Practical Spring Cloud Function

Developing Cloud-Native Functions for Multi-Cloud and Hybrid-Cloud Environments

Banu Parasuraman

Apress®

Practical Spring Cloud Function

Developing Cloud-Native Functions for Multi-Cloud and Hybrid-Cloud Environments

Banu Parasuraman

Apress®

Practical Spring Cloud Function: Developing Cloud-Native Functions for Multi-Cloud and Hybrid-Cloud Environments

Banu Parasuraman
Frisco, TX, USA

ISBN-13 (pbk): 978-1-4842-8912-9 ISBN-13 (electronic): 978-1-4842-8913-6
https://doi.org/10.1007/978-1-4842-8913-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Laura Berendson
Coordinating Editor: Gryffin Winkler
Copy Editor: Kezia Endsley

Cover designed by eStudioCalamar

Cover image by Aamyr on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I would like to dedicate this book to my wife Vijaya and my wonderful children Pooja and Deepika, who stuck with me through the trials and tribulations during the writing of this book. I also dedicate this to my mom, Kalpana Parasuraman.

Table of Contents

About the Author

Banu Parasuraman is a cloud native technologist and a Customer Success Manager (CSM) at IBM, with over 30 years of experience in the IT industry. He provides expert advice to clients who are looking to move to the cloud or implement cloud-native platforms such as Kubernetes, Cloud Foundry, and the like. He has engaged over 25 select companies spread across different sectors (including retail, healthcare, logistics, banking, manufacturing, automotive, oil and gas, pharmaceuticals, and media and entertainment) in the United States, Europe, and Asia. He is experienced in most of the popular cloud platforms, including VMware VCF, Pivotal PCF, IBM OCP, Google GCP, Amazon AWS, and Microsoft Azure. Banu has taken part in external speaking engagements targeted at CXOs and engineers, including at VMworld, SpringOne, Spring Days, and Spring Developer Forum Meetups. His internal speaking engagements include developer workshops on cloud-native architecture and development, customer workshops on Pivotal Cloud Foundry, and enabling cloud-native sales plays and strategies for sales and teams. Lastly, Banu has numerous blogs on platforms such as Medium and LinkedIn, where he promotes the adoption of cloud-native architecture.

About the Technical Reviewer

Manuel Jordan Elera is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations. Manuel won the Springy Award 2013 Community Champion and Spring Champion. In his little free time, he reads the Bible and composes music on his guitar. Manuel is known as `dr_pompeii`. He has tech-reviewed numerous books, including *Pro Spring MVC with Webflux* (Apress, 2020), *Pro Spring Boot 2* (Apress, 2019), *Rapid Java Persistence and Microservices* (Apress, 2019), *Java Language Features* (Apress, 2018), *Spring Boot 2 Recipes* (Apress, 2018), and *Java APIs, Extensions, and Libraries* (Apress, 2018). You can read his detailed tutorials on Spring technologies and contact him through his blog at `www.manueljordanelera.blogspot.com`. You can follow Manuel on his Twitter account, `@dr_pompeii`.

Acknowledgments

It has been a great privilege to write this book and help you understand real-world implementations of Spring Cloud Function. Thank you for reading it.

After my presentation at SpringOne 2020, I received a message on LinkedIn from Steve Anglin at Apress. Steve asked me if I would be willing to write a book about Spring Cloud Function. I was a bit hesitant at first, given that I was occupied with many client engagements, which were taking up most of my work hours. I was worried that I would not do the subject justice, due to my preoccupation with my clients. But after a long contemplation and a heartfelt discussion with my family, I decided to take it on.

I want to thank Steve Anglin, Associate Editorial Director, for reaching out to me and providing me this opportunity to write a book on Spring Cloud Function.

Mark Powers, the Editorial Operations Manager, was instrumental in helping me bring this book to close. With his incessant prodding and nudging, he helped me reached the finish line. Thanks, Mark.

Manuel Jordan, the technical reviewer, was immensely helpful. His comments kept me honest and prevented me from cutting corners. He helped improve the quality of the solutions that I present in this book. Thanks, Manuel.

