6. Make the swap persistent:
   1. *Sudo nano /etc/fstab*
      1. */var/swap swap swap defaults 0 0* (add this line at the end)
   2. *Sudo systemctl disable dphys-swapfile*

To actually build EDGELESS: *CARGO\_INCREMENTAL=0 cargo build --release --jobs 2 --no-default-features*

* Note that since we built the repo with the *release* flag, we won’t have any *edgeless/target/debug* folder, but we’ll find everything in *edgeless/target/release*

To build wasm-opt: *CARGO\_INCREMENTAL=0 cargo install wasm-opt --jobs 4*

**VM + RPI experiment, two nodes:** we spawn the e-con, e-orc and a node on the VM. This constitutes a minimal EDGELESS cluster. Before this, we modify the *orchestrator.toml* file and the *node.toml* file (both on the VM and on the RPI):

* *orchestrator.toml: node\_register\_url* gets changed to <VM\_ip>:7004. This will be the port on which the orchestrator will listen for new node registrations.
* *Nodel.toml: node\_register\_url* gets changed to <VM\_ip>:7004. This will be the IP address and port the node will use to connect to the orchestrator.

We then can launch the node on the RPI (*./target/release/edgeless\_node\_d*). We’ll see some output on the orchestrator console, showing that the node has successfully been added to the cluster.

# Trying out TinyFaaS:

**Required dependencies:** Go (>= v1.22), Docker (>= v24), make.  
**Installation:** move into the cloned git repo and **make build  
Starting service: make start.** This will start the TinyFaaS service on port 8080.  
TinyFaaS provides a simple interface to upload and run functions. All the necessary scripts are contained in tinyFaaS/scripts.   
**Uploading a Python function:** in general, the command to upload a function to TinyFaaS is by using the **upload.sh** script. We do **./tinyFaaS/scripts/upload.sh “<FOLDER>” “<NAME>” “<ENV>” “<THREADS>”,** where <FOLDER> is the folder name in which the file containing the function is contained, <NAME> is the name of the file containing the function, <ENV> is the execution environment (python3, nodejs or binary are supported), and <THREADS> is the number of workers that’ll be employed to execute such function.  
Talking specifically about Python, the function to be executed must be provided in a folder that can have whatever name, but must contain a file named **fn.py,** and a **requirements.txt** file, that needs to be included even if the function does not have any requirement (in that case it will just be a blank txt file). The python file must have a **fn** method that will be the one called every time we’ll invoke such function.   
Assume we have the following directory structure: a python\_test folder containing the fn.py and requirements.txt files. The python file is the following:

import typing

def mul():

