**Q1. Serverless Technologies Comparison**

**AWS Lambda:** Is an event driven, pay-as-you-go compute service that lets you run code without provisioning and managing servers.

**AWS Fargate:** Is a serverless compute engine that works with AWS ECS and EKS.

**Google Cloud Run:** Abstracts away all the infrastructure management by automatically scaling up and down from zero, almost instantaneously, depending on traffic.

**Google Cloud Functions:** Small snippets that responds to events. It can connect with Google Cloud or third-party cloud services via triggers to streamline challenging orchestration problems.

**Azure Serverless Functions:** Is an event driven compute service which supports multiple languages. It can scale on demand and only charges for the time code is executed.

**Azure Serverless Kubernetes:** Can elastically provision pods inside container instances that start in seconds without the need to manage additional compute resources.

**Features:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lambda** | **Fargate** | **GCR** | **GCF** | **ASF** | **ASK** |
| Extend with other AWS services.  Custom backend services.  Supports multiple languages.  Automated administration.  Built in fault tolerance.  Automatic scaling.  Fine grained control over performance.  Integrated security.  Pay as per use. | Compatible with ECS and EKS.  Supports AI and ML applications.  Can run data processing workloads.  Can integrate with AWS of third-party metric tools.  Improved security through workload isolation.  Pay only for resources used. | Supports multiple languages, libraries and binaries.  Pay per use model.  Fully managed.  Per instance concurrency.  Fast autoscaling.  Regional, automatically replicates across zones.  Ephemeral and persistent storage.  Integrated monitoring. | End-to-end development and diagnosability  Develop locally, scale globally  No server management.  Trigger code from GCP or any other web app via http.  Pay only for use.  Prevents lock-in using FaaS.  Can connect with GCP or third-party services. | Supports multiple programming languages.  Event driven architecture.  Integrates with Azure key vault.  Flexible pricing options.  Built in application lifecycle management.  Integrates with other Azure cloud services. | Built in best practices.  Multilayer security.  Faster diagnostics and troubleshooting.  Consistent configuration and governance across all services.  Automatic configuration analysis for risk identification.  Integrates with Azure Active Directory.  Flexible pricing plans. |

**Pros and Cons:**

|  |  |  |  |
| --- | --- | --- | --- |
| **AWS** | | | |
| **Lambda** | | **Fargate** | |
| **Pros** | **Cons** | **Pros** | **Cons** |
| Very well matured and established.  Can be pretty much seamless if already invested in AWS.  Generally lower cold start latency. | Debugging can be challenging for complex applications.  Can be quite expensive for long running or CPU intensive tasks. | Isolated execution environments for each task or pod.  No need to manage underlying EC2 instances.  Flexible scaling. | More expensive than managing EC2 instances yourself.  Requires a fair amount of configuration.  Limited customization. |

|  |  |  |  |
| --- | --- | --- | --- |
| **GCP** | | | |
| **Cloud Run** | | **Cloud Function** | |
| **Pros** | **Cons** | **Pros** | **Cons** |
| Extremely easy to deploy containerized applications.  Designed to natively handle HTTP request which is ideal for web APIs and apps.  Supports handling concurrent requests.  Supports multiple languages. | Not ideal for long running or stateful applications.  Cold start latency can be higher as per the size and complexity of applications.  Might not be suitable for non-HTTP applications. | Good integration with other GCP products.  Very easy to get started and deploy.  Supports multiple languages.  Supports modern HTTP2 protocol. | Not very mature and established as AWS.  Higher cold start latency.  Limited number of triggers as compared to AWS. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Azure** | | | |
| **Serverless Function** | | **Serverless Kubernetes** | |
| **Pros** | **Cons** | **Pros** | **Cons** |
| Ideal option for Microsoft centric application stack.  Supports multiple languages.  Supports serverless function in a serverless compute.  Suitable for designing workflows and integrations. | Debugging is challenging.  Higher cold start latency in the consumption plan.  Can be a little complex for simple applications. | Native support for windows containers.  Compatibility with Kubernetes allows smoother migration from a traditional Kubernetes setup.  Can integrate with Azure Logic Apps for workflow orchestration. | Kubernetes is complex and has a learning curve.  Pricing model is difficult to understand and cost can escalate with increased usage.  Although it supports autoscaling, it might not be as fast or as flexible compared to other solutions. |