I also want to thank Nirmal Selvaraj and others at Apress, who worked to bring this book to fruition.

This book would not be possible without the help of my wife Vijaya and daughters Pooja and Deepika, who provided the much-needed emotional support through this journey.

Introduction

I entered the field of Information Technology (IT) 25 years ago, after spending time in sales and marketing. I was an average programmer and was never into hardcore programming. During my early life in IT, I worked as part of a team that built a baseball simulator for the Detroit Tigers. I helped build a video capture driver for that simulator using C++. Even though this was a great project with a lot of visibility, it was never my real passion to be a hard-core programmer.

I soon gravitated toward solution architecture. This seemed to perfectly tie my marketing skills to my technology skills. I began looking at solutions from a marketing lens. This approach formed the basis for writing this book. Because, what good is a technology if we do not know how to apply it in real life?

Functional programming was an emerging technology. Cloud providers such as AWS, Google, and Azure created serverless environments, with innovations such as Firecracker virtualization techniques, that allowed infrastructure to scale down to zero. This allowed customers to derive huge cost savings by not paying for resources that were not in use and subscribing to a pay-per-use model.

Initially, development of these functions that run on serverless environments were built on either NodeJS or Python. These functions were also vendor-specific. Spring.io developed the Spring Cloud Function framework, which allowed the functions to run in a cloud-agnostic environment. The focus was on the "write once, deploy anywhere" concept. This was a game changer in the cloud functions world.

Prior to writing this book, I was a staunch evangelist of Pivotal Cloud Foundry and Kubernetes. I promoted writing code that was portable. When Knative came into being in 2018 as a joint effort between IBM and Google, I was excited. Knative was designed as a serverless infrastructure on top of Kubernetes and made the serverless infrastructure portable. Combining the power and portability of Spring Cloud Function and Knative, you have a true portable system with zero scale-down capabilities.

This was something that I wanted to write and evangelize about. But I felt that writing about the technology and how it worked would not be that exciting. I wanted to write about how people could use this technology in the real world.

In this book, you will see how to program and deploy real-life examples using Spring Cloud Function. It starts with examples of writing code and deploying to AWS Lambda, Google Cloud Function, and Azure Function serverless environments. It then introduces you to the Knative on Kubernetes environment. Writing code and deploying is not enough. Automating the deployment is key in large-scale, distributed environments. You also see how to automate the CI/CD pipeline through examples.

This books also takes you into the world of data pipelines, AI/ML, and IoT. This book finishes up with real-world examples in oil and gas (IoT), manufacturing (IoT), and conversational AI (retail). This book touches on AWS, the Google Cloud Platform (GCP), Azure, IBM Cloud, and VMware Tanzu.

The code for these projects is provided on GitHub at `https://github.com/banup-kubeforce`. It is also available at `github.com/apress/practical-spring-cloud-function`. This allows you to get up to speed on the technologies. So, after completing this book, you will have hands-on experience with AI/ML, IoT, data pipelines, CI/CD, and of course Spring Cloud Function.

I hope you enjoy reading and coding this book.

CHAPTER 1

Why Use Spring Cloud Function

This chapter explores Spring Cloud Function using a sample use case—an HRM (Human Resources Management) system. The focus is on systems that reside in an enterprise. The chapter touches on the FaaS (Functions as a Service) concept and explains how it is gaining momentum in the enterprise. The chapter also digs deeper into its implementations in the cloud. You will learn about some of the portability issues present at the code and container level and read about concepts such as Knative on Kubernetes, which includes container portability. You will also learn about some high-level implementations of Spring Cloud Function on AWS, GCP, Azure, VMware Tanzu, and Red Hat OpenShift.

1.1. Functions as a Service (FaaS)

FaaS is a revolutionary technology. It is a great boon for developers and businesses. FaaS allows businesses to adapt to rapidly changing business needs by enabling their development teams to develop products and features at a "high" velocity, thereby improving their Mean Time To Market (MTTM). Developers can develop functions without worrying about setting up, configuring, or maintaining the underlying infrastructure. FaaS models are also designed to use just the right quantity of infrastructure and

B. Parasuraman, *Practical Spring Cloud Function*,
https://doi.org/10.1007/978-1-4842-8913-6_1

compute time. They also can be scaled to fit exact demand, by focusing on billing for the number of invocations as compared to billing for uptime. FaaS has two parts, as shown in Figure 1-1.

