Martian Bank - Technical Report

CMPE 48A Term Project - Cloud-Native Architecture Implementation

Team Members: Ali Gökçek – Umut Şendağ
Course: CMPE 48A - Cloud Computing
Platform: Google Cloud Platform (GCP)
GitHub Repository: Repository-Link
Demo Video: Demo-Video-Link
Date: December 2025

Contents

1 Executive Summary

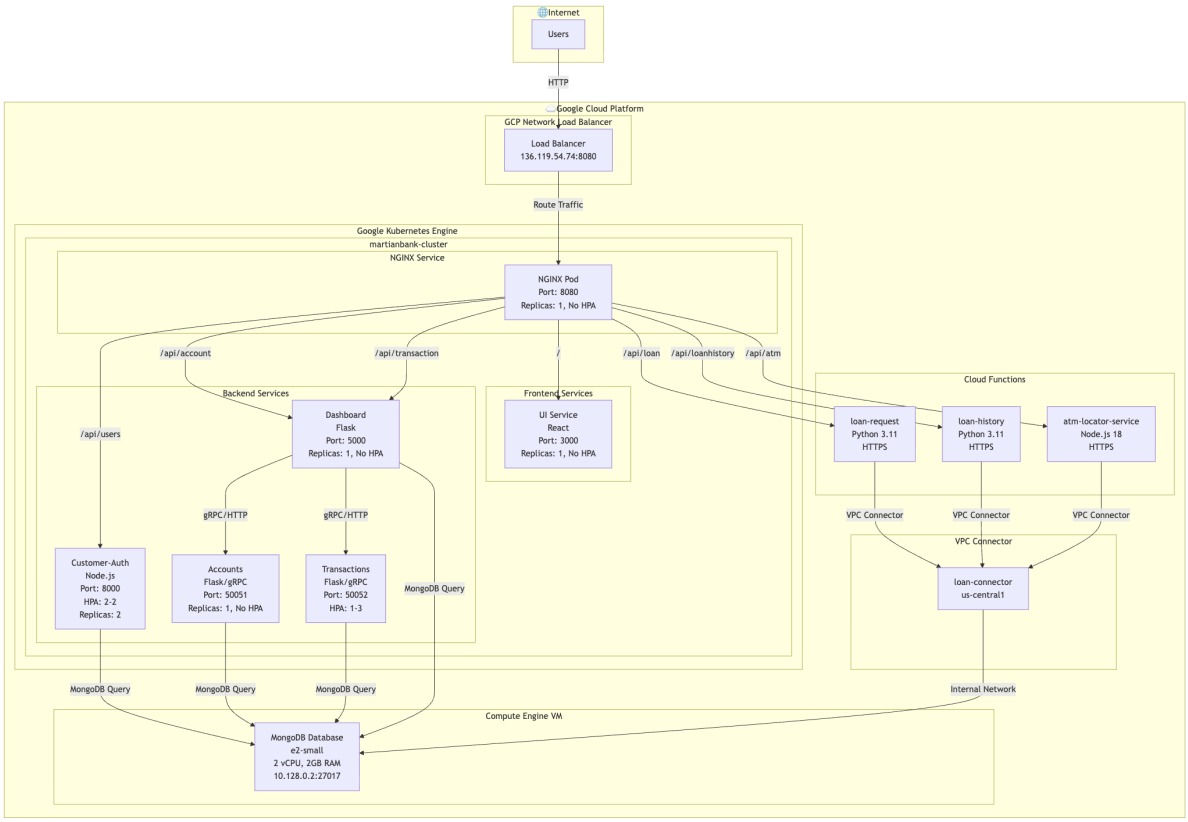
1.1 Project Overview

Martian Bank is a cloud-native microservices banking application successfully deployed on Google Cloud Platform. The system demonstrates a hybrid architecture combining Kubernetes-managed microservices with serverless Cloud Functions, all backed by a MongoDB database running on Compute Engine.

- **Objective:** Design, implement, and evaluate a cloud-native architecture integrating containerized workloads, virtual machines, and serverless functions
- **Application:** Martian Bank - A microservices-based banking platform with user authentication, account management, transaction processing, loan applications, and ATM location services
- **Cloud Platform:** Google Cloud Platform (GCP)
- **Key Technologies:** Kubernetes (GKE), Compute Engine, Cloud Functions, MongoDB, NGINX, React, Node.js, Python/Flask

1.2 Architecture Summary

- **Containerized Workloads:** 6 microservices deployed on GKE (UI, Customer-Auth, Dashboard, Accounts, Transactions, NGINX)
- **Virtual Machines:** MongoDB database on Compute Engine VM (e2-small: 2 vCPU, 2GB RAM)
- **Serverless Functions:** 3 Cloud Functions for loan request, loan history, and ATM locator services
- **Auto-scaling:** Selective HPA for transactions (1–3 replicas, CPU 50%) and customer-auth (2–2 replicas, CPU 50%)
- **Load Balancing:** GCP Network Load Balancer (Layer 4) providing external access at `136.119.54.74:8080`
- **Performance Testing:** Comprehensive Locust testing across multiple scenarios (20–500 users, various configurations)



2 Cloud Architecture Diagram

2.1 High-Level Architecture

Note: A detailed Mermaid diagram is also available in `docs/FINAL_DOCS/ARCHITECTURE_DIAGRAM.md`.
Key Architecture Features:

- NGINX acts as internal API gateway routing to microservices and Cloud Functions
- Cloud Functions access MongoDB via VPC Connector (loan-connector)
- All microservices communicate with MongoDB VM on internal network (`10.128.0.2:27017`)
- External access only through Load Balancer (`136.119.54.74:8080`)
- Selective HPA: Only transactions (1–3) and customer-auth (2–2) have auto-scaling enabled

2.2 Infrastructure Components

2.2.1 Google Kubernetes Engine (GKE)

- **Cluster Name:** martianbank-cluster
- **Location:** us-central1-a
- **Node Pool:** default-pool

- **Machine Type:** e2-medium (2 vCPU, 4GB RAM per node)
- **Node Count:** 3 (auto-scaling disabled, fixed 3 nodes)
- **Total Capacity:** 6 vCPUs total (∼2.8 vCPUs allocatable), 12GB RAM total (∼9GB allocatable)
- **Disk Size:** 40GB per node (pd-balanced)
- **Auto-repair:** Enabled
- **Auto-upgrade:** Enabled

2.2.2 Compute Engine VM

- **Instance Name:** mongodb-vm
- **Machine Type:** e2-small (2 vCPU, 2GB RAM)
- **Purpose:** MongoDB database server
- **Network Configuration:** Internal IP `10.128.0.2:27017` (no external IP)
- **Firewall Rules:** Accepts connections from GKE cluster (`10.12.0.0/14`)
- **OS:** Ubuntu 22.04 LTS

2.2.3 Cloud Functions

- **Function 1:** loan-request (Python 3.11, HTTP trigger, loan application processing)
- **Function 2:** loan-history (Python 3.11, HTTP trigger, loan history retrieval)
- **Function 3:** atm-locator-service (Node.js 18, HTTP trigger, ATM location search)
- **VPC Connector:** loan-connector (us-central1, enables MongoDB VM access)
- **Region:** us-central1

2.2.4 Load Balancer

- **Type:** Network Load Balancer (Layer 4)
- **External IP:** 136.119.54.74
- **Port:** 8080
- **Protocol:** HTTP
- **Backend:** NGINX pods in GKE cluster

3 Component Description and Interactions

3.1 Microservices Overview

Service	Technology	Port	Replicas	HPA	Purpose
UI	React + Redux Toolkit	3000	1	No	Frontend user interface
Customer-Auth	Node.js + Express	8000	2	2–2 (50%CPU)	Authentication & authorization
Dashboard	Python + Flask	5000	1	No	Orchestration layer for Accounts/Transactions
Accounts	Python + Flask + gRPC	50051	1	No	Account management (HTTP/gRPC)
Transactions	Python + Flask + gRPC	50052	1–3	1–3 (50% CPU)	Transaction processing (HTTP/gRPC)
NGINX	NGINX	8080	1	No	Internal API Gateway / Reverse Proxy

