Icarus: a Testing-as-a-Service Utility for Serverless Functions

Thomas Tomtsis[1][0009-0000-2787-3520] and Kyriakos Kritikos[2][0000-0001-9633-1610]

[1] University of the Aegean, Samos, Greece
ttomtsis@protonmail.com
[2] University of the Aegean, Samos, Greece
kkritikos@aegean.gr

Abstract. Testing serverless functions differs significantly from that of conventional software products. Their distributed nature and event-driven architecture make test development inherently complex. Further, the development of functions and their tests often relies on using tools and software development kits managed by cloud providers, resulting in minimal control over the test execution environment and a lack of debugging tools.
As such, we have developed the Icarus RESTful, open-source Testing-as-a-Service tool with a transparent and predictable operation, supplying the user with full control over test creation and execution. Icarus does not burden the user with details of a function's deployment across different providers, does not require the user to write any line of testing code and relies solely on using familiar open-source and well-known tools to automatically compose and execute functional and non-functional tests. Its experimental evaluation shows that it scales well with the user and workload increase, rapidly delivering test results to the user.

Keywords: Serverless, AWS Lambda, Google Cloud Functions, Functional Testing, Performance Testing.

1 Introduction

Vendor lock-in is one of the biggest issues in Cloud Computing. The functional cost and the overall quality of an offered service is directly correlated with the business strategy of the Cloud Provider, thus aligning the Cloud Strategy of an organization with that of the Cloud Provider. While this offers an easy transition towards the Cloud, eventually this approach incurs sizable costs in the event of migration from one provider to another or in the event of investing in a private infrastructure. The complexity and cost of data migration and the cost of service and product transfer amongst providers are critical factors that usually impede Cloud adoption [1].

Serverless Computing is one of the latest developments in cloud computing, where the development and deployment of software foregoes the need of owning and managing IT infrastructure. It is based in the Function-as-a-Service (FaaS) model, where the foundational building block of applications are small, event-driven, single-responsibility pieces of code, named functions. In the FaaS model, function development must

utilize the provider's Software Development Kits (SDKs) and conform to architectural requirements specified by the provider. At the same time, operating costs and scalability vary between providers due to different closed-source implementations of the FaaS paradigm. To make matters even more complicated, functions commonly utilize several other vendor specific implementations of supporting services, such as Message Queues or Serverless Databases, and may rely on proprietary optimization technologies (e.g. AWS Lambda Snap Start) or supporting solutions (e.g. Azure Durable Functions).

As such, it is evident that choosing a suitable Cloud Provider is paramount for a user or organization. This process is often complex and requires manually implementing tests to judge a provider's adequacy. These tests must be executed in the provider's infrastructure, thus incurring extra costs, or be executed in an environment that closely simulates it. The tests must be developed using the relevant vendor specific SDK's or tools. Simulating a vendor's infrastructure is challenging, since realistic scenarios are hard to recreate. At the same time, function development and deployment across different vendors requires working knowledge of each vendor's toolchain and services.

This paper addresses these shortcomings by developing a simple and easy to use RESTful service. This service, which is aptly named Icarus, simplifies the process of developing and executing automated tests across vendors and simultaneously abstracts the complexity of the function's lifecycle management. Users solely provide the location of the function's source code and the test configuration, while Icarus manages the entire lifecycle of the functions, creates and executes the tests, and produces a test report document. Our service utilizes the black box testing technique and supports the automated execution of both functional and performance tests across different cloud providers and respective function implementations. It has been developed as a Software-as-a-Service, thus realizing the vision of Testing-as-a-Service (TaaS) [2], and is available as an open-source software project on GitHub[1]. By utilizing our service, functional and non-functional issues may be detected early in the software development lifecycle without requiring specialized knowledge of vendor specific tools and services.

