A Performance Comparison of Parallel Documents Rendering: VM, Kubernetes and FaaS in AWS

Nikola Dinevski
ndinevski5@gmail.com

***Abstract*—This paper compares the performance of parallel PDF document rendering across three AWS architectures Virtual Machine (EC2) with Docker and Nginx, Kubernetes (EKS), and FaaS (Lambda). Using Puppeteer to render documents, measuring request and render times for varying batch sizes to evaluate scalability, cost, and complexity, providing insights for cloud-based compute-intensive applications.**

***Index Terms*—Cloud Computing, AWS, EKS, EC2, Docker, Kubernetes, Lambda, PDF Rendering, Cloud Performance, Scalability**

I. Introduction

Cloud computing has revolutionized how we develop, deploy, and scale applications. With the growing demand for compute-intensive applications, understanding the trade-offs between different cloud architectures is critical for optimizing performance and cost. Document generation is a common task in various industries, from generating invoices to creating reports, making it an ideal use case for evaluating cloud-based architectures.

In this paper, I explore the performance of parallel PDF document rendering across three AWS services: EC2 with Docker and Nginx, EKS (Kubernetes), and Lambda (FaaS). I am using Puppeteer, a Node.js library, to render PDFs from EJS templates and JSON data, simulating a real-world workload where multiple users are trying to generate documents simultaneously. For each architecture, I configured deployments following best practices to ensure a fair comparison.

I analyzed performance metrics such as request time, render time, and total processing time for varying batch sizes (1, 10, 30, 50, and 100 documents). Additionally, I will discuss the benefits, limitations, and trade-offs of each architecture in terms of scalability, cost-effectiveness, and complexity.

The rest of this paper is organized as follows: In Section II, I detail the methodology and setup for each architecture. In Section III, I present the performance results and analysis. In Section IV, I discuss the benefits and drawbacks of each approach. Finally, in Section V, I conclude with insights and recommendations for practitioners and developers.

II. Architecture

In this section, I will describe the setup and configuration of the three AWS architectures and the local setup used to evaluate PDF document rendering performance: local container (Docker), EC2 with Docker and Nginx, EKS (Kubernetes), and Lambda (FaaS). Each architecture was designed and deployed to follow best practices while ensuring consistency in the evaluation process.

A. Local Container (Docker)

For the local container setup, I developed a Node.js application with an Express endpoint that executes the PDF rendering function. The application was containerized using Docker and published to Docker Hub. This setup simulates a standalone deployment where the service runs on a developer's local machine or a similar isolated environment.

B. EC2 with Docker and Nginx

The EC2 setup involved provisioning an Amazon EC2 instance and installing the necessary dependencies, including Docker and Nginx. The containerized application was pulled from Docker Hub and deployed on the instance. Nginx was configured as a reverse proxy to handle incoming HTTP requests and forward them to the Docker container running the Node.js application.

C. EKS (Kubernetes)

To evaluate Kubernetes, I used Amazon Elastic Kubernetes Service (EKS) with a 3-node cluster. The Docker image was published to Amazon Elastic Container Registry (ECR) and deployed to the cluster as part of a Kubernetes Deployment. A Kubernetes Service of type NodePort was configured to expose the application externally, enabling testing under varying loads.

D. Lambda (FaaS)

For the serverless approach, I reused the same rendering function as a Lambda handler. The function was deployed using the AWS Lambda console and exposed through an API Gateway. This setup eliminates the need to manage infrastructure while providing on-demand scalability.

E. Testing Code Structure

The testing was implemented using a Node.js script to send parallel requests to the deployed services. Sending parallel HTTP POST requests, passing the EJS template and JSON data as payloads. Each service was tested with 1, 10, 30, 50 and 100 parallel requests, and metrics such as average response time, render time, and total time were recorded.

Each architecture was tested under identical conditions to ensure a fair comparison of performance metrics.

III. Performance and Results Analysis

This section presents the performance results for PDF document rendering across the evaluated architectures. Metrics such as request time, render time, and total processing time were collected and analyzed for varying batch sizes (1, 10, 30, 50, and 100 documents). The results highlight the trade-offs between scalability, cost, and complexity for each approach.

