Self-balancing Architectures based on Liquid Functions across Computing Continuums

Josef Spillner
josef.spillner@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

ABSTRACT

Scalable application development is highly influenced by two major trends – serverless computing and continuum computing. These trends have had little intersection, as most application architectures, even when following a microservices or function-based approach, are built around rather monolithic Function-as-a-Service engines that do not span continuums. Functions are thus separated code-wise but not infrastructure-wise, as they continue to run on the same single platform they have been deployed to. Moreover, developing and deploying distributed applications remains non-trivial and is a hurdle for enhancing the capabilities of mobile and sensing domains. To overcome this limitation, the concept of self-balancing architectures is introduced in which liquid functions traverse cloud and edge/fog platforms in a continuum as needed, represented by the abstract notion of pressure relief valves based on resource capacities, function execution durations and optimisation preferences. With CoRFu, a reference implementation of a continuum-wide distributed Function-as-a-Service engine is introduced and combined with a dynamic function offloading framework. The implementation is validated with a sensor data inference and regression application.

CCS CONCEPTS

• **Networks** → **Network experimentation**; • **Computing methodologies** → *Distributed computing methodologies*; • **Software and its engineering** → *Software performance*.

KEYWORDS

serverless computing, continuum computing, liquid software

ACM Reference Format:
Josef Spillner. 2021. Self-balancing Architectures based on Liquid Functions across Computing Continuums. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC '21) Companion (UCC '21 Companion), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3492323.3495589

1 INTRODUCTION

Computing across several devices is challenging and requires appropriate abstractions from software and data engineering to running code in order to fully exploit the available resources. In human-computing interaction, liquid software is an existing concept [6] to let humans move application interfaces between devices. In distributed systems, agents are a related concept given the ability to let application logic migrate between nodes. Agents are supposed to be smart [7] and therefore can decide by themselves when to run at which node, often assisted by appropriate platform support in terms of control loops, awareness and placement mechanisms. In software engineering, architectural principles for autonomous microservices [12] have been defined and similarly aim at better meeting the challenges of distributed systems. Such concepts gain relevance due to the emergence of computing continuums [1], combining several edge and fog devices, including mobile devices and sensors, with multi-cloud infrastructure and platform services. Complex application logic such as distributed or federated machine learning and data analytics is particularly on the rise across devices and are well understood concerning their communication properties [4]. Yet machine learning applications are hard to implement and deploy as per-device patchwork and can supposedly benefit from assuming a working continuum as fundamental higher-level abstraction, resembling an operating system managing heterogeneous distributed resources on behalf of applications.

Software engineers targeting such continuums could build monolithic agents or autonomous microservices compositions but require more appropriate design and engineering abstractions with little deviation from existing development practices. In recent years, cloud functions have gained popularity with software engineers to encapsulate application parts or to provide glue functionality in complex and scalable applications [5]. They convey a notion of serverless computing, hiding much of the runtime and isolation configuration. From the angle of agent technologies, functions are however rather inflexible agents that remain at the nodes they have been deployed to. Similarly, functions can be considered microservices that are instantiated on demand. Only few approaches for function execution in continuums have been proposed before, for instance with Delta, an intelligent function scheduler based on predictors and the funcX platform that can lead to 50%–80% execution time reduction [8]. Similar approaches are Pilot-Edge, characterised by support for heterogeneous workloads and decoupled resource management [10], and Colony, supporting three edge-cloud computing scenarios for workflows based on FaaS agents [9]. Osmotic computing has also been proposed to connect edges with data centres seamlessly [13]. Despite these advances, when infrastructure shifts from single clouds to more spread-out continuum topologies with access restrictions, the application engineer is still faced with the tasks of packaging, placing and invoking functions on these nodes,

UCC '21 Companion, December 6–9, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-9163-4/21/12…$15.00
https://doi.org/10.1145/3492323.3495589

tedious efforts not completely automated by services and abstractions such as Delta or Pilot-Edge. Moreover, Delta's implementation requires expertise and effort to operate along funcX and Globus, and Pilot-Edge is still in early stages as research proposal.

