A Pattern-based Function and Workflow Visual Environment for FaaS Development across the Continuum

George Kousiouris
gkousiou@hua.gr
Harokopio University
Athens, Greece

Szymon Ambroziak
ambro.szymon@gmail.com
GFT
Lodz, Poland

Blazej Zarzycki
Blazej.Zarzycki@gft.com
GFT
Lodz, Poland

Domenico Costantino
costantino@hpe.com
Pointnext Services, HP Italiana Srl
Milan, Italy

Stylianos Tsarsitalidis
stsarsitalidis@hua.gr
Harokopio University
Athens, Greece

Vasileios Katevas
vkatevas@hua.gr
Harokopio University
Athens, Greece

Alessandro Mamelli
alessandro.mamelli@hpe.com
Pointnext Services, HP Italiana Srl
Milan, Italy

Teta Stamati
teta@hua.gr
Harokopio University
Athens, Greece

ABSTRACT

The ability to split applications across different locations in the continuum (edge/cloud) creates needs for application break down into smaller and more distributed chunks. In this realm the Function as a Service approach appears as a significant enabler in this process. The paper presents a visual function and workflow development environment for complex FaaS (Apache OpenwhisK) applications. The environment offers a library of pattern based and reusable nodes and flows while mitigating function orchestration limitations in the domain. Generation of the deployable artefacts, i.e. the functions, is performed through embedded DevOps pipelines. A range of annotations are available for dictating diverse options including QoS needs, function or data locality requirements, function affinity considerations etc. These are propagated to the deployment and operation stacks for supporting the cloud/edge interplay. The mechanism is evaluated functionally through creating, registering and executing functions and orchestrating workflows, adapting typical parallelization patterns and an edge data collection process.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software product lines**.

KEYWORDS

Function as a Service, Software Development, Serverless computing, Function Orchestration

ACM Reference Format:
George Kousiouris, Szymon Ambroziak, Blazej Zarzycki, Domenico Costantino, Stylianos Tsarsitalidis, Vasileios Katevas, Alessandro Mamelli, and Teta Stamati. 2023. A Pattern-based Function and Workflow Visual Environment for FaaS Development across the Continuum. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584934

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04.
https://doi.org/10.1145/3578245.3584934

1 INTRODUCTION

Function as a Service[16](FaaS) enables much more efficient management of service components, executed on demand as functions. It typically offers functionalities such as service gateway management, queue based load levelling architecture for function execution, function packaging and management, automated fine-grained scaling and link with back-end container orchestration systems for execution of the functions. Thus it can significantly alleviate the management of services and applications, adhering more closely to cloud native application design considerations[3] however it comes also with a set of challenges, including maintainable composition models for serverless workflows, function reuse, easy embedding of legacy code and CI/CD processes[4].

To abstract the use of the continuum, application development should combine functions and services in one environment, pattern prototype implementations offered as reusable components, use annotations to indicate dictations and managerial approaches to the underlying management layers and benefit from abstractions and visual development tools for building non-trivial FaaS applications[14].

The purpose of this paper is to present a novel cloud Design Environment for FaaS that is based on these principles.The environment encapsulates the widely used (in the IoT domain) Node-RED[1] web based function framework for event driven applications. It extends its usage for acting as a generic function and workflow creator for FaaS.A preprint description of our work is included in [12]. In this paper we present more details as well as the evolution of the framework and the new features included.

The environment provides the following contributions:

[1]https://nodered.org/

- Visual environment for abstracted function and workflow creation, enriched with a palette of implemented functionalities in the form of patterns (parametric subflows).The latter aid the developer in the adaptation to the FaaS paradigm, used directly in a drag and drop manner in the workflow, and aiming to address aspects such as parallelization of input load, context management, edge data collection, error capturing, performance monitoring and helper functionalities such as asynchronous API calls management
- Inclusion of diverse annotations as guidelines for aspects such as function placement across the cloud/edge continuum, preferences in function locality, scheduling, optimization goals for placement etc.
- Packaging of the created function or workflow into a deployable artefact (i.e. docker image) through pluggable DevOps pipelines including function registration and performance profiling on the target FaaS platform (Apache Openwhisk).

The overall effect is to reduce the learning curve and development time needed, as well as bypass current FaaS limitations in terms of workflow orchestration and definition. The paper proceeds as follows. Section 2 presents related work while Section 3 the main architecture and building blocks of the system. Section 4 presents a set of example case studies while Section 5 concludes the paper.

2 RELATED WORK

A major drawback of current Function as a Service platforms is the availability of tools related to deployment and function reuse[15] as well as function orchestration abilities[7]. The effect of the disadvantages of the available development tools for FaaS is evident on the current FaaS landscape. According to [9], 82% of serverless applications use 5 or fewer functions and only 31% of them include workflows.

