Docker vs. Kubernetes on Google Cloud Platform for Cost-Effective Spring Boot Deployments

Srinivas Adilapuram

B.Tech in Computer Science Engineering
Designation: Software Engineer, Equifax Inc.
Email id: srini.adilas[at]gmail.com

Abstract: ***Deploying Java Spring Boot applications on Google Cloud Platform (GCP) involves choosing cost-effective solutions. While Kubernetes provides powerful orchestration, its high costs make it impractical for smaller-scale applications. This article proposes Docker as an alternative for containerizing Spring Boot applications on a single virtual machine (VM). Docker ensures consistent environments, simplifies management, and reduces infrastructure costs. This approach enhances deployment reliability and streamlines operations, making it ideal for budget-conscious teams.***

Keywords: Docker, Spring Boot, Google Cloud Platform, containerization, virtual machine, cost-effective deployment

1.Introduction

Java Spring Boot is a framework that simplifies the development of Java-based applications [1]. It provides pre-configured templates that reduce the effort needed to set up new projects. Developers use Spring Boot to build microservices, REST APIs, and enterprise-level applications [2]. It ensures flexibility, scalability, and rapid development cycles.

Spring Boot works by embedding essential tools like Tomcat, Jetty, or Undertow servers directly into the application [3]. It eliminates the need for manual server configurations. Developers only focus on writing business logic. It manages dependencies, configurations, and integrations with minimal setup. However, deploying Spring Boot applications is not always straightforward. [4]

One common challenge is environmental consistency. Applications often work differently in development, testing, and production environments. Misaligned configurations or missing dependencies lead to errors and delays. Additionally, managing multiple Spring Boot applications on a single virtual machine (VM) can cause resource conflicts [5]. These issues waste time, increase operational costs, and strain team resources.

Addressing these challenges is critical for cost efficiency. Issues such as debugging errors, resource conflicts, and manual scaling increase operational costs and strain team productivity. Streamlining the deployment process with tools that ensure consistency and scalability is essential to maximize performance while minimizing expenses.

While Kubernetes is a popular solution for orchestrating containerized applications, its implementation is often expensive and complex, especially for smaller-scale deployments. A more practical alternative is Docker [6]. Docker encapsulates applications and their dependencies into portable, isolated containers. This enables consistent environments across all stages of deployment. By running Dockerized Spring Boot applications on a single VM in Google Cloud Platform (GCP), teams can avoid the high costs and overhead associated with Kubernetes clusters [7]. Docker provides reliability, scalability, and cost savings via better resource usage and reduced debugging time [8].

2.Literature Review

The use of Docker and containerization has been a significant development in application deployment. Researchers and forums have discussed its impact on performance, cost, and scalability extensively. Docker simplifies application management. It packages applications with their dependencies into isolated containers. [9]

Spring Boot is very popular for developing microservices. Soni et al. (2017) described how Spring Boot provides pre-configured tools, reducing setup time for Java developers. This helps streamline development and testing [1]. Sharma (2019) also extended this by showcasing Spring Boot's compatibility with cloud platforms like Google Cloud, which improves scalability [5].

Kubernetes is another popular orchestration tool. However, it is resource-intensive for small-scale deployments. Vasireddy et al. (2023) noted Kubernetes excels at auto-scaling but requires significant infrastructure, making it costly for smaller teams [7]. Onyebuchi (2021) suggested Docker as an alternative for teams operating on limited budgets. He found that Docker achieves similar results on single VMs without the complexity of Kubernetes [6].

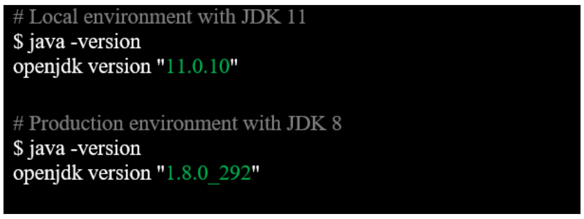
3.Problem Statement: Challenges of Single VM deployments

Deploying Java Spring Boot applications in a single virtual machine (VM) environment on Google Cloud Platform (GCP) without Kubernetes orchestration poses significant challenges. While Kubernetes excels at auto-scaling and managing resources for large-scale applications, its implementation requires multiple nodes and additional infrastructure. This significantly increases costs, making Kubernetes impractical for smaller teams and projects.

The high infrastructure costs, management overhead, and additional fees associated with Kubernetes clusters create barriers for organizations working on limited budgets. A single VM setup with Docker offers a simpler and more cost-effective solution. Without proper containerization, teams face issues such as environmental inconsistencies, dependency conflicts, and inefficient resource allocation. These problems lead to debugging delays, wasted resources, and restricted scalability.

