FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC

Josef Spillner[1], Cristian Mateos[2], and David A. Monge[3]

[1] Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/), 8401 Winterthur, Switzerland
`josef.spillner@zhaw.ch`
[2] ISISTAN Research Institute - CONICET - UNICEN
Campus Universitario, Paraje Arroyo Seco, Tandil (7000), Buenos Aires, Argentina
`cristian.mateos@isistan.unicen.edu.ar`
[3] ITIC Research Institute, National University of Cuyo
Padre Jorge Contreras 1300, M5502JMA Mendoza, Argentina
`dmonge@uncu.edu.ar`

Abstract. The adoption of cloud computing facilities and programming models differs vastly between different application domains. Scalable web applications, low-latency mobile backends and on-demand provisioned databases are typical cases for which cloud services on the platform or infrastructure level exist and are convincing when considering technical and economical arguments. Applications with specific processing demands, including high-performance computing, high-throughput computing and certain flavours of scientific computing, have historically required special configurations such as compute- or memory-optimised virtual machine instances. With the rise of function-level compute instances through Function-as-a-Service (FaaS) models, the fitness of generic configurations needs to be re-evaluated for these applications. We analyse several demanding computing tasks with regards to how FaaS models compare against conventional monolithic algorithm execution. Beside the comparison, we contribute a refined FaaSification process for legacy software and provide a roadmap for future work.

1 Research Direction

The ability to turn programmed functions or methods into ready-to-use cloud services is leading to a seemingly serverless development and deployment experience for application software engineers [1]. Without the necessity to allocate resources beforehand, prototyping new features and workflows becomes faster and more convenient to application service providers. These advantages have given boost to an industry trend consequently called Serverless Computing. The more precise, almost overlapping term in accordance with Everything-as-a-Service (XaaS) cloud computing taxonomies is Function-as-a-Service (FaaS) [4]. In the FaaS layer, *functions*, either on the programming language level or as abstract concept around binary implementations, are executed synchronously or asynchronously through multi-protocol triggers. Function instances are provisioned on demand through coldstart or warmstart of the implementation in conjunction with an associated configuration in few milliseconds, elastically scaled as needed, and charged per invocation and per product of period of time and resource usage, leading to an almost perfect pay-as-you-go utility pricing model [11]. FaaS is gaining traction primarily in three areas. First, in Internet-of-Things applications where connected devices emit data sporadically. Second, for web applications with light-weight backend tasks. Third, as glue code between other cloud computing services. In contrast to the industrial popularity, no work is known to us which explores its potential for scientific and high-performance computing applications with more demanding execution requirements.

From a cloud economics and strategy perspective, FaaS is a refinement of the platform layer (PaaS) with particular tools and interfaces. Yet from a software engineering and deployment perspective, functions are complementing other artefact types which are deployed into PaaS or underlying IaaS environments. Fig. 1 explains this positioning within the layered IaaS, PaaS and SaaS service classes, where the FaaS runtime itself is subsumed under runtime stacks. Performing experimental or computational science research with FaaS implies that the two roles shown, end user and application engineer, are adopted by a single researcher or a team of researchers, which is the setting for our research.