NGINX Routing:

- `/` → UI (3000)
- `/api/users` → Customer-Auth (8000)
- `/api/account` → Dashboard (5000)
- `/api/transaction` → Dashboard (5000)
- `/api/loan` → Cloud Function (loan-request)
- `/api/loanhistory` → Cloud Function (loan-history)
- `/api/atm` → Cloud Function (atm-locator-service)

3.2 Database Component

MongoDB on Compute Engine:

- **VM:** e2-small (2 vCPU, 2GB RAM), Internal IP: `10.128.0.2:27017`
- **Database:** bank (collections: users, accounts, transactions, loans, atms - 13 ATM records)
- **Access:** Internal only, authenticated (root user), firewall allows GKE cluster (`10.12.0.0/14`)

3.3 Serverless Functions

Function	Runtime	URL	Purpose
loan-request	Python 3.11	https://loan-request-gcb4q3froa-uc.a.run.app	Process loan applications
loan-history	Python 3.11	https://loan-history-gcb4q3froa-uc.a.run.app	Retrieve loan history
atm-locator-service	Node.js 18	https://atm-locator-service-gcb4q3froa-uc.a.run.app	Search ATMs

Common Configuration: VPC Connector (loan-connector), MongoDB connection via VPC

4 Deployment Process

4.1 Prerequisites

Martian Bank is deployed on GCP using a hybrid architecture: core services run on GKE and serverless components run on Cloud Functions (Gen 2), both using a shared MongoDB backend reachable only through private networking.

4.1.1 GCP Project & Billing

- Create or select a GCP project and ensure billing is enabled.
- Enable the required GCP APIs (Container, Compute, Cloud Functions, Cloud Run, Cloud Build, Artifact Registry, and Serverless VPC Access).

Listing 1: Enable required GCP APIs

```
gcloud services enable \
  container.googleapis.com \
  compute.googleapis.com \
  cloudfunctions.googleapis.com \
  run.googleapis.com \
  cloudbuild.googleapis.com \
  artifactregistry.googleapis.com \
  vpcaccess.googleapis.com

gcloud config get-value project
```

4.1.2 Local Tooling

- Install and configure: `gcloud`, `kubectl`, `helm`
- Verify installations before continuing.

Listing 2: Install/verify gcloud, kubectl, helm

```
# gcloud SDK
curl https://sdk.cloud.google.com | bash
exec -l $SHELL

# kubectl
gcloud components install kubectl

# Helm
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

# Verify
gcloud version
kubectl version --client
helm version
```

4.1.3 Docker

- Docker is required to build and push service images.
- Configure Docker authentication for pushing images to Google registries.

Listing 3: Configure Docker authentication

```
gcloud auth configure-docker
```

4.2 Infrastructure Setup

4.2.1 Step 1 – MongoDB VM Setup

MongoDB is hosted on a private Compute Engine VM (no public IP). This design isolates the database from the public internet and forces all access through GCP internal networking.

Listing 4: Create MongoDB VM

```
gcloud compute instances create mongodb-vm \
  --zone=us-central1-a \
  --machine-type=e2-small \
  --image-family=ubuntu-2204-lts \
  --image-project=ubuntu-os-cloud \
  --boot-disk-size=40GB

gcloud compute ssh mongodb-vm --zone=us-central1-a
# Install MongoDB Community Edition, enable auth, bind appropriately, create admin user.
# Further scripts for DB configuration can be found on README.md on the repository.
```

4.2.2 Step 2 – Firewall Rules (Critical)

Firewall rules are configured so that MongoDB accepts traffic only from trusted internal ranges (e.g., the GKE cluster Pod/Node CIDRs). This prevents external connections and reduces the attack surface.

Listing 5: Create Firewall Rule

```
gcloud compute instances add-tags mongodb-vm \
  --zone=us-central1-a \
  --tags=mongodb

gcloud compute firewall-rules create allow-mongodb-from-gke \
  --network=default \
  --direction=INGRESS \
  --rules=tcp:27017 \
  --source-ranges=10.12.0.0/14 \ # GKE IP range
  --target-tags=mongodb
```

4.2.3 Step 2.5 – Serverless VPC Access Connector (Required)

Cloud Functions (Gen 2) can only reach a private MongoDB VM when deployed with a Serverless VPC Access connector.

Listing 6: Create Serverless VPC Access Connector

```
gcloud compute networks vpc-access connectors create loan-connector \
  --region=us-central1 \
  --subnet=default \
  --min-instances=2 \
  --max-instances=3
```

4.2.4 Step 3 – Build & Push Docker Images

All core services are containerized and pushed to a GCP container registry (e.g., Artifact Registry or GCR), then referenced by the Helm chart during deployment.

Listing 7: Docker authentication and image push (conceptual)

```
# Authenticate (once)
gcloud auth configure-docker

# Build & push per service (example pattern)
docker build -t REGION-docker.pkg.dev/$PROJECT_ID/REPO/service:TAG .
docker push REGION-docker.pkg.dev/$PROJECT_ID/REPO/service:TAG
```

4.3 Application Deployment

4.3.1 Step 4 – GKE Cluster

Listing 8: Create GKE cluster

```
gcloud container clusters create martianbank-cluster \
  --zone=us-central1-a \
  --num-nodes=3 \
  --machine-type=e2-medium \
  --disk-size=40GB

gcloud container clusters get-credentials martianbank-cluster --zone=us-central1-a
```

4.3.2 Step 5 – Deploy GKE Services with Helm

GKE microservices are deployed with Helm for repeatability and configuration management.

Listing 9: Deploy Martian Bank with Helm

```
kubectl create namespace martianbank

helm install martianbank ./martianbank \
  --namespace martianbank \
  --set SERVICE_PROTOCOL=http \
  --set DB_URL="mongodb://root:PASSWORD@VM_IP:27017/bank?authSource=admin" \
  --set mongodb.enabled=false \
  --set nginx.enabled=true
```

4.3.3 Step 6 – Deploy Cloud Functions (Gen 2)

Serverless components (loan and ATM services) are deployed as Cloud Functions (Gen 2). When they need database access, they are deployed with the VPC connector.

Listing 10: Cloud Functions (Gen 2) deployment example

```
gcloud functions deploy loan-request \
  --gen2 \
  --runtime=python311 \
  --region=us-central1 \
  --source=. \
  --entry-point=process_loan_request \
  --trigger-http \
  --allow-unauthenticated \
  --vpc-connector=loan-connector \
  --set-env-vars="DB_URL=mongodb://root:PASSWORD@VM_IP:27017/bank?authSource=admin"
```

4.4 Autoscaling, Access, and Operations

4.4.1 Horizontal Pod Autoscaling (HPA)

To handle varying load conditions, HPAs are configured for selected GKE services. In this project, HPAs are created manually via `kubectl` (not through Helm charts).

Listing 11: Install Metrics Server (required for HPA)

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/
    download/components.yaml
kubectl get deployment metrics-server -n kube-system
```

Listing 12: Create HPAs for transactions and customer-auth

```
kubectl autoscale deployment transactions -n martianbank \
  --min=1 \
  --max=3 \
  --cpu=50%

kubectl autoscale deployment customer-auth -n martianbank \
  --min=2 \
```

```
  --max=2 \
  --cpu=50%

kubectl get hpa -n martianbank
```

4.4.2 Access Application

Listing 13: Get external IP for NGINX entrypoint

```
kubectl get service nginx -n martianbank -w
```

Open:

Listing 14: Application URL format

```
http://EXTERNAL_IP:8080
```

4.4.3 Uninstall / Cleanup

Listing 15: Uninstall helm release and namespace cleanup

```
helm uninstall martianbank -n martianbank
kubectl delete namespace martianbank
```

4.5 Verification Steps

1. Check Pod Status:

```
kubectl get pods -n martianbank
# Expected: All 6 + 1 pods in Running state
```

2. Verify Services:

```
kubectl get services -n martianbank
# Expected: nginx service with EXTERNAL-IP assigned (136.119.54.74)
```

3. Test Endpoints:

```
# Test UI
curl http://136.119.54.74:8080/

# Test authentication
curl -X POST http://136.119.54.74:8080/api/users/auth \
  -H "Content-Type: application/json" \
  -d '{"email":"test@example.com","password":"password"}'

