# Containerized microservices on Kubernetes vs Serverless Architecture for Scalable Web Applications

Paweł Skorupa

DAT351 Project

Western Norway University of Applied Sciences

November 12, 2024

# Contents

# 1 Introduction

In recent years, organizations have increasingly adopted cloud-based approaches to build and deploy scalable, efficient, and flexible applications. This report examines two prominent architectures for developing and running web applications in the cloud: containerized microservices on Kubernetes and serverless architecture. These two approaches represent distinct philosophies in cloud computing, each with unique benefits, challenges, and use cases. The discussion focuses on understanding the structural and functional differences between these architectures, as well as their impact on performance, scalability, and operational management for modern web applications.

Cloud computing, broadly defined as the on-demand delivery of computing resources via the internet on a pay-per-use basis, provides the infrastructure and tools necessary to deploy applications without the overhead of physical hardware management. Within this landscape, two approaches stand out: Kubernetes and serverless. Kubernetes, a powerful container orchestration platform originally developed by Google, has become the leading choice for managing containerized applications at scale. By organizing applications into isolated containers that run in clusters, Kubernetes offers extensive control over deployment, scaling, and recovery. Serverless, on the other hand, abstracts infrastructure concerns even further, allowing developers to run application functions on-demand without having to manage the underlying server resources.

This report will begin with an exploration of the fundamental concepts behind containerized microservices on Kubernetes and serverless architecture. It will outline the key components, advantages, and limitations of each, followed by a comparison of performance, and scalability. By deploying a test application on both Kubernetes and serverless environments, this study will provide insights into the suitability of each approach for different web application scenarios.

## 1.1 Motivation

The rapid adoption of cloud-native technologies has highlighted the need to choose optimal deployment strategies based on specific application requirements. With applications growing in complexity and requiring flexible, scalable, and cost-effective solutions, understanding the comparative advantages of Kubernetes and serverless is essential. The motivation for this study is to assess both deployment models to understand their impact on application design, management, and scalability in real-world settings. This investigation will help identify which model may be best suited for applications with different operational needs.

## 1.2 Background

Kubernetes was initially developed by Google and open-sourced in 2014, quickly becoming the industry standard for container orchestration. It provides a robust ecosystem for managing complex, microservices-based applications, allowing companies to deploy, scale, and manage containerized applications efficiently. On the other hand, serverless computing was popularized by Amazon Web Services (AWS) with the launch of AWS Lambda in 2014, enabling developers to run code in response to events without provisioning or managing servers. Serverless has since gained traction for use cases that prioritize cost-efficiency, event-driven processing, and reduced infrastructure overhead.

As the demand for scalable cloud solutions continues to grow, comparing Kubernetes and serverless architectures provides valuable insights into how organizations can choose between them based on factors like application type, scalability requirements, and operational complexity. This report explores both deployment models through theoretical analysis and practical testing, contributing to a broader understanding of cloud-native application deployment strategies.

# 2 Architecture Overview

This chapter delves into the two core architectures evaluated in this report: **Kubernetes for containerized microservices** and **serverless architecture**. Each approach offers distinct mechanisms for deploying and scaling applications in the cloud, and each has specific advantages and limitations. In this section, we will explore the underlying structure, benefits, and challenges of Kubernetes and serverless, as well as the scenarios where each architecture excels.

## 2.1 Kubernetes for Containerized Microservices

Kubernetes, originally developed by Google and later open-sourced in 2014, has become the leading container orchestration platform for deploying and managing containerized applications at scale. At its core, Kubernetes enables the deployment of applications in isolated containers across clusters of virtual or physical machines, abstracting away infrastructure complexity while providing flexibility in scaling, rolling updates, and fault tolerance. Kubernetes is often leveraged for microservices architectures, where applications are composed of independently deployable services, each running in its own container.

### 2.1.1 Key Components of Kubernetes

A Kubernetes cluster consists of several essential components that work together to manage applications and resources:

- **Nodes**: Physical or virtual machines within the cluster, each node runs a container runtime (usually Docker or containerd) and includes a set of management tools.
- **Pods**: The smallest deployable units in Kubernetes, each pod contains one or more containers that share network and storage resources.
- **ReplicaSets**: Ensures a specified number of identical pods are running, providing redundancy and facilitating horizontal scaling.
- **Services**: Abstracts and manages network access to pods, enabling communication within and outside the cluster.
- **Ingress**: Manages external access to services, typically providing HTTP and HTTPS routing.

