Part 4: Stress Test

Load testing was conducted using Locust against the AWS-deployed Product API to evaluate scalability, stability, and performance under increasing concurrency. Each test ran for approximately two minutes and simulated a realistic workload consisting of 70% GET requests to retrieve products and 30% POST requests to update product details (20% valid 204 responses and 10% 400 validation responses). This distribution reflects a typical read-heavy e-commerce environment where product lookups dominate traffic.

Under baseline load (50 users, 5 users/sec spawn rate), the system processed 14,222 total requests with an average response time of approximately 110 ms and zero failures. Throughput reached roughly 117 requests per second. Response time percentiles remained stable, with the 95th percentile under 200 ms. This indicates that at moderate load, the ECS service and ALB routing layer handled traffic efficiently with minimal latency variation.

At higher load (200 users, 20 users/sec spawn rate), total processed requests increased to 56,739, and throughput scaled proportionally to approximately 465 requests per second. Average response time increased only slightly to around 113 ms, while maintaining zero failures. The near-linear increase in throughput with minimal latency growth demonstrates that the application scales efficiently under increased concurrency and that no immediate bottleneck was encountered at this level.

Under stress conditions (1000 users, 100 users/sec spawn rate), the system processed 269,979 total requests with an aggregate throughput of approximately 2216 requests per second. The average response time increased to about 131 ms, with the 95th percentile still remaining under 200 ms and zero failures recorded. Although latency increased compared to baseline, the system remained stable and did not exhibit request errors or timeouts. This suggests that the Go-based Gin server, combined with in-memory storage and ECS infrastructure, can sustain high concurrency while maintaining predictable performance.

A comparison was also conducted using FastHttpUser at the same stress configuration (1000 users, 100 users/sec spawn rate). With FastHttpUser, the system processed 276,846 total requests at approximately 2274 requests per second. Average response time slightly improved to around

121 ms. The throughput improvement compared to the standard HttpUser configuration (~2216 RPS vs ~2274 RPS) was modest, indicating that the primary bottleneck under stress was not the Locust client but rather server-side processing, network latency, or infrastructure constraints. This aligns with expectations, as FastHttpUser typically provides larger gains when client-side request generation becomes CPU-bound.

Overall, performance scaled nearly linearly from baseline to stress levels, with no observed failures across all tests. Latency increased gradually but remained within acceptable bounds even at 1000 concurrent users. The use of an in-memory map[int]Product structure provides O(1) average lookup time, which benefits the dominant GET workload. In a production environment with a persistent database, additional latency and resource considerations would arise, potentially requiring caching layers or horizontal scaling strategies.

In conclusion, the deployed Product API demonstrates strong scalability characteristics under read-heavy workloads, stable latency under high concurrency, and effective utilization of AWS ECS infrastructure. The system sustained over 2200 requests per second without failure, indicating that the current architecture is robust for moderate-scale traffic patterns.