Load Balancer Filter-Based Approach To Enable Distributed API Rate Limiting

Thivaharan Kalyanasundaram, Kobinarth Panchalingam, Tharsigan Jegatheesan, Adeesha Wijayasiri
University of Moratuwa
Moratuwa, Sri Lanka
{thivaharan.20, kobinarth.20, tharsigan.20, adeeshaw}@cse.mrt.ac.lk

Srinath Perera
WSO2 LLC
Santa Clara, USA
srinath@wso2.com

***Abstract*—Efficient and accurate rate limiting is crucial for managing API traffic in distributed systems, ensuring fair resource allocation, preventing abuse, maintaining reliability, and enabling monetization. Unlike single-node approaches, distributed rate limiting poses challenges in maintaining consistent API rate limits across multiple nodes. This research explores implementing distributed API rate limiting through load balancers, evaluating six algorithms integrated via Lua filters that enforce client-specific global rate limits while balancing performance and accuracy. To synchronize client states across load balancers, three mechanisms are assessed: Redis, MySQL, and Conflict-Free Replicated Data Types (CRDTs). Latency, throughput, and throttling deviation serve as key metrics to evaluate the results. To reduce the potential state synchronization overhead among load balancers, this research introduces a novel asynchronous batch-quota-based implementation. Experiments with a microservices benchmarking application and API traffic simulation in Google Cloud Platform (GCP) demonstrate that the load balancer-based rate limiting framework introduces minimal performance impact, with latency overhead remaining below 1% for certain configurations, proving its high viability. The Sliding Window Log algorithm had the lowest throttling deviation, while all algorithms showed negligible performance differences. For state synchronization, CRDTs exhibited the lowest latency overhead, followed by Redis and MySQL, with all three offering comparable consistency. These findings provide practical insights for API providers when selecting rate-limiting strategies for distributed environments.**

***Index Terms*—API Rate Limiting, Algorithms, Load Balancers, State Synchronization, Distributed Systems**

I. Introduction

The evolution of Application Programming Interfaces (APIs) has transformed software development and system integration. Beyond their technical role, APIs drive innovation, enhance user experiences, and fuel business growth across industries [1]. Industry projections indicate continued API adoption, with Gartner predicting a 30% rise in API demand by 2026, driven by AI and large language models (LLMs) [2]. This rise of *API economy* reflects how organizations will continue to leverage APIs to monetize their data, services, and functionalities, fostering collaboration and innovation [3]. This surge in API adoption introduces significant challenges. As the number of APIs and consumers grows, managing performance, security, and fair usage becomes increasingly complex.

A critical component of API management is *rate limiting*, which restricts the number of requests an *API client* can make to an API endpoint within a defined time frame. The term is often used interchangeably with *throttling*, though throttling may also refer to dynamically slowing down requests based on system load. It is essential for preventing server overloads, mitigating denial-of-service (DoS) attacks, reducing service disruptions, and preventing financial losses for providers. Additionally, it ensures fair resource allocation among clients, preventing monopolization by a single user. It also enables API providers to monetize their services through tiered access levels and subscription plans.

Distributed API rate limiting refers to the implementation of rate limits in a distributed environment, where APIs are deployed across multiple independent nodes and accessed through a system of load balancers. This approach is essential for maintaining API performance, security, and fairness, particularly as modern applications increasingly rely on distributed architectures to enhance scalability, resilience, and geographic redundancy. Unlike single-node rate limiting, where all decisions are made locally, distributed rate limiting requires coordinating multiple nodes while maintaining a globally consistent enforcement of *client-specific* rate limits.

One of the key challenges in distributed API rate limiting is maintaining a consistent *state* across multiple nodes. Client-specific data, such as request counters, timestamps, and quotas, must be synchronized across nodes to ensure accurate enforcement. However, inconsistencies can arise due to network delays, concurrent updates, and node failures, allowing clients to exceed rate limits by sending requests to different nodes before state is synchronized. Achieving strong state consistency may introduce high synchronization overhead, thereby increasing processing time, resource consumption, and degrading performance. Weaker state consistency implementations may reduce the rate limit enforcement accuracy.

Scalability is another challenge, as the rate limiting system must scale to handle the increasing number of requests and

clients across multiple distributed nodes. The selection of appropriate *rate limiting algorithms* is also essential, as different use cases may require varying trade-offs between accuracy, performance, and resource utilization. Providers must carefully evaluate their specific rate limiting needs - considering factors like request patterns, accuracy requirements, and system efficiency - to choose implementations that balance effective request control with optimal performance in distributed environments.

Our approach in this research is to implement distributed API rate limiting at the *load balancer* level, rather than delegating it to application servers or external services. Load balancers, as the central entry point for API traffic, are well-suited for enforcing rate limits, particularly in distributed environments. By integrating rate limiting within load balancers, API usage can be centrally controlled across all deployed nodes, ensuring uniform policy enforcement and simplifying management. Additionally, since load balancers are designed to handle high traffic volumes and scale horizontally, this approach can efficiently accommodate the increasing demand.

Implementing rate limiting at the load balancer level also eliminates the need for communication between application servers for state synchronization, minimizes load, and provides clear separation of concerns. This approach also removes the dependency on additional services for rate limiting, as load balancers are already part of the network path, simplifying the setup and integration of rate limiting capabilities.

To effectively implement distributed rate limiting through load balancers, choosing an appropriate rate limiting algorithm for the specific use case and *synchronizing the state* between load balancer instances with minimal inaccuracies and performance overheads is crucial. Potential candidates for achieving this synchronization include in-memory databases like Redis, relational databases like MySQL, and Conflict-Free Replicated Data Types (CRDTs).

To guide this research, the following key questions are addressed:

- **RQ1** How does implementing distributed API rate limiting through load balancers using different algorithms impact latency, throughput, and throttling accuracy, and how can they be further optimized?
- **RQ2** How do different state synchronization mechanisms for synchronizing rate limit data between load balancers compare in terms of performance and consistency?

The rest of the paper is organized as follows: Section II discusses related work, Section III details the methodology of our approach, and Section IV elaborates on the implementations. Section V presents the evaluation plan and discusses the results of our approach. Finally, Section VI concludes the paper with key findings and future directions.

II. Related Work

API rate limiting can be enforced through various patterns [4]. It is typically implemented at different layers, including API gateways, reverse proxies, application servers, or through external services. Rate limits may be applied *globally* across all distributed nodes or *locally* on individual nodes. These limits can be based on a client's API key, IP address, access token, or any other unique identifier.