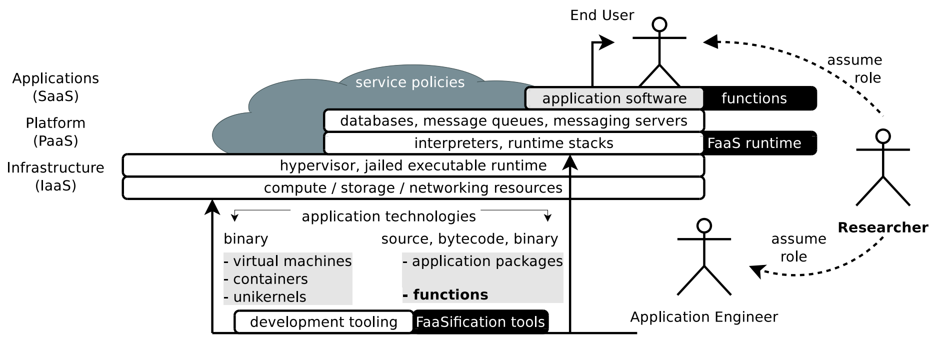

Fig. 1. Positioning of FaaS in cloud application development

The necessity to conduct research on FaaS for further application domains stems from the unique execution characteristics. Service instances are heuristically stateless, ephemeral, and furthermore limited in resource allotment and execution time. They are moreover isolated from each other and from the function management and control plane. In public commercial offerings, they are billed in subsecond intervals and terminated after few minutes, but as with any cloud application, private deployments are also possible. Hence, there is a trade-off between advantages and drawbacks which requires further analysis. For example, existing parallelisation frameworks cannot easily be used at runtime as function instances can only, in limited ways, invoke other functions without the ability to configure their settings. Instead, any such parallelisation needs to be performed before deployment with language-specific tools such as Pydron for Python [10] or Calvert's compiler for Java [3]. For resource- and time-demanding applications, no special-purpose FaaS instances are offered by commercial cloud providers. This is a surprising observation given the multitude of options in other cloud compute services beyond general-purpose offerings, especially on the infrastructure level (IaaS). These include instance types optimised for data processing (with latest-generation processors and programmable GPUs), for memory allocation, and for non-volatile storage (with SSDs). Amazon Web Services (AWS) alone offers 57 different instance types. Our work is therefore concerned with the assessment of how current generic *one-size-fits-all* FaaS offerings handle scientific computing workloads, whether the proliferation of specialised FaaS instance types can be expected and how they would differ from commonly offered IaaS instance types. In this paper, we contribute specifically (i) a refined view on how software can be made fitting into special-purpose FaaS contexts with a high degree of automation through a process named *FaaSification*, and (ii) concepts and tools to execute such functions in constrained environments.

In the remainder of the paper, we first present background information about FaaS runtimes, including our own prototypes which allow for provider-independent evaluations. Subsequently, we present four domain-specific scientific experiments conducted using FaaS to gain broad knowledge about resource requirements beyond general-purpose instances. We summarise the findings and reason about the implications for future scientific computing infrastructures.

2 Background on Function-as-a-Service

2.1 Programming Models and Runtimes

The characteristics of function execution depend primarily on the FaaS runtime in use. There are broadly three categories of runtimes:

1. Proprietary commercial services, such as AWS Lambda, Google Cloud Functions, Azure Functions and Oracle Functions.
2. Open source alternatives with almost matching interfaces and functionality, such as Docker-LambCI, Effe, Google Cloud Functions Emulator and OpenLambda [6], some of which focus on local testing rather than operation.
3. Distinct open source implementations with unique designs, such as Apache OpenWhisk, Kubeless, IronFunctions and Fission, some of which are also available as commercial services, for instance IBM Bluemix OpenWhisk [5]. The uniqueness is a consequence of the integration with other cloud stacks (Kubernetes, OpenStack), the availability of web and command-line interfaces, the set of triggers and the level of isolation in multi-tenant operation scenarios, which is often achieved through containers.

In addition, due to the often non-trivial configuration of these services, a number of mostly service-specific abstraction frameworks have become popular among developers, such as PyWren, Chalice, Zappa, Apex and the Serverless Framework [8]. The frameworks and runtimes differ in their support for programming languages, but also in the function signatures, parameters and return values. Hence, a comparison of the entire set of offerings requires a baseline.

The research in this paper is congruously conducted with the mentioned commercial FaaS providers as well as with our open-source FaaS tool *Snafu* which allows for managing, executing and testing functions across provider-specific interfaces [14]. The service ecosystem relationship between Snafu and the commercial FaaS providers is shown in Fig. 2. Snafu is able to import services from three providers (AWS Lambda, IBM Bluemix OpenWhisk, Google Cloud Functions) and furthermore offers a compatible control plane to all three of them in its current implementation version. At its core, it contains a modular runtime environment with prototypical maturity for functions implemented in JavaScript, Java, Python and C. Most importantly, it enables repeatable research as it can be deployed as a container, in a virtual machine or on a bare metal workstation. Notably absent from the categories above are FaaS offerings in e-science infrastructures and research clouds, despite the programming model resembling widely used job submission systems. We expect our practical research contributions to overcome this restriction in a vendor-independent manner. Snafu, for instance, is already available as an alpha-version launch profile in the CloudLab testbed federated across several U.S. installations with a total capacity of almost 15000 cores [12], as well as in EGI's federated cloud across Europe.

