Technische Universität Berlin

Fakulät IV

Supervisors: David Bermbach

Tobias Pfandzelter

WiSe 2020/2021

Mercury

Project summary

Alexander Fritsch 402290

Amit Nautiyal 408787

Simine Kashi 452996

Xin Cheng 415342

2022-08-17

Contents

1 Introduction [Amit]

In the following chapter, we will describe our motivation, the overall scope and the general framework of our Mercury DSP project.

1.1 Motivation

In current times, modern applications are being deployed on cloud services. But with the current advancements in domains like *5G* mobile application development, *Internet of Things (IoT)*, cloud applications somehow are not able to handle these domains due to factors like latency, limitation of bandwidth between the sensor and cloud [4] etc. So, we have noticed a push in merging fog computing along with cloud services [1] [3] [2], to combine edge computing with cloud computing to cope up with these challenges. The emerging Fog computing paradigm offers better quality-of-service.

In Fog Computing, resources in the cloud along with compute and storage within the network, in intermediary nodes or at the network edge, are used as the application platform. All these nodes have different capabilities in terms of their attached storage, compute or memory. Distributing these components across the network can lead to improved access latency (e.g., a client sends a request to a nearby server instead of to the cloud), can save bandwidth (e.g., by filtering data that comes in from IoT sensors at the network edge), can increase privacy (e.g., data is saved only on nodes in a specified location instead of being sent to the cloud), and more. But with these benefits, we also have a challenge i.e. the applications now are multi-tenant in a heterogeneous environment and are to be distributed over a widely distributed Fog Platform. So, unlikely deploying an application traditionally over a single cloud platform, the applications now have to deployed over various compute nodes, with the business logic spitted over different areas of the network. And we need need to make this application deployment in the Fog as easy as possible.

1.2 Objective And Scope

Our task is to build the *Mercury* toolkit to deploy Fog applications onto the FogStore platform. To make this application deployment feasible we need to provide an abstraction to application developers. So at our disposal, we have two-component *tinyFaaS* and *FReD*. *tinyFaaS* [6] is a *function-as-a-service (FaaS)* compute platform that lets developers deploy code of serverless functions onto Fog nodes. Each *tinyFaaS* instance is autonomous and tied to a single physical host. Applications are deployed onto it in terms of functions. A function is a piece of stateless code, it cannot store any data, all data that it needs has to be passed with the event that comes into *tinyFaaS*.
The other part is *Fog-Replicated Datastore(FReD)*. *FReD* is the data management platform, it cannot perform any computation but stores data in the Fog. The data format of *FReD* is a collection of key groups. Each key group is a group of items, each a key and value and Clients can read, modify, and write items in these key groups. These two components are integrated using triggers. A trigger can be defined for a *FReD* key group on a specific *FReD* node. When an item is created or modified in that key group, an event is sent to that trigger node. In effect, this trigger node can be a *tinyFaaS* instance. This means that modified data can automatically be sent to *tinyFaaS* to perform a function.

So, our task is to build the Mercury toolkit which can deploy the Fog applications onto the *FogStore* platform. To attain this, *FReD* and *tinyFaaS* needs to be set up in a given Fog environment by extending the existing tinyFaaS or *FReD* with new features. It is required to build the application descriptions via JSON [5] based test file, a *command-line-interface(CLI)* and a *graphical user interface(GUI)*. Further, the mercury tool is to build in such a way that it takes the application descriptions, understands them, and deploys the key group configuration to *FReD* and functions to *tinyFaaS*. And thus a frontend is to be developed to create these application descriptions.

Our scope will include the building of application descriptions through a JSON file, CLI, and interactive front end(GUI). And any modified data in the key group (either while creating or

modifying it) can automatically be sent to *tinyFaaS* to perform a function. This completes the distributed storage and computes platform for the Fog.

1.3 Structure

Chapter 2 describes the requirements for our product. It is separated in functional and non-functional requirements.
Chapter 3 show our thoughts on the concept and design of the application. It gives a first a detailed description of the product components, describes our approach and illustrates a first high-level architecture.
Chapter 4 provides insight into the implementation part of our work. Important aspects of our implementation will be shown here.
Chapter 5 takes up the objectives and evaluates whether we have achieved them.
Chapter 6 summarizes the project work, describes the problems that occurred and gives an outlook on future work.

2 Requirements [Xin]

For this section, the project desired outcome features will be subdivided into *functional* and *non-functional* requirements. Functional requirements define the basic system behaviour, which specifies what the system does or must not do. Functional requirements are product features and focus on user requirements. While non-functional requirements specify how the system should do it. Non-functional requirements are product properties and focus on user expectations, and covering the system performance, usability and scalability.

2.1 Functional Requirements

1. Configuration: the main purpose of the project is to build 3 ways to realize easy configuration of *FReD* and *tinyFaaS*, which require building a backend service to deploy, manage and update all components of the application to the correct *FReD* and *tinyFaaS* nodes.
2. IaC: ease of deployment and management with the configuration file, here we use JSON. The configuration should take the form of a JSON file, which is easy to edit, copy, and distribute, and conduct the configuration process.
3. CLI deployment: the project should provide CLI to realize the ease of use, the CLI application should have basic comments available to perform configuration. One main command should be FReD <*pathOfConfigurationFile*> which should require configuration JSON file path as a variable, and based on what defined in the JSON file to realize create key groups, configure replica nodes, deploy functions to *tinyFaas*, and configure trigger nodes.
4. GUI: build a web interface to visualize and design applications using drag and drop components and custom data transformation code, which should enable the user to control the component placement and conforming to data movement, showing the infrastructure as data flow alike format.
5. Extend *FReD* trigger node: when replica nodes receive the new update data, the request should be sent to *tinyFaaS's reverse proxy*, which acting as the trigger node. To realize which, *tinyFaas* GRPC[7] gateways should be extended.
6. Connection network: the network between FogStore, backend service, frontend service, *tinyFaaS* mgmt service and reverse proxy should be built, which ensure the internal communication between docker containers to realize the configuration and functionality. *FReD* and *tinyFaas* internal connections are already built, so we need to ensure Front End and Back End are connected, also the intermediate connections between them are well established.

