CLOUD NATIVE COMPUTING FOUNDATION

CNCF WG-Serverless Whitepaper v1.0

Abstract

This paper describes a new model of cloud native computing enabled by emerging "serverless" architectures and their supporting platforms. It defines what server-less computing is, highlights use cases and successful examples of serverless computing, and shows how serverless computing differs from (and interrelates with) other cloud application development models such as Infra-structure-as-a- Service (IaaS), Platform-as-a-Service (PaaS), and container orchestration or Containers-as-a-Service (CaaS).

This paper, published by the CNCF Serverless Working Group, includes a logical description of the mechanics of a generic serverless platform with an associat-ed programming model and message format, but it does not prescribe a standard. It introduces several industry serverless platforms and their capabili-ties, but it does not recommend a particular implementation.

The CNCF Serverless Working Group is a forum for the CNCF community to explore the intersection of cloud native and serverless technology. The working group focuses on defining common terminology, scope of serverless as it relates to cloud native technology. This includes identifying common use cases and patterns with existing serverless implementations and analyzing the role of serverless relative to container orchestration. Their work will summarize potential next steps for the community and/or CNCF, outlining areas for possible harmonization, candidate projects and interoperability work.

To get involved in CNCF's work to advance serverless computing, join the CNCF Serverless Working Group or the community project CloudEvents, a draft specification for a common, vendor-neutral format for event data that is aimed to be proposed to the CNCF TOC as an official project later this year.

WG Chair/TOC Sponsor: Ken Owens (Mastercard)

WG Members (alphabetical by last name):
Sarah Allen (Google), Ben Browning (Red Hat), Lee Calcote (SolarWinds), Amir Chaudhry (Docker), Doug Davis (IBM), Louis Fourie (Huawei), Antonio Gulli (Goo-gle), Yaron Haviv (iguazio), Daniel Krook (IBM), Orit Nissan-Messing (iguazio), Chris Munns (AWS), Ken Owens (Mastercard), Mark Peek (VMWare), Cathy Zhang (Huawei), Chris A.

Additional Contributors (alphabetical by last name):
Kareem Amin (Clay Labs), Amir Chaudhry (Docker), Sarah Conway (Linux Founda-tion), Zach Corleissen (Linux Foundation), Alex Ellis (ADP), Brian Grant (Google), Lawrence Hecht (The New Stack), Lophy Liu, Diane Mueller (Red Hat), Bálint Pató, Peter Sbarski (A Cloud Guru), Peng Zhao (Hyper)

ABOUT CNCF

The Cloud Native Computing Foundation (CNCF) hosts critical projects of cloud native software stacks, including Kubernetes® and Prometheus™. CNCF provides a neutral home for collaboration, bringing together the industry's top developers, end users and vendors, including the world's largest public cloud providers.

Cloud native computing uses an open source software stack to orchestrate containerized services on any public, private or hybrid cloud. CNCF is part of The Linux Foundation, a nonprofit organization. For more information about CNCF, please visit: https://www.cncf.io/.

Table of Contents

Serverless Computing

What is serverless computing?

Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.

Serverless computing does not mean that we no longer use servers to host and run code; nor does it mean that operations engineers are no longer required. Rather, it refers to the idea that consumers of serverless computing no longer need to spend time and resources on server provisioning, maintenance, updates, scaling, and capacity planning. Instead, all of these tasks and capabilities are handled by a serverless platform and are completely abstracted away from the developers and IT/operations teams. As a result, developers focus on writing their applications' business logic. Operations engineers are able to elevate their focus to more business critical tasks.

There are two primary serverless personas:

1. **Developer:** writes code for, and benefits from the serverless platform which provides them the point of view that there are no servers nor that their code is always running.
2. **Provider:** deploys the serverless platform for an external or internal customer.

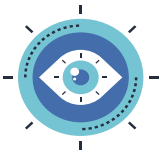
Servers are still required to run a serverless platform. The provider will need to manage servers (or virtual machines or containers). The provider will have some cost for running the platform, even when idle. A self-hosted system can still be considered serverless: typically one team acts as the provider and another as the developer.

A serverless computing platform may provide one or both of the following:

1. Functions-as-a-Service (FaaS), which typically provides event-driven computing. Developers run and manage application code with functions that are triggered by events or HTTP requests. Developers deploy small units of code to the FaaS, which are executed as needed as discrete actions, scaling without the need to manage servers or any other underlying infrastructure.
2. Backend-as-a-Service (BaaS), which are third-party API-based services that replace core subsets of functionality in an application. Because those APIs are provided as a service that auto-scales and operates transparently, this appears to the developer to be serverless.

Serverless products or platforms deliver the following benefits to developers:

1. Zero Server Ops: Serverless dramatically changes the cost model of running software applications through eliminating the overhead involved in the maintenance of server resources.

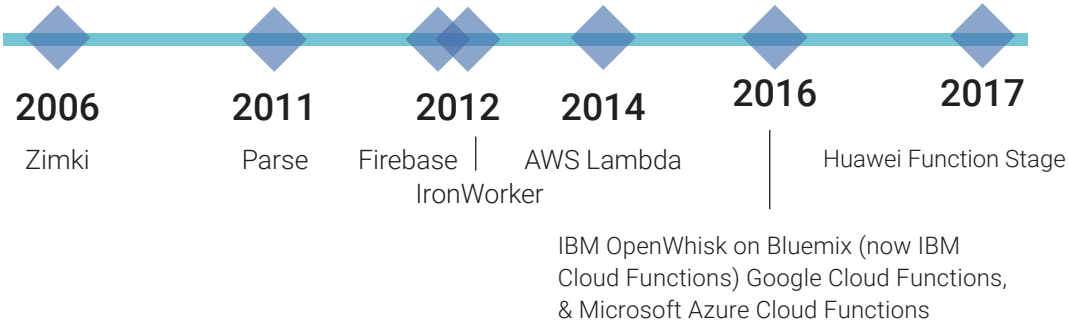


a. No provisioning, updating, and managing server infrastructure. Managing servers, virtual machines and containers is a significant overhead expense for companies when one includes headcount, tools, training, and time. Serverless vastly reduces this kind of expense.

b. Flexible Scalability: A serverless FaaS or BaaS product can instantly and precisely scale to handle each individual incoming request. For the developer, serverless platforms have no concept of "pre-planned capacity," nor do they require configuring "auto-scaling" triggers or rules. The scaling occurs automatically without intervention from the developer. Upon completion of the request processing, the serverless FaaS automatically scales down the compute resources so that there is never idle capacity.

2. No Compute Cost When Idle: One of the greatest benefits of serverless products from a consumer perspective is that there are no costs resulting from idle capacity. For example, serverless compute services do not charge for idle virtual machines or containers; in other words, there is no charge when code is not running or no meaningful work is being done. For databases, there is no charge for database engine capacity waiting idly for que ries. Of course this does not include other costs such as stateful storage costs or added capabilities/functionality/feature set.

A short history of serverless technology



While the idea of an on-demand, or "pay as you go," model can be traced back to 2006 and a platform called Zimki, one of the first uses of the term serverless is from Iron.io in 2012 with their IronWorker product, a container-based distributed work-on-demand platform.

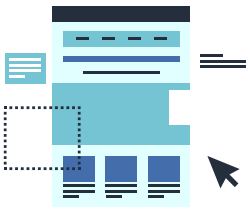
There have since been more serverless implementations in both public and private cloud. First there were BaaS offerings, such as Parse in 2011 and Firebase in 2012 (acquired by Facebook and Google, respectively). In November 2014, AWS Lambda was launched, and early 2016 saw announcements for IBM OpenWhisk on Bluemix (now IBM Cloud Functions, with the core open source project governed as Apache OpenWhisk), Google Cloud Functions, and Microsoft Azure Cloud Functions. Huawei Function Stage launched in 2017. There are also numerous open source serverless frameworks. Each of the frameworks, both public and private, have unique sets of language runtimes and services for handling events and processing data.