a = 12

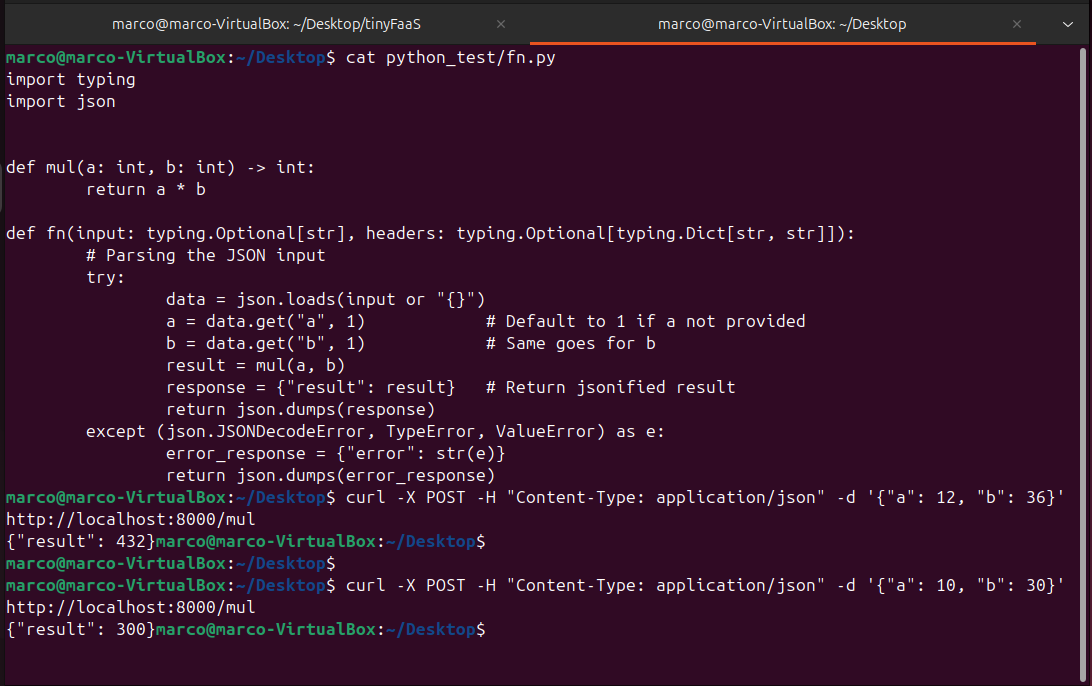
b = 36

return str(a \* b)

def fn(input: typing.Optional[str], headers: typing.Optional[typing.Dict[str, str]]):

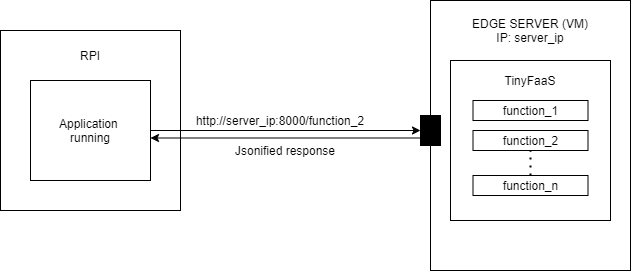
res = mul()

return res

**Invoking functions:** we can do something like **curl “**[**http://localhost:8000/<function\_name>**](http://localhost:8000/%3cfunction_name%3e)**”**. If we do like this for the **mul** example, we’ll get a string response saying “432”, which is indeed the result of 12x36.  
So, it’s just a function thet executes the multiplications of two hardcoded numbers.   
A **fn** function seems to always have to be present.  
We can upload such function with the following command:   
**./tinyFaaS/scripts/upload.sh “./python\_test” “mul” “python3” 1**So if now we list the available functions, we’ll see the **mul** one. After uploading, the cmd shows how to reach the uploaded function (for HTTP it’s usually port 8000).  
 **Passing arguments to functions:** arguments can be passed with the POST method like so:  
**curl -X POST -H “Content-Type: application/json” -d ‘{“<arg1-name>”: <arg1-value>, “<arg2-name>”: <arg2-value>, …}’ http://<tinyFaaS-server-IP>:<port>/<fn-name>**  


**About setting up an application that could use such setup:** we must (I believe) think beforehand if an application we’re developing is going to have its functions offloaded. Since TinyFaaS uses a precise format to upload, invoke and delete functions, the application should be packaged accordingly to have folders and functions like described above. Deciding if an application’s functions will need to be potentially offloaded is not a very complex task if we take into account what the application does and what is the client that will run it: if the resources are constrained (like when we’re operating on a wearable device), it could be a good idea to offload.  
At this point, we have some alternatives for when the application is running:

* We can decide to upload all the functions of the application that we tagged as “offloadable” (i.e. those that are destined to be run on the edge server whenever there’s the need) when the application gets installed on the device. We get a longer installation time but then we know that while the app runs, we won’t have to upload any function at runtime and then invoke it right after its upload, since they are all already uploaded, and just need to be invoked at the time of need. This introduces yet another challenge, because TinyFaaS does not offer an API to upload functions from hosts which do not run TinyFaaS (TinyFaaS needs to be running on the host to be able to upload functions), so from the RPI we won’t be able to upload anything to the TinyFaaS instance running on the VM), and we must upload everything manually. If we decide to offload functions from an existing application, functions need to be adapted so that they are compliant with the format TinyFaaS requires. We could write **wrappers** to such functions so that the logic is the same and we adapt them to JSONify the parameters and de-JSONify the response from TinyFaaS. Wrappers could have the format of the mul example above.
* We can decide to install the application normally, without uploading any function at installation time. At that point, it’s obvious that whenever we’ll have the need to offload one of them, it’ll first have to be uploaded on the server, and this has to be done at runtime. This will take some time during the app execution. Of course, the benefit is that installation time will be shorter than in the first method, and time will be “lost” only because of the upload of the function (when the function is uploaded, it remains uploaded, as long as the server is not turned off). Based on the challenge we highlighted in the previous point, this option will only be viable if we find a way to upload functions from the client to the server, with the client not running TinyFaaS.
* We can install the app normally, without deciding to upload any function on the server at the moment. We make the app also do its first run without any offload strategy, and we monitor the invocation frequency of the app’s functions (for example, by counting the number of times they get called), and at app closing we order the gathered data to get (e.g.) the top X most frequently invoked functions (or we can do it by understanding how much each function is energy-hungry on the device, even better). We then could run a background task that uploads this these X functions on the server, so that at the second and beyond app usage, they’ll be invoked directly on the server and no additional time will be needed during installation or during execution. Of course, if offloading the functions will result in reduced power consumption, the first app usage will need more power, since no offloading is taking place (only data can possibly give us such insights). In this setup, the challenge highlighted in the previous points is a bit relaxed, as we’ll have the time to upload functions on TinyFaaS in the timespan between the two application executions.



**About the impossibility to dynamically upload functions using REST APIs:** we’ll likely need to preload functions on TinyFaaS (i.e. on the VM which acts as the server). This is a choice that should not affect the development and results of this thesis work, since we mostly focus on the advantages/disadvantages of offloading such functions to an external server in terms of power consumption and AoI, and the measurement experiments will likely be conducted while the application is running.

**Security problems of using this setup:** data travels using HTTP.

# Computation offloading:

**FaaS:** FaaS (Function as a Service) is a cloud computing model that enables developers to deploy and execute individual functions or pieces of code without needing to manage the underlying infrastructure. By abstracting away the server architecture, developers can focus directly and primarily on writing the application logic. Usually, in this scenario, functions are invoked by specific events (e.g. HTTP requests, DB updates, …), and the server executes such functions only when the event occurs.

Different frameworks that enable computation offloading have been taken into consideration in this work. Based on the analysis of such, we can divide the options in two main categories: those that use WebAssembly and those which don’t. The first class of frameworks try to leverage the power of WebAssembly to provide a high-performance environment for functions execution. Examples of frameworks belonging to this class are Faasm and Sledge. Such frameworks, due to the usage of this technology, automatically rule out interpreted languages such as Python, the one we’re using in this work, unless we decide to cross-compile our Python functions using additional frameworks like Pyodide. This means that first, each function that has been tagged as ‘offloadable’, would have to be compiled to WebAssembly, and then uploaded to the chosen framework to be called later on.  
The choice of which framework to use in this work resorted then to an option that could natively support Python functions offloading, this being for example TinyFaaS.

# Sample client application:

We need to find an application to be ran on the RPI that involves some tasks that are resource-intensive and/or time-demanding, and understand if offloading the functions is a good idea or not.