Most state-of-the-art research on API rate limiting focuses on non-distributed setups, where a single node enforces local rate limits. The study by Malki, Zdun and Pautasso analyzes the impact of API rate limiting on the reliability of microservices-based architectures, using Kong API Gateway's built-in local rate limiting functionality [5]. Subsequent research by Malki and Zdun examines the combined effects of rate limiting, load balancing, and request bundling in microservices [6]. Both these studies do not explore the effects of different rate limiting algorithms nor the complexities of distributed rate limiting. However, they provide a foundation for workload simulations and benchmarking.

Several other studies have investigated distributed rate limiting, but primarily in network bandwidth control rather than API management. The work by Stanojevic and Shorten [7] introduces a server-centric approach using upstream routers to regulate network traffic. Similarly, work [8] proposes algorithms to enforce network bandwidth limits using a centralized token bucket mechanism. Although these techniques share similarities with API rate limiting, they do not address API-specific requirements such as client-based enforcement.

Henriksson and Bennhage [9] critically evaluate Spotify's rate limiting system, which leverages CRDTs to manage global state using Token Bucket algorithm. They propose an alternative approach that integrates optimistic replication and queuing mechanism to improve accuracy and efficiency. While their study focuses on Spotify's proprietary implementation, their findings highlight the advantages of using CRDTs for state synchronization in distributed rate limiting systems.

In existing industrial API implementations, rate limiting of APIs is commonly enforced in two ways: directly at the *endpoints* or through *middlewares*. Endpoint implementations involve applying rate limiting mechanisms directly at either the client-side or server-side. Alternatively, middleware implementations handle these controls within an intermediary infrastructure, such as API gateways, load balancers, or proxy services, which manage the traffic between the client and server.

Client-side rate limiting is less common than traditional server-side approaches due to challenges in enforcement and synchronization. Doorman [10] is one such system that allows clients interacting with shared resources to voluntarily limit their usage. However, maintaining fairness across numerous clients, ensuring consistent enforcement, and managing complex client-side configurations make this approach less practical. Additionally, it lacks the centralized control and security of server-side rate limiting, reducing its effectiveness in managing API traffic efficiently.

There are several libraries used for API rate limiting that require integration at the application level [11]–[15]. These libraries implement various rate limiting algorithms, with some supporting distributed rate limiting by storing client state in an external database or cache. However, implementing rate

limiting at the application level adds complexity, increases overhead, and requires modifying existing services. Additionally, since requests must reach the application servers to check rate limits, this introduces extra network calls and potential performance delays. A more efficient approach is to enforce rate limits earlier, within a middleware such as a load balancer or API gateway, ensuring centralized control and reducing the application-level burden.

Reference [16] compares four common rate limiting algorithms using the Bucket4j server-side library [11], which uses the Token Bucket algorithm by default. The authors analyze factors like ease of implementation, data traffic handling, data starvation, and memory usage. While this study provides an analysis of these algorithms, it focuses solely on the Bucket4j library and does not consider a distributed API setup with state synchronization mechanisms, making it difficult to evaluate the algorithms in a realistic distributed environment. Additionally, the study lacks detailed performance metrics and does not incorporate real-world traffic patterns in its experiments. Despite these limitations, the work provides a good foundation for understanding these algorithms.

API gateways are another candidate where rate limiting processes are generally implemented along with other functionalities [17]. It can provide local as well as distributed API rate limiting solutions. Kong API Gateway and WSO2 API Gateway support distributed API rate limiting by integrating with Redis to centrally store and synchronize state among multiple instances. Furthermore, intermediate reverse proxies are used to rate limit APIs according to the specific needs of API providers [18], [19].

Load balancers, on the other hand, are often positioned at the front of distributed services, arguably offering a more effective way to implement distributed rate limiting. They provide a natural point of interception for incoming requests, eliminating the need for additional setup and seamlessly scaling with the overall system [20], [21]. Also, research [6] concludes that implementing both load balancing and rate limiting to APIs improves the overall reliability of micro services architecture. Nginx [22] supports local rate limiting, whereas Envoy Proxy [23] enables both local and global rate limiting through an externally deployed service.

Solutions discussed in both academic research as well as industrial implementations have used various algorithms to rate limit APIs, based on their specific use cases and needs. Table I summarizes the widely used algorithms and their sources.

As a summary, existing research on API rate limiting is biased toward non-distributed implementations, and prior work on distributed rate limiting is largely focused on network bandwidth control rather than APIs. Middleware-based approaches, particularly load balancer based rate limiting, remain unexplored in academia despite being used in industries. Furthermore, no comprehensive studies have compared the performance and accuracy of different rate limiting algorithms or explored ways to further optimize them with different state synchronization mechanisms in distributed environments.

TABLE I. LIST OF RATE LIMITING ALGORITHMS

Algorithm	Source
Fixed Window Counter	[5], [12], [18]
Sliding Window Log	[5], [18]
Sliding Window Counter	[24], [25]
Token Bucket	[5], [9], [16], [22]
Leaky Bucket	[5], [22], [26]
GCRA	[27]

III. METHODOLOGY

This study proposes a *distributed API rate limiting framework* built on load balancers. The framework utilizes load balancers to enforce rate limiting policies while ensuring state consistency across multiple instances through *external data stores*. Fig. 1 presents a high-level overview of the framework executed in load balancers. The following subsections provide a detailed discussion of the selected load balancer and state synchronization mechanisms.

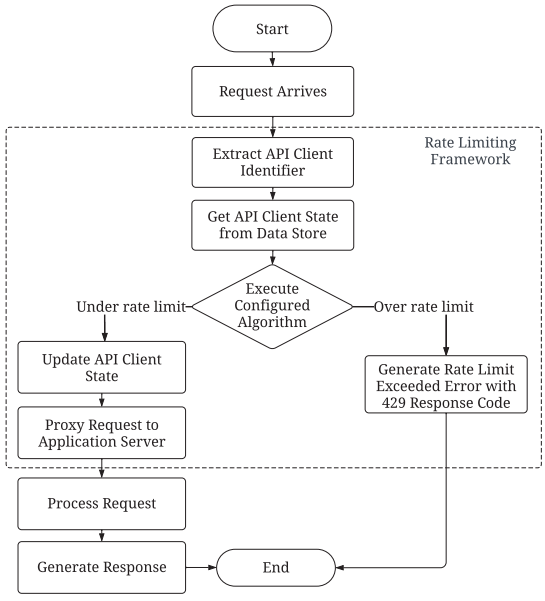

Fig. 1. Rate Limiting Framework

A. Load Balancer

Nginx has been chosen as the candidate for load balancer, in which the distributed rate limiting framework will be implemented. It is a widely used open-source HTTP server, reverse proxy, and load balancer known for its high performance, stability, and low resource consumption. As of February 2025, Nginx serves 20.34% of the top 1,180,650,484 websites in the world, which is the highest among all web servers [28].

