EXECUTIVE SERIES

THE GORILLA GUIDE TO...®
EXPRESS EDITION

Serverless on Kubernetes

Joep Piscaer

INSIDE THE GUIDE:

- How Serverless is Changing the Game
- Getting the Most out of Kubernetes
- Fission and Kubernetes Offer Freedom of Choice

TAKE A QUICK WALK THROUGH THE IT JUNGLE!

THE GORILLA GUIDE TO...

Serverless on Kubernetes

Express Edition

AUTHOR
Joep Piscaer

Printed in the United States of America.

ACTUALTECH MEDIA

Okatie Village Ste 103-157

Bluffton, SC 29909

www.actualtechmedia.com

TABLE OF CONTENTS

CHAPTER 1

The Serverless Revolution

Serverless computing is a code execution model that abstracts away all the infrastructural plumbing underneath the code, allowing the developer to focus solely on their code. A serverless application is run by a platform that hides the implementation details from the user. These applications are made up of independent smaller services, many of which are event-driven, short-lived, and stateless.

Serverless is a level of abstraction of, and a decoupling from, the underlying infrastructure constructs. In a microservice architecture, monolithic applications are broken up into small services that can be developed, deployed, and scaled individually.

Serverless architectures are at the extreme end of the microservice spectrum, being even more fine-grained and loosely coupled. Serverless functions complement more traditional microservice and virtual machine (VM)-based approaches and regular third-party cloud services for event queueing, messaging, databases, and more.

With serverless, the organization or person writing the code doesn't have to care about the infrastructure underneath. As such, it's a form of utility computing.

Functions-as-a-Service (FaaS) is a technology for serverless computation. In a FaaS system, the unit of execution is a *function of code* written in a programming language (most FaaS systems support a wide range of languages). A developer specifies one or more functions and the conditions (events) under which those functions shall execute. Since it is serverless, the FaaS system automatically provisions resources to host and execute the functions when the specified conditions are met, and later tears them down when no longer needed. Since FaaS is the most widely used form of serverless, this guide will use those two terms interchangeably.

The serverless architecture is a boon for developers, allowing them to focus on just the code rather than all the surrounding plumbing like containers, deployment scripts, and monitoring. As is always the case,

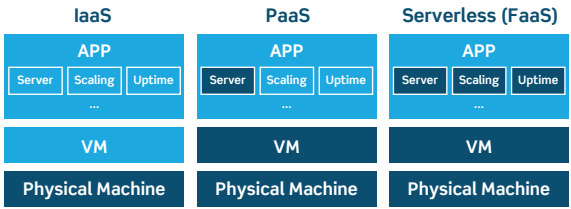


Figure 1: Comparing service architectures.

though, there are trade-offs in language support, code compatibility, performance, and cost.

One way to understand FaaS is to compare it to the popular 'IFTTT', or 'If This, Then That' web service that allows you glue together devices, web apps and more based on Triggers (this) and Actions (that).[1] IFTTT is popular in home automation scenarios, creating interaction between the environment (the weather, time of day) and IoT devices like thermostats, video door bells, and lighting systems.

Serverless serves similar "glue between services" use cases, and was first popularized in mobile app development to stitch together databases, authentication, and other commodity services that make up the back-

[1] https://ifttt.com/

A Clean Slate

A key characteristic of serverless is its statelessness. Functions are invoked from a clean state every time; any persistent state required for the function needs to be externally stored. This is similar to the twelve-factor app concept[1] for building Software-as-a-Service (SaaS) apps. Generally, functions will use a database, and a distributed cache or object store to store state across requests.

[1] https://12factor.net/

end of an app. It continues to be used in similar ways today, building IoT back-ends, APIs, and data processing pipelines.

Not only do they serve similar use cases, their architectures are even similar with triggers (this), and functions (that). We'll dive into specifics of the serverless architecture later.