From the engineer's perspective, the main requirement to be fulfilled is therefore that these tasks become effortless and automated, including the declarative and conditional offloading and placement of functions to a single node or all nodes but also the migration and priorisation depending on load metrics, data affinity and expected capacities. This paper therefore marries the concepts of serverless computing, continuum computing and FaaSification by introducing the concept of light-weight liquid functions in the sense of agent code that will execute at the right nodes, requiring intermediate services only on unmanaged nodes, and requiring no explicit packaging and deployment. Section §2 introduces the concept including the liquid functions model and resource management concerns as well as an appropriate system architecture. Section §3 presents a reference implementation, including limitations thereof. Section §4 then validates the concept and the implementation by running representative functions over multiple continuum configurations.

2 CONCEPT

2.1 Application and System Model

The liquid functions concept assumes that the goal is to engineer scalable applications that span multiple nodes in a continuum. The shape of the continuum is often dependent on the application domain. Such applications are common on mobile devices (device–cloud or device–edge datacentre–cloud continuum) and in environmental sensing (sensor node–fog–cloud or sensor node–multicloud continuum). To facilitate liquid functions, the system model therefore assumes a continuum with at least partial visibility of the nodes according to a graph model, often reduced to trees or even sequences but potentially forming fully connected graphs. A node that is visible to another is meant to be eligible as target for requests, leading to the formation over overlay call graphs. Nodes can be managed or unmanaged concerning function execution. Fully managed nodes, such as commercial FaaS platforms, can be integrated as leaf nodes. Moreover, nodes can be access-protected, requiring credentials at the node previously appearing in the call graph. Finally, one node is the home node that serves as application entrypoint. A typical graph for a continuum is shown in Fig. 1 along with a potential overlay call graph. It represents a sensing edge home node that is in the field and thus not physically protected, but able to communicate with an intermediary node for submitting results but also for offloading computation. Due to higher protection levels, the intermediary can host the credentials to cloud platforms and can further offload computation to them. Leaf nodes without the ability to forward requests receive requests. This refers in particular to cloud platforms on which massive cloud-native elasticity is desired.

2.2 Function Model

Serverless computing providers have introduced a sophisticated functions model to accomodate for application needs. Functions can be deployed as code (FaaS) or as containers (CaaS), invoked synchronously or asynchronously, instantiated ad-hoc or from a provisioned concurrency pool, and configured according to expected

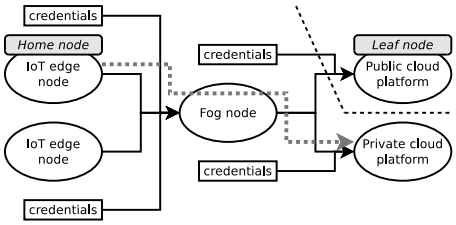

Figure 1: Exemplary graph of nodes in a continuum

memory consumption and isolation level. In the existing FaaS and CaaS models, functions are always invoked where they have been previously deployed. Their scalability is generally limited to the upper limit defined in a cloud region, and multiple regions need to be combined to reach beyond. Moreover, specific edge computing variations of FaaS such as Cloudflare Workers, Lambda@Edge and Greengrass exist but are likewise constraining functions to execute at static locations. Instead, the proposed function model introduces the concept of liquid functions that traverse a continuum as needed. It allows for further differentiation into three groups of liquid functions that are controlled by policy markers:

(1) Mobile functions. These are functions with relative portability, e.g. packaged as multi-architecture container images or portable code, that can be freely migrated between nodes or deployed to multiple nodes in parallel with subsequent selection at invocation time. The only restriction for mobility comes from the node topology, as some nodes may not be globally accessible but only through gateways.
(2) Pinned functions. Functions that are marked to remain statically bound to a node for technical or operational reasons. Among them are monitoring functions that need to operate once per node.
(3) Embedded functions. Functions that are not split off from application code but rather execute in-process, with mechanisms in place for offloading as pinned or mobile function.