From a workflow perpsective, the only open source platform that has a native workflow functionality[6] is Openwhisk[2], although it only supports simple sequences of function chains. Code or Text-based plugin orchestrators with more orchestration primitives include the IBM Composer[3] (javascript library form[7]), FaaSflow[4]) for OpenFaaS and the Google Cloud Functions[5] YAML based syntax. The latter appears in Fig. 1, as a comparison between this form (left) and an equivalent Node-RED flow (right) implementing the same functionality. It is evident that when scaling to larger workflows, this type of definition does not serve well simplicity and design.

The most advanced is the AWS Step functions which also includes a visual programming style and extended operators (including state management). On the other hand, it is an option that increases vendor lock-in. Kubeflow[8] is used primarily for AI pipelines and includes a relevant definition language as well as an editor extension (Elyra) for visual creation. However due to its pipeline nature, it refers more to a static sequence of operations while inputs and outputs between nodes of the workflow are passed through the defined object storage files. Thus it does not portray the abilities of an actual runtime environment for merging messages and calls between functions.

Apache Airflow is a workflow design and management tool that can be used to create Directed Acyclic Graphs (DAGs) of operations across components. Airflow has complex operators, however it can not be used as a generic function editor, the components need to pre-exist. Similarly, TriggerFlow[5] offers diverse workflow primitives and eventing mechanisms for orchestration. The main difference of our work, except for the dual function creation and workflow orchestration, is the ability to import ready-made functionality in the form of reusable subflows and Node-RED nodes as well as the visual support for the workflow creation. The same applies for customly created user flows that can be easily shared and reused. Furthermore, the ability to decide at development time how to group functions,and potentially group many of them in the same container runtime, creates significant advantages in terms of minimizing orchestration overheads[13].

3 CLOUD DESIGN ENVIRONMENT ARCHITECTURE

3.1 Design Environment Overview and Architecture

The overview of the proposed Design and Development Environment appears in Fig. 2 and is a part of the PHYSICS platform[6] based on Apache Openwhisk FaaS. The UI environment is an embedded container of a Node-RED server coupled with a Control UI. The Node-RED container as well as the SFG one run locally on the developer side, and contact the remaining cloud-based components.

Node-RED extensions[2] provided by this work include:

- Ready-made pattern subflows (see section 3.2 for details), dealing with parallelization, context management, data collection, workflow primitives etc. One significant feature used is the ability to group many functions into subflows, that then appear as regular nodes in the palette. This helps creating groups of functions that serve a specific purpose, reusing function logic as well as abstracting from the specific implementation details.
- Helper subflows that aim to address a specific issue (e.g. contacting asynchronous APIs and converting a polling process for the result to a pushing one)
- Semantic annotation nodes (see section 3.3 for details) for giving directives to the deployment layers.

The first two cases can be dragged and dropped directly in an arbitrary workflow, wired together and used to create a functional flow. The third one adds metadata in the created Node-RED flow specification. Any other node or flow from the Node-RED repository[7] can be imported if needed.

Once a flow has been created, it can be packaged and tested within the Control UI of the environment. From then on, the process contacts the Serverless Function Generator for extracting the code/flows and settings from the Node-RED environment. This is fed as an input to a Jenkins pipeline. The latter starts from a

[2]https://openwhisk.apache.org/
[3]https://github.com/apache/openwhisk-composer
[4]https://github.com/s8sg/faas-flow
[5]https://cloud.google.com/functions/docs/tutorials/workflows
[6]https://physics-faas.eu/
[7]http://flows.nodered.org

```
- randomgen_function:
    call: http.get
    args:
        url: https://REGION/-PROJECT_ID/.cloudfunctions.net/randomgen
    result: randomgen_result
- multiply_function:
    call: http.post
    args:
        url: https://REGION/-PROJECT_ID/.cloudfunctions.net/multiply
        body:
            input: ${randomgen_result.body.random}
    result: multiply_result
- return_result:
    return: ${multiply_result}
```

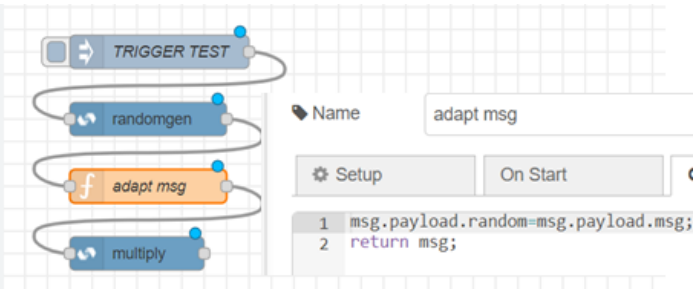

Figure 1: Example workflow definition code snippet in Google Cloud Functions compared to equivalent Node-RED definition