- The function code encapsulates the business logic in any language, such as Java, C#, Python, Node, and so on.
- The underlying container hosts an application server and an operating system.

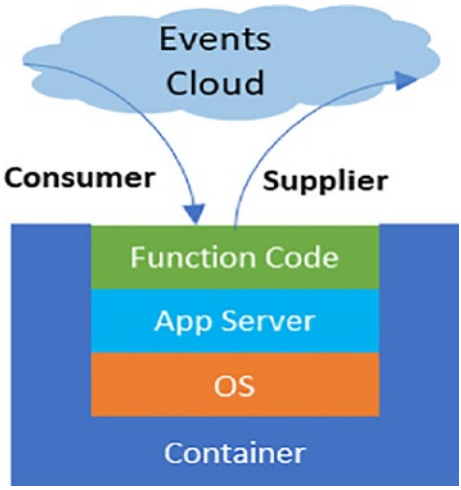

Figure 1-1. *FaaS component architecture*

1.1.1. Implementation of an Enterprise Application

Imagine all the infrastructure needed to run a single payroll application on the cloud. This application may consume only 16GB of RAM and eight vCPUs, but you are charged continuously for the entire duration that the application is active. Using a simple AWS pricing formula, this works out to around $1,000 per year. This cost is for the whole time the application

is hosted and active, regardless of use. Of course, you can cost-justify it through a TCO (Total Cost of Ownership) calculation, which helps you determine how your application can bring in revenue or value that compensates for the expense. This revenue-generation model is more suitable to applications that generate revenue for the company, such as an ecommerce site. It is more difficult to prove the value that a supporting application, running in the backend of an enterprise, brings to a company.

1.1.2. Migration ROI for a Portfolio of Application

The value equation gets more complex if you plan to migrate an extensive portfolio of apps in your enterprise.

Let's for a moment assume, as a CTO or CIO of a company, you have a portfolio of about one thousand applications that you plan on migrating to the cloud. The key factors to consider, among the many, include:

- What is the current infrastructure supporting the apps?
- What is the utilization of these apps?

The utilization of apps is an essential factor in determining the value of the application. Consider this—after analyzing the utilization of apps, you find that this portfolio includes the following distribution:

- 10% with 80% utilization
- 40% with 50% utilization
- 50% with 20% utilization

If you calculate the billing cost using an AWS cost calculator, you see that you will spend $1 million per year. This spend is for applications that are critical and highly utilized, as well as for applications that are minimally utilized. This cost is due to the fact that the cloud providers charge for the entire duration the application is active and consuming the infrastructure. The key here is that the infrastructure is fully allocated

for the application's life. Imagine how much you could save if the infrastructure was allocated only for the duration that the application was active and serving. This would be a great cost and resource saving approach. Cloud providers have thought through this because they also faced the pressure of finite infrastructure and considered the time needed to provision additional infrastructure.

1.1.3. The Serverless Functions Concept

To work around the problem of finite infrastructure utilization, AWS created Lambda serverless functions. This was a genius invention. Subscribers to this service pay only for the time the application is invoked. The infrastructure is unallocated when it is not invoked. This way, AWS can save on infrastructure by repurposing the infrastructure for other needy applications while transferring the cost savings to the customer. This is a win-win. It's worth considering whether you can apply this same approach to all the enterprise applications in your company today. You would be able to save a lot of money. Also, if you were to bring this technology to the datacenter, you would be able to reap the benefits that AWS realized. Isn't this wonderful?

