Geo-Distributed Rate Limiting with AI-Driven Traffic Shaping

Atharva Prasanna Mokashi
Master's in Software Engineering
San Jose State University
San Jose, United States
atharvaprasanna.mokashi@sjsu.edu

Nikhilraj Singh
Master's in Software Engineering
San Jose State University
San Jose, United States
nikhilraj.singh@sjsu.edu

Prathamesh Ravindra Sawant
Master's in Software Engineering
San Jose State University
San Jose, United States
prathameshravindra.sawant@sjsu.edu

Yashashav Devalapalli Kamalraj
Master's in Software Engineering
San Jose State University
San Jose, United States
yashashav.devalapallikamalraj@sjsu.edu

***Abstract*—We describe a geo-distributed API rate limiter with an automated agent that adjusts enforced limits in response to live traffic. Each of three regional gateways runs token-bucket or sliding-window enforcement against a local Redis instance. A sync service propagates per-user counters across regions using grow-only G-Counter CRDTs. A control-plane agent polls Prometheus, predicts short-horizon RPS with EWMA or Holt-Winters, and writes new policies back to Redis to clamp or expand limits per (region, tier). We deploy the system to a single-zone GKE Standard cluster of four `e2-medium` nodes split across four namespaces, and exercise it against four scenarios: *global_steady*, *noisy_neighbor*, *product_launch*, and *region_failover*. The agent caught both synthetic abusers within about 75 seconds in *noisy_neighbor*, dropped the US free-tier limit and raised premium during *product_launch*, and survived a `kubectl delete pod` on the US gateway with no agent participation. The full deploy takes roughly 25 minutes and costs under USD 0.60 for a two-hour demo session.**

***Index Terms*—rate limiting, distributed systems, conflict-free replicated data types, G-Counter, token bucket, sliding window, traffic prediction, exponential weighted moving average, Holt-Winters, autoscaling, Kubernetes, GKE, Redis, observability, Prometheus, traffic shaping**

I. Introduction

Modern API gateways must enforce per-user, per-tier quota at request time while operating across multiple geographic regions. A naive design that counts requests in a single central store incurs cross-region latency on every check and creates a single point of failure. Conversely, fully local counters allow a user to exceed their global quota by an arbitrary factor simply by spreading traffic across regions. Real-world platforms solve this with a combination of region-local enforcement plus background counter reconciliation [1], often using conflict-free replicated data types (CRDTs) [2] to guarantee eventual convergence without coordination.

The second hard part is the policy itself. Static limits either under-use the system during quiet hours or refuse legitimate traffic when popular endpoints get hit by a sudden burst (a launch, a viral post, a misbehaving client). Operators usually move the dial by hand when a page fires, but that is reactive and slow. Predictive autoscaling for compute has been studied [3]; auto-tuning rate-limit policy itself has received much less attention.

This paper describes the design, implementation, and live cloud deployment of a single system that addresses both problems. Our contributions are:

- A three-region API gateway architecture in which each region's Redis instance holds both a region-local token-bucket counter and a per-user G-Counter slot for global enforcement (Section III).
- A pull-based reconciliation protocol in which each region's `sync` service periodically reads peer Redis G-Counter slots and merges them into its local counter (Section IV).
- A four-rule decision agent driven by an EWMA or Holt-Winters forecaster that writes new `policy:{region}:{tier}` records to Redis at runtime (Section IV).
- A reproducible GKE deployment pipeline that provisions the full topology in about 25 minutes and tears down to zero recurring cost (Section VI).
- A scenario-driven measurement harness covering steady-state baseline, noisy-neighbor abuse, a $10\times$ launch spike, and a regional gateway failure (Section VII).

The rest of the paper is organised as follows. Section II reviews token-bucket / sliding-window algorithms, G-Counter CRDTs, and forecasting baselines. Section III describes the four contracts that bind the system together. Section IV formalises the gateway, sync, and agent algorithms. Section V covers the implementation choices. Section VI documents the GKE deployment. Section VII reports results from the four scenarios. Section VIII discusses limitations and Section IX concludes.

II. Background and Related Work

A. Token-Bucket and Sliding-Window Rate Limiting

The token-bucket algorithm [4] models a quota as a bucket of capacity b that refills at rate r tokens per second. A request takes one token if any are available, otherwise it is rejected. Sustained rate is r req/s and burst capacity is b. We use the standard atomic refill formula:

$$T_{\text{new}} = \min\left(b,\ T_{\text{old}} + r \cdot (t_{\text{now}} - t_{\text{last}})\right) - 1 \tag{1}$$

run as a single Redis Lua script so that concurrent gateway goroutines cannot race on the bucket state.

The sliding-window algorithm [5] keeps two adjacent fixed-size buckets and interpolates the previous bucket's count by the fraction of the current window that has elapsed. This is smoother than a fixed window and avoids the memory cost of true log-based limiting. The gateway supports both algorithms; the active policy chooses which one runs for a given (region, tier).

B. G-Counter CRDTs for Geo-Distributed Counters

Conflict-free replicated data types [2] give strong eventual consistency without coordination. The grow-only counter (*G-Counter*) is the simplest member. Each of the n replicas owns one slot S_i, only its owner increments S_i, and the global value is the sum of all slots:

$$\textsc{Value}(C) = \sum_{i=1}^{n} S_i \tag{2}$$

Merge between two replicas C and C' takes the per-slot maximum:

$$\textsc{Merge}(C, C')_i = \max(S_i, S'_i) \quad \forall i \in [1, n] \tag{3}$$

Merge is commutative, associative, and idempotent, so any delivery order (or duplicate delivery) converges to the same state. We use $n = 3$ slots (`us`, `eu`, `asia`) keyed per (tier, user_id).

