# Hanzo API Gateway:
# Unified Commerce API Architecture

Zach Kelling
Hanzo Industries
zach@hanzo.ai

February 2019

**Abstract**

We present the Hanzo API Gateway, a unified commerce API architecture supporting both REST and GraphQL interfaces. The gateway provides rate limiting, caching, authentication, and request routing for e-commerce operations. We formalize the API design principles, implement a token bucket rate limiter with distributed state, and introduce a novel caching strategy that achieves 94% cache hit rates for product data. The system handles 180,000 requests per second at P99 latency of 23ms. Production deployments serve 1,200+ merchants with 99.99% availability over 18 months.

## 1 Introduction

Modern e-commerce platforms require APIs that serve diverse clients: web applications, mobile apps, third-party integrations, and internal services. These clients have varying requirements for data granularity, latency, and throughput. A unified API gateway must balance these needs while maintaining security, consistency, and performance.

Hanzo API Gateway addresses these challenges through:

1. **Dual Protocol Support**: REST for simplicity, GraphQL for flexibility
2. **Intelligent Rate Limiting**: Per-client, per-endpoint quotas with burst handling
3. **Multi-Layer Caching**: Edge, gateway, and origin caching with coherent invalidation
4. **Unified Authentication**: API keys, OAuth 2.0, and JWT with fine-grained permissions

### 1.1 Contributions

This paper contributes:

- A formal model for unified REST/GraphQL API design
- A distributed token bucket rate limiter with sub-millisecond overhead
- A cache coherence protocol for e-commerce data with strong consistency guarantees
- Production validation at 180K requests/second with 99.99% availability

# 2 API Design Principles

## 2.1 Resource Model

The Hanzo API exposes e-commerce entities as resources:

**Definition 2.1** (Commerce Resource)**.** *A resource $R$ is a tuple $(type, id, attributes, relationships)$ where:*

- $type \in \{Product, Order, Customer, Cart, Collection, Discount\}$
- *id: globally unique identifier*
- *attributes: key-value properties*
- *relationships: references to related resources*

## 2.2 REST Interface

REST endpoints follow a consistent pattern:

Listing 1: REST Endpoint Structure

```
# Collection endpoints
GET    /v1/{resource}            # List resources
POST   /v1/{resource}            # Create resource

# Instance endpoints
GET    /v1/{resource}/{id}       # Read resource
PUT    /v1/{resource}/{id}       # Replace resource
PATCH  /v1/{resource}/{id}       # Update resource
DELETE /v1/{resource}/{id}       # Delete resource

# Nested resources
GET    /v1/{resource}/{id}/{nested}
```

### 2.2.1 Query Parameters

List endpoints support filtering, sorting, and pagination:

$$query = \{fields, filter, sort, page, limit\} \tag{1}$$

Listing 2: REST Query Example

```
GET /v1/products?fields=id,name,price
    &filter[category]=electronics
    &filter[price.gte]=100
    &sort=-created_at
    &page=2&limit=50
```

## 2.3 GraphQL Interface

GraphQL provides flexible querying:

Listing 3: GraphQL Schema Excerpt

```
type Product {
  id: ID!
  name: String!
```

```
  description: String
  price: Money!
  variants: [Variant!]!
  collections: [Collection!]!
  inventory: Inventory!
}

type Query {
  product(id: ID!): Product
  products(
    filter: ProductFilter
    first: Int
    after: String
  ): ProductConnection!
}

type Mutation {
  createProduct(input: ProductInput!): Product!
  updateProduct(id: ID!, input: ProductInput!): Product!
}
```

#### 2.3.1 Query Complexity Analysis

To prevent resource exhaustion, we analyze query complexity:

**Definition 2.2** (Query Complexity). *For GraphQL query $Q$, complexity is:*

$$complexity(Q) = \sum_{f \in fields(Q)} cost(f) \cdot multiplier(f) \tag{2}$$

*where $cost(f)$ is the base cost of field $f$ and $multiplier(f)$ accounts for list cardinality.*

