ANITA B.ORG

Not Just *functions*: a 360° Guide to Serverless and FaaS

Vasuki Ashok
Oracle Corporation
vasuki.ashok@oracle.com

L. Ümit Yalçinalp
Oracle Corporation
umit.yalcinalp@oracle.com

ABSTRACT

This paper presents a comprehensive guide to Serverless and FaaS concepts in cloud computing and the industry best practices. It contrasts FaaS with PaaS and lists down a set of applicable use cases with concrete examples. The challenges faced by cloud developers and vendors security implications, the paper concludes with the pros and cons and anticipated improvements.

AUDIENCE

This paper is intended to be a guide for a beginner/intermediate technical audience willing to understand what FaaS (Functions as a Service) is about, where to use it, and the best practices and considerations that have emerged in the industry as technology matures.

INTRODUCTION

FaaS and Serverless architecture has emerged (Han 2017) as a key component of cloud offerings in the industry within the last decade. Today, multiple cloud vendors including (AWS), (Google), (OracleFn), and (Microsoft) offer such capabilities enabling developers to utilize their cloud platforms.

The underlying concept of serverless architecture is the notion that a service developer doesn't manage any server or processes on the cloud. The Life Cycle Management (LCM) of the application's resource requirements such as,

- Provisioning of servers
- Auto scaling of nodes (adopting the number of servers based on load)

are completely the responsibility of the cloud vendor.

Since the developer does not manage the resources, he/she focuses solely on the business logic as if all resources in the data center are available on demand. The cloud vendor ensures that the required resources are made available to the service. In this model, while the *Business Functions* are considered to be provided by the application developer, and the service itself is made available to consumers by the cloud vendors, on an on-demand basis.

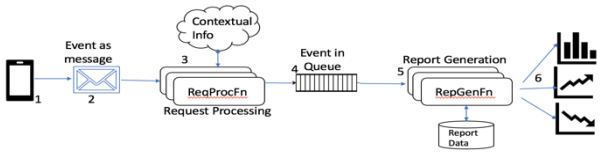
On the surface, this may look very similar to the goal of PaaS (Platform as a Service) architecture, which manages similar responsibilities. PaaS services provide *end to end business functionality* with built-in security access control and authorization models. A PaaS service typically focuses on container based deployment with *multiple entry points* that collaborate with other services to provide end to end functionality. A typical PaaS services may experience significantly higher start up time, as the functionality supported is coarse granular requiring multiple interactions. In addition, a PaaS service serves multiple requests and addresses different concerns (such as an account management service). In contrast, FaaS services address fine granular computational requirements (such as computing an international shipping cost of goods) or triggering of some long running background jobs. As functions can be *reused in multiple apps*, it is possible to stitch several FaaS functions to build a PaaS service. In a composite, each such function incurs a start time (less than 20 ms) and introduces a network latency thus might result in a longer execution time, if not designed correctly. Such composite services may be hard to diagnose and debug.

ARCHITECTURE & USE CASES

FaaS architecture is based on the fact that every business functionality can be divided into multiple *micro workflows* (functions) which can further be divided into a sequence of event triggered actions. The result of an action may be another event that can trigger a domino of actions leading to the completion of the business functionality.

Let's take an example of a business function, where a customer can request an analytics report from a mobile App. This requires two different functions (*ReqProcFn and RepGenFn)*, to work in sequence as shown below:

1. The customer selects the parameters that determine the required report and clicks the *GenerateReport* button.



Mobile request processing generating Analytics reports

2. The button click event generates an HTTP request to the *ReqProc fn*.
3. If the FaaS function server is already running, *ReqProcFn* reads the request, gets the required contextual information from its context repository, processes the request and generates the output and adds it to the queue.
4. The addition of an entry in the queue generates the next event which initiates a request to the *RepGenFn*.
5. *RepGenFn* interprets the request and reads required data from its repository and generates the report.
6. The generated report is sent to the target as requested in step 1. Depending on the target, zero or more functions may be invoked.

Some of the other use cases of serverless functions are:

- Static Web Page serving, eliminating the need for VM hosting for websites (such as loading of web pages only when required- ex: catalogs; providing JIT webservices)
- Mobile Backend Apps (User action triggers backend processing; Backend simplification without hosting)
- IoT Data Processing (making IoT processing only on event boundaries; Scaling IoT sensor data recording; Consolidating IoT management in the cloud)
- Streaming Data Analysis (Real time analytics based on events; On Demand Reporting; Flexibility with Analytics a Service)
- Cognitive Processing (Image processing; Identity Validation with Automation; Decentralization.

DEVELOPER PERSPECTIVE

A developer implementing the business logic addressing any of the use cases similar to the ones above is not constrained by the resource requirements. However, they are limited by other constraints unique to Serverless (AWSDev) development. These constraints include:

1. Statelessness: Serverless components are stateless. Since these functions are hosted as required, they do not have any persistent file system. Hence, any state context required must be persisted in a resource such as a database, network store or network cache which is outside the function.
2. ColdStart Delay: The services might show significant delay to serve the first request. Depending on the vendor it constrains the developer's choice of language and the state repository to optimize the startup time.
3. Runtime Limit: The execution time is bound by an upper limit enforced by the cloud vendor. Different vendors have different upper limits constraining the duration for which the function can run. This might limit the target functionality to be deployable to a specific cloud vendor. For instance, AWS allows lambda functions to run for a max of 15 minutes, whereas Azure limits it to 10 minutes.
4. Network Bandwidth: A fine granular function may be subject to frequent network calls. If the vendor billing is based on the network bandwidth consumption, hosting a fine granular function with FaaS might end up costing more than having a dedicated server.

CLOUD PROVIDER CHALLENGES

A cloud vendor providing FaaS needs to effectively share its resources among multiple customers who may not have mutually exclusive resource requirements. In addition to scheduling the resources, deploying the functions on demand in environments with proper network isolation, a cloud vendor also needs to support the following:

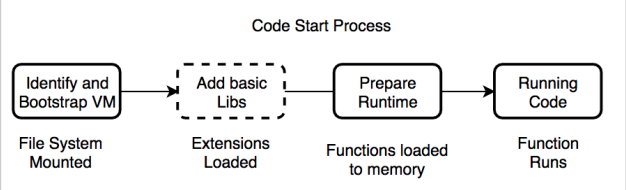
- Function Development Tooling (Tools for developers to develop and deploy their solutions - IDE).
- Multiple Programming Language support (Java/Python/PHP/Go are some of the typical runtimes needed for developed services)
- Environment Support (K8S, Docker, Windows, VMs, Linux and others)
- Run time support for automatic scale-in/scale-out, handling of rogue services
- Tooling for monitoring and managing of the resources
- Optimized of cold start costs (analytics on usage behavior, ML techniques for predictability)
- Authentication and isolation of resources
- Providing shared vs dedicated resources, based on usage patterns
- Financial viability (Billing, usage metering, etc.)

GENERAL CHALLENGES

In addition to solving individual challenges, both developers and cloud vendors need to work together to address the following issues

Cold Start/Request Processing Cycle

One of the constraints in meeting zero-time scalability is called *cold start*. It is defined as the time taken by the service provider to bring the function code to a running state to serve the requests. A cold start happens when the first request arrives for a Serverless function after it went idle or when new nodes are started up due to the increasing load. The Cold Start phenomenon is depicted below:



As seen in the diagram, first the VM to run the code is identified and bootstrapped as part of which it is attached to a file system. Depending on the requirement, additional libraries and extensions might have to be added to this node with the required language specific runtime. In case of container-based services, these extensions may not be required. Once the environment is ready, the function binary is loaded from a registry to the environment. In case of container-based functions, the container is loaded from the registry and the service/container is started. At this point the service is ready to serve the requests.

The frequency of invocation is an indicator of the cold startup cost (Roberts 2018). While a very frequently used function may not experience any cold start delay, a rarely used function will always experience longer startup time as it needs to be loaded from the registry.

Cloud vendors can address these issues by providing options to customers either to keep the functions warm always or based on well-known schedule usage patterns determined by machine learning algorithms.

Security Implications

As serverless functions become more popular, it also increases the risk as developers may not be aware of the dangers and do not incorporate security considerations into their design (Osborne 2018). Most important aspect is not paying attention to end to end security requirements for runtime execution of functions, such as

- Running functions with overprivileged roles to cater to multiple use cases, or
- Forgetting to secure the artifacts themselves in deployed files such as secrets, db passwords, or
- Not considering the overall context of end to end functionality with many functions stitched together
- Using insecure 3rd party dependencies which might expose the whole system to DoS attacks.

Thus, the overall system composition becomes crucial for secure execution of FaaS functions. A cloud vendor cognizant of these issues can address these requirements i.e. need for secret store, policy store and secure library registry etc. as part of the infrastructure. This area will continue to be the focus of cloud vendors and Application architects developing FaaS based services.

OUTCOMES/CONCLUSION

FaaS has very desirable properties for backend and event-based processing for developers *as it enables composition & reuse without LCM costs*. However, any system is as good as its weakest link, and FaaS is no exception: it deceptively hides the inherent issues of a cloud environment that must be optimized for resource allocation, sharing, and scalability. Successful deployment with FaaS requires developers to not only focus on their app logic but also the startup time, the event distribution rate (to minimize the burden of cold starts) and security implications in a complex system, when many functions are chained together. These are well beyond the lure of just focusing on application logic.

The future for FaaS is heading to publication of functions that enable composition in vertical segments as well as helping developers with different cloud environments, by building more abstractions. This helps in the deployment/ runtime support for complex compositions, especially in mitigating the security risks. More work needs to be done for end to end tracing and debugging. Optimizing startup times based on usage patterns, pre-defined schedules and/or subscription plans are viable options for cloud vendors. One toolkit is opensource (OracleFn), that we work on at Oracle with the community.

REFERENCES

(Han2017)https://medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2
(OracleFn,2017)https://developer.oracle.com/opensource/serverless-with-fn-project see also https://github.com/fnproject/flow
(Roberts2018)
https://www.martinfowler.com/articles/serverless.html
(AWS) https://aws.amazon.com/lambda/resources/
(Osborne2018) https://www.zdnet.com/article/the-top-10-risks-for-apps-on-serverless-architectures/
(Google)https://Cloud.google.com/serverless
(Microsoft)https://azure.microsoft.com/en-us/services/functions/
(AWSDev)https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf

PARTICIPATION STATEMENT

Both authors will participate if the submission is accepted.

BIO

Vasuki Ashok is an architect at Oracle in the Cloud Security and Identity organization providing security solutions for Oracle Cloud Services. She has been awarded 4 patents for her work in Oracle Cloud Security and Identity. She has experience in working on different aspects of Identity Management. She started her computing career with the E&RNET project, which established the first educational network that interconnected all the major educational and research institutions in India. She has over 20 years of experience in the computing industry after her Ph.D in Computer Science in the from Indian Institute of Technology, Madras.

Ümit Yalcinalp is an architect at Oracle who focuses on identity management for SaaS applications, in particular LCM and modernization of enterprise and SaaS deployments. Prior to Oracle, she was an architect for Creative Cloud at Adobe focusing on identity management, a visiting professor at Mills College, a researcher at SAP Labs and author and contributor to many standards papers, books, patents in Cloud computing, Java, XML and Web Services. She frequently presents at GHC and founder of Turkish women in computing community at ABI. She has over 25 years of experience in the industry after her Ph.D. in Computer Science from Case Western University.