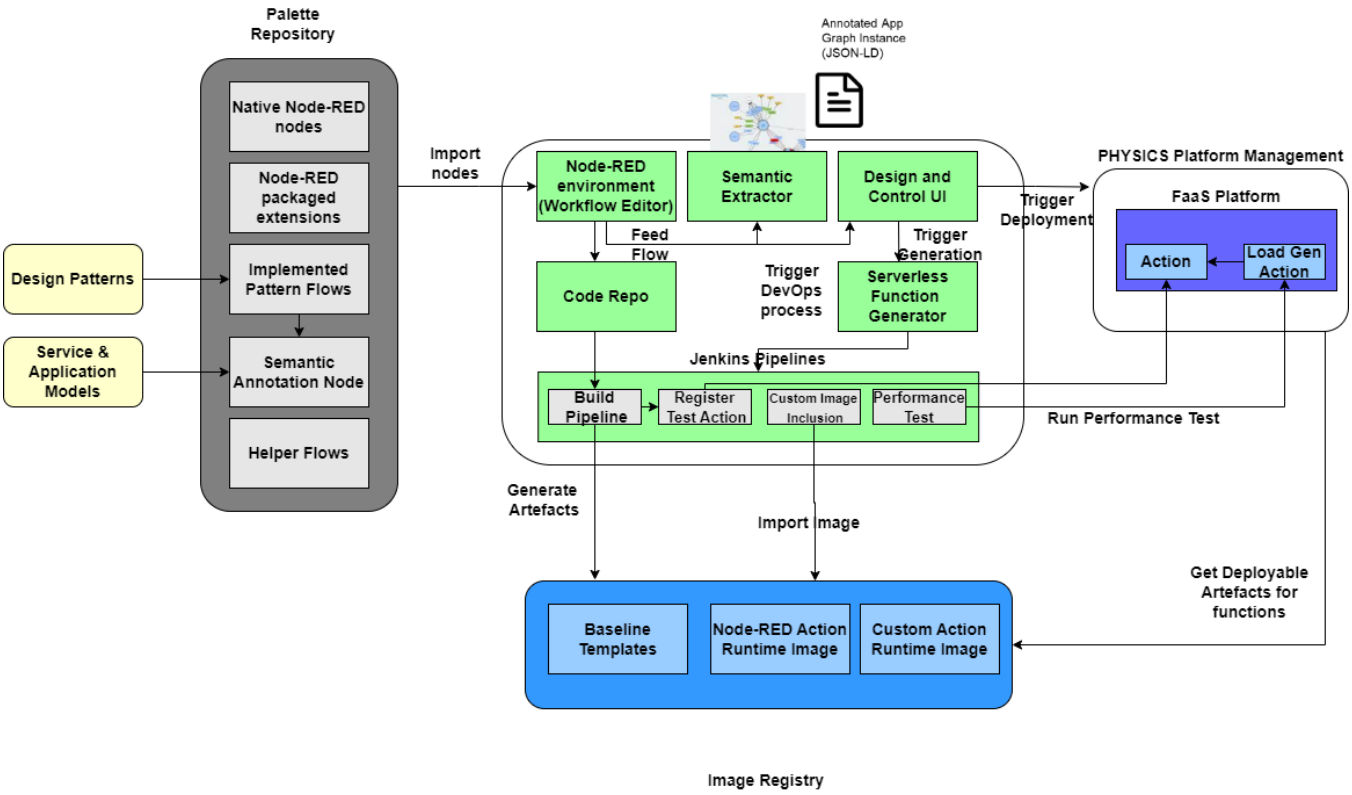

Figure 2: Architecture of the PHYSICS Design and Development Environment

(customizable) baseline template docker image, it injects the case-specific flows and settings and pushes the image to a docker registry. It also registers the image to an available Openwhisk installation as a docker function. Now the function has been created and it is ready to be invoked for final testing.

Once the final testing is finished, the developer can deploy the selected set of flows into the production cluster. In this case, the flows pass through the Semantic Extractor component, which extracts declared semantic annotations, maps them to triples based on ontological concepts, converts them to JSON-LD and creates the application graph including the application structure, location of the images for each function etc. This is then forwarded to the PHYSICS platform management layer for the deployment. This layer is responsible for respecting user requirements and converts the annotations to Kubernetes keywords or Openwhisk parameters (e.g. for hardware deployment needs, function affinity, sizing etc.).

3.2 Pattern-based Flow creation

A pattern is defined as *"a proven series of activities which are supposed to overcome a recurring problem in a certain context, particular objective, and specific initial condition"*[18]. Patterns have been a very useful tool for dictating design principles as well as driving

abstract implementations for a specific domain [10]. For the cloud domain, a detailed catalogue can be found in [3] while for FaaS a relevant list can be found in [17]. In general a pattern may be driven from specific problems of a domain (e.g. networking and reliability faults in distributed systems, state handling in FaaS etc.) or even more basic primitives migration to a new model (e.g. moving MPI-style concepts to the FaaS execution model).