Nginx's functionalities can be extended through Lua scripting via OpenResty [29], enabling the implementation of custom procedures. While Nginx offers built-in local rate limiting, it lacks native support for global distributed rate limiting. By using Lua scripting and integrating with external data stores,

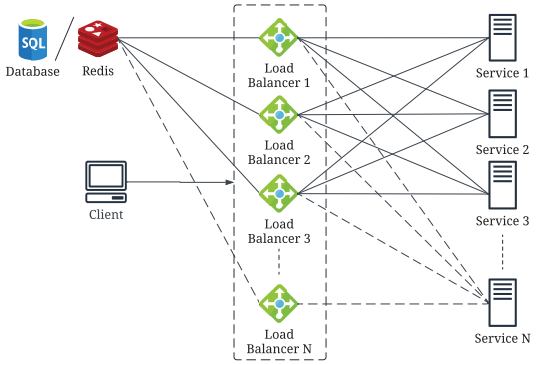

Fig. 2. State Synchronization Using Central Data Stores

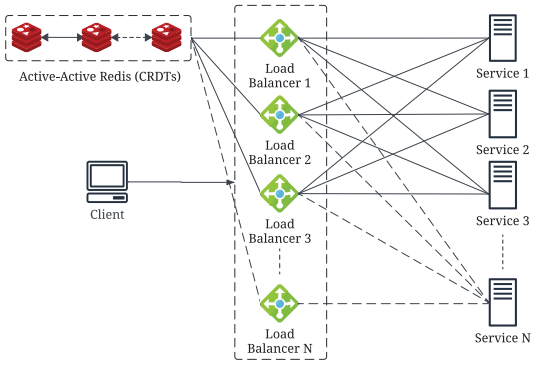

Fig. 3. State Synchronization Using Active-Active Redis (CRDTs-based)

we can implement a distributed rate limiting framework. This setup will enable us to conduct experiments, benchmarking and comparisons, making it a suitable candidate for our research.

B. State Synchronization

In this context, *state* refers to counters, timestamps, or quotas associated with each unique client making requests to API services, based on the specific rate limiting algorithm. Effective state synchronization among load balancers is crucial for ensuring the consistency and accuracy of rate limiting. In this study, we have selected *centralized* approaches like Redis and MySQL, as well as *decentralized* approaches like Conflict-free Replicated Data Types (CRDTs) for this purpose, as depicted in Fig. 2 and 3.

Redis is a high-performance, in-memory data store widely used for caching and real-time analytics due to its low latency and high throughput capabilities [30]. It supports multiple data structures, making it ideal for various rate limiting algorithms. Its ability to perform operations in memory reduces latency, making it suitable for high-traffic environments where rate limits need frequent checking and updating. Redis's native support for replication and clustering ensures reliable performance and fault tolerance in distributed rate limiting scenarios.

Modern relational databases offer strong transactional guarantees and ACID properties, ensuring that rate limit state is updated reliably and consistently. Among this, MySQL has been selected for its robustness and widespread adoption in enterprise applications. It offers strong consistency through its support for transactions via InnoDB, ensuring that updates to rate limit states are atomic and isolated [31]. It also provides the option of persisting the client's request history.

CRDTs are specialized data structures designed for distributed systems to achieve *eventual consistency* without conflicts [32], [33]. These data types allow concurrent updates from multiple nodes without the need for real-time coordination, making them suitable for environments where network partitions or temporary disconnections may occur. An additional advantage is that load balancers can communicate with any of the CRDT nodes based on their proximity or availability, reducing network access time and minimizing delays in rate limit enforcement. However, the impact of eventual consistency on the accuracy of rate limiting should be studied. We will be using CRDTs available in *Redis Enterprise Active-Active Geo-Distribution* implementation, which are in-built with the functionalities we require but incur subscription costs in production environments [34].

IV. Implementation

The surveyed algorithms were implemented as shown in Algorithm 1, 2, 3, 4, 5 and 6 as Lua scripts and initially tested in a local setup using JMeter. This provided insights into how each algorithm handles different traffic patterns and constraints, revealing their strengths, limitations, and the optimizations needed for effective adaptation in distributed environments. Each algorithm was later developed in *multiple versions*, varying based on the state synchronization mechanism, algorithm-specific parameters, and asynchronous implementations where applicable. All implemented versions are published as open-source for OpenResty in [35].

A. Fixed Window Counter

This divides time into fixed windows and tracks the number of requests within each window. If the number of requests exceeds the limit during a window, subsequent requests are denied until the next window starts. This algorithm is simple and memory-efficient, requiring only a counter per window. It is easy to implement and provides predictable rate limiting behavior. However, it suffers from the boundary problem, where a burst of requests at the end of one window and the beginning of the next can exceed the limit within a short time. This makes it unsuitable for scenarios requiring a smooth distribution of requests.

B. Sliding Window Log

The Sliding Window Log algorithm offers a more precise approach by recording the timestamp of each request in a log, and continuously checking the number of requests within a rolling time window. This provides highly accurate rate limiting since it maintains an exact history of request timestamps. This precision allows smoother rate enforcement, avoiding sudden spikes allowed by the Fixed Window Counter. However, it has a significant memory overhead, as it must store

Algorithm 1 Fixed Window Counter

```
Initialize:
    count ← 0
    window_start_time ← -1
procedure RATE_LIMIT(window_size, max_requests)
    current_time ← get_unix_timestamp()
    window_start ← floor(current_time / window_size) × window_size
    if window_start ≠ window_start_time then
        count ← 0
        window_start_time ← window_start
    end if
    if count < max_requests then
        count ← count + 1
        return true                          ▷ Request Allowed
    end if
    return false                             ▷ Request Denied
end procedure
```

Algorithm 3 Sliding Window Counter

```
Initialize:
    counter ← empty map
procedure RATE_LIMIT(window_size, max_requests)
    current_time ← get_unix_timestamp()
    cur_window ← floor(current_time / window_size) × window_size
    prev_window ← cur_window - window_size
    elapsed ← current_time - cur_window
    weight ← 1 - (elapsed / window_size)
    count ← counter[prev_window] × weight + counter[cur_window]
    if count < max_requests then
        counter[cur_window] ← counter[cur_window] + 1
        return true                          ▷ Request Allowed
    end if
    return false                             ▷ Request Denied
end procedure
```

and process timestamps for each request. Performance can also degrade under heavy traffic due to the continuous need for log maintenance and lookup.