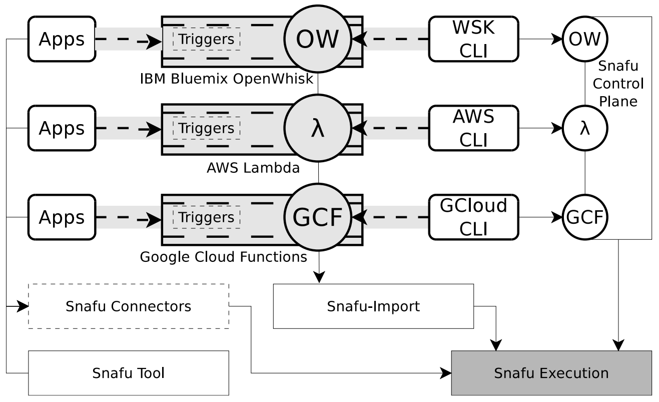

Fig. 2. Snafu and its ecosystem and tooling

Using Snafu, it is possible to adhere to the diverse programming conventions and execution conditions at commercial services while at the same time controlling and lifting the execution restrictions as necessary. In particular, it is possible to define memory-optimised, storage-optimised and compute-optimised execution profiles which serve to conduct the anticipated research on generic (general-purpose) versus specialised (special-purpose) cloud offerings for scientific computing. Snafu can execute in single process mode as well as in a load-balancing setup where each request is forwarded by the master instance to a slave instance which in turn executes the function natively, through a language-specific interpreter or through a container. Table 1 summarises the features of selected FaaS runtimes.

Table 1. FaaS runtimes and their features

Runtime	Languages	Programming model	Import/Export
AWS Lambda	JavaScript, Python, Java, C#	Lambda	–
Google Cloud Functions	JavaScript	Cloud Functions	–
IBM Bluemix OpenWhisk	JavaScript, Python, Swift, Docker	OpenWhisk	–
Fission	JavaScript, Python, Go, C#, PHP	Fission	–
Kubeless	JavaScript, Python	Kubeless	–
Snafu	JavaScript, Python, C, Java	Lambda, OpenWhisk, Cloud Functions	Lambda, OpenWhisk, Cloud Functions, Fission

2.2 Providers and Performance

Commercial FaaS offerings differ not only in their programming conventions, but also vastly by performance, cost and the combined utility defined as cost-duration product. Fig. 3 informs about the benchmark of a compute-intensive function, $fib(38)$, implemented in Python as a *portable hosted function* which runs on bare metal as well as in AWS Lambda, IBM Bluemix OpenWhisk and Azure Functions. The performance in the FaaS environments will unlikely be faster than an approximate performance barrier indicated by the benchmark's runtime with the fastest widely available processor cores, exemplified by a 9.5 s execution time on a recent Intel Xeon E5-2660 v3 (Haswell) with 2.6 GHz and 8.5 s on an Intel i7-4800MQ with 2.7 GHz. Conversely, it is often much slower, and the offerings do not allow for explicitly paying more for better performance. The rather odd (although even) number 38 has thus been chosen so that on the slowest commercial FaaS offering, Lambda with 128 MB instances, the function terminates successfully before the obligatory five minutes timeout.

Lambda performs proportional to the memory assignment, thus leading to a constant price. OpenWhisk raises the price proportional to the memory assignment while keeping a constant performance. In contrast, Azure measures the memory use but does not allow for its configuration. Finally, while Google Cloud Functions do not appear in the same diagram due to its limitation to JavaScript, its pricing model almost resembles the one of Lambda although the

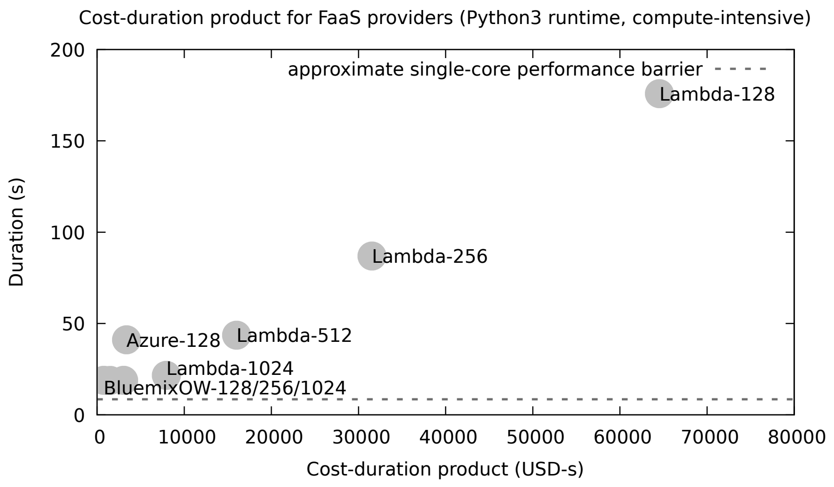

Fig. 3. CDP benchmark of FaaS providers, averaged over ten iterations, without free tier discounts

high-memory instances performance decreases in relation. The compute-intensive recursive benchmark implementation is shown in Listing 1.1. Functions with other characteristics will be analysed in detail in the next section.