As serverless application engineering is meant to reduce complexity, two request-driven invocation and execution modes are provided:

(1) Local execution. A function is executed at the node that receives the request. Mobile and pinned functions always execute locally.
(2) Forwarded execution. Based on contextual information such as the node capacities and system load, requests may be forwarded to other nodes according to defined topologies. For instance, an edge node may hand off requests to a cloud platform. The hypothesis is that the forwarding leads to load balancing across nodes and, from a system topology perspective, to self-balancing over time.

Depending on the algorithmic complexity and the desired isolation level, engineering and operating complexity can be further reduced by relying on function annotations in code. As opposed to previously proposed annotations in the literature, such as `@CloudFunction` [3], the concept of liquid functions permits a late decision on where functions shall be placed and executed. Hence,

an annotation of the type `@LiquidFunction` will take the load situation and topology constraints in the continuum into account.

In summary, liquid functions are characterised by their ability to be either embedded, pinned or mobile, by executing based on direct or forwarded requests, and if the programming language and complexity level permits, by offloading dynamically based on annotations.

2.3 Resource Management

The dynamic placement of functions on nodes participating in the continuum requires up-to-date knowledge about the available resources and the application needs. Three resource factors per node are of primary interest: CPU performance CP, CPU utilisation CU and memory utilisation MU. Among the factors per application, response time or makespan as well as concurrently serveable requests, limited by main memory and other spatial resources, are in the focus. More complex models exist to take further resources such as disks and network links as well as isolation overheads into account, e.g. Delta's and Pilot-Edge's models or the proposed Ephemeral Continuum based on context-aware resource federation [2], The liquid functions approach is meant to be able to use different resource management models whenever appropriate. Due to the novelty of these models, two rather simple bootstrapping models are introduced specifically to realise and investigate liquid functions in closer detail. These two low-complexity resource management models $LFRM0$ and $LFRM1$ shall be sufficient for reaping the benefits of liquid stateless compute-centric functions while allowing to be complemented with more complex models as the need arises.

In the first simple resource management model $LFRM0$, a benchmark function is executed upon initialisation of each node to determine CP. Subsequently, the inverse utilisation of all CPUs and the main memory is continuously determined in % and MB, respectively, and a logarithmic score S is determined as $s = CP \times log_{100}(CP * CU) + log_{1024}(CM)$. The higher the score, the more attractive a node is for placing and invoking functions. Function characteristics are not considered.

A second simple resource management model $LFRM1$ takes the opposite approach of being resource-agnostic while applying machine learning to the correlation of function distribution to execution correctness and performance. However, it requires a history of executions to lower the score of a node whenever faults or slowness occurs on it upon handling function execution requests. This model has two sub-models: one for the initial invocation of functions ($LFRM1 - Initial$), and one more aligned with practical needs to identify on which node to invoke the next function instance given a previous placement ($LFRM1 - NextRouting$). For instance, the historic information may inform that node A is the most suitable for one request and B for two requests. Once a first request is assigned to A and a second one arrives, routing it to A as well may be the next suitable option.

2.4 System Architecture

With the aim to achieve a self-balancing system for future continuum applications, the system architecture needs to implement the application, function and system models as well as appropriate resource management mechanisms, initially based on $LFRM0$ and $LFRM1$. To maintain modularity, parts of the system should be implemented as functions themselves, pinned to all unmanaged nodes, including the home node. This leads to an abstract system architecture that connects the function-based software application with nodes in the continuum topology. It assumes a number of pinned functions per node that gather statistics and facilitate the offloading mechanisms. Fig. 2 summarises the abstract system architecture, using an exemplary function `X` running either within the application (on an unspecified home node) or on one of the two nodes in the continuum, `LF1` or `LF2`. In case the function runs on `LF2`, it can be proxy-invoked via `LF1` or invoked directly from the caller that is assumed to be either collocated with the remaining application code or residing outside the network as function client.

