Final Mastery Report

Ananya Hegde

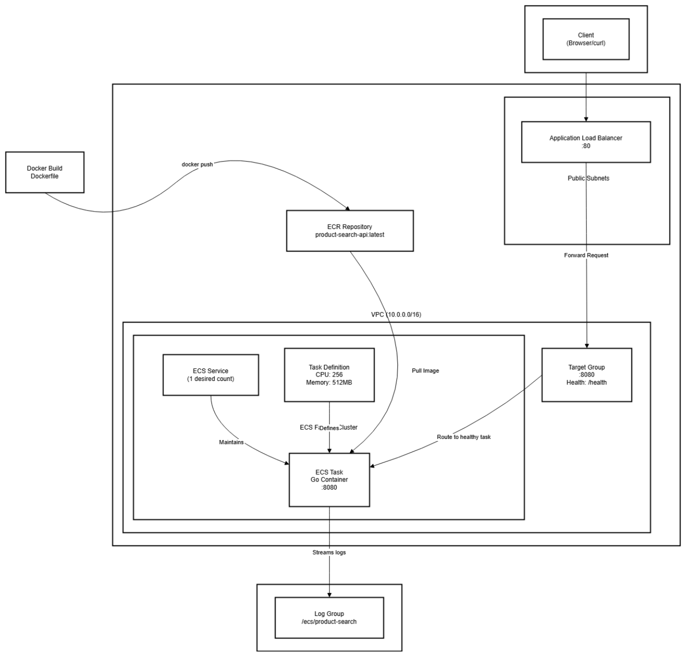
Github Repo

Overview

The project implements a Go-based product search API that searches through 100,000 in-memory products and returns relevant results. To understand how best to run this service in real-world scenarios, it was deployed and benchmarked across multiple environments - locally using Docker containers, and in the cloud using AWS ECS Fargate behind an ALB. The analysis demonstrates that local setups are ideal for fast development and debugging, and managed cloud environments are best suited for secure, scalable production workloads.

System Architecture

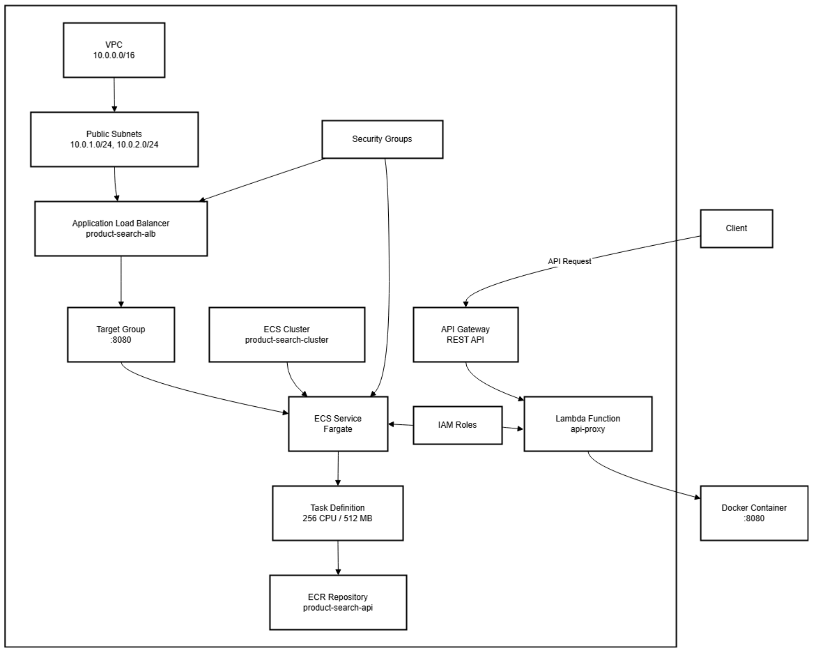
AWS



The application is hosted on AWS using a fully managed container approach. The container image used to run the API is stored in ECR, and each deployment automatically retrieves the latest version of that image. AWS performs continuous health checks to make sure only healthy application instances receive traffic. Incoming traffic first goes through an ALB, which distributes requests to the running application containers. The load balancer sits in publicly accessible subnets, while the containers run in protected private subnets. This separation improves both security and reliability.

Logging and monitoring are handled through AWS CloudWatch, and permissions for secure communication between services are managed with IAM roles. This setup offers a scalable and resilient environment with minimal operational overhead, since the platform handles the container lifecycle without needing to manage physical or virtual servers.