Listing 1.1. Compute-intensive portable hosted Fibonacci function in Python

```
import time
def fib(x):
  if x in (1, 2):
    return 1
  return fib(x - 1) + fib(x - 2) # recursion
# AWS Lambda entry point
def lambda_handler(event, context):
  return fib(38)
# IBM Bluemix OpenWhisk entry point
def main(event):
  return {'ret': fib(38)}
# Microsoft Azure entry point (could be conditional)
import os
datain = open(os.environ['req']).read()
response = open(os.environ['res'], 'w')
response.write(str(fib(38)))
response.close()
```

3 Scientific Computing Experiments with Functions

In order to get a broad understanding of the feasibility and utility of FaaS models for diverse computing tasks, four experiments are conducted to compare the performance and other resource-related characteristics. The selected domains are mathematics (calculation of π), computer graphics (face detection), cryptology (password cracking) and meteorology (precipitation forecast). The first three experiments are synthetic, while the fourth one uses FaaSification to analyse an existing non-FaaS application. We will refer to the domain and function execution characteristics to infer statements about the possible and desirable degree of parallelisation without resource contention.

3.1 Mathematics: Calculation of π

A common formula to calculate arbitrary digits of π in parallel sequences for unlimited precision is Baily-Borwein-Plouffe (BBP). The implementation in Python uses the *Decimal* type explicitly due to the otherwise limited precision of built-in floating point numbers, which contrasts the unlimited digits of built-in integer numbers. The experiment is set up to calculate 2000 digits with a theoretic precision of 10000 digits.

Fig. 4 compares the BBP calculation performance between the native Python 3 execution, the optimised (JiT-compiled) PyPy 3 execution, as well as the FaaS equivalents with in-process and out-of-process parallelisation, or multi-threading and multi-processing, respectively.

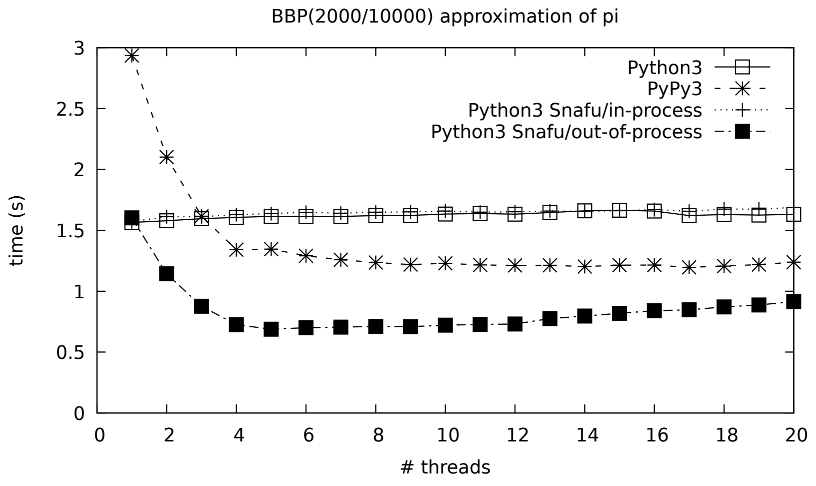

Fig. 4. Comparison of BBP(2000/10000) implementation performance with Python/PyPy 3.5

