Distributed Rate Limiter: Design Document

Overview

A distributed rate limiting system that enforces request quotas across multiple application instances using Redis as a shared state store. The key challenge is maintaining accurate rate limits in a distributed environment while handling Redis failures, clock skew, and achieving high performance through sharding.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** All milestones - this foundational understanding applies to rate limiting algorithms, multi-tier systems, Redis integration, sharding, and API design.

Mental Model: The Nightclub Bouncer System

Think of distributed rate limiting like managing capacity across a chain of popular nightclubs during New Year's Eve. Each nightclub has a fire safety capacity limit, but customers don't just visit one location - they might hop between venues throughout the night. The challenge is ensuring that the combined crowd across all locations doesn't exceed what the fire department allows, while also managing individual venue limits and VIP customer quotas.

In a traditional single-venue scenario, you'd have one bouncer with a mechanical clicker counter at the door. Every person entering gets counted up, every person leaving gets counted down. The bouncer always knows exactly how many people are inside and can make instant decisions. This represents **local rate limiting** - one application instance tracking its own request counts in memory.

But when you expand to multiple venues (distributed application instances), the problem becomes exponentially more complex. Now you need to coordinate between multiple bouncers across the city. If your fire department limit is 1000 people total across all venues, and Bouncer Alice at Club Downtown has seen 400 people enter while Bouncer Bob at Club Uptown has seen 450 people, what happens when someone approaches Bouncer Charlie at Club Westside? Charlie needs to know the current total (850) before deciding whether to allow the 851st person in.

The naive approach would be to have each bouncer call every other bouncer before making a decision: "Hey Alice, how many people do you have? Hey Bob, what's your count?" This creates several problems. First, it's slow - every admission decision requires multiple phone calls. Second, it's fragile - if Alice doesn't answer her phone, does that mean Charlie should assume zero people at Downtown? Third, it creates race conditions - while Charlie is making his phone calls, Alice might admit 50 more people, making Charlie's decision based on stale data.

The **distributed rate limiting solution** introduces a **central coordination system** (Redis) that acts like a real-time dispatch center. Instead of bouncers calling each other, they all report to and check with dispatch. When someone wants to enter Charlie's venue, Charlie radios dispatch: "Request to admit one person." Dispatch has the real-time total across all venues and responds: "Approved, new total is 851" or "Denied, at capacity." This provides several critical capabilities:

Atomic Decision Making: Dispatch can check the current count and increment it in a single atomic operation, preventing the race condition where two bouncers simultaneously think they're admitting the 1000th person when they're actually admitting the 999th and 1000th.

Global State Visibility: Every bouncer gets decisions based on the true global state, not stale local information or incomplete peer-to-peer communication.

Hierarchical Limits: Dispatch can enforce multiple types of limits simultaneously. Maybe the fire department allows 1000 total people, but Club Downtown has a structural limit of 300, and VIP customers get guaranteed access to 50 spots regardless of the general admission count.

Resilience Through Fallback: If the radio system goes down, bouncers can fall back to local-only decisions. They might occasionally exceed the global limit during the outage, but the venues remain operational rather than shutting down entirely.

However, this coordination comes with new challenges that don't exist in the single-venue scenario:

Network Latency: Radio calls to dispatch take time. In high-traffic periods, the delay between "request to admit" and "approved/denied" could cause customer frustration.

Central Point of Failure: If dispatch goes offline, all venues are affected. The system needs fallback strategies and redundancy.

Clock Synchronization: If Club Downtown's clocks are 5 minutes fast, their "hourly" limits might reset at different times than Club Uptown's, creating windows where the global limit can be exceeded.

Hot Spot Management: If a celebrity shows up at Club Downtown and generates massive traffic, that venue might overwhelm dispatch with admission requests, slowing down decisions for all other venues.

This nightclub analogy maps directly to distributed rate limiting concepts:

- **Venues** = Application instances
- **Bouncers** = Rate limiter components within each instance
- **Customers** = Incoming HTTP requests
- **Fire department capacity** = Global rate limits
- **Venue structural limits** = Per-instance or per-API limits
- **VIP quotas** = Per-user rate limits
- **Dispatch center** = Redis cluster
- **Radio communication** = Network calls to Redis
- **Mechanical clickers** = In-memory counters for local fallback
- **Clock synchronization** = Time-based window alignment across nodes

Existing Approaches Comparison

Understanding the landscape of rate limiting approaches helps illuminate why distributed rate limiting requires careful design decisions. Each approach represents different trade-offs between accuracy, performance, complexity, and resilience.

Local vs Distributed Rate Limiting

Aspect	Local Rate Limiting	Distributed Rate Limiting
State Storage	In-memory within each application instance	Shared external store (Redis, database)
Decision Latency	Microseconds (memory access)	Milliseconds (network + storage access)
Accuracy	Perfect for single-instance traffic	Perfect for cluster-wide traffic
Failure Behavior	Independent failures per instance	Coordinated failures across cluster
Implementation Complexity	Simple hash maps and timers	Atomic operations, consensus, fallback logic
Resource Usage	Minimal CPU and memory overhead	Network bandwidth, Redis memory, connection pools
Scaling Characteristics	Limits scale with instance count	Limits remain constant regardless of instance count

Local rate limiting excels in scenarios where each application instance handles completely independent traffic or where approximate limiting is acceptable. For example, if you have 10 application instances and want to limit each user to 100 requests per hour, local limiting would give each user 1000 requests per hour cluster-wide (100 per instance). This might be acceptable for rough abuse prevention but fails for precise quota enforcement or protecting downstream services with hard capacity limits.

Distributed rate limiting becomes essential when you need precise control over cluster-wide request rates. Consider a payment processing API where the downstream banking service can only handle 1000 transactions per minute total. With local limiting, you'd need to divide this quota across instances (100 per instance if you have 10 instances), but this creates problems: if traffic is unevenly distributed, some instances might exhaust their quota while others remain idle, leading to artificial throttling despite available global capacity.

> **Key Insight**: The choice between local and distributed rate limiting fundamentally depends on whether your limiting goal is per-instance abuse prevention (local) or cluster-wide resource protection (distributed).

In-Memory vs Persistent Storage

Aspect	In-Memory Storage	Persistent Storage (Redis)
Performance	Fastest (no I/O)	Fast (network + memory access)
Durability	Lost on process restart	Survives restarts and failures
Sharing	Cannot share between instances	Natural sharing across cluster
Memory Usage	Grows with active rate limit keys	Centralized memory usage
Consistency	Eventually consistent across instances	Strongly consistent with atomic operations
Operational Complexity	No external dependencies	Requires Redis cluster management

In-memory storage using hash maps or similar structures provides the lowest latency for rate limiting decisions. Popular libraries like Google's `golang.org/x/time/rate` or Java's Guava RateLimiter implement sophisticated algorithms entirely in memory. However, this approach has fundamental limitations in distributed systems:

- **State Isolation**: Each instance maintains separate counters, making precise global limits impossible
- **Cold Start Problems**: New instances start with empty rate limit state, potentially allowing bursts that exceed intended limits
- **Restart Penalties**: Process restarts reset all counters, effectively giving users fresh quota allocations

Persistent storage through Redis or similar systems enables true distributed coordination but introduces new complexities:

- **Network Partitions**: What happens when an application instance can reach Redis but other instances cannot?
- **Redis Failures**: How do you maintain availability when the coordination layer fails?
- **Data Consistency**: Ensuring that concurrent updates from multiple instances don't corrupt rate limit counters

> **Decision: Redis as Coordination Layer**
>
> - **Context**: Need for atomic operations, high availability, and performance in distributed rate limiting
> - **Options Considered**:
>   1. Database-backed counters with transactions
>   2. Redis with Lua scripts for atomicity
>   3. Distributed consensus systems (etcd, Consul)
> - **Decision**: Redis with Lua scripts
> - **Rationale**: Redis provides microsecond-latency atomic operations through Lua scripts, built-in expiration for time-window management, and mature high-availability clustering. Database transactions add unnecessary overhead for simple counter operations, while consensus systems optimize for different use cases (configuration management) rather than high-throughput counting.
> - **Consequences**: Enables precise distributed rate limiting with excellent performance, but requires Redis operational expertise and introduces dependency on Redis availability.

Algorithm Complexity vs Accuracy Trade-offs

Different rate limiting algorithms make different trade-offs between implementation complexity, memory usage, and limiting accuracy:

Algorithm	Memory Usage	Accuracy	Burst Handling	Implementation Complexity
Fixed Window Counter	O(1) per key	Poor (2x limit possible)	Allows full limit per window	Simple
Sliding Window Log	O(n) per key (n=requests)	Perfect	Precise burst control	Moderate
Sliding Window Counter	O(k) per key (k=sub-windows)	Good (configurable accuracy)	Smooth burst handling	Moderate
Token Bucket	O(1) per key	Good (allows configured burst)	Explicit burst capacity	Simple
Leaky Bucket	O(n) per key (n=queued requests)	Perfect (no bursts)	No burst allowance	Complex

Fixed Window Counter represents the simplest approach: reset a counter every time period (e.g., every minute). This creates the "boundary problem" where a user could make 1000 requests in the last second of one minute and another 1000 requests in the first second of the next minute, effectively achieving 2000 requests per minute despite a 1000/minute limit.

Sliding Window Log maintains a timestamp for every request within the current window. This provides perfect accuracy but consumes memory proportional to the request rate, making it expensive for high-traffic scenarios.

Token Bucket offers a middle ground by modeling rate limits as a bucket that starts full of tokens, loses tokens with each request, and refills at a steady rate. This naturally handles bursts (empty the bucket quickly) while enforcing long-term rate limits (bucket refill rate).

> **Architecture Decision**: The distributed rate limiter implements multiple algorithms to handle different use cases. Token bucket for APIs that benefit from burst allowance, sliding window counter for smooth rate enforcement, and sliding window log for scenarios requiring perfect accuracy despite higher memory costs.

Fallback Strategy Comparison

When the coordination layer (Redis) becomes unavailable, different fallback strategies offer different trade-offs:

Fallback Strategy	Availability	Accuracy During Outage	Recovery Behavior
Fail Open	100% (no limiting)	0% (unlimited requests)	Immediate return to normal
Fail Closed	0% (reject all)	100% (no limit violations)	Immediate return to normal
Local Fallback	High (degraded limiting)	Poor (per-instance limits)	Gradual convergence
Circuit Breaker	Variable (configurable)	Variable (configurable)	Staged recovery testing

Fail Open prioritizes availability by allowing all requests through during Redis outages. This suits scenarios where brief periods of unlimited access are preferable to service disruption, such as content delivery or social media APIs.

Fail Closed prioritizes protection by rejecting requests during outages. This suits scenarios where exceeding limits could cause cascading failures, such as payment processing or database write APIs.

Local Fallback attempts to maintain degraded rate limiting by switching to per-instance limits during outages. Each instance divides the global limit by the number of instances and enforces this reduced limit locally. This provides some protection while maintaining availability, but can lead to under-utilization if traffic is unevenly distributed.

Circuit Breaker Pattern implements intelligent failure detection and recovery testing. Rather than immediately resuming full Redis usage when connectivity returns, the circuit breaker gradually increases traffic to Redis while monitoring for continued failures.

> **Decision: Graceful Degradation with Local Fallback**
>
> - **Context**: Need to balance availability with protection during Redis outages
> - **Options Considered**: Fail open, fail closed, local fallback with circuit breaker
> - **Decision**: Local fallback with gradual recovery
> - **Rationale**: Maintains both availability and some level of protection during outages. Circuit breaker prevents thundering herd problems when Redis recovers. Local fallback provides predictable behavior that operations teams can reason about.
> - **Consequences**: Enables continued operation during Redis outages with degraded but predictable rate limiting. Requires careful configuration of per-instance quotas and circuit breaker thresholds.

Common Pitfalls in Distributed Rate Limiting

⚠️ Pitfall: Assuming Network Calls Are Instantaneous

Many developers initially design distributed rate limiters as if Redis calls have zero latency. They write code that makes synchronous Redis calls in the request path without considering timeout handling, connection pooling, or retry logic. Under load, this creates cascading delays where rate limiting decisions become the bottleneck rather than the protection mechanism.

The fix requires treating every Redis interaction as a potentially slow network operation with explicit timeouts, connection reuse, and circuit breaker patterns. Rate limiting checks should typically complete within 1-2 milliseconds; anything slower suggests architectural problems.

⚠️ Pitfall: Ignoring Clock Skew Between Instances

Time-based rate limiting algorithms assume synchronized clocks across all application instances. In practice, server clocks can drift by seconds or minutes, causing time window boundaries to misalign. This creates windows where users can exceed limits by making requests to instances with fast clocks just before window boundaries and slow clocks just after boundaries.

The solution involves using Redis server time for all time-based calculations rather than application instance time, or implementing clock synchronization monitoring with alerts when drift exceeds acceptable thresholds.

⚠️ Pitfall: Not Handling Redis Memory Pressure

Redis operates as an in-memory database, and rate limiting can generate enormous numbers of keys (user IDs, IP addresses, API endpoints combined with time windows). Without proper key expiration and memory management, Redis can run out of memory, causing either data eviction or service failures.

The fix requires careful key naming with TTL management, monitoring Redis memory usage, and implementing key cleanup strategies for inactive rate limit entries. Every rate limit key should have an explicit expiration time, typically 2x the rate limit window duration.

Implementation Guidance

This section provides concrete technology recommendations and starter code for building the distributed rate limiter foundation.

Technology Recommendations

Component	Simple Option	Advanced Option
Redis Client	`go-redis/redis/v9` (Redis client)	`go-redis/redis/v9` with cluster support
HTTP Framework	`net/http` with middleware	`gin-gonic/gin` or `gorilla/mux`
Configuration	Environment variables + `os.Getenv`	`viper` configuration management
Metrics	`expvar` for basic metrics	`prometheus/client_golang`
Logging	`log/slog` (Go 1.21+)	`uber-go/zap` for structured logging
Testing	`testing` + `testcontainers` for Redis	`stretchr/testify` + test containers

Recommended Project Structure

```
distributed-rate-limiter/
├── cmd/
│   ├── server/main.go              ← HTTP server with rate limiting middleware
│   └── dashboard/main.go           ← Management dashboard server
├── internal/
│   ├── ratelimit/                  ← Core rate limiting logic
│   │   ├── limiter.go              ← Main rate limiter interface
│   │   ├── algorithms/             ← Rate limiting algorithm implementations
│   │   │   ├── token_bucket.go     ← Token bucket algorithm
│   │   │   ├── sliding_window.go   ← Sliding window algorithms
│   │   │   └── algorithm.go        ← Common algorithm interface
│   │   ├── storage/                ← Storage backends
│   │   │   ├── redis.go            ← Redis backend implementation
│   │   │   ├── local.go            ← Local fallback storage
│   │   │   └── storage.go          ← Storage interface
│   │   └── config/                 ← Configuration and rule management
│   │       ├── rules.go            ← Rate limit rule definitions
│   │       └── manager.go          ← Dynamic configuration management
│   ├── api/                        ← REST API for management
│   │   ├── handlers.go             ← HTTP handlers for CRUD operations
│   │   └── middleware.go           ← Rate limiting middleware
│   ├── dashboard/                  ← Real-time dashboard components
│   │   ├── websocket.go            ← WebSocket handlers for real-time updates
│   │   └── metrics.go              ← Metrics collection and aggregation
│   └── sharding/                   ← Consistent hashing and sharding
│       ├── consistent_hash.go      ← Consistent hash ring implementation
│       └── node_manager.go         ← Redis node health and management
├── pkg/                            ← Public interfaces (if needed)
├── configs/                        ← Configuration files
│   └── rate_limits.yaml            ← Default rate limit rules
├── scripts/                        ← Deployment and utility scripts
│   ├── redis_setup.sh              ← Redis cluster setup script
│   └── load_test.sh                ← Load testing script
├── docs/                           ← Additional documentation
└── docker-compose.yml              ← Local development environment
```

Infrastructure Starter Code

Redis Connection Manager (`internal/ratelimit/storage/redis.go`):

```go
package storage

import (
    "context"
    "time"
    "github.com/redis/go-redis/v9"
)

// RedisConfig holds Redis connection configuration
type RedisConfig struct {
    Addresses    []string      `json:"addresses"`
    Password     string        `json:"password"`
    DB           int           `json:"db"`
    PoolSize     int           `json:"pool_size"`
    ReadTimeout  time.Duration `json:"read_timeout"`
    WriteTimeout time.Duration `json:"write_timeout"`
    DialTimeout  time.Duration `json:"dial_timeout"`
}

// RedisStorage implements the Storage interface using Redis
type RedisStorage struct {
    client redis.UniversalClient
    config RedisConfig
}

// NewRedisStorage creates a new Redis storage backend with connection pooling
func NewRedisStorage(config RedisConfig) (*RedisStorage, error) {
    // TODO: Create Redis universal client (handles both single instance and cluster)
    // TODO: Configure connection pool with proper timeouts
    // TODO: Test connectivity with ping command
    // TODO: Return configured storage instance
}

// CheckAndUpdate atomically checks current count and updates if limit allows
func (r *RedisStorage) CheckAndUpdate(ctx context.Context, key string, limit int64, window time.Duration) (allowed bool,
remaining int64, resetTime time.Time, err error) {
    // TODO: This will be implemented with Lua scripts in Redis Backend Integration section
    // For now, return placeholder values for basic connectivity testing
    return true, limit-1, time.Now().Add(window), nil
}
```