Inconsistent Development, Testing, and Production Environments

The absence of a standardized runtime environment often results in discrepancies across development, testing, and production stages. Developers frequently encounter the infamous "it works on my machine" issue due to differences in configuration, operating systems, or dependency versions [9]. For instance:



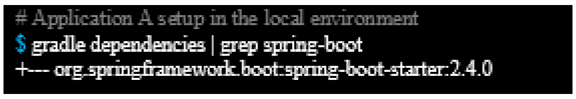
```
# Local environment with JDK 11
$ java -version
openjdk version "11.0.10"

# Production environment with JDK 8
$ java -version
openjdk version "1.8.0_292"
```

Figure 1: Mismatched functionality represented as Docker Code

Such mismatches end up breaking functionality, introducing bugs, and therefore leading to significant debugging time and resources. This delays project timelines, frustrates teams, and increases costs as a direct result. [1]

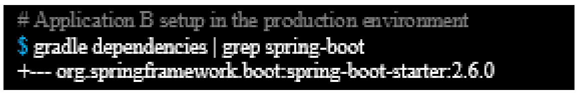
Dependency Conflicts

A single VM setup often involves running multiple applications [10], which can create dependency conflicts. Different applications may require different versions of the same library. Without isolation, this results in failures or unexpected behavior. For example:



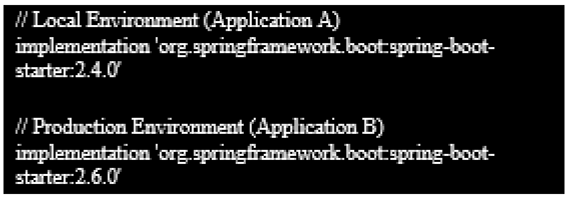
```
# Application A setup in the local environment
$ gradle dependencies | grep spring-boot
+--- org.springframework.boot:spring-boot-starter:2.4.0
```

Figure 2: Local Environment (Application A)



```
# Application B setup in the production environment
$ gradle dependencies | grep spring-boot
+--- org.springframework.boot:spring-boot-starter:2.6.0
```

Figure 3: Production Environment (Application B):



```
// Local Environment (Application A)
implementation 'org.springframework.boot:spring-boot-
starter:2.4.0'

// Production Environment (Application B)
implementation 'org.springframework.boot:spring-boot-
starter:2.6.0'
```

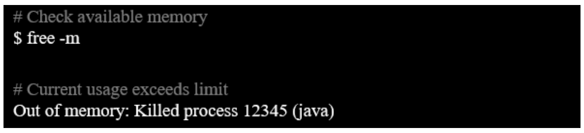
Figure 4: Highlighting Conflict

In this example, Application A uses Spring Boot version 2.4.0 locally, while Application B relies on version 2.6.0 in production. These mismatches can result in errors when deploying to a single VM where both applications coexist, as the shared runtime environment cannot accommodate conflicting dependencies.

Without Docker's isolation capabilities, such issues disrupt workflows, delay debugging, and increase costs. This often leads to "it works on my machine" problems.

Restricted Scalability

A single VM has limited resources (CPU, memory, and storage). Scaling applications within this setup requires manual resource allocation. If user demand increases suddenly, the application may crash due to insufficient resources. For example:



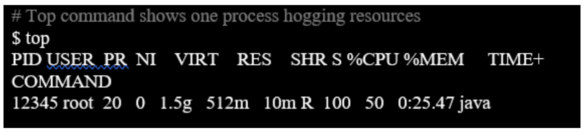
```
# Check available memory
$ free -m

# Current usage exceeds limit
Out of memory: Killed process 12345 (java)
```

Figure 5: Manual Scaling with Limited Resources

Unlike Kubernetes, which auto-scales resources based on demand, manual scaling is error-prone and inefficient. This affects the reliability of critical operations such as file transfers. [1] [4] [5]

Lack of Isolation

Running multiple applications on a single VM without containerization leads to resource contention [11]. One misbehaving process can consume excessive CPU or memory, degrading the performance of other applications. For instance:



```
# Top command shows one process hogging resources
$ top
PID USER  PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
COMMAND
12345 root  20   0   1.5g   512m   10m R  100   50   0:25.47 java
```

Figure 6: Resource Contention Due to Lack of Isolation

Figure 6 shows how one poorly behaving process can monopolize system resources, negatively impacting other applications running on the same VM due to the lack of isolation.