From an implementaion point of view, the pattern scope aligns well with the ability of Node-RED to create subflow nodes that group entire internal workflows with properties and configurations on top of them. This enables their usage in a general and repeating context, while hiding underlying complexity. This is especially important during migration to the FaaS model. The latter is based heavily on event driven, function oriented programming which produces difficulties for developers accustomed to more streamlined paradigms like object oriented. Furthermore, in many cases the developers/end users of such a mechanism are data scientists or scientific engineers more used to scripting languages and not experienced software developers. Finally, because functions in FaaS are more fine-grained, a FaaS application can result in potentially large numbers of them in a project repository. Thus grouping them can ease their management.

3.2.1 Parallelization Pattern Example. As part of the environment, a set of such patterns has been created[2] and is included in the extended Node-RED palette embedded in the editor. The Single Process Multiple Data (SPMD) or Split-Join is presented in Fig. 3. These types of parallelization were commonly based on typical parallel technologies like MPI.

In a function programming style, the parallelization can be performed by splitting an initial array message in individual messages, upon which the same computation will be applied. Each split message then triggers a respective execution. A Join node waits for all the respective partial messages to complete. In our case, different options are available for the execution such as external function execution (intercontainer parallelization similar to MPI for exploiting multiple nodes) or multiprocess/multithread execution inside the same container (intracontainer parallelization similar to OpenMP for parallelization in the same node). In both cases the provisioning of containers for the execution is performed by the FaaS platform itself.

The split size (how many rows in the initial array will be included in each execution) is a parameter to be configured. A too fine grained value can spawn too many containers and result in extreme inter-process overheads on the computational resources.

It is not necessary for a pattern flow to be only deployed in a function oriented style. For example, the aforementioned SplitJoin flow can reside on a Node-RED server at the edge, responsible for collecting the data and then triggering the parallelized computational step. The latter can target a large scale Openwhisk installation on a more central cloud location that can spawn multiple function containers for an effective computation.

Another example of a service pattern is the batch request aggregator in [11]. From a developer point of view, the mean of development is the same inside the environment, the only aspect that changes is the use of a relevant annotation (see next section on annotations) to indicate that the specific artefact should be deployed as a service and not a function.

3.2.2 Unreliable IoT Edge Data Collection Pattern. For the case of data collection, especially in resource constrained environments such as Smart Agriculture greenhouses, the edge device (typically a Raspberry Pi) may be too constrained to run a FaaS platform, even a light version of it such as OWL[8]. For this reason, the inclusion of flows as simple services in a Node-RED server may be the best way to utilize the resource.

However, in these cases, other problems may exist (e.g. volatility of networking conditions), which results in frequent network failures and missing values. Completeness of these data are key in order to run agronomic simulations and calculate the necessary management of the plants. To this end, a reliable Edge Extract-Transform-Load (ETL) pattern has been developed (Fig. 4). It consists of approximately 23 inner javascript functions that are hidden in the subflow. The pattern receives the data item through an incoming message. The input layer can use whatever available Node-RED node to interact with the main IoT system. Node-RED originates from the IoT world so it has extensive node support for many sensor protocols. The flow tries initially to send the data item to the central service for a limited and configurable number of consecutive times based on the retry pattern. If these attempts fail, the data item is stored locally. Periodically, a cron job (e.g. every hour) is set to try to resend all failed items up to this point.

3.3 Semantic annotations inclusion

Semantic annotations can dictate specific developer needs and expose the capabilities of the baseline management stacks. With relation to the cloud/edge continuum, they can be used to dictate the needed deployment location of a given artefact like the aforementioned SplitJoin flow. Therefore during development, the computational function that is intended to run on the central cloud can be annotated as such, whereas the orchestrating flow on the edge as a service. Other considerations may include different types of scheduling, affinity rules between functions, memory sizing allocations for the needed containers etc. In the case of the PHYSICS platform, two different scheduling capabilities exist, one for the typical Kubernetes scheduler and one taking into consideration container layer locality. Furthermore, a placement optimization takes place for the functions, therefore affinity rules can aid in putting constraints in the respective placement process.