**NOTE:** if we adopt the EDGELESS mentality, we can just offload functions that will return intermediate results, we are not limited to the offloading of the whole flow (this means we can write an EDGELESS-compliant copy of *each* function)

Potential candidates:

* Image classification with pre-trained models. Classification involves heavy computations.
* Video processing: frame-by-frame processing is computationally expensive (e.g., applying a filter to each frame)
* Data encryption: strong algorithms like AES or RSA are computationally heavy. These computations could be candidates for offloading. Moreover, encryption is crucial in real-world scenarios.
* Security applications:
  + Password cracking simulator: RPI orchestrates the task, manages the password lists and reports results. Can itself crack passwords or decide to offload the task to the edge server. Also, the dictionary attack or rainbow table variations could be an idea.
  + Cryptographic keys generation: the RPI generates secure cryptographic keys for data encryption or sign. Parts of the process (e.g. generating very large prime numbers) are either executed locally or offloaded. The edge server, if required to, will perform some of these computations and hand back the results to the RPI.
  + Steganography analyzer: the RPI receives images or audio files and checks for hidden data inside them (e.g. in pixel configurations or frequency patterns). Computationally heavy tasks will in this case be Fourier transforms or pixel-by-pixel analysis.
  + Password strength evaluator: the RPI runs an interface to evaluate passwords strength.
* Data compression: we could assume the RPI is able to capture videos (we pass a video file to it and pretend it captured it itself).
  + The RPI could start compressing the video frames by itself but then decide to offload the compression task to the edge server

# Experimental setup:

**RaspberryPI:** acts as the client device in our experimental setup.We’re going to use a 3B+ one. **Why?** Because it has an Ethernet port which is useful in our case, since we need to synchronize the clocks via NTP. Also, its power usage is less than a version 4 or 5 RPI.  
Programs can be passed to the RPI via SCP and can be started via CMD or the RPI graphical interface.

**What happens in terms of power usage and AoI when a function is executed locally on the client VS when it gets executed on one of the edge platforms?**

Connecting the client to one of such servers hosting the platform add latency due to communication, but of course the server has a higher computational power than the client alone.

**How to add artificial delays on the network?** In an emulated way using ???, an emulator which runs in a Linux environment in which we can setup simple rules to add artificial delays to outgoing packets. This allows also to evaluate scenarios in which the server offering computational power is closer or further away from the client. This is significant since we also need to take into account the fact that the transceiver of the client will stay in higher or lower power usage states also depending on the network latency. Typically, the higher the latency, the higher the power used to transmit data.

Then, we could setup **dynamic choice mechanism** to determine, depending on various factors, if it is optimal to offload tasks so that this action guarantees the best tradeoff between AoI and power usage.

**What about NTP?** Needed to synchronize the clocks of RPI and notebook. RPI will be the client and the notebook will be the server. This is needed to get valid AoI measurements, and we’ll use an ethernet cable to connect RPI and notebook so that latency that WiFi would introduce will be zeroed. (bachelor thesis for a reference)

**If Faasd gets chosen as the offloading option, this might be the setup for offloading:** faasd offers an PI that allows to interact with it programmatically in Python. The setup uses faasd’s REST API that provides endpoints for deploying functions (POST /system/functions), invoking functions (POST /function/<function-name> or GET /function/<function-name>), managing functions, querying system information. The base URL for faasd is typically http://<faasd-host>:8080. We can integrate this mechanism in Python using the **requests** library:

import requests

# faasd details

faasd\_url = "http://<faasd-host>:8080"

username = "admin"

password = "<your-password>" -> /var/lib/faasd/secrets/basic-auth-password

# Function deployment payload

function\_payload = {

"service": "hello-python",

"image": "ghcr.io/your-username/hello-python:latest",

"envProcess": "python3 index.py",

"network": "func\_functions"

}

# Deploy the function

response = requests.post(

f"{faasd\_url}/system/functions",

json=function\_payload,

auth=(username, password)

)

if response.status\_code == 202:

print("Function deployed successfully!")

else:

print(f"Error deploying function: {response.status\_code} - {response.text}")

Then to invoke a deployed function:

# Invoke the function

data = {"name": "World"} # Example payload

response = requests.post(

f"{faasd\_url}/function/hello-python",

json=data,

auth=(username, password)

)

if response.status\_code == 200:

print(f"Function response: {response.text}")

else:

print(f"Error invoking function: {response.status\_code} - {response.text}")

Related to our setup: when offloading is required (it’s the RPI that decides it), we’ll deploy the function to the faasd server via the API (if not already deployed) and then invoke it. The Ubuntu VM (i.e. the edge server) will have the duty to run faasd with deployed functions, and accept requests from the RPI to deploy or invoke them.