7. Provable: Build and deploy one application for demonstration.

2.2 Non-Functional Requirements

1. Deployments: the implemented IaC should be easy to deploy to a new environment, which should automatic the installation of required packages, software, environments as much as possible.
2. Utility: the configuration process should be easy and scalable, the new application should be easy to deploy in either of 3 ways.
3. Traceability: clear logs should be provided for debugging and issue tracing, also the correct status of the new deploy should show logging capabilities for checking the status of the new deploy.
4. Responsiveness: the front-end should be react-app, which will ensure ease of modification in further process, also the web application should have fast responding when there are changes from the user, to optimize the performance from the front end.
5. Self-explain: the code should have necessary comments for developers to understand, and the configuration file content should be easy to understand and modify. Relationships between *FReD*, *tinyfaas* and iacTest should be documented, so they can easily be extended by other teams working on the project.

3 Concept & Design [Simine & Alexander]

3.1 Mercury basis : FogStore

FogStore is a platform aiming to enable easy deployment of applications in the Fog, thanks to the abstraction it brings. FogStore can be defined by its two different parts, tinyFaaS and FReD (as Fog-Replicated Datastore). FReD handles the data management and the storage of the data in the

Fog, whereas tinyFaaS enables to compute this data. The application developers can deploy the code of serverless functions into the Fog nodes thanks to this lightweight function-as-a-service platform.

FReD

To manage the data and its placement in the Fog, FReD uses *key groups*. Each keygroup is a group of items described by a key and a value. Items can be read, modified or written into the keygroups. Multiple FReD nodes can be part of the same keygroup, at the same time. This way, data is automatically replicated in all the keygroup's nodes. The *APIs* provided by FReD allow us to change the composition of a keygroup or the nodes settings such as their expiry. A trigger can also be defined in each keygroup, upon a chosen FReD node. When an item is changed (either created or modified) in one of the keygroup's nodes, the event is sent to this trigger node. The latter can be a tinyFaaS instance, so the new data can be used in a function. The serverless functions of tinyFaaS can also write new data into FReD by using the provided APIs.

tinyFaaS

Applications which have to be deployed in the Fog also need to process data via functions. Thus, tinyFaaS is also important, as it is the computing platform of FogStore which allows it. Each instance of tinyFaaS is autonomous and tied to a single physical host. When an event is sent to one of them, the function, which is stateless, is invoked and runs once. The function itself cannot store data due to its statelessness. That is why, we need to pass all the data needed in the parameters of the function within the event. Each instance of *tinyFaaS* can also query *FReD* and store data outputs in *FReD* nodes.

FogStore is a complete platform to handle data processing and storage, in the Fog. However, it is still complex and long to deploy applications via FogStore, as the infrastructure setup has

to be done manually, node by node. The Mercury project is aiming to deploy fog applications in an easier and faster way, thanks to its Infrastructure as Code (IaC) Tool interpreting the client configuration file, which describes the wished infrastructure and needed functions. Mercury uses FogStore for the deployment, storage and processing of data of the client application.

3.2 Architecture

Regarding the architecture, we decided to have a linear data flow, which begins with a JSON file input and ends when the infrastructure deployment state is displayed either on the web application or on the terminal, depending on the interface the user chooses. The user has three different ways to communicate with the Mercury toolkit :

1. either the user chooses to upload the JSON file, via the web application interface named 'File Upload'
2. or the JSON file can be uploaded via the Command Line Interface (CLI), following some instructions provided on the web application
3. or finally, the user can choose to use the Graphical User Interface (GUI), available on the web application, which will then generate the JSON file internally

These interfaces are explained in more details in section 3.4.1.

To make this data flow works, the data from the JSON file has to be validated by our Infrastructure as Code (IaC) Tool. When this is done, our Application Programming Interface (API) interacts with FReD nodes, which are already available, and the tinyFaaS management service, which has the role to register serverless functions to tinyFaaS. Once this is done, our Connector has the responsibility to link the deployed FReD nodes to TinyFaaS instances depending on the user's will, which was described in the JSON file. A message is displayed to the user once the deployment is done. If any problems occur, the user is informed, especially if it concerns the way the JSON file is written, as our Validator checks if the file is well filled. To summarize, as

shown in figure 1, the three interfaces to Mercury can be found on our frontend, which then communicates with our IaC Tool. This tool communicates with FogStore, from our API for the infrastructure setup and from our Connector for the communication between FReD nodes and TinyFaaS serverless functions.