1.1.4. Applying the Serverless Functions Concept to an Enterprise Application

Let's dig deeper into the concept of functions and how AWS realizes the magic of infrastructure savings. Functions are tiny code pieces with a single input and a single output, and a processing layer (a predicate) acting as the glue. Compare this to enterprise apps, which are designed to do many things. Take a simple payroll system, for example. A payroll system has multiple input interfaces and multiple output interfaces. Here are some of those interfaces:

- Timecard system to get the hours employees worked in a month
- Performance evaluation system
- Peer feedback system
- Inflation adjustment calculator system
- The outgoing interface to the IRS
- The outgoing interface to the medical insurance provider
- An outgoing interface to the internal web portal where employees can download their paystubs

Running this payroll application is not trivial. I have seen such a payroll system use the following:

- Fourteen dedicated middleware application servers
- Two RDBMS database stores
- Two integration tools such as message queues and FTP
- Four dedicated bare-metal servers, with each server configured with 128GB RAM, 32 CPUs, 4TB of HDD, 10TB of vSAN, and the like

The key factor in determining whether this application can be hosted on a serverless functions infrastructure like Lambda is the time it takes for the application to boot up (the startup time or cold start) and the time it takes for the application to shut down (the shutdown time). The faster the startup and shutdown times, the larger the cost savings.

It is also important that these times be quick so that they don't cause disruptions. If you were to research the startup times for large enterprise applications like the payroll application, you would find that it is not pretty. An average startup time is around 15 minutes for all components to

come up and another 15 minutes for the application to come down. This would not fly. Imagine if you deployed this application to an AWS Lambda serverless function. Thirty minutes of downtime for each invocation? This will not work. Your users would abandon the application entirely. As you can see, large applications cannot benefit from resource release and resource reassignment because they take a long time to start up and shut down, which would impact the general operation of the application.

Can you make this large payroll application behave in an expected way for serverless functions? The answer is yes. A lot of refactoring is required, but it can be done.

Serverless Function in the Cloud

All cloud providers have now incorporated the serverless functions into their infrastructure offerings. AWS has Lambda Functions, Google has Cloud Functions, and Azure has Azure Functions. These providers, in the quest for making their customers captive, made sure to introduce proprietary elements into their environments. The two components that are essential to run the functions are:

- Serverless function code that serves the functions
- Serverless infrastructure (containers) that supports the code

Why Is It Important for Serverless Functions to be Non-Proprietary?

Enterprises are gravitating toward a multi-cloud, hybrid-cloud approach to their cloud strategy. As you can see in Figure 1-2, the survey of 3,000 global respondents indicated that 76 percent already work in a multi-cloud environment. This means they are not bound to one single cloud provider. The adoption of a multi-cloud strategy alleviates the risk of vendor lock-in.

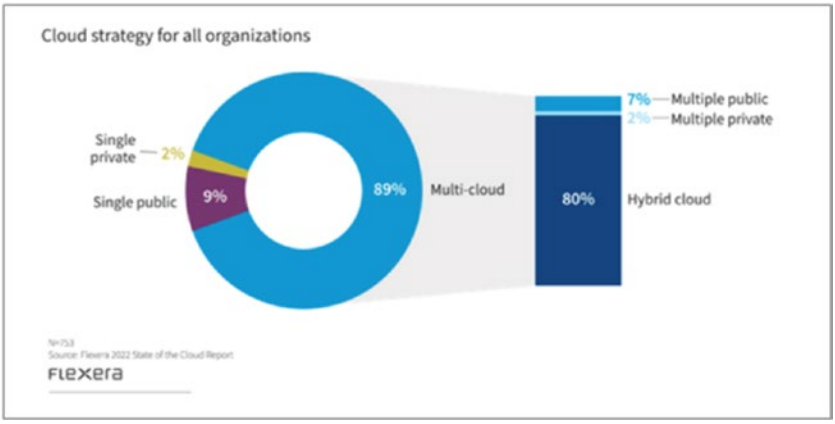

Figure 1-2. *Multi-cloud adoption report*
Source: https://info.flexera.com/CM-REPORT-State-of-the-Cloud?lead_source=Website%20Visitor&id=Blog

In a multi-cloud world, it is essential that enterprises subscribe to services that can be easily ported between clouds. This is especially important for commodity services.

FaaS, or serverless functions, have of late become a commodity with all the providers having some services around FaaS. It is therefore imperative that FaaS containers not have proprietary code.