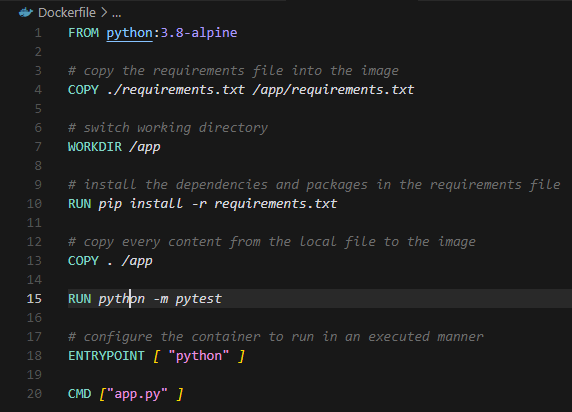
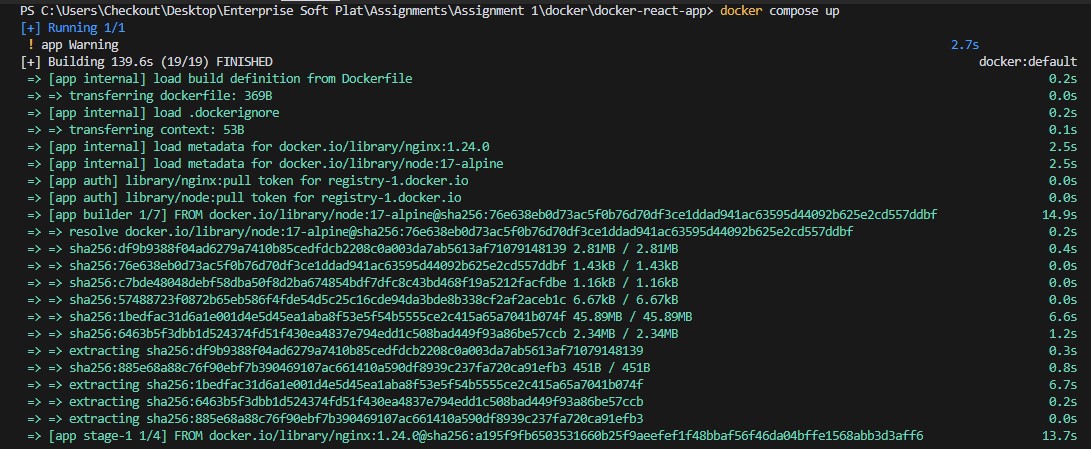
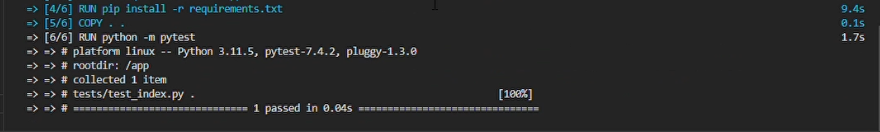
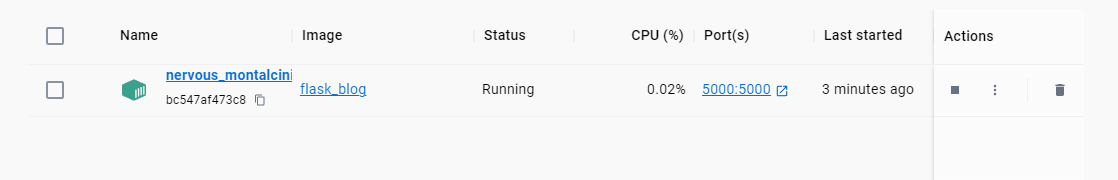
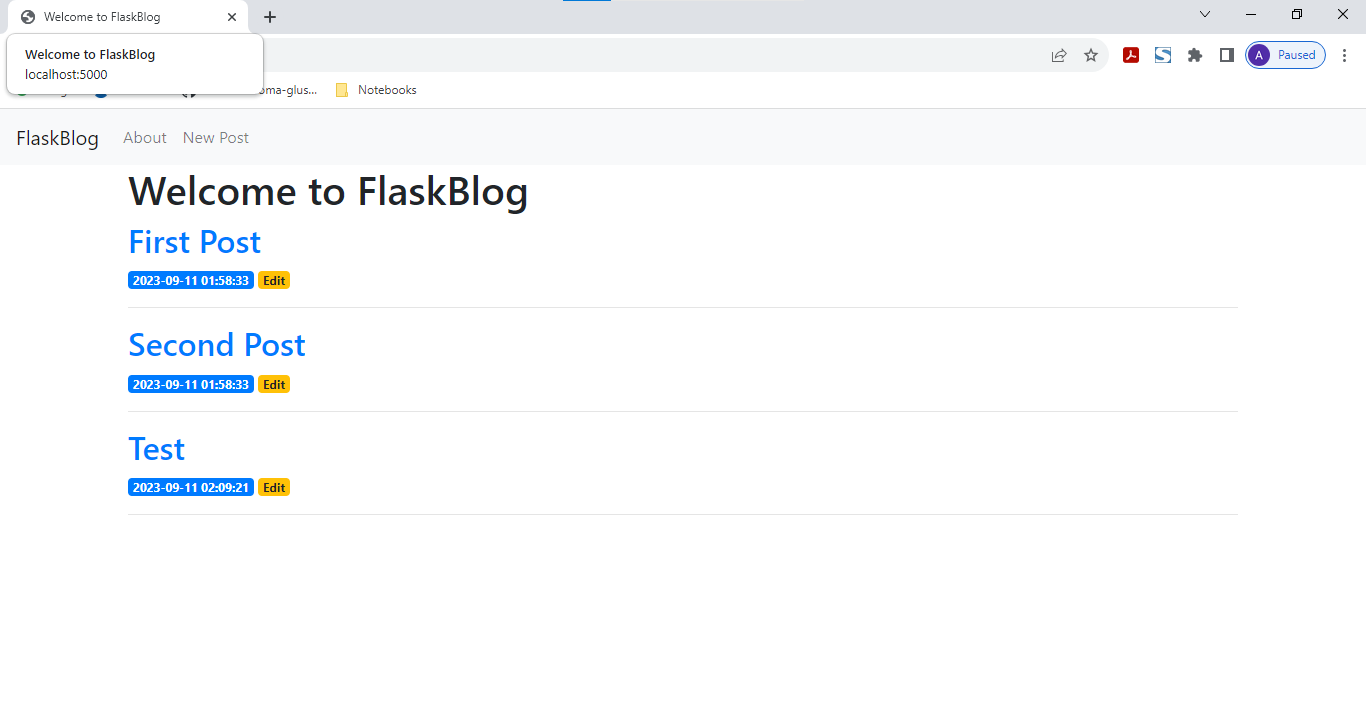
**Proposed Feature as a product manager:**

