Distributed URL Shortener

This project is a high-performance URL shortening service designed to handle heavy read/write traffic using a microservices-oriented architecture. By leveraging Redis for caching and Kafka for asynchronous event processing, the system achieves low redirect latency.

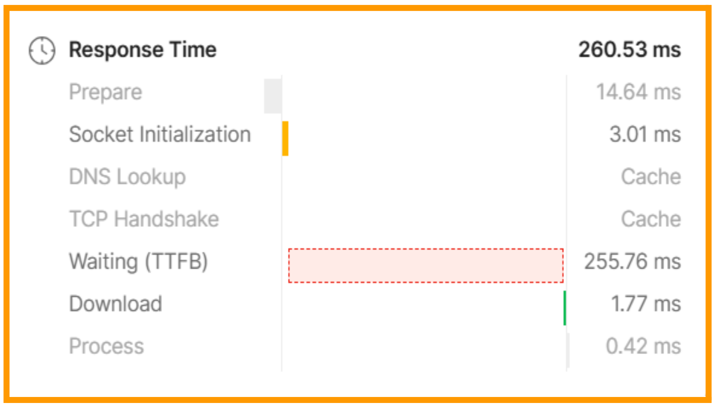
Key Technologies: * Backend: Node.js, Express.

- Database: MongoDB.
- Performance: Redis (Cache-Aside Pattern).
- Scalability: Apache Kafka & Zookeeper.
- Infrastructure: Nginx & Docker Compose

2. Problem Statement: The Latency Gap

In a standard URL shortener, every request requires a database lookup. During initial stress testing, my system exhibited high latency because every redirect was "blocking" while waiting for MongoDB to find the record and update the click count.

- **Observed Latency:** ~260ms.
- **The Goal:** Reduce latency to under 50ms and decouple analytics from the main request-response cycle.



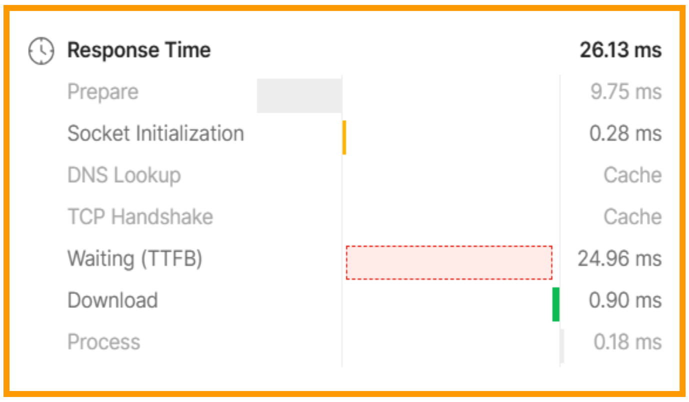
Response Time	260.53 ms
Prepare	14.64 ms
Socket Initialization	3.01 ms
DNS Lookup	Cache
TCP Handshake	Cache
Waiting (TTFB)	255.76 ms
Download	1.77 ms
Process	0.42 ms

3. Architecture & Performance Engineering

a) Redis Caching (Cache-Aside Strategy)

I implemented Redis to store the mapping of `shortId` to `redirectURL`. This prevents redundant database queries for frequently accessed links.

- Implementation: The backend checks Redis first; if data is missing, it fetches from MongoDB and "warms" the cache for 24 hours.
- Result: Redirect latency dropped to ~26ms, a 90% improvement in speed.

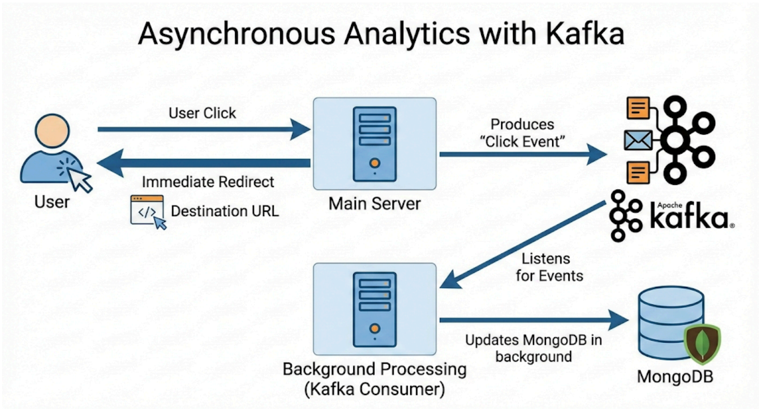


Response Time	26.13 ms
Prepare	9.75 ms
Socket Initialization	0.28 ms
DNS Lookup	Cache
TCP Handshake	Cache
Waiting (TTFB)	24.96 ms
Download	0.90 ms
Process	0.18 ms

b). Asynchronous Analytics with Kafka

To ensure the redirect is never delayed by analytics logging, I integrated Apache Kafka.