### 2.1.2 Benefits of Kubernetes for Microservices

Kubernetes provides several advantages, especially for microservices-based applications:

- **Scalability**: Kubernetes allows for horizontal scaling of applications, enabling clusters to adjust to changing workloads.
- **Reliability and Resilience**: Kubernetes can detect and restart failed containers, self-healing to ensure high availability.
- **Flexible Deployment Models**: It supports rolling updates, blue-green deployments, and canary releases, allowing seamless application updates with minimal downtime.
- **Ecosystem and Extensibility**: Kubernetes has a robust ecosystem with various integrations, allowing developers to add monitoring, logging, security, and networking tools.

### 2.1.3 Challenges of Kubernetes

Despite its strengths, Kubernetes also presents challenges:

- **Complexity**: Kubernetes has a steep learning curve and requires expertise in cluster management, networking, and container orchestration.
- **Resource Overhead**: Managing a Kubernetes cluster, especially at scale, can demand significant computational resources.
- **Operational Management**: Kubernetes requires ongoing maintenance, monitoring, and cost management, which can be complex in large environments.

Overall, Kubernetes is an ideal solution for applications requiring fine-grained control over infrastructure and for organizations capable of managing the complexity associated with container orchestration.

## 2.2 Serverless Architecture

Serverless architecture abstracts the infrastructure layer, enabling developers to run applications and services without provisioning or managing servers. The serverless model, popularized by services like AWS Lambda, Google Cloud Functions, and Azure Functions, allows developers to focus on code rather than infrastructure. Serverless functions run in response to events (such as HTTP requests or database triggers) and automatically scale up or down based on demand. While serverless originally referred to function-as-a-service (FaaS) models, it has expanded to include backend services such as serverless databases, message queues, and storage solutions.

### 2.2.1 Key Components of Serverless Architecture

The core components of serverless architecture are as follows:

- **Functions-as-a-Service (FaaS)**: Small, stateless functions that execute code in response to events, scaling automatically and billed based on actual execution time.
- **Event Sources**: Triggers for serverless functions, which may include HTTP endpoints, database events, messaging queues, or scheduled timers.
- **Managed Backend Services**: Serverless applications often rely on managed services for databases, storage, authentication, and messaging, minimizing operational overhead.
- **API Gateway**: Provides a RESTful or WebSocket interface to connect client applications to serverless functions, handling request routing and authorization.

### 2.2.2 Benefits of Serverless Architecture

Serverless architecture offers several notable advantages:

- **Automatic Scaling**: Serverless functions scale automatically based on incoming requests, allowing applications to handle high traffic with ease.
- **Cost Efficiency**: Billed based on actual usage rather than reserved resources, serverless is cost-effective for workloads with variable or intermittent traffic.
- **Reduced Operational Overhead**: Serverless providers handle infrastructure management, reducing the need for operational expertise.
- **Rapid Development and Deployment**: Serverless functions are typically small and modular, enabling faster iteration and easier deployment.

### 2.2.3 Challenges of Serverless

While serverless is appealing, it has its own limitations:

- **Cold Start Latency**: Serverless functions that have been inactive may experience initial latency due to the time taken to initialize the function environment.
- **Limited Control**: The serverless model abstracts away infrastructure, which can limit control over networking, storage, and processing power.
- **Complexity in Distributed Architectures**: Building complex workflows or stateful applications may require intricate design patterns to manage inter-function communication and state.
- **Vendor Lock-In**: Serverless functions are often tied to specific cloud providers, creating dependencies that can be challenging to migrate.

Serverless is well-suited for applications with unpredictable workloads, event-driven tasks, and minimal infrastructure requirements. Its automatic scaling and operational efficiency make it a powerful choice for lightweight, event-based applications or backend tasks.

# 3 Implementation Process

This chapter outlines the steps taken to implement, containerize, and deploy a simple Flask-based REST API on both Kubernetes (using Minikube) and AWS Lambda (using Zappa). It also describes the process of testing these deployments using Artillery, a load-testing tool. The technologies used in each stage of this project are briefly introduced to provide context on their roles and benefits in achieving scalable, cloud-based application deployments.