AWS Fargate is an alternative to self-managed EC2 instances. Although it relieves the user from the complexities of managing the EC2 instances, Fargate is costly. AWS tries to address this cost problem using Fargate Spot Instances but to make it optimal, it requires a substantial amount of human intervention, for example, to specify the percentage of spot instances vs the on-demand instances.

What I would suggest as a product manager for Fargate Spot Instances is a feature which could be called Fargate Spot Auto Optimization. This feature will focus on managing the percentage of Spot Instances and on demand instances with real time pricing. It will optimize costs while managing the workload stability. [This](https://tomgregory.com/aws-fargate-spot-vs-fargate-price-comparison/) blog post (November 2022) shows a demonstration of how Fargate Spot can reduce costs up to 66% which is a huge saving. Although Fargate Spot instances are saving resources in terms of money, it requires tedious and time-consuming human intervention to manage the on demand and spot instance to achieve the optimum costing. By implementing the Fargate Spot Auto Optimization feature, cost savings of 66% could go even higher up to 70%. It also removes the manual task of adjusting the spot and on demand instances making Fargate more accessible. This feature will also ensure enough on-demand instances to maintain workload stability.

**Q2. Deploy app using Docker**

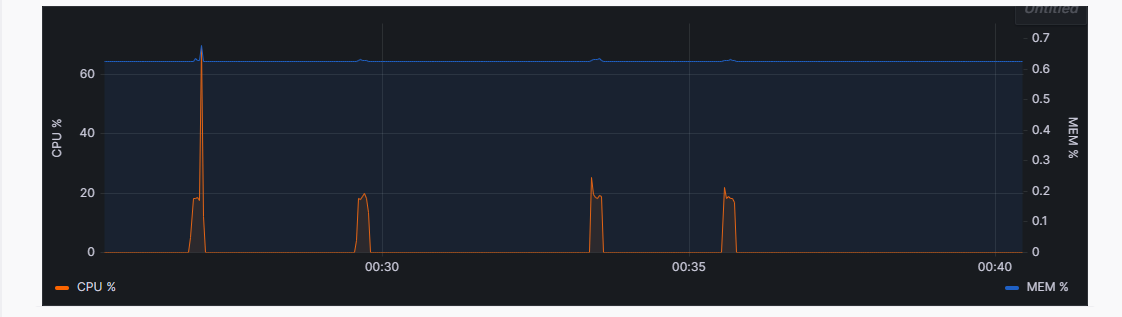
**Steps:**

1. Cloned a flask app from git `git clone https://github.com/do-community/flask\_blog.git`
2. Creating a Dockerfile  
   
3. Running `docker build -t flask\_blog .`  
   
4. It can also run tests for us  
   
5. After step 5, the container shows up in docker desktop client  
   
6. As expected, the app us running on port 5000 as specified in `Dockerfile`  
   

**Q3. Docker Container Metrics**

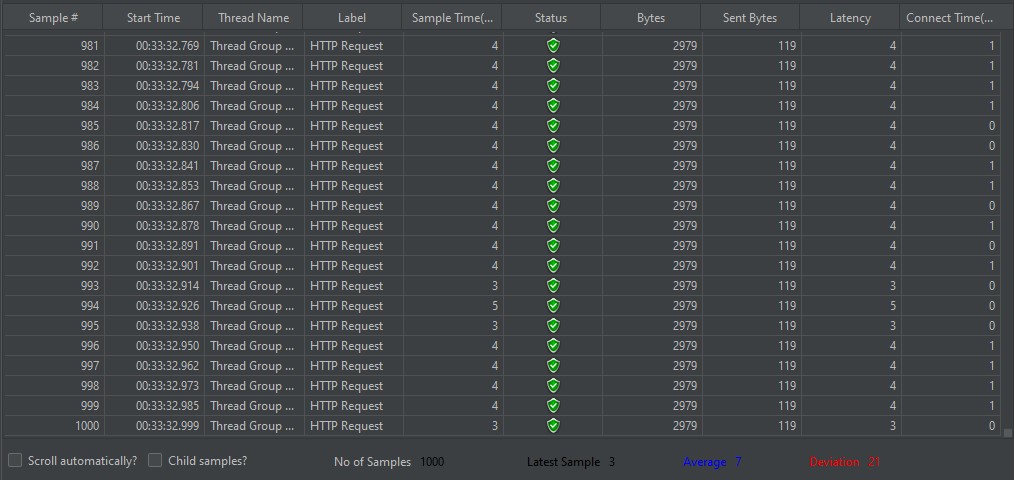
I used Apache JMeter benchmarking tool to perform load testing on HTTP requests and used an extension called Container Watch for CPU and Memory usage. My findings are as follows:

As the graph shows, CPU and Memory utilization bumps up-to more than 60% when the request simulation is carried out. After the simulation is complete, memory utilization comes down in the range of 18% - 25%.



**JMeter Load Testing:**

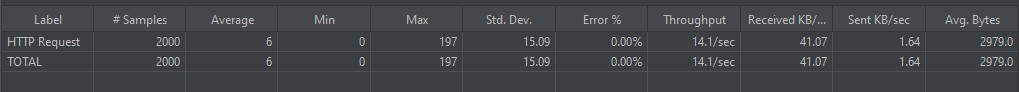
The following table shows 1000 simulated requests with the timestamp, size of the data sent, request latency, etc.

****

This table shows the summary of the above 1000 HTTP requests. Some of the important metrics are:

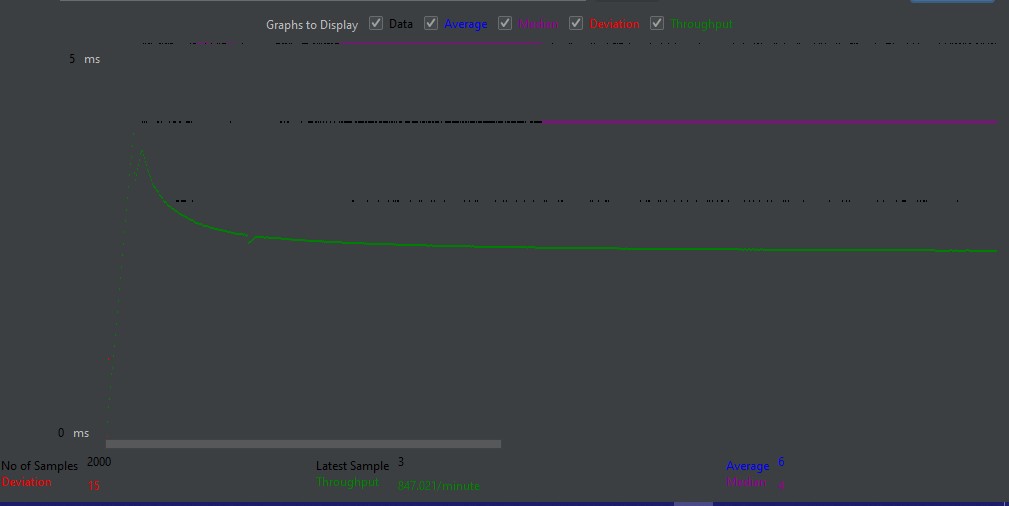
**Error %:** 0

**Throughput:** 14.1/sec

****

I carried out the simulation twice. The following graph shows the data of 2000 samples. Some of the important metrics are:

**Throughput:** 847.021/min)



**Conclusion:**

In conclusion, our comparison of Docker and Vagrant deployments for a web app highlights Docker's faster startup time as a significant advantage. However, Vagrant is more resource-efficient and offers precise environment control. The choice depends on project needs, favoring Docker for speed and scalability and Vagrant for resource-conscious scenarios.