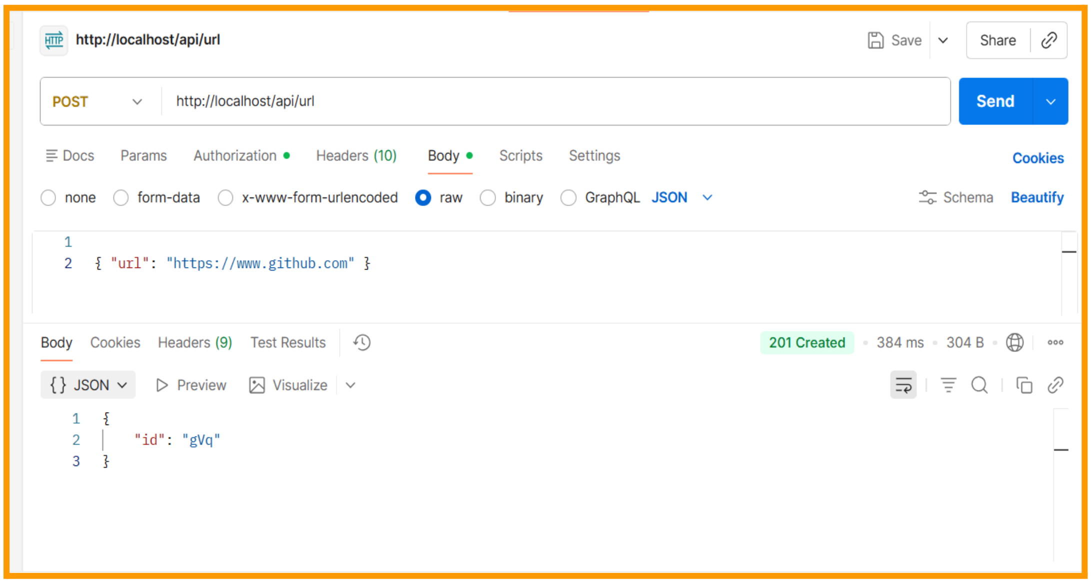
- Process: The main server produces a "Click Event" to a Kafka topic and redirects the user immediately.
- Background Processing: A separate Kafka Consumer listens for these events and updates MongoDB in the background without affecting the user's experience.



4. Security: Distributed Rate Limiting

To protect the system from DDoS attacks and automated scraping, I developed a **Distributed Rate Limiter** using Redis.

- **Algorithm:** I utilized the **Token Bucket Algorithm** to allow for small traffic bursts while enforcing a strict upper limit (configured to 10 requests/minute per IP).
- **Distributed State:** Rate limits are keyed by User ID/IP in a centralized Redis cluster. This ensures that the security policy is enforced consistently across all **Docker containers**, preventing users from bypassing limits by hitting different backend instances.