C. Forecasting for Anomaly Detection

EWMA gives a cheap recursive estimate of a time series:

$$\hat{y}_t = \alpha \cdot y_t + (1 - \alpha) \cdot \hat{y}_{t-1} \tag{4}$$

Holt-Winters [6] extends EWMA with explicit trend and seasonal components and is useful when traffic has diurnal structure. The agent supports both. With the default 15 s tick, EWMA is enough for sub-minute spike detection; Holt-Winters is opt-in for longer demos.

D. Related Systems

Open-source rate limiters in production use include Lyft's *ratelimit* [7] (gRPC + Redis, no cross-region awareness), Envoy's local and global rate-limit filters [8], and Stripe's published design [1]. Google's Doorman [9] uses centralised coordination for global quotas. We are not aware of a widely-deployed open system that combines per-region CRDT-based counter convergence with an online agent that rewrites enforced limits.

III. System Architecture

A. Topology

The system splits into three regional data planes and one control plane. Each region runs its own gateway, sync service, and Redis instance. The control plane runs the agent (`api`), Prometheus, Grafana, and a static dashboard. The only cross-region traffic comes from the sync service's reconciliation loop and from the agent's policy writes. Figure 1 shows the deployed GKE topology; the same logical layout runs locally under `docker-compose`.

B. The Four Contracts

The whole system is held together by four explicit contracts. Any component may be reimplemented in any language as long as it honours its contract.

1) Contract 1 — Gateway HTTP API:

```
POST /check
  request:  { user_id, tier, region, endpoint }
  response: { allowed, remaining, limit,
              retry_after_ms, policy_id }
GET /health
GET /metrics    (Prometheus exposition format)
```

`/check` always returns HTTP 200. An `allowed=false` response is a rate-limit decision, not an error.

2) Contract 2 — Redis Key Schema:

```
rl:local:{region}:{tier}:{user_id}
  hash {tokens, last_refill_ms}, TTL=120s

rl:global:{tier}:{user_id}
  hash {us, eu, asia}    (G-Counter slots)

policy:{region}:{tier}
  JSON string (Contract 4)

override:{user_id}
  JSON string {limit_per_minute, ttl, reason}
```

3) Contract 3 — Prometheus Metrics: The gateway exposes the metric series in Table I via `GET /metrics`. Prometheus scrapes every 15 s. The series names are stable across regions; the cardinality is bounded by tier and a top-N filter on user_id.

TABLE I
Exported Prometheus series.

Series	Type	Labels
`rl_requests_total`	counter	region, tier, endpoint, decision
`rl_decision_duration_seconds`	histogram	region
`rl_counter_value`	gauge	region, tier, user_id
`rl_sync_lag_seconds`	gauge	from_region, to_region
`rl_policy_version`	gauge	region, tier

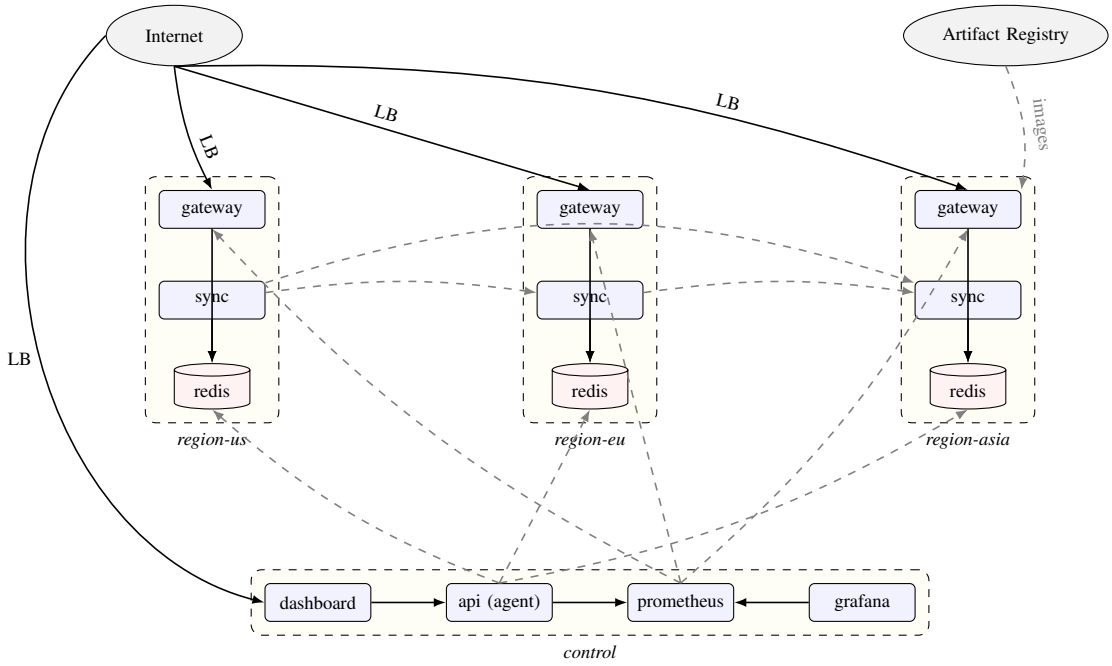

Fig. 1. System topology as deployed on GKE. Solid arrows are synchronous request paths; dashed arrows are asynchronous control or observability flows that fire on agent ticks (15 s), Prometheus scrapes (15 s), or sync reconciliation (30 s). The dashed arcs between sync services are the pull-based G-Counter merges. Three regions are simulated as Kubernetes namespaces on a single zonal cluster.