Serverless Greases the DevOps Wheel

Although at first glance serverless may seem counter-intuitive for those in DevOps-culture organizations where development teams do their own operations, it really isn't. Serverless decouples the bits that make up the runtime (language-specific environments, containers, operating systems, VMs, physical hardware, networks, storage, and so on) and the tooling in the developer's pipeline (to build, test, and deploy code) from the actual code put in production, minimizing the

Waste Not

Looking at this from the lean software development perspective[1], we see that most steps in the developer pipeline are 'waste.' Waste, in this context, refers to technically necessary steps, like compiling or packaging, that provide no value to the customer. Even if those steps increase the quality of the code, like writing unit tests or deployment specifications, they provide little actual customer value.

[1] https://en.wikipedia.org/wiki/Lean_software_development

This means that it's important to reduce variation and other waste in the pipeline, as this leads to better customer value, delivered more quickly.

The comparisons to serverless computing are obvious: serverless removes a tremendous amount of waste from the developer pipeline by abstracting and standardizing.

Also, as any developer can tell you, shortening the pipeline from writing a line of code to putting it in production is a major advantage. Any optimization in the compilation, testing and packaging portions of the pipeline seriously enhances developer speed and efficiency. It also contributes to creating short and specific feedback cycles, which helps improve quality of the code, as the developer doesn't have to switch context between different features they might be working on.

operational part of the development workflow. It frees the coder from any concern about the plumbing.

This isn't to say that operations aren't done anymore, but rather that it's abstracted away from the developer. The function still has to be monitored, deployed, secured, supported, scaled, and debugged; these things are still happening, but they're merely packaged up as part of the service or platform.

While cost benefits are often cited as the major reason for using FaaS, it's reduction in lead time that may be the most exciting improvement. Instead of spending time on inefficiencies, product development teams can now focus more time on continuous experimentation, which will lead to more innovation and greater market advantages.

Serverless' Big Benefits

The biggest advantage of serverless computing is the clear separation between the developer and the operational aspects of putting code into production. This simplified model of "who does what" in the layer cake of the infrastructure underneath the code lets the operational folks standardize their layers (e.g., container orchestration and container images), while the developers are free to develop and run code without any hassle.

Because of the small and highly standardized surface area between development and operations, operational folks have greater control over what's running in production. Thus, they can respond more quickly to updates and upgrades, security patches, or changing requirements.

The use of container technologies like Docker and Kubernetes has increased developer velocity and significantly decreased the complexity of building, deploying,

In contrast to typical container-based microservices approaches, the container images – which are still operating systems and file systems at heart – aren't part of the developer pipeline and lifecycle. They are instead part of the stack the operations team controls and manages; this is a more natural fit, as they have necessary skills to manage the non-functional aspects of the infrastructure like security and performance.

and managing the supporting infrastructure as compared to VM-based approaches. But despite these advances, there is still a lot of relative friction from the developer standpoint in terms of going through required steps that add no direct value to their workflow, like building a container for every new code release or managing auto-scaling, monitoring, and logging. Even with a perfect pipeline, these jobs become naturally inert, requiring additional work to change them for a new release, software version upgrade, or security patch.

Although the default operating model is one in which each group stays within their set boundaries, there are possibilities to cross those lines. A typical example is when

Flexible Pricing

The industry pricing model for serverless function execution is the same across the board for major cloud providers. Pricing varies with the amount of memory you allocate to your function. This is a pricing overview across Amazon, Microsoft and Google from October 2018:

- Requests: $0.20 - $0.40 per million requests
- Compute: $0.008 - $0.06 per hour at 1 GiB of RAM
- Data ingress: $0.05 – $0.12 per GB
- API Gateway: $3.00 – $3.50 per million requests

These pricing models are a direct motivation to optimize a function for performance: paying for execution time directly correlates cost with performance. Any increase in performance will not only make your customers happy, but also reduce the operational cost.

This is a radically different pricing approach than we've seen with VMs and containers: paying for only what you actually use. When a function is idle, you're not paying at all, which negates the need to estimate resource usage beforehand; scaling from zero to peak is done dynamically and flexibly.

a developer needs additional package dependencies, like libraries, in the execution environment. In a container-based microservices approach, this would have been the developer's problem; in the serverless approach, it's part of the solution, abstracted away from the developer. The containers that execute the functions are short-lived, automatically created and destroyed by the FaaS platform based on runtime need.

This leads to a shorter pipeline and fewer objects (like containers) that need to be changed with a new code release, making deployments simpler and quicker. Since there's no need to completely rebuild the underlying containers, as most new code releases are loaded dynamically into existing containers, deployment time is significantly shortened.

