FaaSET: A Jupyter Notebook to Streamline Every Facet of Serverless Development

Robert Cordingly
University of Washington
Tacoma, Washington, USA
rcording@uw.edu

Wes Lloyd
University of Washington
Tacoma, Washington, USA
wlloyd@uw.edu

ABSTRACT

Function-as-a-Service platforms require developers to use many different tools and services for function development, packaging, deployment, debugging, testing, orchestration of experiments, and analysis of results. Diverse toolchains are necessary due to the differences in how each platform is designed, the technologies they support, and the APIs they provide, leading to usability challenges for developers.

To combine support for all of the tasks and tools into a unified workspace, we created the FaaS Experiment Toolkit (FaaSET). At the core of FaaSET is a Jupyter notebook development environment that enables developers to write functions, deploy them across multiple platforms, invoke and test them, automate experiments, and perform data analysis all in a single environment.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

Jupyter, Function-as-a-Service, Serverless, Development, Profiling, Tools

ACM Reference Format:
Robert Cordingly and Wes Lloyd. 2022. FaaSET: A Jupyter Notebook to Streamline Every Facet of Serverless Development. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3491204.3527464

1 INTRODUCTION

To streamline the process of developing Function-as-a-Service (FaaS) applications, invoking and testing functions, executing experiments, training performance models, and processing results, we created the Function-as-a-Service Experiment Toolkit (FaaSET) notebook [13]. The FaaSET notebook supports many FaaS platforms including AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Azure Functions, and OpenFaaS [1, 6, 7, 10]. Platform agnostic functions are written inside a Jupyter notebook [8] as standard Python functions and then are automatically packaged, deployed, and can be invoked from inside the FaaSET notebook. FaaSET builds upon a strong foundation of tools designed specifically for FaaS development, deployment, and experimentation, such as the Serverless Application Analytics Framework (SAAF), FaaS Runner, and more, used in [3, 5, 11, 12].

Previously, function development required using various tools or integrated development environments (IDEs) to write and deploy functions. Deployment processes alone could require vastly different workflows depending on the platform. After deploying an application, custom tools and scripts are commonly used to run experiments on FaaS platforms. Data analysis requires another set of applications such as R, Excel, or Python notebooks to train models or aggregate results from experiments. This fragmentation of tools creates workflows that are locked into specific platforms or experiments, that are neither portable or extensible, and that are potentially slower than optimized tools. FaaSET addresses these challenges by providing a unified workspace that supports the full lifecycle of FaaS function development and analysis.

This paper describes the following contributions:

(1) We present the Function as a Service Experiment Toolkit (FaaSET), a Jupyter notebook and library of tools that aid in developing, deploying, and experimenting with serverless functions.
(2) We provide an example FaaSET notebook hosted on Google Colaboratory to quickly introduce users to the features and tools in FaaSET [13].
(3) We evaluate the performance implications of using different client infrastructures to host the FaaSET notebook and execute experiments on FaaS platforms.

2 RELATED WORK

Many tools exist to streamline the development process of serverless functions. For example, on AWS Lambda, simple functions can be written and tested using the Cloud9 IDE directly from the AWS website [2]. The Serverless Framework provides many tools to aid in FaaS application development, deployment, and monitoring [14]. Plugins for existing development environments, such as Visual Studio Code, support integration with FaaS platforms such as Azure Cloud Functions, AWS Lambda, and Google Cloud Functions [15].

While many of these tools focus on aiding development for a specific platform, others, such as Lithops, support developing functions across many clouds to support big data analytic workloads [9]. Although these tools offer features to solve specific problems, FaaSET aims to be a platform-neutral toolkit supporting the entire FaaS development lifecycle from deployment, to experimentation, to data processing and analysis.

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04.
https://doi.org/10.1145/3491204.3527464

3.2 Serverless Application Analytics Framework

To aid in performance profiling of serverless applications, functions in FaaSET can be deployed with the Serverless Application Analytics Framework (SAAF) [4]. SAAF collects metrics from multiple sources inside the Linux operating system, including the **/proc** filesystem, local files under **/tmp**, and environment variables created by the FaaS platform. SAAF's design allows all metrics to be collected by simply including the framework in the deployment package and adding a few lines of code to the beginning and end of the function's source code. Each commercial FaaS platform (e.g., AWS Lambda, IBM Cloud Functions) exposes or hides different metadata about the underlying Linux environments that run functions. SAAF is designed for FaaS platforms; it adds minimal overhead to functions and works around different levels of infrastructure obfuscation of each platform. FaaSET utilizes the publishing and deployment tools used by SAAF to support multiple FaaS platforms.