Rate Limit Rule Configuration (`internal/ratelimit/config/rules.go`):

```go
package config

import (
    "time"
)

// RateLimitRule defines a single rate limiting rule
type RateLimitRule struct {
    ID          string        `json:"id" yaml:"id"`
    Name        string        `json:"name" yaml:"name"`
    KeyPattern  string        `json:"key_pattern" yaml:"key_pattern"`   // e.g., "user:{user_id}", "ip:{ip_address}"
    Algorithm   string        `json:"algorithm" yaml:"algorithm"`        // "token_bucket", "sliding_window_counter", etc.
    Limit       int64         `json:"limit" yaml:"limit"`               // Number of requests allowed
    Window      time.Duration `json:"window" yaml:"window"`             // Time window for the limit
    BurstLimit  int64         `json:"burst_limit,omitempty" yaml:"burst_limit,omitempty"` // For token bucket
    Enabled     bool          `json:"enabled" yaml:"enabled"`
    Priority    int           `json:"priority" yaml:"priority"`         // Higher priority rules checked first
    CreatedAt   time.Time     `json:"created_at" yaml:"created_at"`
    UpdatedAt   time.Time     `json:"updated_at" yaml:"updated_at"`
}

// RuleManager handles loading and updating rate limit rules
type RuleManager struct {
    rules map[string]*RateLimitRule
    // TODO: Add mutex for concurrent access
    // TODO: Add file watcher for dynamic updates
    // TODO: Add Redis pub/sub for distributed rule updates
}

// LoadRules loads rate limit rules from configuration file
func (rm *RuleManager) LoadRules(configPath string) error {
    // TODO: Read YAML configuration file
    // TODO: Parse rules and validate configuration
    // TODO: Store rules in memory with indexing by key pattern
    // TODO: Set up file watcher for automatic reloading
}

// GetMatchingRules returns all rules that match the given request context
func (rm *RuleManager) GetMatchingRules(userID, ipAddress, apiEndpoint string) []*RateLimitRule {
    // TODO: Match request attributes against key patterns
```

```
    // TODO: Return rules sorted by priority (highest first)

    // TODO: Handle wildcard patterns and parameter substitution

}
```

Basic Rate Limiter Interface (`internal/ratelimit/limiter.go`):

```go
package ratelimit

import (
    "context"
    "time"
)

// RateLimitResult contains the result of a rate limit check
type RateLimitResult struct {
    Allowed     bool          `json:"allowed"`
    Remaining   int64         `json:"remaining"`
    RetryAfter  time.Duration `json:"retry_after,omitempty"`
    ResetTime   time.Time     `json:"reset_time"`
    RuleID      string        `json:"rule_id"`
    Algorithm   string        `json:"algorithm"`
}

// RateLimitRequest contains parameters for a rate limit check
type RateLimitRequest struct {
    UserID      string `json:"user_id,omitempty"`
    IPAddress   string `json:"ip_address,omitempty"`
    APIEndpoint string `json:"api_endpoint,omitempty"`
    UserAgent   string `json:"user_agent,omitempty"`
    Tokens      int64  `json:"tokens,omitempty"` // Number of tokens to consume (default: 1)
}

// Limiter is the main interface for rate limiting operations
type Limiter interface {
    // Check performs a rate limit check and updates counters if allowed
    Check(ctx context.Context, req RateLimitRequest) (*RateLimitResult, error)

    // Preview checks rate limit status without updating counters
    Preview(ctx context.Context, req RateLimitRequest) (*RateLimitResult, error)

    // Reset clears rate limit counters for the given request pattern
    Reset(ctx context.Context, req RateLimitRequest) error
}

// DistributedLimiter implements the Limiter interface with Redis backend
type DistributedLimiter struct {
```

```
    storage     Storage
    ruleManager *config.RuleManager
    localFallback Limiter // Used when Redis is unavailable
}

// NewDistributedLimiter creates a new distributed rate limiter
func NewDistributedLimiter(storage Storage, ruleManager *config.RuleManager) *DistributedLimiter {
    return &DistributedLimiter{
        storage:     storage,
        ruleManager: ruleManager,
        // TODO: Initialize local fallback limiter
    }
}

// Check performs distributed rate limiting with multi-tier evaluation
func (dl *DistributedLimiter) Check(ctx context.Context, req RateLimitRequest) (*RateLimitResult, error) {
    // TODO: Get matching rules from rule manager
    // TODO: Evaluate rules in priority order with short-circuit logic
    // TODO: For each rule, generate Redis key and check limit
    // TODO: Return first rule that denies the request, or allow if all pass
    // TODO: Handle Redis failures with local fallback
}
```

Core Algorithm Skeleton

Token Bucket Algorithm (`internal/ratelimit/algorithms/token_bucket.go`):

```go
package algorithms

import (
    "context"
    "time"
)

// TokenBucketConfig defines parameters for token bucket algorithm
type TokenBucketConfig struct {
    Capacity   int64         `json:"capacity"`    // Maximum tokens in bucket
    RefillRate int64         `json:"refill_rate"` // Tokens added per second
    Window     time.Duration `json:"window"`      // Window for rate calculation
}

// TokenBucketState represents current state stored in Redis
type TokenBucketState struct {
    Tokens        int64     `json:"tokens"`
    LastRefillTime int64    `json:"last_refill_time"` // Unix nanoseconds
}

// TokenBucket implements token bucket rate limiting algorithm
type TokenBucket struct {
    config TokenBucketConfig
    storage Storage
}

// CheckAndUpdate performs atomic check-and-update for token bucket algorithm
func (tb *TokenBucket) CheckAndUpdate(ctx context.Context, key string, tokensRequested int64) (allowed bool, remaining int64,
resetTime time.Time, err error) {
    // TODO 1: Calculate current time in nanoseconds for precision
    // TODO 2: Execute Lua script in Redis for atomic check-and-update:
    //   - Get current bucket state (tokens, last_refill_time)
    //   - Calculate tokens to add based on time elapsed and refill rate
    //   - Add tokens to bucket, capping at capacity
    //   - If sufficient tokens available, subtract requested tokens and allow
    //   - If insufficient tokens, deny and calculate retry-after time
    //   - Update bucket state with new token count and refill time
    //   - Return result with remaining tokens and reset time
    // TODO 3: Handle Redis errors with appropriate fallback strategy
    // TODO 4: Parse Lua script response and construct result object
```

```
    // Placeholder implementation for initial testing
    return true, tb.config.Capacity - tokensRequested, time.Now().Add(tb.config.Window), nil
}
```

Language-Specific Implementation Hints

Redis Lua Script Execution in Go:

```go
// Use redis.NewScript() to precompile Lua scripts for better performance
script := redis.NewScript(`
    -- Your Lua script here
    return {allowed, remaining, reset_time}
`)

// Execute with automatic retry and connection management
result, err := script.Run(ctx, redisClient, []string{key}, arg1, arg2).Result()
```

Time Handling for Rate Limiting:

```go
// Always use UTC to avoid timezone issues across distributed instances
now := time.Now().UTC()

// Use UnixNano() for high-precision timestamps in Redis
timestamp := now.UnixNano()

// Calculate time windows with proper boundary alignment
windowStart := now.Truncate(windowDuration)
windowEnd := windowStart.Add(windowDuration)
```

Context and Timeout Management:

```go
// Set reasonable timeouts for Redis operations (1-2ms typical)
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Millisecond)
defer cancel()

// Always check for context cancellation in long-running operations
select {
case <-ctx.Done():
    return nil, ctx.Err()
default:
    // Continue with Redis operation
}
```

Milestone Checkpoint: Basic Rate Limiter

After implementing the foundational components, verify the system with these checkpoints:

Test 1: Redis Connectivity

```bash
go run cmd/server/main.go

# Expected: Server starts without errors, connects to Redis

# Check: Redis logs show successful connections from Go client
```

Test 2: Basic Rate Limiting

```bash
# Send requests to test endpoint

for i in {1..10}; do

  curl -H "X-User-ID: test-user" http://localhost:8080/api/test

done

# Expected: First N requests succeed, subsequent requests return 429 Too Many Requests

# Check: Response includes X-RateLimit-* headers with correct values
```

Test 3: Configuration Loading

```bash
# Modify configs/rate_limits.yaml and restart server

# Expected: New rules take effect without code changes

# Check: Different endpoints have different rate limits as configured
```

Common Issues and Debugging:

- **"Connection refused"**: Ensure Redis is running on expected port (6379)
- **"WRONGTYPE Operation"**: Redis key collision with existing data, use `FLUSHDB` to clear
- **"Context deadline exceeded"**: Increase Redis operation timeouts or check network latency
- **"Rate limit headers missing"**: Ensure middleware is properly installed in HTTP handler chain

Goals and Non-Goals

> **Milestone(s):** All milestones - this foundational understanding applies to rate limiting algorithms, multi-tier systems, Redis integration, sharding, and API design throughout the project.

Mental Model: The Traffic Control Center

Think of our distributed rate limiter as a **city-wide traffic control center** coordinating traffic lights across an entire metropolitan area. Each traffic light (application instance) needs to make real-time decisions about allowing vehicles (requests) to pass through intersections (API endpoints). However, unlike independent traffic lights that only consider local conditions, our traffic control center must coordinate globally to prevent citywide congestion.

The traffic control center maintains a **shared understanding** of traffic flow across all intersections. When a major event creates a surge of vehicles heading downtown, every traffic light needs to know about the overall traffic load, not just what's happening at their local intersection. Some intersections might need stricter controls (per-user limits), while major highways require global coordination (system-wide limits). The control center must continue functioning even when communication to some traffic lights is temporarily disrupted (graceful degradation).

This analogy captures the essential challenge: **local decisions with global coordination**. Each application instance must make millisecond decisions about request acceptance while maintaining awareness of system-wide resource consumption patterns.

Functional Goals

The distributed rate limiter must provide comprehensive request quota enforcement across multiple application instances with precise control over different limiting strategies. These capabilities form the foundation for protecting system resources while maintaining fair access for legitimate users.

Core Rate Limiting Capabilities

Our system must support multiple **rate limiting algorithms** with different behavioral characteristics. The token bucket algorithm provides burst handling capabilities, allowing short periods of activity above the sustained rate while preventing long-term abuse. Users can temporarily exceed their baseline quota during legitimate usage spikes, but cannot sustain high request rates indefinitely. The sliding window counter algorithm offers memory-efficient approximate limiting with configurable accuracy trade-offs. This approach reduces memory overhead compared to exact tracking while maintaining reasonable accuracy for most use cases. The sliding window log algorithm provides precise request tracking with exact compliance checking, storing individual request timestamps to enable perfect accuracy at the cost of increased memory usage.

Algorithm	Accuracy	Memory Usage	Burst Handling	Use Case
Token Bucket	Good	Low	Excellent	API endpoints with bursty traffic
Sliding Window Counter	Good	Low	Limited	High-traffic endpoints requiring efficiency
Sliding Window Log	Perfect	High	None	Critical endpoints requiring exact limits

The system must implement **multi-tier rate limiting** with hierarchical enforcement across different dimensions. Per-user limits prevent individual users from consuming excessive resources, with different quotas based on subscription tiers or usage patterns. Per-IP limits protect against unauthenticated abuse and distributed attacks, providing a safety net when user identification is unavailable or compromised. Per-API endpoint limits ensure that no single API can overwhelm system resources, with different thresholds based on endpoint complexity and resource requirements. Global system limits provide the ultimate protection against total system overload, aggregating usage across all users, IPs, and endpoints.

The tier evaluation must follow a **short-circuit strategy** where exceeding any tier immediately blocks the request without evaluating remaining tiers. This approach minimizes computational overhead while ensuring the most restrictive applicable limit takes precedence. The system must support configurable priority ordering, allowing administrators to specify whether user limits should be checked before IP limits or vice versa based on their specific threat model.

Dynamic Configuration Management

Rate limit rules must be **dynamically configurable** without requiring application restarts or deployments. The `RuleManager` must support real-time rule updates, additions, and deletions through the management API. Rule changes must propagate to all application instances within a configurable time window, typically under 30 seconds for non-emergency changes and under 5 seconds for emergency rate limit adjustments.

Configuration Operation	Target Latency	Consistency Model	Rollback Support
Rule Creation	< 30 seconds	Eventually consistent	Full rollback
Rule Modification	< 30 seconds	Eventually consistent	Previous version restore
Emergency Rate Limit	< 5 seconds	Strong consistency	Manual override
Rule Deletion	< 60 seconds	Eventually consistent	Soft delete with restore

The system must support **rule pattern matching** using flexible key patterns that can incorporate user IDs, IP addresses, API endpoints, and custom attributes. Pattern matching should support wildcards, regular expressions, and hierarchical matching to enable sophisticated routing of requests to appropriate rate limit rules.

Atomic Operations and Consistency

All rate limit checks must be **atomic operations** that prevent race conditions between checking current usage and updating counters. The `CheckAndUpdate` method must implement check-and-increment as a single atomic operation, ensuring that concurrent requests cannot bypass limits by checking usage simultaneously before any updates occur.

The system must maintain **eventual consistency** across the distributed cluster while providing strong consistency for individual rate limit decisions. When a request is approved and counted against a limit, that decision must be immediately reflected in subsequent rate limit checks for the same key, even under high concurrency.

State Persistence and Recovery

Rate limit state must survive individual application instance failures and restarts. The Redis backend must maintain all rate limiting counters, token bucket states, and sliding window data with appropriate expiration policies to prevent indefinite memory growth.

The system must support **state recovery** mechanisms for reconstructing local fallback state from Redis data when application instances restart. This capability ensures that transitioning between Redis-backed and local fallback modes maintains reasonable accuracy rather than resetting all limits to their initial values.

Non-Functional Goals

The distributed rate limiter must meet stringent performance, reliability, and scalability requirements while maintaining operational simplicity and cost-effectiveness.

Performance Requirements

Rate limit checks must complete with **sub-5 millisecond latency** for the 95th percentile under normal load conditions. This requirement ensures that rate limiting does not become a bottleneck in request processing pipelines. The system must support at least **100,000 rate limit checks per second per application instance** while maintaining this latency target.

Performance Metric	Target	Measurement Method
P50 Latency	< 1ms	Request-response time for CheckAndUpdate
P95 Latency	< 5ms	95th percentile across all rate limit algorithms
P99 Latency	< 10ms	Including Redis network round trips
Throughput	100K ops/sec	Per application instance sustained load
Memory Usage	< 100MB	Per application instance excluding Redis

The Redis integration must implement **connection pooling** with configurable pool sizes to minimize connection establishment overhead. Connection reuse must be balanced with connection health monitoring to prevent using stale or failed connections that would increase error rates.

Reliability and Availability

The system must achieve **99.9% availability** for rate limiting decisions, measured as successful rate limit checks divided by total check attempts. This requirement must be met even during Redis node failures, network partitions, and planned maintenance activities.

Graceful degradation must ensure that rate limiting continues functioning when Redis is unavailable, falling back to local per-instance limiting with reduced accuracy. The fallback mode should maintain at least 80% of the intended rate limiting effectiveness while preserving system stability.

Failure Scenario	Recovery Time	Degraded Capability	Maintained Capability
Single Redis Node	< 1 second	Cross-instance coordination	Per-instance limiting
Redis Cluster	< 30 seconds	Global rate limits	Local fallback limits
Network Partition	< 5 seconds	Distributed coordination	Independent operation
Application Restart	< 10 seconds	Warm cache state	Cold start with Redis sync

The system must implement **circuit breaker patterns** for Redis operations, automatically switching to local fallback mode when Redis error rates exceed configurable thresholds. Circuit breaker state must be shared across application instances to prevent cascading failures.

Scalability Requirements

The system must **horizontally scale** to support at least 1000 application instances sharing rate limit state through the Redis backend. This scaling must be achieved through consistent hashing and sharding strategies that distribute load evenly across Redis nodes.

Hot key detection must identify when specific rate limit keys experience disproportionate access patterns and automatically implement mitigation strategies such as key replication or local caching. The system should handle scenarios where a small number of users or API endpoints generate the majority of rate limit checks.

Redis cluster scaling must support **dynamic node addition and removal** with minimal disruption to ongoing rate limiting operations. Consistent hashing with virtual nodes must minimize key redistribution when cluster topology changes occur.

Monitoring and Observability

The system must provide comprehensive metrics for **rate limiting effectiveness, performance, and resource usage**. Metrics must be exported in Prometheus format with appropriate labels for filtering and aggregation across different dimensions.

Metric Category	Key Metrics	Labels
Rate Limiting	Requests allowed/denied, limit utilization	algorithm, tier, rule_id, endpoint
Performance	Check latency, throughput, error rate	instance, redis_node, operation
Resource Usage	Redis memory, connection pool size, CPU	node, algorithm, key_pattern
Health	Circuit breaker state, fallback mode	instance, failure_type, recovery_time

Real-time dashboards must display current rate limit utilization across all tiers and provide alerting when usage approaches configured thresholds. Historical data must be retained for capacity planning and pattern analysis.

Explicit Non-Goals

To maintain project scope and complexity at an intermediate level, several advanced features are explicitly excluded from this implementation. These non-goals help focus development efforts on core distributed rate limiting concepts while avoiding enterprise-grade complexity.

Advanced Algorithm Features

Adaptive rate limiting that automatically adjusts limits based on system load, response times, or external signals is not included. While adaptive limiting can provide superior resource utilization, it introduces significant complexity around feedback loops, oscillation prevention, and parameter tuning that would distract from learning core distributed systems concepts.

Machine learning-based anomaly detection for identifying suspicious traffic patterns or predicting optimal rate limits is excluded. Such features require expertise in ML model training, feature engineering, and online learning systems that exceed the scope of an intermediate distributed systems project.