Compared to the related work, Icarus not only supports both testing kinds but also does not require the user to write any single line of testing code. Further, it parallelizes the execution for both functional and non-functional tests. In addition, the testing report supplies significant new knowledge, especially in the context of performance testing, as it incorporates a statistical performance model for the tested function, one per each cloud provider. This model incorporates all factors influencing function performance, including workload, resources (e.g., main memory size) and location. As such, it is more complete than other similarly constructed models within the literature. This new knowledge is of paramount importance not only for better understanding a function's performance features across different providers and configurations, but also for facilitating the user in the selection of the most suitable provider according to his/her needs.

The remaining of this paper is structured as follows. Section 2 reviews related work. Section 3 analyses the design and implementation of Icarus TaaS. Section 4 describes the way Icarus was experimentally evaluated and showcases the respective results. Finally, the last section concludes the paper and draws directions for further research.

[1] https://github.com/ttomtsis/icarus

2 Related Work

Testing serverless applications has been the subject of research during the last few years, mainly in the context of function performance testing and benchmarking. A stellar example of this has been the benchmarking suite developed by Copik et al. [3] which introduces a provider-agnostic FaaS platform model and a set of metrics for analyzing both function cost and performance. Please note that the work of Copik et al. [3] is among the few to use a characteristic set of serverless applications to evaluate their suite and does not simply rely on microbenchmarks, which are quite common throughout the literature but are not so useful and practical as they rely on very small functions.

Similarly, Maissen et al. [4] offer a benchmarking suite for four well-known FaaS platforms (e.g., AWS Lambda and Google Cloud Functions) that supports specific runtimes and relies extensively on Docker containers to manage the functions lifecycle. Apart from producing performance models, the suite includes a billing costs calculator.

Somu et al. [5]support function chaining (apart from single functions) and multiple function triggers for python3 functions and rely on JMeter to conduct load tests.

Malawski et al. [6] developed a benchmarking tool, able to execute well known benchmarks like Linmark, that relies on the Serverless Framework to manage the function lifecycle.

To the best of our knowledge, Icarus is a complete testing utility for serverless applications, enabling automated functional testing of serverless functions, a feature not exhibited so far amongst the already developed tools. Further, Icarus can perform a wide variety of non-functional testing, not being limited to load tests, supports all available function runtimes in AWS Lambda and Google Cloud Functions, and utilizes the novel Terraform CDK for Java to automate the process of deployment, thus not relying on complex container techniques or cloud provider CLI's. In addition, Icarus minimizes the user effort in configuring the function testing as there is a single configuration to supply for testing the function across multiple providers and regions in contrast to the other tools. Finally, Icarus can produce more precise statistical functional performance models by considering all factors that influence function performance, including function location, memory layout and in the case of Google Cloud Functions vCPU cores.

3 System Development

Icarus was developed using the waterfall model as the project's requirements were well-defined and improbable to change, at least during its development. As such, the project's lifecycle included four distinct phases: Requirements engineering, design, implementation, and validation/testing.

The project requirements were carefully produced and devised by considering the current gaps in the literature, the actual needs of function developers, especially in terms of testing, and the current best practices in SaaS development.

3.1 Functional Requirements

The functional requirements are summarized as follows:

- Provide automated support for both functional and non-functional testing guided by user-supplied testing configuration
- The user functional testing configuration must supply a suitable test case description to automatically produce and execute the right tests and as well as generate the functional testing report.
  - Simply speaking, each test case must include the function input to supply as well as the desired output and HTTP status code.
  - As such, each test case is executed by sending requests over the function based on the designated output and checking whether the produced output and received HTTP status code are the expected ones. This checking is incorporated in the respective functional testing report.
- As part of the test execution process, the target function's lifecycle would be managed by the application, abstracting the complexity away from the user.
- The user non-functional testing configuration includes the test parameters, the function's resource configurations and the metrics to be collected per each provider.
  - Test parameters must be specified by summing all user-created load profiles. Every load profile includes the number of concurrent users, the users think time, the ramp up, initial start delay of the profile and total load time. By chaining load profiles, we enable the execution of a wide range of performance tests.
  - A resource configuration includes the function's memory configuration, the deployed region and in case of Google Cloud Functions the number of vCPU cores.
- During non-functional testing for a specific provider, the system should deploy the function according to its resource configuration and then execute all its load profiles
  - Each load profile is executed multiple times and then the measurements being produced must be averaged.
  - All average measurements across all resource configurations must then be fed into a statistical tool to produce via linear regression the function's performance model for the current cloud provider. This model should be stored in the non-functional report along with all these average measurements associated to their load profile and resource configuration.
- User (profile) management functions, including user registration and authentication, must be implemented to enable the system to be used only by registered users.
  - Each user profile should be associated with a set of accounts that the user has on cloud providers he/she intends to exploit. These accounts are to be used by the system to conduct function deployments on behalf of the user.