These are just a few examples; for a more complete and up-to-date list see the Serverless Landscape document. The Detail View: Serverless Processing Model section contains more detail about the entire FaaS model.

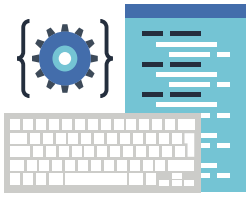
Serverless use cases

While serverless computing is widely available, it is still relatively new. In general, a serverless approach should be considered a top choice when the workload is:

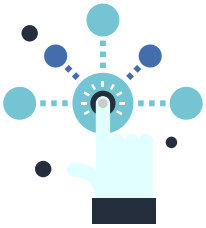
Asynchronous, concurrent, easy to parallelize into independent units of work



Infrequent or has sporadic demand, with large, unpredictable variance in scaling requirements



Stateless, ephemeral, without a major need for instantaneous cold start time



Highly dynamic in terms of changing business requirements that drive a need for accelerated developer velocity

For example, for common HTTP-based applications, there are clear upsides in terms of automated scale and a finer-grained cost model. That said, there can be some tradeoffs in using a serverless platform. For example, function startup after a period of inactivity may result in performance declines if the number of instances of the function drops down to zero. Therefore, choosing whether to adopt a serverless architecture requires a careful look at both the functional and nonfunctional aspects of the compute model.

Non-HTTP-centric and non-elastic scale workloads that weren't good fits for an IaaS, PaaS, or CaaS solution can now take advantage of the on-demand nature and efficient cost model of a serverless architecture. Some of these workloads include:

- Executing logic in response to database changes (insert, update, trigger, delete)
- Performing analytics on IoT sensor input messages (such as MQTT messages)
- Handling stream processing (analyzing or modifying data in motion)
- Managing single time extract, transform, and loarequire a lot of processing for a short time (ETL)
- Providing cognitive computing via a chatbot interface (asynchronous, but correlated)
- Scheduling tasks performed for a short time, such as cron or batch style invocations
- Serving machine learning and AI models (retrieving one or more data elements such

as tables, NLP, or images and matching against a pre-learned data model to identify text, faces, anomalies, etc.)

- Continuous integration pipelines that provision resources for build jobs on-demand, instead of keeping a pool of build slave hosts waiting for jobs to be dispatched

This section describes existing and emerging workloads and use cases where serverless architectures excels. It also includes details on early results, patterns, and best practices distilled from early success stories.

Each of these scenarios show how serverless architectures have addressed a technical problem where it would be inefficient or impossible with Iaas, PaaS, or CaaS. These examples:

- Solved a brand new problem efficiently where an on-demand model wasn't available
- Solved a traditional cloud problem much more efficiently (performance, cost)
- Showed a dimension of "largeness", whether in size of data processed or requests handled
- Showed resilience by scaling automatically (up and down) with a low error rate
- Brought a solution to market much faster than previously possible (days to hours)

The workloads listed in this section can be run on a public cloud (hosted serverless platform), on premises, or at the edge.

Multimedia processing

A common use case, and one of the earliest to crystallize, is the implementation of functions that execute some transformational process in response to a new file upload. For example, if an image is uploaded to an object storage service such as Amazon S3, that event triggers a function to create a thumbnail version of the image and store it back to another object storage bucket or Database-as-a-Service. This is an example of a fairly atomic, parallelizable compute task that runs infrequently and scales in response to demand.

Examples include:

- Santander built a proof of concept using serverless functions to process mobile check deposits using optical character recognition. This type of workload is quite variable, and processing demand on payday—once every two weeks—can be several times larger than the most idle time of the pay period.

- Categorizing a film automatically by passing each video frame through an image recognition service to extract actor, sentiment, and objects; or processing drone footage of a disaster area to estimate the extent of damage.

Database changes or change data capture (CDC)

In this scenario, a function is invoked when data is inserted, modified, or deleted from a database. In this case, it functions similarly to a traditional SQL trigger, almost like a side effect or action parallel to the main synchronous flow. The effect is to execute an asynchronous piece of logic that can modify something within that same database (such as logging to an audit table), or in turn invoke an external service (such as sending an email) or updating an additional database such as in the case of DB CDC (change data capture) use case. These use cases can vary in their frequency and need for atomicity and consistency due to business need and distribution of services that handle the changes.

Examples include:

- Auditing changes to a database, or ensuring that they meet a particular quality or analytics standard for acceptable changes.
- Automatically translating data to another language as or shortly after it's entered.

IoT sensor input messages

With the explosion of autonomous devices connected to networks comes additional traffic that is both massive in volume and uses lighter-weight protocols than HTTP. Efficient cloud services must be able to quickly respond to messages and scale in response to their proliferation or sudden influx of messages. Serverless functions can efficiently manage and filter MQTT messages from IoT devices. They can both scale elastically and shield other services downstream from the load.

Examples include:

- GreenQ's sanitation use case (the Internet of Garbage) where the truck pickup route was optimized based on the relative fullness of trash receptacles.

- Using serverless on an IoT device (like AWS Greengrass) to collect local sensor data, normalize it, compare with triggers, and push events up to an aggregation unit/cloud.

Stream processing at scale

Another non-transactional, non-request/response type of workload is processing data within a potentially infinite stream of messages. Functions can be connected to a source of messages that must each be read and processed from an event stream. Given the high performance, highly elastic, and compute intensive processing workload, this can be an important fit for serverless. In many cases, stream processing requires comparing data against a set of context objects (in a NoSQL or in-mem DB) or aggregating and storing data from streams into a object or a database system.

Examples include:

- Mike Roberts has a good Java/AWS Kinesis example handling billions of messages efficiently.
- SnapChat uses serverless on Google AppEngine to process messages.

Chat bots

Interacting with humans doesn't necessarily require millisecond response time, and in many ways a bot that replies to humans may actually benefit from a slight delay to make the conversation feel more natural. Because of this, a degree of initial latency from waiting for the function to be loaded from a cold start may be acceptable. A bot may also need to be extremely scalable when added to a popular social network like Facebook, Whack, so pre-provisioning an always-on daemon in a PaaS or IaaS model in anticipation of sudden or peak demand may not be as efficient or cost-effective as a serverless approach.

Examples include:

- Support and sales bots that are plugged into large social media services such as Facebook or other high traffic sites.
- Messaging app Wuu uses Google Cloud Functions to enable users to create and share content that disappears in hours or seconds.
- See also the HTTP REST APIs and web applications below.

Batch jobs or scheduled tasks

Jobs that require intense parallel computation, IO, or network access for only a few minutes a day in an asynchronous manner can be a great fit for serverless. Jobs can consume the resources they need efficiently for the time they run in an elastic manner, and not incur resource costs for the rest of the day when they are not used.

Examples include:

- A scheduled task could be a backup job that runs every night.
- Jobs that send many emails in parallel scale out function instances.

HTTP REST APIs and web applications

Traditional request/response workloads are still quite a good fit for serverless whether the workload is a static web site or one that uses a programming language like JavaScript or Python to generate a response on demand. Even though they may incur a startup cost for the first user, there is precedent for that type of delay in other compute models, such as the initial compilation of a JavaServer Page into a servlet, or starting up a new JVM to handle additional load. The benefit is that individual REST calls (each of the 4 GET, POST, UPDATE, and DELETE endpoints in a microservice, for example) can scale independently and be billed separately, even if they share a common data backend.

Examples include:

- Australian census ported to a serverless architecture shows speed of development, cost improvements, and autoscaling.
- "How I cut my AWS bill by 90% by going serverless."
- AutoDesk example: "Costs a small fraction (~1%) of the traditional cloud approach."
- Online coding/education (exam, test, etc.) runs exercise code in an event-driven environment, and provides feedback to the user based on a comparison with expected results for that exercise. The serverless platform runs the answer-checking on demand and scale as needed, paying for only the time during which code is running.