Geographic distribution with multi-region rate limiting coordination is not implemented. While global rate limiting across continents presents interesting challenges around latency, consistency, and network partitions, it requires infrastructure complexity beyond the Redis cluster approach used here.

Enterprise Integration Features

Authentication and authorization for the rate limiting system itself is simplified. Production systems require sophisticated access controls, API key management, and integration with enterprise identity providers, but these concerns are orthogonal to rate limiting algorithms and distributed coordination.

Excluded Feature	Rationale	Alternative Approach
OAuth2/JWT Integration	Authentication complexity exceeds scope	Simple API key validation
RBAC for Rate Limit Rules	Authorization logic unrelated to core learning	Basic admin/read-only roles
Audit Logging	Compliance features beyond technical focus	Basic operation logging
Encryption at Rest	Security implementation complexity	Redis AUTH password only

Service mesh integration with Istio, Envoy, or similar platforms is not provided. While transparent rate limiting through sidecar proxies offers operational advantages, it requires understanding service mesh architectures, Envoy filter development, and Kubernetes operators that distract from rate limiting fundamentals.

Database persistence for rate limit rules and historical data is excluded in favor of file-based configuration and Redis storage. Database integration introduces schema design, migration management, and ORM complexity without teaching additional rate limiting concepts.

Operational Complexity Features

Multi-tenancy with strict isolation between different customer environments is not implemented. True multi-tenancy requires namespace isolation, resource quotas, and security boundaries that significantly complicate the architecture without adding educational value for rate limiting concepts.

Automatic scaling and provisioning of the Redis cluster based on load patterns is excluded. While auto-scaling is valuable for production systems, it requires infrastructure automation, monitoring thresholds, and capacity planning logic that exceeds the project scope.

Disaster recovery and backup/restore capabilities are simplified. Production systems require point-in-time recovery, cross-region replication, and automated failover procedures that involve significant operational complexity beyond rate limiting algorithms.

> **Design Principle: Learning-Focused Scope** These non-goals ensure that learners can focus on mastering distributed rate limiting concepts without being overwhelmed by peripheral enterprise features. Each excluded feature represents a potential future enhancement once core concepts are solidified.

Performance and Scale Limitations

The system is designed for **medium-scale deployments** rather than hyperscale environments. Supporting millions of unique rate limit keys, petabytes of historical data, or tens of thousands of application instances would require specialized data structures, storage engines, and coordination protocols that exceed intermediate complexity.

Real-time analytics and complex aggregations over rate limiting data are limited to basic usage metrics. Advanced analytics like percentile calculations, time-series forecasting, or correlation analysis with business metrics require specialized analytics databases and query engines.

Scale Limitation	Design Target	Beyond Scope
Application Instances	1,000 instances	10,000+ instances
Rate Limit Keys	1M active keys	100M+ keys
Redis Cluster Size	10-20 nodes	100+ nodes
Historical Retention	30 days metrics	Long-term data warehouse

These limitations ensure that the implementation remains comprehensible and deployable on modest infrastructure while teaching all essential distributed rate limiting concepts. Organizations requiring hyperscale capabilities can use this implementation as a foundation for understanding the principles before adopting specialized commercial solutions.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Redis Client	`go-redis/redis` with basic connection pooling	`go-redis/redis` with cluster support and sentinel
Configuration	YAML files with `gopkg.in/yaml.v3`	etcd with watch-based dynamic updates
Metrics	Prometheus `prometheus/client_golang`	Custom metrics with multiple exporters
HTTP Framework	Standard `net/http` with `gorilla/mux`	Gin or Echo with middleware support
Logging	`logrus` or `zap` structured logging	OpenTelemetry with distributed tracing
Testing	Standard `testing` package with testify	Property-based testing with `gopter`

For this intermediate-level project, we recommend the simple options to maintain focus on rate limiting concepts rather than framework complexity. The advanced options can be adopted later as operational requirements grow.

Project Structure

```
distributed-rate-limiter/
├── cmd/
│   ├── server/main.go              ← HTTP API server entry point
│   ├── cli/main.go                 ← Command-line management tool
│   └── dashboard/main.go           ← Real-time dashboard server
├── internal/
│   ├── limiter/                    ← Core rate limiting logic
│   │   ├── distributed.go          ← DistributedLimiter implementation
│   │   ├── algorithms/             ← Rate limiting algorithms
│   │   │   ├── token_bucket.go     ← TokenBucket implementation
│   │   │   ├── sliding_window.go   ← Sliding window algorithms
│   │   │   └── interface.go        ← Common algorithm interface
│   │   └── fallback.go             ← Local fallback limiter
│   ├── storage/                    ← Redis backend integration
│   │   ├── redis.go                ← RedisStorage implementation
│   │   ├── scripts.go              ← Lua script management
│   │   └── sharding.go             ← Consistent hashing logic
│   ├── config/                     ← Configuration management
│   │   ├── rules.go                ← RuleManager implementation
│   │   ├── loader.go               ← Configuration file loading
│   │   └── validation.go           ← Rule validation logic
│   ├── api/                        ← HTTP API handlers
│   │   ├── handlers.go             ← Rate limit check endpoints
│   │   ├── management.go           ← Rule management endpoints
│   │   └── middleware.go           ← Rate limit headers middleware
│   └── metrics/                    ← Monitoring and metrics
│       ├── collector.go            ← Prometheus metrics
│       └── dashboard.go            ← Real-time dashboard data
├── configs/
│   ├── rules.yaml                  ← Rate limit rule definitions
│   └── redis.yaml                  ← Redis cluster configuration
├── scripts/
│   └── lua/                        ← Redis Lua scripts
│       ├── token_bucket.lua        ← Token bucket algorithm
│       ├── sliding_window.lua      ← Sliding window algorithm
│       └── utils.lua               ← Common script utilities
└── tests/
    ├── integration/                ← Multi-instance integration tests
    ├── chaos/                      ← Chaos engineering tests
    └── benchmarks/                 ← Performance benchmarks
```

This structure separates concerns clearly while maintaining the `internal/` convention for non-exported packages. Each package has a single responsibility and minimal dependencies on other internal packages.

Core Configuration Structures

```go
// RedisConfig defines connection parameters for Redis cluster integration
type RedisConfig struct {
    Addresses     []string      `yaml:"addresses"`     // Redis cluster node addresses
    Password      string        `yaml:"password"`      // AUTH password for Redis
    DB            int           `yaml:"db"`            // Database number (0-15)
    PoolSize      int           `yaml:"pool_size"`     // Connection pool size per node
    ReadTimeout   time.Duration `yaml:"read_timeout"`  // Socket read timeout
    WriteTimeout  time.Duration `yaml:"write_timeout"` // Socket write timeout
    DialTimeout   time.Duration `yaml:"dial_timeout"`  // Connection establishment timeout
}

// NewRedisConfig creates a RedisConfig with sensible defaults
func NewRedisConfig() RedisConfig {
    return RedisConfig{
        Addresses:    []string{"localhost:6379"},
        Password:     "",
        DB:           0,
        PoolSize:     DEFAULT_POOL_SIZE,
        ReadTimeout:  DEFAULT_TIMEOUT,
        WriteTimeout: DEFAULT_TIMEOUT,
        DialTimeout:  DEFAULT_TIMEOUT,
    }
}
```

Rate Limit Rule Definition

```go
// RateLimitRule defines a rate limiting policy that can be applied to requests
type RateLimitRule struct {
    ID          string        `yaml:"id" json:"id"`                   // Unique rule identifier
    Name        string        `yaml:"name" json:"name"`               // Human-readable rule name
    KeyPattern  string        `yaml:"key_pattern" json:"key_pattern"` // Pattern for matching requests
    Algorithm   string        `yaml:"algorithm" json:"algorithm"`     // Algorithm: "token_bucket", "sliding_window_counter",
"sliding_window_log"
    Limit       int64         `yaml:"limit" json:"limit"`             // Maximum requests per window
    Window      time.Duration `yaml:"window" json:"window"`           // Time window duration
    BurstLimit  int64         `yaml:"burst_limit" json:"burst_limit"` // Maximum burst size (token bucket only)
    Enabled     bool          `yaml:"enabled" json:"enabled"`         // Whether rule is active
    Priority    int           `yaml:"priority" json:"priority"`       // Evaluation priority (higher = first)
    CreatedAt   time.Time     `yaml:"created_at" json:"created_at"`   // Rule creation timestamp
    UpdatedAt   time.Time     `yaml:"updated_at" json:"updated_at"`   // Last modification timestamp
}
```

Essential Constants

```go
const (
    // Redis connection pool configuration
    DEFAULT_POOL_SIZE = 10              // Connections per Redis node
    DEFAULT_TIMEOUT   = 5 * time.Millisecond  // Redis operation timeout

    // Rule priority levels for common use cases
    PRIORITY_HIGH = 100                 // Emergency rate limits
    PRIORITY_LOW  = 1                  // Default background limits

    // Rate limiting algorithms
    ALGORITHM_TOKEN_BUCKET       = "token_bucket"
    ALGORITHM_SLIDING_WINDOW_LOG = "sliding_window_log"
    ALGORITHM_SLIDING_COUNTER    = "sliding_window_counter"

    // Rate limit header names
    HEADER_LIMIT_REMAINING = "X-RateLimit-Remaining"
    HEADER_LIMIT_RESET     = "X-RateLimit-Reset"
    HEADER_RETRY_AFTER     = "Retry-After"
)
```

Milestone Checkpoints

After completing Milestone 1 (Rate Limiting Algorithms):

- Run `go test ./internal/limiter/algorithms/...` - all algorithm tests should pass
- Create a simple CLI tool that can test each algorithm with configurable parameters
- Verify that token bucket allows bursts above sustained rate but prevents long-term abuse
- Test sliding window algorithms with requests clustered at window boundaries
- Expected behavior: Token bucket should allow 10 requests immediately if capacity=10, then throttle to refill rate

After completing Milestone 2 (Multi-tier Rate Limiting):

- Run integration test with simultaneous per-user and global limits
- Verify that the most restrictive limit takes precedence using short-circuit evaluation
- Test rule pattern matching with wildcards and regular expressions
- Expected behavior: Request matching both user limit (100/hour) and global limit (1000/hour) should be limited by user quota first

After completing Milestone 3 (Redis Backend Integration):

- Start Redis locally and run `go test ./internal/storage/...`
- Test graceful degradation by stopping Redis mid-test and verifying local fallback
- Verify that concurrent requests don't bypass limits using atomic Lua scripts
- Expected behavior: Two application instances should coordinate limits correctly through Redis

After completing Milestone 4 (Consistent Hashing & Sharding):

- Deploy Redis cluster with 3 nodes and test key distribution
- Add/remove Redis nodes and verify minimal key redistribution
- Test hot key detection with skewed request patterns
- Expected behavior: Keys should be evenly distributed, with <10% redistribution when adding nodes

After completing Milestone 5 (Rate Limit API & Dashboard):

- Start the HTTP API server and test rule management endpoints with curl
- Verify rate limit headers are included in all API responses
- Test the real-time dashboard updates as rate limits are consumed
- Expected behavior: Dashboard should show live usage percentages updating every second

Each milestone should build incrementally, with later milestones reusing and extending earlier implementations rather than replacing them.

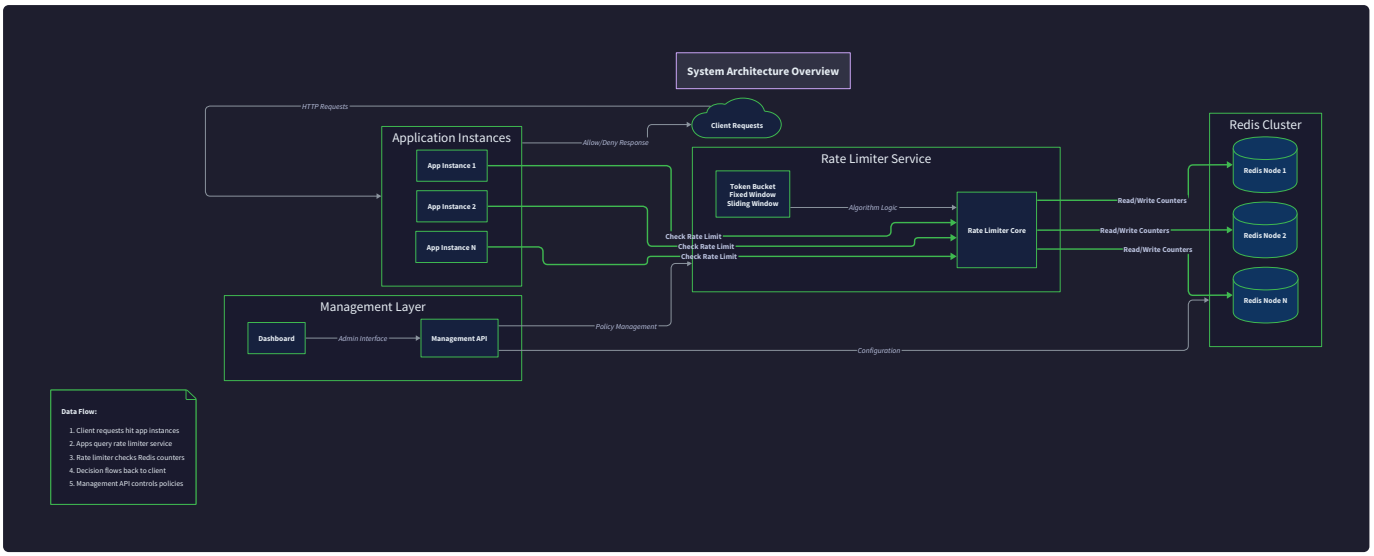
High-Level Architecture

> **Milestone(s):** All milestones - this architectural foundation enables rate limiting algorithms (Milestone 1), multi-tier evaluation (Milestone 2), Redis backend integration (Milestone 3), consistent hashing and sharding (Milestone 4), and API management (Milestone 5).

The distributed rate limiter requires careful coordination between multiple components to maintain consistent quota enforcement across application instances while handling failures gracefully. Think of this system like a **coordinated nightclub security operation** - instead of just one bouncer at the door counting patrons, you have multiple entry points (application instances) that must communicate with a central dispatch system (Redis) to maintain an accurate headcount and enforce capacity limits consistently across all entrances.

This mental model captures the core challenge: each application instance acts as an independent bouncer checking IDs and counting entries, but they all must coordinate through a shared communication system to ensure the overall venue doesn't exceed capacity. When the radio system goes down (Redis failure), each bouncer falls back to local counting with reduced accuracy, but the doors stay open.

The architecture balances several competing concerns through a layered design. At the foundation, Redis provides atomic operations and cluster-wide state sharing. Above that, the rate limiting algorithms implement different quota enforcement strategies with varying accuracy and performance trade-offs. The multi-tier evaluation layer applies hierarchical limits across user, IP, API, and global dimensions. Finally, the management layer provides dynamic configuration and real-time monitoring capabilities.



> **Key Architectural Principle**: The system maintains functionality even during partial failures through graceful degradation. Each component can operate in a reduced-capability mode when dependencies are unavailable, preventing cascading failures while maintaining core rate limiting capabilities.

Component Overview

The distributed rate limiter consists of seven primary components, each with distinct responsibilities and failure modes. Understanding these components and their interactions is crucial for implementing a robust system that handles the complexities of distributed quota enforcement.

DistributedLimiter Component

The `DistributedLimiter` serves as the orchestration layer, coordinating multi-tier rate limit evaluation and managing fallback strategies during Redis failures. This component implements the primary `Check()` method that application code calls to verify whether a request should be allowed or rejected.

Responsibility	Description	Failure Behavior
Request Orchestration	Coordinates multi-tier rate limit checks with short-circuit evaluation	Continues with remaining tiers if one fails
Algorithm Selection	Routes requests to appropriate algorithm implementation based on rule configuration	Falls back to token bucket if specified algorithm unavailable
Fallback Coordination	Switches to local rate limiting when Redis becomes unavailable	Maintains reduced accuracy with per-instance limits
Result Aggregation	Combines results from multiple tiers into final allow/deny decision	Uses most restrictive result from successful tier checks

The `DistributedLimiter` maintains no persistent state itself - it acts purely as a coordinator that delegates to storage backends and algorithm implementations. This stateless design ensures that multiple application instances can run identical rate limiting logic without coordination overhead.

> **Decision: Stateless Coordinator Design**
>
> - **Context**: Need to run identical rate limiting logic across multiple application instances
> - **Options Considered**: Stateful coordinator with instance coordination, stateless coordinator with shared storage, leader-election based coordination
> - **Decision**: Stateless coordinator with all persistent state in Redis
> - **Rationale**: Eliminates complex coordination between application instances, simplifies deployment and scaling, enables any instance to handle any request
> - **Consequences**: All state must be externalized to Redis, requires careful atomic operation design, enables horizontal scaling without instance affinity

Storage Backend Components

The storage layer abstracts persistent state management through a common `Storage` interface, with `RedisStorage` as the primary implementation and local storage options for fallback scenarios.

RedisStorage Component

`RedisStorage` implements distributed state management using Redis cluster operations with atomic Lua scripts. This component handles the complexity of coordinating rate limit state across multiple application instances while maintaining consistency guarantees.