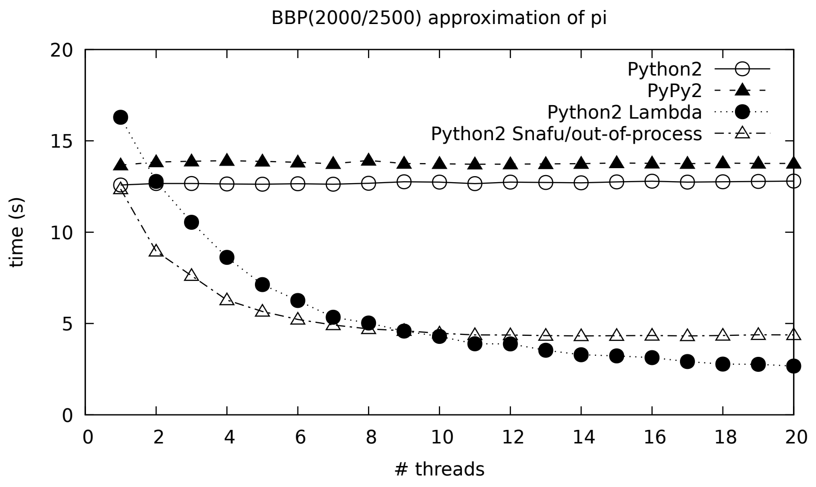

Fig. 5. Comparison of BBP(2000/2500) implementation performance with Python 2.7

Fig. 5 shows the equivalent performance measurements when running all implementations with Python 2. While this version is in maintenance mode and any development of the language and libraries will cease around the year 2020, it is still widely used, most prominently in the AWS Lambda service which is one of the few commercial FaaS services offering a native Python runtime. An interesting observation is that the programming language version matters. While PyPy 3 scales much better than the corresponding CPython implementation for a number of threads equal to or larger than the number of physical cores, PyPy 2 shows no such behaviour. In both cases, an external out-of-process function execution or hosted function execution is faster despite network transmission, due to overcoming multi-threading restrictions in the Python interpreters. Consequently, using compute-optimised FaaS is beneficial from a performance point of view when multi-processing, especially at scale, is not an option.

3.2 Computer Graphics: Face Detection

Face detection and recognition have become widely used techniques in social networks, robotics and photo management applications, but also in surveillance networks. The OpenCV library is among the commonly used tools to perform face detection and mark or extract the corresponding sections in photos. Fig. 6 shows an example of a person's face detected and marked by OpenCV in a lake scenery photo. This experiment performs the same detection and marking on a large number of photos.

A reference dataset with faces is provided by Faces in the Wild [2] which serves as useful input for input-output-centric file processing. The dataset contains 30281 images with a total size of 1.4 GB.

Fig. 6. Face detection and marking in photos using OpenCV with Python 2.7

As OpenCV is not yet widely available for Python 3, the implementation is based on Python 2. Fig. 7 shows the performance measurements for the function of the face detection using the public dataset again using pure local execution as well as function execution in Snafu and Lambda. Due to the I/O-centric nature of the function, the Lambda performance lags significantly and only becomes competitive when 19 or more threads are used. With in-function parallelisation, this effect can be remedied so that Lambda already executes faster with 10 threads or more. Slower Lambda functions are furthermore forcibly terminated due to reaching the timeout barrier.

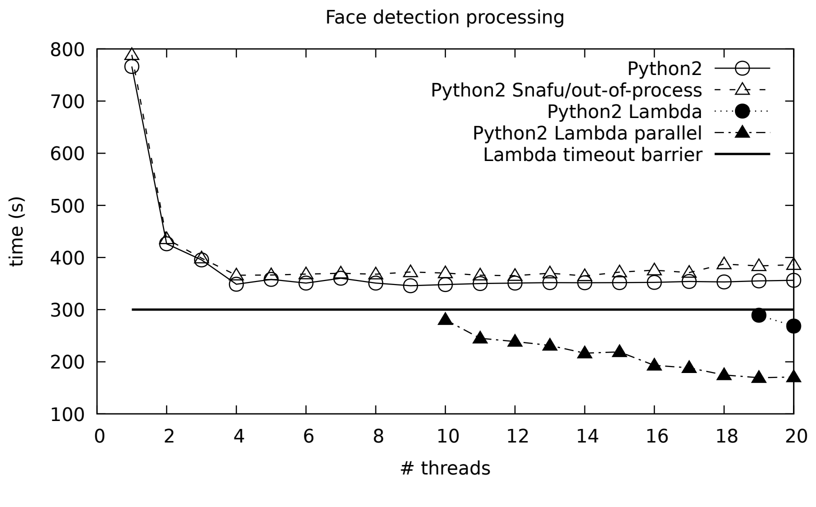

Fig. 7. Comparison of FacesDetect implementation performance with Python 2.7

The I/O lag in Lambda results from the use of the S3 object storage service compared to direct file system access in the other cases. This discrepancy makes evident the need for research tools beyond just the FaaS execution, as functions are often interlocked with additional cloud services. First prototypes such as the Atlassian LocalStack which simulates AWS services are already available [7]. Furthermore, the latency could be reduced by deploying function instance-affine local storage, preferably in memory, a storage-optimised facility not yet offered commercially by any of the FaaS providers.

3.3 Cryptology: Password Cracking

Digital forensics tools include functionality to crack passwords based on leaked or otherwise published hashes. Research on secure hashes ensures that only a brute-force attempt to crack them will succeed. Parallelisation and map-reduce operations are helpful to speed up the process.