A special set of nodes (semantic annotators) is available in the palette. The instantiation of such a subflow is processed by the Semantic Extractor component mentioned in Section 3 and the key value pairs of the node properties get passed to the application description. An example of such nodes appears in Fig. 5, to indicate scheduling, execution type (service or function), function sizing or locality requirements. While some of the semantic nodes can be simple key-value pairs, others can also embed logic as in the case of the Locality annotator. This retrieves the updated list of the available clusters in the continuum from a relevant platform

[8] https://github.com/kpavel/openwhisk-light

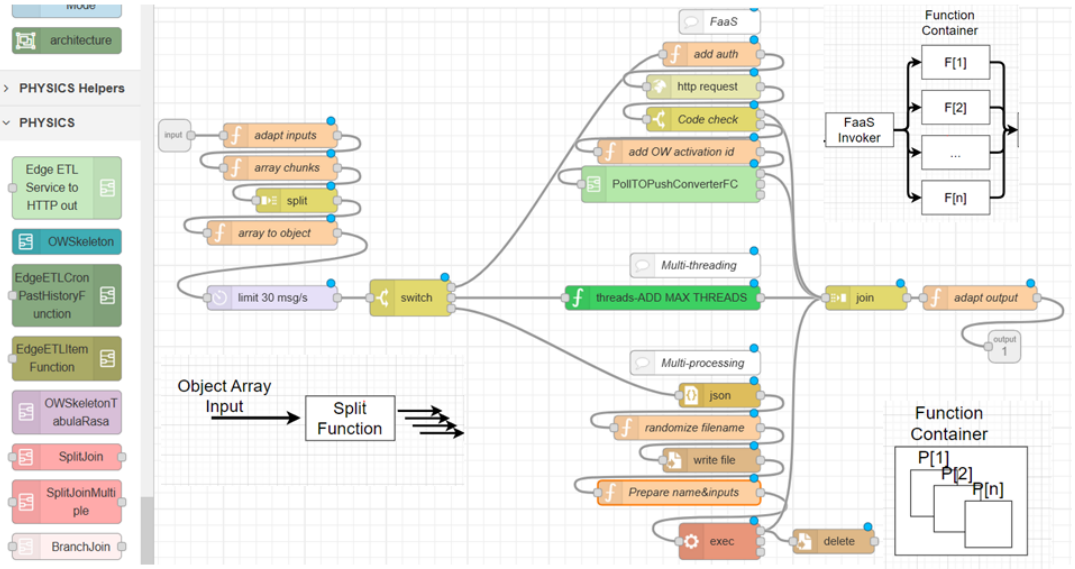

Figure 3: Example Split(Fork)-Join Pattern Implementation Hidden Inside the Subflow

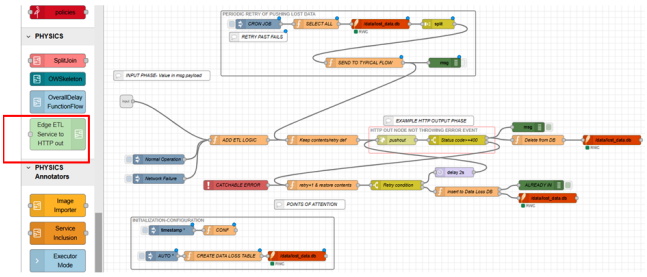

Figure 4: Edge ETL Service Subflow Implementation

endpoint and then updates the respective node definition drop down options.

3.4 Helper Functional and Non Functional Flows

Other functionalities can also prove helpful in reducing the complexity of a FaaS environment. For example, typically FaaS (as well as many other APIs) relies on asynchronous API calls for triggering a function execution. The client must trigger initially the execution and can not block waiting for the response. An activation ID is returned instead, based on which the client can poll afterwards for the result. Although it seems a straightforward operation, details

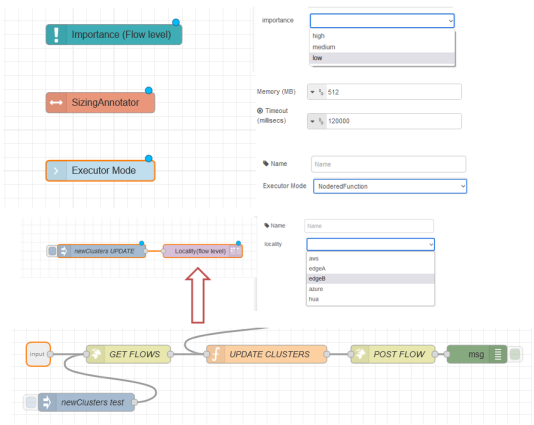

Figure 5: Examples of Semantic Annotators with static or dynamic behaviour

of the calls like repetitions, checking error codes etc. can insert significant complexity. For this reason, an abstracted mechanism has been created (Poll2PushConverter). The mechanism also supports function chains[17], a bypass pattern for the typical FaaS timeout of a function.

From a performance point of view, it is also important to monitor the execution of functions on the platform, as well as to easily retrieve error logs or statistics. For this reason, a relevant monitoring node has been created (Fig. 11). The node is based on a configurable sliding window and can be configured to filter needed function monitoring as well as fetch logs from erroneous function executions.