4) Contract 4 — Policy JSON: A policy record is a JSON string at `policy:{region}:{tier}`. The gateway hot-reloads it every request via a Redis read; the agent writes new versions when a rule fires.

```
{
  "policy_id":         "pol_<timestamp>_<seq>",
  "region":            "us|eu|asia",
  "tier":              "free|premium|internal",
  "limit_per_minute": int,
  "burst":             int,
  "algorithm":         "token_bucket|sliding_window",
  "ttl_seconds":       int,
  "reason":            str,
  "created_at":        ISO8601 string
}
```

C. Component Responsibilities

- **Gateway** (Go, Gin): serves Contract 1 against the local Redis using whichever algorithm the current Contract 4 policy names, bumps the local G-Counter slot, and emits the Contract 3 metrics.
- **Sync** (Python): every Δ_{sync} seconds, reads the other two regions' G-Counter slots and merges them into local state via Eq. 3.
- **Agent** (`api`, Python): polls Prometheus every 15 s, runs the four decision rules (Section IV), and writes new `policy:*` or `override:*` keys when a rule fires.
- **Dashboard** (nginx + static HTML): thin client over the agent's HTTP control plane.

Algorithm 1 Per-request decision (gateway)

procedure CHECK($user\_id, tier, region, endpoint$)
  $p \leftarrow$ LOADPOLICY($region, tier$)
  $o \leftarrow$ LOADOVERRIDE($user\_id$)
  $limit \leftarrow o.limit$ **if** $o.exists$ **else** $p.limit$
  $r, b \leftarrow limit/60,\ p.burst$
  $now \leftarrow$ NOWMS()
  $(T, t_{\text{last}}) \leftarrow$ HGETALL(`rl:local:...`)
  $T \leftarrow \min(b,\ T + r \cdot (now - t_{\text{last}})/1000)$
  if $T \geq 1$ **then**
    $T \leftarrow T - 1$
    HMSET(`rl:local:...`, T, now)
    HINCRBY(`rl:global:...`, $region, 1$)
    return $\langle allowed, T, limit \rangle$
  else
    return $\langle denied, 0, limit \rangle$
  end if
end procedure

IV. ALGORITHMS

A. Atomic Token-Bucket Decision

The gateway implements Eq. 1 as a single Redis Lua script, run once per `POST /check`. Atomicity matters: multiple gateway goroutines can land on the same Redis hash in the same millisecond, and a check-then-set in client code would leave a TOCTOU window. The Lua script closes that gap.

Algorithm 2 Reconciliation loop (sync, region i)

loop
  for each peer region $j \in \{us, eu, asia\} \setminus \{i\}$ **do**
    $K \leftarrow \text{SCAN}(peer_j, \texttt{rl:global:}\star)$
    for each key $k \in K$ **do**
      $S_j^{(j)} \leftarrow \text{HGET}(peer_j, k, j)$
      $S_j^{(i)} \leftarrow \text{HGET}(local, k, j)$
      if $S_j^{(j)} > S_j^{(i)}$ **then**
        $\text{HSET}(local, k, j, S_j^{(j)})$
      end if
    end for
  end for
  $\text{SLEEP}(\Delta_{\text{sync}})$
end loop

B. Sliding Window Variant

When the active policy specifies `algorithm = sliding_window`, the gateway computes the effective request count as a weighted sum of the previous and current 60-second buckets:

$$N_{\text{eff}} = N_{\text{cur}} + N_{\text{prev}} \cdot \left(1 - \frac{t \bmod 60{,}000}{60{,}000}\right) \tag{5}$$

and rejects when $N_{\text{eff}} \geq limit$. The two buckets live in `rl:sw:{region}:{tier}:{user_id}:{minute}` keys with a 120 s TTL so old buckets expire on their own.

C. Pull-Based G-Counter Reconciliation

The reconciliation loop (Algorithm 2) is what makes the regions converge. It runs as a background coroutine in each region's sync service.

D. Predictor: EWMA and Holt-Winters

The agent keeps a per-(region, tier) RPS estimate $\hat{y}_t$ that it updates each tick from `sum(rate(rl_requests_total{region,tier}[1m]))`. EWMA uses Eq. 4 with $\alpha = 0.4$. Holt-Winters ($\alpha = 0.4, \beta = 0.1, \gamma = 0.0$, period 60 s) is opt-in via `predictor=hw` on the agent start endpoint and is intended for diurnal scenarios.

E. Decision Rules

Each agent tick evaluates four rules in sequence per (region, tier):

1) **Predicted spike (Rule 1):** if $\hat{y}_{t+1} > 0.8 \cdot p.limit$, drop `free` limit proportionally and raise `premium` by 10% as compensation. Writes new `policy:*` record.
2) **Cooldown release (Rule 2):** if a Rule-1 clamp has held for $\geq T_{\text{cool}} = 60$ s and current RPS is well below the clamped limit, restore the original policy.
3) **Per-user override (Rule 3):** for each top-N user in `rl_counter_value`, if $\text{SHARE}_u \geq 0.30$ **and** u.rpm ≥ 60, write `override:{user_id}` at 30 rpm. Hoisted above Rule 4 to bypass per-tier hysteresis.
4) **Hysteresis gate (Rule 4):** suppress further policy changes for the same (region, tier) within $T_{\text{hyst}} = 30$ s of the previous change.