In addition, compiling, packaging, and deployment are simple compared to container and VM-based approaches, which usually force an admin to redeploy the entire container or VM. FaaS only requires a dev to upload a ZIP file of new code; and even that can be automated, using source version control systems like Git. There's no configuration management tooling, rolling restart scripts, or redeployment of containers with the new version of the code.

Time isn't the only thing that's saved, either. Because of the fine-grained level of execution, FaaS services are metered and billed per millisecond of runtime or per number of requests (i.e., triggers). Idle functions aren't billed.

In other words, you don't pay for anything but code execution. Gone are the days of investing in data center space, hardware, and expensive software licenses, or long and complex projects to set up a cloud management or container orchestration platform.

This also means that you're not stuck with the amortization of investments or long-term contracts, but free to change consumption monthly or even daily. This allows teams to change direction or try something new on an extremely small scale, with similarly small operational costs associated with experimentation. This fosters a culture of trying new things, taking small steps, and learning from mistakes early, which are basic tenets of any agile organization.

As you can see, serverless is a great option for dynamic applications. The provider automatically scales functions horizontally based on the number of incoming requests, which is great for handling high traffic peaks.

CHAPTER 2

The Serverless Landscape

The Players

Even though serverless feels like the hot new thing, it's actually not new; it's been around for about five years. Node.js is the predominant language in the field, but Java, Go, Python, and C# are also popular. Different platforms provide different ways to invoke other languages indirectly, too.

As you might expect, all big public cloud vendors have a serverless play: Amazon has Lambda, Google has Cloud Functions and Microsoft has Azure Functions.

The landscape is much larger than just the big service offerings, though, as **Figure 2** shows. There are many frameworks, cloud services, and on-premises platforms available, and the landscape is evolving quickly. This gives you choices for building out your serverless infrastructure. Let's start with an overview of the ones you're most likely to know about.

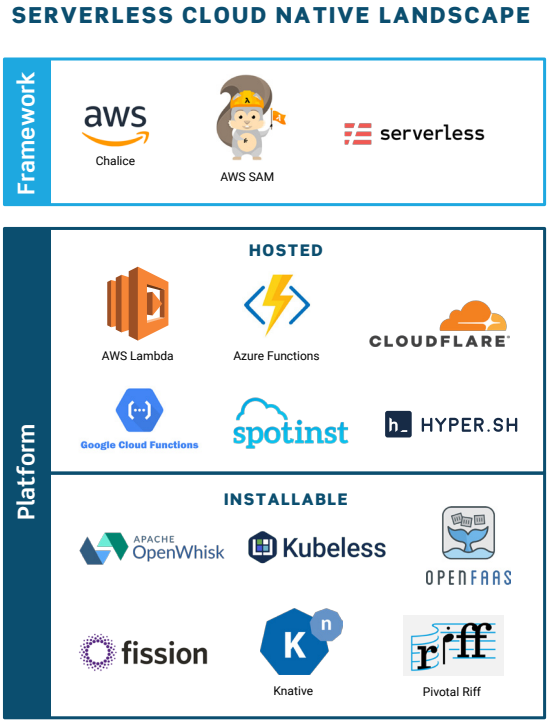


Figure 2: A slice of the growing serverless ecosystem.

AWS Lambda

Amazon launched Lambda in 2014, which was the first commercially available serverless platform. It's part of the Amazon Web Services (AWS) cloud computing port-

folio, and is tightly integrated in that ecosystem. It runs in the AWS cloud, with no option to run on-premises and only limited options for running locally on a developer's machine. Future offerings may include the ability to run Lambda functions closer to the edge. There's also a serverless database option, called Aurora Serverless.

Lambda functions can be triggered by numerous events, including:

- Database changes
- File and object storage changes
- Messages in a publish/subscribe queue
- Scheduling
- Authentication
- HTTP requests (via an API Gateway)

This is only a sample, as there are many more. Use cases include image processing and object uploads to S3, updates to DynamoDB tables, responding to website clicks, or responding to sensor data from IoT-connected devices.