Algorithm 2 Sliding Window Log

```
Initialize:
    requests_log ← empty list
procedure RATE_LIMIT(window_size, max_requests)
    current_time ← get_unix_timestamp()
    Remove entries from requests_log that are older than (current_time − window_size)
    if length of requests_log < max_requests then
        append current_time to requests_log
        return true                          ▷ Request Allowed
    end if
    return false                             ▷ Request Denied
end procedure
```

C. Sliding Window Counter

This combines elements of both Fixed Window Counter and Sliding Window Log. It divides time into smaller sub-windows and maintains a count of requests in each sub-window. The overall request rate is then calculated by summing the counts of the current and previous sub-windows, with a weight for the oldest sub-window based on the elapsed time. This balances accuracy and memory efficiency by approximating the request rate using sub-window counters. It smooths out abrupt rate enforcement inconsistencies seen in the Fixed Window Counter. However, it still introduces some imprecision, as the counting granularity depends on the chosen number of sub-windows. Algorithm 3 uses two sub-window counters for this approximation.

D. Token Bucket

Token Bucket works by maintaining a bucket filled with tokens, each representing a unit of request capacity. Tokens are added to the bucket at a constant rate, and each incoming request consumes a token (or multiple tokens for a heavy request). If the bucket has no tokens left, the request is denied. This replenishment process can be carried out either via a separate process that periodically adds tokens at a fixed rate or by dynamically calculating and adding the number of tokens at each request, based on the elapsed time. For our implementations, we have used the second approach.

This algorithm is highly flexible and allows short bursts of requests while maintaining a steady long-term rate. It ensures fairness by allowing users to save up unused tokens for later bursts. However, allowing short bursts could overwhelm the system during high traffic scenarios. Selecting the size and refill rate of the bucket requires careful consideration to effectively balance burst handling with overall traffic control.

Algorithm 4 Token Bucket

```
Initialize:
    tokens ← bucket_capacity
    last_refill_time ← get_unix_timestamp()
procedure RATE_LIMIT(bucket_capacity, refill_rate)
    current_time ← get_unix_timestamp()
    elapsed_time ← current_time - last_refill_time
    new_tokens ← elapsed_time × refill_rate
    tokens ← min(bucket_capacity, tokens + new_tokens)
    last_refill_time ← current_time
    if tokens ≥ 1 then
        tokens ← tokens - 1
        return true                          ▷ Request Allowed
    end if
    return false                             ▷ Request Denied
end procedure
```

E. Leaky Bucket

The Leaky Bucket algorithm controls request rates by processing them at a fixed pace, similar to a leaking bucket. Incoming requests are added to the bucket, and if it overflows, excess requests are discarded. This enforces a steady request rate, preventing bursts and ensuring uniform load distribution. However, sudden spikes can fill the queue quickly, leading to the rejection of new requests. Asynchronous responses may also cause delays due to queued processing.

In our implementation, a queue is maintained to store request leak times. When a request arrives, the necessary delay is determined to maintain the expected leak rate, its leak time is computed, and it is appended to the queue. For subsequent requests, the leak time is calculated based on the previous request. The bucket capacity is determined by the leak rate and the maximum acceptable delay for user experience.

Algorithm 5 Leaky Bucket

```
Initialize:
    queue ← empty queue
procedure RATE_LIMIT(leak_rate, bucket_capacity)
    current_time ← get_unix_timestamp()
    last_leak_time ← current_time
    if queue is not empty then
        last_leak_time ← leak_time of last request
    end if
    dequeue leak times that are older than current_time
    if length of queue < bucket_capacity then
        default_delay ← 1 / leak_rate
        time_diff ← current_time - last_leak_time
        delay ← 0
        if time_diff ≠ 0 then
            delay ← max(0, default_delay - time_diff)
        end if
        leak_time ← current_time + delay
        enqueue leak_time
        return true, delay                    ▷ Request Allowed
    end if
    return false                              ▷ Request Denied
end procedure
```

F. Generic Cell Rate Algorithm (GCRA)

GCRA extends the Leaky Bucket concept to offer more flexible traffic management while maintaining minimal memory usage. It operates by calculating a theoretical arrival time (TAT) for each request, which is adjusted based on the emission interval and burst capacity. It enforces equal spacing between requests, ensuring a smooth and predictable traffic flow. To provide some flexibility for users, GCRA permits limited bursts while still preventing sustained overflows.

Adapting these algorithms in distributed environments presents a unique set of challenges compared to traditional non-distributed environments. Achieving effective rate limiting in such distributed environments requires balancing both performance and accuracy. Concurrent access from multiple clients in high-traffic scenarios introduces the risk of race conditions, which can lead to inconsistent states if not properly managed. Additionally, frequent synchronization of the rate limiting state with external data stores may impose significant overhead, affecting performance.

Algorithm 6 Generic Cell Rate Algorithm

```
Initialize:
    TAT ← -1
procedure RATE_LIMIT(window_size, max_requests, burst_capacity)
    emission_interval ← window_size / max_requests
    delay_tolerance ← emission_interval * burst_capacity
    current_time ← get_unix_timestamp()
    if TAT = -1 then
        TAT ← current_time
    else
        TAT ← max(current_time, TAT)
    end if
    allow_at ← TAT - delay_tolerance
    if current_time ≥ allow_at then
        TAT ← TAT + emission_interval
        return true                           ▷ Request Allowed
    end if
    return false                              ▷ Request Denied
end procedure
```

In this context, *race conditions* can arise primarily due to two reasons:

1) Client's requests may be routed to multiple Nginx instances, leading to concurrent access to the shared data store by Nginx instances.
2) Multiple worker processes within an Nginx instance may concurrently access the shared data store or the local shared memory space, in case of batch-quota-based implementations.

Without proper concurrency control, these factors can result in inaccurate rate limit enforcement.

To mitigate these concurrency issues, several approaches were evaluated with consideration of their performance impact. In Redis, in-built data structures supporting atomic operations were used for simple operations, while Lua scripts, Redis locks, and Multi-Exec transactions were tested for more complex logic. After this evaluation, Lua scripts, which execute atomically within Redis, were chosen for their high performance due to a single network round trip and the ability to be cached locally. Similarly, MySQL-based implementations leveraged stored procedures, per-client row-level locks, and transactions with the *read-committed* isolation level. Additionally, explicit locks provided by OpenResty were used within Nginx to control concurrent access to shared memory space between multiple worker processes.