# Test ATM locator
curl -X POST http://136.119.54.74:8080/api/atm \
  -H "Content-Type: application/json" \
  -d '{}'
```

4. Check Cloud Functions:

```
# Test loan request
curl -X POST https://loan-request-gcb4q3froa-uc.a.run.app \
  -H "Content-Type: application/json" \
  -d '{"name":"Test","email":"test@example.com",...}'

# Test ATM locator
curl -X POST https://atm-locator-service-gcb4q3froa-uc.a.run.app/api/atm \
  -H "Content-Type: application/json" \
  -d '{}'
```

5. Verify Database Connectivity:

```
# Test MongoDB connection from a pod
kubectl run -it --rm mongo-test --image=mongo:latest --restart=Never -n martianbank -- \
  mongosh "mongodb://root:PASSWORD@10.128.0.2:27017/bank?authSource=admin"
```

6. Verify HPAs:

```
kubectl get hpa -n martianbank
# Expected: transactions and customer-auth HPAs active
```

5 Locust Experiment Design

5.1 Testing Framework

- **Tool:** Locust
- **Purpose:** Simulate realistic user behavior and generate traffic
- **Test Duration:** 5 minutes per scenario
- **Metrics Collected:** RPS, average response time, error rates, resource utilization, number of replicas

5.2 Test Design

Test Approach: Performance testing uses `comprehensive_system_test.py` to simulate complete user journeys through the Martian Bank system. Tests are executed via `run_custom_simulation.py` wrapper script with interactive or command-line configuration.

User Journey: Registration → Login → Account Creation → Account Viewing → Transactions → Transaction History → ATM Search → Loan Application → Loan History → Profile Updates → Session Management

Task Weights: `view_account_details` (20), `view_all_accounts` (15), `check_transaction_history` (12), `internal_transfer` (10), `search_atm_locations` (5), `check_loan_history` (4), `apply_for_loan` (3), `update_profile` (2), `logout_and_login` (1)

Components Tested: Customer-Auth (GKE), Accounts (GKE), Transactions (GKE), Loan Services (Cloud Function), ATM Locator (Cloud Function), NGINX Load Balancer, MongoDB VM

User Personas: Casual Banking User (70%, 5–15s wait), Heavy Transaction User (30%, 1–3s wait)

5.3 Example Test Execution Scripts

Interactive Mode:

```
cd performance_locust
python run_custom_simulation.py
```

Command-Line Mode:

```
cd performance_locust
python run_custom_simulation.py \
  --users 200 \
  --spawn-rate 10 \
  --duration 5m \
  --name baseline_test
```

Results: Saved to `performance_locust/results/[test_name]/` as HTML reports and CSV files (`*_stats.csv`, `*_stats_history.csv`, `*_failures.csv`)

5.4 Independent Variables

The following independent variables were identified and tested:

Variable	Values Tested	Description
Concurrent Users	20, 50, 200, 500	Number of simultaneous users
Spawn Rate	5, 10, 20 users/second	Rate at which users are spawned (ramp-up speed)
Test Duration	5 minutes	Duration of each test scenario
HPA CPU Threshold	50%, 70%	CPU utilization threshold for auto-scaling
HPA Min/Max Replicas	Various (1–1, 1–2, 1–3, 1–5, 1–10, 2–2)	HPA scaling range for different services
VM Resources	e2-small (2 vCPU, 2GB), e2-custom-6-8192 (6 vCPU, 8GB)	MongoDB VM machine type
Node Autoscaling	Enabled/Disabled	GKE cluster node auto-scaling
DB Pool Size	Default (100), 200	MongoDB connection pool size
bcrypt Salt Rounds	8, 10	Password hashing computational cost

Initial Configuration (Baseline):

- VM: e2-small (2 vCPU, 2GB RAM)
- DB pool size: Default (100)
- Hashing auth (bcrypt): 10 rounds
- HPA min/max: 1/1 (no HPA initially)
- Node autoscaling: Off
- HPA CPU threshold: 70%

5.5 Dependent Variables

Variable	Metric	Unit
Response Time	p50, p95, p99 latency	ms
Throughput	Requests per second	RPS
Error Rate	Percentage of failed requests	%
CPU Utilization	Average CPU usage	%
Memory Utilization	Average memory usage	%
Pod Scaling Events	Number of replicas over time	count

5.6 Test Scenarios

Multiple experiments were conducted by changing **one major factor at a time** (ramp-up rate, HPA settings, VM resources, cluster autoscaling, bcrypt cost, DB pool size), while keeping the rest of the system configuration stable. Scenario names follow the convention `[phase]_[users]_[spawn_rate]` (e.g., `initial_200_10`).

5.6.1 Experiment 1: Reduce ramp-up (baseline comparison)

Scenarios:

- **initial_200_10:** 200 users, 10 users/sec (initial baseline)
- **initial_200_5:** 200 users, 5 users/sec (**only change:** ramp-up 10 → 5)

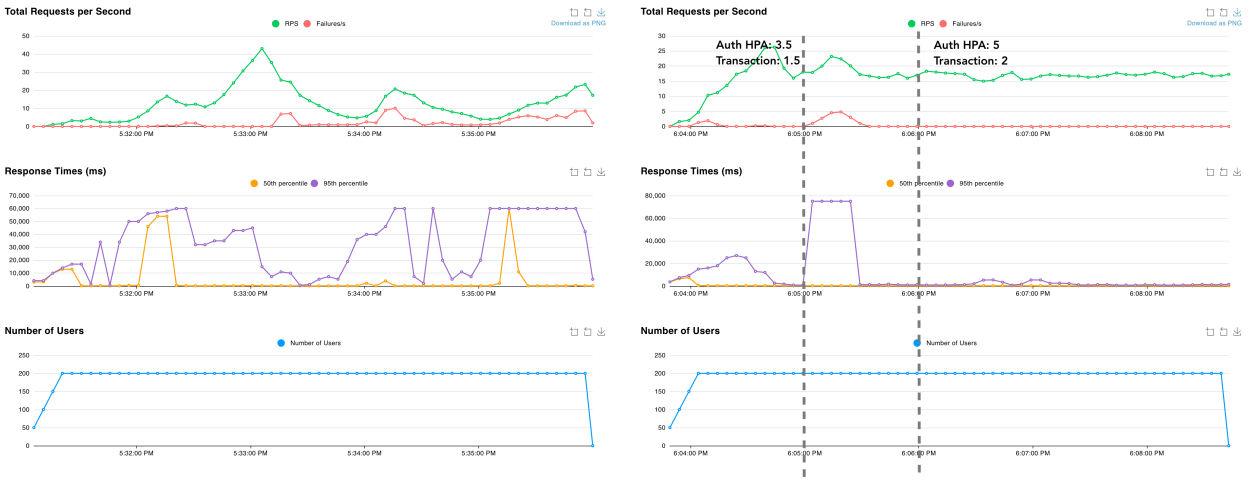
Results:

- **Throughput (RPS):** 12.56 → 12.63 (nearly unchanged)
- **Overall failure rate:** 16.18% → 12.76% (improved with slower ramp-up)
- **Latency:** Median (p50) 220ms → 210ms; aggregated p95 = 60,000ms in both runs (severe long-tail)
- **Primary failure source (Transactions):**
  - Internal Transfer failures: 80.44% (292/363) → 69.06% (212/307)
  - Transaction History failures: 66.52% (292/439) → 65.84% (266/404)
- **Auth behavior:** Login remains very slow (avg 38.84s → 35.74s); Register failures improved 11.50% (23/200) → 2.00% (4/200)

HPA Experiments

5.6.2 Experiment 2: Enable HPA (baseline ramp-up)

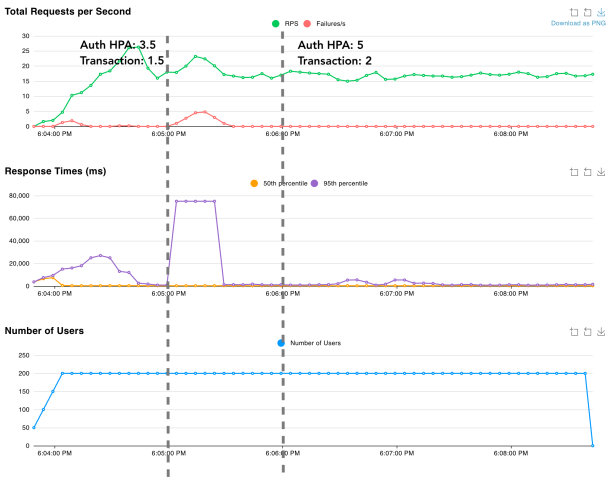
- **Test:** `200_10`
- **Changes vs initial:**
  - Customer-auth HPA: min 1, max 5
  - Transactions HPA: min 1, max 2



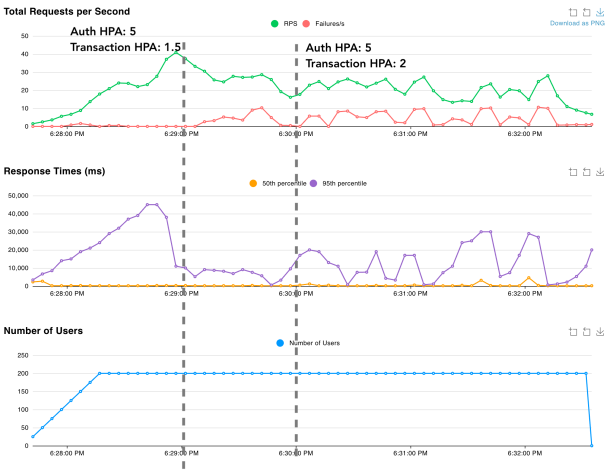
Initial (no HPA): `initial_200_10` **Enable HPA:** `HPA_enabled_200_10`

Figure 1: Experiment 2 graph comparison.

- **Results & inferences:**
  - **System stability improved significantly:** throughput increased and overall failure rate dropped sharply vs `initial_200_10`; transaction-related failures were largely eliminated.
  - **Auth improved but shifted the bottleneck:** login latency improved substantially, while **registration became the dominant issue** (high failures / long tail), indicating the critical path is not solved purely by adding auth replicas.
  - **HPA cannot fully match fast ramp-up:** in the first ∼1–2 minutes, existing pods absorb the burst while HPA reacts, causing a brief failures/s spike and elevated p95 before replicas reach their maxima.
  - **Practical takeaway:** for bursty ramp-up, we will test higher auth `minReplicas` and slower spawn rate; CPU-based HPA is reactive and will always lag initial spikes.



HPA, baseline ramp-up: `200_10`

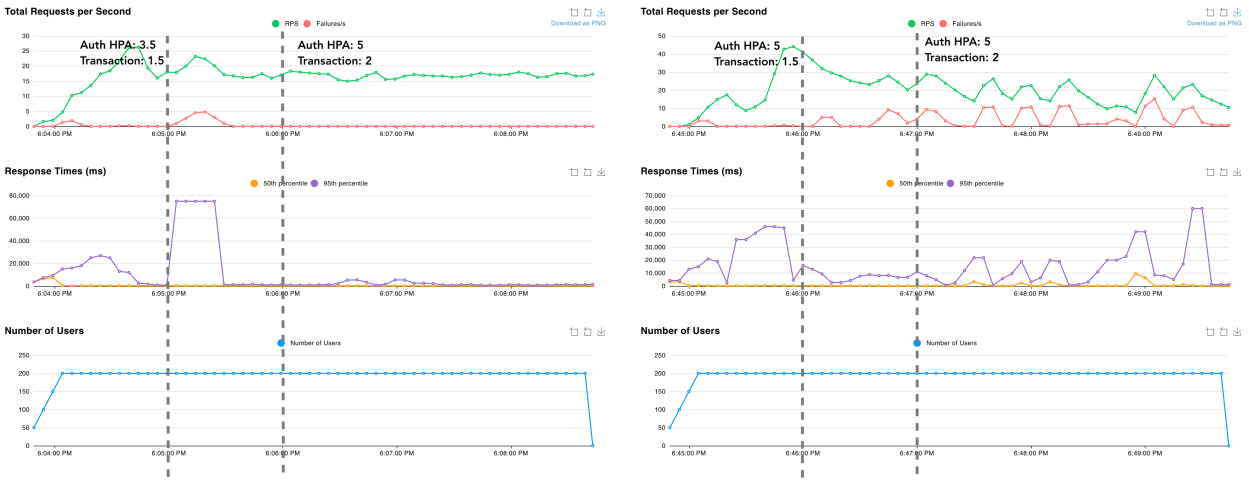


HPA, reduced ramp-up: `200_5`

Figure 2: Experiment 2.1: impact of slower ramp-up under identical HPA settings.

5.6.3 Experiment 2.1: Enable HPA with Reduced Ramp-up

- **Test:** `200_5`
- **Changes vs Experiment 2: only** spawn rate reduced (`200_10` → `200_5`); HPA settings unchanged.
- **Results & inferences (vs `enableHPA_podAuth5_podTrans2_200_10`):**
  - **Throughput increased:** aggregated **RPS** 16.69 → 20.51.