LocalStack

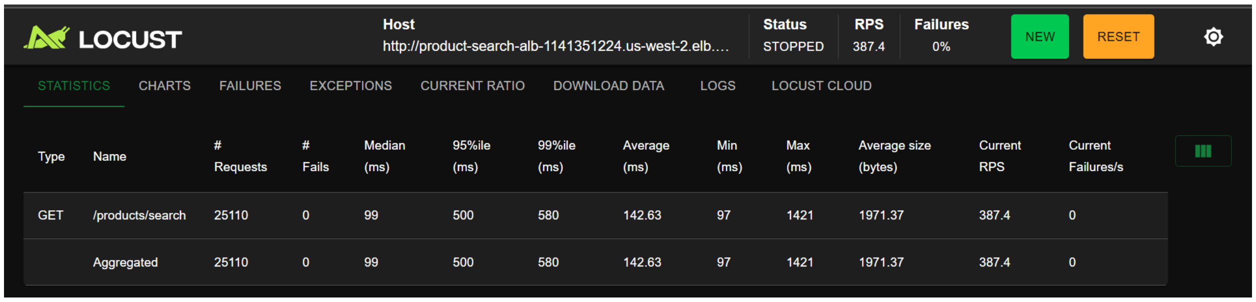


The LocalStack setup creates a local simulation of AWS services running entirely on my local machine within Docker containers. While LocalStack creates all the same AWS resources as the production environment - including the VPC, subnets, ECS cluster, load balancer, and ECR repository - the actual execution model differs significantly from real AWS. The Go application runs in a Docker container on port 8080, separate from LocalStack's simulated ECS service. To handle API requests, I configured API Gateway with a Lambda function that acts as a proxy, forwarding traffic from LocalStack's endpoint (localhost:4566) to the Go container using Docker's internal networking. The setup uses Terraform to provision all resources like in AWS, making the infrastructure code portable between local development and production. This approach allowed me to test the same Terraform configurations locally that deploy to AWS, though the actual request routing works through the Lambda proxy rather than the ECS/ALB path that would handle traffic in production.

Performance Testing

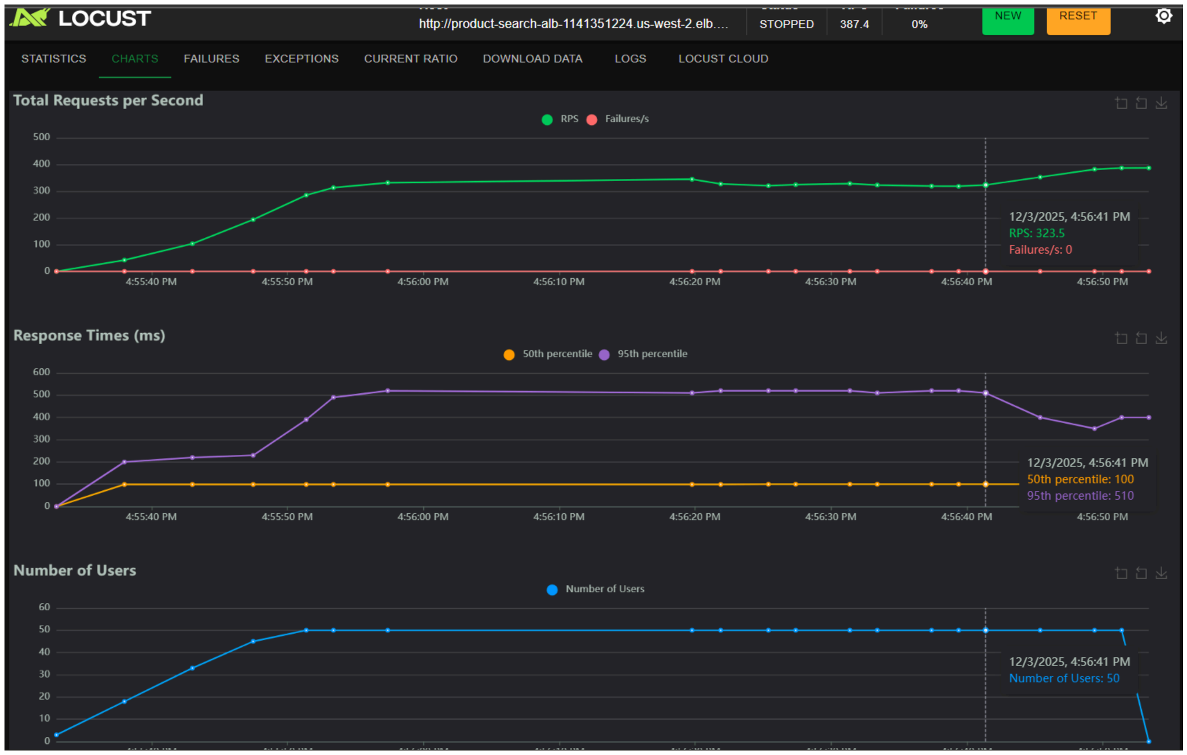
The two deployments were compared with one another based on the locust testing performance metrics for each.