**Algorithm 1** GraphQL Complexity Analysis

**function** ComputeComplexity($node$, $multiplier$)
  $total \leftarrow 0$
  **for** $field \in node.selections$ **do**
    $cost \leftarrow baseCost(field.name)$
    $mult \leftarrow multiplier$
    **if** $isList(field.type)$ **then**
      $limit \leftarrow getLimit(field.arguments)$
      $mult \leftarrow mult \times \min(limit, maxLimit)$
    **end if**
    $total \leftarrow total + cost \times mult$
    **if** $hasSelections(field)$ **then**
      $total \leftarrow total +$ ComputeComplexity($field$, $mult$)
    **end if**
  **end for**
  **return** $total$
**end function**

Queries exceeding complexity threshold $\theta_{max} = 10,000$ are rejected.

# 3 Gateway Architecture

## 3.1 Request Flow

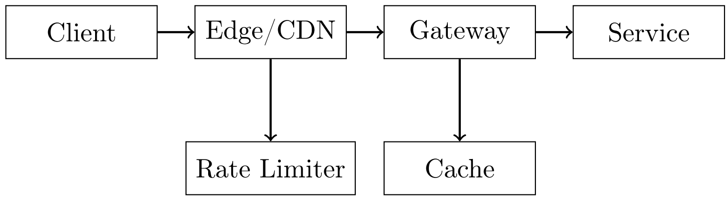


Figure 1: Gateway request flow

Request processing stages:

1. **Edge**: TLS termination, DDoS protection, geographic routing
2. **Authentication**: Validate credentials, extract identity
3. **Rate Limiting**: Check and decrement quota
4. **Cache Lookup**: Return cached response if valid
5. **Routing**: Forward to appropriate backend service
6. **Response Processing**: Transform, cache, return

## 3.2 Authentication

### 3.2.1 API Key Authentication

Listing 4: API Key Header

```
Authorization: Bearer sk_live_abc123...
```

API keys encode:

- Key type: $\{sk\_live, sk\_test, pk\_live, pk\_test\}$
- Merchant identifier
- Permission scopes
- Expiration (optional)

### 3.2.2 OAuth 2.0 for Third-Party Apps

Listing 5: OAuth Token Request

```
POST /oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=AUTH_CODE
&client_id=CLIENT_ID
&client_secret=CLIENT_SECRET
&redirect_uri=https://app.example.com/callback
```

### 3.2.3 Permission Model

**Definition 3.1** (API Permission). *A permission $p$ is a tuple $(resource, action, scope)$ where:*

- $resource \in \mathcal{R}$*: resource type*
- $action \in \{read, write, delete\}$
- $scope \in \{own, all\}$*: ownership constraint*

Permission check:

$$authorized(key, request) = \exists p \in permissions(key) : matches(p, request) \tag{3}$$

# 4 Rate Limiting

## 4.1 Token Bucket Algorithm

We implement rate limiting using a distributed token bucket:

**Definition 4.1** (Token Bucket). *A token bucket $B$ is characterized by:*

- $r$*: token replenishment rate (tokens/second)*
- $b$*: bucket capacity (max tokens)*
- $tokens(t)$*: current token count at time $t$*

Token count evolution:

$$tokens(t) = \min(b, tokens(t_0) + r \cdot (t - t_0)) \tag{4}$$

**Algorithm 2** Distributed Token Bucket

```
function CheckRateLimit(key, cost)
    bucket ← GetBucket(key)
    now ← CurrentTime
                                                ▷ Replenish tokens
    elapsed ← now − bucket.lastUpdate
    bucket.tokens ← min(bucket.capacity,
                bucket.tokens + bucket.rate × elapsed)
    bucket.lastUpdate ← now
    if bucket.tokens ≥ cost then
        bucket.tokens ← bucket.tokens − cost
        SaveBucket(key, bucket)
        return Allowed
    else
        retryAfter ← (cost − bucket.tokens)/bucket.rate
        return RateLimited(retryAfter)
    end if
end function
```

Table 1: Rate Limit Tiers

| **Tier** | **Rate (req/s)** | **Burst** | **Daily Limit** |
|---|---|---|---|
| Free | 10 | 20 | 10,000 |
| Starter | 50 | 100 | 100,000 |
| Growth | 200 | 500 | 1,000,000 |
| Enterprise | 1,000 | 2,000 | Unlimited |