  - **Overall reliability worsened:** failure rate **2.13%** → **15.9%**, driven by **Transactions** (Internal Transfer and View History timeouts/failures).
  - **Auth registration improved:** Register failures **50%** → **7.5%**, consistent with giving HPA more time to scale during ramp-up.
  - **Ramp-up is not the only limiter:** despite reaching max replicas, **transaction failures persisted** and tail latency remained unstable, pointing to a backend constraint (e.g., DB/connection pool/timeouts) and/or `maxReplicas=2` being insufficient under sustained load.

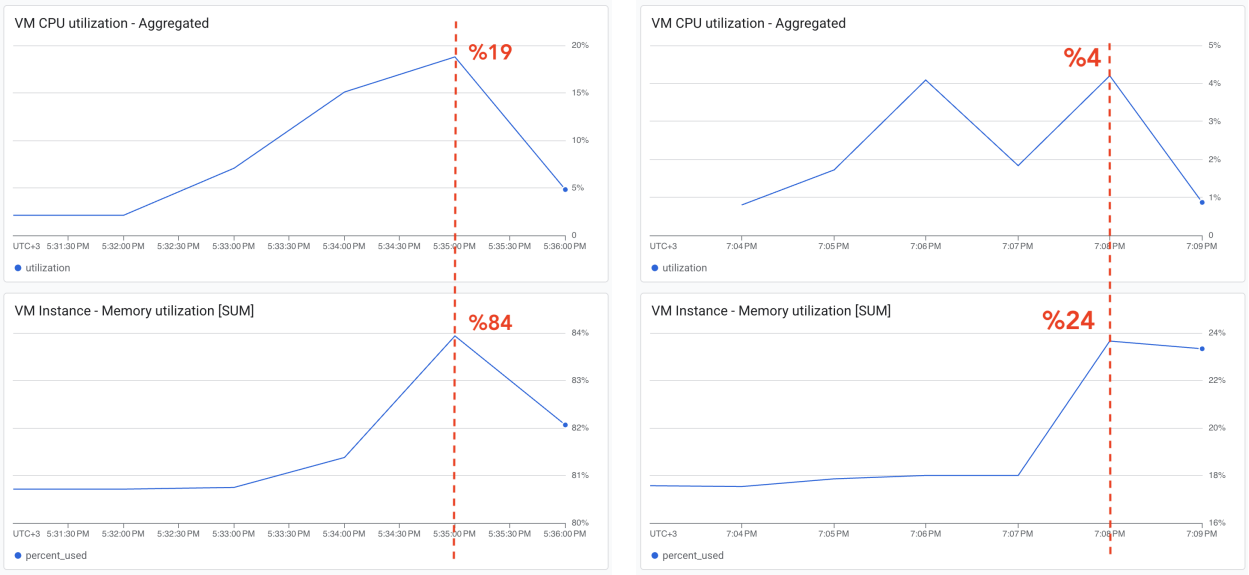


CPU target 70%: Experiment 2 **CPU target 50%:** Experiment 2.2

Figure 3: Experiment 2.2: lowering HPA CPU target.

5.6.4 Experiment 2.2: Enable HPA with Lower CPU Target

- **Test:** `200_10`
- **Change vs Experiment 2:** HPA target CPU utilization **70%** → **50%** (more aggressive scaling); min/max replicas unchanged.
- **Results & inferences (vs `enableHPA_podAuth5_podTrans2_200_10`):**
  - **Apparent throughput increased:** aggregated **RPS** 16.69 → 19.79, but this includes failed requests (not a pure win).
  - **Reliability degraded sharply:** failure rate **2.13%** → **16.69%**.
  - **Regression concentrated in Transactions:** Internal Transfer failures 0/604 → 543/690; View History failures 1/739 → 413/757.
  - **Auth registration improved (still slow):** Register failures 50% → 11%, consistent with earlier auth scaling (3.5 pods vs 5 pods) under burst.
  - **Main takeaway:** lowering CPU target triggers earlier scaling, but **does not resolve the bottleneck**; transaction failures persist after scaling, pointing to a backend limit (e.g., DB/connection pool/timeouts) and/or insufficient transaction capacity at `maxReplicas=2`.



Initial VM (2 vCPU, 2GB): peak ∼19% CPU, ∼84% memory

Upgraded VM (6 vCPU, 8GB): peak ∼4% CPU, ∼24% memory

Figure 4: Experiment 3: VM resource utilization comparison.

Infrastructure Optimization Experiments

5.6.5 Experiment 3: Increase VM Resources (MongoDB)

- **Test:** `200_10`
- **Change vs initial:** MongoDB VM upgraded from `e2-small` (2 vCPU, 2GB) to `e2-custom-6-8192` (6 vCPU, 8GB).
- **Results & inferences (vs `initial_200_10`):**
  - **Throughput improved:** aggregated **RPS** 12.56 → 15.94.
  - **Failure rate changed little:** 16.18% → 15.90% (minor improvement).
  - **Latency improved:** aggregated avg 9.53s → 6.97s; median 220ms → 200ms; p95 60s → 53s (long-tail remains severe).
  - **Transactions still dominate failures:** Internal Transfer ≈80% failures and View History ≈66% failures in both runs, although average transaction latency decreased.
  - **VM utilization indicates the DB VM was not saturated:** initial VM peaked at **∼19% CPU / ∼84% memory**, while the upgraded VM peaked at only **∼4% CPU / ∼24% memory**. Therefore, increasing VM resources had a bounded effect; persistent failures likely come from constraints outside raw VM CPU/memory (e.g., timeouts/connection handling/application-level bottlenecks).

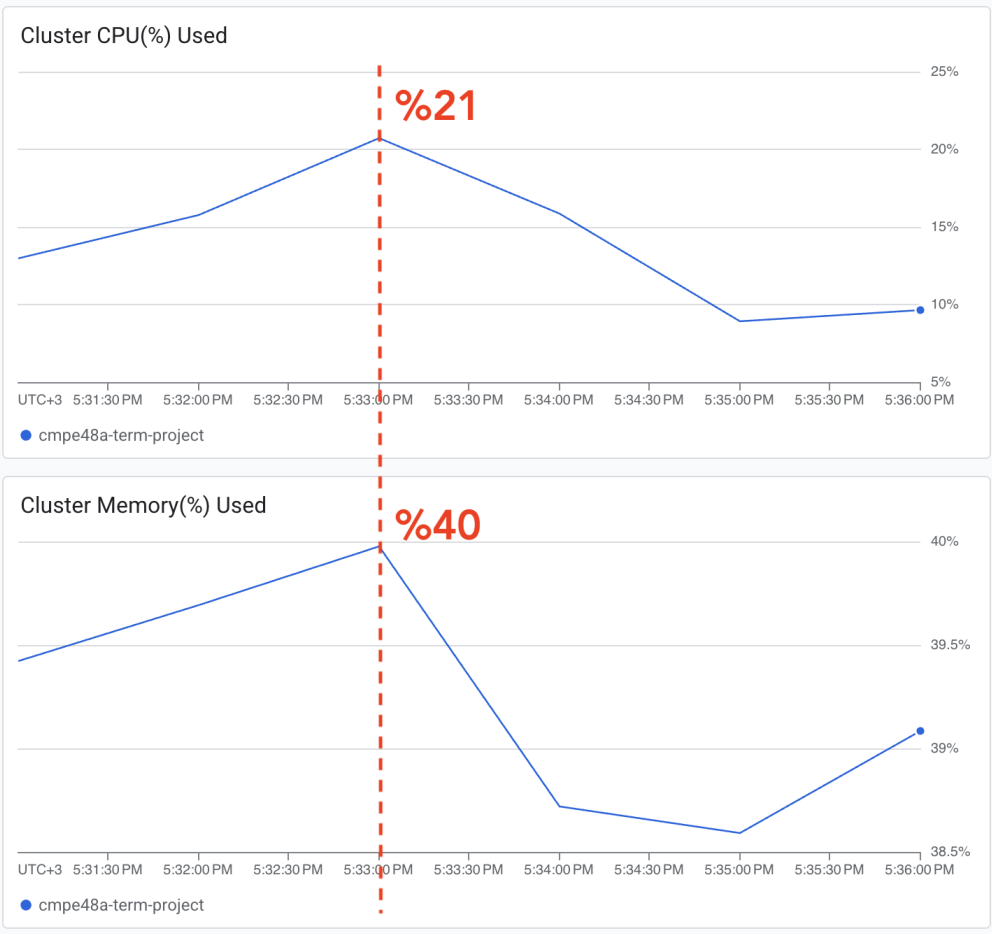
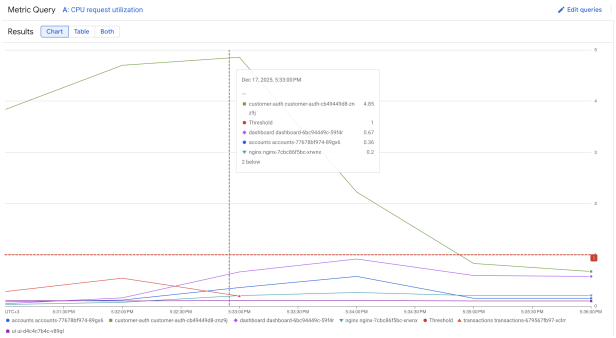


Figure 5: Experiment 4: Cluster resource utilization during `initial_200_10` with no-node autoscaling.

5.6.6 Experiment 4: Enable GKE node autoscaling

- **Test:** `200_10`
- **Changes vs initial:** Cluster node autoscaling enabled
- **Results & inferences (vs `initial_200_10`):**
  - **Throughput improved:** aggregated **RPS** 12.56 → 15.86.
  - **Failure rate improved slightly:** 16.18% → 15.66% (still high).
  - **Latency improved moderately:** aggregated avg 9.53s → 6.81s; median 220ms → 200ms; p95 60s → 43s (long-tail persists).
  - **Transactions still dominate failures:** Internal Transfer failures 292/363 → 382/450; View History failures 292/439 → 356/554.
  - **Cluster utilization stayed low:** peak **∼21% CPU** and **∼40% memory**, so autoscaling likely did not materially increase effective capacity; the bottleneck appears to be application/backend constraints rather than node resources.



Baseline auth CPU: `initial_200_10` **%485 at max**

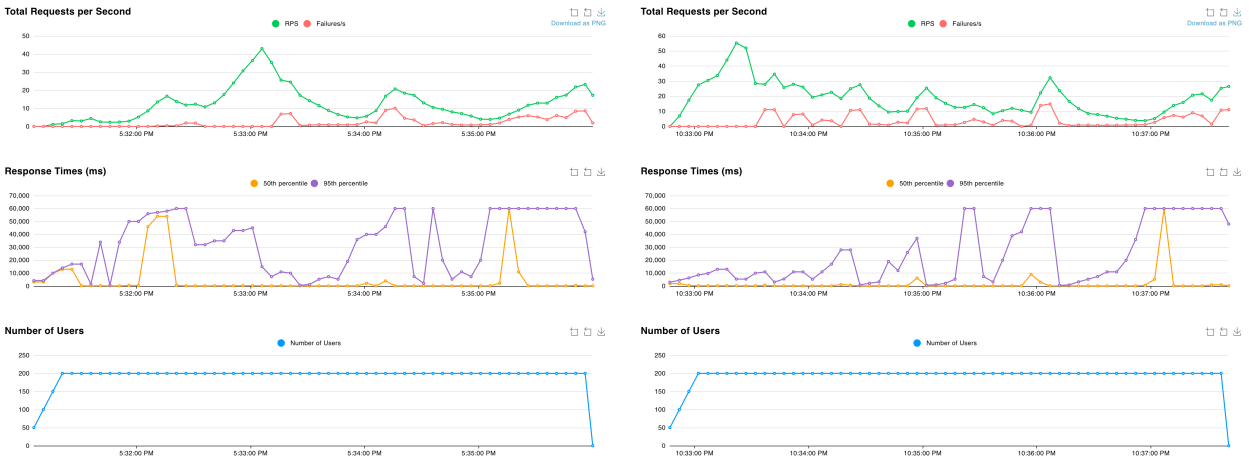
bcrypt=8 auth CPU: `bcrypt_8_200_10` **%239 at max**

Figure 6: Experiment 5: customer-auth CPU request utilization comparison.

Application-Level Performance Optimizations (5.x Series)

5.6.7 Experiment 5: Reduce bcrypt cost factor (10 → 8)

- **Test:** `200_10`
- **Changes vs initial:** bcrypt rounds in customer-auth reduced from **10** to **8**.
- **Results & inferences (vs `initial_200_10`):**