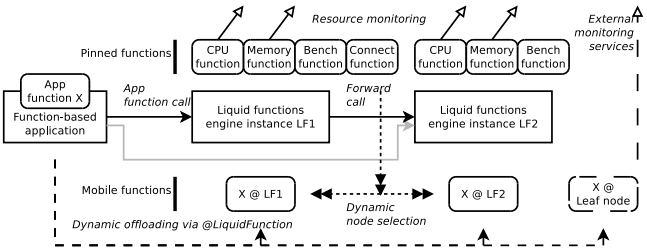

Figure 2: System architecture for continuum functions

2.5 Application Engineering Processes

Software applications are built either with an explicit distributed liquid function orientation or implicitly by marking functions on the code level with the mentioned `@LiquidFunction` annotation. Depending on the chosen engineering process, the system architecture is extended with libraries to handle offloading into the balanced runtime system and into leaf nodes. Depending on the resource management model, the load balancing and therefore the achievement of correctness and performance of the function execution may not take effect immediately. Hence, a typical engineering process is as follows: An application is implemented in code. Relevant code parts are annotated to execute as liquid functions. An initial placement and routing of invocations is conducted. Some function invocations may fail or become slower than others. The routing is adjusted until 100% correctness along with a peak performance are reached as irrevocable optimisation goals.

3 IMPLEMENTATION

3.1 Runtime Overview

CoRFu – Continuum-Ready Functions – is an open source reference implementation of the proposed system architecture for distributed FaaS[1]. It is a Python service that provides to clients a generic HTTP endpoint accepting requests of the form `/module/function [/parameters]`. Moreover, it listens on a message bus on which asynchronous or forwarded function invocations can communicate their results, in addition to cascading replies back to the caller in the case of forwarding.

[1] CoRFu implementation: https://doi.org/10.5281/zenodo.5650815

Topology files of the pattern `*.topology` describe the nodes participating in a continuum. CoRFu loads one of these files on startup and then proceeds to send API requests for setting up the interconnects between instances, achieving fully software-defined connectivity. Functions are loaded dynamically from Python module files that are placed in CoRFu's module folder. Deployment is outside the scope of CoRFu, although the implementation comes with a separate deployment client that uses SSH-based automation to run CoRFu instances according to the topology and deploy application functions accordingly. Moreover, a library (described below) can be used to offload functions dynamically. Multi-threading is used on several occasions. First, to handle incoming HTTP requests. Second, to run background activities such as resource monitoring that is entirely based on pinned liquid functions. Third, to invoke asynchronous application functions.

3.2 Library Overview

Through a custom CoRFu-related Python library, dynamic offloading of liquid functions into the continuum can be achieved based on the `@LiquidFunction` annotation on the code level, conveying the approach closer to software engineers. The functions are either written in Python directly, with or without in-code dependencies, or are encapsulated in Docker container images wrapped by Python executors. Any Python code that is offloaded is pickled through the Dill library [11], an extension to Python's own pickling framework known from the Pathos parallel graph framework for heterogeneous computing that brings the ability to pickle entire functions as first-class citizens. Pickling result in binary code files (`*.code`) that are loaded, again through Dill, as functions by CoRFu in addition to human-readable Python files (`*.py`). Function dependencies are handled as follows:

(1) In-code dependencies: By analysing the abstract syntax tree (AST) of a function, dependencies on other functions or modules are captured.
(2) Out-of-code file dependencies: Data files or external executables are currently not captured. However, extending the implementation to trace file open calls would be possible.
(3) Container image dependencies: By running `docker pull`, container images are transparently deployed on any node in the correct architecture, assuming multi-arch availability. The downside of this approach is an initial coldstart effect.

3.3 Limitations

CoRFu implements the minimum subset of a distributed FaaS system architecture that is necessary to validate its benefits in continuum topologies. It does not implement differential isolation (e.g. process execution, Docker containers or V8 isolates) or secure function execution (e.g. through enclaves) that would be necessary for production deployment, although it pragmatically does support Python-wrapped processes and containers. On the upside, this is beneficial for portability across CPU architectures. Due to its proof-of-concept nature, security is not in the focus at all and all functions execute in process without further isolation or authentication. Likewise, CoRFu does not implement function management interfaces that can be compared to the control planes of commercial FaaS. For the deployment and inter-instance connectivity, security measures such as certificates and credentials are also not implemented in the prototype. All of the missing functionality is present in other implementations and is therefore considered possible to add with modest engineering effort.