3.2 Non-Functional Requirements

The non-functional requirements set are summarized as follows:

- The system must support multiple concurrent users (at least 50) and respective requests

- The system should be performant (response time must be less than 5ms per request) and scalable
- The system should offer a high security level
  - It must support HTTPS
  - It must support basic user authentication with salting-based password hashing
  - It must support OAuth2 user authentication (with Auth0 as the default authentication provider) for those users that require using external authentication services
  - Access to services should be restricted to authenticated users
  - Each authenticated user should be able to solely see his/her own data
- The system must be developed in form of a RESTful API with high REST maturity
  - Testing must be performed asynchronously as it can take a long time to execute depending on the testing configuration length and the function execution time
- The system should operate across platforms by exploiting container technology
- The system should use an IaC (Infrastructure-as-Code) tool such that it can support function deployment across multiple serverless platforms
- The system must support at least the AWS Lambda and Google Cloud Function platforms. By supporting these two platforms we aim to compare the already intensely studied and industrially significant AWS Lambda with the less studied and gaining in popularity Google Cloud Functions
  - Icarus must support the novel Google Cloud Functions V2, which allows the manual configuration of vCPU cores. This can lead to the production of more complete statistical function performance models as we take into consideration another resource factor apart from the main memory size.

3.3 Application Design

During the design stage, we followed a top-down approach to the system design, by modelling suitable UML diagrams.

Context Diagram

Initially, we designed a context diagram (see Fig. 1) to set the boundaries of our application and define the external systems with which it will interoperate. Icarus would support two user types, visitors and authenticated users. Auth0 would be used as the default OAuth2 authentication provider. GitHub would be used when a user specifies a public GitHub repository as the source of the function's source code. This is a convenience feature, aiming to ease integrating Icarus into a project's development lifecycle. Google Cloud Platform and Amazon Web Services would be used to deploy the functions and their supporting infrastructure and destroy it upon completion of the test.

Component Diagram

Next, we designed our RESTful API's component diagram, shown in Figure 2, to designate the API's internal architecture. Then, other diagrams followed like use-case diagrams and process diagrams. Due to the imposed size restrictions, we do not have

the space to show all these diagrams. Thus, we stick to the component diagram to explain which are the API's main components, what is their intended functionality and how they interact with each other in which cases. Finally, we provide some implementation details, including the URL of the Icarus public repository on GitHub.

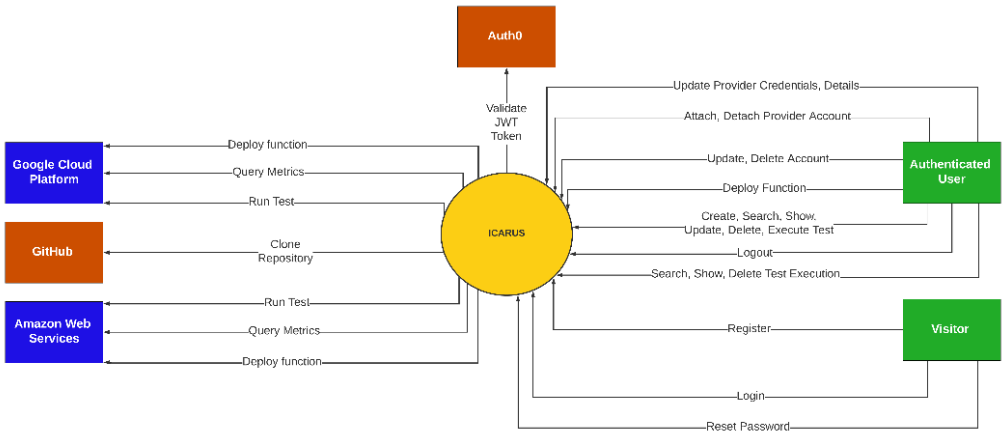

Fig. 1. Icarus Context Diagram

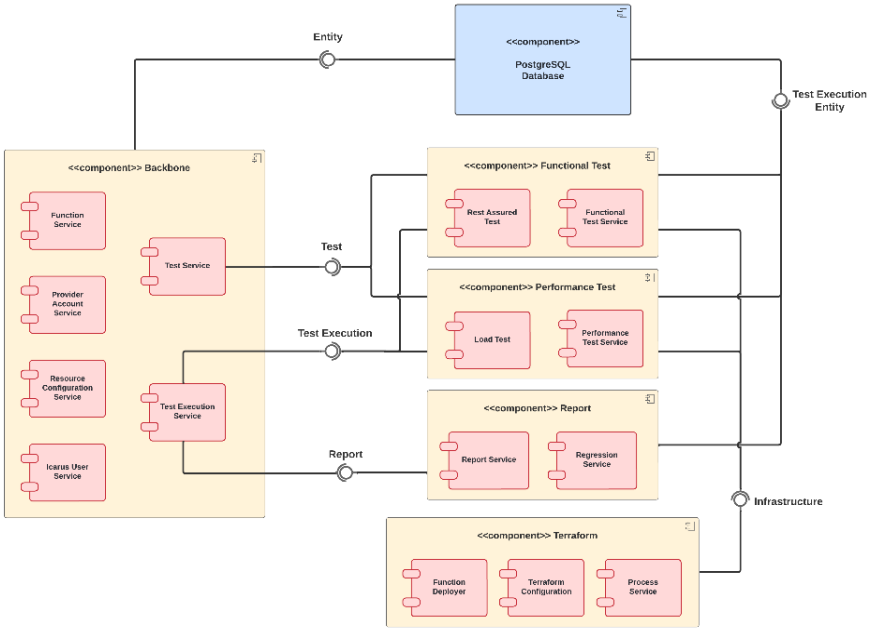

Fig. 2. Icarus Component Diagram

Icarus has been structured in several components to meet the project's requirements.

The *Backbone super-component:* Encapsulates basic components required for test creation and execution. Prior to a test's creation and execution, Icarus requires a linked AWS or GCP account (which will be used to execute the tests and deploy the function), a description of the function to be tested (containing details like runtime, source code location etc.), and a resource configuration that contains details about the function's deployment. Each of these requirements is handled by a respective sub-component. *Function Service* manages the Function entities, *Provider Account Service* manages the user's cloud provider accounts, *Resource Configuration Service* manages the function configurations (memory layouts, regions, vCPU cores) on the cloud providers, *User Service* manages the application's users, *Test Service* manages test functionality common for both functional and performance tests (such as checking for the existence of the target function's source code prior to deploying), and *Test Execution Service* offers basic support functionality during test execution, such as report production and deletion of infrastructure after the tests have been completed.

PostgreSQL Database: Icarus uses a (PostgreSQL) database to store the required entities and user data.

Functional Test super-component: The *Functional Test* super-component executes the functional tests. It mainly encompasses the *Functional Test Service* that handles the smooth creation and execution of each Rest Assured functional test as prescribed by the user configuration by using the *RestAssured Test* sub-component. Functional tests contain test cases and each test case contains test case members. Icarus creates a RestAssured test per each test case member associated with the functional test. RestAssured tests' creation and execution is done concurrently to improve performance.

Performance Test super-component: The *Performance Test* super-component is responsible for performance test execution. Similarly to the *Functional Test Service* super-component, the *Performance Test Service* sub-component handles the creation and execution of each performance test as prescribed by the user configuration by using the *Load Test* sub-component, which is responsible for the creation and configuration of a JMeter load test. When executing a performance test, Icarus refers to the resource configurations that have been associated with the test and deploys a number of different configurations for the function accordingly. The deployment of the different configurations is being handled by Terraform and as such Icarus has little effect on the performance of the deployment process. After deployment, Icarus creates a load test for every resource version of the deployed function and JMeter handles its execution.

Report super-component: The *Report* super-component manages the successful creation of a report document, containing the respective test execution's results. This is accomplished via subcomponents *Report Service* and *Regression Service*. In case of a performance test, *Regression Service* performs a linear regression analysis on the test results and adds the constructed performance model to the report. *Report Service* manages the creation and storage of a pdf report document by using Eclipse BiRT.

Terraform super-component: The *Terraform* super-component is used to manage the lifecycle of the functions, from creation to deletion. This is accomplished through its subcomponents, *Function Deployer*, *Terraform Configuration* and *Process Service*.

The *Function Deployer*'s purpose is twofold, it manages the creation of Terraform configuration files through the Terraform CDK for Java and interfaces with the local Terraform binary to deploy and delete the serverless functions and supporting infrastructure. The *Process Service* is a supporting component used by *Function Deployer* to execute Terraform commands in the local system using the host's Terraform binary. As its name suggests, *Terraform Configuration* is used to configure Terraform.

3.4 Icarus Implementation

The implementation of the Icarus service as an open-source Maven project relied on the Java 21 programming language and Spring Boot v3.1.5. framework. As such Icarus functions as a RESTful web service utilizing Tomcat and listening for requests in port 8080. PostgreSQL v15.5 was used as the underlying database. To manage the lifecycle of the functions, Terraform v1.6.5 and the Terraform CDK for Java v0.19.0 were used. For automated performance testing, we chose the well-known JMeter v5.6.2 tool whereas for automated functional testing we relied on RestAssured v5.3.2. We utilized Eclipse's BiRT v4.8.0 for report generation.

4 Experimental Evaluation

The evaluation process focused on checking the system response time and its behavior when executing functional tests with varying workload. The function tested took the form of a pre-deployed 'hello world' type function deployed in AWS Lambda. The target function was warmed up before executing the test plan, enabling to exclude cold start delays incurred by AWS Lambda from our test results. In addition, the AWS region where the function was deployed had a sufficient instance number (1000 instances) to not incur throttling. Icarus and its PostgreSQL database were deployed in the local system. Further, user requests were authenticated by Icarus using Basic Authentication. Following this setup ensured that we were studying the internal performance characteristics of Icarus more closely and excluded any other moving parts and external systems.

Icarus was evaluated in a laptop (local system) with 8GB RAM and an Intel Core i5-1135G7 processor running Windows 11. We used the JMeter load testing tool with the Measure-Command PowerShell utility. JMeter executed a test plan in headless mode and the Measure-Command utility measured the execution time of the command.

We focused on two distinct scenarios as part of the evaluation process. The first scenario studied the application's performance under a varying number of concurrent users. Every user executes a functional test containing a single test case that checks if the function's response matches the string 'Hello World'. For every configuration of the first scenario (exact number of concurrent users), we executed the test thirty times and removed the two biggest and smallest execution times. Afterwards we divided the result by 26 and successfully calculated the average response time experienced across all users for that configuration of the scenario. We studied this scenario starting with ten concurrent users and scaling upwards to fifty concurrent users, in steps of ten.

The second scenario studied the application's performance under a steady number of thirty concurrent users with varying workload configurations. Each workload configuration consists of a steady number of test cases. Same as before, for every workload configuration the scenario was executed thirty times. We removed the two biggest and smallest execution times and divided the result by 26 to calculate the average response time per user experienced across all users for that workload configuration of the scenario. We studied this scenario starting with ten identical test cases and scaling upwards to forty test cases per user, in steps of ten.

To account for JMeter's initialization times, we conducted further experiments to measure the average initialization time of JMeter and thus redact it from the calculated average execution times. We created an empty test plan containing a thread group with zero loops and used Measure-Command to study JMeter's behavior. After conducting this experiment thirty times and calculating the average initialization time, we redacted it from the measurements we had obtained from both scenarios.

Below, we showcase the two experiment results in two graphs. Scenario A represents the first scenario, where we study the effects of the increasing concurrent user number, whereas scenario B represents the second scenario where we study the effects of thirty concurrent users with varying workloads. By studying the graphs, we can clearly see the linear increase in response time in both scenarios. This indicates that Icarus scales well with the increase of its workload.

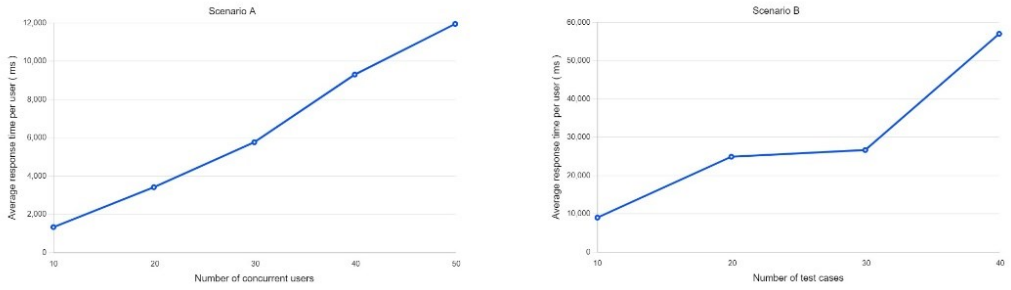

Fig. 3. Performance Test results for Scenarios A (left) and B (right)

5 Conclusions

In this paper, we presented the Icarus TaaS, a complete testing service that touches on a subject often ignored in related FaaS tools, the functional testing of serverless functions. Further, Icarus covers the non-functional function testing by also having the ability to produce more complete function performance models than those of the existing FaaS tools, for both the intensely studied AWS Lambda and the under-researched Google Cloud Functions, through incorporating resource features, such as vCPU cores and main memory size, as well as the function (deployment) location.

We aim to further expand the base functionality of Icarus by supporting multiple trigger types, complex function chains and serverless applications, thus covering a wider spectrum of serverless applications and use cases. In addition, we plan to support automated test case data generation as currently such data are given by the user in the test configuration. All these new features will provide more test automation and thus greatly reduce the test effort of the end users who will then be able to focus mainly on the core development task of function / serverless application implementation.

References

1. Opara-Martins, J., Sahandi, R., Tian, F.: Critical review of vendor lock-in and its impact on adoption of cloud computing. In: International Conference on Information Society (i-Society 2014). pp. 92–97. IEEE, London, United Kingdom (Nov 2014).
2. Gao, J., Xiaoying Bai, Wei-Tek Tsai, Uehara, T.: Testing as a Service (TaaS) on Clouds. In: 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering, pp. 212-223. IEEE, San Francisco, CA, USA (2013).
3. Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., Hoefler, T.: Sebs: A serverless benchmark suite for function-as-a-service computing. In: Proceedings of the 22nd International Middleware Conference, pp. 64-78. ACM, Quebec City, Canada (2021).
4. Maissen, P., Felber, P., Kropf, P., Schiavoni, V.: FaaSdom: a benchmark suite for serverless computing. In: Proceedings of the 14th ACM international conference on distributed and event-based systems, pp. 73-84. ACM, Montreal, Quebec, Canada (2020).
5. Somu, N., Daw, N., Bellur, U., Kulkarni, P.: Panopticon: A comprehensive benchmarking tool for serverless applications. In: 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS), pp. 144-151. IEEE, Bengaluru, India (2020).
6. Malawski, M., Figiela, K., Gajek, A., Zima, A.: Benchmarking heterogeneous cloud functions. In: In: Heras, D.B., Bougé, L., Mencagli, G., Jeannot, E., Sakellariou, R., Badia, R.M., Barbosa, J.G., Ricci, L., Scott, S.L., Lankes, S., Weidendorfer, J. (eds.) Euro-Par 2017: Parallel Processing Workshops, vol. 10659, pp. 415–426. Springer International Publishing, Cham (2018).