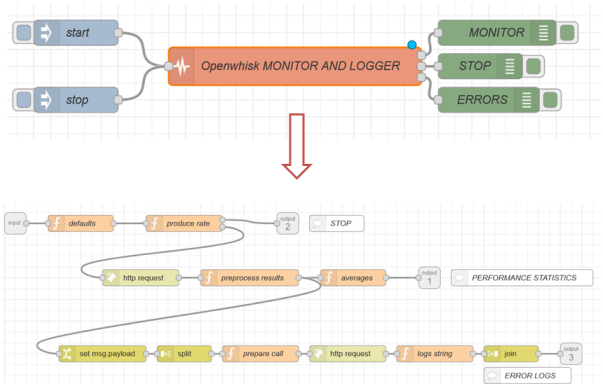

Figure 6: Monitoring Node Overview and Implementation

4 EXAMPLE CASE STUDIES

For experimenting with the environment, a number of case studies have been implemented. A demo video for the usage of the environment can be found in [1], including the creation and annotation of a flow, building of the relevant function, testing its deployment and execution as well as applying the parallelization pattern.

4.1 Simple Hello World Function

For the main use case, the developer starts from a skeleton subflow (Fig. 7) that implements the Openwhisk custom docker function specification (a POST /init for initialization and a POST /run for the actual execution). Any Docker container that includes such an endpoint can be registered as an Openwhisk function. Inside this flow, any node-RED packaged node as well as npm-based library can be imported and exploited. The developer can create any wiring, since their execution is performed within the Node-RED runtime of the Openwhisk action, not limited by any workflow language specification limitation of the FaaS platform. They can also invoke other external actions as part of the logic. A key point is to capture errors that occur in the flow. If that does not happen, then the flow inside the action will stall and only be detected as a function timeout error. This leads to increased costs for the execution, larger overheads for the back-end as well as misleading error reporting.

Once the flow is finished, building the function image can begin. Build times do not depend on the flow size or complexity. They may depend on the number of extra npm/Node-RED nodes that need to be installed to the baseline template but also on the baseline template itself and the size its external dependencies. The statistics of 100 builds based on the default Node-RED template appear in 8. The average is about 3 minutes needed. The peaks and red lines indicate builds that have been aborted due to temporary network unavailabilities of the testbed. For more complicated templates, e.g. with Tensorflow dependencies, the according build time is around 10 minutes.

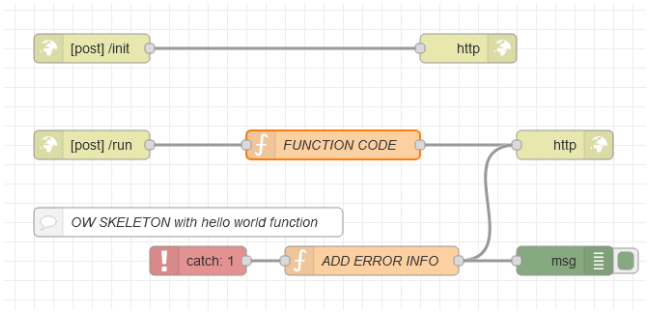

Figure 7: OW Skeleton Interface Flow

One advantage of this inclusion of many functions inside the same flow runtime is that if for example this flow is sequential then there are significantly less container overheads[13], since we do not need a separate container for each individual and potentially small function. On the other hand, the Node-RED runtime is more heavyweight (487MB) than a typical nodeJS one (356MB). Abstraction always comes with a penalty in performance so the question is to quantify this. To this end a hello world implementation was compared in both runtimes.

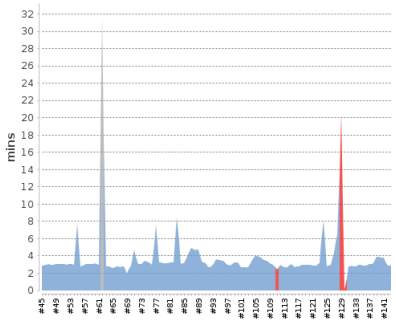

Figure 8: Function Image Build Times

For each case, 100 executions were conducted on each target function for cold and warm starts. Cold starts need to spawn a new container whereas warm ones reuse an existing one. The cold start time ((Fig. 9)) for the Node-RED runtime is considerably higher (with an average of 7.6 seconds compared to 2.5 seconds in the case of nodejs). This is due to the larger container, as well as the start-up time of the Node-RED environment). This can be dealt with through the usage of prewarm containers in Openwhisk, i.e. containers that are spawned proactively. In the hot execution case, the two execution modes are very similar (227 compared to 252 milliseconds).