In distributed API rate limiting, synchronizing state with external data stores for every request may introduce significant communication overhead, leading to increased latency. To address this, *asynchronous batch-quota-based* versions of

applicable algorithms were developed. Instead of querying and updating the client state in data store for each request, load balancer instances fetch quotas in batches from the global data store, manage them locally, and use them for rate limit decisions on subsequent requests, until the batch is exhausted. This approach is implemented for Fixed Window Counter, Sliding Window Log, Sliding Window Counter, and Token Bucket algorithms. However, implementing them for Leaky Bucket and GCRA was impractical as Leaky Bucket queues requests and processes them in real-time, making batch processing infeasible, while GCRA relies on theoretical arrival time calculations, which do not align with batch-based updates.

These asynchronous versions can improve performance by minimizing external data store network calls, reducing contention, and lowering synchronization latency. This optimization is particularly beneficial when network latency has a significant impact. However, it may lead to inaccuracies in the overall global rate limit in scenarios where multiple load balancers fetch a local batch quota for a specific client and synchronize unevenly. Batch-quota fetching can be implemented in two ways:

1) **Lazy Update:** The shared data store is updated only after a local batch is exhausted. This approach may allow requests to exceed the rate limit.
2) **Eager Update**: The shared data store is updated while fetching a new quota. This approach may restrict requests before the actual global rate limit is reached.

The maximum deviation from the configured rate limit depends on the *number of load balancers* and the *maximum fetchable batch quota*. Larger batch sizes and more nodes increase the potential for deviation, either exceeding or restricting the effective limit depending on the update strategy used.

This maximum deviation can be controlled by adjusting the maximum amount of batch quota a load balancer can fetch at a time. In static batch quota implementation, a load balancer fetches a percentage of the remaining global quota for a specific client, stores it locally, and uses it for subsequent requests. By adjusting this percentage, the maximum batch quota a load balancer can fetch is controlled. Additionally, in dynamic batch quota implementation, the load balancer tracks a client's request history over previous time windows and adjusts the next batch quota to be fetched based on recent client behavior. This dynamic approach further enhances rate-limiting accuracy in asynchronous implementations, reducing under-fetching or over-fetching.

API providers should allow a certain *grace percentage* over the configured rate limits when using asynchronous versions of the algorithms to ensure adherence to Service Level Agreements (SLAs) even in case of maximum deviations, offering a practical trade-off for potential better average latency. However, these deviations are negligible if requests from a specific client are typically routed through the same load balancer. Benchmarking these asynchronous implementations will help evaluate how it enhances performance and assess accuracy trade-offs.

Further, the communication between load balancers and the external data store should also be properly configured to handle high traffic scenarios. Connection pooling should be configured in load balancers to minimize connection provisioning overheads. Frequently executed rate limiting logic should be cached locally within the data store to reduce the network bandwidth of requests. This can be implemented through mechanisms like Lua scripts in Redis and stored procedures in MySQL.

V. Evaluation

The goal is to test and compare the performance and accuracy of various rate limiting algorithms for distributed APIs, within load balancers, using different state synchronization mechanisms. The experiments are conducted in both local and cloud deployment setups. A traffic pattern modeled after real-world scenarios was used to obtain benchmark results. To enable the reproducibility of our study, we have published all test artifacts as open-access [36].

A. Benchmark Workload Scenario

We have used TeaStore [37] as the backend service, which is a renowned micro-service benchmarking and reference application designed to emulate a basic web store environment. It contains five distinct services: WebUI, Auth, Recommender, Persistence, and Image, all of which communicate via REST. This architecture makes TeaStore an ideal platform to generate realistic API traffic. HTTP requests for these services will be made directly through load balancers, rather than using the existing WebUI component. Our test scenario, which is extracted from TeaStore, includes eight key HTTP transactions, each consisting of multiple GET and POST requests simulating typical API usage. Users perform actions such as logging in, browsing the store for products, adding these products to the shopping cart, and then logging out.

B. Experimental Setup

Apache JMeter version 5.6.3 is used as the load generator to simulate client traffic. Load balancers, data stores, and backend services are deployed as Docker containers for easy deployment and management across environments. OpenResty version 1.27.1, Redis version 7.4.2, MySQL version 8.4.4, Redis Enterprise version 7.8.4 and TeaStore version 1.4.2 have been utilized for the setup.

Rate limiting algorithms are implemented as Lua filters within Nginx. The selected algorithm will be dynamically mounted to Nginx containers at runtime through the Nginx configuration file. To effectively support asynchronous versions of the algorithms, Nginx is configured with a shared memory space among worker processes for local state storage.

Experiments are conducted in two distinct environments:

- **Local Setup:** Initial development and functional testing are carried out in a local environment.

- **Cloud Deployment:** Performance of the implementations is evaluated in a public cloud environment, simulating real-world cloud conditions. Conclusions are derived from these results.

The final deployment is hosted on Google Cloud Platform (GCP), utilizing four Compute Engine instances in a single-region, as shown in Table II. Each virtual machine (VM) is provisioned with an e2-standard-4 machine type, featuring 4 vCPUs, 16 GB of RAM, HDD-based persistent disks and running Ubuntu 22.04 as the operating system.

TABLE II. GCP VM SETUP

VM	Region
Data Store (Redis, MySQL)	us-central1
Load Balancer 1	us-central1
Load Balancer 2	us-central1
Application Server (Teastore)	us-central1

For the CRDT-based setup, Redis Enterprise clusters are deployed as Docker containers within each Load Balancer VM and synchronized using eventual consistency. Logging is configured via JMeter to log the timestamps of requests along with their response codes and latency. Our evaluation metrics are calculated based on these logs. VM instances are monitored for resource consumption by the GCP Ops Agent. This logging and monitoring infrastructure will help in detailed performance analysis of our setups.

C. Test Configurations

The discussed scenario was applied iteratively over a 10 minute period, including a ramp-up period of 10 seconds, accounting for over 45,000 requests to test each implementation. API requests from each client were distributed between the two load balancers in a round-robin manner. A global aggregated rate limit was enforced across all TeaStore services and applied individually to each client. Clients subscribed to the APIs are uniquely identified by *API tokens*, which are assigned during the API subscription process and are appended with each request.

A setup thread was first employed to send warm-up requests for 60 seconds, ensuring that the system reached a stable state before the main testing began. The test execution included 100 concurrent users (clients) divided into 3 distinct thread groups, each simulating different request rates to model diverse user traffic patterns, as shown in Table III. Request rate was controlled using JMeter's Constant Throughput Timer, which maintains a consistent request rate for each thread within a thread group. Requests exceeding the rate limit were subjected to a retry mechanism.