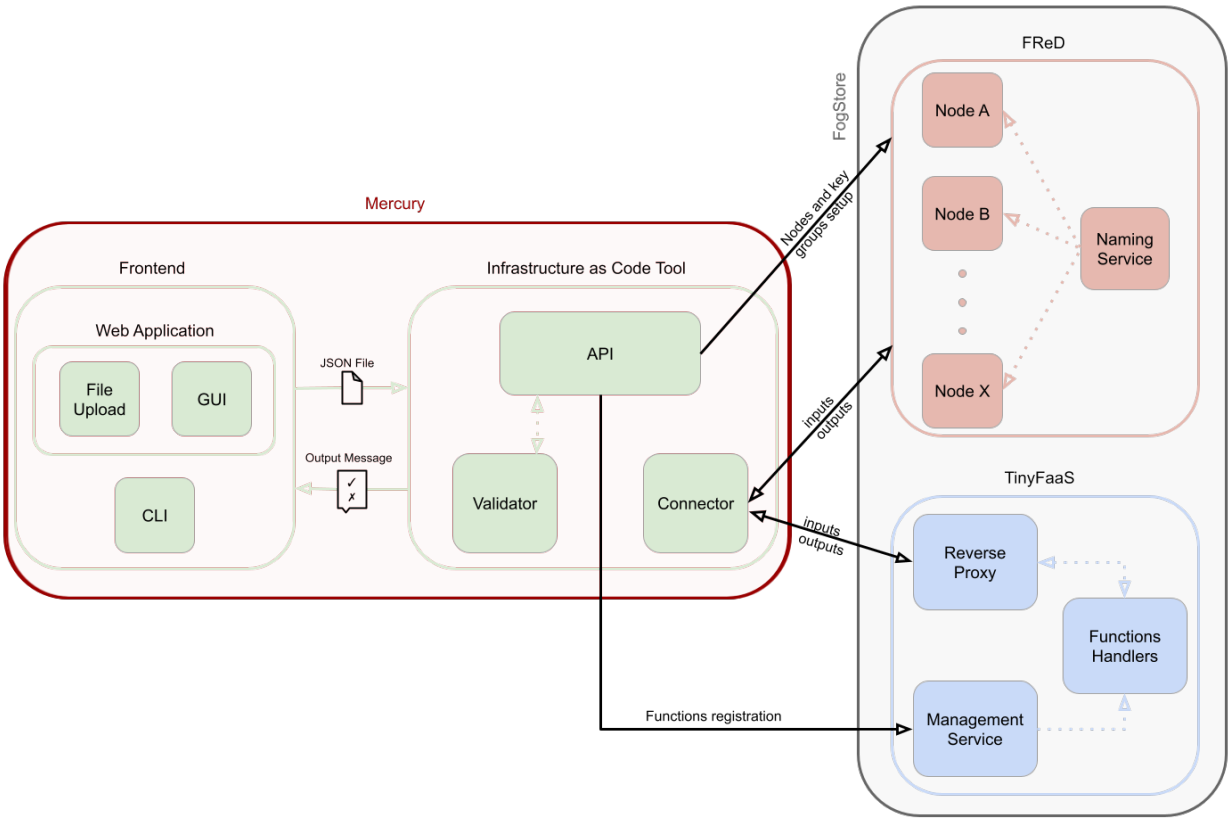

Figure 1: Final Mercury architecture with the connections to FogStore

3.3 JSON File

To model the infrastructure that you want to configure, we chose JSON. One of the great advantages of JSON is the simplicity of implementation and use. Parsers are available for many popular programming languages that allow JSON objects to be understood and then processed.

Either the developer writes the JSON file himself or he uses the graphical modelling tool provided by us. In this tool, the created topology is read and processed to a JSON. So even in this case, the configuration process involves processing a JSON object.

Below, figure 2 is an example JSON file. This creates a key group, with two replicas (nodeA, nodeB). In addition to this, it contains a tinyFaaS function called 'celsFahrConverter'. Within a JSON file, an unlimited number of key groups can be specified, provided that their names are unique. Within each key group, any number of functions and replicas can be specified, whereby both replica names and function names may only be assigned once.

3.3.1 Keygroup Definition

A keygroup can be thought of as a kind of shell that is then filled with replicas and functions. To create a keygroup, all that is required is a unique name and the attribute 'Mutable', which determines whether new values may be added into it. We recommend creating each keygroup at least initially in a mutable state, otherwise, it is impossible to store data in it.

The name must be specified as a string and the mutability as a boolean.

3.3.2 Replica Definition

Each keygroup should be equipped with at least one replica, otherwise, the storage of data is not possible. Once multiple replicas are specified, FReD can use them to exchange data between replicas. As the name implies, these simply replicate the data of the other replicas. This mechanism prevents data loss due to failed nodes. A replica can be in multiple key groups at the same time, which means that it stores the data of both key groups.

The duration for which newly injected data is stored on a single node can be set by the developer. If 0 is set as Expiry, the result is that, contrary to expectation, the data is temporarily stored without restriction (permanently). All numerical values other than 0 specify the expiry

in minutes. This allows the Mercury user to specify how long data should be stored on which nodes.

The name must be specified as a string and the expiry as an integer.

```
{
   "FReDTemplateFormatVersion":"2020-30-11",
   "Name":"myTemperatureFReD",
   "Nodes":{
      "RootAddress":"172.26.1.1:9001",
      "NodeNames": [
            "nodeA",
            "nodeB",
            "nodeC"
            ]
      },
   "KeyGroups":[
      {
         "Name":"Celsius",
         "Mutable": true,
         "Replicas":[
            {
               "Name": "nodeA",
               "Expiry": 1000
            },
            {
               "Name": "nodeB",
               "Expiry": 100
            }
         ],
         "Functions": [
            {
               "Name": "celsFahrConverter",
               "CodeURL": "https://github.com/alexfritsch10/dsp-tinyFaaS-examples/archive/main.zip",
               "subfolder_path": "dsp-tinyFaaS-examples-main/celsiusToFahrenheit",
               "threads": 1
            }
         ]
      }
   ]
}
```

Figure 2: Exemplary JSON file for defining infrastructure

3.3.3 Function Defintion

The creation of functions requires four attributes. The function name must be unique in all key groups. Two tinyFaaS functions cannot be registered with the same name. If the same functionality is to be guaranteed for several key groups, then the same function code can be used, but it must run under a different function name. The function code is referenced using two attributes, the GitHub repository URL and the subfolder path. The number of threads can be used to define how many tinyFaaS containers the specified function should run in. The function name, code URL and subfolder path must be specified as a string, the thread count as an integer.

```
module.exports = (req, res) => {

    const keyVal = req.url.split("/"); // -> ["", "celsius1", "15.8", ""]
    const key = keyVal[1];
    const value = keyVal[2];

    const celsius = parseFloat(value);
    const fahrenheit = ((celsius * 9/5) + 32).toFixed(2);

    const data = {
        "keygroup": "Fahrenheit",
        "key": key,
        "value": fahrenheit.toString()
    };

    res.send(data);
}
```

Figure 3: code example of a function that converts a value from Celsius to Fahrenheit

3.4 Frontend

The frontend of the application has to be accessible by modern web browsers providing a good user experience for users on a desktop computer. The user interface (UI) has to be simple enough that a new user can understand and interact with all UI components on the website intuitively.