  - **Auth became dramatically faster and more reliable:** Login avg **38.84s** → **5.16s** (median 42s → 5.2s); Register failures **11.5%** → **0%** and Register avg **48.0s** → **8.24s**.
  - **Auth CPU pressure dropped:** customer-auth CPU request utilization peak decreased from about **4.85** (baseline) to about **2.40** (bcrypt=8), indicating bcrypt was a significant CPU bottleneck.
  - **System-level reliability did not improve:** aggregated failure rate increased (607/3753 → 1123/5677), with failures dominated by **Transactions** (Internal Transfer 292/363 → 586/622; View History 292/439 → 537/733).
  - **Key takeaway:** reducing bcrypt rounds fixes the **authentication bottleneck**, but the end-to-end bottleneck shifts to **transaction/database/timeout constraints**; faster auth likely increases downstream pressure on Transactions.



Baseline Locust: `initial_200_10` **bcrypt=8 Locust:** `bcrypt_8_200_10`

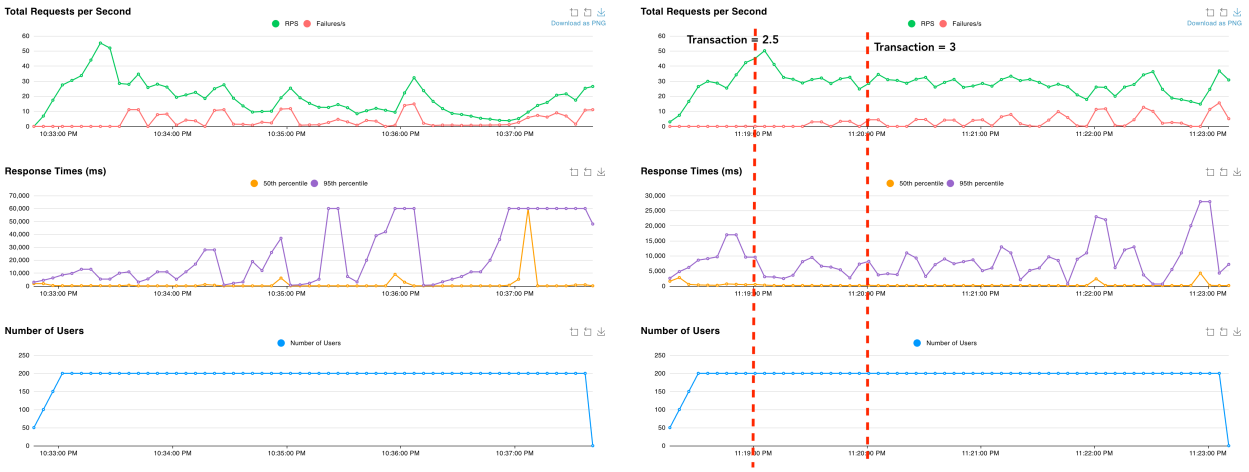
Figure 7: Experiment 5: Locust performance graphs comparison.

5.6.8 Experiment 5.1: Increase Transactions DB connection pool size

- **Test:** `200_10`
- **Change:** Transactions service DB pool size increased to **200**.
- **Results & inferences (vs `bcrypt_8_200_10`):**
  - **Throughput increased:** aggregated **RPS** 12.56 → 17.83.
  - **Overall reliability worsened:** failure rate 16.18% (607/3753) → 19.31% (1029/5330).
  - **Transactions regressed significantly:** Internal Transfer failures 292/363 → 521/547; View History failures 292/439 → 508/672.
  - **Auth is not the limiter in this run:** Login avg 38.84s → 6.51s; Register failures 11.5% → 0%.
  - **Key takeaway:** increasing the Transactions DB pool size alone did *not* resolve the bottleneck; transaction failures remained dominant (and higher), suggesting constraints such as DB contention/timeouts/transaction-service behavior. Higher end-to-end throughput may also amplify downstream pressure on Transactions. Consequently, we revert the pool size increment.

5.6.9 Experiment 5.2: Final HPA tuning for Transactions

- **Test:** `200_10`
- **Notes/Resets:** DB pool size reset to initial; bcrypt kept at 8
- **Changes:** Transactions HPA max replicas set to 3
- **Results & inferences (vs `bcrypt_8_200_10`):**
  - **Throughput increased strongly:** aggregated **RPS** 19.01 → 28.70.
  - **Reliability improved:** aggregated failure rate **19.78%** (1123/5677) → **10.84%** (929/8573).



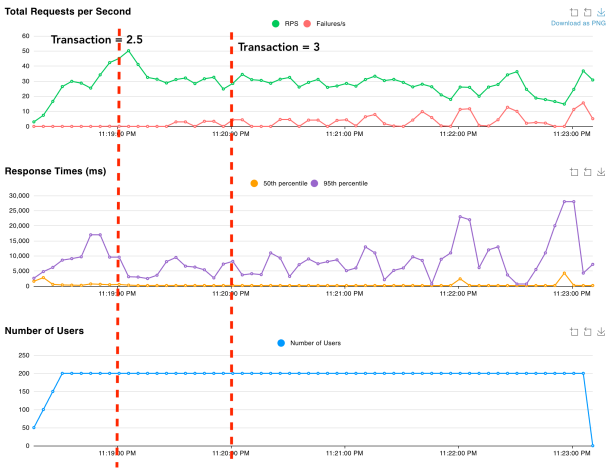
Baseline (bcrypt=8): `bcrypt_8_200_10` **Tuned Transactions HPA:** `maxReplicas=3`

Figure 8: Experiment 5.2: Locust graph comparison.

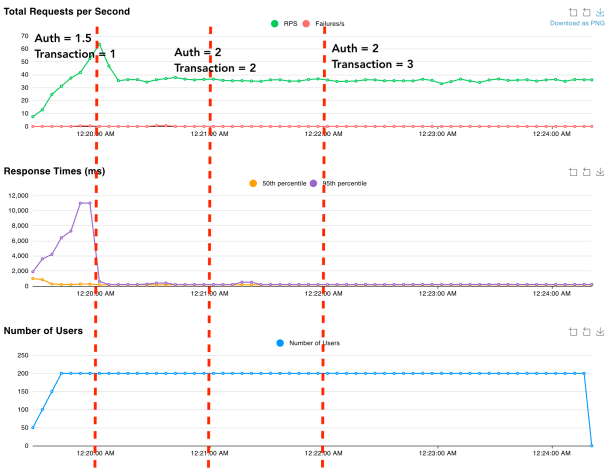
- **Transactions improved materially:** Internal Transfer failures **94.2%** (586/622) → **55.8%** (558/1000); View History failures **73.3%** (537/733) → **29.8%** (371/1243).
- **Latency improved:** aggregated avg **5.50s** → **1.87s**; tail improved (p95 **42s** → **10s**, p99 **60s** → **20s**).
- **Auth remained healthy (bcrypt=8):** login stayed fast (avg 5.16s → 7.72s) and register stayed at 0% failures.
- **Key takeaway:** increasing Transactions scaling headroom (`maxReplicas=3`) reduces the dominant bottleneck (transaction timeouts/failures) and improves end-to-end throughput/latency, but failures are still non-trivial, motivating further tuning (e.g., min replicas/CPU target).

5.6.10 Experiment 5.2.1: Adjust auth HPA max and lower HPA CPU target

- **Test:** `200_10`
- **Changes vs 5.2:** HPA CPU target 70% → 50% and customer-auth HPA max set to 2
- **Results & inferences (vs Experiment 5.2):**
  - **Throughput increased:** aggregated **RPS** 28.70 → 36.16.
  - **Failures nearly eliminated:** failure rate **10.84%** (929/8573) → **0.09%** (10/10791).
  - **Latency improved dramatically:** aggregated avg **1.87s** → **0.36s**; p95 **10s** → **0.45s**; median **200ms** → **170ms**.
  - **Transactions stabilized:** Internal Transfer failures **55.8%** (558/1000) → **0.22%** (3/1340); View History failures **29.8%** (371/1243) → **0.19%** (3/1595).
  - **Auth remained acceptable under max=2:** only **1** login failure; register had **0** failures (still multi-second latency, but no longer causing systemic errors).
  - **Key takeaway:** lowering the CPU target (more responsive scaling) and capping auth to 2 did *not* reduce performance; instead, the system became **stable at 200_10** with near-zero errors.



Experiment 5.2: Transactions max=3, CPU target 70%



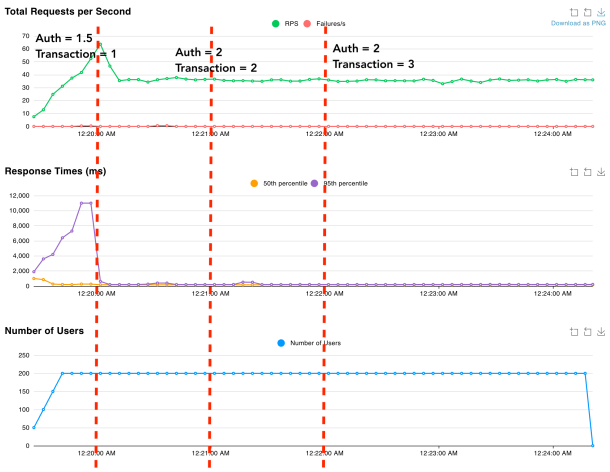