The associated experiment consists of 100 SHA256-hashed passwords of up to 3 characters for which all possible combinations are tried in a comparison. The implementation makes use of two layers of parallelisation: Conventional parallel processing of up to 10 workers dividing the passwords among them, always returning a single result, and map-reduce processing of up to 10 mappers dividing the character ranges per character, returning the first successful match. The implementation in Python further makes use of the built-in concurrency framework which provides a unified futures interface for multi-threading and multi-processing. Due to the compute-centric algorithm and Python's global interpreter lock, the use of multi-threading does not lead to speed-ups. Hence, multi-processing is compared with a multi-function mode called *function futures* contributed by us which outsources the function and mapping calls to Lambda similar to the processing mode of PyWren [8].

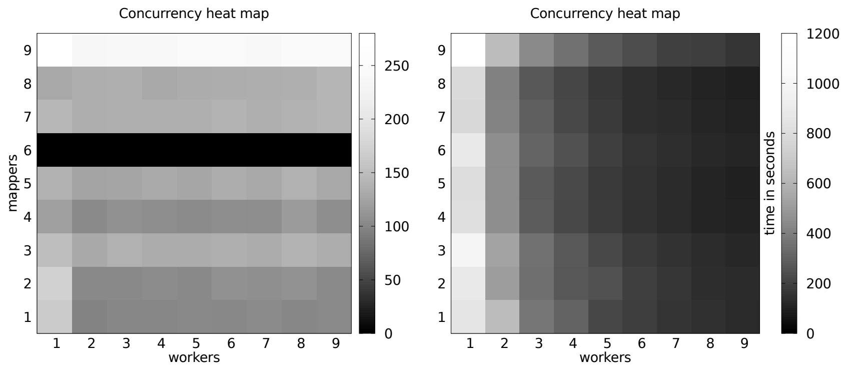

Fig. 8. Cracking of 100 passwords through a double-map-reduce process with combinations of workers and mappers; left: local multi-processing; right: multi-function on Lambda (darker is faster)

Fig. 8 compares the local multi-processing execution and the multi-function execution on Lambda. The Lambda setup consists of 512 MB of memory assignment to each function instance which affects the absolute performance. This is why the z-axis (brightness scale) has been normalised between the graphs. Comparing both results, the linear worker scaling is evidently more predictable whereas the mapper scaling is not contributing due to the overhead compared to the tiny processing spans of each mapper. Furthermore, the local system exposed a bug in Python's multi-processing code which led to some combinations fail sometimes, any with 6 mappers fail consistently, and any with 9 mappers resulting in a runtime almost twice as high as the preceding ones.

3.4 Meteorology: Precipitation Forecast

Weather forecasting is a typical use case for supercomputing environments. The forecast of precipitation in particular relies on many different models, including heuristic and fuzzy ones, which are parameterised with a multitude of variables and hyperparameters. However, while some forecasts run continuously, specialised forecasts only need to be run at certain times at full scale with high re-use factor of smaller proven functions, which could mean that FaaS is a suitable deployment model for this domain.

The experiment consists of running an implementation which confirms precipitation data from the Buenos Aires and Mendoza metropolitan regions in Argentina. It is based on neuronal nets implemented in Python, the Keras library and TensorFlow. As it is an existing application which can not readily be deployed into a FaaS environment, the research interest shifts to an earlier stage in the service provisioning lifecycle, to the pre-deployment software development. Tools are required which transform existing code into functions in conformance with the programming conventions expected by the target provider. The transformation process is consequently called FaaSification. In the experiment in question, several monolithic functions are used which need to be subdivided into several functions as part of this process to not exceed the FaaS providers' execution time limits. We have designed and implemented a tool called function splitter for evaluating the feasibility to split Python functions automatically. The tool employs the concept of *worm functions*. While a function instance may be terminated early due to a timeout, any other instance invoked by it just before the timeout will not be affected and will get its own counter. Thus, a function instance's state can be carried on to another instance of the same or another, sequentially related, partial function. The function splitter traces the use of local variables in a first partial function F_1 and adds them to the signature of the subsequent partial function F_2. The concept is explained with an example in Listing 1.2 which shows the split after line 1 of the original function F which contains a call to another function G.

Fig. 9 shows the performance of the forecast function F split into two partial functions, F_1 and F_2, with an increasing number of lines in F_1 instead of F_2. One interesting observation is that independent from the division into two functions, the runtime of each successive function instance increases substantially in the long term. A second observation is the pivotal point in which F_1 almost swaps its execution time with F_2, which happens in line 18 of the code.

Listing 1.2. Example for function splitting based on worm functions

```
# original                # partials
def F(x):                 def F1(x):
    y = x + 1                 y = x + 1
    --------                  return F2(x, y)
    --------              def F2(x, y):
    z = y - 2                 z = y - 2
    z += G(x)                 z += G(x)
    return z                  return z
```

This effect is a sure signal of a call to another function, either local or remote, which in turn needs to be subdivided to bring the partial function runtimes below the acceptable limits, corresponding to the call to G in the example. Currently, even the fastest partial execution time of F_2, which is 289 s, would almost reach Lambda's 300 s limit, and some even exceed Cloud Function's 540 s limit.

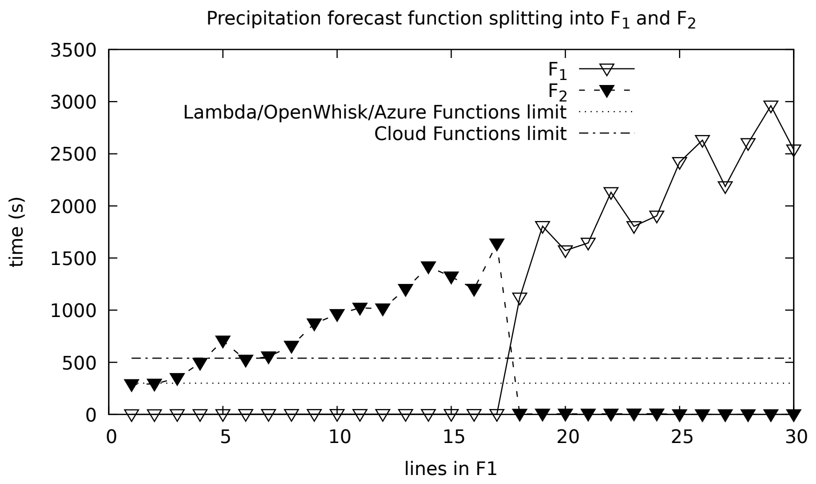

Fig. 9. Characteristics of forecast function splitting

The function splitter has been integrated with Lambada, a FaaSification tool for Python which is publicly available [13], which otherwise had performed a 1:1 translation of functions in the code to hosted functions in FaaS environments.

4 Findings

A general observation is that despite the still immature programming and deployment models for FaaS, experimental implementations from four different scientific computing domains have been successfully executed on both commercial and self-hosted FaaS runtimes.

The computing requirements of the four domains differ significantly with respect to the utilisation of compute, storage and network resources. Fig. 10 gives an exemplary insight into the processing as well as disk and network input-output characteristics of the face detection function from the computer graphics domain and the password cracking function from the cryptology domain. Heavy spikes of usually 5 MB/s read and write operations, peaking in more than 9 MB/s, can be observed in the first, and low CPU use due to network I/O waiting in the second.

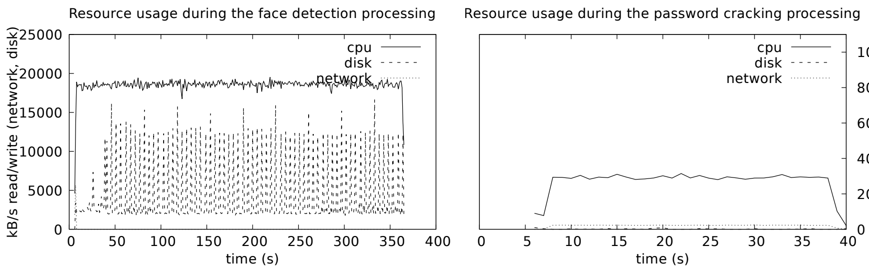

Fig. 10. Resource characteristic of two selected functions

Apart from the runtime, the findings also cover the development time. Through our work on scientific applications, we are now able to suggest the following refined classification of FaaSification process levels.

– *Shallow FaaSification*: classes or function collections divided into corresponding FaaS units. Functions or methods are the atomic units on this level.
– *Medium FaaSification*: functions divided by lines into regrouped or split FaaS units. Lines of code are the atomic units on this level.
– *Deep FaaSification*: single lines divided into multiple FaaS units or parameterised FaaS instances. Instructions are the atomic units on this level.

Existing function deployment and execution implementations cover these levels to various degrees, calling for future work towards deep FaaSification for special-purpose FaaS instances. The function execution environment Snafu performs a simple shallow FaaSification for different programming languages assuming single-file implementations. Existing transformation tools such as the triaged Lambada, but also structurally similar ones such as Podilizer or Termite, perform a thorough shallow FaaSification which also works for complex projects with multiple source files for Python and Java, respectively. Lambada furthermore now contains an implementation for medium and partial deep FaaSification, although in the case of subdivided functions, all statements are still executed serially instead of in parallel, suggesting further research to combine the work with automated parallelisation.