According to the state of the art, we decided to develop a single page application (SPA) meaning that once the user loads the initial website, the browser never has to load any other web pages because the entire web application is already loaded. Using this method, only dynamic content needs to be transferred between the server and the users' browser resulting in a faster user experience. If the frontend is static (which it is when building a SPA), the web page can also be deployed independently of other components and thereby ensuring a higher availability if needed.

The UI should display two types of components.

The first type of components is those for configuring the FogStore. There are three of these components available to the user. Two of these allow web-based configuration, with the third simply providing instructions on how to configure FogStore via a CLI we have developed. From the available components of this type, the user chooses one, and only this one is displayed below. It is possible to change the chosen component so that different parts of the configuration can be handled by different components. The second kind of components is those for the interaction with the existing (before configured) infrastructure. Since during the interaction it must be exactly paid attention, into which Keygroup new data points are fed and which data flow thereby develops, it is helpful, if e.g. in the case of the graphic modelling, if the configuration component remains visible during the interaction. Accordingly, a maximum of two components is displayed on the website, one of the first type (Interface) and the second type (Interaction). This concept preserves a good overview and allows simplified work with the FogStore.

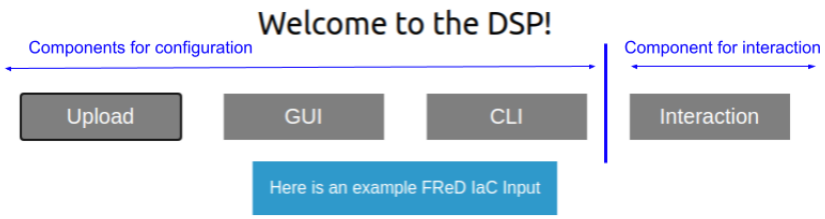

Figure 4: Different components for the UI: configuration and interaction

3.4.1 Interfaces

1. File Upload: This interface allows the user to select a local JSON file in which the desired configurations are defined, upload them, and have Mercury execute them.
2. CLI: The Command Line Interface also offers the possibility (similar to File Upload) to let Mercury process a local file. In our Web App, you will find instructions on how to install the CLI on your PC and thus access Mercury's functionalities. In addition to uploading a local file, the CLI also provides the ability to work with the pre-configured infrastructure, injecting data points into the various key groups and reading them out.
3. GUI: This graphical user interface allows the user to model the desired configurations and assignments using an intuitive GUI, upload them and have Mercury execute them. Available elements are key groups, replicas and functions. For each of these elements, different attributes must be defined for the configuration to take place. To represent the data flow, the GUI provides options for assigning and mapping. Thus, multiple replica nodes, as well as multiple functions, can be assigned to a key group. As soon as the user has completed the configuration, he can have the created topology processed.

3.4.2 Interaction

The ability to interact with the existing infrastructure, store and read data points is provided in the file upload and GUI via the fourth component of the web app, the interaction component. In the arranged unit of the application, simple commands can be used to insert new data points into the respective key groups and to access existing data. Depending on which configuration the user has chosen, a tiny-FaaS function registered for a replica node may be executed to process the new data point when new data points are inserted.

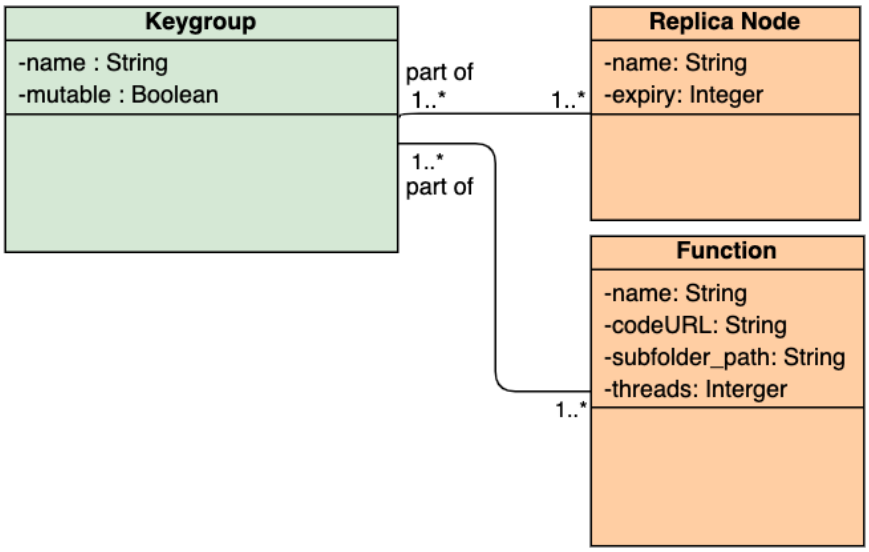

Figure 5: Representation of the topology of a graphical modeling

3.5 Backend

3.5.1 Application Programming Interface

To go beyond configuring infrastructure to providing the ability to read and write data, an interface is needed for standardized and efficient communication between the front end and the back end. The REST API must provide the ability to process JSON files by processing the topology described therein. Processing includes creating key groups, adding replicas to key groups, and registering and linking tinyFaaS functions. Also, it must be able to handle requests for interaction. This includes inserting data points into specific key groups and reading data from a specific key group. That is why we decided on three endpoints:

1. Configure Infrastructure

   This endpoint receives a JSON object/file which it should execute. Before the actual execution, it is to be checked whether the received data correspond to all guidelines and the necessary attributes of the Keygroups, Replica Nodes and functions were indicated at all

and in the correct data type.