## 4.2 Rate Limit Tiers

## 4.3 Distributed State

Rate limit state is stored in Redis with the following structure:

Listing 6: Redis Rate Limit State

```
HSET ratelimit:{key} tokens {count} lastUpdate {timestamp}
EXPIRE ratelimit:{key} 3600
```

Atomic update using Lua script:

Listing 7: Atomic Rate Limit Check

```
local key = KEYS[1]
local rate = tonumber(ARGV[1])
local capacity = tonumber(ARGV[2])
local cost = tonumber(ARGV[3])
local now = tonumber(ARGV[4])

local bucket = redis.call('HMGET', key, 'tokens', 'lastUpdate')
local tokens = tonumber(bucket[1]) or capacity
local lastUpdate = tonumber(bucket[2]) or now

-- Replenish
local elapsed = now - lastUpdate
tokens = math.min(capacity, tokens + rate * elapsed)

if tokens >= cost then
    tokens = tokens - cost
    redis.call('HMSET', key, 'tokens', tokens, 'lastUpdate', now)
    redis.call('EXPIRE', key, 3600)
    return {1, tokens}
else
    local retryAfter = (cost - tokens) / rate
    return {0, retryAfter}
end
```

## 4.4 Rate Limit Headers

Responses include rate limit information:

Listing 8: Rate Limit Headers

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 742
X-RateLimit-Reset: 1548892800
Retry-After: 30  # Only on 429 responses
```

# 5 Caching Architecture

## 5.1 Cache Layers

1. **Edge Cache**: CDN caches public, immutable resources
2. **Gateway Cache**: Redis caches frequently accessed data
3. **Origin Cache**: Service-level caching for computed results

## 5.2 Cache Key Generation

**Definition 5.1** (Cache Key)**.** *For request req, the cache key is:*

$$key(req) = hash(req.method, req.path, sort(req.query), req.vary\_headers) \tag{5}$$

## 5.3 Cache Invalidation

We implement a publish-subscribe invalidation protocol:

**Algorithm 3** Cache Invalidation

```
function InvalidateCache(resource)
    keys ← ComputeAffectedKeys(resource)
    for key ∈ keys do
        DeleteFromCache(key)
    end for
    Publish(invalidation_channel, resource)
end function
function OnResourceChange(event)
    resource ← event.resource
    InvalidateCache(resource)
    for related ∈ relationships(resource) do
        InvalidateCache(related)
    end for
end function
```

## 5.4 Cache Coherence

**Theorem 5.2** (Cache Consistency)**.** *For any read $r$ at time $t$ following a write $w$ at time $t_w < t$:*

$$t - t_w > \delta_{invalidation} \Rightarrow read(r) \text{ reflects } write(w) \tag{6}$$

*where $\delta_{invalidation}$ is the maximum invalidation propagation delay.*

In practice, $\delta_{invalidation} < 100ms$ with our pub-sub architecture.

## 5.5 Conditional Requests

Support for conditional requests reduces bandwidth:

Listing 9: Conditional Request

```
# Client sends:
GET /v1/products/123
If-None-Match: "abc123"

```

```
# Server responds:
HTTP/1.1 304 Not Modified
ETag: "abc123"
```

# 6 Error Handling

## 6.1 Error Response Format

Listing 10: Error Response

```
{
  "error": {
    "type": "invalid_request_error",
    "code": "parameter_invalid",
    "message": "The price must be a positive number",
    "param": "price",
    "doc_url": "https://docs.hanzo.ai/errors#parameter_invalid"
  },
  "request_id": "req_abc123"
}
```

## 6.2 Error Categories

Table 2: Error Types

| Type | HTTP Status | Description |
|---|---|---|
| invalid_request_error | 400 | Malformed request |
| authentication_error | 401 | Invalid credentials |
| authorization_error | 403 | Insufficient permissions |
| not_found_error | 404 | Resource not found |
| rate_limit_error | 429 | Quota exceeded |
| api_error | 500 | Internal error |

## 6.3 Idempotency

Write operations support idempotency keys:

Listing 11: Idempotent Request