In order to become useful for a wider group of users in scientific computing, future research needs to concentrate on deep FaaSification, linking it to compiler research in which optimisations, code rewrites and parallelisation take place behind the scenes from an engineering point of view, outperforming almost any manual target-specific optimisation.

Table 2 summarises means to overcome limitations in contemporary FaaS environments with the purpose to execute resource-demanding scientific computing jobs, divided into three categories. Our contributions are found within all of the categories.

Table 2. Limitations and solutions

Limitation	Solution	Status
	Resources	
CPU	compute-optimised hardware	not commercially offered
	manual parallelisation	*function futures* (in this paper 🗎)
	automated parallelisation	Pydron [10]
Memory	map-reduce	reference architecture [9]
Network	local/function-affine services	not commercially offered
	Time and Cost	
Runtime	bypassing temporal limits	*worm functions* (in this paper 🗎)
	standard benchmarks	future work
Cost	self-hosted runtimes	available, but lack adaptive function migration
	Software	
Environment	simulated services	Localstack [7]
Development	function subdivision	*Deep FaaSification* (in this paper 🗎)

5 Summary and Repeatability

The execution of resource-intensive jobs in controlled environments with requirements on repeatability is an atypical use case for serverless computing whose predominant value proposition is exactly hiding any infrastructural configuration. On the other hand, the true on-demand provisioning and billing of hosted functions makes them attractive for research tasks. Our analysis has shown that in many domains of scientific and high-performance computing, solutions can be engineered based on simple functions which are executed on commercially offered or self-hosted FaaS platforms. In this paper we have contributed novel FaaS-related concepts (worm functions, function futures, deep FaaSification) and tools (function subdivision) to improve this engineering process. Furthermore, we have argued for the usefulness of our previous tools (Snafu, Lambada) for researchers and practitioners in this context.

We still see the need for future work in standard benchmarks, tooling for debugging and autotuning, and improved transformation tools to allow for more conventional software to run natively as functions without the need to use abstraction layers such as containers.

All applications, data and scripts used in our experiments are made available for anybody interested to recompute the results and repeat the analysis through the corresponding Open Science Framework repository located at `https://osf.io/8qt3j/`.

References

1. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless Computing: Current Trends and Open Problems. arχiv:1706.03178 (June 2017)
2. Berg, T.L., Berg, A.C., Edwards, J., Forsyth, D.A.: Who's in the Picture. In: Neural Information Processing Systems (NIPS). pp. 137–144. Vancouver, British Columbia, Canada (December 2004)
3. Calvert, P.: Parallelisation of Java for Graphics Processors. Ph.D. thesis, Trinity College (May 2010)
4. Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N.C., Hu, B.: Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In: 8th IEEE International Conference on Cloud Computing (CLOUD). pp. 621–628. New York City, New York, USA (June 2015)
5. Glikson, A., Nastic, S., Dustdar, S.: Deviceless edge computing: extending serverless computing to the edge of the network. In: 10th ACM International Systems and Storage Conference (SYSTOR). Haifa, Israel (May 2017)
6. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless Computation with OpenLambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). Denver, Colorado, USA (June 2016)
7. Hummer, W.: A fully functional local AWS cloud stack. online: `https://github.com/localstack/localstack` (July 2017)
8. Jonas, E., Venkataraman, S., Stoica, I., Recht, B.: Occupy the Cloud: Distributed Computing for the 99%. preprint at arχiv:1702.04024 (February 2017)
9. Mallya, S., Li, H.M.: Serverless Reference Architecture: MapReduce. online: `https://github.com/awslabs/lambda-refarch-mapreduce` (October 2016)
10. Müller, S.C., Alonso, G., Amara, A., Csillaghy, A.: Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 645–659. Broomfield, Colorado, USA (October 2014)
11. Rao, D., Ng, W.K.: Information Pricing: A Utility Based Pricing Mechanism. In: 14th IEEE International Conference on Dependable, Autonomic and Secure Computing. pp. 754–760. Auckland, New Zealand (August 2016)
12. Ricci, R., Eide, E.: Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. ;login: The Usenix Magazine 39(6) (December 2014)
13. Spillner, J.: Transformation of Python Applications into Function-as-a-Service Deployments. arχiv:1705.08169 (May 2017)
14. Spillner, J.: Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. arχiv:1703.07562 (March 2017)