Provided that the file fulfils all regulations, the execution of the same begins. The tinyFaaS functions are registered and the key groups are created with the specified nodes.

2. Get Data

   This endpoint is available to the user to read the values of individual data objects from the FogStore. The parameters Keygroup Name and Data Object Key are necessary for this.

3. Insert Data

   This endpoint is responsible for inserting a new data object into a specific key group. Necessary parameters are key group name, data object key and data object value. Optionally, the ID of a tinyFaaS function can also be specified, from which the new data object is processed during insertion. If this is not specified, the new data point will only be stored in the key group and will not be forwarded to a tinyFaaS function.

3.5.2 Connector

Both FReD and tinyFaaS are already existing and independently functioning systems. They need to be connected with the help of another component, the connector. The connector is an implementation of the trigger provided by FReD. You can define triggers for each key group, which will be triggered as soon as a new data object is inserted into it or removed from it. For our use case, only the case of insertion is relevant. The task of the connector is to check whether a tinyFaaS function should be called (the handler attribute is set during insertion) on the newly inserted data object and, if necessary, to forward it to this function. The forwarding is potentially done to the tinyFaaS reverse proxy, which then executes the corresponding function with the new data object.

As soon as the result of the tinyFaaS function has been forwarded to the connector by the tinyFaaS reverse proxy, the connector can inject the newly calculated value into a key group

defined in the tinyFaaS function. Potentially, with this insertion also a handler could be specified, to which the datapoint has to be applied.

3.5.3 Validator

To be sure the setup of the infrastructure can be done efficiently, the JSON file input has to be correct and coherent. We decided to implement a Validator which checks different aspects and layers of the input or generated file. First, the type of the values are checked to see if there are of type list, string or boolean, e.g. if it is a node name then a string is required. The typography of the keys are also checked, as a specific model is expected and can be found on our web application in the 'example input'. Furthermore, It also checks whether the nodes or functions described in each keygroup are correctly defined beforehand in the part listing the available nodes or functions, respectively. Finally, it checks the uniqueness of each keygroup name or node name or function name, as it is important to make the deployment works correctly.

4 Implementation [Simine & Alexander]

4.1 Methodology

We decided to split the team into two groups of two people, in order to work efficiently.

We planned to work on the backend first. The well understanding of FogStore was important before implementing our API. Once the foundations of our backend was developed along with the Validator file, we were able to configure some already-available FReD nodes. Our web application was created afterwards, so we could have a clear user interface to uplolad and process our JSON file example. Furthermore, we were able to register functions on tinyFaaS, from a git URL described in the JSON file. We also established a connection between FReD and tinyFaaS thanks to our Connector, so data from FReD could be pass to some serverless functions and the

result output from tinyFaaS could be store in FReD nodes. Finally, we implemented dynamic changes. Now, the user can set up the infrastructure on FReD and can still be able to change some settings as modifying the expiry of a specific node, adding a new keygroup or adding a node in a keygroup.

4.2 Development Environment

The following software development stack was used for the implementation:

- Programming Languages: JavaScript, HTML, CSS, Python, Golang
- IDE: PyCharm, Visual Studio Code
- Technologies and Frameworks: React JS, Docker
- Productivity Tools: GitHub, Slack, Zoom

4.3 Frontend

As we briefly explained in Section 3, we decided to use a single page application (SPA for short) when developing the frontend. This offers two major advantages over multi-page applications:

1. SPA applications are fast because the required files (HTML, CSS and JavaScript) only need to be loaded once. Once that is done, only the dynamic content needs to be loaded.
2. SPA development is simplified and streamlined because no code needs to be written to render pages on the server. Single Page Applications are easy to implement in production.

4.3.1 React and Frameworks

We decided to use the JavaScript framework React for this frontend web application. Not only are HTML structures expressed via JSX, but the recent trends also tend to put CSS management inside JavaScript as well. All components express their User Interface (UI) within render

functions using JSX, a declarative XML-like syntax that works within JavaScript. This enables the developer to leverage the power of a full programming language to build your view. This includes temporary variables, flow controls, and directly referencing JavaScript values in scope.

4.3.2 User Interface Design

While building the interface, a large focus on intuitiveness was set as the target user audience should be able to view and interact with information provided on this webpage without any problems. We tried to keep the information density as low as possible so that the web app appears tidy and clean at all times. We ensured this by keeping both the interaction and configuration elements minimalistic. Also, it should quickly be clear in which way FogStore can be interacted with. It should become clear that the first step is to configure the infrastructure that can subsequently be worked with.

The UI consists of four components. Three of them represent the interfaces to configure the existing infrastructure. The fourth enables subsequent interaction with the infrastructure.

During the entire time, the user should be able to select and display the interface he prefers. In addition to the interface to configure the infrastructure, it should also be possible to display the component to interact with FogStore.

To create the graphical user interface, we used the React Flow library. This is an easy-to-integrate, light-weight and intuitive library that makes it possible to model flow graphs. With the help of React Flow, the nodes available to the user (Keygroup, Replica and Function) can be strongly personalized, e.g. adding connecting elements to the edges of the nodes or defining input fields.

4.4 Backend

The core of our project lies in the IaC Tool, our backend. It manages the deployment of a fog application infrastructure from a JSON file given by the user, thanks to the different calls it makes to FReD and TinyFaaS. The tool consists of three main parts :

1. a Validator, which verifies that the JSON file is well written,
2. an API, which has the role of a coordinator between FReD and TinyFaaS to enable the setup of the infrastructure and its modifications,
3. a Connector, which handles the communications between FReD and TinyFaaS, once the setup is done. It establishes the connections between the two entities of FogStore, FReD and tinyFaaS, to enable data processing.