The rate limiting framework was tested with five algorithms: Fixed Window Counter, Sliding Window Log, Sliding Window Counter, Token Bucket and GCRA. The Leaky Bucket algorithm was excluded as it queues requests and processes them at a constant rate, unlike other algorithms that reject excess requests. This difference makes direct comparison infeasible for real-time request rejection and fairness.

TABLE III. TEST CONFIGURATION FOR USER TRAFFIC SIMULATION

User Group	Percentage of Users	Request Rate (req/min)
High Traffic Users	5%	120 (Above limit)
Moderate Traffic Users	25%	90 (Below limit)
Low Traffic Users	75%	30 (Well below limit)

Each algorithm was configured with a rate limit of 100 requests per minute, with parameters tailored to its respective mechanism. Table IV summarizes the configurations used for each algorithm during testing. All algorithms were evaluated across the three state synchronization mechanisms discussed.

Asynchronous batch-quota-based versions were tested with Redis-based implementation, using a static batch-quota fetch of 50% of the remaining global quota and a lazy update mechanism. Additionally, tests were also conducted without the rate limiting framework, where requests were directly proxied from the load balancer to the application server without any rate limiting checks. This allowed for assessing and comparing the overhead introduced by the rate limiting framework on API performance.

TABLE IV. ALGORITHM CONFIGURATIONS FOR TESTING

Algorithm	Parameter	Value
Fixed Window Counter	window_size	60 seconds
	max_requests	100
Sliding Window Log	window_size	60 seconds
	max_requests	100
Sliding Window Counter	window_size	60 seconds
	max_requests	100
	sub_window_count	5
Token Bucket	bucket_capacity	5
	refill_rate	1.67 tokens/second
GCRA	window_size	60 seconds
	max_requests	100
	burst_capacity	5

D. Evaluation Metrics

The *independent variables* in the experimental setup are the rate limiting algorithm and the state synchronization mechanism used by the load balancers. To evaluate performance, we have selected latency and throughput, the key metrics commonly used to benchmark microservice-based architectures and APIs [38]. Additionally, throttling deviation will be measured to assess the accuracy of rate limit enforcement.

Latency measures the time from request submission to response, while throughput captures the number of requests processed per unit time. These metrics are used to evaluate both the overhead introduced by the rate limiting framework on API access and the comparative performance of different rate limiting algorithms and state synchronization mechanisms. Application servers are not considered bottlenecks, as the VMs are provisioned with adequate resources.

Throttling deviation quantifies how accurately an implementation enforces the configured limits. It is measured as

the percentage of requests that deviate from the configured rate limit across all clients in a given test scenario. For each client, the number of deviating requests is determined using Algorithm 7 and aggregated to compute the total number of deviations in the test.

Algorithm 7 Calculating Number Of Deviations Per Client

```
Initialize:
    total_deviation ← 0
    window_size ← 60 seconds
    rate_limit ← maximum requests allowed in window
Sort client request logs by request.timestamp in ascending order
for each request in client request logs do
    window_start ← request.timestamp - window_size
    window_end ← request.timestamp
    successful_requests ← Count requests with response code 200 from window_start to window_end
    if successful_requests < rate_limit then
        expected_response ← 200
    else
        expected_response ← 429
    end if
    actual_response ← request.response_code
    if actual_response ≠ expected_response then
        total_deviation ← total_deviation + 1
    end if
end for
```

Once the total number of deviations is obtained, throttling deviation is calculated using the following formula:

$$Throttling\ Deviation = \frac{Total\ Deviations}{Total\ Requests} \times 100\%$$

A high throttling deviation indicates that the rate-limiting implementation fails to accurately enforce the configured rate limit. This deviation can occur due to the framework being either overly restrictive, causing false positives (throttling requests that should be allowed), or overly relaxed, causing false negatives (allowing requests that should be throttled).

By analyzing and interpreting these metrics, valuable insights can be provided for the development of rate limiting strategies for distributed API deployments.

E. Results Analysis

The rate limiting framework introduced negligible latency overhead, compared to the non-rate limited implementation as observed in Table V. Comparing latency between algorithms based on the state synchronization mechanism as in Fig. 4, shows that despite theoretical differences in computational complexity and client state handling, all algorithms performed similarly, suggesting that these differences have little impact on response times in real-world API access. This uniform and low-latency performance of algorithms is largely due to optimizations in the execution environment for efficient concurrency handling and reduced network round-trips to data stores, as discussed in Section IV.

Among the state synchronization mechanisms, CRDT-based synchronization exhibited the lowest latency across all algorithms, with almost negligible overhead compared to the non-rate-limited implementation. This is due to its ability to be deployed in proximity to load balancers and its eventual consistency model, which could offer even better performance in multi-regional deployments. Standard Redis-based synchronization performed second-best, making it a strong alternative, particularly for single-region deployments. In contrast, the MySQL-based implementation showed a comparatively higher latency, likely due to disk access overhead compared to in-memory Redis. It could be beneficial in scenarios where persisting client requests for analysis are required.

TABLE V. LATENCY OVERHEAD DUE TO RATE LIMITING FRAMEWORK

Algorithm	State Synchronization					
	CRDT		Redis		MySQL	
	ms	%	ms	%	ms	%
Fixed Window Counter	1.43	0.54	2.76	1.04	7.50	2.82
Sliding Window Log	0.61	0.23	1.04	0.39	8.62	3.24
Sliding Window Counter	2.09	0.79	7.36	2.77	8.45	3.18
Token Bucket	1.17	0.44	1.31	0.49	8.99	3.38
GCRA	1.07	0.40	2.86	1.08	7.74	2.91

Fig. 5 indicates that the throttling deviation of each algorithm remained consistent regardless of the synchronization mechanism used. This suggests that all tested state synchronization methods maintained a similar level of consistency. The eventual consistency model of Redis Enterprise clusters using CRDTs had no significant impact on throttling deviation, comparatively. Among the tested algorithms, Sliding Window Log exhibited the lowest throttling deviation. This is attributed to its ability to log request timestamps with microsecond precision, enabling precise tracking of requests within the current window.

In contrast, the Fixed Window Counter showed the highest throttling deviation due to its rigid window boundaries, which can cause enforcement inconsistencies at window transitions. The Sliding Window Counter demonstrated better accuracy, with its deviation influenced by the number of sub-windows used. In this test, a configuration with five sub-windows yielded lower deviation (0.56%) compared to a two-sub-window setup (0.72%). The throttling deviation of Token Bucket depended on its bucket capacity, as it allows request bursts based on this parameter. Similarly, GCRA also accommodates bursts but operates using theoretical arrival times, making its deviation sensitive to the configured burst size. The specific configurations for each algorithm used in these tests are detailed in Table IV.