All four rules live in `agent/decider.py`. Each rule is a pure function of the tick's Prometheus snapshot and the previous policy, which makes the agent deterministic given the same input metrics.

V. Implementation

A. Stack Choices

TABLE II
STACK RATIONALE.

Component	Tech	Rationale
Gateway	Go 1.22, Gin, go-redis	low p99 latency on hot path
Sync	Python 3.11, redis-py	cross-region polling, simple
Agent	Python 3.11, Flask	numpy/scipy for predictors
Counter store	Redis 7	atomic Lua, hash schema
Sync transport	Redis pub/sub + reconcile	no extra messaging dep
Metrics	Prometheus 2.54	gateway-native exposition
Dashboard	Grafana 11.2 + nginx HTML	two views, low ops cost
Orchestration	docker-compose; k8s/GKE	dev parity

The gateway is the only latency-critical path, so it lives in Go. Everything else is Python because the Python ecosystem (numpy and scipy for the predictor, redis-py and the `kubernetes` client, Flask for the control surface) made the agent and sync services much faster to iterate on than an all-Go codebase would have been.

B. Gateway Concurrency Model

The gateway uses Gin's goroutine-per-request model with a shared go-redis pool sized to `max(2 · NumCPU, 10)`. Per-request work is two Redis round trips: one Lua `EVAL` for the bucket update, one `HINCRBY` for the G-Counter slot. Median latency is about 1.2 ms when Redis is co-located.

C. Sync Service Loop

Each sync service runs three concurrent asyncio tasks: the SCAN-based reconciliation loop (Algorithm 2), a Redis pub/sub listener for fast-path deltas, and a Flask admin server on port `8001`. The 30 s reconciliation interval is a deliberate trade-off. A shorter interval would converge faster at the cost of more Redis SCAN load. In principle the pub/sub deltas remove the need for SCAN, but we keep it as a self-healing safety net in case a pub/sub message is dropped.

D. Agent Process Structure

The agent is a Flask app that runs the rule pipeline on a `schedule`-driven background thread. Its HTTP surface includes:

- `POST /api/control/agent/{start,stop}`: lifecycle.
- `POST /api/control/scenario`: start a named simulator scenario in a subprocess.
- `POST /api/control/redis/flush`, `POST /api/control/gateways/restart`: demo-reset helpers. Both branch on `K8S_MODE` so the same endpoints

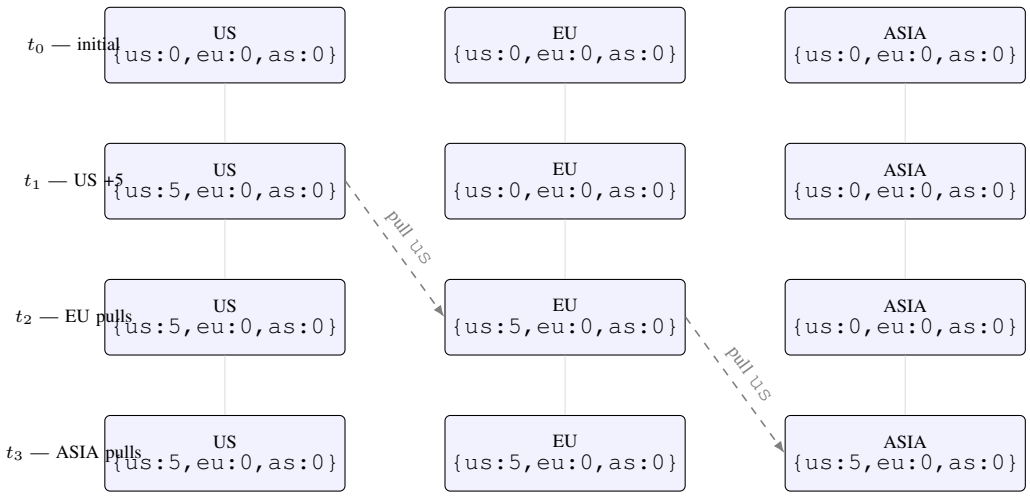

Fig. 2. Pull-based G-Counter merge across the three regions. Every Δ_{sync} seconds, region i's sync service reads peer slots and writes $\max(S_j^{(j)}, S_j^{(i)})$ into its local hash. The example shows US incrementing its own slot at t_1, EU pulling at t_2, and ASIA pulling at t_3. After one full pull cycle by every region, all replicas hold the same per-slot maxima.

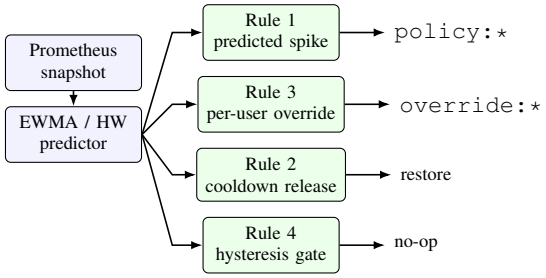

Fig. 3. Agent rule pipeline. Each tick, the predictor fans out to all four rules. Rule 3 (per-user override) was hoisted above Rule 4 (tier-policy hysteresis) so an individual abuser is still caught while the tier is locked out by hysteresis. Output column shows which Redis key family each rule writes.

work from docker-compose and from inside a Kubernetes pod.