Mobile backends

Using serverless for mobile backend tasks is also attractive. It allows developers to build on the REST API backend workload above the BaaS APIs, so they can spend time optimizing a mobile app and less on scaling its backend. Examples include: optimizing graphics for a video game and not investing in servers when the game becomes a viral hit; or for consumer business applications that need quick iterations to find product/market fit, or when time-to-market is critical. Another example is in batching notifications to users or processing other asynchronous tasks for an offline-first experience.

Examples include:

- Mobile apps that need a small amount of server-side logic; developers can focus their effort on native code development.
- Mobile apps that use direct-from-mobile access to BaaS using configured security policy, such as Firebase Auth/Rules or Amazon Cognito, with event-triggered serverless compute.
- "Throwaway" or short-term use mobile applications, such as the scheduling app for a large conference, that has very little demand on the weekends before and after the conference, but needs to scale up and down greatly; surges post-keynote based on schedule viewing demands over the course of the event on Monday and Tuesday mornings, then back down at midnight those days.

Business Logic

The orchestration of microservice workloads that execute a series of steps in a business process is another good use case for serverless computing when deployed in conjunction with a management and coordination function. Functions that perform specific business logic such as order request and approval, stock trade processing, etc. can be scheduled and coordinated with a stateful manager. Event requests from client portals can be serviced by such a coordination function and delivered to appropriate serverless functions.

Examples include:

- A trading desk that handles stock market transactions and processes trade orders and confirmations from a client. The orchestrator manages the trades using a graph of states. An initial state accepts a trade request from a client portal and delivers the request to a microservice function to parse the request and authenticate the client. Subsequent states steer the workflows based on a buy or sell transaction, validate fund balances, ticker, etc. and send a confirmation to the client. On receipt of a confirmation request event from the client, follow-on states invoke functions that manage execution of the trade, update the account, and notify the client of the completion of the transaction.

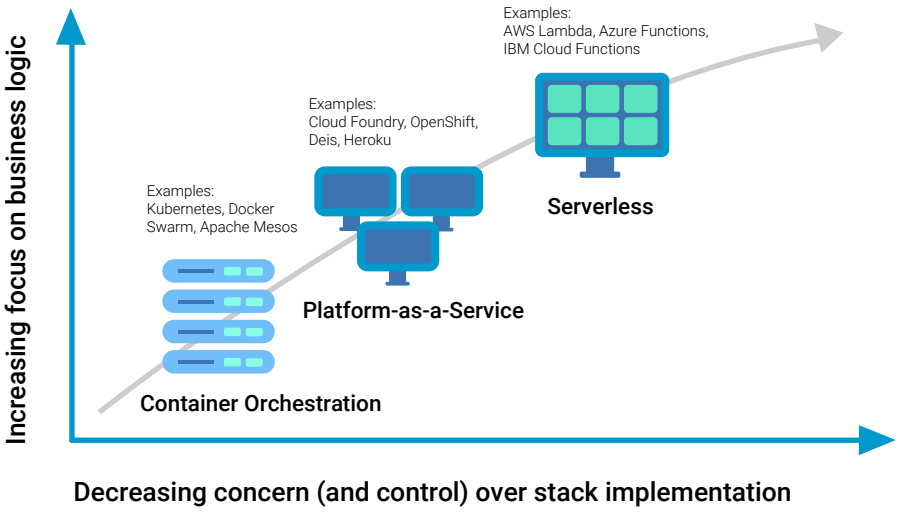
Continuous Integration Pipeline

A traditional CI pipeline includes a pool of build slave hosts waiting idle for jobs to be dispatched. Serverless is a good pattern to remove the need for pre-provisioned hosts and reduce costs. Build jobs are triggered by new code commit or PR merged. A function call is invoked to run the build and test case, executing only for the time needed, and not incurring costs while unused. This lowers costs and can reduce bottlenecks through autoscaling to meet demand.

Examples include:

- Serverless CI - Hyper.sh integration for Buildbot

Serverless vs. Other Cloud Native Technologies



There are three primary development and deployment models that most application developers might consider when looking for a platform to host their cloud-native applications. Each model has its own set of differing implementations (whether an open source project, hosted platform, or on-premises product). These three models are commonly built upon container technology for its density, performance, isolation, and packaging features, but containerization is not a requirement.

In order of increasing abstraction away from the actual infrastructure that is running their code, and toward a greater focus on only the business logic that's developed, they are Container Orchestration (or Containers-as-a-Service), Platform-as-a-Service, and Serverless (Functions-as-a-Service). All of these approaches provide a way to deploy cloud-native application, but they prioritize different functional and non-functional aspects based on their intended developer audience and workload type. The following section lists some of the key characteristics of each.

Keep in mind that no single approach is a silver bullet for all cloud-native development and deployment challenges. It's important to match the needs of your particular workload against the benefits and drawbacks of each of these common cloud-native development technologies. It's also important to consider that subcomponents of your application may be more suitable for one approach versus another, so a hybrid approach is possible.

Container Orchestration

Containers-as-a-Service (CaaS) - Maintain full control over infrastructure and get maximum portability. Examples: Kubernetes, Docker Swarm, Apache Mesos

Container orchestration platforms like Kubernetes, Swarm, and Mesos allow teams to build and deploy portable applications, with flexibility and control over configuration, which can run anywhere without the need to reconfigure and deploy for different environments.

Benefits include maximum control, flexibility, reusability and ease of bringing existing containerized apps into the cloud, all of which is possible because of the degree of freedom provided by a less-opinionated application deployment model.

Drawbacks of CaaS include significantly more added developer responsibility for the operating systems (including security patches), load balancing, capacity management, scaling, logging and monitoring.

Target audience

- Developers and operations teams who want control over how their application and all of its dependencies are packaged and versioned, ensuring portability and reuse across deployment platforms.
- Developers looking for high performance among a cohesive set of interdependent, independently scaled microservices.
- Organizations moving containers to the cloud, or deploying across private/public clouds, and who are experienced with end-to-end cluster deployments.

Developer/Operator experience

- Create a Kubernetes cluster, Docker Swarm stack, or Mesos resource pool (done once).
- Iterate and build container images locally.
- Push tagged application images to a registry.

- Deploy containers based on container images to cluster.
- Test and observe application in production.

Benefits

- The developer has the most control, and the responsibility that comes with that power, for what's being deployed. With container orchestrators one can define the exact image versions to deploy, and in what configuration, along with policies that govern their runtime.
- Control over runtime environment (e.g. runtimes, versions, minimal OS, network configuration).
- Greater reusability and portability of container images outside the system.
- Great fit for bringing containerized apps and systems to the cloud.

Drawbacks

- More responsibility for the filesystem image and execution environment, including security patches and distribution optimizations.
- More responsibility for managing load balancing and scaling behavior.
- Typically more responsibility for capacity management.
- Typically longer startup times, today.
- Typically less opinionated about application structure, so there are fewer guide rails.
- Typically more responsibility for build and deployment mechanisms.
- Typically more responsibility for integration of monitoring, logging, and other common services.

Platform-as-a-Service

Platform-as-a-Service (PaaS) - *Focus on the application and let the platform handle the rest.* Examples: Cloud Foundry, OpenShift, Deis, Heroku

Platform-as-a-Service implementations enable teams to deploy and scale applications using a broad set of runtimes, binding to a catalog of data, AI, IoT, and security services through injection of configuration information into the application, without having to manually configure and manage a container and OS. It is a great fit for existing web apps that have a stable programming model.

Benefits include easier management and deployment of applications, auto-scaling and pre-configured services for the most common application needs.

Drawbacks include lack of OS control, granular container portability and load balancing and application optimization as well as potential vendor lock-in and needing to build and manage monitoring and logging capabilities on most PaaS platforms.

Target audience

- Developers who want a deployment platform that enables them to focus on application source code and files (not packaging them) and without worrying about the OS.
- Developers who are creating more traditional HTTP-based services (apps and APIs) with routable hostnames by default. However, some PaaS platforms now support generic TCP routing as well.
- Organizations that are comfortable with a more established model of cloud computing (as compared to serverless) supported by comprehensive docs and many samples.