4.2 Abstract Use and Test of the Fork-Join Pattern

The Fork-Join pattern of Fig. 3 is hidden behind a subflow node in the palette and used in any Node-RED flow. It can be executed both as as a service as well as a function mode. For this it needs to be wrapped around the Openwhisk Skeleton interface. The needed arguments include the name of the worker Openwhisk action to invoke (the function responsible for processing each input chunk) or the process script inside the container, and the initial array of input data to be split. The resulting flow appears in Fig. 10.

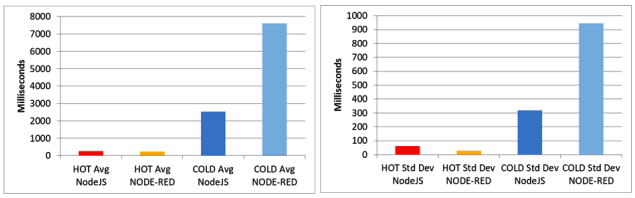

Figure 9: Hello World Function Comparison

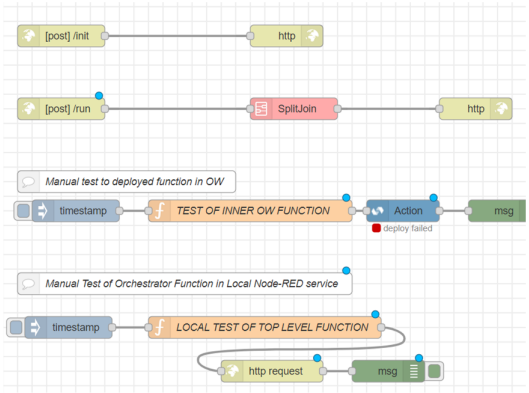

Figure 10: Orchestrator Function with the Split Join Pattern

One benefit of the environment Node-RED server is that one can create testing flows to be executed within that development server. Thus there is no need for building and debugging deployments on the FaaS platform. During the tests for the creation of this flow, approximately 10 errors were encountered, with 70% of them fixed locally. The remaining ones refered to the input and output format from Openwhisk to the function. These needed a FaaS deployment to detect errors, at least for an inexperienced user. The total debugging savings for the specific flow appear in Table 1.

4.3 Dashboard Monitoring

Through the exploitation of the monitoring subflow, the user can also embed into their local Node-RED environment a dashboard monitoring the main parts of the remote Openwhisk statistics. The node has been configured in this case to get the moving average of the last 5 minutes for all executed functions with a polling period

Table 1: Testing Time Saved Through Environment

Case	Measured Quantity
Number of errors	10
Errors solved with Node-RED test	7
Errors needing FaaS deployment	3
Local Node-RED test	1-2 seconds
Image, FaaS deployment and test	3-4 minutes
Time saved	21-28 minutes

of 10 seconds. An indicative view of the user monitoring appears in Fig. 11

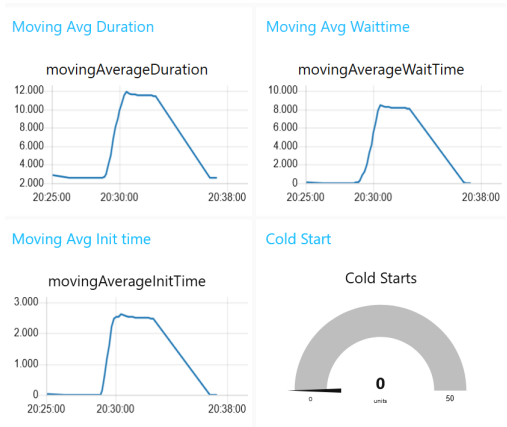

Figure 11: Openwhisk Monitoring from the Environment

4.4 Edge Data Collection Experimentation

The usage of the pattern is as abstract as seen in Fig. 12. Only the data input layer needs to be created, for obtaining the data items and pushing them to the subflow. The specific flow uses also the Executor Mode semantic node for indicating the deployment as a service.

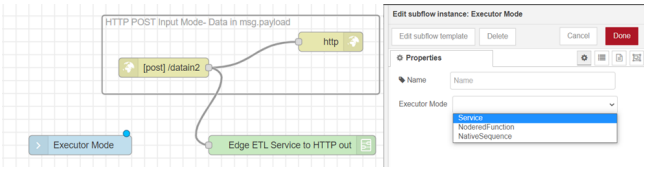

Figure 12: Usage of Edge ETL Subflow in Data Collection

From a functional point of view, it needs to send all data, without duplicates or missing values, as well as handle large periods of outages (i.e. large number of local points for resending). In order to simulate the outage, the target URL for the cloud storage was set to change every 5 minutes to an existing endpoint and every 3 minutes to a non-existent one.