## 3.1 Creating a Flask REST API

The REST API was built using **Flask**, a lightweight web framework for Python. Flask is widely adopted for building APIs due to its simplicity and flexibility, which makes it a popular choice for small to medium-scale web applications. The application itself was structured around a single endpoint /tasks, supporting GET and POST methods for managing a list of tasks, as well as DELETE functionality for removing individual tasks. The minimal codebase was structured as follows:

- **GET /tasks** - Returns a list of tasks in JSON format.
- **POST /tasks** - Adds a new task to the list, returning the created task.
- **DELETE /tasks/<id>** - Deletes a task based on its ID.

Flask allowed for rapid development and straightforward API routing, making it ideal for this project's focus on deployment and scalability rather than application complexity.

## 3.2 Dockerizing the Flask Application

Once the Flask API was developed, it was packaged into a Docker container to facilitate deployment across different platforms. Docker is a containerization technology that isolates applications and their dependencies into portable, self-contained units called containers. This provides consistency across different environments and enables the application to be deployed with ease on Kubernetes.

The process of Dockerizing the application involved creating a Dockerfile, which specified:

1. A base image (e.g., Python 3.11).
2. Copying the application code and installing dependencies.
3. Configuring the container to start the Flask server when launched.

Dockerizing the application also allowed for efficient testing on local machines using commands like docker run, ensuring that the container worked correctly before deploying it to a Kubernetes environment.

## 3.3 Deployment on Kubernetes Using Minikube

To deploy the Dockerized Flask app on **Kubernetes**, the project used **Minikube**, a local Kubernetes environment that simulates a Kubernetes cluster. This enabled testing of the deployment and scaling of the application in a controlled environment before committing to a full cloud-based deployment.

**Steps for Kubernetes Deployment**

1. **Creating Kubernetes Deployment and Service Configurations**: A Kubernetes deployment configuration was created to specify the container image, replica count (number of pod instances), and restart policy. A Kubernetes service of type LoadBalancer was also created to expose the API to external traffic, routing requests to the pods running the Flask application.
2. **Testing Scaling and Load Balancing**: By adjusting the replicas count in the deployment configuration, the project tested the application's ability to handle varying traffic loads using multiple instances (pods) of the Flask app.
3. **Accessing the Service**: Minikube's tunneling feature was used to make the LoadBalancer accessible locally, allowing testing of the API endpoints on a designated external IP and port.

Kubernetes, with its self-healing, auto-scaling, and load-balancing capabilities, provides flexibility and control, making it an effective platform for managing containerized applications.

## 3.4 Deployment on AWS Lambda Using Zappa

For serverless deployment, the project utilized **AWS Lambda** to deploy the Flask API as a function-based service, eliminating the need to manage infrastructure manually. **Zappa**, a Python-based framework, was used to automate this deployment process. Zappa simplifies the conversion of WSGI-compatible Python applications (such as those built with Flask or Django) into serverless applications that can be hosted on AWS Lambda.

**Steps for AWS Lambda Deployment**

1. **Configuring Zappa**: A zappa_settings.json file was created to define deployment settings such as the S3 bucket for code storage, the runtime environment, and the application entry point (app.app for Flask apps).
2. **Deploying with Zappa**: Using Zappa commands, the application was packaged, uploaded, and deployed to AWS Lambda. Zappa also automatically created an API Gateway to handle HTTP requests, routing them to the Lambda function.
3. **Testing the API on AWS**: Once deployed, the API endpoints were accessible via the URL provided by API Gateway, and functionality was verified by making requests to /tasks.

AWS Lambda, by abstracting infrastructure management and offering a pay-per-use model, is highly scalable and ideal for event-driven applications with variable workloads.

## 3.5 Performance Testing Using Artillery

**Artillery** is a load-testing tool that enables the simulation of different traffic loads and patterns on applications, generating metrics on response times, throughput, and error rates. For this project, Artillery was used to compare the performance of the Flask API deployed on Kubernetes with its serverless deployment on AWS Lambda.

**Steps for Testing with Artillery**

1. **Defining Test Scenarios**: Artillery scenarios were created to test both the GET and POST endpoints, with specified request rates and durations. This allowed for a consistent comparison of how each deployment handled simultaneous traffic.
2. **Running Tests**: Artillery was run from the command line, targeting both the Kubernetes and Lambda deployments to record metrics on response time, success rate, and system stability under load.
3. **Analyzing Results**: Results from Artillery provided insights into the latency, scalability, and resilience of each architecture, helping to identify strengths and bottlenecks in the Kubernetes and Lambda-based deployments.