Lambda is backed by performance objectives. AWS's goal is to start a Lambda instance within 100 milli-

seconds of an event, but there are limits to the total duration of a function; it's currently capped at fifteen minutes. Although Lambda functions are elastic and scale automatically, they're also limited to 1,000 concurrent executions by default in a given region, per account. This limit can be easily reached, especially when combining production and testing.

Microsoft Azure Functions

Microsoft's Azure Functions is a relatively young service, but very similar to Lambda. Since it's part of the Azure ecosystem, the underlying infrastructure runs Windows, not Linux. Besides some unique language support (C#, F#), there are two major selling points for Azure Functions.

First is the ability to run on Azure Stack, which runs in the data center. This puts Azure Functions much closer to existing on-premises workloads, which many enterprises still run (and will run for years to come). This makes Azure Functions a great use case for serverless developers in those organizations that run the majority of their workloads on-premises.

The second important distinction is tight integration with Visual Studio, Microsoft's Integrated Development Environment, or IDE. This integration offers the ability to debug functions locally from a cloud-triggered

event. As any developer will recognize, being able to breakpoint a remotely running function is very useful.

Azure Functions also offers the ability to keep functions in hot standby, mitigating cold startup latency problems. Otherwise, functions have the same type of runtime limitations as AWS (five minutes, by default) and concurrent executions (200 concurrent executions per function in a region).

Google Cloud Functions

Google's Cloud Functions (GCF) is the newest service of the three big cloud providers, although the PaaS-like App Engine has been around since 2008. The biggest difference between GCF and the others is its trigger support, which is focused on Google's Pub/Sub messaging bus, the de facto standard for inter-service communication in the Google world.

In many ways, GCF's very similar to Lambda, but it remains a fairly simple alternative. Google has a Firebase-integrated version of GCF to cater to mobile backend developers.

The Alternatives

Besides the "big three," there are other serverless options, broadly divided into three categories:

- **Other Vendors.** IBM and Oracle have FaaS services in their public clouds, too. These are similar to the three we've discussed, but aren't as widely used. There are also a number of vendors in the twelve-factor camp, like Auth0, that offer serverless frameworks and services.
- **Edge.** A number of edge computing specialists, like CloudFlare, have begun to offer FaaS services at the edge. These are aimed at use cases that need close proximity to users and devices, like IoT and web. Lambda has a similar offering called Lambda@Edge, running in the CloudFront content delivery network.
- **Framework.** Rather than being a managed service, these frameworks fall under their own umbrella. They offer freedom on where to run and are generally not priced in via the consumption model. They are mostly free and open source, or at least, not tied to a cloud vendor's ecosystem. These frameworks are infrastructure-agnostic and can be used as a building block by a service provider or as part of an existing on-premises technology stack. These

run on top of container platforms like Docker and Kubernetes. Examples of these include OpenFaaS, serverless.com, and Fission.

The frameworks have several key advantages over the commercially available services, including greater control over both the infrastructure it runs on and pricing. The rest of this book will put the spotlight on one particular framework and the advantages it can provide you in your serverless journey: Fission, from Platform9.

Introducing Fission

Fission is an open source, Kubernetes-native serverless functions framework with support for public, private, and hybrid clouds. Support for Kubernetes enables

Smooth Starting

Getting started with Fission is easy, needing just a couple of Helm or kubectl commands to deploy it on a laptop, in an on-premises Kubernetes cluster or in a cloud service. The installation steps can be found on the project website.[1]

[1] https://docs.fission.io/latest/installation/

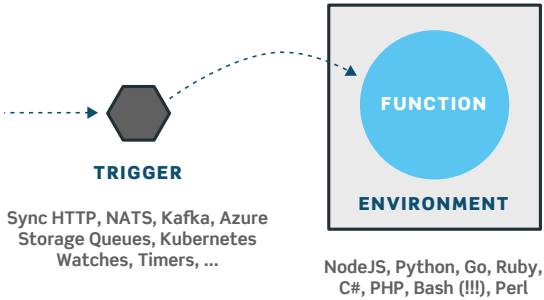


Figure 3: The core pieces of Fission.

the portability of Fission functions with the ability to create once and deploy anywhere for consistency in code development. Accelerate your software delivery pipeline without sacrificing quality.