An indicative value for a new data item in real life is 1 new data item per 10 minutes. In order to speed up the experiment, client

data generation was set to 2 calls per second. The periodic cron job, for past failed data resubmission was 2 minutes. Experiment duration was set to 2 hours. Each call was documented through a unique ID both at the edge and cloud side. Both logs were joined at the end in order to detect missing or duplicate values. None was detected from an overall 18500 samples. Indicatively the specific application before the pattern resulted in 50% lost points.

The edge database size across the experiment appears in Fig. 13, fluctuating due to the difference in the simulated outage intervals. The maximum number of unsent points reaches about 1400 values. During the cron job triggering, the pattern tries to send these. Thus they will create a burst, which however was manageable and results in an empty local database at the end of each cycle.

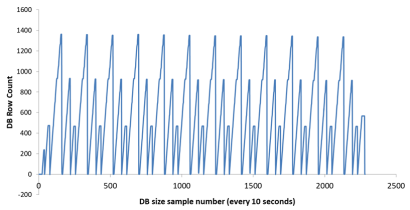

Figure 13: Evolution of Failed Data Edge DB size

5 CONCLUSIONS

Concluding, the presented Design and Development Environment offers features improving the currently limited functionality of FaaS workflows. It enables better manageability and faster development of functions due to the pattern based approach. Minimal knowledge is needed on the inner workings of each pattern, thus reducing the learning curve.

The environment builds upon Node-RED capabilities and transforms it to a generic function and workflow creator. Function orchestration from the Node-RED runtime enables the usage of more complex workflow primitives and elevates from the text based workflow means of other platforms. The embedded DevOps processes generate the deployable artefacts, including all necessary dependencies and giving the ability to start from customized dockerfiles. Versatile annotations are introduced that can forward developer options to the underlying management stacks.

For the future, the aim is to extend the collection of patterns and annotations, while linking them to runtime modelling mechanisms (as in the case of [11]. For example, a key parameter to determine is the split size of the Fork-Join parallelization, in order not to overstress the back-end.

ACKNOWLEDGMENT

The research presented has received funding from the European Union's Project H2020 PHYSICS (GA 101017047).

REFERENCES

[1] 2022. PHYSICS Design Environment Demo. https://www.youtube.com/watch?v=DO2sFdfCD-o.
[2] 2022. PHYSICS pattern flows for Cloud/Edge and Openwhisk. https://flows.nodered.org/collection/HXSkA2JJLcGA.
[3] 2023. Microsoft Cloud Design Patterns Catalogue and documentation. https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns.
[4] Cristina Abad, Ian T Foster, Nikolas Herbst, and Alexandru Iosup. 2021. Serverless Computing (Dagstuhl Seminar 21201). In *Dagstuhl Reports*, Vol. 11. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[5] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2021. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems* (2021).
[6] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. 2016. Cloud-native, event-based programming for mobile applications. In *Proc. of the International Conference on Mobile Software Engineering and Systems*. 287–288.
[7] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. Faas orchestration of parallel workloads. In *Proc. of the 5th International Workshop on Serverless Computing*. 25–30.
[8] Ekaba Bisong. 2019. Kubeflow and kubeflow pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 671–685.
[9] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. 2021. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering* (2021).
[10] Pooyan Jamshidi, Claus Pahl, and Nabor C Mendonça. 2017. Pattern-based multi-cloud architecture migration. *Software: Practice and Experience* 47, 9 (2017), 1159–1184.
[11] G. Kousiouris. 2021. A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments. In *40th IEEE International Performance Computing and Communications Conference (IPCCC 2021)*. IEEE.
[12] George Kousiouris, Szymon Ambroziak, Domenico Costantino, Stylianos Tsarsitalidis, Evangelos Boutas, Alessandro Mamelli, and Teta Stamati. 2022. Combining Node-RED and Openwhisk for Pattern-based Development and Execution of Complex FaaS Workflows. https://doi.org/10.48550/ARXIV.2202.09683
[13] George Kousiouris, Chris Giannakos, Konstantinos Tserpes, and Teta Stamati. 2022. Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*.
[14] George Kousiouris and Dimosthenis Kyriazis. 2021. Functionalities, Challenges and Enablers for a Generalized FaaS based Architecture as the Realizer of Cloud/Edge Continuum Interplay.. In *CLOSER*. 199–206.
[15] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software* 149 (2019), 340–359.
[16] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. 2017. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 162–169.
[17] Davide Taibi, Nabil El Ioini, Claus Pahl, and Jan Raphael Schmid Niederkofler. 2020. Patterns for serverless functions (function-as-a-service): A multivocal literature review. (2020).
[18] Agung Wahyudi, George Kuk, and Marijn Janssen. 2018. A process pattern model for tackling and improving big data quality. *Information Systems Frontiers* 20, 3 (2018), 457–469.