4 VALIDATION

4.1 Resources and Functions

Two short-running stateless functions for sensed image inference (tuning) and linear regression, both part of an application for sensor data processing, are evaluated. The linear regression is performed across two million trained records through Scikit-Learn, executing fully isolated and parallelisable through a Docker container. The image tuning, on the other hand, is a pure Python implementation using SciPy, executing in-process and only pseudo-parallelisable due to the global interpreter lock of the Python interpreter. Both functions are highly compute- and memory-intensive while having modest I/O requirements apart from small file loading.

First, the baseline is determined across four resources: An edge node (*edge*: Raspberry Pi 4x ARM CPU 1.5 GHz, 4 GB RAM), an intermediary fog node (fog: 4x x86_64 CPU 2.6 GHz, 16 GB RAM) a cloud node ($cloud1$: 8x x86_64 vCPU 2.5 GHz, 16 GB RAM) and a second cloud node ($cloud2$: like $cloud1$ but with 3.0 GHz vCPUs). The edge node is the home node according to the liquid functions model and has direct access to the fog and cloud nodes, but also indirect access to the cloud nodes through forwarding by the fog node. The baseline determination provides the outer hull for all resource management activities primarily with regard to the correctness of being able to execute a certain number of instances per function, and secondarily with regard to the performance. All experiments are repeated four times to yield meaningful average values. Subsequently, the benefits of liquid function distribution is assessed by awaiting a balanced distribution of function requests, through the algorithms $LFRM0$, $LFRM1 - Initial$ and $LFRM1 - NextRouting$. While $LFRM0$ is an online algorithm requiring no training, $LFRM1$ is trained with the results from the baselines to predict optimal invocation routing.

With this setup, the end-to-end application engineering process including annotation-triggered deployment of functions into the continuum and achieving a load-balanced execution is performed to validate the liquid functions approach holistically.

4.2 Baseline Results

The baseline consists of combinations of even instance counts from 2 to 20 per node. Fig. 3(a) shows that the edge is resource-constrained and unable to serve more than eight concurrent linear regression function requests. As can be seen in Fig. 3(b), the edge does benefit from multiple cores, but with only four cores the speed-up is minimal, reverted beyond four concurrent requests, and unreliable (and hence unusable for prediction) beyond eight concurrent requests. Whenever the load is too high, it may even become unstable, its CPU frequency-capped and throttled, and finally it may even perform sudden reboots. Hence, an artificial cold start situation is also shown in the figure with slightly better performance values, but is unlikely to occur in long-running operation in practice. On the other hand, the clouds execute successfully even for 20 concurrent requests, and also benefit significantly from multi-core