Fission is made up of three core concepts:

1. Functions
2. Triggers
3. Environments

An illustration of the relationship can be seen in **Figure 3**.

Functions

A function is something that Fission executes. It's the code a developer has written; for instance, a piece of business logic. It adheres to certain technical characteristics commonly found in event-driven programming.

Here's a simple example of a "Hello, world!" function written in NodeJS:

```
module.exports = async function(context) {
  return {
    status: 200,
    body: "Hello, world!\n"
  };
}
```

Functions are generally written in an asynchronous way, so they can run in parallel for easy horizontal scaling. They're also stateless by nature, assuming anything in memory on local disks can be deleted. Any persistence is stored externally to the function, in a file system, object store, or database.

Environments

Environments are the language-specific parts of Fission. An environment contains just enough software to build and run the function. It consists of a container with the language runtime, a web server, and fission-specific parts that allow functions to load dynamically.

Fission supports many languages out of the box. A new language, or a customization of an existing language environment, is as easy as creating or modifying the underlying containers.

Trigger

Functions are invoked on the occurrence of an event; a trigger is what configures Fission to use that event to invoke a function. In other words, a trigger is a binding of events to function invocations.

There are a number of types of triggers supported by fission:

1. HTTP (specific URL or endpoint)
2. Time (cron)
3. Message Queue (based on queue topic subscription)
4. Kubernetes Watch (watches for changes in Kubernetes objects)

For HTTP requests, the fission router handles the mapping of triggers to functions, keeps track of which actual containers run a given function, and forwards requests (and sends responses back) to and from functions.

Technical Overview

You've seen the functional concepts of Fission; now let's look under the hood. Fission is made up of a set of microservices running on Kubernetes.

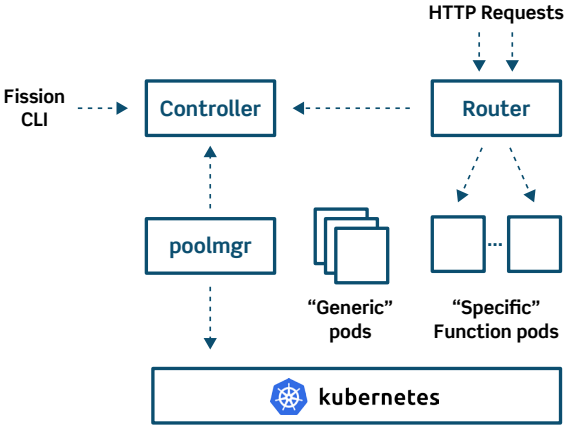


Figure 4: Fission architecture.

Fission provides CLI tools for generating these specification files, validating them, and applying them to a Fission installation. Note that running *apply* more than once is equivalent to running it once: if the desired state as defined in the configuration is reached, it won't change.

Fission Services

Fission consists of various services, each running in containers (see **Figure 4**):

- **Controller.** This is the brains of Fission, and it keeps track of functions, HTTP routes, event triggers and environment images. It serves the Fission API to the client.
- **Pool Manager** and other executors. It manages pools of idle environment containers, manages the loading of functions into these containers dynamically, and kills idle function instances. A second type of executor enables automatic horizontal scaling of functions.
- **Router**, which handles requests and routes them to function instances. It has a cache of request and service mappings to route traffic to an existing container, or request an instance from the Pool Manager where needed.

- **The Fission CLI**. This is the user interface, used to interact with the fission system. It uses a declarative, file-based approach for Fission objects, like functions and environments. This way, you can track the Fission specifications along with the source code in the version control system. This allows Fission to be integrated seamlessly into the developer workflow and existing (CI/CD) pipelines.

CHAPTER 3

Serverless in the Cloud and On-Premises

Kubernetes and Serverless: Like Peanut Butter and Jelly

Kubernetes is an open source solution for automating deployment, scaling, and management of containerized applications. And running on top of Kubernetes means it is very portable, so it will run anywhere Kubernetes runs: on your laptop, in a public cloud, in an on-premises data center, or in a managed Kubernetes service provider.