A. Average Times Per Document

The following table summarizes the average response and generation times per document for 100 parallel requests:

Service	Avg Response Time (s)	Avg Generation Time (s)
Local Container	0.79	0.11
Lambda	0.7	0.22
EC2	0.89	0.27
EKS	0.91	0.28

TABLE I
Average Times Per Document (100 Requests)

B. Total Processing Times for Batches

Total processing times for varying batch sizes were measured to assess the scalability and efficiency of each architecture. The results are shown in the following table:

Num. Documents	Lambda	Local	EC2	EKS
1	1.9	0.96	1.17	1.28
10	2.82	5.42	6.12	3.05
30	7.51	15.54	17.51	9.64
50	9.3	30.43	33.22	17.56
100	14.06	63.41	68.23	35.93

TABLE II
Total Processing Times for Batch Sizes (in seconds)

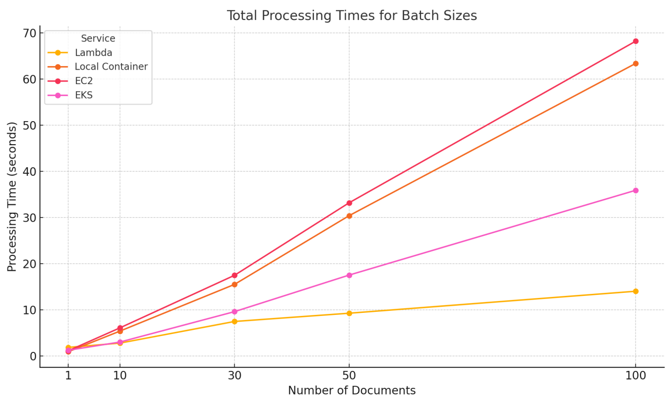

Fig. 1. Graphical Representation of Total Processing Times for Batch Sizes

C. Cost Analysis

Based on the execution times for 100 documents from Table II, cost estimates for each service were calculated.

Lambda, with 2048 MB memory, charges \$0.0000000333 per millisecond. For 100 documents taking 14.06 seconds (14,060 ms), the cost is approximately \$0.00047.

The EC2 instance ('t4g.small') costs \$0.0168 per hour. With an execution time of 68.23 seconds (0.01896 hours), the cost is approximately \$0.00032 per 100 documents. Although we should keep in mind that the EC2 instance keeps running even if there are no documents being generated. Which means the actual cost is much higher.

EKS nodes ('c6g.large') cost \$0.02628 per hour for all nodes. For a 3-node cluster and an execution time of 35.93 seconds (0.00998 hours), the cost for 100 documents is approximately \$0.00026. However, EKS also incurs additional costs such as control plane charges (\$0.10 per hour) and potential load balancer usage, which can increase the overall cost significantly for persistent workloads. Also it keeps running all instances regardless of whether documents are being generated or not. The actual cost is much higher.

Finally, running the Local Container on a dedicated server is assumed free in this simulation but would cost between \$0.05 to \$0.10 per hour on a dedicated infrastructure. With a processing time of 63.41 seconds (0.01761 hours), the estimated cost for 100 documents is approximately \$0.001 to \$0.002.

IV. Benefits and Drawbacks

A. Benefits and Drawbacks of Each Architecture

The comparison between Lambda, EC2, EKS, and Local Container reveals distinct trade-offs in terms of cost, scalability, and complexity. These differences are summarized below, highlighting the strengths and limitations of each service.

a) Lambda (FaaS): Lambda offers the advantage of **true serverless scalability**, automatically provisioning resources to handle incoming requests. This makes it particularly efficient for small and sporadic workloads, as you only pay for the exact execution time. However, the performance for larger workloads, such as 100 parallel requests, demonstrates limitations. While theoretically, serverless functions should handle parallel requests with consistent latency, practical constraints such as **cold starts** and the need to instantiate multiple Lambda functions increase processing time significantly, as seen in the **14.06 seconds** for 100 documents. Additionally, Lambda has execution time limits and resource constraints, making it less suitable for resource-intensive tasks.

b) EC2 with Docker and Nginx: EC2 provides a flexible, single-instance environment with predictable performance. The **t4g.small** instance demonstrated good performance for single or small batch requests. However, for larger workloads, such as 100 documents, the total processing time of **68.23 seconds** indicates potential bottlenecks in resource availability or configuration. EC2 also requires manual provisioning and maintenance, including managing scaling and ensuring high availability. The continuous cost of running an instance, even during idle periods, adds to its drawbacks.