Serverless functions become portable when they do not use proprietary code. Portable serverless functions allow for workload mobility across clouds. If, for instance, AWS Lambda functions are costly and Azure Functions are cheap, enterprises can avail the cost savings and move that Lambda workload to Azure Functions with very little effort.

The subsequent sections discuss in detail these portability issues and explain how you can solve them.

1.2. Code Portability Issues

Listing 1-1 shows the sample AWS Lambda code written in Java. This code was generated using the AWS SAM (Serverless Application Model) template. When observing the code, you can see that the AWS-specific library references and method calls bind the code to AWS. It is not much, but it is potent. In an enterprise, you typically have hundreds if not thousands of pieces of code that you must refactor if you want to move this type of code to another cloud provider. This is a costly affair.

Listing 1-1. Sample Code Using the AWS SAM Framework

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    public APIGatewayProxyResponseEvent handleRequest(final APIGatewayProxyRequestEvent input, final Context
context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\", \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }

    private String getPageContents(String address) throws IOException{
        URL url = new URL(address);
        try(BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

AWS Specific Code

AWS Specific Code

The following section explores the portability of the underlying serverless container, which impacts how multi-cloud migrations are conducted.

1.2.1. Serverless Container Portability Issue

What about Lambda's underlying serverless framework? Is it portable?

If you deep dive into AWS Lambda, the virtualization technology used is Firecracker. Firecracker uses KVM (a kernel-based virtual machine) to create and manage microVMs. You can find more information on Firecracker at https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

The minimalist design principle that Firecracker is built on allows for fast startup and shutdown times. Google Cloud Functions, on the other hand, use gVisor and are not compatible with Firecracker. gVisor is an application kernel for containers. More information can be found at https://github.com/google/gvisor.

Azure Functions take a totally different approach of using the PaaS offering app service as their base. So, you can see that the major cloud providers use their own frameworks for the managing functions' containers. This makes it difficult for functions to move between clouds in a multi-cloud environment. This portability issue becomes more pronounced due to the lack of portability at the container level.

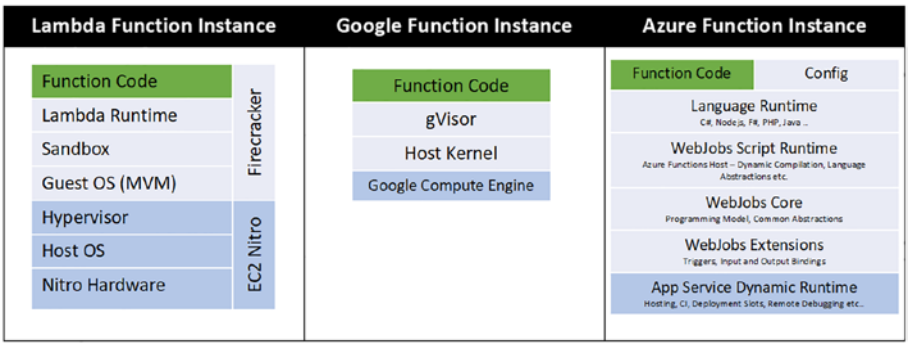

Figure 1-3. *Serverless architecture comparison*

You can see that the code and containers both differ from the provider and are not easily portable.

In the discussions so far, you have seen the following issues related to FaaS:

- Portability of code
- Portability of the serverless container
- Cold start of the serverless environment

How do you solve these issues?

Enter Spring Cloud Function and Knative. Spring Cloud Function addresses function code portability, and Knative addresses container portability.

Information on Spring Cloud Function is available at https://spring.io/projects/spring-cloud-function, and information about Knative is available at https://knative.dev/docs/.

The following sections deep dive into each of these topics.

1.3. Spring Cloud Function: Writing Once and Deploying to Any Cloud

As you learned from the earlier discussion, writing functions for AWS Lambda, Google Cloud Functions, or Azure Functions is a proprietary activity. You have to write code specific to a hyperscaler. Hyperscalers refer to large-scale cloud providers like AWS, Google, or Azure, who have a complete mix of hardware and facilities that can scale to 1000s of servers. This is not bad if your strategy is to have a strong single hyperscaler relationship, but over time, when your strategy changes to a hybrid cloud or multi-cloud, you may have to rethink your approach.