Feature	Implementation	Consistency Guarantee
Atomic Operations	Lua scripts for check-and-update	Strong consistency within single Redis node
Connection Pooling	Redis universal client with configurable pool size	Automatic failover between cluster nodes
Cluster Support	Consistent hashing for key distribution	Eventually consistent across cluster during splits
Health Monitoring	Periodic ping with circuit breaker logic	Automatic fallback when health check fails

The Redis integration uses Lua scripts to ensure atomicity of complex operations like token bucket refill-and-consume cycles. Without atomic operations, race conditions between multiple application instances could lead to incorrect quota enforcement - imagine two bouncers at different doors both thinking they're letting in the "last" patron simultaneously.

Local Storage Fallback

When Redis becomes unavailable, the system maintains functionality through local storage implementations that provide per-instance rate limiting. While accuracy is reduced (each instance enforces limits independently), the system continues protecting against abuse rather than failing open.

Storage Type	Use Case	Accuracy Trade-off
In-Memory Map	Development and fallback	Lost on process restart
Local Database	Persistent fallback	Per-instance limits only
File-based Storage	Simple persistence	High latency, suitable for low-traffic fallback

Algorithm Implementation Components

Each rate limiting algorithm is implemented as a separate component that works with the storage abstraction. This design allows mixing different algorithms for different use cases within the same deployment.

TokenBucket Component

The `TokenBucket` implementation manages refill rates and burst capacity using atomic compare-and-swap operations in Redis. The algorithm maintains a bucket state with current token count and last refill timestamp.

State Field	Purpose	Update Frequency
tokens	Current available tokens	Every request (decremented) and refill interval
last_refill_time	Nanosecond timestamp of last refill	Every refill calculation
capacity	Maximum bucket size	Configuration only
refill_rate	Tokens added per time window	Configuration only

The token bucket algorithm allows controlled bursts above the sustained rate, making it ideal for APIs that need to handle occasional spikes while maintaining average throughput limits. The refill calculation happens atomically with token consumption to prevent race conditions.

SlidingWindow Implementations

Both sliding window counter and sliding window log algorithms track request patterns over time windows, but with different memory and accuracy trade-offs.

Algorithm	Memory Usage	Accuracy	Best For
Sliding Window Counter	O(1) per key	Approximate (up to 2x limit at boundaries)	High-traffic APIs needing memory efficiency
Sliding Window Log	O(requests) per key	Exact	Critical APIs requiring precise enforcement

RuleManager Component

The `RuleManager` handles dynamic configuration of rate limiting rules, supporting real-time updates without service restarts. Rules are organized by priority and matching patterns to enable complex hierarchical rate limiting scenarios.

Rule Matching	Pattern Type	Example	Priority Handling
User ID	Exact match	`user:12345`	Higher priority overrides lower
IP Address	CIDR blocks	`192.168.1.0/24`	More specific patterns win
API Endpoint	Path patterns	`/api/v1/upload/*`	Endpoint-specific before global
User Agent	Regex patterns	`bot|crawler`	Pattern complexity affects performance

The rule evaluation engine uses short-circuit evaluation to minimize Redis operations - once a rate limit is exceeded, evaluation stops immediately rather than checking remaining tiers.

> **Decision: Priority-Based Rule Evaluation**
>
> - **Context**: Need to handle overlapping rate limit rules for the same request
> - **Options Considered**: First-match wins, most restrictive wins, priority-based evaluation
> - **Decision**: Priority-based evaluation with configurable rule precedence
> - **Rationale**: Provides predictable behavior for complex rule sets, allows override patterns for special cases, enables debugging through explicit rule ordering
> - **Consequences**: Requires careful priority assignment, evaluation order affects performance, enables sophisticated rate limiting policies

ConsistentHashRing Component

For horizontally scaled Redis deployments, the `ConsistentHashRing` distributes rate limiting keys across multiple Redis nodes while minimizing redistribution during topology changes.

Hash Ring Feature	Purpose	Implementation
Virtual Nodes	Uniform distribution	100-500 virtual nodes per physical node
Key Placement	Consistent node assignment	SHA-256 hash of rate limit key
Node Addition	Minimal key movement	Only keys between old and new node positions move
Hot Key Detection	Load balancing	Monitor request frequency per key

The consistent hashing approach ensures that adding or removing Redis nodes only affects a small fraction of keys, maintaining cache locality and avoiding thundering herd problems during topology changes.

Metrics and Monitoring Components

Real-time observability is crucial for distributed rate limiting since quota violations may indicate either legitimate traffic spikes or attack patterns.

MetricsCollector Component

Metric Category	Examples	Collection Method
Rate Limit Decisions	Allowed/denied counts per rule	In-process counters
Algorithm Performance	Token bucket refill latency	Histogram metrics
Storage Operations	Redis operation latency/errors	Redis client middleware
Hot Key Detection	Request frequency distribution	Sliding window counters

Dashboard Components

The real-time dashboard requires efficient data streaming to avoid overwhelming the rate limiting system with monitoring queries.

Dashboard Feature	Data Source	Update Frequency
Current Usage Gauges	Redis state queries	1 second intervals
Historical Trends	Time-series metrics	10 second aggregation
Rule Configuration	RuleManager API	On-demand with WebSocket push
Health Status	Component health checks	5 second intervals

> **Critical Design Insight**: The monitoring system must not become a bottleneck or single point of failure for the rate limiting system. Dashboard queries are rate-limited and use read replicas where possible to avoid affecting production rate limiting performance.

Recommended Module Structure

A well-organized module structure enables parallel development of different components while maintaining clear dependency boundaries. The structure follows domain-driven design principles with infrastructure concerns separated from business logic.

```
distributed-rate-limiter/
├── cmd/
│   ├── server/                    # HTTP API server
│   │   └── main.go                # Server entry point with configuration
│   ├── cli/                       # Management CLI tool
│   │   └── main.go                # CLI for rule management and testing
│   └── dashboard/                 # Dashboard web server
│       └── main.go                # Dashboard with WebSocket endpoints
├── internal/
│   ├── limiter/                   # Core rate limiting logic
│   │   ├── limiter.go             # DistributedLimiter implementation
│   │   ├── limiter_test.go        # Multi-instance integration tests
│   │   ├── request.go             # RateLimitRequest and RateLimitResult
│   │   └── interface.go           # Limiter interface definition
│   ├── algorithms/                # Rate limiting algorithm implementations
│   │   ├── tokenbucket/           # Token bucket algorithm
│   │   │   ├── tokenbucket.go     # TokenBucket component
│   │   │   ├── tokenbucket_test.go # Algorithm correctness tests
│   │   │   └── state.go           # TokenBucketState and TokenBucketConfig
│   │   ├── slidingwindow/         # Sliding window algorithms
│   │   │   ├── counter.go         # Sliding window counter implementation
│   │   │   ├── log.go             # Sliding window log implementation
│   │   │   └── window_test.go     # Boundary condition tests
│   │   └── interface.go           # Common algorithm interface
│   ├── storage/                   # Storage backend implementations
│   │   ├── redis/                 # Redis backend
│   │   │   ├── storage.go         # RedisStorage implementation
│   │   │   ├── scripts.go         # Lua scripts for atomic operations
│   │   │   ├── pool.go            # Connection pool management
│   │   │   ├── health.go          # Health checking and circuit breaker
│   │   │   └── redis_test.go      # Redis integration tests
│   │   ├── local/                 # Local storage fallback
│   │   │   ├── memory.go          # In-memory map storage
│   │   │   └── file.go            # File-based persistence
│   │   └── interface.go           # Storage interface definition
│   ├── config/                    # Configuration management
│   │   ├── rules.go               # RuleManager implementation
│   │   ├── loader.go              # Configuration file loading
│   │   ├── validation.go          # Rule validation logic
│   │   └── config_test.go         # Rule matching tests
│   ├── sharding/                  # Consistent hashing and node management
│   │   ├── hasher.go              # ConsistentHashRing implementation
│   │   ├── nodes.go               # Node health and topology management
│   │   ├── rebalancer.go          # Hot key detection and rebalancing
│   │   └── sharding_test.go       # Hash distribution tests
│   ├── metrics/                   # Observability and monitoring
│   │   ├── collector.go           # MetricsCollector component
│   │   ├── dashboard.go           # Dashboard data aggregation
│   │   └── alerts.go              # Alerting logic for quota violations
│   ├── api/                       # HTTP API handlers
│   │   ├── handlers/              # Request handlers
│   │   │   ├── ratelimit.go       # Rate limit check endpoints
│   │   │   ├── rules.go           # Rule management CRUD endpoints
│   │   │   └── metrics.go         # Metrics and dashboard endpoints
│   │   ├── middleware/            # HTTP middleware
│   │   │   ├── headers.go         # Rate limit header injection
│   │   │   ├── logging.go         # Request logging and tracing
│   │   │   └── recovery.go        # Panic recovery middleware
│   │   └── server.go              # HTTP server setup and routing
│   └── util/                      # Shared utilities
│       ├── time.go                # Time utilities for clock skew handling
│       ├── hash.go                # Consistent hashing utilities
│       └── testing.go             # Test helpers for integration tests
├── pkg/                           # Public API packages
│   ├── client/                    # Go client library
│   │   ├── client.go              # HTTP client for rate limit checks
│   │   └── client_test.go         # Client integration tests
│   └── types/                     # Shared type definitions
│       ├── rules.go               # RateLimitRule and related types
│       ├── results.go             # RateLimitResult and error types
│       └── config.go              # Configuration structures
├── configs/                       # Configuration file examples
│   ├── rules.yaml                 # Example rate limiting rules
│   ├── redis.yaml                 # Redis cluster configuration
```

```
│   └── server.yaml             # Server configuration
├── scripts/                     # Deployment and development scripts
│   ├── setup-redis.sh          # Redis cluster setup script
│   ├── load-test.sh            # Rate limiter load testing
│   └── migrate-rules.sh        # Rule migration utilities
├── deployments/                 # Kubernetes and Docker configurations
│   ├── k8s/                    # Kubernetes manifests
│   └── docker/                 # Docker compose configurations
└── docs/                        # Additional documentation
    ├── algorithms.md           # Algorithm comparison and selection guide
    ├── deployment.md           # Production deployment guide
    └── troubleshooting.md      # Common issues and solutions
```

Module Dependency Guidelines

The module structure enforces clear dependency boundaries to prevent circular imports and enable independent testing of components.

Layer	Allowed Dependencies	Prohibited Dependencies
`cmd/` packages	Any internal package	None - these are entry points
`internal/limiter/`	`algorithms/`, `storage/`, `config/`, `metrics/`	`api/`, `cmd/` - business logic independent of transport
`internal/algorithms/`	`storage/` interface only	Concrete storage implementations
`internal/storage/`	`util/` only	Algorithm or limiter packages
`internal/config/`	`util/`, `pkg/types/`	Storage or algorithm implementations
`internal/api/`	`limiter/`, `config/`, `metrics/`	Storage implementations directly

> **Decision: Hexagonal Architecture with Interface Boundaries**
>
> - **Context**: Need to enable parallel development and independent testing of components
> - **Options Considered**: Layered architecture, microservices with RPC, hexagonal architecture with interfaces
> - **Decision**: Hexagonal architecture with storage and algorithm abstractions
> - **Rationale**: Enables testing without Redis dependencies, supports multiple storage backends, allows algorithm experimentation without changing core logic
> - **Consequences**: Requires discipline to maintain interface boundaries, adds abstraction overhead, enables comprehensive unit testing

Configuration Management Structure

Rate limiting rules require dynamic updates without service restarts, necessitating a structured approach to configuration management.

Configuration Type	File Location	Update Method	Validation
Rate Limit Rules	`configs/rules.yaml`	HTTP API with immediate propagation	Schema validation + rule conflict detection
Redis Configuration	`configs/redis.yaml`	Environment variables + config file	Connection testing during startup
Server Settings	`configs/server.yaml`	Environment variables only	Port availability and permission checks
Algorithm Parameters	Embedded in rules	Rule update API	Algorithm-specific parameter validation

The configuration system supports environment variable overrides for deployment-specific settings while maintaining rule definitions in version-controlled YAML files.

Common Pitfalls

Understanding common mistakes in distributed rate limiter architecture helps avoid subtle bugs that only manifest under high load or failure conditions.

⚠️ Pitfall: Circular Dependencies Between Components

Many implementations create circular import cycles by having storage components depend on algorithm implementations while algorithms depend on storage interfaces. This typically happens when trying to implement algorithm-specific optimizations in the storage layer.

Why it's wrong: Circular dependencies prevent independent testing and make the codebase fragile to changes. They also indicate that separation of concerns is violated.

How to fix: Use dependency inversion with interfaces. Storage components should only know about generic operations, while algorithms implement their logic using storage interface methods.

⚠️ Pitfall: Blocking Operations in Request Path

Synchronous Redis operations in the request handling path can cause cascading failures when Redis latency spikes. This is especially problematic when checking multiple rate limit tiers sequentially.

Why it's wrong: A slow Redis query blocks the entire request, leading to thread pool exhaustion and service unavailability even when Redis is only temporarily slow.

How to fix: Use timeouts for all Redis operations, implement circuit breaker patterns, and provide local fallback that can execute when Redis operations timeout.

⚠️ Pitfall: Inconsistent Clock Sources

Using different time sources (system clock vs Redis TIME command) across components leads to inconsistent rate limit window calculations and can cause requests to be incorrectly allowed or denied.

Why it's wrong: Clock skew between application instances and Redis nodes causes time-based calculations to diverge, leading to unpredictable rate limiting behavior.

How to fix: Use a consistent time source strategy - either always use Redis TIME command for distributed consistency, or ensure NTP synchronization across all nodes and use local clocks consistently.

⚠️ Pitfall: Missing Graceful Degradation Strategy

Failing hard when Redis is unavailable means rate limiting becomes a single point of failure that can take down the entire application.

Why it's wrong: Rate limiting is a protection mechanism, not a core business function. Its unavailability should not prevent the application from functioning.

How to fix: Implement local fallback storage that provides reduced-accuracy rate limiting when Redis is unavailable. Design the system to "fail open" with monitoring rather than "fail closed" and break the application.

Implementation Guidance

This section provides the foundational code structure and infrastructure components needed to implement the distributed rate limiter architecture. The code focuses on providing complete, working infrastructure that learners can build upon rather than implementing the core rate limiting algorithms themselves.

Technology Recommendations

Component	Simple Option	Advanced Option
Redis Client	`github.com/go-redis/redis/v8` with single instance	`github.com/go-redis/redis/v8` with cluster support
HTTP Framework	Standard `net/http` with custom routing	`github.com/gin-gonic/gin` for middleware ecosystem
Configuration	YAML files with `gopkg.in/yaml.v3`	`github.com/spf13/viper` for multiple formats
Metrics	Simple counters with `expvar`	`github.com/prometheus/client_golang`
Logging	Standard `log` package	`github.com/sirupsen/logrus` with structured logging
Testing	Standard `testing` package	`github.com/stretchr/testify` for assertions

Recommended File Structure Setup

Start by creating the basic directory structure and go module:

```bash
mkdir distributed-rate-limiter

cd distributed-rate-limiter

go mod init github.com/yourusername/distributed-rate-limiter
```

Create the core directories:

```bash
mkdir -p
{cmd/{server,cli,dashboard},internal/{limiter,algorithms/{tokenbucket,slidingwindow},storage/{redis,local},config,sharding,metrics,api/{handlers,middleware},util},pkg/{client,types},configs,scripts,deployments/{k8s,docker},docs}
```

Infrastructure Starter Code

Complete Redis Configuration and Connection Management

`internal/storage/redis/config.go`:

```go
package redis

import (
    "context"
    "crypto/tls"
    "time"
    "github.com/go-redis/redis/v8"
)

// RedisConfig holds all Redis connection and operation parameters
type RedisConfig struct {
    // Addresses contains Redis node addresses for cluster or single instance
    Addresses []string `yaml:"addresses" json:"addresses"`

    // Password for Redis authentication
    Password string `yaml:"password" json:"password"`

    // DB database number (ignored in cluster mode)
    DB int `yaml:"db" json:"db"`

    // PoolSize maximum number of socket connections
    PoolSize int `yaml:"pool_size" json:"pool_size"`

    // ReadTimeout for socket reads
    ReadTimeout time.Duration `yaml:"read_timeout" json:"read_timeout"`

    // WriteTimeout for socket writes
    WriteTimeout time.Duration `yaml:"write_timeout" json:"write_timeout"`

    // DialTimeout for establishing new connections
    DialTimeout time.Duration `yaml:"dial_timeout" json:"dial_timeout"`

    // EnableTLS enables TLS encryption
    EnableTLS bool `yaml:"enable_tls" json:"enable_tls"`

    // TLSConfig for custom TLS settings
    TLSConfig *tls.Config `yaml:"-" json:"-"`
}
```

```go
// DefaultRedisConfig returns configuration with sensible defaults
func DefaultRedisConfig() RedisConfig {
    return RedisConfig{
        Addresses:    []string{"localhost:6379"},
        Password:     "",
        DB:           0,
        PoolSize:     DEFAULT_POOL_SIZE,
        ReadTimeout:  DEFAULT_TIMEOUT,
        WriteTimeout: DEFAULT_TIMEOUT,
        DialTimeout:  DEFAULT_TIMEOUT,
        EnableTLS:    false,
    }
}

// Validate checks configuration parameters for correctness
func (c RedisConfig) Validate() error {
    if len(c.Addresses) == 0 {
        return errors.New("at least one Redis address required")
    }

    if c.PoolSize <= 0 {
        return errors.New("pool_size must be positive")
    }

    if c.ReadTimeout <= 0 || c.WriteTimeout <= 0 || c.DialTimeout <= 0 {
        return errors.New("all timeout values must be positive")
    }

    return nil
}
```