Fig. 6 indicates that there is minimal latency difference between normal and asynchronous batch-quota-based versions of the algorithms while using Redis, in a single-region cloud deployment. This is primarily due to Redis's in-memory

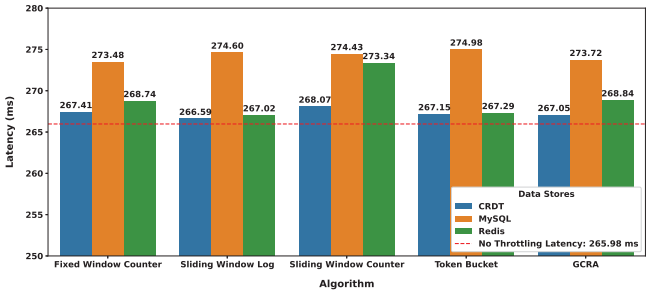

Fig. 4. Average Latency Comparison by Algorithms and Data Stores

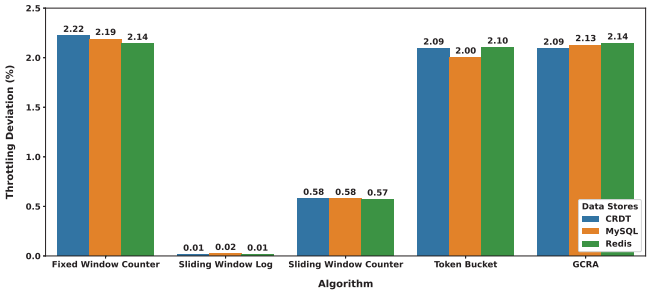

Fig. 5. Throttling Deviation (%) Comparison by Algorithms and Data Stores

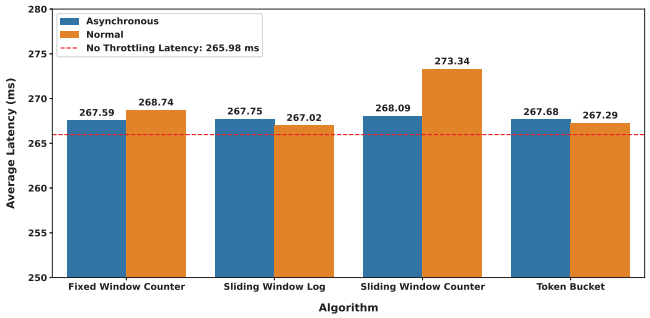

Fig. 6. Average Latency Comparison: Asynchronous vs. Normal versions of Algorithms using Redis

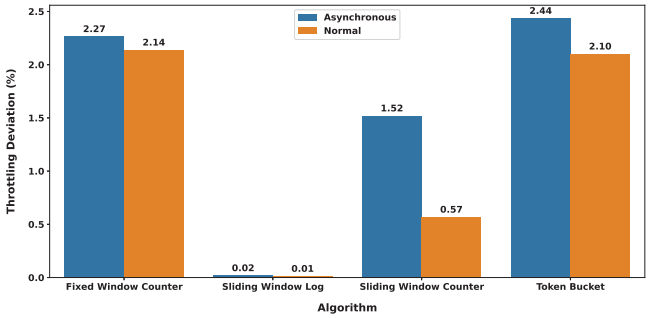

Fig. 7. Throttling Deviation (%) Comparison: Asynchronous vs. Normal versions of Algorithms using Redis

storage and highly optimized data structures, which enable rapid data access. Additionally, in single-region cloud setups, the network overhead between the load balancer and Redis is negligible, making normal versions accessing Redis as efficient as asynchronous versions accessing Nginx's local shared memory.

The performance advantages of asynchronous batch-quota-based versions become more apparent in scenarios where network latency between the load balancer and the data store is significant, such as in multi-region deployments. In such cases, batching requests can reduce the frequency of network calls, leading to improved overall response times. Fig. 7 indicates that there is a slightly higher throttling deviation in the asynchronous versions compared to the normal versions, which is in line with expectations. This deviation can be mitigated by reducing the static batch fetch percentage.

The throughput observed in all tests remained consistent with our configurations, as in Table III. This indicates that the rate limiting framework effectively managed the traffic and workload while maintaining the defined throughput targets across all implementations. Nginx's event-driven, non-blocking architecture combined with its multi-process model allowed it to efficiently handle concurrent requests. Additionally, OpenResty's optimized LuaJIT environment ensured that rate limiting logic was processed efficiently in-memory, while non-blocking request handling prevented external data store querying from affecting overall throughput.

VI. Conclusion and Future Work

Concerning **RQ1**, the results indicate that implementing distributed API rate limiting through load balancers introduces minimal performance impact, making it a highly viable and practical solution. In certain implementations, latency overhead remained below 1%, demonstrating its efficiency in real-world deployments. The differences in latency across the rate limiting algorithms when using a specific state synchronization mechanism are negligible in real-world scenarios. Throughput remained stable across tests, demonstrating that the framework effectively handled the configured traffic and workload. Throttling accuracy varied between algorithms, with Sliding Window Log exhibiting the lowest deviation due to precise timestamp tracking, while Fixed Window Counter had the highest deviation due to its rigid boundaries. Sliding Window Counter, Token Bucket, and GCRA showed moderate deviations, influenced by their respective configuration parameters, such as sub-window count and burst allowances.

Concerning **RQ2**, the choice of state synchronization mechanism significantly impacted latency, with CRDT-based synchronization achieving the lowest overhead due to its efficient eventual consistency model and ability to be deployed in proximity to load balancers. Standard Redis performed second best, making it a viable alternative, particularly for single-region setups. MySQL-based synchronization, however, introduced higher latency due to disk-based storage overhead. All three mechanisms showed similar throttling deviations for a particular algorithm, indicating a similar level of consistency.

These findings suggest that API providers should prioritize CRDT-based synchronization for optimal performance, with standard Redis serving as a strong alternative when CRDTs are not feasible.

Future work will focus on evaluating the scalability and effectiveness of state synchronization mechanisms and asynchronous implementations in multi-region setups. Additionally, stress testing should be conducted to investigate the performance and breaking points of different implementations under high-load scenarios. Also, further optimizations for the batch-quota-based implementations can be explored, such as integrating message queues to enable more efficient sharing of local quotas between load balancers.