4.4.1 API, Connector and Validator

When the Application Programming Interface receives the JSON file, it calls the Validator to check if the JSON schema constraints are fulfilled. Indeed, the Validator checks whether :

1. the type of each field is correct, for example, it checks if the mutable attribute of each keygroup is a boolean
2. there is coherence in the data fields, for example, it checks if the nodes in each keygroup are already declared in the field that lists all the available nodes, for the infrastructure
3. the uniqueness of names are respected

If any errors occur, the output message is displayed on the web application or on the terminal if the client chose the Mercury Command Line Interface. Each possible error has a specific explanatory message that can help the user identify the mistake.

We chose to code our backend - API, Connector and Validator - in python. We used a web framework for python to build our REST API, named FastAPI. Thanks to it, we could manage

multiple endpoints as mentionned in the section 3.5.1.

We also had to enables connections to multiple ports :

1. port 80 : to connect our API to the tinyFaaS management service (in order to register functions in tinyFaaS)
2. port 8080 : to connect our Connector to tinyFaaS reverse proxy (in order to pass data from FReD as function's parameters or to store function's result outputs into FReD nodes)
3. port 8081 : this is the port of our API
4. port 3000 : to connect our React web application to our API

4.4.2 Dynamic changes

The infrastructure setup was static at the beginning of this project. We decided it could be more convenient if the user was allowed to make changes without having to re-deploy it from scratch. The last feature we implemented was the Dynamic changes in our API. Now, even if the infrastructure is deployed on FReD nodes, the user can still change some of the initial settings, such as the expiry of nodes in keygroups, the name of the keygroups, or even add or delete nodes or functions in keygroups.

With Mercury, the user can set up the infrastructure, modify it afterwards if needed and can process data from FReD thanks to tinyFaaS serverless functions. The possible result of these functions can also be stored back in FReD nodes. This way, several keygroups can be linked.

5 Evaluation [Xin]

With the rapid development of IoT and cloud storage, more and more applications and components in the IoT ecosystem are using the cloud to store their data, but due to security concerns

and bandwidth limits, using fog nodes to pre-processing the data on the edge could significantly increase the performance and reliability of the application. However, configuring the nodes for application is manual and tedious, which raise the demand for automating the configuration process. Based on this, we have developed an IaC program, which shows the advantages of ease of use, reliable performance, visibility, and automatic deployment.

5.1 Usability

One of the main goals of the project is to make building and deploying applications on FogStore easy and reproducible. We have created a user-friendly front end web application to realize flow-based development. As mentioned in Concept Design, there are 2 frontend portals for the user to deploy their application replica node and trigger functions. The backend API will receive a formatted JSON file as input and extract variables from the file to conduct deployment. We provide an example on the portal for the user to reference, so the user can upload the JSON file through the upload interface. Backend will check for validity of the JSON schema, and return error message or success feedback correspondingly. Compare with an original way of writing codes directly on test/main.go file to create replica nodes, register key group, we automate it just by click-and-deploy methodology.

Tools	Interaction endpoint	Target framework	High-level programming	Code-snippet input not required	Generate Big Data program
Lemonade	Flow-based GUI tool	Spark ML (via Python APIs)	✓	✓	✓
Apache Zeppelin	Interactive shell	Multi-language back-end including Spark and Flink	×	×	×
Apache NiFi	Flow-based GUI tool	Interfaces with Spark and Flink	✓	×	×
Apache Beam	Flow-based programming API	Unified Programming model for Big Data systems including Spark and Flink	×	×	×
Microsoft Azure	Flow-based GUI tool	Includes Spark	✓	×	×
QryGraph	Flow-based GUI tool	Pig	✓	✓	✓
Nussknacker	Flow-based GUI tool	Flink	✓	✓	✓
QM-IConf	Flow-based GUI tool	Storm	✓	✓	✓
Our approach prototyped in aFlux	Flow-based GUI tool	Spark, Flink [17]. *Extensible*	✓	✓	✓

Figure 6: Comparison of high-level Spark programming with existing solutions

A graphical programming interface following the flow-based programming paradigm, which supports for customisation of configuration and auto-generation of native code is considered ideal in this scenario. There are much flow-based GUI tools available on the market, as shown in Table 1, all tools except Apache Beam and Apache Zeppelin provide a GUI to design an application. As the study shows, a set of unified APIs is more difficult to use for less-skilled programmers than a graphical tool [2]. For the tools without a GUI, for example, Apache Zeppelin, it has an interpreter which can take SQL queries or python code snippets, to run in multiple target environments [2], but for users who's not familiar with python and SQL commends, this could be inconvenient. In contrast, our application provides a graphical flow-based portal, which supports customisation of components used in a graphical flow, editable at the webpage for easier accessibility, automates code generation for the user and directly send to the backend to finish the configuration of nodes for the new application.