Experiment 5.2.1: Auth max=2, Transactions max=3, CPU target 50%

Figure 9: Experiment 5.2 vs 5.2.1: Locust graph comparison.

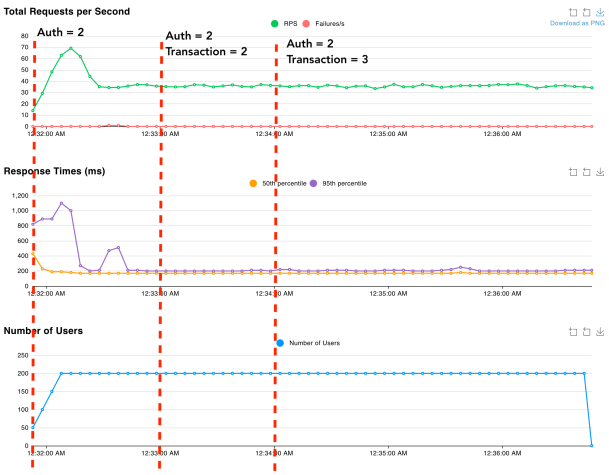
This indicates the previous failures were not primarily due to insufficient auth replicas, and that the dominant transaction instability can be mitigated under the tuned HPA behavior.

5.6.11 Experiment 5.2.2: Fix auth minimum replicas

- **Test:** `200_10`
- **Changes vs 5.2.1:** Customer-auth HPA min set to 2
- **Results & inferences (vs Experiment 5.2.1):**
  - **Throughput increased slightly:** aggregated **RPS** 36.16 → 37.31.
  - **Failures stayed near-zero:** 10/10791 → 9/11134 (both ≈**0.1%**).
  - **Latency improved substantially:** aggregated avg **362ms** → **194ms**; p95 **450ms** → **240ms**.
  - **Auth became consistently and significantly fast:** Login avg **3.68s** → **454ms**; Register avg **6.14s** → **830ms** (0 failures in both).
  - **Transactions remained stable:** Internal Transfer failures 3/1340 → 1/1399; View History failures 3/1595 → 8/1659 (still negligible).
  - **Key takeaway:** setting `minReplicas=2` for customer-auth removes cold-start/scale-up sensitivity under burst, improving end-to-end latency while keeping the near-zero error rate achieved in 5.2.1.



Experiment 5.2.1: Auth max=2, min=1; CPU target 50%



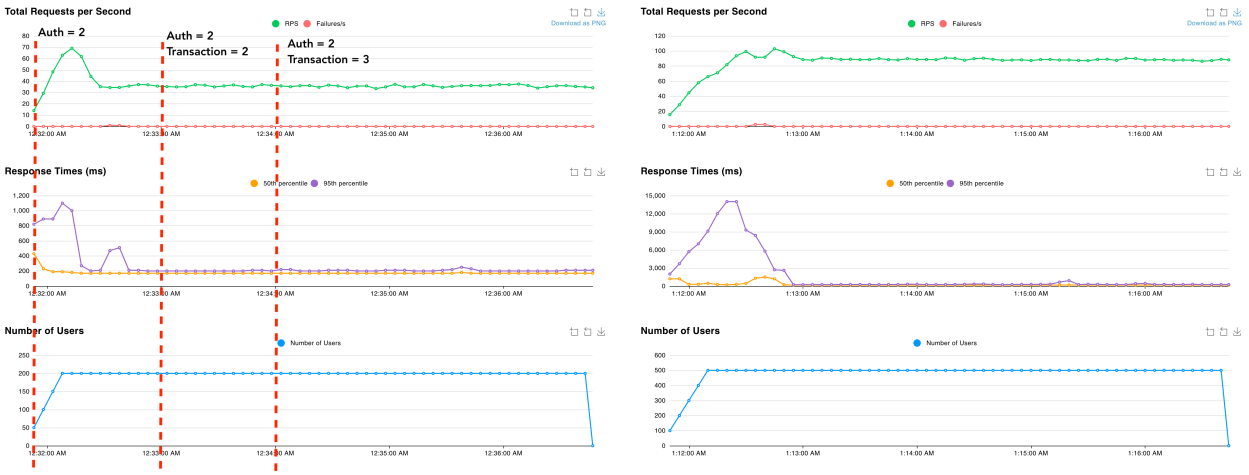
Experiment 5.2.2: Auth max=2, **min=2**; CPU target 50%

Figure 10: Experiment 5.2.2: impact of fixing customer-auth minimum replicas.

5.6.12 Experiment 6: Stress test the optimal configuration at higher load

Optimal Configuration:

- VM: e2-small (2 vCPU, 2GB RAM)
- DB pool size: Default (100)
- Hashing auth (bcrypt): 8 rounds (10 rounds before)
- Auth HPA min/max: 2/2 (no HPA before)
- Transaction HPA min/max: 1/3 (no HPA before)
- Node autoscaling: Off
- HPA CPU threshold: 50% (70% before)
- **Test:** `500_20`
- **Results & inferences (vs optimal `200_10` run):**
  - **Throughput scaled up:** aggregated **RPS 37.31** (`200_10`) → **86.47** (`500_20`).
  - **Reliability remained strong:** failure rate stayed around ∼**0.1%** (9/11134 → 26/25835). Transaction failures were low (Internal Transfer 12/3229, View History 14/3797).
  - **Tail latency increased under stress:** aggregated avg **194ms** → **573ms**; p95 **240ms** → **2200ms** and p99 **800ms** → **9800ms** (with max up to **25.4s**), indicating queueing/backpressure effects at peak load even though error rate stayed low.
  - **Auth stayed stable but remained comparatively slower:** Login median ∼6.7s and Register median ∼8.6s at `500_20` (0 failures), suggesting auth cost still contributes to end-to-end latency under high concurrency.
  - **Key takeaway:** the tuned configuration is **robust** at `500_20` (low failures and high throughput), but **long-tail latency** becomes the main degradation dimension at higher load.

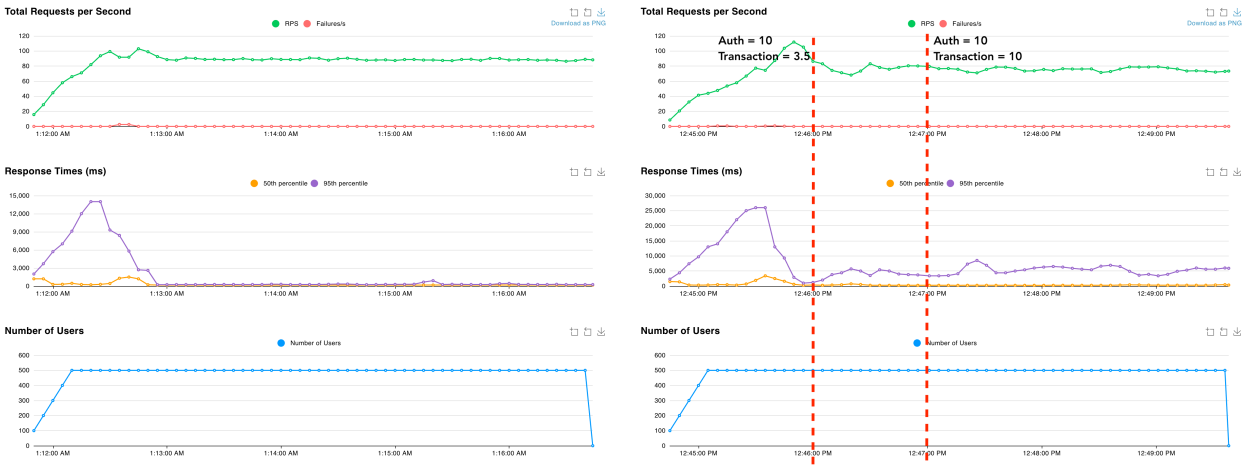


Optimal baseline: `200_10` **Stress test:** `500_20`

Figure 11: Optimal configuration under baseline vs stress load.

5.6.13 Experiment 6.1: Stress test with increased HPA ceiling

- **Test:** `500_20`
- **Changes vs 6:** HPA max replicas increased to 10
- **Results & inferences (vs Experiment 6):**
  - **Throughput decreased:** aggregated **RPS** 86.47 → 73.73.
  - **Failures remained low in both runs:** 26/25835 → 15/22011 (≈**0.1%** order).
  - **Latency worsened under max=10:** aggregated avg **573ms** → **1.57s**; p95 **2.2s** → **6.1s**; p99 **9.8s** → **20s** (max **25.4s** → **35.3s**).
  - **Node capacity became the limiter:** with **node autoscaling OFF**, raising HPA ceilings caused many pods to remain **Pending** (insufficient cluster resources). Extra replicas therefore could not translate into effective throughput and may increase contention/scheduling overhead.
  - **Observations:** Auth service quickly consumed most of the cluster's resources by aggressively scaling up to 7-10 pods, which lead to inadequate unused resources for the Transaction service to scale up. Consequently, transaction latency increased significantly.
  - **Key takeaway:** increasing HPA max replicas without adding node capacity (or enabling node autoscaling) can **degrade performance** at high load; scaling is limited by the cluster, not the HPA object.



Experiment 6: optimal config (`maxTrans=3`, `maxAuth=2`)

Experiment 6.1: increased ceilings (`maxAuth=10`, `maxTrans=10`)

Figure 12: Experiment 6 vs 6.1: Locust graph comparison under `500_20`.