c) EKS (Kubernetes): EKS excels in scenarios requiring distributed workloads, leveraging Kubernetes' orchestration capabilities. With **three c6g.large nodes**, EKS achieved significant scalability, processing 100 documents in **35.93 seconds** by distributing tasks across pods. However, the cost model includes persistent expenses, such as the control plane charge ($0.10/hour) and the hourly cost of the cluster ($0.02628/hour for three nodes), irrespective of workload. These additional costs, coupled with the complexity of managing Kubernetes clusters, make EKS more suitable for high-throughput, continuously running workloads rather than intermittent tasks.

d) Local Container (Docker): The Local Container setup is cost-effective for development and testing, as it incurs no infrastructure costs. It performed well for smaller workloads, processing 10 documents in **5.42 seconds**. However, scalability is limited by the host machine's resources, and the time required for 100 documents (**63.41 seconds**) reflects this constraint. For production workloads, running a Local Container on a dedicated server introduces additional hosting costs, making it less cost-efficient compared to cloud-native solutions.

B. Summary of Results

This comparison highlights that:

Lambda is ideal for small, sporadic workloads but suffers from cold start delays and scalability limitations for large batches.

EC2 offers predictable performance but requires manual scaling and incurs costs during idle periods.

EKS provides excellent scalability and efficiency for distributed workloads but has significant persistent costs and complexity.

Local Container is suitable for development and testing but lacks scalability for production-scale tasks.

These results underscore the importance of aligning workload characteristics with the appropriate cloud architecture to optimize performance, cost, and operational complexity.

V. Conclusion and Recommendations

This study compares the performance and cost-efficiency of PDF document rendering across AWS services—Lambda, EC2, EKS, and Local Container. Each architecture has unique strengths and trade-offs, making them suitable for different use cases.

For small-scale workloads, **Lambda** stands out due to its cost-efficiency and serverless scalability. However, its performance falls for high-concurrency scenarios due to cold starts and scaling delays.

For predictable workloads requiring higher resource limits, **EC2** offers a straightforward approach, with the drawback of constant infrastructure costs.

EKS is optimal for large-scale, continuous workloads, leveraging Kubernetes' orchestration capabilities to distribute tasks effectively. However, its high complexity and persistent costs limit its feasibility for low-utilization scenarios.

The **Local Container** setup is suitable for development and testing or for small-scale, single-host deployments but lacks the scalability and operational efficiency of cloud-based solutions.

Practitioners and developers should carefully assess workload characteristics, scalability requirements, and budget constraints to choose the most suitable architecture. In many cases, hybrid approaches combining multiple architectures can offer the best balance of cost and performance.

References

[1] AWS Lambda Pricing, Amazon Web Services, 2025. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[2] Amazon EC2 Pricing, Amazon Web Services, 2025. [Online]. Available: https://aws.amazon.com/ec2/pricing/

[3] Amazon EKS Pricing, Amazon Web Services, 2025. [Online]. Available: https://aws.amazon.com/eks/pricing/

[4] Puppeteer Documentation, Google, 2025. [Online]. Available: https://pptr.dev/

[5] Node.js Documentation, OpenJS Foundation, 2025. [Online]. Available: https://nodejs.org/

[6] Docker Documentation, Docker, Inc., 2025. [Online]. Available: https://docs.docker.com/

[7] Kubernetes Documentation, The Kubernetes Authors, 2025. [Online]. Available: https://kubernetes.io/docs/

[8] NGINX Documentation, F5, Inc., 2025. [Online]. Available: https://nginx.org/en/docs/

[9] A. Brogi, S. Forti, and A. Ibrahim, "Comparing Cost and Performance of Microservices and Serverless in AWS," in *Advances in Service-Oriented and Cloud Computing*, Springer, 2023, pp. 65–79. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-61816-1$_5$

[10] "AWS Lambda vs. Amazon EC2: Comprehensive Comparison," NAKIVO, 2023. [Online]. Available: https://www.nakivo.com/blog/aws-lambda-vs-amazon-ec2-which-one-to-choose/

[11] "Exploring AWS EKS (Elastic Kubernetes Service): Container Orchestration Made Easy", Christopher Adamson, 2023 [Online]. Available: https://medium.com/@christopheradamson253/exploring-aws-eks-elastic-kubernetes-service-container-orchestration-made-easy-5c2b6fa3dbb7