As a result, this lack of isolation jeopardizes application reliability and forces teams to invest additional effort in monitoring and troubleshooting. [5]

Increased Operational Costs

Debugging, maintaining dependencies, and manually scaling applications consume valuable time and resources. Operational costs increase as teams spend more hours resolving issues instead of building new features. Moreover, single VM setups often require additional monitoring tools to ensure stability, further escalating expenses. [11]

High Kubernetes Overhead

Kubernetes solves many of the problems listed above, but can be quite resource-intensive. This is because it requires multiple nodes to form a cluster, which increases infrastructure costs [12]. For example:

```
# Minimum cluster setup (3 nodes)
apiVersion: v1
kind: Node
metadata:
  name: node-1
  name: node-2
  name: node-3
```

Figure 7: Kubernetes Cluster Setup Overhead

The YAML configuration above is an example of a basic Kubernetes cluster setup with three distinct nodes. The goal is to show Kubernetes' resource-intensive nature, which increases infrastructure costs. GCP can charge additional fees for managing more clusters, making Kubernetes expensive for smaller setups. The cost often outweighs the benefits, especially when simpler solutions like Docker on a single VM can address the operational needs without being as resource or cost intensive. [2]

Solution: Adopting Docker for Containerized Deployments

1. Preparing the Spring Boot Application

For the first step, you will be creating a Spring Boot application. Use the spring-boot-starter-web dependency for REST APIs. A simple Dockerfile is needed to containerize the application.

Here's the pom.xml Configuration for Spring Boot application.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

Figure 8: Configuring Maven dependencies for a basic Spring Boot application. This configuration sets up the required libraries for a web-based API.

2. Creating the Dockerfile

The Dockerfile specifies instructions to build the Docker image for the Spring Boot application.

```
# Step 1: Use a base image
FROM openjdk:17-jdk-slim

# Step 2: Set working directory inside the container
WORKDIR /app

# Step 3: Copy application JAR to the container
COPY target/my-spring-boot-app.jar app.jar

# Step 4: Expose the application port
EXPOSE 8080

# Step 5: Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Figure 9: Dockerfile to containerize the Spring Boot application.

This file defines the container's environment. The ENTRYPOINT command ensures the container runs the application on startup.

3. Building the Docker Image

Use the docker build command to create the image. Tag the image appropriately for easy reference. The command for this is: docker build -t spring-boot-app:latest. This step compiles the Dockerfile and packages the application into a deployable container image.

4. Running the Docker Container

Run the container locally to test its functionality. Use the -p flag to map the container port to the host machine. The command for this is: docker run -d -p 8080:8080 spring-boot-app:latest This command deploys the application in a container and binds port 8080 of the container to the host's port 8080.

5. Deploying to GCP VM

Transfer the Docker image to a GCP virtual machine. Use the Google Container Registry (GCR) to host the image and pull it from the VM.

The command to give here is to push Docker image to GCR. This involves inputting the following functions:

```
docker tag spring-boot-app gcr.io/[PROJECT-ID]/spring-boot-app
docker push gcr.io/[PROJECT-ID]/spring-boot-app
```

Figure 11: Uploading the Docker image to GCR.

With this, the image should now be accessible globally within your GCP project. Next, you need to deploy the Dockerized Spring Boot application on a GCP VM. For this use the following function:

```
docker pull gcr.io/[PROJECT-ID]/spring-boot-app
docker run -d -p 8080:8080 gcr.io/[PROJECT-ID]/spring-boot-app
```

Figure 12: Deploying the Dockerized Spring Boot application on a GCP VM.

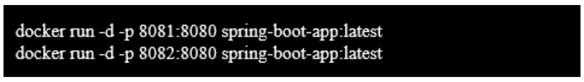
With this step, you will be able to run the application within the cloud environment, ensuring consistent performance.

6. Managing Dependencies and Isolation

Docker containers isolate dependencies, preventing conflicts between applications. Each container runs in its own environment, ensuring stability. To verify that containers are running properly for deployment, use the following command: docker ps This command confirms active containers and validates that no conflicts exist.

7. Scaling Within a Single VM

To scale, replicate the Docker container. Use multiple instances to distribute traffic and enhance performance.