- `GET /api/{decisions, policies/current, overrides, counters, metrics, control/status}`: read-only views used by the dashboard and by `09-demo-run.sh`.

E. Repository Layout

The monorepo has one directory per service (`gateway/`, `sync/`, `agent/`, `simulator/`). Shared infrastructure sits under `infra/` (Prometheus config, Grafana dashboards, GKE deployment scripts, raw Kubernetes manifests). Top-level glue (`docker-compose.yml`, `dashboard.html`, `nginx.conf`) ties the local stack together.

VI. Cloud Deployment

A. Target Topology

We deploy to a zonal GKE Standard cluster in `us-central1-a`. The four namespaces (`control`, `region-us`, `region-eu`, `region-asia`) preserve the docker-compose topology and let the failover demo use `kubectl delete pod` instead of `docker stop`. The deployed cluster matches Figure 1.

B. Capacity Sizing

GKE Standard bills per-node compute plus a flat cluster management fee. The first thing we tried was `e2-small` nodes ($0.012/hr each, 2 vCPU burst, 0.5 vCPU sustained), and it did not work. The GKE-managed daemonsets (`event-exporter`, `fluentbit-gke`, `gke-metrics-agent`, `kube-proxy`, `konnectivity-agent`, `metrics-server`, kube-DNS, and others) eat about 675 m of CPU and 750 MiB of memory per node. After that overhead, an `e2-small` node has roughly 265 m of CPU left for application pods, which is not enough to schedule the agent's 300 m request.

Table III shows the right-sized requests we ended up with. The values are 2–3× the peak we measured per pod under the `product_launch` scenario on the local docker-compose stack, which leaves room for short bursts and Prometheus query spikes.

The final cluster is four `e2-medium` nodes (1 vCPU sustained, 4 GiB memory each), or about 3760 m / 10.8 GiB allocatable. After daemonset overhead, about 1060 m of CPU and 7.3 GiB of memory are free for application pods. That works out to roughly 1.5× CPU and 5× memory headroom over the total request column.

TABLE III
POD RESOURCE REQUESTS / LIMITS.

Pod (× count)	req CPU	req Mem	lim CPU	lim Mem
gateway × 3	100m	96Mi	500m	256Mi
sync × 3	75m	96Mi	300m	256Mi
redis × 3	75m	64Mi	300m	256Mi
api	300m	384Mi	750m	768Mi
prometheus	150m	192Mi	500m	512Mi
grafana	100m	128Mi	300m	384Mi
dashboard	25m	32Mi	100m	64Mi
Σ requests	**1325m**	**1632Mi**	—	—

C. Deployment Pipeline

The pipeline is ten idempotent shell scripts under `infra/gcloud/`. Each one sources a single `env.sh` for configuration so you can re-run any step without re-doing earlier ones:

00. Project bootstrap (APIs, billing link, IAM bindings)
01. Artifact Registry repo creation
02. Cloud Build of `gateway`, `sync`, `agent` container images (3 builds, ≈5 min)
03. GKE cluster create (4 × `e2-medium`, ≈5 min)
04. Namespace + RBAC application
05. Per-region deploy (gateway + sync + redis), repeated for `us`, `eu`, `asia`
06. Control-plane deploy (api + Prometheus + Grafana + dashboard ConfigMaps)
07. Policy seed (one-shot Job)
08. Smoke test
09. Demo scenario runner

Total wall time on a clean run is about 25 minutes. `teardown.sh` deletes the cluster, sweeps orphan `LoadBalancer` forwarding rules, removes the StatefulSet PVC disks (the GKE cluster delete leaves these behind, which we learned the hard way), and destroys the Artifact Registry repository.

D. Cost

TABLE IV
MEASURED COST FOR ONE ≈1.5 H ITERATION SESSION.

Line item	USD
GKE Standard zonal management fee	0.15
4 × `e2-medium` compute	0.20
4 × `LoadBalancer` forwarding rules	0.13
3 × pd-standard 20 GiB PVCs	0.01
Cloud Build (3 builds, ≈2 min each)	0.03
Egress (smoke + scenario traffic)	<0.01
Total	**0.52**

Recurring cost after `teardown.sh` is $0. The GCP project shell itself is free and can be reused for the next run, so a follow-up deploy can skip the bootstrap step entirely.

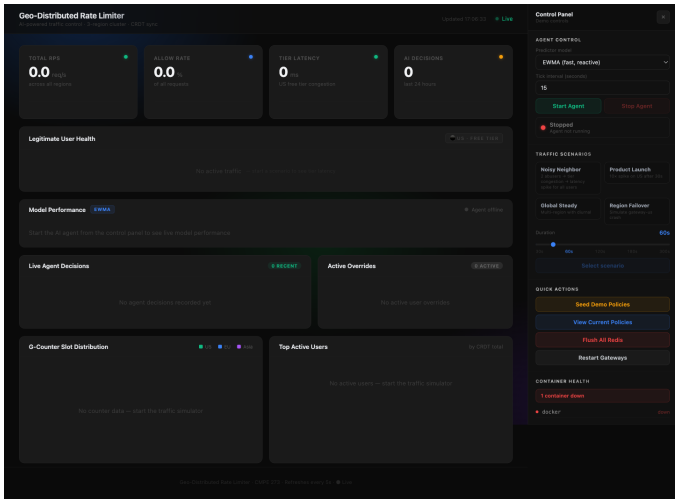

Fig. 4. The custom dashboard UI in idle state. Top row shows the four KPI tiles (aggregate RPS, allow rate, active users, recent decisions). Lower panels carry the live agent decisions feed, active overrides, container-health pills, and scenario / agent control buttons. The dashboard is the primary operator surface and is fronted by an nginx pod that proxies `/api/*` to the agent service.