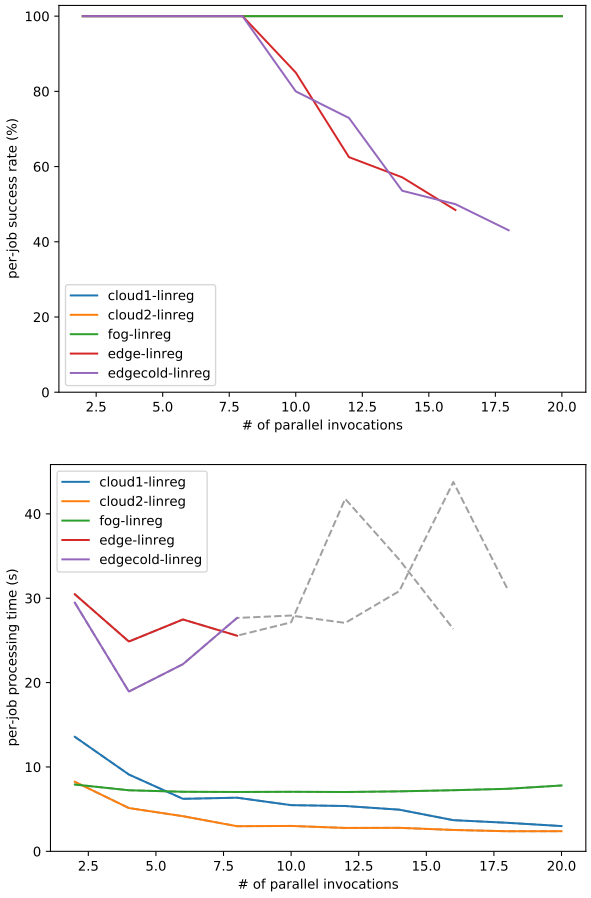

Figure 3: Correctness (a) and performance (b) of linear regression function – baseline per node

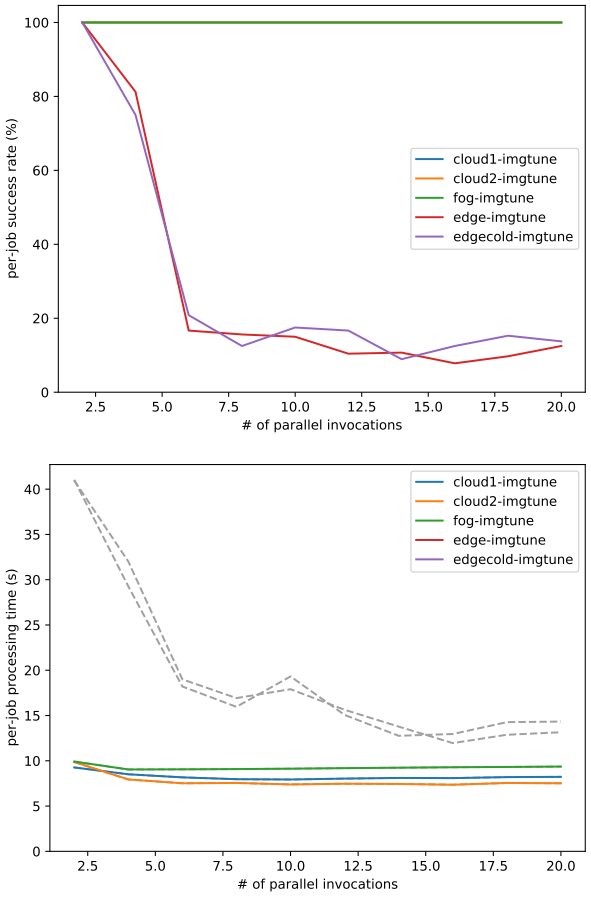

Figure 4: Correctness (a) and performance (b) of image tuning function – baseline per node

execution, achieving almost monotonic albeit small speedups with increasing invocation concurrency.

For comparison, Fig. 4(a,b) shows the correctness and performance of the image tuning function. Again and to a larger degree, the memory limitation of the edge node becomes apparent and motivates a coordinated and immediate offloading due to the edge configuration being completely unusable for prediction.

4.3 Metrics Inference

The baseline experiments with up to 20 function instances for each of the four nodes open up a configuration space with a theoretic maximum of 160,000 placement combinations. The baseline experiments cover 9,000 of them, or approximately 6%. For each of the up to 80 concurrent function placements, the best combination is determined as the minimum across all maximum durations per node, and recorded as inference knowledge base for $LFRM1-Initial$, represented as 325 lines of JSON. Moreover, the baseline delivers 17994 best next routing decisions in the range of 1–4 incoming requests. These are also recorded for $LFRM1-NextRouting$, represented as approximately 180,000 lines of JSON. Interpolation is used to look up desired combinations not covered by the baseline, for instance, any odd number of function instances per node.