```
aligokcek1@AliG-2 ~ % kubectl get pods -n martianbank
NAME                            READY   STATUS    RESTARTS   AGE
accounts-77678bf974-c5dxl       1/1     Running   0          10h
customer-auth-cb49449d8-8nx2n   1/1     Running   0          11h
customer-auth-cb49449d8-ckkfj   1/1     Running   0          10h
customer-auth-cb49449d8-cskm7   1/1     Running   0          3m31s
customer-auth-cb49449d8-dwslz   1/1     Running   0          3m41s
customer-auth-cb49449d8-krl47   0/1     Pending   0          3m26s
customer-auth-cb49449d8-mmw68   0/1     Pending   0          3m26s
customer-auth-cb49449d8-pvjvv   1/1     Running   0          3m31s
customer-auth-cb49449d8-wd8v2   0/1     Pending   0          3m26s
customer-auth-cb49449d8-xcbw5   1/1     Running   0          3m31s
customer-auth-cb49449d8-zrp88   1/1     Running   0          3m41s
dashboard-6bc94449c-64ld4       1/1     Running   0          10h
nginx-7cbc86f5bc-4bkzw          1/1     Running   0          10h
transactions-679567fb97-2bnlz   0/1     Pending   0          3m1s
transactions-679567fb97-4c8cz   0/1     Pending   0          2m31s
transactions-679567fb97-87kjb   0/1     Pending   0          2m31s
transactions-679567fb97-898tp   0/1     Pending   0          2m31s
transactions-679567fb97-9qvrj   0/1     Pending   0          3m1s
transactions-679567fb97-lpkbf   0/1     Pending   0          2m31s
transactions-679567fb97-pnlwk   0/1     Pending   0          2m46s
transactions-679567fb97-qp24r   1/1     Running   0          10h
transactions-679567fb97-t7pwj   0/1     Pending   0          3m16s
transactions-679567fb97-zbxmv   0/1     Pending   0          2m31s
ui-d4c4c7b4c-8tt7p              1/1     Running   0          10h
```

Figure 13: Experiment 6.1: evidence of node exhaustion (many pods in `Pending` state especially Transaction pods) when HPA ceilings were increased while node autoscaling was disabled.

6 Cost Breakdown

6.1 Monthly Cost Estimation

We have optimized resource selection to ensure the total monthly cost remains well below the $300 free trial budget.

Projected Monthly Cost: $\sim$ $183.55

Component	Resource Configuration	Estimated Monthly Cost	Justification
GKE Nodes	3x e2-medium instances (2 vCPU, 4GB RAM)	$139.20	Sufficient resources to host the microservices and system pods
GKE Management	Zonal Cluster	$0.00	Management fees waived for a single zonal cluster
Database VM	1x e2-small instance (2 vCPU, 2GB RAM)	$11.10	Adequate for MongoDB while minimizing compute costs
Load Balancing	Network Load Balancer (Layer 4)	$18.25	External traffic routing and ingress management
Storage (Disks)	40GB disks per node (3x40GB)	$4.00	Boot disks and storage allocation
Serverless	Cloud Functions (Gen 2)	$0.00	Expected to remain within free tier (first 2M invocations/month)
Container Registry	Image storage (∼1GB)	∼ $1.00	Docker image storage
Network Egress	Data transfer	∼ $5–$10	Variable based on traffic volume
Total		∼ $183.55	**(61% of total budget)**

6.2 Cost Optimization Strategies

Strategies Implemented:

1. **Right-sized GKE Nodes:** 3x e2-medium instances provide sufficient capacity
2. **Fixed Cluster Size:** No node auto-scaling (fixed 3 nodes) for predictable costs
3. **Optimized Database VM:** e2-small instance provides adequate MongoDB performance while minimizing compute costs
4. **Selective HPA:** Only transactions and customer-auth services have auto-scaling enabled
5. **Cloud Functions Free Tier:** Loan and ATM ssection dervices use Cloud Functions (expected within free tier)
6. **Efficient Storage:** 40GB disks per node using pd-balanced for cost efficiency
7. **Zonal Cluster:** Single zonal cluster eliminates management fees
8. **Resource Management:** Requests/limits prevent resource waste