As organizations start to adopt cloud services, they will start using a variety of services, from different vendors, in conjunction with existing workloads in the on-premises data center. Having a single serverless experience across all those is good for the developer, as it means doing away with the various services they would otherwise have had to learn. This enables the developer or dev team to get things done more quickly, deliver higher quality code, and minimize additional training efforts.

The Fission framework handles container lifecycle duties like creating and building VMs, abstracting away most of the complexity of operating a Kubernetes environment.

Fission itself runs as a set of microservices on top of Kubernetes.

A well-designed serverless-based application architecture is inherently scalable, especially when deployed to an elastic capacity provider, such as the public cloud or a large IaaS provider. By utilizing the intelligence in the underlying Kubernetes platform, functions can automatically scale up or down horizontally.

Avoiding Lock-In

Most current serverless offerings are services: they're part of a portfolio of technologies in one of the major clouds, and run as part of that ecosystem. These commit customers to the cloud provider's ecosystem, forcing them to use cloud-specific services. This means that functions in one cloud aren't portable to another cloud provider, requiring refactoring when moving a function to a different provider.

Other providers try to tie you into their ecosystem in subtle ways by nudging users to use ecosystem-specific services and technologies. Fission takes the opposite

Lock-in like this happens in subtle ways; for instance, being forced to use the packaged (and often monetized) monitoring solution. This almost defeats the purpose of Kubernetes, which is a freely available technology that works across clouds, in on-premises environments and locally. Fission prevents this lock-in and dependency, and promotes decoupling, re-use of code, and portability, all of which reduce friction during the lifetime of the function. This allows developers to modernize applications, even when using on-premises infrastructure.

route, offering all the benefits of serverless without any of the cost or lock-in. It's multi-cloud, multi-tool friendly, enabling developers to choose the best tool for the job, instead of forcing the default options in a given cloud ecosystem. Fission gives maximum freedom in the developer continuous integration (CI) and continuous delivery (CD) pipeline, as well as production tooling such as monitoring and tracing.

The Fission advantages go beyond lock-in, too. For example, many organizations with existing

investments in private data centers end up with spare server capacity, like CPU and memory; that's just the way physical hardware is bought. Fission can run on this idle and already paid-for server capacity, effectively giving you a free FaaS platform. Free serverless is a major advantage of the pay-per-use model of public cloud providers, which can become expensive in a hurry.

Even for on-premises Kubernetes environments with little to no spare capacity, Fission is a good fit economically, as expanding the data center with just additional server capacity (leveraging other data center investments like network and storage) is almost certainly cheaper than a public FaaS service.

Running your serverless alongside existing containers and integrating with container-based tooling – including monitoring, logging, and so on – eases the operational burden and simplifies the adoption of a serverless framework. This allows Fission functions running on Kubernetes to use services like message queues and databases running on the same platform.

Most applications are a hybrid of functions and containers. It makes sense to run different components of an application physically near each other, where possible. Not only does this improve latency, it also

eliminates the cloud vendors' notoriously expensive ingress and egress fees.

Freedom of Choice with Kubernetes and Fission

Kubernetes is taking the world by storm, quickly replacing virtualization stacks and IaaS services with container-based approaches. Kubernetes' deployment experience had a rocky start, being notoriously difficult to install and configure, but most of that has been overcome; many public cloud vendors and service providers now offer a hosted and managed Kubernetes service that negates most of this complexity. Examples include Amazon EKS, Google GKE, and Platform9 Managed Kubernetes.

For functions running on Fission, it's easy to take advantage of the rich Kubernetes ecosystem and the wide range of data services it supports, like message queues and databases, as well as integrating with underlying infrastructure components for software-defined storage and networking.

Instead of forcing tenants to use specific monetized services, running serverless on Kubernetes provides the option of using free and open source tooling instead. This means running free and open source data

services and middleware (databases, message queues, key/value stores), web servers, and more. This is why Fission also integrates with open source projects like Prometheus, which we'll talk about a little later.

Faster, Easier Development

The Kubernetes-based approach enables the ability to extend Fission's language and runtime support to anything that runs in a container. Adding a new language is relatively easy; you can create a new container image or modify one of the existing environments to suit your needs.