AWS



Host: http://product-search-alb-1141351224.us-west-2.elb....
Status: STOPPED
RPS: 387.4
Failures: 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/products/search	25110	0	99	500	580	142.63	97	1421	1971.37	387.4	0
	Aggregated	25110	0	99	500	580	142.63	97	1421	1971.37	387.4	0

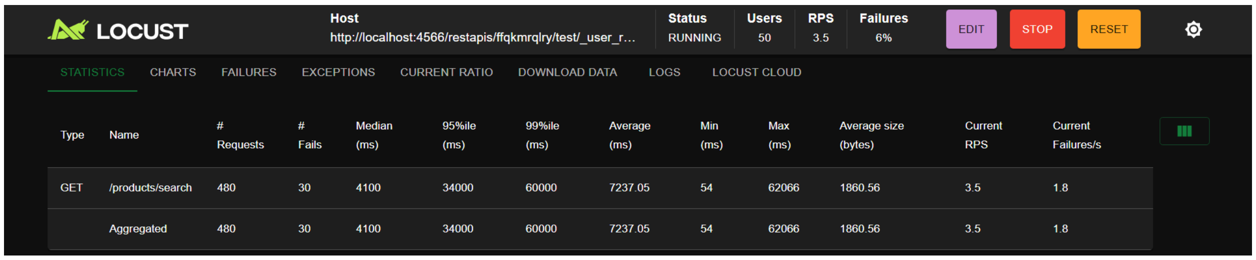


The load testing was conducted using Locust to evaluate the AWS infrastructure's performance under constant heavy traffic. The test simulated 50 concurrent users making continuous requests to the /products/search endpoint, with a ramp up speed of 3.

The system demonstrated strong performance with zero failures across 25,110 requests. It maintained a steady throughput of approximately 387 requests per second after the initial ramp-up period, showing the infrastructure could handle this load comfortably without degradation. The RPS graph remained stable throughout the test period. Response times also remained consistently good throughout the test. The median response time was 99ms, with the 95th percentile at 500ms and 99th percentile at 580ms. These metrics indicate that most users experienced fast responses, with even the slowest requests staying well under a second.

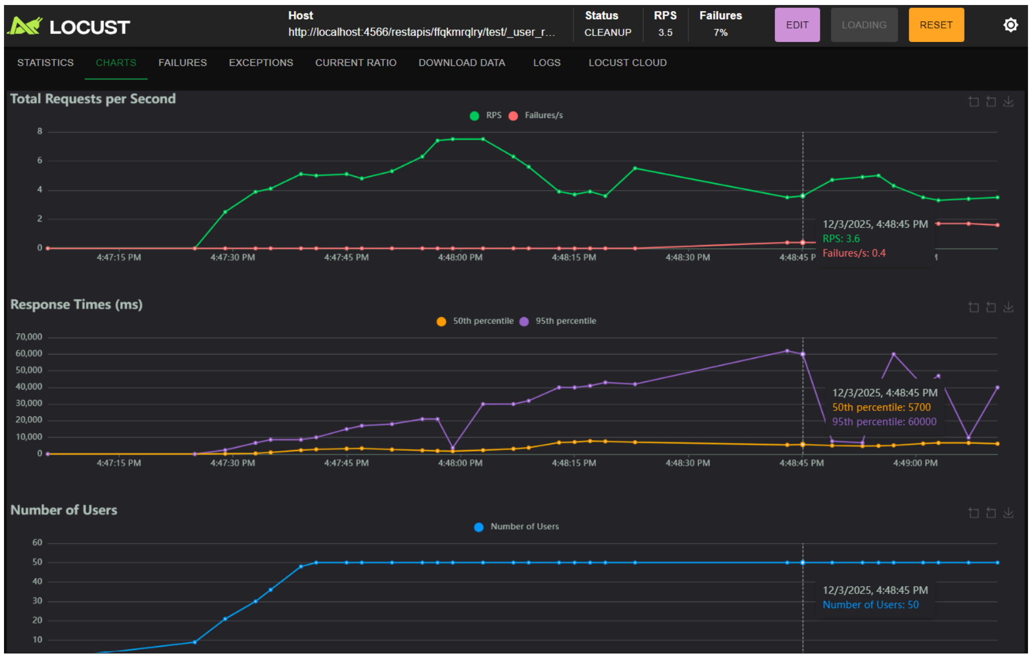
The AWS ECS infrastructure handled the load test well, delivering reliable performance without any errors or significant latency spikes under sustained traffic.