> *A hybrid cloud comprises a private cloud and a public cloud and is managed as one entity. Multi-cloud includes more than one public cloud and does not have a private cloud component.*

This is where the Spring Cloud Function comes in. The Spring.io team started the Spring Cloud Function project with the following goals:

- Promote the implementation of business logic via functions.
- Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
- Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
- Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

Source: https://spring.io/projects/spring-cloud-function

The key goals are decoupling from a specific runtime and supporting a uniform programming model across serverless providers.

Here's how these goals are achieved:

- Using Spring Boot
- Wrapper beans for Function<T, R> (Predicate), Consumer<T>, and Supplier<T>
- Packaging functions for deployments to target platforms such as AWS Lambda, Azure Functions, Google Cloud Functions, and Knative using adapters
- Another exciting aspect of Spring Cloud Function is that it enables functions to be executed locally. This allows developers to unit test without deploying to the cloud

Figures 1-4 and 1-5 show how you can deploy Spring Cloud Function. When Spring Cloud Function is bundled with specific libraries, it can be deployed to AWS Lambda, Google Cloud Functions, or Azure Functions.

Figure 1-4. *Deploying Spring Cloud Function directly to FaaS environments provided by the cloud providers*

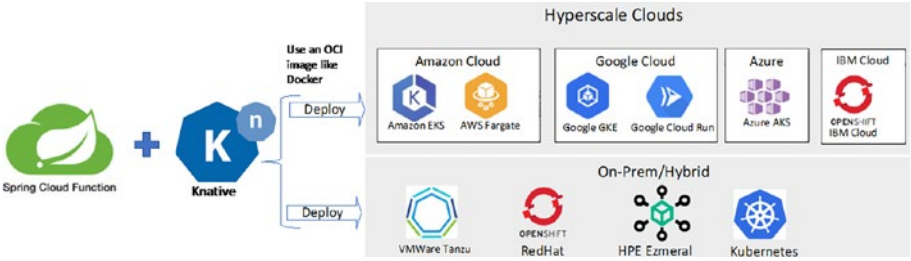

Figure 1-5. *Deploying Spring Cloud Function on Knative serverless configured on Kubernetes environments provided by the cloud providers*

When Spring Cloud Function is containerized on Knative, it can be deployed to any Kubernetes offering, whether on the cloud or on-premises. This is the preferred way to deploy it on hybrid and multi-cloud environments.

1.4. Project Knative and Portable Serverless Containers

Having a portable serverless container is also important. This minimizes the complexity and time required to move between cloud providers. Moving between cloud providers to take advantage of discounted pricing goes a long way toward saving costs. One methodology used is called *cloud bursting* (Figure 1-6). Cloud bursting compensates for the lack of infrastructure on-premises by adding resources to the cloud. This is usually a feature of a hybrid cloud.

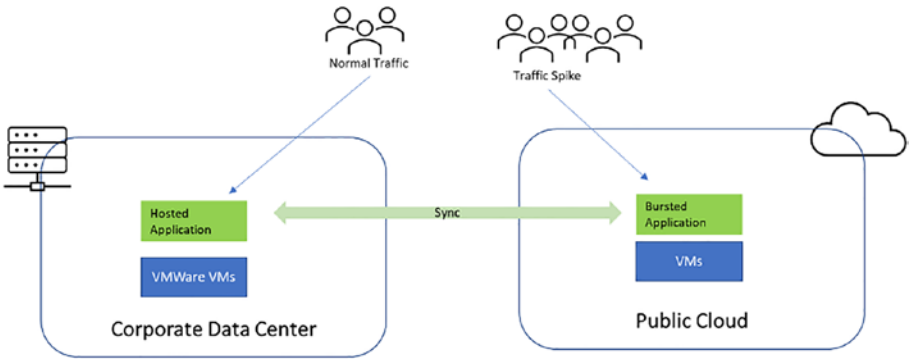

Figure 1-6. *Cloud bursting*

Figure 1-6 shows that, to compensate for the lack of resources in a private cloud during a traffic spike, resources are allocated to the public cloud where the traffic is routed. When the traffic spike goes down, the public cloud resources are removed. This allows for targeted use of costs and resources—that is, it uses additional resources only during the traffic spike period. The burst of activity during an eCommerce event like Cyber Monday is a great example of a traffic spike.