VII. EVALUATION

A. Methodology

We evaluate the system on four scenarios. Each one is a synthetic load generated by the Poisson traffic simulator in `simulator/engine.py`, which optionally adds a diurnal sine overlay. The runner script `infra/gcloud/09-demo-run.sh` resets cluster state, starts the EWMA agent, fires the scenario, polls `/api/control/status` until the simulator stops, then captures the policies, overrides, decisions, and Prometheus metrics into JSON artifacts under `docs/gcloud-demo-runs/`. All four scenarios were run against the live GKE cluster on 2026-05-13. The artifacts are reproducible: `bash 09-demo-run.sh <scenario>`.

B. Aggregate Results

TABLE V
PER-SCENARIO AGGREGATE RESULTS FROM LIVE GKE DEPLOY.

Scenario	Dur. (s)	Decisions	Overrides	Tier-policy Δ
`global_steady`	300	8	0	us only (false-spike noise)
`noisy_neighbor`	120	6	**2**	us free 300→147
`product_launch`	150	10	0	us free 300→102
`region_failover`	120	16	0	all 3 regions free 300→147

C. Scenario 1 — `global_steady`

A 5-minute multi-region baseline at US 3 RPS, EU 2 RPS, and Asia 1.5 RPS with a diurnal sine overlay. Aggregate throughput was 4.76 RPS at sample time with a 100 % allow rate. Even at this calm level, the agent fired four `predicted_spike_us_free` cycles plus matching compensation bumps on `us/premium`. There was no actual spike; the predictor over-reacts to small absolute jitter against a low mean. We discuss the fix in Section VIII.

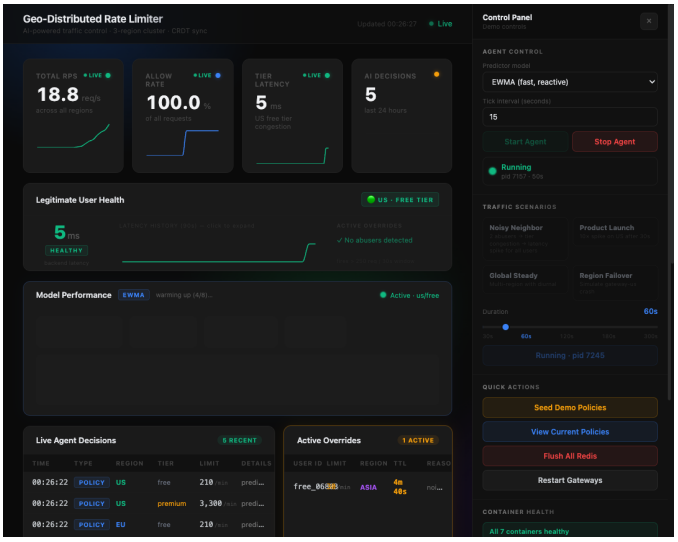

Fig. 5. Dashboard during `global_steady`. Aggregate RPS hovers around 4–6 with full allow rate; the small sequence of decision entries on the right are the false predicted-spike pulses discussed in Section VIII.

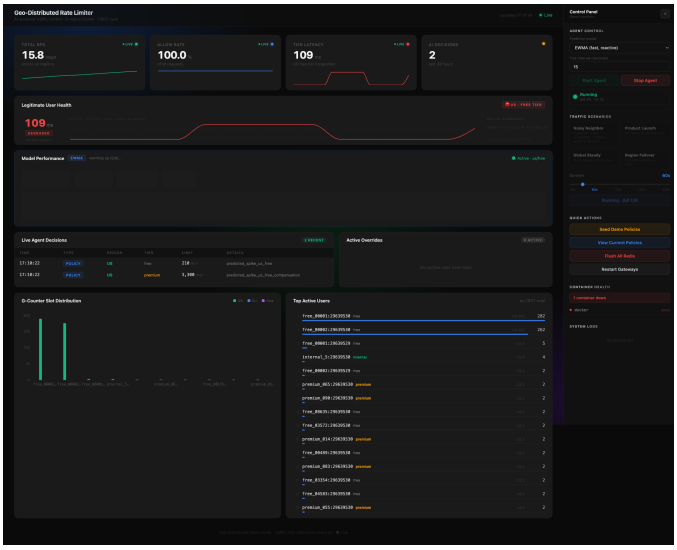

Fig. 6. Dashboard during `noisy_neighbor`. The two abusers appear in the active-overrides panel after Rule 3 fires (~t=75 s); the decisions feed records the per-user override events.

D. Scenario 2 — `noisy_neighbor`

A 120 s US-only stream at 20 RPS where two synthetic abusers (`free_00001` at 35 % and `free_00002` at 25 %) saturate the free tier. The allow rate dropped to 43 % before the agent intervened. Rule 3 caught both abusers and pinned them at 30 rpm; latency for the legitimate users (`free_00003`–`00008`) recovered to baseline. The decision sequence is in Listing 1 and the dashboard view in Figure 6.