LocalStack



Host	Status	Users	RPS	Failures
http://localhost:4566/restapis/ffqkmrqlry/test/_user_r...	RUNNING	50	3.5	6%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/products/search	480	30	4100	34000	60000	7237.05	54	62066	1860.56	3.5	1.8
	Aggregated	480	30	4100	34000	60000	7237.05	54	62066	1860.56	3.5	1.8



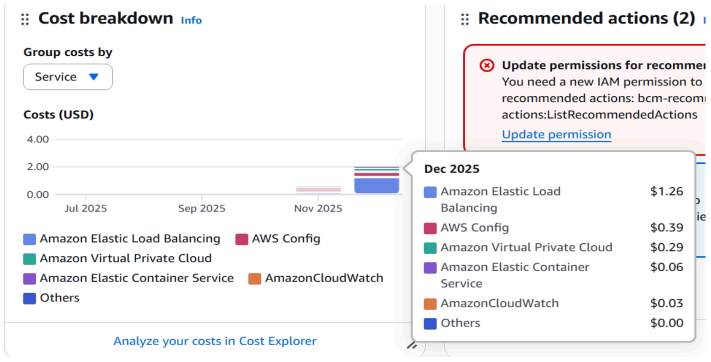
The LocalStack load testing showed significant performance degradation compared to the AWS deployment. The test attempted to simulate the same 50 concurrent users accessing the /products/search endpoint through LocalStack's API Gateway and Lambda proxy setup. The throughput graph shows highly unstable performance, with RPS fluctuating throughout the test, never reaching a stable state. Response times were extremely poor, with a median of 4100ms. The 95th percentile reached 3400 ms and the 99th percentile hit 6000 ms - meaning 1% of requests took over a minute to complete. The response time graph shows a continuous upward trend throughout the test, suggesting the system was accumulating latency and couldn't keep up with incoming requests. The test also experienced a 7% failure rate (30 out of 480 total requests), with errors including HTTP 502 Bad Gateway responses and retry failures for various search queries.

Deployment Analysis

Performance Tradeoffs

The app when deployed on AWS shows stable performance in response times, while LocalStack does not. The performance difference makes LocalStack unsuitable for any performance or load testing. AWS demonstrated no failures across 25,000+ requests with built-in redundancy across availability zones, health checks, and automatic container replacement. LocalStack showed a 6% failure rate, experiencing connection timeouts and 502 errors. AWS provides consistent, predictable performance under load, whereas LocalStack shows unstable throughput and degrading response times. This tells us that LocalStack is not the best to use when testing the performance and reliability aspects of a service.

Cost Analysis



Update permissions for recommen
You need a new IAM permission to
recommended actions: bcm-recom
actions:ListRecommendedActions

Dec 2025

Amazon Elastic Load Balancing	$1.26
AWS Config	$0.39
Amazon Virtual Private Cloud	$0.29
Amazon Elastic Container Service	$0.06
AmazonCloudWatch	$0.03
Others	$0.00

AWS services incur real costs, as can be seen in the cost breakdown. For a simple ECS setup with ALB, minimum load testing incurred around $1.26. This, when scaled to production levels of testing, would prove costly. Meanwhile LocalStack provides a free (or at least cost-effective), local emulation of AWS services, making it especially useful for early-stage testing of services like Terraform. Resources can be created, destroyed, and modified safely without worrying about billing or service limits. This shows that LocalStack is ideal for infrastructure testing and validation, while AWS should be reserved for realistic performance testing and production deployments.

Operational Overhead

LocalStack requires minimal setup, just Docker and Terraform on a local machine. However, troubleshooting is complex when things don't work as expected. AWS handles all infrastructure management and maintenance, but requires some understanding of multiple services like ECS, ALB, IAM, VPC. LocalStack is great when there is not a lot of complex routing involved, while AWS requires more careful planning but provides managed services that reduce long-term operational burden.

What To Use When

As with almost anything, it depends entirely on the use case. LocalStack is best suited for development and infrastructure testing, where the goal is to validate Terraform configurations or experiment with cloud resources at a low cost. The ability to create and destroy infrastructure almost instantly without cost makes it ideal for CI/CD pipelines and development workflows. However, load testing results show that LocalStack provides unreliable request and response performance, making it unsuitable for realistic performance testing.

AWS on the other hand, should be used at the deployment stage or whenever application behavior, performance, scalability, or reliability matters. It provides accurate load testing, exposes real network conditions, and reflects true operational cost and performance trade-offs.

To conclude, LocalStack is a cost-free development and IaC testing environment, while AWS is the only practical choice for performance evaluation and production deployment.