Complete Redis Storage Implementation with Health Checking

`internal/storage/redis/storage.go`:

```go
package redis

import (
    "context"
    "fmt"
    "sync"
    "time"

    "github.com/go-redis/redis/v8"
)

// RedisStorage implements distributed rate limiting storage using Redis
type RedisStorage struct {
    client redis.UniversalClient
    config RedisConfig

    // Health checking
    healthy    bool
    healthLock sync.RWMutex
    lastCheck  time.Time

    // Circuit breaker state
    failureCount int
    lastFailure  time.Time
}

// NewRedisStorage creates Redis storage with connection pooling and health monitoring
func NewRedisStorage(config RedisConfig) (*RedisStorage, error) {
    if err := config.Validate(); err != nil {
        return nil, fmt.Errorf("invalid Redis configuration: %w", err)
    }

    // Create universal client (works with both single instance and cluster)
    var client redis.UniversalClient

    if len(config.Addresses) == 1 {
        // Single instance mode
        client = redis.NewClient(&redis.Options{
            Addr:         config.Addresses[0],
```

```
        Password:     config.Password,
        DB:           config.DB,
        PoolSize:     config.PoolSize,
        ReadTimeout:  config.ReadTimeout,
        WriteTimeout: config.WriteTimeout,
        DialTimeout:  config.DialTimeout,
        TLSConfig:    config.TLSConfig,
    })
} else {
    // Cluster mode
    client = redis.NewClusterClient(&redis.ClusterOptions{
        Addrs:        config.Addresses,
        Password:     config.Password,
        PoolSize:     config.PoolSize,
        ReadTimeout:  config.ReadTimeout,
        WriteTimeout: config.WriteTimeout,
        DialTimeout:  config.DialTimeout,
        TLSConfig:    config.TLSConfig,
    })
}

storage := &RedisStorage{
    client:  client,
    config:  config,
    healthy: true,
}

// Verify connection
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

if err := storage.client.Ping(ctx).Err(); err != nil {
    return nil, fmt.Errorf("failed to connect to Redis: %w", err)
}

// Start health checking goroutine
go storage.healthChecker()
```

```
    return storage, nil
}

// healthChecker runs periodic health checks in background
func (r *RedisStorage) healthChecker() {
    ticker := time.NewTicker(5 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)

        err := r.client.Ping(ctx).Err()
        cancel()

        r.healthLock.Lock()
        r.lastCheck = time.Now()
        if err != nil {
            r.healthy = false
            r.failureCount++
            r.lastFailure = time.Now()
        } else {
            r.healthy = true
            r.failureCount = 0
        }
        r.healthLock.Unlock()
    }
}

// IsHealthy returns current health status
func (r *RedisStorage) IsHealthy() bool {
    r.healthLock.RLock()
    defer r.healthLock.RUnlock()
    return r.healthy
}

// Close closes Redis connections
func (r *RedisStorage) Close() error {
    return r.client.Close()
```

```
}
```

Core Type Definitions

`pkg/types/rules.go`:

```go
package types

import (
    "time"
)

// RateLimitRule defines a rate limiting rule with all configuration parameters
type RateLimitRule struct {
    // ID uniquely identifies this rule
    ID string `yaml:"id" json:"id"`

    // Name human-readable rule name
    Name string `yaml:"name" json:"name"`

    // KeyPattern defines which requests this rule applies to
    // Examples: "user:*", "ip:192.168.1.*", "api:/upload/*"
    KeyPattern string `yaml:"key_pattern" json:"key_pattern"`

    // Algorithm specifies which rate limiting algorithm to use
    Algorithm string `yaml:"algorithm" json:"algorithm"`

    // Limit maximum number of requests allowed in the time window
    Limit int64 `yaml:"limit" json:"limit"`

    // Window time duration for the rate limit
    Window time.Duration `yaml:"window" json:"window"`

    // BurstLimit allows temporary bursts above the sustained limit (token bucket only)
    BurstLimit int64 `yaml:"burst_limit" json:"burst_limit"`

    // Enabled whether this rule is currently active
    Enabled bool `yaml:"enabled" json:"enabled"`

    // Priority determines evaluation order (higher numbers evaluated first)
    Priority int `yaml:"priority" json:"priority"`

    // CreatedAt timestamp when rule was created
    CreatedAt time.Time `yaml:"created_at" json:"created_at"`
```

```
    // UpdatedAt timestamp when rule was last modified
    UpdatedAt time.Time `yaml:"updated_at" json:"updated_at"`
}

// RateLimitRequest contains all context needed to evaluate rate limits
type RateLimitRequest struct {
    // UserID for authenticated user rate limiting
    UserID string `json:"user_id"`

    // IPAddress for IP-based rate limiting
    IPAddress string `json:"ip_address"`

    // APIEndpoint for per-endpoint rate limiting
    APIEndpoint string `json:"api_endpoint"`

    // UserAgent for bot detection and user-agent-based limiting
    UserAgent string `json:"user_agent"`

    // Tokens number of tokens to consume (default 1)
    Tokens int64 `json:"tokens"`
}

// RateLimitResult contains the decision and metadata from a rate limit check
type RateLimitResult struct {
    // Allowed whether the request should be permitted
    Allowed bool `json:"allowed"`

    // Remaining number of requests remaining in current window
    Remaining int64 `json:"remaining"`

    // RetryAfter duration to wait before next allowed request
    RetryAfter time.Duration `json:"retry_after"`

    // ResetTime when the rate limit window resets
    ResetTime time.Time `json:"reset_time"`

    // RuleID which rule triggered this result
```

```
    RuleID string `json:"rule_id"`


    // Algorithm which algorithm was used
    Algorithm string `json:"algorithm"`
}
```

Storage Interface Definition

`internal/storage/interface.go`:

```go
package storage

import (
    "context"
    "time"
)

// Storage defines the interface for rate limiting storage backends
type Storage interface {
    // CheckAndUpdate atomically checks current count and updates if under limit
    // Returns: allowed, remaining, resetTime, error
    CheckAndUpdate(ctx context.Context, key string, limit int64, window time.Duration) (bool, int64, time.Time, error)

    // Preview checks current status without updating counters
    Preview(ctx context.Context, key string, limit int64, window time.Duration) (bool, int64, time.Time, error)

    // Reset clears all counters for a key
    Reset(ctx context.Context, key string) error

    // IsHealthy returns whether storage backend is available
    IsHealthy() bool

    // Close releases resources
    Close() error
}
```

Core Component Skeletons

Rate Limiter Interface and Core Types

`internal/limiter/interface.go`:

```go
package limiter

import (
    "context"

    "github.com/yourusername/distributed-rate-limiter/pkg/types"
)

// Limiter defines the interface for rate limiting implementations
type Limiter interface {
    // Check performs rate limit evaluation and updates counters
    Check(ctx context.Context, req types.RateLimitRequest) (*types.RateLimitResult, error)

    // Preview checks rate limit status without updating counters
    Preview(ctx context.Context, req types.RateLimitRequest) (*types.RateLimitResult, error)

    // Reset clears rate limit counters for a request context
    Reset(ctx context.Context, req types.RateLimitRequest) error
}
```

Main DistributedLimiter Component (Core Logic Skeleton)

`internal/limiter/limiter.go`:

```go
package limiter

import (
    "context"
    "fmt"

    "github.com/yourusername/distributed-rate-limiter/internal/config"
    "github.com/yourusername/distributed-rate-limiter/internal/storage"
    "github.com/yourusername/distributed-rate-limiter/pkg/types"
)

// DistributedLimiter coordinates rate limiting across multiple tiers with fallback
type DistributedLimiter struct {
    storage       storage.Storage
    ruleManager   *config.RuleManager
    localFallback Limiter
}

// NewDistributedLimiter creates distributed rate limiter instance
func NewDistributedLimiter(storage storage.Storage, ruleManager *config.RuleManager) *DistributedLimiter {
    // TODO: Create local fallback limiter for when Redis is unavailable
    // TODO: Initialize metrics collection

    return &DistributedLimiter{
        storage:     storage,
        ruleManager: ruleManager,
        // localFallback: localLimiter,
    }
}

// Check performs multi-tier rate limit evaluation with short-circuit logic
func (d *DistributedLimiter) Check(ctx context.Context, req types.RateLimitRequest) (*types.RateLimitResult, error) {
    // TODO 1: Check if storage backend is healthy, switch to fallback if not
    // TODO 2: Get matching rules from rule manager based on request context
    // TODO 3: Sort rules by priority (highest priority first)
    // TODO 4: Evaluate each rule using short-circuit logic:
    //         - Stop immediately if any rule denies the request
    //         - Track the most restrictive remaining count across all rules
    // TODO 5: For each rule that applies:
```

```
    //         - Generate appropriate Redis key (user:123, ip:1.2.3.4, etc.)
    //         - Call storage.CheckAndUpdate with rule's limit and window
    //         - If denied, return immediately with rule details
    // TODO 6: If all rules pass, return allowed=true with most restrictive remaining count
    // TODO 7: Handle storage errors by falling back to local limiter
    // TODO 8: Update metrics for allowed/denied decisions

    return nil, fmt.Errorf("not implemented")
}

// Preview checks rate limit status without updating counters
func (d *DistributedLimiter) Preview(ctx context.Context, req types.RateLimitRequest) (*types.RateLimitResult, error) {
    // TODO 1: Same logic as Check() but call storage.Preview instead of CheckAndUpdate
    // TODO 2: Return most restrictive result across all matching rules

    return nil, fmt.Errorf("not implemented")
}

// Reset clears rate limit counters for a request
func (d *DistributedLimiter) Reset(ctx context.Context, req types.RateLimitRequest) error {
    // TODO 1: Get matching rules for the request
    // TODO 2: For each rule, generate Redis key and call storage.Reset
    // TODO 3: Handle partial failures gracefully

    return fmt.Errorf("not implemented")
}
```

Configuration Constants

`internal/storage/redis/constants.go`:

```go
package redis

import "time"

const (
    // DEFAULT_POOL_SIZE maximum Redis connections per instance
    DEFAULT_POOL_SIZE = 10

    // DEFAULT_TIMEOUT for Redis operations
    DEFAULT_TIMEOUT = 5 * time.Millisecond

    // Circuit breaker thresholds
    MAX_FAILURE_COUNT = 5
    CIRCUIT_OPEN_TIME = 30 * time.Second
)
```

`internal/config/constants.go`:

```go
package config

const (
    // Priority levels for rule evaluation order
    PRIORITY_HIGH = 100
    PRIORITY_MEDIUM = 50
    PRIORITY_LOW = 1

    // Algorithm identifiers
    ALGORITHM_TOKEN_BUCKET = "token_bucket"
    ALGORITHM_SLIDING_WINDOW_LOG = "sliding_window_log"
    ALGORITHM_SLIDING_COUNTER = "sliding_window_counter"
)
```

Language-Specific Implementation Hints

Redis Lua Script Development

When implementing atomic operations, Redis Lua scripts ensure consistency across distributed instances:

- Use `redis.call('TIME')` within Lua scripts to get consistent timestamps
- Store nanosecond timestamps as integers to avoid floating point precision issues
- Return multiple values from Lua scripts to minimize round-trips: `return {allowed, remaining, reset_time}`
- Test Lua scripts thoroughly - Redis script errors are harder to debug than Go code

Go Context and Timeout Handling

- Always pass context with timeouts to Redis operations: `ctx, cancel := context.WithTimeout(parent, 50*time.Millisecond)`
- Use `context.WithCancel` for health checking goroutines to enable graceful shutdown

- Check `ctx.Done()` in long-running operations to respect cancellation
- Wrap Redis errors with `fmt.Errorf("redis operation failed: %w", err)` for better error tracking

Concurrent Map Access

The rule manager needs thread-safe access patterns:

- Use `sync.RWMutex` for rule maps - most operations are reads
- Hold read locks for the minimum time necessary: get rule list, release lock, then process
- Use `sync.Map` for metrics counters that are updated frequently from multiple goroutines
- Avoid nested locking - always acquire locks in the same order to prevent deadlocks

Milestone Checkpoints

After completing the architecture setup:

1. **Connectivity Test**: Run `go test ./internal/storage/redis/...` - should successfully connect to Redis and perform basic operations
2. **Rule Loading Test**: Create a test YAML file with sample rules and verify `RuleManager.LoadRules()` parses correctly
3. **Interface Compliance**: Run `go build ./...` - all interface implementations should compile without errors
4. **Health Check Test**: Start Redis, verify health status is `true`, stop Redis, verify it changes to `false` within 10 seconds

Expected behavior verification:

- `RedisStorage.IsHealthy()` should return `true` when Redis is running
- Rule manager should load rules from YAML and match patterns correctly
- DistributedLimiter creation should not panic when given valid storage and rule manager

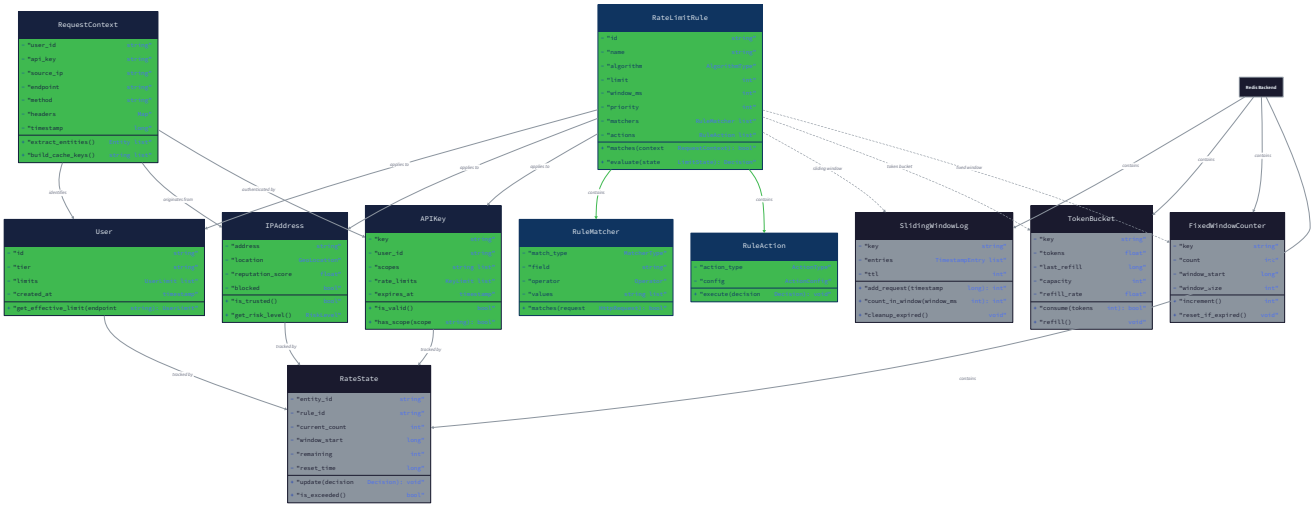
Common setup issues:

- Redis connection failures: Check Redis is running on expected port with `redis-cli ping`
- Module import errors: Ensure `go mod tidy` has been run and all dependencies are available
- Configuration validation errors: Verify YAML syntax and required fields are present

Data Model

> **Milestone(s):** All milestones - this data model foundation enables rate limiting algorithms (Milestone 1), multi-tier evaluation (Milestone 2), Redis backend integration (Milestone 3), sharding (Milestone 4), and API design (Milestone 5).

The data model serves as the foundation for distributed rate limiting by defining how rate limit rules are configured, how state is tracked across multiple application instances, and how metrics are collected for monitoring and alerting. Think of this as the blueprint that architects use before constructing a building - every component needs to understand the same data structures to work together effectively.



The data model addresses three critical challenges in distributed rate limiting. First, it must represent complex rate limit rules that can apply to multiple dimensions (user, IP, API endpoint) with different algorithms and priorities. Second, it must define how rate limiting state is stored in Redis with atomic operations to prevent race conditions across multiple application instances. Third, it must capture metrics and monitoring data that provide visibility into system behavior and enable proactive capacity management.

Rate Limit Rule Structure

Rate limit rules define the policies that govern request admission across different dimensions and tiers. Think of rate limit rules as the security protocols at different checkpoints in an airport - each checkpoint has specific criteria (first class vs economy, domestic vs international) and enforcement mechanisms (document check, metal detector, baggage scan) that determine whether a passenger can proceed.

The rule structure must support multi-dimensional matching, where a single HTTP request might be subject to multiple overlapping rules based on user identity, source IP address, API endpoint, and global system capacity. This creates a hierarchical evaluation system where rules are applied in priority order, with higher priority rules taking precedence over lower priority ones.

> **Decision: Unified Rule Structure with Pattern Matching**
>
> - **Context**: Rate limits need to apply across multiple dimensions (user, IP, API) with different algorithms and priorities
> - **Options Considered**: Separate rule types per dimension vs unified rule structure vs hardcoded tiers
> - **Decision**: Single unified `RateLimitRule` structure with pattern-based key matching
> - **Rationale**: Unified structure reduces complexity, enables flexible rule composition, and simplifies rule management API
> - **Consequences**: More flexible but requires careful pattern design to avoid rule conflicts and performance issues

The `RateLimitRule` structure captures all necessary configuration for rate limit enforcement:

Field	Type	Description
`id`	string	Unique identifier for the rule, used for updates and deletion
`name`	string	Human-readable name for the rule, displayed in dashboard and logs
`key_pattern`	string	Pattern for matching rate limit keys (supports wildcards and templating)
`algorithm`	string	Rate limiting algorithm identifier (`token_bucket` , `sliding_window_counter` , `sliding_window_log`)
`limit`	int64	Maximum number of requests allowed within the time window
`window`	time.Duration	Time window duration for rate limit evaluation
`burst_limit`	int64	Maximum burst capacity for token bucket algorithm (ignored for other algorithms)
`enabled`	bool	Whether this rule is actively enforced
`priority`	int	Rule evaluation priority (higher numbers evaluated first)
`created_at`	time.Time	Timestamp when rule was created
`updated_at`	time.Time	Timestamp when rule was last modified

The `key_pattern` field uses a templating system that allows rules to match multiple rate limit dimensions. For example, a pattern like `user:{user_id}:api:/v1/upload` creates rate limit keys specific to both user identity and API endpoint. This enables fine-grained control where a user might have different rate limits for different API operations.