Figs. 5 and 6 overlays the baseline results from Fig. 3(b) and Fig. 4(b) with the expected optimal performance results from the knowledge base available to the $LFRM1-Initial$ model (dashed grey line) Moreover, they show the actual results (coloured lines) with both the dynamic $LFRM0$ model that has no expectation equivalent and the $LFRM1-Initial$ model that are occasionally even better than expected, but in general match the expectations. For both functions, due to the fast execution on $cloud2$, most of the invocations are scheduled there except for the case of two concurrent requests, which due to network overhead are faster when running on the locally connected fog node (linear regression) or $cloud1$ (image tuning), and for the case of 20 concurrent requests, which due to emerging overload of $cloud2$ leads to offloading 14 (linear regression) or 10 (image tuning) of the requests to $cloud1$. The edge node is never an option for either function.

Although $LFRM0$ beats $LFRM1-Initial$ performance-wise for most concurrency configurations, it is less predictable and subject to the frequency and precision of determining the resource utilisation. Overall, the liquid functions approach is proven to work and deliver

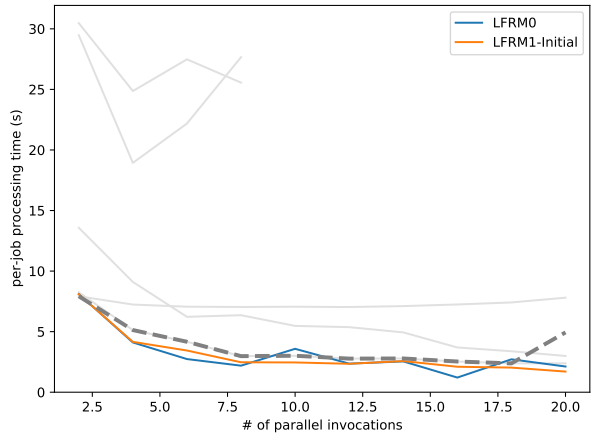

Figure 5: Performance of linear regression function when applying the *LFRM*0 and *LFRM*1 − *Initial* models

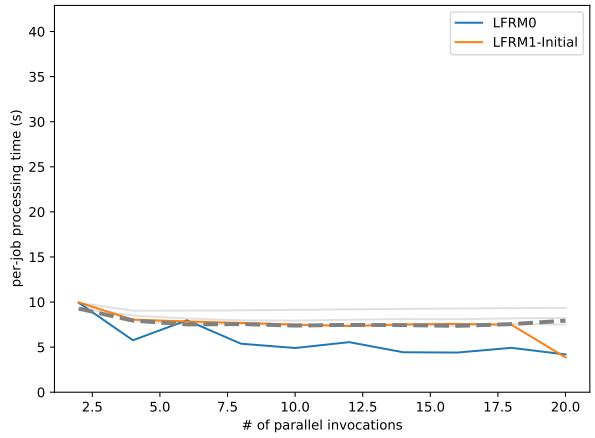

Figure 6: Performance of image tuning function when applying the *LFRM*0 and *LFRM*1 − *Initial* models

minimised response times across devices. Future research efforts can be directed on optimising the resource management models within the existing framework.

5 CONCLUSIONS

This paper has introduced the concept of liquid functions in continuums, technically backed up with two simple resource management models *LFRM*0 and *LFRM*1, a runtime system implementation CoRFu, and a library for engineers of function-based applications targeting both pure Python-based and Docker-encapsulated functions. The implementation has been shown to be usable in practice and effective when comparing execution correctness and performance against the baseline resources.

Due to the massive uptake of mobile devices, sensing in IoT and industrial equipment, and other edge and fog resources, the question about how to address the emerging continuums remains relevant. This question concerns in particular the development and deployment processes of software running in these continuums. With liquid functions, serverless computing concepts are re-interpreted for continuums, and are abstracted to become useful for applications highly distributed across a topology of nodes that do not necessarily have full mutual visibility, suggesting usefulness for online machine learning algorithms in IoT among other use cases.

Future work remains to be conducted in workload characterisation and prediction as well as standardisation in function and resource behaviour descriptions. Moreover, due to the proliferation of continuums, distributed (including balanced, federated and decentralised) function execution will require further investigation and applied domain-specific experiments.