Artillery's load testing enabled an objective performance comparison between Kubernetes and serverless architectures, facilitating the analysis of their suitability for scalable web applications.

# 4 Test Summary

This section summarizes the performance testing conducted on a containerized Flask API deployed on both Kubernetes (using Minikube) and AWS Lambda. Tests were run at various request rates (10, 50, 100, and 500 requests per second) to assess each deployment's response time and success rate under different loads, with some tests at higher rates repeated to observe the impact of prolonged load.

## 4.1 Summary of Results

| | | AWS Lambda | K8s (2 pods) | K8s (4 pods) |
|---|---|---|---|---|
| Test 1: 10 requests per second, duration: 30 s | Average response time [ms] | 85.7 | 35.3 | - |
| | Success rate [%] | 98.60 | 100 | - |
| Test 2: 50 requests per second, duration: 30 s | Average response time [ms] | 65.7 | 21.1 | - |
| | Success rate [%] | 90.40 | 100 | - |
| Test 3: 100 requests per second, duration: 30 s | Average response time [ms] | 56.5 | 17.3 | - |
| | Success rate [%] | 51.63 | 100 | - |
| Test 4: 500 requests per second, duration: 30 s | Average response time [ms] | 76.1 | 3018 | 2298.2 |
| | Success rate [%] | 0.65 | 16.67 | 23.69 |
| Test 5: 50 requests per second, duration: 120 s | Average response time [ms] | 59.2 | 28.4 | - |
| | Success rate [%] | 97.43 | 100 | - |

*Table 1: summary of tests results*

- **Response Time**: Across most test scenarios, the Kubernetes deployment showed lower average response times compared to AWS Lambda. Kubernetes consistently responded faster, especially at moderate loads. Lambda's response times, though acceptable at lower loads, showed higher latency as the request rate increased.
- **Reliability (Success Rate)**: At lower loads (up to 100 requests per second), Kubernetes maintained a 100% success rate, while AWS Lambda experienced minor dips in reliability.

As traffic increased, Lambda's success rate dropped significantly, reflecting potential limitations in concurrent request handling. Kubernetes also experienced reliability issues at the extreme load of 500 requests per second, though it benefited somewhat from an increased pod count, while Lambda struggled to maintain reliability at that rate.

- **Performance Over Time**: During a prolonged moderate load (50 requests per second over two minutes), Kubernetes demonstrated stable, low response times and a perfect success rate. Lambda also performed well in terms of stability, though it had slightly higher response times and a small decrease in the success rate.

## 4.2 Key Findings

1. **Low to Moderate Load**: Kubernetes delivered superior response times and a high success rate, handling low to moderate traffic with minimal latency. Lambda provided consistent responses but showed slightly higher response times and occasional dips in success rates, likely due to platform limitations.
2. **High Load**: Both deployments encountered limitations at very high loads, though Kubernetes had some resilience with additional pods. Lambda maintained faster response times even under heavy load but exhibited significant reliability issues, indicating its potential constraints on concurrency at extreme levels.
3. **Overall Performance and Suitability**: Kubernetes (via Minikube) demonstrated greater reliability and lower latency at consistent traffic levels, making it well-suited for applications with stable or predictable loads. Lambda, with a more dynamic scaling model, could serve applications with sporadic traffic spikes but may not be as effective for consistent high-load scenarios without specific concurrency management.

# 5 Summary

This project set out to compare two deployment models for a scalable web application—containerized microservices on Kubernetes versus a serverless architecture using AWS Lambda—by implementing and testing a simple Flask API in both environments. While the tests provided valuable insights, the comparison had certain limitations, particularly since the Kubernetes deployment was constrained to a local Minikube instance. This restricted scalability and did not fully emulate a production-grade, cloud-hosted Kubernetes environment, such as AWS Elastic Kubernetes Service (EKS). EKS was not tested due to budget constraints, which leaves open the question of how a fully cloud-managed Kubernetes cluster would perform compared to Lambda under similar conditions.

This project thus serves as a solid starting point for future, more comprehensive testing on cloud-hosted Kubernetes services. With the deployment and testing processes now well-defined, future work can expand this framework to evaluate Kubernetes on AWS EKS, enabling a more realistic assessment of Kubernetes's scalability and performance in cloud-native environments. The

insights gained here provide a strong foundation for ongoing performance comparisons and decision-making around Kubernetes and serverless architecture in real-world applications.