5.2 Scalability

As shown in our architecture, each of the components of the project final delivery is separated and driven by docker, which provides scalability. The Fred part has one container for each of the replica nodes; the connector and backend are situated in a different container as well, which ensure separation and reusability. When new functions are developed on the backend, only the iactest container should be re-built, leaving others untouched. Developers can also start new containers if the business requires more replica nodes. Another advantage docker brings is auto-scaling. In the further development process, it might be beneficial that Fred nodes can scale up more instances based on usage. Docker makes it possible and easy to achieve. The only concern regarding that would be performance drop, but according to study [3], the same level of performance and scalability compare with native execution is possible. Our project is a tool to make different systems interact with each other in an interactive and visually-appealing way. Advanced connectivity of systems allows eliminating repetitive programming tasks and

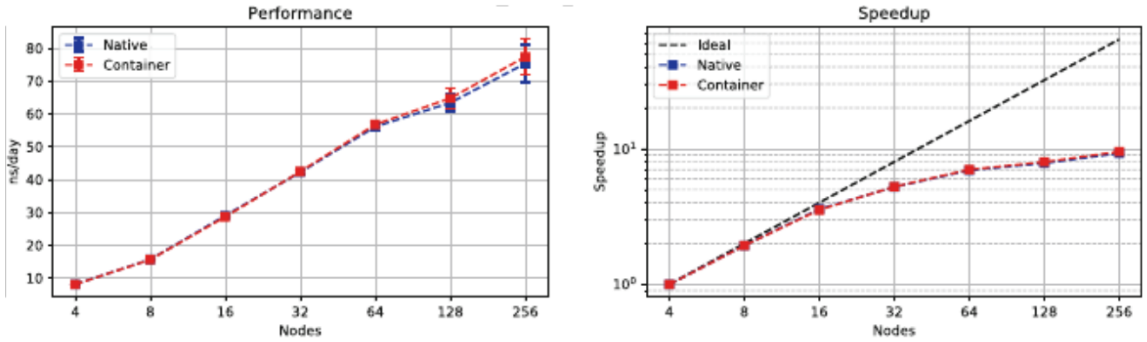

Figure 7: Figure 2. Comparison of performance and speedup between native and container versions of GROMACS on Piz Daint

free developers on more advanced topics. Developers can easily create new APIs to extend the functions and abilities of the application.

5.3 Automatic Deployment

To start the application, multiple dependencies, environments and plugins are required, which are listed in section 4. Since the application includes one backend, two frontends, and involving the start and connects of two other projects - Fred and tinyFaaS, to download the dependencies and build a network for internal communication have to follow specific procedures. To ease the deployment of the application, the application development environment is provided as a single-command installation package, and the auto development comments are written in Ansible. Ansible is a popular IT automation engine that automates cumbersome configuration and deployment tasks [1]. It will push out small Ansible Modules to the nodes to connect and use an SSH agent to execute the modules and then removes them when finished. Since there are no servers, daemons or databases required [1], these modules can reside anywhere in the machines. All the required resources and settings for running the application are declared within the package playbook.yml file. For further development, if there are new dependencies added or new configuration command required to start the application, the developers can easily change the

Ansible script, which shows the advantages of ease of maintenance and reusability. Ansible also enables the security of the system for the next step of development. By defining Playbook roles, further developers can define secure any part of the system, set firewall rules, lockdown users and groups, or other customize security policies.

6 Conclusion [Amit]

This final chapter summarizes the project result and reviews the progress of development. The technical and non-technical requirements listed in chapter 6.2 are compared to the final product and any discrepancies are explained. Finally, potential extensions and future work is surveyed in an outlook

6.1 Summary

In the span of one semester, this group of 4 students build a Mercury toolkit that deploys the Fog applications onto the Fog-Store platform. To attain this, FReD and tinyFaaS are set up in a given Fog environment by extending the existing tinyFaaS or FReD with new features. A way for application developers to build these application descriptions: using a JSON based text file, using a command-line interface, or using a graphical user interface is presented. A tool(Mercury) that takes the application descriptions, understands them, and deploys the key group configuration to FReD and functions to tinyFaaS is build by extending tinyFaaS and FReD with new features. To verify the features, a sample application i.e. temperature converter(section 3.3.3) (using mocked input data) is to build and deployed. Finally, a UI (section 4.3 and 3.4) with a visual editor which can be used to create the application descriptions described above is developed.

6.2 Requirement Fulfillment

Table 1 shows which functional requirements from section 2 are actually fulfilled. All the 7 criteria are fulfilled and referred to in the below table. To configure, the backend service is developed to deploy, manage and update.CLI application is developed along with comments for ease of use. Further, tinyFaas GRPC gateways are extended to enhance FReD trigger node. For the connection network, the network between FogStore, backend service, frontend service, TinyFaaS mgmt service and reverse proxy is successfully built.

Table 1: Functional Requirements

Criteria	Description	Fulfilled
Configuration	the main purpose of the project is to build 3 ways to realize easy configuration of FReD and TinyFaaS, which require building a backend service to deploy, manage and update all components of the application to the correct FReD and TinyFaaS nodes.	✓
IaC	ease of deployment and management with the configuration file, here we use json. The configuration should take the form of a json file, which is easy to edit, copy, and distribute, and conduct the configuration process.	✓

Continued on next page

Table 1 – continued from previous page

Criteria	Description	Fulfilled
CLI deployment	the project should provide CLI to realize the ease of use, the CLI application should have basic comments available to perform configuration. One main command should be FReD <pathOfConfigurationFile> which should require configuration JSON file path as a variable, and based on what defined in the JSON file to realize create key groups, configure replica nodes, deploy functions to tinyFaas, and configure trigger nodes.	✓
GUI	build a web interface to visualize and design applications using drag and drop components and custom data transformation code, which should enable the user to control the component placement and conforming to data movement, showing the infrastructure as data flow alike format.	✓

Continued on next page

Table 1 – continued from previous page

Criteria	Description	Fulfilled
Extend FReD trigger node	when replica nodes receive the new update data, the request should be sent to tinyFaaS reverse proxy, which acting as the trigger node. To realize which, tinyFaas GRPC gateways should be extended.	✓
Connection network	the network between FogStore, backend service, frontend service, TinyFaaS mgmt service and reverse proxy should be built, which ensure the internal communication between docker containers to realize the configuration and functionality. Fred and TinyFaas internal connections are already built, so we need to ensure FE and BE are connected, also the intermediate connections between them are well established.	✓
Provable	Build and deploy one application for demonstration.	✓

Table 2 shows which non-functional requirements from section 2.2 are actually fulfilled. All 4 criteria are full filled and referred to in the below table. The deployment makes the installation

automatically along with the logs. The deployment is made successfully in all the 3 ways as mentioned. The tracing facility is enabled by logging the deployments. Further, the front end enables the modifications. To provide ease to the developer the code is self explainable, substantial comments are provided and the relationships between FReD, tinyfaas and iacTest is clearly and well documented.