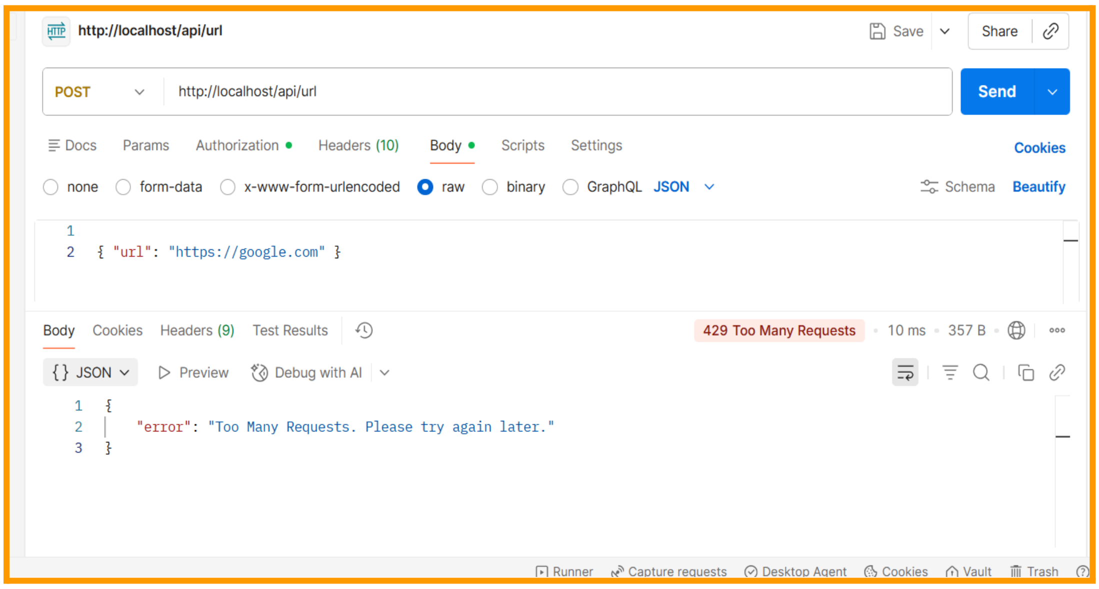
POST http://localhost/api/url

Body: raw JSON

```
{ "url": "https://www.github.com" }
```

201 Created · 384 ms · 304 B

```
{
    "id": "gVq"
}
```



POST http://localhost/api/url

Body: raw JSON

```
{ "url": "https://google.com" }
```

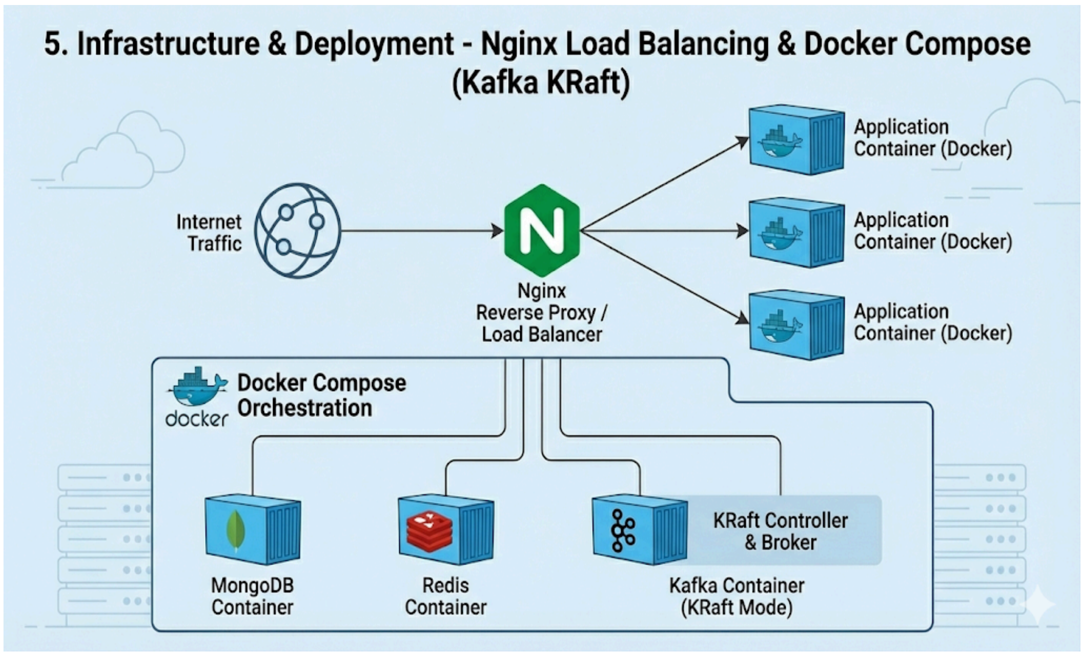
429 Too Many Requests · 10 ms · 357 B

```
{
    "error": "Too Many Requests. Please try again later."
}
```

5. Infrastructure & Deployment

The entire system is containerized for consistent deployment across any environment.

- Load Balancing: Nginx acts as a Reverse Proxy, distributing incoming traffic.
- Orchestration: Docker Compose manages the lifecycle of the Application, MongoDB, Redis, Kafka .


5. Infrastructure & Deployment - Nginx Load Balancing & Docker Compose (Kafka KRaft)
Internet Traffic
Nginx Reverse Proxy / Load Balancer
Application Container (Docker)
Application Container (Docker)
Application Container (Docker)
docker
Docker Compose Orchestration
MongoDB Container
Redis Container
KRaft Controller & Broker
Kafka Container (KRaft Mode)

6. Technical Challenges & Solutions

Challenge 1: Data Consistency & Atomic Updates

- **The Issue:** During initial integration, I encountered a data inconsistency bug where the database operation result (e.g., `{ modifiedCount: 1 }`) was inadvertently cached in Redis instead of the actual URL document. This caused subsequent redirect requests to fail.
- **The Solution:** I refactored the controller logic to utilize `findOneAndUpdate`. This atomic operation ensures the updated

document is returned immediately and cached correctly, maintaining data integrity between MongoDB and Redis.

Challenge 2: Distributed Message Durability

- **The Issue:** A critical requirement was ensuring that click analytics were never lost, even if a broker went offline during a traffic spike.
- **The Solution:** I configured the Kafka cluster using **KRaft** (Kafka Raft Metadata mode) instead of Zookeeper. This modern architecture simplifies the deployment while ensuring reliable leader election and message persistence across the cluster.

Challenge 3: Concurrency & Race Conditions

- **The Issue:** In a distributed system, simultaneous requests can lead to "Race Conditions" where the rate limiter counts incorrectly. If two requests read the counter at `9` at the exact same millisecond, both might increment to `10` and pass, breaking the strict limit.
- **The Solution:** I utilized **Redis Atomic Operations** (`INCR` and `EXPIRE`). Since Redis operates on a single-threaded event loop, it processes these operations sequentially. This guarantees 100% accurate counting and eliminates race conditions, even under high concurrency.