Developer/Operator experience

- Iterate on applications, build and test in local web development environment.
- Push application to PaaS, where it is built and runs.
- Test and observe application in production.
- Update configuration to ensure high availability and scale to match demand.

Benefits

- The developer's frame of reference is on the application code, and the data services to which it connects. There's less control over the actual runtime, but the developer avoids a build step and can also choose scaling and deployment options.
- No need to manage underlying OS.
- Buildpacks provide influence over the runtime, giving as much or as little control (sensible defaults) as desired.
- Great fit for many existing web apps with a stable programming model.

Drawbacks

- Loss of control over OS, possibly at the mercy of buildpack versions.
- More opinionated about application structure, tending towards 12-Factor microservices best practices at the potential cost of architecture flexibility.
- Potential platform lock-in.

Serverless

Functions-as-a-Service (FaaS) - Write logic as small pieces of code that respond to a variety of events. Examples: AWS Lambda, Azure Functions, IBM Cloud Functions based on Apache OpenWhisk, Google Cloud Functions, Huawei Function Stage and Function Graph, Kubeless, iron.io, funktion, fission, nuclio

Serverless enables developers to focus on applications that consist of event-driven functions that respond to a variety of triggers and let the platform take care of the rest - such as trigger-to-function logic, information passing from one function to another function, auto-provisioning of container and run-time (when, where, and what), auto-scaling, identity management, etc.

The benefits include the lowest requirement for infrastructure management of any of the cloud native paradigms. There is no need to consider operating or file system, runtime or even container management. Serverless enjoys automated scaling, elastic load balancing and the most granular "pay-as-you-go" computing model.

Drawbacks include less comprehensive and stable documentation, samples, tools, and best practices; challenging debugging; potentially slower responses times; lack of standardization and ecosystem maturity and potential for platform lock-in.

Target audience

- Developers who want to focus more on business logic within individual functions that automatically scale in response to demand and closely tie transactions to cost.
- Developers who want to build applications more quickly and concern themselves less with operational aspects.
- Developers and teams creating event-driven applications, such as those that respond to database changes, IoT readings, human input etc.
- Organizations that are comfortable adopting cutting-edge technology in an area where standards and best practices have not yet been thoroughly established.

Developer/Operator experience

- Iterate on function, build and test in local web development environment.
- Upload individual functions to the serverless platform.
- Declare the event triggers, the functions and its runtime, and event-to-function relationship.
- Test and observe application in production.

- No need to update configuration to ensure high availability and scale to match demand.

Benefits

- The developer point of view has shifted farther away from operational concerns like managing the deployment of highly available functions and more toward the function logic itself.
- The developer gets automated scaling based on demand/workload.
- Leverages a new "pay-as-you-go" cost model that charges only for the time code is actually running.
- OS, runtime, and even container lifecycle is completely abstracted (serverless).
- Better fit for emerging event-driven and unpredictable workloads involving IoT, data, messages.
- Typically stateless, immutable and ephemeral deployments. Each function runs with a specified role and well-defined/limited access to resources.
- Middleware layers will get tuned/optimized, will improve application performance over time.

- Strongly promotes the microservices model, as most serverless runtimes enforce limits on the size or execution time of each individual function.
- As easy to integrate third-party APIs as custom-built serverless APIs, both scaling with usage, with flexibility of being called from client or server.

Drawbacks

- An emerging computing model, rapid innovation with less comprehensive and stable documentation, samples, tools, and best practices.
- Due to the more dynamic nature of the runtime, it may be more challenging to debug when compared to IaaS and PaaS.
- Due to the on-demand structure, the "cold start" aspect on some serverless runtimes could be a performance issue if the runtime removes all instances of a function when idle.
- In more complex cases (e.g., functions triggering other functions), there can be more operational surface area for the same amount of logic.
- Lack of standardization and ecosystem maturity.
- Potential for platform lock-in due to the platform's programming model, eventing/message interface and BaaS offerings.

Which Cloud Native Deployment Model Should You Use?

In order to determine which model is best for your particular needs, a thorough evaluation of each approach (and several model implementations) should be made. This section will provide some suggestions for areas of consideration as there is no one-size-fits-all solution.

Evaluate Features and Capabilities

Experiment with each approach. Find what works best for your needs from a functionality and development experience point of view. You're trying to find the answers to questions such as:

- Does my application seem like a fit based on the workloads described in the earlier section where serverless proves its value? Do I anticipate a major benefit from serverless versus the alternatives that justifies a change?
- How much control do I really need over the runtime and the environment in which it runs? Do minor runtime version changes affect me? Can I override the defaults?
- Can I use the full set of features and libraries available in my language of choice? Can I install additional modules if needed? Do I have to patch or upgrade them myself?
- How much operational control do I need? Am I willing to give up over the lifecycle of the container or execution environment?
- What if I need to change the code of my service? How fast can I deploy it?
- How do I secure my service? Do I have to manage that? Or can I offload to a service that can do it better?

Evaluate and Measure Operational Aspects

Gather performance numbers such as time to recovery with PaaS and a Container Orchestrator as well as cold starts with a Serverless platform. Explore and quantify the impact of other important non-functional characteristics of your application on each platform, such as:

Resiliency:

- How do I make my application resilient to a data-center failure?
- How do I ensure continuity of service while I deploy updates?

- What if my service fails? Will the platform automatically recover? Will it be invisible to end-users?

Scalability:

- Does the platform support auto-scaling in case I have a sudden change in demand?
- Is my application designed to take advantage of stateless scaling effectively?
- Will my serverless platform overwhelm any other components such as a database? Can I manage or throttle back-pressure?

Performance:

- How many function invocations per second per instance or per HTTP client?
- How many servers or instances will be required for given workload?
- What is the delay from invocation to response (in cold and warm start)?
- Is the latency between the microservices, vs co-located features within a single deployment, an issue?

One of the potential outcomes of the CNCF Serverless Working Group could be a decision framework for when to choose a particular model, and how to test given a set of recommended tools. See the Conclusion section for more detail.

Evaluate and Consider the Full Spectrum of Potential Costs

This covers both development costs and runtime resource costs.

- Not everyone will have the luxury of starting their development activities from scratch. Therefore, the cost of migrating existing applications to one of the cloud native models needs to be carefully considered. While a lift-and-shift model to containers may seem the cheapest, it may not be the most cost effective in the long run. Likewise, the on-demand model of serverless is very attractive from a cost perspective, but the development effort needed to split a monolithic application into functions could be daunting.
- How much will integration with dependent services cost? Serverless compute may appear the most economical at first, but it may require more expensive third party service costs, or autoscale very quickly which may result in greater usage fees.
- Which features/services do the platforms offer for free? Am I willing to buy into a vendor's ecosystem at the potential cost of portability?

Running an Application Based on Multiple Platforms

When looking at the various cloud hosting technologies that are available it may

not be obvious but there is no reason why a single solution needs to be used for all deployments. In fact, there is no reason why the same solution needs to be used within a single application. Once an application is split into multiple components, or microservices, you then have the freedom to deploy each one separately on completely different infrastructures, if that's what's best for your needs.

Likewise, each microservice can also be developed with the best technology (i.e. language) for its particular purpose. The freedom that comes with "breaking up of the monolith" brings new challenges though, and the following sections highlight some of the aspects that should be considered when choosing a platform and developing your microservices.

Split Components Across Deployment Targets

Think about matching the right technology to the right job, for example an IoT demo might use both a PaaS application to handle requests to a dashboard of connected devices and a set of serverless functions to handle MQTT message events from the devices themselves. Serverless isn't a magic bullet, but rather a new option to consider within your cloud native architecture.

Design for More Than One Deployment Target

Another design choice is to make your code as generic as possible, allow it to be tested locally, and rely on contextual information, such as environment variables, to influence how it runs in particular environments. For example, a set of plain old Java objects might be able to run within any of the three major environments, and exact behavior tailored based on available environment variables, class libraries, or bound services.