**Considerations:**

* **Deployment overhead:** deploying functions dynamically at runtime might introduce latency. One solution might be to deploy functions that we know are used frequently in advance.
* **Caching:** if functions are often invoked, we should avoid re-deploying them
* **Security:** all this goes over HTTP

# BUILDING THE SETUP:

**Components:** RPI 3B+, power monitor, ethernet adapter, ethernet cable from PC to RPI, cables from power monitor to RPI, cables from power monitor to PC, SD card for RPI OS.

**RPI credentials:** [pi, raspberry]

**Copying files from Windows to RPI:** using SCP. From **Ubuntu** app in Windows navigate to **/mnt/c/Users/Marco/Desktop** and pass files via **scp <filename> pi@<RPI-IP-address>:/home/pi/Desktop**

* **Otii – RPI connections:** power supply from Otii to RPI. UART channel: GND from Otii GND pin to pin 6, 9, 14, 20, 25, 30, 34 or 39, TX from Otii TX to pin 10 (GPIO15), RX from Otii RX to pin 8 (GPIO14). Remember to hot the power on button on the Otii program to power up the RPI!
* **RPI –** **PC connections:** Ethernet cable from RPI to Ethernet PC adapter. HDMI connection if GUI needs to be used.
* **Otii power supply**
* **Otii – PC connections:** power supply connection from Otii to PC
* **SD card:** insert into RPI socket

# Power Monitor – QOITECH:

**User manual:** <https://docs.qoitech.com/user-manual>  
**Otii Arc Pro power monitor specs:** <https://docs.qoitech.com/user-manual/otii/hardware/otii-arc-pro>  
**Python scripting:** <https://docs.qoitech.com/user-manual/advanced-guides/python-scripting>

# NTP server:

**Why do we need to setup this?** Check Cristofani’s thesis work.

* **NTP server:** hosted on the PC. Need to setup it as a server and synch. It with a higher-strata server. Commands must be run as admin:
  + Stopping the windows time service:
    - net stop W32Time
  + Modifying the registry settings to configure the PC as an NTP server and specify the server to synchronize with:
    - reg add “HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/services/W32Time/Config” /v LocalCLockDispersion /t REG\_DWORD /d 0 /f
    - reg add "HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/Services/W32Time/Parameters" /v LocalNTP /t REG\_DWORD /d 1 /f
    - reg add "HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/services/W32Time /TimeProviders/NtpServer" /v Enabled /t REG\_DWORD /d 1 /f
    - reg add "HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/services/W32Time/Config" /v AnnounceFlags /t REG\_DWORD /d 5 /f
  + Setting the Windows time service to start automatically:
    - sc config W32Time start=auto
  + Starting the Windows time service:
    - net start W32Time
* **NTP client:** (see Cristofani thesis)
  + Disabling automatic synchronization techniques:
    - $ systemctl disable –now system-timesyncd
    - $ systemctl disable –now ntp
  + Installing the command to enforce synchronization:
    - $ sudo apt-get install ntpdate
  + Manually triggering the synchronization:
    - $ sudo ntpdate <nt-server-address>

# THESIS SUMMARY:

1. Abstract
2. Introduction
3. Background
   1. Edge computing, FaaS
   2. Age of information, latency, how these are collated with power consumption of client devices
   3. Motivations
4. Experimental setup
   1. Setup
   2. Automation script
   3. Client and server tools
      1. Client sample program
      2. Serverless framework proposals
         1. OpenFaaS
         2. TinyFaaS
         3. EDGELESS
      3. Tested serverless frameworks: two solutions at opposite edges of the spectrum
         1. TinyFaaS
         2. EDGELESS
      4. EDGELESS serverless framework
         1. System overview
         2. Why it fits our needs
         3. Experiments that lead to the final working setup