> **Critical Design Insight**: The pattern-based approach allows a single rule to generate different Redis keys based on request context, enabling both specific limits (per-user API limits) and aggregate limits (global API limits) using the same rule structure.

The `priority` field determines rule evaluation order during multi-tier rate limiting. Rules with higher priority values are evaluated first, allowing system-wide protections (priority 100) to take precedence over user-specific limits (priority 50). This prevents lower-priority rules from consuming system resources when higher-priority limits are already exceeded.

Common Rule Configuration Patterns:

Pattern Type	Key Pattern	Example Usage
Per-User Global	`user:{user_id}`	1000 requests per hour per user across all APIs
Per-User API	`user:{user_id}:api:{api_endpoint}`	100 requests per minute for upload API per user
Per-IP	`ip:{ip_address}`	500 requests per hour per IP address
Global API	`api:{api_endpoint}`	10000 requests per minute for upload API across all users
Global System	`global`	100000 requests per minute system-wide

Redis Data Structures

Redis serves as the distributed state store for rate limiting counters and algorithm-specific state. The data structures must support atomic check-and-update operations to prevent race conditions when multiple application instances evaluate the same rate limit key simultaneously. Think of Redis as a high-speed synchronized ledger that multiple bank tellers can access simultaneously, with built-in mechanisms to ensure no two tellers can modify the same account balance at exactly the same time.

> **Decision: Algorithm-Specific Redis Key Schemas**
>
> - **Context**: Different rate limiting algorithms require different state representations in Redis
> - **Options Considered**: Unified state format vs algorithm-specific schemas vs separate Redis databases
> - **Decision**: Algorithm-specific key schemas with consistent naming conventions
> - **Rationale**: Optimizes Redis operations for each algorithm while maintaining operational simplicity
> - **Consequences**: More complex key management but better performance and clearer debugging

Each rate limiting algorithm requires different data structures in Redis to maintain its state effectively:

Token Bucket Algorithm Redis Schema:

Key Format	Data Type	Fields	Description
`tb:{rule_id}:{key}:state`	Redis Hash	`tokens`, `last_refill_time`	Current token count and last refill timestamp
`tb:{rule_id}:{key}:config`	Redis Hash	`capacity`, `refill_rate`, `window`	Algorithm configuration parameters

The token bucket state uses a Redis hash to store both the current token count and the timestamp of the last refill operation. This enables the atomic Lua script to calculate how many tokens should be added based on elapsed time and then update both the token count and timestamp in a single atomic operation.

Sliding Window Counter Algorithm Redis Schema:

Key Format	Data Type	Fields	Description
`swc:{rule_id}:{key}:{bucket_id}`	Redis String	counter value	Request count for specific time bucket
`swc:{rule_id}:{key}:meta`	Redis Hash	`window_size`, `bucket_size`, `last_cleanup`	Algorithm parameters and maintenance info

The sliding window counter divides the time window into smaller buckets, with each bucket stored as a separate Redis key. This allows expired buckets to be cleaned up automatically using Redis TTL, while active buckets can be summed to get the current window count. The bucket ID is calculated as `floor(current_timestamp / bucket_size)`.

Sliding Window Log Algorithm Redis Schema:

Key Format	Data Type	Fields	Description
`swl:{rule_id}:{key}:log`	Redis Sorted Set	score=timestamp, member=request_id	Individual request timestamps
`swl:{rule_id}:{key}:count`	Redis String	current count	Cached count for performance optimization

The sliding window log uses a Redis sorted set where the score is the request timestamp and the member is a unique request identifier. This allows efficient range queries to count requests within the current time window and automatic cleanup of expired entries using `ZREMRANGEBYSCORE`.

Cross-Algorithm Metadata Schema:

Key Format	Data Type	Fields	Description
`rl:rules:{rule_id}`	Redis Hash	serialized `RateLimitRule`	Rule configuration for distributed access
`rl:metrics:{key}:{timestamp}`	Redis Hash	`requests`, `allowed`, `denied`, `algorithm`	Aggregated metrics for monitoring
`rl:health:{node_id}`	Redis String	timestamp	Node health heartbeat

The metadata keys enable configuration distribution and health monitoring across the distributed rate limiting system. Rule configurations are cached in Redis so that any application instance can access the current rule set without requiring a central configuration service.

Redis Key Expiration Strategy

All rate limiting keys use Redis TTL (Time To Live) to prevent memory leaks and automatically clean up obsolete state. The TTL values are set based on the rate limit window duration plus a safety buffer:

Algorithm	TTL Formula	Rationale
Token Bucket	`2 * window_duration`	Allows for clock skew and delayed cleanup
Sliding Window Counter	`bucket_duration + window_duration`	Ensures all buckets in window remain available
Sliding Window Log	`window_duration + 1 minute`	Prevents log growth while handling delayed requests

Metrics and Monitoring Data

Comprehensive metrics collection enables operators to understand system behavior, detect anomalies, and plan capacity upgrades. The metrics system captures both real-time operational data and historical trends for analysis. Think of this as the instrument panel in an aircraft cockpit - pilots need both immediate readings (altitude, speed) and trend information (fuel consumption over time) to make informed decisions.

The metrics data model supports multiple aggregation levels to balance storage efficiency with query flexibility. Raw metrics are collected at high granularity for immediate operational needs, while aggregated metrics provide efficient historical analysis.

Real-Time Metrics Structure:

Field	Type	Description
`timestamp`	int64	Unix timestamp in milliseconds for precise ordering
`rule_id`	string	Rate limit rule that generated this metric
`key`	string	Rate limit key (may be hashed for privacy)
`algorithm`	string	Rate limiting algorithm used
`requests`	int64	Number of requests evaluated
`allowed`	int64	Number of requests allowed
`denied`	int64	Number of requests denied
`remaining_quota`	int64	Remaining capacity in current window
`window_reset_time`	int64	When current window resets (Unix timestamp)
`node_id`	string	Application instance that recorded this metric

Aggregated Metrics Structure:

Field	Type	Description
`time_bucket`	string	Time bucket identifier (e.g., "2024-01-15T14:30:00Z")
`bucket_size`	string	Aggregation interval ("1m", "5m", "1h", "1d")
`dimensions`	map[string]string	Aggregation dimensions (rule_id, algorithm, etc.)
`total_requests`	int64	Sum of all requests in time bucket
`total_allowed`	int64	Sum of allowed requests
`total_denied`	int64	Sum of denied requests
`avg_remaining_quota`	float64	Average remaining quota across measurements
`min_remaining_quota`	int64	Minimum remaining quota observed
`max_remaining_quota`	int64	Maximum remaining quota observed
`unique_keys`	int64	Number of distinct rate limit keys active

System Health Metrics Structure:

Field	Type	Description
`node_id`	string	Application instance identifier
`timestamp`	int64	Metric collection timestamp
`redis_latency_p50`	float64	50th percentile Redis operation latency (milliseconds)
`redis_latency_p99`	float64	99th percentile Redis operation latency (milliseconds)
`redis_errors`	int64	Number of Redis operation errors
`local_fallback_active`	bool	Whether local fallback is currently active
`rules_loaded`	int64	Number of rate limit rules currently loaded
`active_keys`	int64	Number of rate limit keys with recent activity
`memory_usage_bytes`	int64	Application memory usage
`cpu_usage_percent`	float64	Application CPU usage percentage

Decision: Hierarchical Metric Aggregation

- **Context**: Need both real-time monitoring and historical trend analysis with storage efficiency
- **Options Considered**: Store only raw metrics vs pre-aggregate everything vs hierarchical aggregation
- **Decision**: Hierarchical aggregation with multiple time granularities
- **Rationale**: Balances query performance, storage efficiency, and operational flexibility
- **Consequences**: More complex metric processing but enables both real-time dashboards and historical analysis

Common Pitfalls

⚠️ **Pitfall: Redis Key Explosion** When designing Redis key schemas, it's tempting to create highly granular keys for every possible combination of dimensions. However, this can lead to millions of keys in Redis, consuming excessive memory and degrading performance. Instead, use consistent key patterns with appropriate TTL values and consider using hashed key suffixes for high-cardinality dimensions like user IDs.

⚠️ **Pitfall: Inconsistent Rule Priority Handling** Rule priorities must be consistently interpreted across all application instances. A common mistake is using different priority comparison logic (ascending vs descending) in different components, leading to inconsistent rate limit enforcement. Always document priority semantics clearly and use constants like `PRIORITY_HIGH` and `PRIORITY_LOW` to make ordering explicit.

⚠️ **Pitfall: Missing Rule Validation** Rate limit rules should be validated when loaded to prevent runtime errors. Common validation failures include negative limits, zero or negative time windows, invalid algorithm names, and circular rule dependencies. Implement comprehensive validation with clear error messages to prevent misconfigured rules from reaching production.

⚠️ **Pitfall: Metric Storage Overflow** Without proper aggregation and cleanup, metrics can consume more storage than the actual application data. Implement time-based partitioning for metrics storage and automatically delete old partitions. Use sampling for high-frequency metrics and focus detailed collection on anomalous events.

Implementation Guidance

This implementation guidance provides the concrete data structures and Redis integration patterns needed to build the distributed rate limiting data model.

Technology Recommendations

Component	Simple Option	Advanced Option
Configuration Format	YAML files with validation	etcd/Consul with dynamic updates
Redis Client	`go-redis/redis` with connection pooling	`go-redis/redis` with cluster support
Metrics Storage	In-memory aggregation with periodic export	ClickHouse/InfluxDB for time series
Rule Validation	JSON Schema validation	Custom validation with dependency analysis

Recommended Module Structure

```
internal/
  config/
    rule.go              ← RateLimitRule definition and validation
    loader.go            ← Rule loading from YAML/JSON
    manager.go           ← RuleManager with pattern matching
  storage/
    redis.go             ← RedisStorage implementation
    interface.go         ← Storage interface definition
    lua_scripts.go       ← Embedded Lua scripts for atomic operations
  metrics/
    collector.go         ← Metrics collection and aggregation
    types.go             ← Metric data structures
    exporter.go          ← Export metrics to external systems
  models/
    types.go             ← Core data type definitions
```

Core Data Structure Definitions

```go
package models

import (
    "time"
    "github.com/go-redis/redis/v8"
)

// RedisConfig holds Redis connection and behavior configuration
type RedisConfig struct {
    Addresses    []string      `yaml:"addresses" json:"addresses"`
    Password     string        `yaml:"password" json:"password"`
    DB           int           `yaml:"db" json:"db"`
    PoolSize     int           `yaml:"pool_size" json:"pool_size"`
    ReadTimeout  time.Duration `yaml:"read_timeout" json:"read_timeout"`
    WriteTimeout time.Duration `yaml:"write_timeout" json:"write_timeout"`
    DialTimeout  time.Duration `yaml:"dial_timeout" json:"dial_timeout"`
}

// RedisStorage provides Redis-backed rate limiting state storage
type RedisStorage struct {
    client redis.UniversalClient
    config RedisConfig
}

// RateLimitRule defines a rate limiting policy with matching criteria and limits
type RateLimitRule struct {
    ID          string        `yaml:"id" json:"id"`
    Name        string        `yaml:"name" json:"name"`
    KeyPattern  string        `yaml:"key_pattern" json:"key_pattern"`
    Algorithm   string        `yaml:"algorithm" json:"algorithm"`
    Limit       int64         `yaml:"limit" json:"limit"`
    Window      time.Duration `yaml:"window" json:"window"`
    BurstLimit  int64         `yaml:"burst_limit" json:"burst_limit"`
    Enabled     bool          `yaml:"enabled" json:"enabled"`
    Priority    int           `yaml:"priority" json:"priority"`
    CreatedAt   time.Time     `yaml:"created_at" json:"created_at"`
    UpdatedAt   time.Time     `yaml:"updated_at" json:"updated_at"`
}
```

```go
// RuleManager manages rate limit rules with pattern matching and priority sorting
type RuleManager struct {
    rules map[string]*RateLimitRule
}

// RateLimitRequest contains context information for rate limit evaluation
type RateLimitRequest struct {
    UserID      string `json:"user_id"`
    IPAddress   string `json:"ip_address"`
    APIEndpoint string `json:"api_endpoint"`
    UserAgent   string `json:"user_agent"`
    Tokens      int64  `json:"tokens"`
}

// RateLimitResult contains the outcome of rate limit evaluation
type RateLimitResult struct {
    Allowed    bool          `json:"allowed"`
    Remaining  int64         `json:"remaining"`
    RetryAfter time.Duration `json:"retry_after"`
    ResetTime  time.Time     `json:"reset_time"`
    RuleID     string        `json:"rule_id"`
    Algorithm  string        `json:"algorithm"`
}
```

Redis Storage Implementation Skeleton

```go
package storage


// NewRedisStorage creates a Redis storage instance with connection pooling

func NewRedisStorage(config RedisConfig) (*RedisStorage, error) {

    // TODO 1: Validate config parameters (addresses not empty, positive timeouts)

    // TODO 2: Create Redis universal client with cluster support

    // TODO 3: Test connection with PING command

    // TODO 4: Load Lua scripts into Redis for atomic operations

    // TODO 5: Return configured RedisStorage instance

}


// CheckAndUpdate atomically checks current usage and updates counters

func (r *RedisStorage) CheckAndUpdate(ctx context.Context, key string, limit int64, window time.Duration) (bool, int64,
time.Time, error) {

    // TODO 1: Determine algorithm from key prefix

    // TODO 2: Select appropriate Lua script for algorithm

    // TODO 3: Prepare script arguments (key, limit, window, current_timestamp)

    // TODO 4: Execute Lua script atomically in Redis

    // TODO 5: Parse script response (allowed, remaining, reset_time)

    // TODO 6: Handle Redis errors with circuit breaker pattern

    // TODO 7: Return rate limit decision with metadata

}
```

Rule Management Implementation Skeleton

```go
package config


// LoadRules loads rate limit rules from YAML configuration file
func (rm *RuleManager) LoadRules(configPath string) error {
    // TODO 1: Read YAML file from configPath
    // TODO 2: Parse YAML into slice of RateLimitRule structs
    // TODO 3: Validate each rule (positive limits, valid algorithms, pattern syntax)
    // TODO 4: Check for rule ID conflicts
    // TODO 5: Sort rules by priority (highest first)
    // TODO 6: Update rm.rules map atomically
    // TODO 7: Log successful rule loading with count
}

// GetMatchingRules returns rules that match the request context
func (rm *RuleManager) GetMatchingRules(userID, ipAddress, apiEndpoint string) []*RateLimitRule {
    // TODO 1: Create request context map with userID, ipAddress, apiEndpoint
    // TODO 2: Iterate through rules in priority order
    // TODO 3: For each rule, expand key_pattern template with request context
    // TODO 4: Check if expanded pattern matches request (wildcards, exact match)
    // TODO 5: Add matching rules to result slice
    // TODO 6: Return rules sorted by priority (highest first)
    // Hint: Use text/template package for pattern expansion
}
```

Metrics Collection Implementation Skeleton

```go
package metrics


// CollectMetrics gathers rate limiting metrics for monitoring and alerting
func (c *MetricsCollector) CollectMetrics(result *RateLimitResult, duration time.Duration) {
    // TODO 1: Extract timestamp and dimensions from result
    // TODO 2: Create RealTimeMetric struct with current values
    // TODO 3: Add to in-memory buffer with timestamp-based key
    // TODO 4: Update running counters (total requests, allowed, denied)
    // TODO 5: Check if aggregation interval has passed
    // TODO 6: If interval passed, compute aggregated metrics and export
    // TODO 7: Clean up old metrics from memory buffer
}
```

Configuration File Example

```yaml
# Example rate limit rules configuration

rules:

  - id: "user-global"

    name: "Per-User Global Limit"

    key_pattern: "user:{user_id}"

    algorithm: "token_bucket"

    limit: 1000

    window: "1h"

    burst_limit: 50

    enabled: true

    priority: 50


  - id: "upload-api"

    name: "Upload API Limit"

    key_pattern: "api:/v1/upload"

    algorithm: "sliding_window_counter"

    limit: 10000

    window: "1m"

    enabled: true

    priority: 100


  - id: "ip-limit"

    name: "Per-IP Address Limit"

    key_pattern: "ip:{ip_address}"

    algorithm: "sliding_window_log"

    limit: 500

    window: "1h"

    enabled: true

    priority: 75
```

Milestone Checkpoints

After implementing data structures:

- Run `go test ./internal/models/...` - all type definitions should compile without errors
- Test rule validation with invalid configurations - should return specific error messages
- Verify Redis key generation matches expected patterns for each algorithm

After implementing Redis storage:

- Run `go test ./internal/storage/...` - Redis operations should be atomic and consistent
- Test with Redis cluster to verify consistent hashing works correctly
- Verify Lua scripts execute atomically even under high concurrency

After implementing rule management:

- Load sample configuration file - should parse all rules and sort by priority
- Test pattern matching with various request contexts - should match correct rules
- Verify rule updates propagate to all application instances within configured interval

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Rules not loading	YAML syntax error	Check application logs for parsing errors	Validate YAML syntax and required fields
Redis keys growing infinitely	Missing TTL on keys	Use `redis-cli` to check key expiration	Set appropriate TTL in Lua scripts
Inconsistent rate limiting	Clock skew between nodes	Compare timestamps in Redis vs application logs	Use Redis TIME command for consistent timestamps
High memory usage	Too many metric data points	Monitor metrics buffer size	Implement time-based cleanup and aggregation
Pattern matching fails	Template expansion error	Log expanded patterns vs expected keys	Debug template syntax and variable names

Rate Limiting Algorithms

> **Milestone(s):** Milestone 1 - Rate Limiting Algorithms

The heart of any distributed rate limiting system lies in its algorithms for tracking and enforcing request quotas over time. While the distributed coordination and Redis integration add complexity, the fundamental challenge remains: how do we accurately measure request rates and make allow/deny decisions in real-time? This section explores three complementary algorithms that form the foundation of our rate limiting system, each with distinct characteristics that make them suitable for different use cases and performance requirements.

Mental Model: Water Flow Control Systems

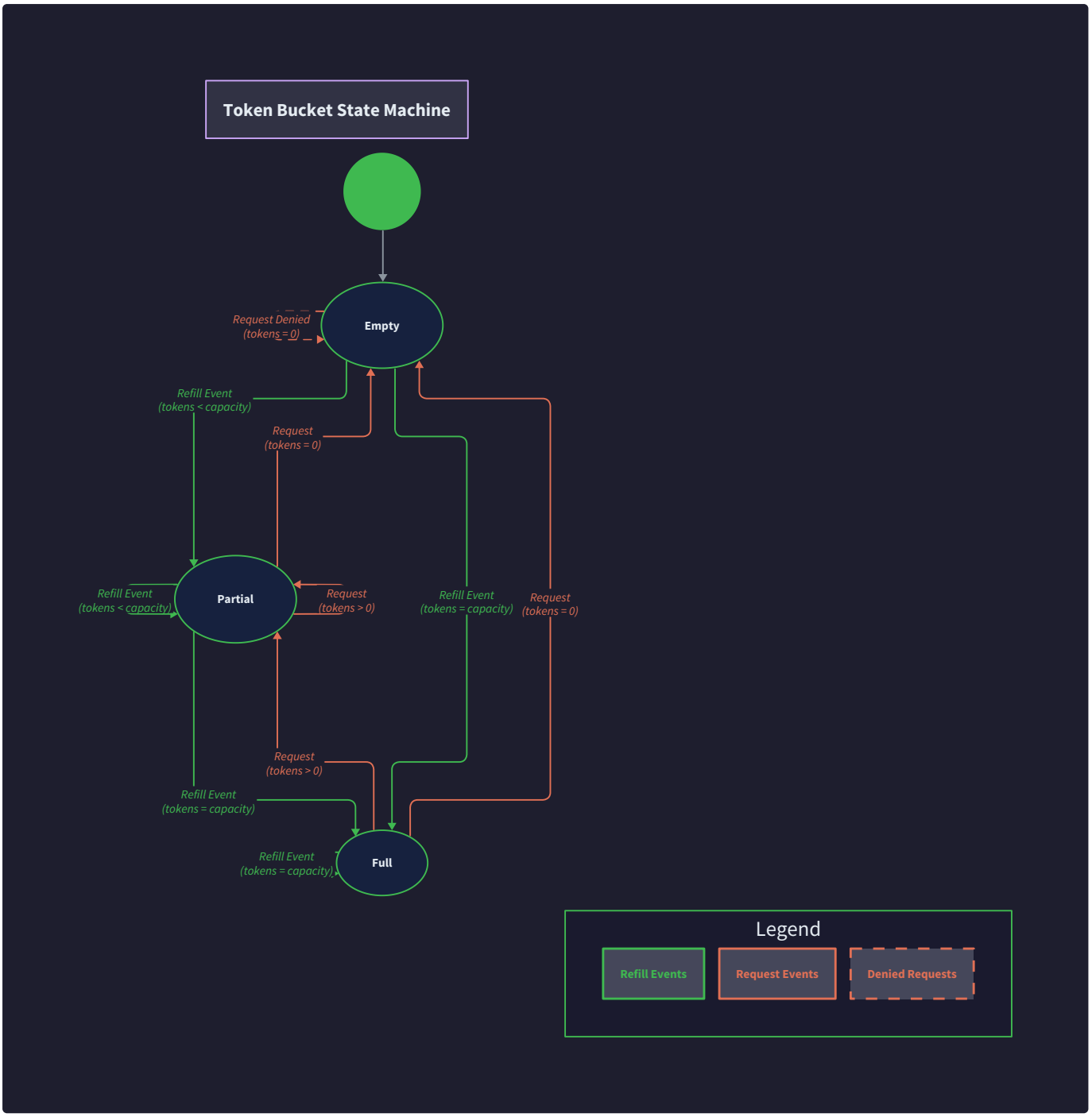
Before diving into the technical details of rate limiting algorithms, it's helpful to think about water flow control systems that we encounter in everyday life. Each rate limiting algorithm maps naturally to a different type of water control mechanism, helping us understand their behaviors and trade-offs intuitively.

Token Bucket as a Water Storage Tank: Imagine a water storage tank with a fixed capacity that receives water at a steady rate from a source pipe. When you need water (make a request), you can take it instantly from the tank if water is available. The key insight is that the tank allows you to consume water faster than the refill rate for short periods - you can drain the entire tank quickly if needed, but then you must wait for it to refill at the steady rate. This mirrors how token bucket allows controlled bursts above the sustained rate while preventing long-term overconsumption.

Sliding Window Counter as a Dam with Spillways: Picture a dam with multiple spillway gates that open and close on a schedule. Every minute, one gate opens to release exactly 1000 gallons, while simultaneously another gate closes. You're measuring the total flow by looking at all currently open gates. This approximates the flow rate over the last hour by dividing it into discrete time buckets. It's memory-efficient (you only track a few bucket counts) but imprecise at bucket boundaries - if all gates opened at the start of their window, you might see 2000 gallons flow in one minute even though the average is 1000 per minute.

Sliding Window Log as Individual Raindrop Tracking: Imagine tracking every individual raindrop that falls on your roof, recording the exact timestamp of each drop. To know the rainfall rate in the last hour, you count all drops within the 60-minute window ending right now. This gives you perfect accuracy - you know exactly how many drops fell in any time period. However, during a heavy downpour, you must remember millions of individual timestamps, making this approach memory-intensive but perfectly precise.

These water analogies capture the essential trade-offs: token bucket provides controlled bursting, sliding window counter offers memory efficiency with some accuracy loss, and sliding window log delivers perfect accuracy at the cost of memory usage. Understanding these mental models helps us choose the right algorithm for different scenarios in our distributed rate limiting system.



Token Bucket Algorithm Design

The **token bucket algorithm** serves as our primary rate limiting mechanism because it naturally handles the most common real-world requirement: allowing controlled bursts while maintaining a sustainable long-term rate. This algorithm maintains a bucket that holds a fixed number of tokens, with new tokens added at a steady rate. Each request consumes one or more tokens, and requests are denied when insufficient tokens remain.

The algorithm operates on two fundamental parameters that define its behavior. The **bucket capacity** determines the maximum burst size - how many requests can be processed instantly when the system has been idle. The **refill rate** controls the sustained throughput - how many requests per second the system allows over extended periods. These parameters work together to provide both responsiveness and protection.

Token Bucket State Management

Our token bucket implementation maintains state through a `TokenBucketState` structure that tracks the current token count and the timestamp of the last refill operation. This design enables efficient state updates by calculating how many tokens should be added based on elapsed time since the last refill, avoiding the need for background processes or timers.

Field	Type	Description
`tokens`	`int64`	Current number of tokens available in the bucket
`last_refill_time`	`int64`	Unix timestamp in nanoseconds of the last refill calculation

The state transitions follow a predictable pattern that maps to our water storage tank analogy. When the bucket is **full**, incoming tokens are discarded (the tank overflows), and requests are immediately approved until tokens are exhausted. During **partial fill** states, each request decreases the token count, and the refill calculation determines how many tokens to add based on elapsed time. When **empty**, requests must wait for tokens to be replenished through the steady refill process.

Atomic Token Bucket Operations

The core challenge in distributed token bucket implementation lies in ensuring atomic check-and-update operations. In a single-threaded environment, checking the current token count and updating it can be separate operations. However, in a distributed system with multiple application instances, we must prevent race conditions where two instances simultaneously check the same token count and both approve requests that should collectively exceed the limit.

Our solution employs Redis Lua scripts to guarantee atomicity. The script performs the following operations as a single indivisible unit:

1. **Calculate elapsed time** since the last refill by comparing the current timestamp with `last_refill_time` stored in Redis
2. **Compute tokens to add** by multiplying elapsed time by the refill rate, respecting the maximum bucket capacity
3. **Update the token count** with newly calculated tokens, ensuring it never exceeds the configured capacity
4. **Check token sufficiency** by comparing the requested token amount with the available tokens
5. **Deduct tokens if sufficient** and update both the token count and last refill timestamp
6. **Return the decision** along with remaining tokens and time until next token availability

This atomic operation prevents the classic race condition where multiple instances read the same token count, each conclude sufficient tokens exist, and all approve requests that collectively exceed the limit.

Burst Handling and Refill Logic

The token bucket's burst handling capability distinguishes it from simpler rate limiting approaches. When a system has been idle, the bucket accumulates tokens up to its full capacity, enabling it to handle sudden traffic spikes without rejecting requests. This behavior closely mimics real-world usage patterns where traffic often arrives in bursts rather than smooth, evenly-distributed streams.

The refill logic implements a **continuous refill model** rather than periodic batch refills. Instead of adding tokens every second through a background process, we calculate the exact number of tokens that should have been added based on elapsed time whenever a rate limit check occurs. This approach eliminates the need for background workers and ensures consistent behavior regardless of request timing patterns.

For example, consider a token bucket configured with 100 tokens capacity and 50 tokens per second refill rate. If the bucket starts full and receives 100 requests instantly, all are approved and the bucket becomes empty. Subsequent requests must wait, but after 2 seconds of no activity, the bucket will have accumulated 100 tokens again (2 seconds × 50 tokens/second), ready for another burst.

Configuration and Tuning Parameters

Token bucket behavior is controlled through the `TokenBucketConfig` structure, which encapsulates all necessary parameters for proper operation:

Field	Type	Description
`capacity`	`int64`	Maximum number of tokens the bucket can hold (burst size)
`refill_rate`	`int64`	Number of tokens added per second (sustained rate)
`window`	`time.Duration`	Time window for rate calculations (typically 1 second)

The relationship between these parameters determines the algorithm's characteristics. A high capacity relative to refill rate creates a "bursty" system that can handle large traffic spikes but takes longer to recover. A low capacity relative to refill rate creates a "smooth" system that provides steady throughput with limited burst capability.

Token Bucket Implementation Architecture

The `TokenBucket` type encapsulates the algorithm logic and integrates with our distributed storage layer:

Method	Parameters	Returns	Description
`Check`	`ctx context.Context, key string, tokens int64`	`*RateLimitResult, error`	Atomically checks and updates token bucket state
`Preview`	`ctx context.Context, key string, tokens int64`	`*RateLimitResult, error`	Returns current state without consuming tokens
`Reset`	`ctx context.Context, key string`	`error`	Resets bucket to full capacity
`GetState`	`ctx context.Context, key string`	`*TokenBucketState, error`	Returns current bucket state for monitoring

The `Check` method serves as the primary interface for rate limiting decisions. It accepts a context for timeout control, a key identifying the specific rate limit bucket, and the number of tokens requested. The method returns a `RateLimitResult` containing the decision, remaining tokens, and timing information needed for proper HTTP response headers.

> **Design Insight**: Token bucket's strength lies in its intuitive burst behavior that matches real-world traffic patterns. Unlike algorithms that strictly enforce per-second limits, token bucket allows natural traffic spikes while preventing sustained overload. This makes it ideal for user-facing APIs where occasional bursts are acceptable but sustained abuse must be prevented.

Sliding Window Algorithms Design

While token bucket algorithms excel at handling burst traffic patterns, sliding window algorithms provide more predictable rate limiting behavior by measuring request rates over precise time windows. Our system implements two sliding window variants: **sliding window counter** for memory efficiency and **sliding window log** for perfect accuracy. Understanding their trade-offs helps us select the appropriate algorithm based on specific rate limiting requirements.

Sliding Window Counter Algorithm

The sliding window counter algorithm divides time into fixed-size buckets and maintains a count of requests in each bucket. To determine the current rate, it examines all buckets within the sliding window and aggregates their counts. This approach provides memory efficiency by storing only bucket counts rather than individual request timestamps, making it suitable for high-traffic scenarios where memory usage must be controlled.

Our implementation maintains a configurable number of sub-windows within each rate limiting window. For example, a 60-second rate limit window might be divided into 12 sub-windows of 5 seconds each. This granularity affects both memory usage and accuracy - more sub-windows provide better accuracy but consume more memory.

The algorithm tracks the following data for each rate limit key:

Field	Type	Description
`window_start`	`int64`	Unix timestamp of the current window start time
`bucket_counts`	`[]int64`	Array of request counts for each sub-window bucket
`bucket_timestamps`	`[]int64`	Array of timestamps for each bucket to handle expiration
`total_count`	`int64`	Cached total count across all buckets for efficiency

When processing a rate limit check, the algorithm follows these steps:

1. **Calculate current bucket index** based on the current timestamp and bucket duration
2. **Expire old buckets** by zeroing counts for buckets outside the sliding window
3. **Update current bucket** by incrementing the count for the active time bucket
4. **Aggregate total count** across all non-expired buckets within the window
5. **Compare against limit** and return the appropriate decision with remaining quota

The key advantage of this approach lies in its **bounded memory usage**. Regardless of request volume, each rate limit key consumes memory proportional to the number of sub-windows rather than the number of requests. A system processing millions of requests per second still only stores a few dozen bucket counts per rate limit key.

However, this efficiency comes with an accuracy trade-off known as the **boundary condition problem**. Consider a rate limit of 1000 requests per minute using 5-second buckets. If 1000 requests arrive in the first second of minute 1 and another 1000 requests arrive in the first second of minute 2, the

algorithm might allow both bursts because they fall in different minute-windows, even though 2000 requests occurred within a 61-second period.

Sliding Window Log Algorithm

The sliding window log algorithm eliminates the boundary condition problem by maintaining exact timestamps for each request within the sliding window. When evaluating a rate limit, it counts all requests that occurred within the precise time window ending at the current moment, providing perfect accuracy regardless of request timing patterns.

This algorithm stores request timestamps in a time-ordered data structure:

Field	Type	Description
`request_timestamps`	`[]int64`	Sorted array of Unix timestamps for requests in the window
`window_duration`	`int64`	Duration of the sliding window in nanoseconds
`last_cleanup`	`int64`	Timestamp of last cleanup operation to remove expired entries

The request evaluation process provides exact rate calculations:

1. **Remove expired timestamps** older than the current window by scanning from the beginning of the array
2. **Count remaining timestamps** to determine current request count within the window
3. **Compare against rate limit** to make the allow/deny decision
4. **Add current timestamp** if the request is allowed
5. **Return result** with exact remaining quota and reset time

This precision comes at a significant memory cost. During traffic spikes, the algorithm must store individual timestamps for every request within the window duration. A system allowing 10,000 requests per minute must maintain up to 10,000 timestamps in memory per rate limit key. For applications with thousands of rate-limited keys, this memory usage can become prohibitive.

Algorithm Accuracy Comparison

To understand the practical implications of each algorithm's accuracy characteristics, consider how they handle edge cases:

Scenario	Token Bucket Behavior	Sliding Window Counter	Sliding Window Log
Burst at window boundary	Allows up to full capacity instantly	May allow up to 2× limit at boundaries	Perfect accuracy, never exceeds limit
Steady traffic	Enforces long-term average rate	Enforces average with small variations	Enforces exact rate continuously
Irregular patterns	Smooths bursts through token accumulation	Approximates rate with bucket granularity	Measures exact rate regardless of pattern
Memory usage	O(1) per key	O(buckets) per key	O(requests in window) per key
Computational cost	O(1) per request	O(1) per request	O(log n) per request for cleanup

Sliding Window Implementation Strategy

Both sliding window algorithms integrate with our Redis backend through specialized Lua scripts that ensure atomic operations. The scripts handle the complex logic of timestamp management, expiration, and count aggregation while maintaining consistency across multiple application instances.

For sliding window counter, the Lua script manages bucket rotation and count aggregation:

```lua
-- Sliding window counter operations happen atomically in Redis

-- 1. Calculate current bucket based on timestamp

-- 2. Expire old buckets outside the window

-- 3. Increment current bucket count

-- 4. Sum all active bucket counts

-- 5. Return decision and remaining quota
```

For sliding window log, the script manages timestamp arrays with careful memory cleanup:

```lua
-- Sliding window log operations happen atomically in Redis

-- 1. Remove timestamps older than window duration

-- 2. Count remaining timestamps

-- 3. Add new timestamp if under limit

-- 4. Return exact remaining quota
```

LUA

> **Performance Consideration**: Sliding window log's memory usage can grow large during traffic spikes, making it unsuitable for high-volume systems without careful memory management. However, its perfect accuracy makes it ideal for strict rate limiting scenarios where boundary conditions cannot be tolerated, such as payment processing or security-sensitive APIs.