Fission supports many languages out of the box, including:

- Node.js
- Python
- Go
- Ruby
- Java
- C# / .NET
- Binary (for executables or scripts)
- Perl
- PHP
- And more

Deployment and Operations

Fission supports declarative deployments using build specs. These specs describe Fission resources like functions and triggers and allow developers to deploy functions anywhere. This helps manage the complexity of deployments across different environments, making sure that a function is deployed across environments consistently.

These build specs use Kubernetes' custom resources and are stored as configuration files, which can be checked into version control. In a later release, Fission will support automatic deployment from the source repository.

In addition, Fission automatically generates the initial configuration. The initial configuration is a ready-to-use template for further customization, saving the developer time initially while still being flexible.

Fission also saves time once the code is written. The anxiety-inducing moment for every developer is deploying to production. Automated canary deployments help manage that risk by sending only a small amount of traffic to the new version initially. As trust is gained, more traffic is pushed to the new version, ultimately removing older versions from the production roster. Conversely, if it is found that the end-users are

experiencing issues, Fission will re-route users to older, more stable code so the issue can be resolved.

Fission has configuration settings for the distribution of traffic between versions (and the shift over time) and the threshold error rate.

Monitoring and Metrics

Monitoring is a traditionally tricky area for FaaS, because of the short-term nature of containers and functions, and the amount of engineering cloud providers had to put into a monitoring solution for *their* version of FaaS.

For many FaaS operations, this means that monitoring solutions are very basic, and don't integrate into more traditional third-party monitoring solutions or open APIs. But because of Fission's integration with Kubernetes and service meshes like Istio, much of the grunt work has already been done. Fission has integrated with Kubernetes monitoring, resulting in a first-class monitoring experience for FaaS.

Fission aggregates function logs using Fluentd; the logs are then stored in a database, providing a lightweight and searchable solution.

Fission is also integrated with Prometheus, the de facto standard metrics system. Fission automatically tracks the number of requests (function call count), timing (execution time and overhead) and success/failure rate metrics, response size, and error codes for all functions. These are fed into Prometheus automatically, without adding any code to the functions. In addition, it adds contextual information (cold vs. hot starts) to these metrics to allow better interpretation.

Balancing Cost and Performance

Ideally, functions that don't run cost nothing. But we want every service to respond quickly, even if they're called for the first time. Since costs for disk and memory vary widely, there are tradeoffs in performance vs. cost to be aware of.

For instance, how do you make sure the cost for idle functions is small, while keeping latency low for often-used functions?

This is the "cold start performance" problem. All FaaS services experience this problem, and each solves it with a different approach.

Fission's approach is to provide a tunable cost-performance tradeoff. It also provides a pool of pre-warmed environments for functions.

This is part of a larger issue: That the actual cost of public cloud FaaS services depend heavily on the usage pattern of functions. In some cases, running the same code in containers or even VMs can be much cheaper than running them as serverless functions. Also, cost across clouds vary. In many pay-per-use models, which is popular with the public cloud, there is a real danger of costs getting out of hand, even when compared to containers or VMs.

By contrast, Kubernetes-based FaaS, regardless of where it's running, has a simpler cost model. You pay per-container, per-VM, or even per-server for capacity. If you can then use spare resources sitting unused in previously-purchased hardware, you can change the cost structure completely.

In this scenario, optimizing for cost doesn't mean scaling back resources, limiting performance, and increasing latency; instead, it means using what you already have in a smart way, providing new ways to develop code without breaking the bank.

CHAPTER 4

Serverless in the Real World

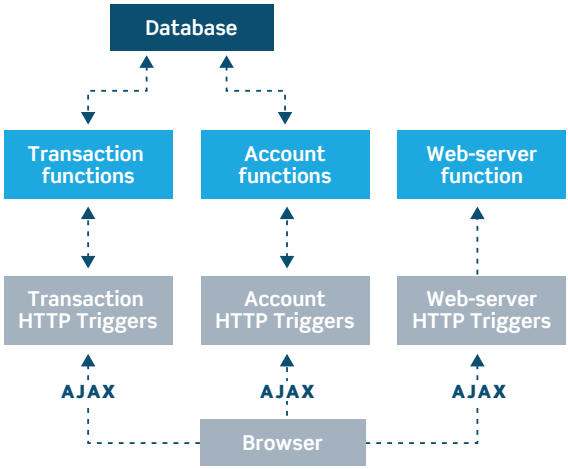
In this last part of the guide, we'll show some practical examples of serverless. The first one is a common banking application using web and API technologies. It uses a database running on Kubernetes, CockroachDB, with various functions in Fission interacting directly.