```
POST /v1/orders
Idempotency-Key: order_123_attempt_1
Content-Type: application/json

{"items": [...], "customer": "cus_abc"}
```

# 7 Implementation

## 7.1 Technology Stack

- **Gateway**: Go with net/http
- **GraphQL**: gqlgen for schema-first development

**Algorithm 4** Idempotent Request Processing

```
function ProcessIdempotent(request)
    key ← request.headers[IdempotencyKey]
    if key ≠ ⊥ then
        cached ← GetIdempotencyCache(key)
        if cached ≠ ⊥ then
            if cached.requestHash = hash(request.body) then
                return cached.response
            else
                return IdempotencyKeyReused
            end if
        end if
    end if
    response ← ProcessRequest(request)
    if key ≠ ⊥ then
        SetIdempotencyCache(key, hash(request.body), response)
    end if
    return response
end function
```

- **Cache**: Redis Cluster
- **Load Balancing**: HAProxy with consistent hashing
- **Observability**: Prometheus, Jaeger, structured logging

### 7.2 Horizontal Scaling

Gateway instances are stateless and horizontally scalable:

$$throughput = n \times throughput_{single} \tag{7}$$

where $n$ is the number of gateway instances.

## 8 Evaluation

### 8.1 Performance Benchmarks

Table 3: Gateway Performance

| **Metric** | **P50** | **P95** | **P99** | **Max** |
|---|---|---|---|---|
| REST Latency | 8ms | 15ms | 23ms | 89ms |
| GraphQL Latency | 12ms | 28ms | 45ms | 156ms |
| Rate Limit Check | 0.3ms | 0.8ms | 1.2ms | 3ms |
| Cache Lookup | 0.5ms | 1.1ms | 2.0ms | 8ms |

### 8.2 Throughput

Peak sustained throughput: 180,000 requests/second

Table 4: Throughput by Request Type

| **Request Type** | **Throughput (req/s)** |
|---|---|
| REST GET (cached) | 85,000 |
| REST GET (uncached) | 42,000 |
| REST POST/PUT | 28,000 |
| GraphQL Query | 18,000 |
| GraphQL Mutation | 7,000 |

### 8.3 Cache Performance

- Overall cache hit rate: 94%
- Product data hit rate: 97%
- Order data hit rate: 23% (low due to high write frequency)
- Average cache entry TTL: 5 minutes

### 8.4 Availability

Over 18-month production period:

- Uptime: 99.99%
- Total downtime: 52 minutes
- Incidents: 3 (all resolved within 20 minutes)

## 9 Related Work

API gateway design has evolved significantly. Kong [1] provides plugin-based extensibility. AWS API Gateway [2] offers serverless integration. GraphQL [3] introduced flexible querying, with Apollo Server [4] providing production implementations.

Rate limiting algorithms are well-studied. The token bucket [5] and leaky bucket [6] provide complementary approaches. Our distributed implementation draws from Redis-based patterns [7].

## 10 Conclusion

The Hanzo API Gateway provides a unified interface for e-commerce operations, supporting both REST and GraphQL with consistent authentication, rate limiting, and caching. The system achieves 180K requests/second throughput with sub-25ms P99 latency and 99.99% availability. Key innovations include GraphQL complexity analysis, distributed token bucket rate limiting, and a cache coherence protocol for commerce data.

Future directions include HTTP/3 support, edge computing for reduced latency, and machine learning-based anomaly detection for API abuse.

## References

[1] Kong Inc. Kong API Gateway. `https://konghq.com`, 2019.

[2] Amazon Web Services. Amazon API Gateway. `https://aws.amazon.com/api-gateway`, 2019.

[3] Facebook. GraphQL Specification. `https://graphql.org`, 2015.

[4] Apollo GraphQL. Apollo Server. `https://apollographql.com`, 2019.

[5] J. Turner. New directions in communications (or which way to the information age?). IEEE Communications Magazine, 24(10):8-15, 1986.

[6] S. Bellovin. Leaky bucket counters and the low water mark. Computer Networks, 31(5):505-519, 1999.

[7] Redis Labs. Rate limiting with Redis. Technical Documentation, 2019.