REFERENCES

[1] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. 2019. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *Int. J. High Perform. Comput. Appl.* 33, 6 (2019). https://doi.org/10.1177/1094342019877383

[2] Emanuele Carlini, Patrizio Dazzi, Luca Ferrucci, and Matteo Mordacchini. 2021. Efficient Resources Distribution for an Ephemeral Cloud/Edge continuum. *CoRR* abs/2107.07195 (2021). arXiv:2107.07195 https://arxiv.org/abs/2107.07195

[3] Serhii Dorodko and Josef Spillner. 2018. Selective Java code transformation into AWS Lambda functions. In *Proceedings of the European Symposium on Serverless Computing and Applications, ESSCA@UCC 2018, Zurich, Switzerland, December 21, 2018 (CEUR Workshop Proceedings, Vol. 2330)*, Josef Spillner (Ed.). CEUR-WS.org, 9–17. http://ceur-ws.org/Vol-2330/paper2.pdf

[4] Yubin Duan, Ning Wang, and Jie Wu. 2021. Minimizing Training Time of Distributed Machine Learning by Reducing Data Communication. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 1802–1814. https://doi.org/10.1109/TNSE.2021.3073897

[5] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Alex Ozdemir, Shuvo Chatterjee, Matei Zaharia, Christos Kozyrakis, and Keith Winstein. 2019. Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg. *login Usenix Mag.* 44, 3 (2019). https://www.usenix.org/publications/login/fall2019/fouladi

[6] Andrea Gallidabino, Cesare Pautasso, Tommi Mikkonen, Kari Systä, Jari-Pekka Voutilainen, and Antero Taivalsaari. 2017. Architecting Liquid Software. *J. Web Eng.* 16, 5&6 (2017), 433–470. http://www.rintonpress.com/xjwe16/jwe-16-56/433-470.pdf

[7] Andrea Giordano, Giandomenico Spezzano, and Andrea Vinci. 2016. Smart Agents and Fog Computing for Smart City Applications. In *Smart Cities - First International Conference, Smart-CT 2016, Málaga, Spain, June 15-17, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9704)*, Enrique Alba, Francisco Chicano, and Gabriel Luque (Eds.). Springer, 137–146. https://doi.org/10.1007/978-3-319-39595-1_14

[8] Rohan Kumar, Matt Baughman, Ryan Chard, Zhuozhao Li, Yadu N. Babuji, Ian T. Foster, and Kyle Chard. 2021. Coding the Computing Continuum: Fluid Function Execution in Heterogeneous Computing Environments. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*. IEEE, 66–75. https://doi.org/10.1109/IPDPSW52791.2021.00018

[9] Francesc Lordan, Daniele Lezzi, and Rosa M. Badia. 2021. Colony: Parallel Functions as a Service on the Cloud-Edge Continuum. In *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12820)*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). Springer, 269–284. https://doi.org/10.1007/978-3-030-85665-6_17

[10] Andre Luckow, Kartik Rattan, and Shantenu Jha. 2021. Pilot-Edge: Distributed Resource Management Along the Edge-to-Cloud Continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 874–878. https://doi.org/10.1109/IPDPSW52791.2021.00130

[11] Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. 2012. Building a Framework for Predictive Science. *CoRR* abs/1202.1056 (2012). arXiv:1202.1056 http://arxiv.org/abs/1202.1056

[12] Anders Mikkelsen, Tor-Morten Grønli, Damian A. Tamburri, and Rick Kazman. 2020. Architectural Principles for Autonomous Microservices. In *53rd Hawaii International Conference on System Sciences, HICSS 2020, Maui, Hawaii, USA, January 7-10, 2020*. ScholarSpace, 1–10. http://hdl.handle.net/10125/64546

[13] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer F. Rana, and Rajiv Ranjan. 2016. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. *IEEE Cloud Comput.* 3, 6 (2016), 76–83. https://doi.org/10.1109/MCC.2016.124