REFERENCES

[1] K. T. Shishmano, V. D. Popov, and P. E. Popova, "Api strategy for enterprise digital ecosystem," in *2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T)*, 2021, pp. 129–134.

[2] Gartner, Inc. (2024, Mar.) Gartner predicts more than 30% of the increase in demand for apis will come from ai and tools using large language models by 2026. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2024-03-20-gartner-predicts-more-than-30-percent-of-the-increase-in-demand-for-apis-will-come-from-ai-and-tools-using-llms-by-2026

[3] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain, and S. Dustdar, "From the service-oriented architecture to the web api economy," *IEEE Internet Computing*, vol. 20, no. 4, pp. 64–68, 2016.

[4] S. Serbout, A. El Malki, C. Pautasso, and U. Zdun, "Api rate limit adoption – a pattern collection," in *Proceedings of the 28th European Conference on Pattern Languages of Programs*, ser. EuroPLoP '23. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3628034.3628039

[5] A. El Malki, U. Zdun, and C. Pautasso, "Impact of api rate limit on reliability of microservices-based architectures," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 19–28.

[6] A. E. Malki and U. Zdun, "Combining api patterns in microservice architectures: Performance and reliability analysis," in *2023 IEEE International Conference on Web Services (ICWS)*, 2023, pp. 246–257.

[7] R. Stanojevic and R. Shorten, "Load balancing vs. distributed rate limiting: An unifying framework for cloud control," in *2009 IEEE International Conference on Communications*, 2009, pp. 1–6.

[8] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, p. 337–348, Aug. 2007. [Online]. Available: https://doi.org/10.1145/1282427.1282419

[9] A. Henriksson and S. Bennhage, "Trading performance for precision in a crdt-based rate-limiting system," Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2021.

[10] Youtube. (2016) Doorman: Global Distributed Client Side Rate Limiting. [Online]. Available: https://github.com/youtube/doorman

[11] Bucket4j. (2024) bucket4j: Java rate limiting library based on token-bucket algorithm. [Online]. Available: https://github.com/bucket4j/bucket4j

[12] Django Ratelimit. Django ratelimit 4.1.0 documentation. [Online]. Available: https://django-ratelimit.readthedocs.io/en/stable/

[13] Google. Guava ratelimiter. [Online]. Available: https://guava.dev/releases/19.0/api/docs/index.html?com/google/common/util/concurrent/RateLimiter.html

[14] Python Software Foundation. Python rate-limiter using leaky-bucket algorithm. [Online]. Available: https://pypi.org/project/pyrate-limiter/

[16] M. V. Bartkov and D. Borovikov, "Selection of a suitable algorithm for the implementation of rate-limiter based on bucket4j," *International Journal of Online and Biomedical Engineering (iJOE)*, vol. 18, no. 04, p. pp. 52–63, 2022. [Online]. Available: https://online-journals.org/index.php/i-joe/article/view/25641

[15] Resilience4j. Getting started with resilience4j-ratelimiter. [Online]. Available: https://resilience4j.readme.io/docs/ratelimiter

[17] S. A. Ali and M. W. Zafar, "Api gateway architecture explained," *INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, vol. 5, no. 1, pp. 76–94, Mar. 2021. [Online]. Available: https://www.ijcst.com.pk/index.php/IJCST/article/view/450

[18] Kong Inc. Rate Limiting - Kong Gateway — Kong Docs. [Online]. Available: https://docs.konghq.com/gateway/latest/get-started/rate-limiting/

[19] WSO2 LLC. Distributed Burst Control, Backend Rate Limiting for API Gateway Cluster - WSO2 API Manager Documentation 4.3.0. [Online]. Available: https://apim.docs.wso2.com/en/latest/design/rate-limiting/advanced-topics/configuring-rate-limiting-api-gateway-cluster/

[20] R. K. Mondal, P. Ray, and D. Sarddar, "Load balancing," *International Journal of Research in Computer Applications & Information Technology*, vol. 4, pp. 1–21, 2016.

[21] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: A big picture," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 2, pp. 149–158, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157817303361

[22] NGINX. About: nginx. [Online]. Available: https://nginx.org/en/

[23] Envoy Project Authors. Envoy proxy - home. [Online]. Available: https://www.envoyproxy.io/

[24] Cloudflare, Inc. (2017, Jun.) How we built rate limiting capable of scaling to millions of domains. [Online]. Available: https://blog.cloudflare.com/counting-things-a-lot-of-different-things

[25] Alibaba Cloud. (2024, May) Alibaba sentinel rate limiting. [Online]. Available: https://www.alibabacloud.com/blog/601162

[26] F. D. Cas, "A practical approach to enhance web apis security using a stateless, open-source, pluggable api gateway," Master's thesis, Dept. of Comput. Sci. and Eng., Polytechnic Univ. of Milan, Milan, Italy, 2023.

[27] Redis. (January 2017) redis-cell: a rate limiting redis module. [Online]. Available: https://redis.io/blog/redis-cell-rate-limiting-redis-module/

[28] Netcraft Ltd. (2025, Feb.) February 2025 Web Server Survey — Netcraft. [Online]. Available: https://www.netcraft.com/blog/february-2025-web-server-survey/

[29] OpenResty, Inc. OpenResty - Scalable Web Platform by Extending NGINX with Lua. [Online]. Available: https://openresty.org/en/

[30] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards scalable and reliable in-memory storage system: A case study with redis," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 1660–1667.

[31] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A survey and comparison of relational and non-relational database," *International Journal of Engineering Research & Technology*, vol. 1, no. 6, p. 1–5, 2012.

[32] Redis. (2022, Mar.) Diving into Conflict-Free Replicated Data Types (CRDTs) - Redis. [Online]. Available: https://redis.io/blog/diving-into-crdts/

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[34] Redis. Active-Active Geo-Distribution (CRDTs-Based) — Redis Enterprise. [Online]. Available: https://redis.io/active-active/

[35] K. Thivaharan, P. Kobinarth, and J. Tharsigan. (2025) rate-limiter-nginx. [Online]. Available: https://github.com/thiva-k/rate-limiter-nginx

[36] ——. (2025) rate-limiter-nginx-artifacts. [Online]. Available: https://github.com/thiva-k/rate-limiter-nginx-artifacts

[37] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 223–236.

[38] N. Bjørndal, L. Araújo, A. Bucchiarone, N. Dragoni, M. Mazzara, and S. Dustdar, "Benchmarks and performance metrics for assessing the migration to microservice-based architectures," *Journal of Object Technology*, 08 2021.