Table 2: Non Functional Requirements

Criteria	**Description**	**Fulfilled**
Deployments	the implemented IaC should be easy to deploy to a new environment, which should automatic the installation of required packages, software, environments as much as possible	✓
Utility	the configuration process should be easy and scalable, the new application should be easy to deploy with either of 3 ways	✓
Traceability	clear logs should be provided for debugging and issue tracing, also the correct status of the new deployment should show logging capabilities for checking the status of the new deployment.	✓

Continued on next page

Table 2 – continued from previous page

Criteria	Description	Fulfilled
Responsiveness	The front-end should be react-app, which will ensure ease of modification in further process, also the web application should have fast responding when there are changes from the user, to optimize the performance from the front end.	✓
Self-explain	the code should have necessary comments for developers to understand, and the configuration file content should be easy to understand and modify. Relationships between FReD, tinyfaas and iacTest should be documented, so it can easily be extended by other teams working on the project	✓

6.3 Outlook

With the present implementation in Mercury, this included building application descriptions through a JSON file, CLI, and interactive front end(GUI). And any modified data in the key group (either while creating or modifying it)can automatically be sent to tinyFaaS to perform a function.

The mercury tool is to build in such a way that it takes the application descriptions, understands

them, and deploys the key group configuration to FReD and functions to tinyFaaS
A frontend is to be developed to create these applications descriptions.
However, there are opportunities to refine the application such as deployment of mercury in production environment while hosting it over any cloud provider like AWS or Azure etc.
All in all, the Mercury deployment was successful.

7 Annexes [Simine & Alexander]

7.1 Installation and Execution

The Mercury project can be installed and executed on a Linux environment with an Ubuntu distribution 18.04 or more recent. To do so, you will need the two different zip files which contain respectively the 'fred' and the 'tinyFaaS' directory, and the two scripts 'fogStoreScript.txt' and 'mercuryScript.txt'. When extracting the two directories, please be aware that it might create a parent directory to the actual required directory. Please check if, when opening the 'fred' directory, you can see a 'test' directory. If it is not the case, get rid of one layer a.k.a. the parent directory. The same applies to 'tinyFaaS' zipped directory.

The extracted directories and the two scripts files have to be in the same directory (usually 'Downloads' but you are free to choose). Please make sure the port 8081 is enable and free for inbound and outbound communications.

You will need to open two different terminal and for each of them 'cd' into the directory where you can find the four previous elements. You will need to know the password of the superuser of your machine in order to be able to use the 'sudo' command. In one of the terminal, please execute the following commands :

1. sudo chmod +x fogStoreScript.txt

2. sudo ./fogStoreScript.txt

In the other terminal, please execute :

1. sudo chmod +x mercuryScript.txt
2. sudo ./mercuryScript.txt

Our project needs a nodeJS version upper than 10.x to run. In the mercuryScript, the version 12.x is being installed. If your machine already has a version of nodeJS installed which does not meet the requirement, please delete it so our script can install the desired one and use it for our project.

Please wait until the fogStoreScript runs correctly (this terminal needs to stay up and running everytime for the project) and the mercuryScript displays the web application page on your browser (on localhost:3000).

Now that you have installed and executed the project on your computer, it is time to try it !
We will share some steps you can do, to value the Mercury tool, in the next part Demo.

7.2 Demo

To check the functionality, the data.json located in the FReD/tests/IacTest can be used. This configures two keygroups, one of which stores temperature values in Celsius units and the other in Fahrenheit units. The two keygroups are connected via a tinyFaaS function, so that with only one API call a newly inserted value is stored both as Celsius value and as Fahrenheit value. For this, the name of the handler has also to be specified during the insertion.

To simulate this exemplary data flow, four steps are necessary, which can be executed both on the website and via the CLI:

1. Upload the data.json
2. mercury process Celsius temp1 15 celsFahrConverter

3. mercury get Celsius temp1

   Expected return value: 15

4. mercury get Fahrenheit temp1

   Expected return value: 59

If you want to test with a decimal, please put a dot and not a coma e.g. 15.8 and NOT 15,8. The respective commands we used above can also be found and explained in the 'Interaction' component of our web application.

7.3 Scripts

This is the fogStoreScript.txt :

```
#!/bin/bash


sudo apt-get update


#install docker
sudo apt-get install \
    apt-transport-https -y\
    ca-certificates -y\
    curl -y\
    gnupg -y\
    lsb-release -y
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
```

```
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io -y
sudo apt-get update
sudo curl -L "https://github.com/docker/compose/releases/download/1.28.6/docker-compose-$
  (uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

#install pip
sudo apt install python3-pip -y

#start the tinyFaaS Containers
cd tinyFaaS
sudo make start
cd ..

#start the fred Containers
cd fred/tests/3NodeTest
sudo make beforeFirstIacTest
sudo make iactest
```

This is the mercuryScript.txt :

```
#!/bin/bash

#install nodejs
```

```
sudo apt -y install curl dirmngr apt-transport-https lsb-release ca-certificates
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
sudo apt-get install nodejs -y
sudo apt-get install npm -y

#install git
sudo apt install git -y

#install the CLI
git clone https://github.com/alexfritsch10/dsp-cli.git
cd dsp-cli
pip3 install .
cd ..

#start the web App
git clone https://github.com/alexfritsch10/dsp-project-frontend.git
cd dsp-project-frontend
sudo npm install
sudo npm start
```