3.3 FaaS Runner

To automate complex experiments on FaaS platforms, we created the FaaS Runner tool. FaaS Runner provides a client-side application used in conjunction with SAAF and the FaaSET notebook. FaaS Runner automates FaaS experiments by using functions and experiment files that define how FaaS functions should be executed and how the results from SAAF should be processed. Experiment files define an experiment's configuration, including how functions are invoked (e.g. synchronously or asynchronously), the degree of concurrency (e.g. parallel with multiple threads or sequentially), how payloads should be distributed, and how multi-function pipelines should be orchestrated. In addition, FaaS Runner automatically applies changes to FaaS function configurations as prescribed in the experiment file.

FaaS Runner can be executed using the FaaSET notebook or run independently. Running experiments within the FaaSET notebook will automatically generate the necessary function and experiment files and import results into the notebook after the experiment completes.

3.4 Example FaaSET Notebook

The FaaSET notebook, SAAF, and FaaS Runner are all open source available to download on GitHub [13]. To make it easy to try out all of our tools, we have developed an example notebook hosted using Google Colaboratory. Using the notebook, it is only necessary to enter AWS credentials and a Role ARN to test deploying functions, running experiments, processing results, and other FaaSET features without configuring the environment. Google Colaboratory is free and provides a low barrier of entry for hosting Jupyter notebooks. The link to our example FaaSET notebook can be found on the front page of our GitHub [13].

4 METHODOLOGY

The FaaSET notebook can be hosted on many different platforms, including Google Colaboratory, locally, or on an IaaS cloud platform such as Amazon EC2. Two key considerations when picking a host client to invoke functions for FaaS experiments are the client's capability to invoke functions concurrently and the network latency between requests. We evaluated the performance capabilities of three different notebook hosts, including the infrastructure provided with the free tier of Google Colaboratory (2 vCPUs) (California), a local Ubuntu Server 20.04 virtual machine with an Intel i9-9990k Processor and 32 GB of RAM (10 vCPUs) (Seattle WA), and a c5.metal EC2 instance (96 vCPUs) located in the same region and subnet as our Lambda function (us-east-1f).

To rule out the performance variability of the function itself and focus on the performance of the client invoking the functions, we created the sleeper function shown in Figure 4. This function sleeps for a given amount of time. We created an experiment that will invoke the function a total of 5,000 times using 1,000 threads (5 times per thread). The maximum available concurrency of the function on AWS Lambda was also 1,000. SAAF reports the epoch start and end time of our function invocations, enabling the number of functions executing simultaneously to be determined. FaaS Runner reports the latency of individual function invocations by calculating the difference between the round trip time (time between when a request is made and a response is received) and the total function runtime returned by SAAF. This experiment utilized every *facet of FaaSET* enabling us to observe different host clients' maximum concurrency and latency.

```
import FaaSET

@FaaSET.cloud_function(platform="AWS",
                       config={"memory": 256})
def sleeper(request, context):
    from SAAF import Inspector
    import time
    inspector = Inspector()
    inspector.inspectAll()
    time.sleep(request['time'])
    inspector.inspectAllDeltas()
    return inspector.finish()

# Define and execute experiment
sleep_experiment = {
  "payloads": [{"time": 10}],
  "memorySettings": [128],
  "runs": 5000,
  "threads": 1000,
  "iterations": 1,
}
sleeper_results = run_experiment(function=sleeper,
     experiment=sleep_experiment)
```

Figure 4: Function and experiment used to evaluate latency and maximum concurrency of different clients.

5 RESULTS

Our experiment revealed that the infrastructure used to execute experiments impacted the performance of running experiments. Since our sleeper function was configured to execute for only 10 seconds, no client could achieve the maximum 1,000 concurrent function invocations. The c5.metal instance achieved the highest concurrency peaking near 450 function instances while averaging

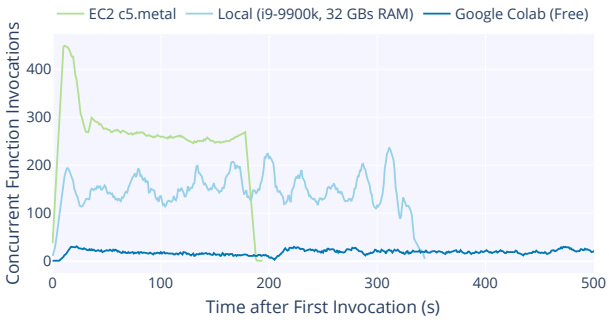

Figure 5: Number of concurrently running function instances using various clients. 5000 total invocations of 10 second sleep function using 1000 threads.