Example 1: Banking Site

Each of the actions on the site, like depositing, withdrawing, and transferring money between accounts and balances are functions running on Fission, triggered by HTTP actions.

Fission lets you easily and quickly create the various functional areas of the site, such as creating an account and various banking activities as functions.

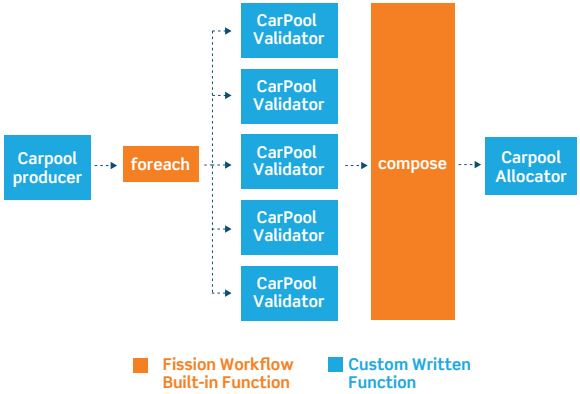
1. When a user visits a web page, the browser hits the web server's HTTP trigger to get the HTML files.
2. Any operations on the web page send an AJAX HTTP request to backend RESTful API functions.



3. Once a function receives requests, it interacts with the database, which launched in a different name-space, to get/insert/update records.
4. After database operations complete, the function response user requests with HTTP code and the message body.

You can run this sample use case on any Fission environment.[2]

[2] https://github.com/fission/fission-bank-sample



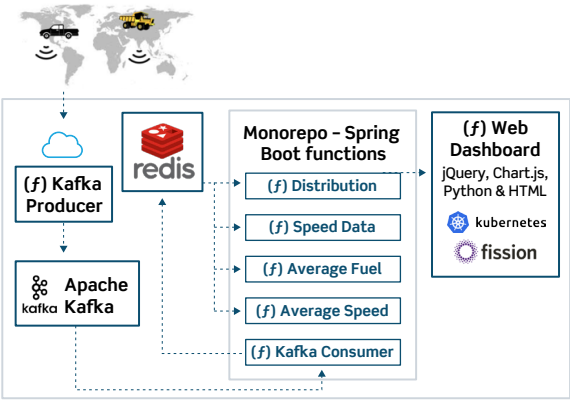
Example 2: Carpool

The carpool application is a great example of combining multiple functions into a single workflow, parallelizing certain functions to optimize the flow.

The application tries to match a car owner offering seats in their car with riders looking for one or more seats in a shared carpool.

As with the previous example, you can run this sample use case on any Fission environment.[3]

[3] https://github.com/fission/fission-workflow-sample



Example 3: Internet of Things

This use case shows how a serverless app consumes sensor data from IoT vehicles to figure out the most optimal route.

As with the others, you can run this sample use case on any Fission environment.[4]

The Unstoppable Force

Serverless is an unstoppable force that's changing the way developers put code into production. It allows them to focus on what's important: developing business logic. It abstracts everything the developer shouldn't have to

[4] https://github.com/fission/fission-kafka-sample

worry about, increasing their velocity, simplifying the pipeline, and shortening the feedback loops.

Organizations adopting serverless are able to adopt to changing requirements more quickly and make it easier to do more and smaller experiments, to quickly discover what works and what doesn't. This leads to better quality code, delivered faster.

The unified serverless experience of Fission allows functions to run locally, on the developer's laptop, in the data center, or in the cloud by leveraging the power of the Kubernetes platform.

The ability to run functions in spare compute capacity in the on-premises data center has more than just cost benefits. It also minimizes network latency and associated network bandwidth costs.

Fission, as an open alternative to cloud-specific FaaS services, has broad language support, but is fully customizable and flexible to run any code you need. It doesn't lock you into one single cloud ecosystem, but gives you the freedom to choose.

If you're considering moving to serverless, you owe it to yourself to give Fission a whirl.