Continue to Use DevOps Pipelines for Any of the Approaches

Most container orchestration platforms, PaaS implementations, and serverless frameworks can be driven by command line tools, and the same container image can potentially be built once and reused across each platform.

Consider Abstractions to ease Portability Between Models

There is a growing ecosystem of third party projects that bridge the gap for porting HTTP-based web applications that currently run on a PaaS or a CaaS to serverless platforms. These include several tools from Serverless, Inc. and the Zappa Framework.

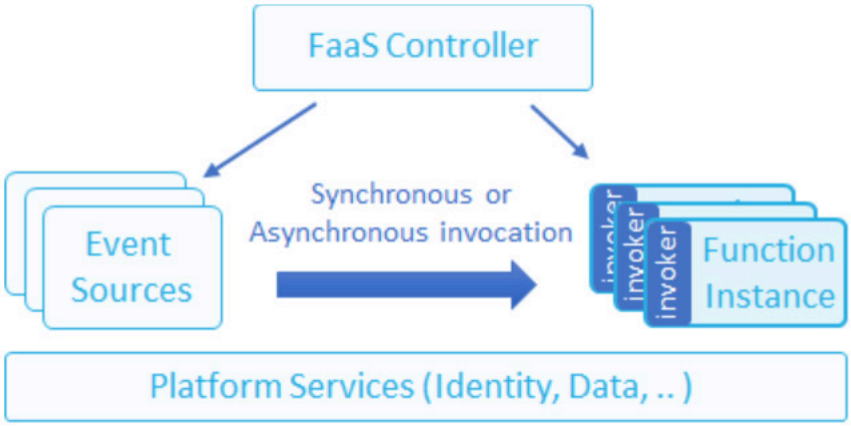
Serverless frameworks provide adaptors that enable applications written using popular web application frameworks such as Python WSGi and JAX-RS REST API to run on

serverless platforms. These frameworks can also provide portability and abstraction of the difference between multiple serverless platforms.

Detail View: Serverless Processing Model

This section summarizes the current function usage within serverless frameworks and draws a generalization of the serverless function requirements, lifecycle, invocation types and the required abstractions. We aim to define the serverless function specification so that the same function could be coded once and used in different serverless frameworks. This section does not define the exact function configuration and APIs.

We can generalize a FaaS solution as having several key elements shown in the following diagram:



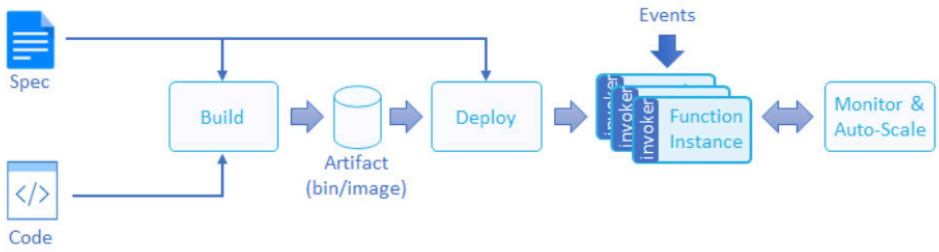
- **Event sources** - trigger or stream events into one or more function instances
- **Function instances** - a single function/microservice, that can be scaled with demand
- **FaaS Controller** - deploy, control and monitor function instances and their sources
- **Platform services** - general cluster or cloud services used by the FaaS solution (sometimes referred to as Backend-as-a-Service)

Let's start by looking at the lifecycle of a function in a serverless environment.

Function LifeCycle

The following sections describe the various aspects of a function's lifecycle and how serverless frameworks/runtimes typically manage them.

Function Deployment Pipeline



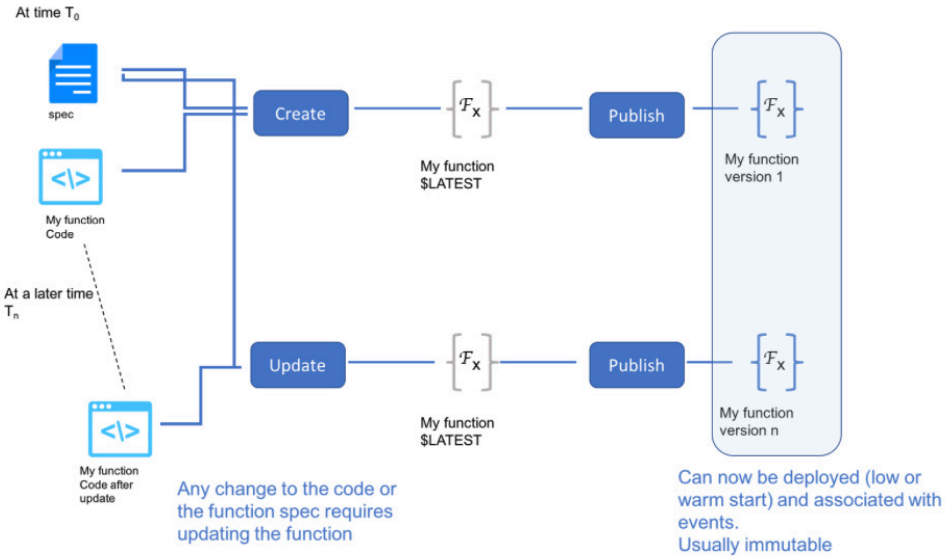
The lifecycle of a function begins with writing code and providing specifications and metadata (see Function Definition below), a "builder" entity will take the code and specification, compile, and turn it into an artifact (a code binary, package, or container image). Artifacts then get deployed on a cluster with a controller entity in charge of scaling the number of function instances based on the events traffic and/or load on the instances.

Function Operations

Serverless frameworks may allow the following actions and methods to define and control function lifecycle:

- Create - Creates a new function, including its spec and code
- Publish - Creates a new version of a function that can be deployed on the cluster
- Update Alias/Label (of a version) - Updating a version alias
- Execute/Invoke - Invoke a specific version not through its event source
- Event Source association - Connect a specific version of a function with an event source
- Get - Returns the function metadata and spec
- Update - Modify the latest version of a function
- Delete - Deletes a function, could delete a specific version or the function with all its versions

- List - Show the list of functions and their metadata
- Get Stats - Return statistics about the runtime usage of a function
- Get Logs - Return the logs generated by a function



When creating a function, providing its metadata (as described later under function spec) as part of the function creation, it will be compiled and possibly published. A function may be started, disabled and enabled later on. Function deployments need to be able to support the following usecases:

- Event streaming, in this use case there may always be events in queue however the processing may need to be paused/resumed through an explicit request
- Warm startup - function that has minimal number of instances at any time, such that the "first" event received has a warm start since the function is already deployed and is ready to serve the event (as opposed to a cold start where the function gets deployed on the first invocation by an "incoming" event)

A user can **Publish** a function, this will create a new version (copy of the "latest" version), the published version may be tagged/labeled or have aliases, see more below.

A user may want to **Execute/Invoke** a function directly (bypass the event source or API gateway) for debug and development processes. A user may specify invocation parameters such as desired version, Sync/Async operation, Verbosity level, etc.

Users may want to obtain function **Statistics** (e.g. number of invocations, average runtime, average delay, failures, retries, etc.), statistics can be the current metric values or a time-series of values (e.g. stored in Prometheus or cloud provider facility such as AWS Cloud Watch).

Users may want to retrive function **Log** data. This may be filtered by severity level and/or time range, and/or content. The Log data is per function, it include events such as function creation and deletion, explicit errors, warnings, or debug messages, and optionally the Stdout or Stderr of a function. It would be prefered to have one log entry per invocation or a way to associate log entries with a specific invocation (to allow simpler tracking of the function execution flow).

Function Versioning and Aliases

A Function may have multiple versions, providing the user the ability to run different level of codes such as beta/production, A/B testing etc. When using versioning, the function version is "latest" by default, the "latest" version can be updated and modified, potentially triggering a new build process on every such change.

Once a user wants to freeze a version he will use a **Publish** operation that will create a new version with potential tags or aliases (e.g. "beta", "production") to be used when configuring an event source, so an event or API call can be routed to a specific function version. Non-latest function versions are immutable (their code and all or some of the function spec) and cannot be changed once published; functions cannot be "un-published" instead they should be deleted.

Note that most implementations today do not allow function branching/fork (updating an old version code) since it complicates the implementation and usage, but this may be desired in the future.

When there are multiple versions of the same function, the user must specify the version of the function he would like to operate and how to divide the traffic of events between the different versions (e.g. a user can decide to route 90% of an event traffic to a stable version and 10% to a beta version a.k.a "canary update"). This can be either by specifying the exact version or by specifying the version alias. A version alias will typically reference to a specific function version.

When a user creates or updates a function, it may drive a new build and deployment depending on the nature of the change.

Event Source to Function Association

Functions are invoked as a result of an event triggered by an event source. There is a n:m mapping between functions and event sources. Each event source could be used to invoke more than a single function, a function may be triggered by multiple event sources. Event source could be mapped to a specific version of a function or to an alias of the function, the latter provides a means for changing the function and deploys a new version without the need to change the event association. Event Source could also be

defined to use different versions of the same function with the definition of how much traffic should be assigned to each.

After creating a function, or at a later point in time, one would need to associate the event source that should trigger the function invocation as a result of that event. This requires a set of actions and methods such as:

- Create event source association
- Update event source association
- List event source associations

Event Sources

Different types of event sources includes:

- Event and messaging services, e.g.: RabbitMQ, MQTT, SES, SNS, Google Pub/Sub
- Storage services, e.g.: S3, DynamoDB, Kinesis, Cognito, Google Cloud Storage, Azure Blob, iguazio V3IO (object/stream/DB)
- Endpoint services, e.g.: IoT, HTTP Gateway, mobile devices, Alexa, Google Cloud Endpoints
- Configuration repositories, e.g.: Git, CodeCommit
- User applications using language-specific SDKs
- SchEnables invocation of functions on regular intervals.

While the data provided per event could vary between the different event sources, the event structure should be generic with the ability to encapsulate specific information with respect to the event source (details under Event data and metadata).

Function Requirements

- The following list describes the set of common requirements that functions, and serverless runtimes, should meet based on the state of art as of today:
- Functions must be decoupled from the underlying implementation of the different event classes
- A Function may be invoked from multiple event sources
  - No need for a different function per invocation method
- Event source may invoke multiple functions
- Functions may require a mechanism for long-lasting bindings with underlying platform services, which may be cross function invocations. Functions could be short-lived but the bootstrap may be expensive if it needs to be done on every

invocation, such as in the case of logging, connecting, mounting external data sources.

- Each function can be written in a different code language from other functions that are using within the same application
- Function runtime should minimize the event serialization and deserialization overhead if possible (e.g. use native language structures or efficient encoding schemes).

Workflow related requirements:

- Functions may be invoked as part of a workflow, where the result of the function is a trigger of another function
- A function can be triggered by an event or an "and/or combination of events"
- One event could trigger multiple functions executed in sequence or parallel
- "and/or combination of events" could trigger m functions running in sequence or parallel or branching
- In the middle of the workflow, different events or function results might be received, which would trigger branching to different functions
- Part or all of a function's result needs to be passed as input to another function
    - Functions may require a mechanism for long-lasting bindings with underlying platform services, which may be cross function invocations or function could be short lived.

Function Invocation Types

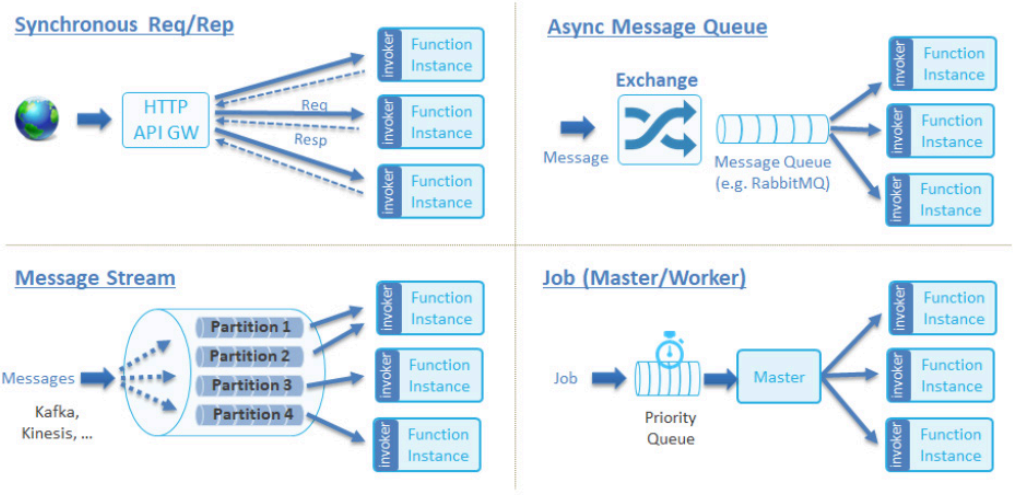
Functions can be invoked from different event sources depending on the different use-cases, such as:

1. Synchronous Request (Req/Rep), e.g. HTTP Request, gRPC call
    - Client issues a request and waits for an immediate response. This is a blocking call.
2. Asynchronous Message Queue Request (Pub/Sub), e.g. RabbitMQ, AWS SNS, MQTT, Email, Object (S3) change, scheduled events like CRON jobs
    - Messages are published to an exchange and distributed to subscribers
    - No strict message ordering. Exactly once processing
3. Message/Record Streams: e.g. Kafka, AWS Kinesis, AWS DynamoDB Streams, Database CDC

- An ordered set of messages/records (must be processed sequentially)
- Usually a stream is sharded to multiple partitions/shards with a single worker (the shard consumer) per shard
- Stream can be produced from messages, database updates (journal), or files (e.g. CSV, Json, Parquet)
- Events can be pushed into the function runtime or pulled by the function runtime

4. **Batch Jobs**, e.g. ETL jobs, distributed deep learning, HPC simulation

- Jobs are scheduled or submitted to a queue, and processed at run time using multiple function instances in parallel, each handling one or more portion of the working set (a task)
- The job is complete when all the parallel workers successfully completed all the computation tasks



Function Code

Function code and dependencies and/or binaries may reside in an external repository such as S3 object bucket or Git repository, or provided directly by the user. If the code is in an external repository the user will need to specify the path and credentials.

The serverless framework may also allow the user to watch the code repository for changes (e.g. using a web hook) and build the function image/binary automatically on every commit.

A function may have dependencies on external libraries or binaries, those need to be provided by the user including a way to describe their build process (e.g. using a Dockerfile, Zip).

Additionally, the function could be provided to the framework via some binary packaging, such as an OCI image.

Function Definition

Serverless function definitions may contain the following specifications and metadata, the function definition is version specific:

- Unique ID
- Name
- Description
- Labels (or tags)
- Version ID (and/or Version Aliases)
- Version creation time
- Last Modified Time (of function definition)
- Function Handler
- Runtime language
- Code + Dependencies or Code path and credentials
- Environment Variables
- Execution Role and Secret
- Resources (Required CPU, Memory)
- Execution Timeout
- Log Failure (Dead Letter Queue)
- Network Policy / VPC
- Data Bindings

Metadata details

Function frameworks may include the following metadata for functions:

- Version - each function version should have a unique identifier, in addition versions can be labeled using one or more aliases (e.g. “latest”, “production”, “beta”). API gateways and event sources would route traffic/events to a specific function version.

- Environment Variables - the user may specify Environment variables that will be provided to the function at runtime. Environment variables can also be derived from secrets and encrypted content, or derived from platform variables (e.g. like Kubernetes EnvVar definition). Environment variables enable developers to control function behavior and parameters without the need to modify code and/or rebuild the function allowing better developer experience and function reuse.
- Execution Role - the function should run under a specific user or role identity that grants and audits its access to platform resources.
- Resources - define the required or maximum hardware resources such as Memory and CPU used by the function.
- Timeout - specify the maximum time a function call can run until it is terminated by the platform.
- Failure Log (Dead Letter Queue) - a path to a queue or stream that will store the list of failed function executions with appropriate details.
- Network Policy - the network domain and policy assigned to the function (for the function to communicate with external services/resources).
- Execution Semantics - specifies how the functions should be executed (e.g. at least once, at most once, exactly once per event).

Data Bindings

Some serverless frameworks allow a user to specify the input/output data resources used by the function, this enables developer simplicity, performance (data connections are preserved between executions, data can be pre-fetched, etc.), and better security (data resources credentials are part of the context not the code).

Bound data can be in the form of files, objects, records, messages etc., the function spec may include an array of data binding definitions, each specifying the data resource, its credentials and usage parameters. Data binding can refer to event data (e.g. the DB key is derived from the event "username" field), see more in: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings.

Function Input

Function input includes event data and metadata, and may include a context object.

Event data and metadata

Event details should be passed to the function handler, different events may have varying metadata, so it would be desirable for functions to be able to determine the type

of event and easily parse the common and event specific metadata.

It can be desirable to decouple the event classes from the implementation, for example: a function processing a message stream would work the same regardless if the streaming storage is Kafka or Kinesis. In both cases, it will receive a message body and event metadata, the message may be routed between different frameworks.

An event may include a single record (e.g. in Request/Response model), or accept multiple records or micro-batch (e.g. in Streaming modes).

Examples for common event data and metadata used by FaaS solutions:

- Event Class/Kind
- Version
- Event ID
- Event Source/Origin
- Source Identity
- Content Type
- Message Body
- Timestamp

Examples for event/record specific metadata

- **HTTP:** Path, Method, Headers, Query Args
- **Message Queue:** Topic, Headers
- **Record Stream:** table, key, op, modified-time, old fields, new fields

Examples of event source structures:

- http://docs.aws.amazon.com/lambda/latest/dg/eventsources.html
- https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings
- https://cloud.google.com/functions/docs/concepts/events-triggers

Some implementations focus on JSON as a mechanism to deliver event information to the functions. This may add substantial serialization/deserialization overhead for higher speed functions (e.g. stream processing), or low-energy devices (IoT). It may be worth considering native language structures or additional serialization mechanisms as options in these cases.

Function Context

When functions are called, frameworks may want to provide access to platform resources or general properties that span multiple function invocations, instead of placing all the static data in the event or forcing the function to initialize platform services on every call.

Context is delivered as a set of input properties, environment variables or global variables. Some implementations use a combination of all three.

Examples for Context:

- Function Name, Version, ARN
- Memory Limit
- Request ID
- Cloud Region
- Environment Variables
- Security keys/tokens
- Runtime/Bin paths
- Log
- Data binding

Some implementations initialize a log object (e.g. as global variables in AWS or part of the context in Azure), using the log object users can track function execution using integrated platform facilities. In addition to traditional logging, future implementations may abstract counter/monitoring and tracing activities as part of the platform context to further improve functions usability.

Data bindings are part of the function context, the platform initiates the connections to the external data resources based on user configuration, and those connections may be reused across multiple function invocations.

Function Output

When a function exits it may:

- Return a value to the caller (e.g. in HTTP request/response example)
- Pass the result to the next execution phase in a workflow
- Write the output to the log

There should be a deterministic way to know if the function succeeded or failed through a returned error value or exit code.

Function output may be structured (e.g. HTTP response object) or unstructured (e.g. some output string).

Serverless Function Workflow

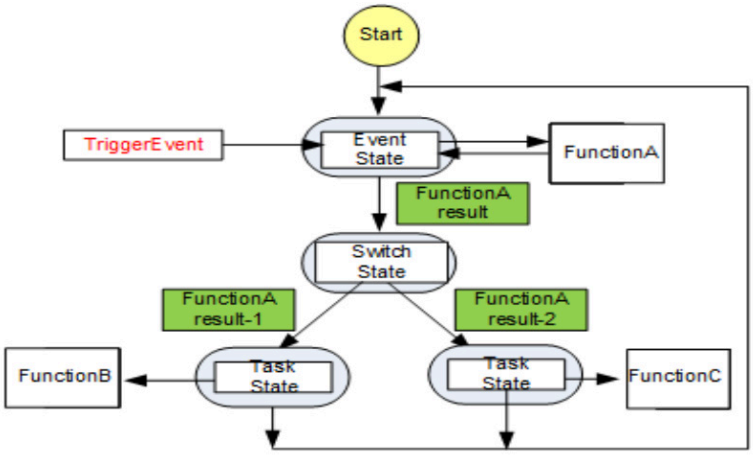
In the serverless domain, use cases fall into one of the following categories:

1. One event triggers one function
2. An and/or combination of events trigger one function
3. One event triggers multiple functions executed in sequence or in parallel
4. The result of the function could be a trigger of another function
5. N events (in and/or) triggers m functions, i.e. an event-function interleaved workflow, eg. event1 triggers function1, completion of function1 together with event2 and event 3 trigger function2, then different result of function2 triggers branching to function3 or function4.

A user needs a way to specify their serverless use case or workflow. For example, one use case could be "do face recognition on a photo when a photo is uploaded onto the cloud storage (photo storage event happens)." Another IoT use case could be "do motion analysis" when a motion detection event is received, then depending on the result of the analysis function, either "trigger the house alarm plus call to the police department" or just "send the motion image to the house owner." Refer to the use cases section for more detailed information.

AWS provides "step function" primitives (state machine based primitives) for the user to specify its workflow, but step function does not allow specification of what event/events triggering what functions in the workflow. Please refer to https://aws.amazon.com/step-functions/.

The following graph is an example of a user's workflow that involves events and functions. Using such a function graph, the user can easily specify the interaction between events and functions as well as how information can be passed between functions in the workflow.



The Function Graph States include the following :

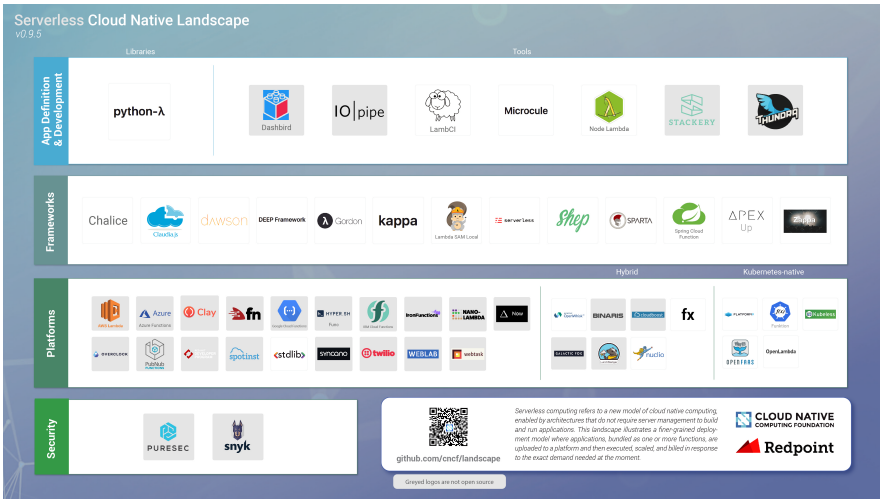
Event State	This state allows for waiting for events from event sources, and then triggering a function run or multiple functions run in sequence or in parallel or in branch.
Operation/Task State	This state allows the run of one or more functions in sequence or in parallel without waiting for any event.
Switch/Choice State	This state permits transitions to multiple other states (eg. a previous function result triggers branching/transition to different next states).
End/Stop State	This state terminates the workflow with Fail/Success.
Pass State	This state injects event data in-between two states.
Delay/Wait State	This state causes the workflow execution to delay for a specified duration or until a specified time/date.

States and associated information need to be saved in some persistent storage for failure recovery. In some use cases, the user may want information from one state to be passed to the next state. This information could be part of the function execution result or part of input data associated with an event trigger. An information filter needs to be defined at each state to filter out the information that needs to be passed between states.

Conclusion

Serverless architectures provide an exciting new deployment option for cloud native workloads. As we saw in the Serverless Workloads section there are certain use cases where serverless technology provides major benefits over other cloud hosting technologies.

However, serverless technology is not a perfect fit for all cases and careful consideration should be given to when it is appropriate. Short-lived, event-driven processing is driving early adoption and use cases for businesses that expect a high rate of change with unpredictable capacity and infrastructure needs are emerging. See the Additional References section for more reading material and insights into serverless computing.



The CNCF Servleress Working Group, in partnership with Redpoint Ventures, recently published a a Serverless Landscape. It illustrates some of the major erverless projects, tooling and services that are available in the ecosystem. It is not intended to represent a comprehensive, fully inclusive serverless ecosystem, nor is it an endorsement, rather just an overview of the landscape.. It is expected that owners of each will provide updates in an attempt to keep it up to date.

Next Steps for the CNCF

With respect to what, if anything, the CNCF should consider doing in this space, the following suggestions are offered for the Technical Oversight Committee's consideration:

- Encourage more serverless technology vendors and open source developers to join the CNCF to share ideas and build upon each other's innovation. For example, keep

the open source projects listed in the Serverless Landscape document updated and the matrix of capabilities maintained.

- Foster an open ecosystem by establishing interoperable APIs, ensuring interoperable implementations with vendor commitments and open source tools. New interoperability and portability efforts similar to CSI and CNI with the help of both platform providers and third-party developer library creators. Some of these may merit their own CNCF working group, or may continue as an initiative of the Serverless WG. For example:
    - Events: define a common event format and API along with metadata. Some initial proposals can be found in the Serverless WG github repo.
    - Deployment: leveraging the existing CNCF members that are also serverless providers, start a new working group to explore possible small steps that can be taken to harmonize on a common set of function definitions, metadata. For example:
        - Application definition manifests, such as the AWS SAM and the OpenWhisk Packaging Specification.
- Function WorkFlow across different providers' serverless platforms. There are many usage scenarios that go beyond a single event triggering a single function and would involve a workflow of multiple functions executed in sequence or in parallel and triggered by different combinations of events + return values of the function in the previous step of the workflow. If we can define a common set of constructs that the developers can use to define their use case workflow, then they will be able to create tools that can be used across different serverless platforms. These constructs specify the relationship/interaction between the events and functions, relationship/interaction between functions in the workflow as well as how to pass information from one function to the next step function, etc. Some examples are AWS Step Function Constructs and Huawei's Function Graph/Workflow Constructs.
- Foster an ecosystem of open source tools that accelerate developer adoption and velocity, exploring areas of concern, such as:
    - Instrumentation
    - Debugability
- Education: provide a set of design patterns, reference architectures, and common vocabulary for new users.
    - Glossary of terms: maintain glossary of terms (Appendix A) in a published form and ensure that Working Group documents use these terms consistently
    - Use cases: maintain list of use cases, grouped by common patterns, creating a shared higher-level vocabulary. Supporting the following goals:
        - For developers who are new to Serverless platforms: increase understanding of common use cases, identifying good entry points

        - For Serverless providers and library/framework authors, facilitate consideration of common needs
    - Sample applications and open source tools in the CNCF GitHub repo, with a preference for highlighting the interoperability aspects or linking to external resources for each provider.
- Provide guidance on how to evaluate functional and nonfunctional characteristics of serverless architectures relative to CaaS or PaaS. This could take the form of a decision tree or recommend a set of tools from within the CNCF project family.
- Begin a process for CNCF outputs (for the suggested documents referenced above), such as from this Serverless Working Group and the Storage Working Groups, to live on as Markdown files in GitHub where they can be collaboratively maintained over time, which is particularly important given the speed of innovation in this space.

Appendix A: Glossary

This section defines some of the terms used in this whitepaper.

Backend-as-a-Service

Applications often leverage services that are managed outside of the application itself - for example, a remote storage service. This allows for the application to focus on its key business logic. This collection of 3rd party services is sometimes referred to as Backend-as-a-Service (BaaS). While these may be used from traditional compute platforms or from Serverless, it is important to note that BaaS plays an important role in the Serverless architecture as it will often be the supporting infrastructure (e.g. provides state) to the stateless Functions themselves. BaaS platforms may also generate events that trigger Serverless compute.

Cold Start

“Cold start” refers to the starting of an instance of the function, typically with new code, from an undeployed state.

Context

A Serverless platform typically provides a Context object as an input parameter when executing a Function, including Trigger metadata and other information about the environment or the circumstances around this specific Function invocation.

Data Binding

Function may require data bindings for long lasting connections to data such as

backend storage (such as mount points/volumes/object store), databases, etc. The data binding may include secure information such as secrets that can not be preserved within the function itself. The data binding may be used across several Function invocations.

Development Framework

The environment in which Functions are developed. This can be local (e.g. on a laptop) or in a hosted environment.

Event

The notification of something that happened.

Event Association

Mapping between event sources and the the specific Functions that are meant to be executed as a result of the event

Event Data

Information pertaining to the Event that occurred. See Event data and metadata for more information.

Event Source

Functions could be invoked through one or more event source types such as HTTP gateways, message queues, streams, etc. or generated based on a change in the system, such as a database write, IoT sensor activation or period of inactivity.

Function/Action

The code that is executed as a result of a Trigger.

Function Graph/Workflow

A developer's Serverless scenario usually involves definition of an Event, Function, Event-Function interaction and coordination between Functions. In some use cases, there are multiple Events and multiple Functions. A Function Graph/Workflow describes the Event-Function interaction and Function coordination. It provides a way for the user to specify what Events trigger what Functions, whether the Functions are executed in sequence or in parallel, transition between Functions, and how information is passed from one Function to the next Function in the workflow. Function Graphs can be viewed as a collection of workflow states and the transition

between these states, with each state having its associated Events and Functions. An example of the Function Graph/Workflow is AWS's step function.

Function Parameters

When a Function is invoked, the Runtime Framework will typically provide metadata about this particular invocation as a set of parameters (see Context).

Functions-as-a-Service

FaaS describes the core functionality of a platform to run functions provided by the end user on demand. It's a core component of a serverless platform, which includes the additional quality-of-service features that manage functions on behalf of the user including autoscale and billing.

Invocation

The act of executing a Function. For example, as a result of an Event.

Runtime Framework

The runtime environment/platform in which Serverless workflows are executed, Triggers are mapped to Functions, Functions hosting container resource and language package/library are dynamically provisioned, and those Functions are executed. Sometimes Runtime Frameworks will have a fixed set of runtime languages in which the Functions can be written.

Trigger

A request to execute a Function. Often Triggers are the result of an incoming Event, such as an HTTP request, database change, or stream of messages.

Warm Start

"Warm starts" refers to the starting of an instance of the function from a stopped (but deployed) state.

Appendix B: Additional References

The following references are provided for those looking for additional resources on serverless computing:

Serverless Architectures by Mike Roberts

Containers vs serverless - Navigating application deployment options by Daniel Krook

Serverless Computing: Current Trends and Open Problems by Ioana Baldini, et al.

Serverless: Background, Challenges and Future by Yaron Haviv

What is serverless good for? by Andreas Nauerz

Serverless Architecture: Five Design Patterns by Mark Boyd

How Two College Kids Built A Better Census by Stefanie Monge

7 AWS Lambda Tips from the Trenches by Mitchell Harris

Why The Future Of Software And Apps Is Serverless by Ken Fromm

The Serverless Guide authored by the community, curated by Serverless, Inc.

Microservice Orchestration for Serverless Computing by Cathy Zhang, Louis Fourie