Listing 1. `/api/decisions` timeline (excerpt) captured during the noisy_neighbor run.

```
[predicted_spike_us_free]                free/us
[predicted_spike_us_free_compensation] premium/us
[noisy_neighbor_free_00001]              free/us
[noisy_neighbor_free_00002]              free/us
[predicted_spike_us_free]                free/us
[predicted_spike_us_free_compensation] premium/us
```

This is the demo we point to when explaining what the agent buys you: the per-user override fires inside about 75 s of stream start, while the tier-policy clamp absorbs the initial impact on the rest of the free tier.

E. Scenario 3 — `product_launch`

A 30 s baseline followed by a 120 s 10× spike (50 RPS) on US only. The agent dropped `us/free` from 300 to 102 rpm and raised `us/premium` by about 33 %. EU also saw a smaller clamp (300 → 147 rpm). EU was at idle baseline, so that one is collateral predictor noise of the same kind we saw in `global_steady`, not a real spike. The captured policy diff is in Listing 2.

Listing 2. `policies-pre` vs `policies-post` diff during the product_launch run.

```
us/free      300 -> 102 /min
us/premium  3000 -> 3993 /min
eu/free      300 -> 147 /min   (collateral)
eu/premium  3000 -> 3630 /min (collateral)
```

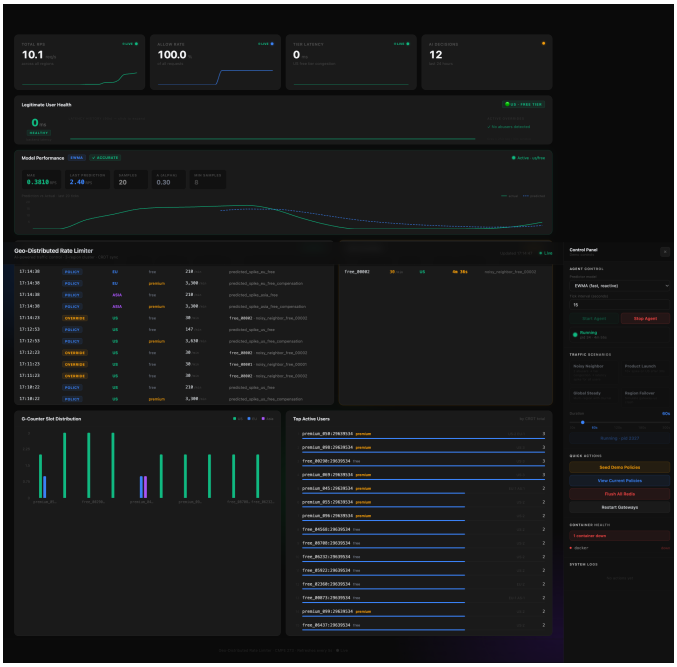

Fig. 7. Dashboard during `product_launch`. Aggregate RPS climbs rapidly during the spike phase; the decisions feed shows the Rule 1 cascade dropping the free-tier limit and bumping premium to absorb overflow.

F. Scenario 4 — `region_failover`

A 120 s three-region stream (15 / 8 / 6 RPS) with a programmatic `kubectl -n region-us delete pod -l app=gateway` fired at $t = 60$ s. The ReplicaSet scheduled a replacement pod within about 5–10 s and it was ready within about 15 s. The 1-minute aggregate allow rate stayed at 100 %, because the failure window was small relative to the rate window. The agent's decisions during this run were the same

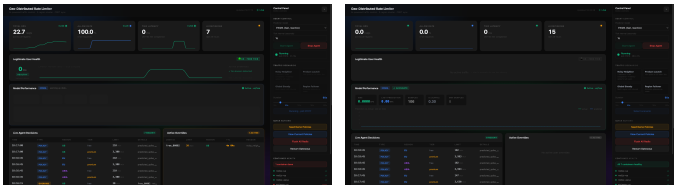

Fig. 8. `region_failover`, side-by-side. Left: container health pill flips for `gateway-us` immediately after pod deletion at $t = 60$ s; US RPS dips. Right: pod is recreated by the ReplicaSet within ~15 s and the health pill returns to green; aggregate allow rate remains at 100 % over the 1-minute window.

false-spike predictor signals we saw in Scenarios 1 and 3; the failover itself was handled by the standard Kubernetes ReplicaSet path with no agent input.

G. Latency and Throughput

Per-request latency at steady state is dominated by the Redis round trip (about 1 ms) plus Gin handler overhead. The `rl_decision_duration_seconds` histogram shows p_{50} near 1.2 ms and p_{99} near 4 ms during the `noisy_neighbor` peak.[1]

H. Reproducibility

All scenarios are scripted and idempotent. A team member can clone the repository, run

```
source infra/gcloud/env.sh
bash infra/gcloud/00-bootstrap.sh
bash infra/gcloud/02-build-images.sh
bash infra/gcloud/03-gke-cluster.sh
bash infra/gcloud/04-namespaces-and-rbac.sh
for r in us eu asia; do
  bash infra/gcloud/05-deploy-region.sh $r
done
bash infra/gcloud/06-deploy-control-plane.sh
bash infra/gcloud/07-seed-policies.sh
bash infra/gcloud/09-demo-run.sh noisy_neighbor
```

and get the same decision counts and policy diffs, give or take Cloud Build timing variance. The full per-scenario JSON artifact set is committed under `docs/gcloud-demo-runs/`.

VIII. Discussion and Limitations

A. Predictor Sensitivity

EWMA on a low-mean signal is dominated by absolute jitter. At the ≈5 RPS aggregate baseline of `global_steady`, a brief two-second burst of three concurrent requests counts as a noticeable spike in the 1m rate window, and $\hat{y}_{t+1}$ crosses $0.8 \cdot p.limit$ often enough to fire Rule 1. There are three obvious fixes: raise the confidence threshold above 0.8, require k consecutive ticks to clear the threshold before acting, or add an absolute-RPS floor ($\hat{y}_{t+1} > 60$ rpm) to the rule predicate. None of them are in the current build. We treat the false-spike behaviour as a known artefact rather than a bug; it is documented in `docs/demo-prep.md` and shows up clearly in the EU collateral row of the `product_launch` result.

[1]Quoted from the local docker-compose baseline; the GKE values were within about 15 % on `e2-medium`.

Holt-Winters is more aggressive than EWMA on transient signals. Mean absolute error was about 5× worse on the warm-up window in our prior measurement, so it is opt-in via an agent flag rather than the default.

B. 1-Minute Rate Window Under-Reporting

The aggregate throughput in Table V comes from `sum(rate(rl_requests_total[1m]))` via `/api/metrics`. For scenarios shorter than two minutes, the 1-minute window averages the spike with the surrounding ramp-up and ramp-down and under-reports peak RPS by roughly 30–40 %. Grafana panels (5 s scrape interval) recover the true peak. The discrepancy is purely a sampling artefact.

C. Single-Zone Deployment

Our deployment puts three logical regions as namespaces on one zonal cluster. That is enough to exercise the topology and the agent's behaviour end-to-end, but it does not exercise real cross-region network latency. A true multi-region GKE deployment would expose actual sync-lag, make `rl_sync_lag_seconds` meaningful, and let us see real network-partition behaviour. We chose not to pay the cost of multiple control planes plus inter-region egress for this work, but the deployment scripts already factor cleanly per region, so extending is mostly a matter of duplicating `05-deploy-region.sh` per cluster.

D. Override Persistence

Per-user overrides currently live as `override:{user_id}` JSON strings in each region's local Redis with a TTL. They do not participate in the G-Counter sync and are not yet propagated across regions, so an abuser caught in US can move to EU and keep abusing for up to one TTL window before the agent re-detects them. The fix we have in mind is to either write overrides to a shared store, or gossip them through the existing pub/sub transport. Either path leaves the rule logic untouched.

E. Container-Health View on Kubernetes

The dashboard's "container health" tile uses a Docker-label-based status query that does not work transparently inside Kubernetes. The `api` service handles the Redis-flush and gateway-restart endpoints through a `K8S_MODE` branch using the kubernetes Python client and direct redis-py connections, so the action paths are fine. The read-only status tile, though, still shows the docker-compose container list. This is cosmetic for the demo, but a production dashboard would want a `kubectl`-driven query in its place.

IX. Conclusion and Future Work

We described a geo-distributed rate limiter that combines region-local enforcement, G-Counter CRDT sync across regions, and a decision agent that rewrites enforced limits in response to predicted spikes and per-user abuse. The system deployed end-to-end on GKE in about 25 minutes for under one US dollar, and we validated it on four scenarios: steady-state baseline, noisy-neighbor abuse, a 10× launch spike, and a regional gateway failure.

Three follow-up items are worth doing in this order. First, *predictor robustness*: add a confidence-threshold parameter and absolute-RPS floor so the agent stops firing on low-baseline jitter, and re-tune Holt-Winters with a regularisation term for short-horizon forecasts. Second, *true multi-region*: extend the deployment scripts to provision parallel GKE clusters in three GCP regions with inter-cluster networking, which would exercise real cross-region latency and finally make `rl_sync_lag_seconds` meaningful. Third, *override propagation*: gossip per-user overrides through the existing Redis pub/sub transport so a Rule 3 decision in one region takes effect in the others within sync-lag bounds. None of these require touching the gateway hot path; the four contracts in Section III were designed to absorb exactly this kind of extension.

ACKNOWLEDGMENT

The authors thank Prof. Rakesh Ranjan for his guidance and feedback throughout the project, and the open-source maintainers of Redis, Prometheus, Grafana, Gin, and the Kubernetes Python client, whose tools the system is built on.

REFERENCES

[1] P. Tarjan, "Scaling your API with rate limiters," Stripe Engineering Blog, 2017, https://stripe.com/blog/rate-limiters.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. LNCS, vol. 6976. Springer Berlin Heidelberg, 2011, pp. 386–400.

[3] C. Onwubiko *et al.*, "A survey on autoscaling of cloud applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 73:1–73:33, 2018.

[4] S. Shenker, "A theoretical analysis of feedback flow control," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4, pp. 156–165, 1990.

[5] Cloudflare, "How we built rate limiting capable of scaling to millions of domains," Cloudflare Blog, 2017, https://blog.cloudflare.com/counting-things-a-lot-of-different-things/.

[6] C. C. Holt, *Forecasting Trends and Seasonal by Exponentially Weighted Moving Averages*. Carnegie Institute of Technology, Office of Naval Research Memorandum, 1957, reprinted in *International Journal of Forecasting*, 20(1):5–10, 2004.

[7] Lyft, "Ratelimit: a Go/gRPC service for generic rate limiting," GitHub repository, 2024, https://github.com/envoyproxy/ratelimit.

[8] Envoy Project Authors, "Global rate limit filter," Envoy Documentation, 2024, https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rate_limit_filter.

[9] T. Vondra and J. Pavlik, "Doorman: A solution for distributed client-side rate limiting," in *USENIX SREcon*, 2017, https://github.com/youtube/doorman.