This cannot be easily achieved with just a portable code. You need containers that are also portable. This way, containers can be moved across cloud boundaries to accommodate traffic spikes. In Figure 1-6, you can see that VMs from VMware are used as containers. Since the VMs hosted in the datacenter and hosted in the cloud are similar in construct, cloud bursting is possible.

Applying this to Functions as a Service, you need a new way to make the underlying serverless containers portable.

One such revolutionary approach in the cloud function world is Knative. The next section dives deep into Knative.

1.4.1. Containers, Serverless Platforms, and Knative

What was the need for containers /serverless platforms?

Over the course of the evolution of IT, there has been a need for secure isolation of running processes. In the early 90's, chroot jail-based isolation allowed developers to create and host a virtualized copy of the software system. In 2008 Linux Containers (LXC) was introduced which gave the developers a virtualized environment. In 2011 Cloud Foundry introduced the concept of a container, and with Warden in 2019 container orchestration became a reality. Docker, introduced in 2013, provided containers that can host any operating system. Kubernetes, introduced in 2014, provided the capability to orchestrate containers based on Docker. Finally, Knative, introduced in 2018, extended Kubernetes to enable serverless workloads to run on Kubernetes clusters.

Serverless workloads (Knative) grew out of the need to help developers rapidly create and deploy applications without worrying about the underlying infrastructure. The serverless computing model takes care of provisioning, management, scheduling, and patching and allows cloud providers to develop the "pay only for resources used" model.

With Knative, you can create portable serverless containers that run on any Kubernetes environment. This allows for FaaS to be portable in a multi-cloud or hybrid-cloud environment.

Besides making developers more productive, the serverless environment offers faster deploys (see Figure 1-7). Developers can use the "fail fast and fail often" model and spin up or spin down code and infrastructure faster, which helps drive rapid innovation.

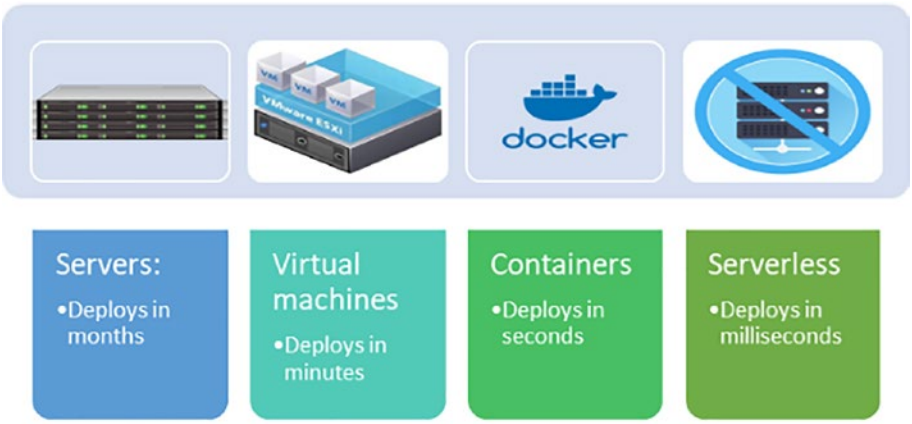

Figure 1-7. *Serverless deploys the quickest*

1.4.2. What Is Knative?

Knative is an extension of Kubernetes that enables serverless workloads to run on Kubernetes clusters. Working with Kubernetes is a pain. The amount of tooling that is required to help developers move their code from the IDE to Kubernetes defeats the purpose of the agility that Kubernetes professes to bring to the environment. Knative automates the process of build packages and deploying to Kubernetes by provider operators that are native to Kubernetes. Hence, the names "K" and "Native".

Knative has two main components:

- *Serving*: Provides components that enable rapid deployment of serverless containers, autoscaling, and point-in-time snapshots
- *Eventing*: Helps developers use event-driven architecture by providing tools to route events from producers to subscribers/sinks

You can read more about Knative at `https://Knative.dev/docs`.