```
docker run -d -p 8081:8080 spring-boot-app:latest
docker run -d -p 8082:8080 spring-boot-app:latest
```

Figure 13: Scaling the Spring Boot application by running multiple containers.

Here, each container handles separate requests, distributing the load and improving reliability.

5. Cost Impacts

Adopting Docker for deploying Spring Boot applications on a VM in GCP brings substantial cost benefits.

The most important aspect that Docker addresses is that it eliminates the need for a multi-node Kubernetes cluster, drastically reducing infrastructure expenses [14]. A single VM with Docker ensures reliable deployments without the overhead of maintaining additional nodes or paying for managed Kubernetes services.

On average, the costs and infrastructure may be as follows:

- **Single VM with Docker:** GCP's e2-standard-4 instance (4 vCPUs, 16 GB RAM) costs approximately $12z3.25 per month if running 24/7 in the US region can support multiple containers, handling small-to-medium-scale workloads efficiently. [13]
- **Kubernetes Cluster:** A GCP Kubernetes Engine (GKE) cluster with a minimum of three e2-standard-4 nodes incurs a base monthly cost of $293.49 for compute (3 x $97.93). Additionally, GKE charges a cluster management fee of $0.10, pro-rated down to each second. This mounts to $74.40 per month, making the total infrastructure cost approximately $367.89 per month. [15]

Docker simplifies container management, reducing operational overhead and eliminating the need for a dedicated DevOps engineer, potentially saving an annual salary of $120,000. [2] [6] It also enhances development efficiency by providing a consistent runtime environment, minimizing deployment discrepancies and debugging time. This consistency accelerates delivery cycles, enabling developers to focus on feature development. For instance, saving 10 hours monthly on debugging for a three-person team at $50 per hour translates to $18,000 annually.

4.Conclusion

Deploying Java Spring Boot applications on a single VM in GCP without Kubernetes orchestration introduces several challenges, including inconsistent environments, resource conflicts, and restricted scalability. These challenges take root from the need for cost-effective, reliable, and scalable solutions. While Kubernetes is a robust orchestration platform offering advanced features like automated scaling and self-healing, its complexity and cost make it more suitable for larger-scale deployments where such capabilities justify the investment. Kubernetes, while effective for large-scale orchestration, often proves cost-prohibitive for smaller setups due to its infrastructure demands and management complexity.

The adoption of Docker as a containerization solution bridges this gap effectively. Docker provides isolated, portable environments that standardize the deployment process across development, testing, and production stages. This eliminates the "it works on my machine" problem by ensuring consistency. Additionally, Docker enables simplified dependency management and efficient resource utilization, addressing critical concerns of reliability and performance. [9] [10]

References

[1] R. K. A. G. a. R. V. R. Soni, Spring: Developing Java Applications for the Enterprise, Packt Publishing Ltd., 2017.
[2] S. A. S. Aggarwal, "Spring Boot Application using Three Layered Architecture in Java," Jaypee University of Information Technology, Solan, H.P., Solan, 2023.
[3] S. Selvaraj, "Building RESTful APIs with Spring Boot (Java).," in Mastering REST APIs: Boosting Your Web Development Journey with Advanced API Techniques, Berkeley, CA, Apress, 2024, pp. 291-347.
[4] Jacky, "Avoiding Pitfalls: Common Challenges in Backend Development with Spring Boot," 25 11 2023. [Online]. Available: https://dev.to/jackynote/avoiding-pitfalls-common-challenges-in-backend-development-with-spring-boot-3ink. [Accessed 30 11 2024].
[5] S. Sharma, astering microservices with java: Build enterprise microservices with Spring Boot 2.0, Spring Cloud, and Angular, Packt Publishing Ltd., 2019.
[6] D. J. R. a. A. F. Silva, "Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies.," Computers, vol. 13, no. 4, p. 94, 2024.
[7] A. Onyebuchi, "Dockerizing Spring Boot Microservices and deploying them on Google Cloud Platform," 2 8 2021. [Online]. Available: https://medium.com/@wizardom/dockerizing-spring-boot-microservices-and-deploying-them-on-google-cloud-platform-5e83cb197198. [Accessed 30 11 2024].
[8] G. R. P. K. Indrani Vasireddy, "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges," International Journal of Innovative Research in Engineering and Management, vol. 10, no. 6, pp. 49-54, 2023.
[9] J. A. Pardo, "“But… it works on my machine…”," 4 12 2023. [Online]. Available: https://medium.com/@josetecangas/but-it-works-on-my-machine-cc8cca80660c. [Accessed 30 11 2024].
[10] K. W. &. C. B. David Moreau, "Containers for computational reproducibility," Nature Reviews Methods Primers volume, p. 50, 13 06 2023.
[11] R. T. C. J. M. P. S. a. P. S. Queiroz, "Container-based virtualization for real-time industrial systems—a systematic review," ACM Computing Surveys, vol. 56, no. 3, pp. 1-38, 2023.

[12] Raheem, "Understanding Pod Overhead in Kubernetes: Impact and Control," 19 10 2023. [Online]. Available: https://medium.com/@abdulraheem.akv/understanding-pod-overhead-in-kubernetes-impact-and-control-6ced59e7dafb. [Accessed 30 11 2024].

[13] Knieling, N., "Google Cloud Platform Pricing," GCloud Compute, 2024. [Online]. Available: https://gcloud-compute.com/e2-standard-4.html. [Accessed: Nov. 30, 2024].

[14] Google Cloud, "Google Kubernetes Engine pricing," Google Cloud, 2024. [Online]. Available: https://cloud.google.com/kubernetes-engine/pricing. [Accessed: Nov. 30, 2024].