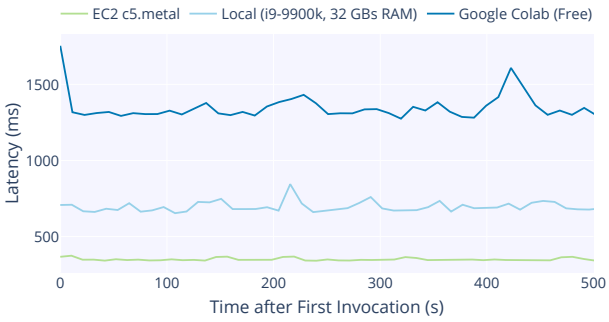

Figure 6: Request latency using a variety of clients when functions are invoked sequentially using 1 thread.

250. The local Ubuntu Server achieved concurrency between 150 to 230 instances. Finally, Google Colaboratory only achieved concurrency of at most 31 function instances. Figure 5 shows the number of function instances we were able to execute over 500 seconds.

One challenge to running large experiments using multiple threads is the memory required to maintain open connections and store data exchanged with the FaaS platform. We found that using 1,000 threads could use up to 11 GB of RAM on the c5.metal, enough to saturate all of the available memory on Google Colaboratory and crash the instance. To avoid this, we limited the maximum number of threads to 200. Unfortunately, Google Colaboratory's limited memory and CPU power with the free tier does not make it a viable option for running experiments that require a large number of concurrent function invocations.

Finally, we evaluated the round-trip latency of each client to our AWS Lambda function deployed in us-east-1f. Like the concurrency test, Google Colaboratory exhibited the worst performance with latency between 1.2 to 1.6 seconds. The local virtual machine showed 650 to 850 ms of latency, and the c5.metal instance had 340 to 350 ms. Figure 6 shows the latency observed throughout our experiment. If running an experiment where latency is an important metric, it is critical to have the host client as physically close to the function instances as possible. For AWS, the only way to do that is to use EC2 instances located in the same region and subnet as the function.

6 CONCLUSIONS

The FaaSET notebook provides a unified workspace where developers and practitioners can develop FaaS applications, perform testing, run experiments, process results, and analyze data without needing to use any other tools or websites. FaaSET is flexible in that it can be deployed to a variety of Juypter servers and configured to use different FaaS platforms to meet the demand of experiments. For developing, deploying, and testing functions and running simple experiments, FaaSET can be hosted using free Jupyter services such as Google Colaboratory. If an experiment requires high function concurrency or low latency, FaaSET can be hosted on powerful cloud virtual machines. Our evaluation found that Google Colaboratory as a free notebook host provided only limited scalability, supporting only 1/10 the maximum concurrency while exhibiting 5x the latency of a c5.metal EC2 instance. FaaSET strives to be a powerful tool to aid developers and practitioners to quickly and easily develop serverless applications and run experiments to understand how they perform on the cloud.

ACKNOWLEDGMENTS

This research is supported by the NSF Office of Advanced Cyberinfrastructure (OAC-1849970), NIH grant R01GM126019, and the AWS Cloud Credits for Research program.

REFERENCES

[1] AWS. 2021. AWS Lambda – Serverless Compute - Amazon Web Services. http://aws.amazon.com/lambda/.
[2] Cloud9. 2022. A cloud IDE for writing, running, and debugging code. https://aws.amazon.com/cloud9/
[3] Robert Cordingly, Wen Shu, and Wes J Lloyd. 2020. Predicting Performance and Cost of Serverless Computing Functions with SAAF. In *6th IEEE International Conference on Cloud and Big Data Computing (CBDCOM 2020)*.
[4] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 67–72.
[5] Robert Cordingly, Hanfei Yu, David Perez Varik Hoang, David Foster, Zohreh Sadeghi, Rashad Hatchett, and Wes J Lloyd. 2020. Implications of Programming Language Selection for Serverless Data Processing Pipelines. In *2020 6th IEEE International Conference on Cloud and Big Data Computing (CBDCOM 2020)*.
[6] Google Cloud. 2021. Google Cloud Function: Event-Driven Serverless Compute Platform. http://cloud.google.com/functions.
[7] IBM. 2021. IBM Cloud Functions. http://ibm.com/cloud/functions.
[8] Jupyter. 2022. https://jupyter.org
[9] Lithops. 2022. https://lithops-cloud.github.io
[10] Microsoft Azure. 2021. Azure Functions. http://azure.microsoft.com/en-us/services/functions/s.
[11] Sterling Quinn, Robert Cordingly, and Wes Lloyd. 2021. Implications of Alternative Serverless Application Control Flow Methods. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. 17–22.
[12] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems. *IEEE Transactions on Services Computing* (2021).
[13] SAAF. 2022. Serverless Application Analytics Framework. http://github.com/wlloyduw/SAAF
[14] Serverless Framework. 2022. https://www.serverless.com
[15] VS Code Extensions - Azure Functions. 2022. https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions