**SERVERLESS IOT DATA PROCESSING** **PHASE 2 PROJECT**

**Abstract**

Serverless computing has gained importance over the last decade as an exciting new field, owing to its large influence in reducing costs, decreasing latency, improving scalability, and eliminating server-side management, to name a few. However, to date there is a lack of in-depth survey that would help developers and researchers better understand the significance of serverless computing in different contexts. Thus, it is essential to present research evidence that has been published in this area. In this systematic survey, 275 research papers that examined serverless computing from well-known literature databases were extensively reviewed to extract useful data. Then, the obtained data were analysed to answer several research questions regarding state-of-the-art contributions of serverless computing, its concepts, its platforms, its usage, etc. We moreover discuss the challenges that serverless computing faces nowadays and how future research could enable its implementation and usage.

**Introduction**

Serverless computing is also known as Function as a Service (FaaS) and represents an emerging category of cloud computing, which allows developers to implement and deploy business applications as a composition of stateless functions. It does not represent the absence of physical servers, as they continue to exist . In turn, for developers, it means they do not need to worry about server management . The main benefit is that developers do not need to manage underlying services and operating systems, as this is responsibility of the FaaS platform that also provides transparent scalability mechanisms. Platforms automatically scale to zero when there is no function running, thus avoiding waste of resources . In turn, Internet of Things (IoT) comes as another trending concept that is closely related to ubiquitous computing, which allows physical objects (things) to “talk” to each other while exchanging information through the Internet . Although originally proposed for the cloud, serverless computing can be a great ally for IoT as FaaS platforms can be placed physically closer to sensors and actuators, on edge and fog layers, in order to run functions with lower latency. This is specially important for mission-critical scenarios that cannot rely on Internet connection. For other scenarios, when intermittent connection is not a problem, functions can also be executed on more than a single layer and benefit from traditional cloud serverless platforms, that provide the illusion of infinite resources for processing huge workloads of IoT data produced by sensors. Our main contributions with this Systematic Literature Review (SLR) are as follows:

* + We propose a taxonomy covering the main characteristics about the 60 selected papers
  + We provide an update discussion about architecture, programming languages, and protocols of serverless applications for IoT, along with the main components employed to incubate and execute serverless functions;
  + We present a real-world use case, novel challenges, and research opportunities in the joint-combination of IoT and serverless, also highlighting technologies that appear as promising for the next years.

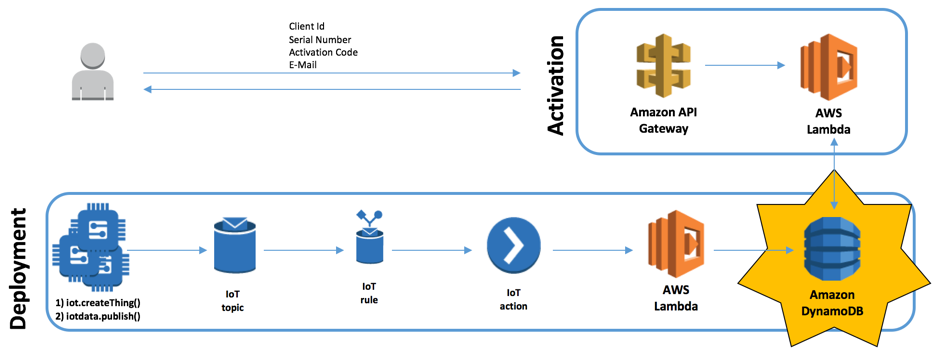


Fig: Serverless AWS IoT Backend with AWS Lambda

**New and upcoming serverless platforms**

There are several serverless projects ranging from open-source projects to vendors that find serverless a natural fit for their business. OpenLambda is an open-source serverless computing platform. The source code is available in GitHub under an Apache License. The OpenLambda paper outlines a number of challenges around performance such as supporting faster function startup time for heterogenous language runtimes and across a load balanced pool of servers, deployment of large amounts of code, supporting stateful interactions on top of stateless functions, using serverless functions with databases and data aggregators, legacy decomposition, and cost debugging.

Some serverless systems are created by companies that see the need for serverless computing in the environments they operate. For example, Galactic Fog added serverless computing to their Gestalt Framework running on top of Mesos D/C. The source code is available under an Apache 2 license. Auth0 has created webtasks that execute serverless functions to support webhook endpoints used in complex security scenarios. This code also available as open source. Recently they announced Project Kratos that allows developers to convert AWS Lambda functions into Docker images, and is available under an Apache 2 license. Additionally they are working with Cloud Foundry to bring multi-cloud serverless support to Cloud Foundry users . LeverOS is an open source project that uses an RPC model to communicate between services. Computing resources in LeverOS can be tagged so repeated function invocations can be targeted to a specific container to optimize runtime performance, such as taking advantage of warm caches in a container.

**Benefits and drawbacks**

Compared to IaaS platforms, serverless architectures offer different tradeoffs in terms of control, cost, and flexibility. In particular, they force application developers to carefully think about the cost of their code when modularizing their applications, rather than latency, scalability, and elasticity, which is where significant development effort has traditionally been spent.

The serverless paradigm has advantages for both consumers and providers. From the consumer perspective, a cloud developer no longer needs to provision and manage servers, VMs, or containers as the basic computational building block for offering distributed services. Instead the focus is on the business logic, by defining a set of functions whose composition enables the desired application behaviour. The stateless programming model gives the provider more control over the software stack, allowing them to, among other things, more transparently deliver security patches and optimize the platform.

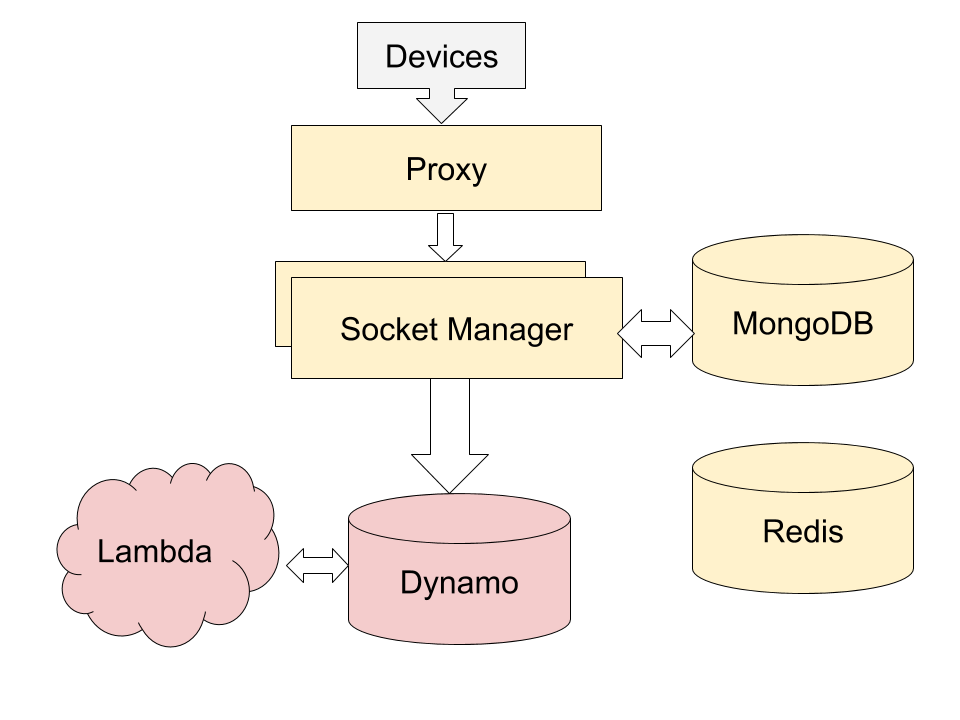
There are, however, drawbacks to both consumers and providers. For consumers, the FaaS model offered by the platform may be too constraining for some applications. For example, the platform may not support the latest Python version, or certain libraries may not be available. For the provider, there is now a need to manage issues such as the lifecycle of the user’s functions, scalability, and fault tolerance Serverless Computing: Current Trends in an application-agnostic manner. This also means that developers have to carefully understand how the platform behaves and design the application around these capabilities. One property of serverless platforms that may not be evident at the outset is that the provider tends to offer an ecosystem of services that augment the user’s functions. For example, there may be services to manage state, record and monitor logs, send alerts, trigger events, or perform authentication and authorization. Such rich ecosystems can be attractive to developers, and present another revenue opportunity for the cloud provider. However, the use of such services brings with it a dependence on the provider’s ecosystem, and a risk of vendor lock-in. 

Fig: IoT framework with AWS Serverless Lambda and Dynamo

**Current state of serverless platforms**

There are many commonalities between serverless platforms. They share similar pricing, deployment, and programming models. The main difference among them is the cloud ecosystem: current serverless platforms only make it easy to use the services in their own ecosystem and the choice of platform will likely force developers to use the services native to that platform. That may be changing as open-source solutions may work well across multiple cloud platforms.

**Programming model**

Serverless functions have limited expressiveness as they are built to scale. Their composition may be also limited and tailored to support cloud elasticity. To maximize scaling, serverless functions do not maintain state between executions. Instead, the developer can write code in the function to retrieve and update any needed state. The function is also able to access a context object that represents the environment in which the function is running (such as a security context). For example, a function written in JavaScript could take the input, as a JSON object, as the first parameter, and context as the second:

function main(params, context) {

return {

payload: ’Hello, ’ + params.name + ’ from ’ + params.place

};

}

**Tools and frameworks**

Creating and managing serverless functions requires several operations. Instead of managing each function independently it is much more convenient to have a framework that can logically group functions together to deploy and update them as a unit. A framework may also make it easier to create functions that are not bound to one serverless service provider by providing abstractions that hide low-level details of each serverless provider. Other frameworks may take existing popular programming models and adapt them for serverless execution. For example, Zappa and Chalice use an @app.route decorator to make it possible to write python code that looks like a webserver but can be deployed as a serverless function:

@app.route("/{name}/{place}")

def index():

return {"hello": name, "from": place }

**Programming model and DevOps challenges**

• **Tools:** Traditional tools that assumed access to servers to be able to monitor and debug applications aren’t applicable in serverless architectures, and new approaches are needed.

• **Deployment:** Developers should be able to use declarative approaches to control what is deployed and tools to support it.

• **Monitoring and debugging:** As developers no longer have servers that they can access, serverless services and tools need to focus on developer productivity. As serverless functions are running for shorter amounts of time there will be many orders of magnitude more of them running making it harder to identify problems and bottlenecks. When the functions finish the only trace of their execution is what serverless platforms monitoring recorded.

• **IDEs:** Higher level developer capabilities, such as refactoring functions (e.g., splitting and merging functions), reverting to an older version, etc. will be needed and should be fully integrated with serverless platforms.

• **Composability:** This includes being able to call one function from another, creating functions that call and coordinate a number of other functions, and higher-level constructs such as parallel executions and graphs. Tools will be needed to facilitate creation of compositions and their maintenance.

• **Long running:** Currently serverless functions are often limited in their execution time. There are scenarios that require long running (if intermittent) logic. Programming models and tools may decompose long running tasks into smaller units and provide necessary context to track them as one long running unit of work.

• **State:** Real applications often require state, and it’s not clear how to manage state in stateless serverless functions - programing models, tools, libraries etc. will need to provide necessary levels of abstraction.

• **Concurrency:** Express concurrency semantics, such as atomicity (function executions need to be serialized), etc.

• **Recovery semantics:** Includes exactly once, at most once, and at least once semantics.

• **Code granularity:** Currently, serverless platforms encapsulate code at the granularity of functions. It’s an open question whether coarser or finer grained modules would be useful.

Project was carried out by :

* Deepika Mol D.K
* Hemavathi. R
* Saranya. P
* Dharshini. C.S