Hybrid Sliding Window Approach

For scenarios requiring both memory efficiency and improved accuracy, our system supports a hybrid approach that combines sliding window counter granularity with boundary condition detection. This algorithm uses fine-grained buckets (1-second buckets for 60-second windows) and applies smoothing logic to reduce boundary effects.

The hybrid algorithm applies a **weighted calculation** that considers partial bucket contributions when evaluating rate limits near window boundaries. Instead of simply summing bucket counts, it calculates what portion of each bucket falls within the exact sliding window and weights the counts accordingly.

This approach provides significantly better accuracy than standard sliding window counter while maintaining bounded memory usage. The computational overhead increases slightly due to weighted calculations, but remains much more efficient than maintaining individual request timestamps.

Algorithm Selection ADR

Choosing the appropriate rate limiting algorithm significantly impacts system performance, accuracy, and operational characteristics. This decision affects memory usage, computational overhead, user experience, and the types of traffic patterns the system can handle effectively.

> **Decision: Multi-Algorithm Support with Configurable Selection**
>
> - **Context**: Different rate limiting scenarios have conflicting requirements. API endpoints serving user traffic need burst handling for good UX, while security-sensitive endpoints require strict accuracy. High-volume systems need memory efficiency, while low-volume critical systems can afford precision overhead.
> - **Options Considered**: Single algorithm for simplicity, token bucket only for burst handling, sliding window log only for accuracy, configurable algorithm selection per rule
> - **Decision**: Implement all three algorithms with per-rule configuration, defaulting to token bucket for most use cases
> - **Rationale**: Different endpoints have fundamentally different requirements that cannot be satisfied by a single algorithm. The additional complexity is justified by the flexibility to optimize each rate limit rule for its specific constraints and requirements.
> - **Consequences**: Increases implementation complexity and testing surface area, but enables optimal algorithm selection for each use case, improving both performance and user experience across different scenarios.

Algorithm Selection Decision Matrix

The choice of rate limiting algorithm should be based on specific requirements and constraints:

Algorithm	Memory Usage	Accuracy	Burst Handling	Best Use Cases
Token Bucket	O(1) per key	Good for sustained rate	Excellent - natural bursting	User-facing APIs, general rate limiting
Sliding Window Counter	O(buckets) per key	Good with boundary effects	Limited - depends on bucket size	High-volume systems, memory-constrained environments
Sliding Window Log	O(requests) per key	Perfect	None - strict enforcement	Security APIs, payment processing, strict SLA enforcement

Configuration Recommendations by Scenario

Based on common distributed system patterns, we recommend the following algorithm selections:

User-Facing API Endpoints should use token bucket with generous burst capacity. Users often interact with applications in bursts - opening a page might trigger multiple API calls simultaneously. Token bucket naturally accommodates this pattern while preventing sustained abuse.

Recommended configuration:

- Algorithm: `ALGORITHM_TOKEN_BUCKET`
- Capacity: 3-5× the sustained rate (allows reasonable bursts)
- Refill rate: Target sustained requests per second
- Example: 100 requests/minute with 20-token capacity allows 20 instant requests followed by ~1.67 requests/second

Security-Sensitive Endpoints should use sliding window log for perfect accuracy. Authentication attempts, password resets, and payment operations cannot tolerate the boundary condition issues that might allow double the intended rate limit.

Recommended configuration:

- Algorithm: `ALGORITHM_SLIDING_WINDOW_LOG`
- Window: Short duration to limit memory usage (5-15 minutes)
- Limit: Conservative values with no burst allowance
- Example: 5 login attempts per 15 minutes with exact enforcement

High-Volume Internal APIs should use sliding window counter with fine-grained buckets. Service-to-service communication often has predictable patterns and can tolerate slight accuracy variations in exchange for memory efficiency.

Recommended configuration:

- Algorithm: `ALGORITHM_SLIDING_COUNTER`
- Buckets: 10-20 sub-windows for reasonable accuracy
- Window: Match service SLA requirements
- Example: 10,000 requests/minute divided into 12 five-second buckets

Global Rate Limits protecting overall system capacity should use sliding window counter with coarse granularity. These limits serve as circuit breakers preventing total system overload, where approximate enforcement is acceptable for memory efficiency.

Algorithm Performance Characteristics

Understanding the computational and memory overhead of each algorithm helps in capacity planning:

Metric	Token Bucket	Sliding Window Counter	Sliding Window Log
Redis operations per check	1 Lua script execution	1 Lua script execution	1 Lua script execution
Memory per key (idle)	~32 bytes	~64 bytes + (8 × buckets)	~64 bytes
Memory per key (active)	~32 bytes	~64 bytes + (8 × buckets)	~64 bytes + (8 × requests in window)
CPU overhead	Minimal	Low	Moderate (timestamp cleanup)
Network overhead	Low	Low	Higher (larger data structures)

Migration Strategy Between Algorithms

Systems may need to change algorithms as requirements evolve. Our design supports algorithm migration through versioned rate limit rules:

1. **Gradual Rollout**: Deploy new rules with different algorithms alongside existing rules
2. **A/B Testing**: Route percentage of traffic to new algorithm for validation
3. **State Transition**: Both algorithms can coexist during migration periods
4. **Monitoring**: Compare accuracy, performance, and user experience metrics
5. **Cutover**: Switch traffic once new algorithm proves stable and effective

> **Common Pitfall**: Choosing sliding window log for high-volume endpoints without considering memory implications. A popular API endpoint processing 100,000 requests per minute with a 5-minute window could require storing 500,000 timestamps per rate limit key. With multiple rate limit dimensions (per-user, per-IP, global), memory usage can quickly become unsustainable.

The flexibility to select appropriate algorithms per rate limit rule enables optimization for diverse requirements within a single system, providing both the performance needed for high-volume endpoints and the accuracy required for security-sensitive operations.

Implementation Guidance

This implementation guidance focuses on building the core rate limiting algorithms that serve as the foundation for our distributed rate limiting system. The algorithms themselves are the primary learning objective, while Redis integration and HTTP handling serve as supporting infrastructure.

Technology Recommendations

Component	Simple Option	Advanced Option
Time Handling	`time.Now()` with Unix timestamps	High-resolution timestamps with `time.Now().UnixNano()`
Math Operations	Standard int64 arithmetic	`math` package for precision calculations
Configuration	Hard-coded constants	YAML configuration with validation
Testing	Basic unit tests	Property-based testing with random inputs

Recommended File Structure

```
internal/algorithms/
  token_bucket.go          ← Token bucket algorithm implementation
  token_bucket_test.go     ← Unit tests for token bucket
  sliding_window.go        ← Sliding window counter and log algorithms
  sliding_window_test.go   ← Unit tests for sliding windows
  types.go                 ← Common algorithm types and interfaces
  config.go                ← Algorithm configuration structures
  testdata/
    algorithm_configs.yaml ← Test configuration files
```

Infrastructure Starter Code

The following infrastructure code handles time management, configuration, and basic Redis operations, allowing you to focus on implementing the algorithm logic:

```go
package algorithms

import (
    "context"
    "time"
)

// Storage interface abstracts the underlying storage mechanism
// This allows algorithms to work with Redis, in-memory, or other backends
type Storage interface {
    // Lua script execution for atomic operations
    ExecuteLua(ctx context.Context, script string, keys []string, args []interface{}) (interface{}, error)

    // Simple get/set operations for non-atomic operations
    Get(ctx context.Context, key string) (string, error)
    Set(ctx context.Context, key string, value string, expiration time.Duration) error

    // Delete operations for cleanup
    Delete(ctx context.Context, keys ...string) error
}

// TimeProvider allows mocking time in tests
type TimeProvider interface {
    Now() time.Time
}

type RealTimeProvider struct{}

func (r RealTimeProvider) Now() time.Time {
    return time.Now()
}

// Configuration structures
type TokenBucketConfig struct {
    Capacity   int64         `yaml:"capacity"`
    RefillRate int64         `yaml:"refill_rate"`
    Window     time.Duration `yaml:"window"`
}

type SlidingWindowConfig struct {
    Limit      int64         `yaml:"limit"`
```

```
    Window     time.Duration `yaml:"window"`
    Buckets    int           `yaml:"buckets"`    // For counter algorithm
}

// Result structures
type RateLimitResult struct {
    Allowed     bool          `json:"allowed"`
    Remaining   int64         `json:"remaining"`
    RetryAfter  time.Duration `json:"retry_after"`
    ResetTime   time.Time     `json:"reset_time"`
    RuleID      string        `json:"rule_id"`
    Algorithm   string        `json:"algorithm"`
}

// State structures for Redis storage
type TokenBucketState struct {
    Tokens         int64 `json:"tokens"`
    LastRefillTime int64 `json:"last_refill_time"`
}
```

Core Algorithm Skeleton Code

Token Bucket Implementation Skeleton:

```go
package algorithms

import (
    "context"
    "encoding/json"
    "fmt"
    "math"
)

type TokenBucket struct {
    config       TokenBucketConfig
    storage      Storage
    timeProvider TimeProvider
}

func NewTokenBucket(config TokenBucketConfig, storage Storage) *TokenBucket {
    return &TokenBucket{
        config:       config,
        storage:      storage,
        timeProvider: RealTimeProvider{},
    }
}

// Check performs atomic token bucket rate limiting check
func (tb *TokenBucket) Check(ctx context.Context, key string, tokensRequested int64) (*RateLimitResult, error) {
    // TODO 1: Get current timestamp in nanoseconds using tb.timeProvider.Now().UnixNano()
    // TODO 2: Execute Redis Lua script with key, timestamp, and configuration parameters
    // TODO 3: Parse Lua script result containing allowed flag, remaining tokens, and next refill time
    // TODO 4: Build RateLimitResult with appropriate retry_after and reset_time values
    // TODO 5: Handle Redis errors by returning error for upstream handling


    // Lua script for atomic token bucket operations
    luaScript := `
        -- TODO 6: Parse input parameters (key timestamp, tokens_requested, capacity, refill_rate, window_ns)
        -- TODO 7: Get current state from Redis or initialize if doesn't exist
        -- TODO 8: Calculate tokens to add based on elapsed time since last refill
        -- TODO 9: Update token count (new_tokens = min(current + added, capacity))
        -- TODO 10: Check if sufficient tokens available for request
        -- TODO 11: If allowed, subtract requested tokens and update state in Redis
```

```
        -- TODO 12: Return result array with [allowed, remaining_tokens, next_refill_timestamp]
    `



    // Hint: Use tb.storage.ExecuteLua() with keys=[key] and args=[timestamp, tokensRequested, tb.config.Capacity,
tb.config.RefillRate, tb.config.Window.Nanoseconds()]

    // Hint: Redis returns []interface{} that you need to type assert to []interface{} then individual elements

    // Hint: Calculate retry_after as time until next token becomes available

}


// Preview checks current state without consuming tokens

func (tb *TokenBucket) Preview(ctx context.Context, key string) (*RateLimitResult, error) {

    // TODO 13: Similar to Check() but with tokensRequested = 0

    // TODO 14: Lua script should not subtract tokens, only return current state

    // Hint: Reuse most logic from Check() but don't modify state

}


// Reset clears the bucket state, effectively filling it to capacity

func (tb *TokenBucket) Reset(ctx context.Context, key string) error {

    // TODO 15: Delete the Redis key to reset state

    // TODO 16: Handle Redis errors appropriately

    // Hint: Use tb.storage.Delete(ctx, key)

}
```

Sliding Window Counter Implementation Skeleton:

```go
type SlidingWindowCounter struct {
    config       SlidingWindowConfig
    storage      Storage
    timeProvider TimeProvider
}

func NewSlidingWindowCounter(config SlidingWindowConfig, storage Storage) *SlidingWindowCounter {
    return &SlidingWindowCounter{
        config:       config,
        storage:      storage,
        timeProvider: RealTimeProvider{},
    }
}

// Check performs atomic sliding window counter rate limiting
func (swc *SlidingWindowCounter) Check(ctx context.Context, key string, increment int64) (*RateLimitResult, error) {
    // TODO 17: Calculate current timestamp and bucket duration
    // TODO 18: Execute Lua script for atomic bucket operations
    // TODO 19: Parse results and build RateLimitResult

    luaScript := `
        -- TODO 20: Calculate current bucket index from timestamp
        -- TODO 21: Load existing bucket counts from Redis hash
        -- TODO 22: Expire buckets outside the sliding window
        -- TODO 23: Increment current bucket count
        -- TODO 24: Sum all active bucket counts
        -- TODO 25: Compare total against limit and return result
    `

    // Hint: Use Redis hash to store bucket counts with bucket index as field name
    // Hint: Calculate bucket index as (timestamp / bucket_duration) % num_buckets
    // Hint: Window contains last N buckets, expire older ones
}
```

Sliding Window Log Implementation Skeleton:

```go
type SlidingWindowLog struct {
    config       SlidingWindowConfig
    storage      Storage
    timeProvider TimeProvider
}

// Check performs atomic sliding window log rate limiting
func (swl *SlidingWindowLog) Check(ctx context.Context, key string, increment int64) (*RateLimitResult, error) {
    // TODO 26: Get current timestamp for window calculations
    // TODO 27: Execute Lua script for atomic timestamp operations
    // TODO 28: Handle memory cleanup to prevent unbounded growth

    luaScript := `
        -- TODO 29: Load current timestamp list from Redis sorted set
        -- TODO 30: Remove timestamps older than window duration
        -- TODO 31: Count remaining timestamps in window
        -- TODO 32: If under limit, add current timestamp to set
        -- TODO 33: Return decision with exact remaining count
    `

    // Hint: Use Redis sorted set (ZSET) with timestamps as scores for efficient range operations
    // Hint: Use ZREMRANGEBYSCORE to remove old timestamps atomically
    // Hint: Use ZCARD to count current timestamps in set
}
```

Algorithm Factory and Selection Logic:

```go
// AlgorithmFactory creates algorithm instances based on configuration
type AlgorithmFactory struct {
    storage Storage
}

func (af *AlgorithmFactory) CreateAlgorithm(algorithmType string, config interface{}) (RateLimitingAlgorithm, error) {
    // TODO 34: Switch on algorithmType (ALGORITHM_TOKEN_BUCKET, etc.)
    // TODO 35: Cast config to appropriate type and validate parameters
    // TODO 36: Return appropriate algorithm instance or error
    // Hint: Use type switches for config casting: config.(*TokenBucketConfig)
}

// Common interface for all algorithms
type RateLimitingAlgorithm interface {
    Check(ctx context.Context, key string, tokens int64) (*RateLimitResult, error)
    Preview(ctx context.Context, key string) (*RateLimitResult, error)
    Reset(ctx context.Context, key string) error
}
```

Language-Specific Hints

- Use `time.Now().UnixNano()` for high-precision timestamps to avoid race conditions with multiple requests in the same millisecond
- Redis Lua scripts return `[]interface{}` that require type assertion: `result[0].(int64)`
- Handle Redis connection errors gracefully - rate limiting failures should not crash the application
- Use `math.Min()` for token calculations to prevent overflow: `math.Min(float64(current + added), float64(capacity))`
- Validate configuration parameters at startup to fail fast on invalid settings

Milestone Checkpoint

After implementing the rate limiting algorithms, verify your implementation:

Testing Commands:

```bash
go test ./internal/algorithms/... -v

go test ./internal/algorithms/... -race  # Check for race conditions

go test ./internal/algorithms/... -bench=. # Performance benchmarks
```

Expected Behavior:

- Token bucket allows bursts up to capacity, then enforces sustained rate
- Sliding window counter approximates rate with small boundary effects
- Sliding window log provides exact rate enforcement with higher memory usage
- All algorithms handle concurrent access safely through atomic Lua scripts

Manual Verification:

- Create token bucket with 10 capacity, 1/second refill rate
- Make 10 instant requests → all should succeed
- Make 11th request immediately → should be denied
- Wait 5 seconds, make 5 requests → all should succeed
- Monitor Redis keys to verify state is properly maintained

Debugging Signs:

- "Token count negative" → Race condition in Lua script logic
- "Memory usage growing unbounded" → Sliding window log not cleaning up old timestamps
- "Inconsistent rate limiting" → Time synchronization issues between instances
- "Redis timeout errors" → Connection pool exhaustion or network issues

The algorithms implemented in this milestone form the foundation for all subsequent distributed rate limiting functionality. Focus on correctness and atomicity - performance optimizations can be added later once the core logic is solid.

Multi-Tier Rate Limiting

> **Milestone(s):** Milestone 2 - Multi-tier Rate Limiting

The power of distributed rate limiting extends far beyond simple request counting. Real-world applications require sophisticated hierarchical controls that operate across multiple dimensions simultaneously - protecting individual users from abuse, preventing single IP addresses from overwhelming the system, ensuring fair API resource allocation, and maintaining global system capacity. This multi-tier approach transforms rate limiting from a binary gate into an intelligent traffic management system that can adapt to complex usage patterns while maintaining predictable system performance.

Mental Model: Cascading Security Checkpoints

Think of multi-tier rate limiting like airport security checkpoints, where passengers must pass through multiple stages of screening before reaching their destination. Each checkpoint serves a different purpose and operates independently, but they work together to ensure overall security and flow management.

At the document check station, each individual passenger (user) has personal limits - they can only travel on their specific ticket with their allocated baggage allowance. This represents **per-user rate limits** that prevent individual accounts from consuming excessive resources through either malicious behavior or buggy client code.

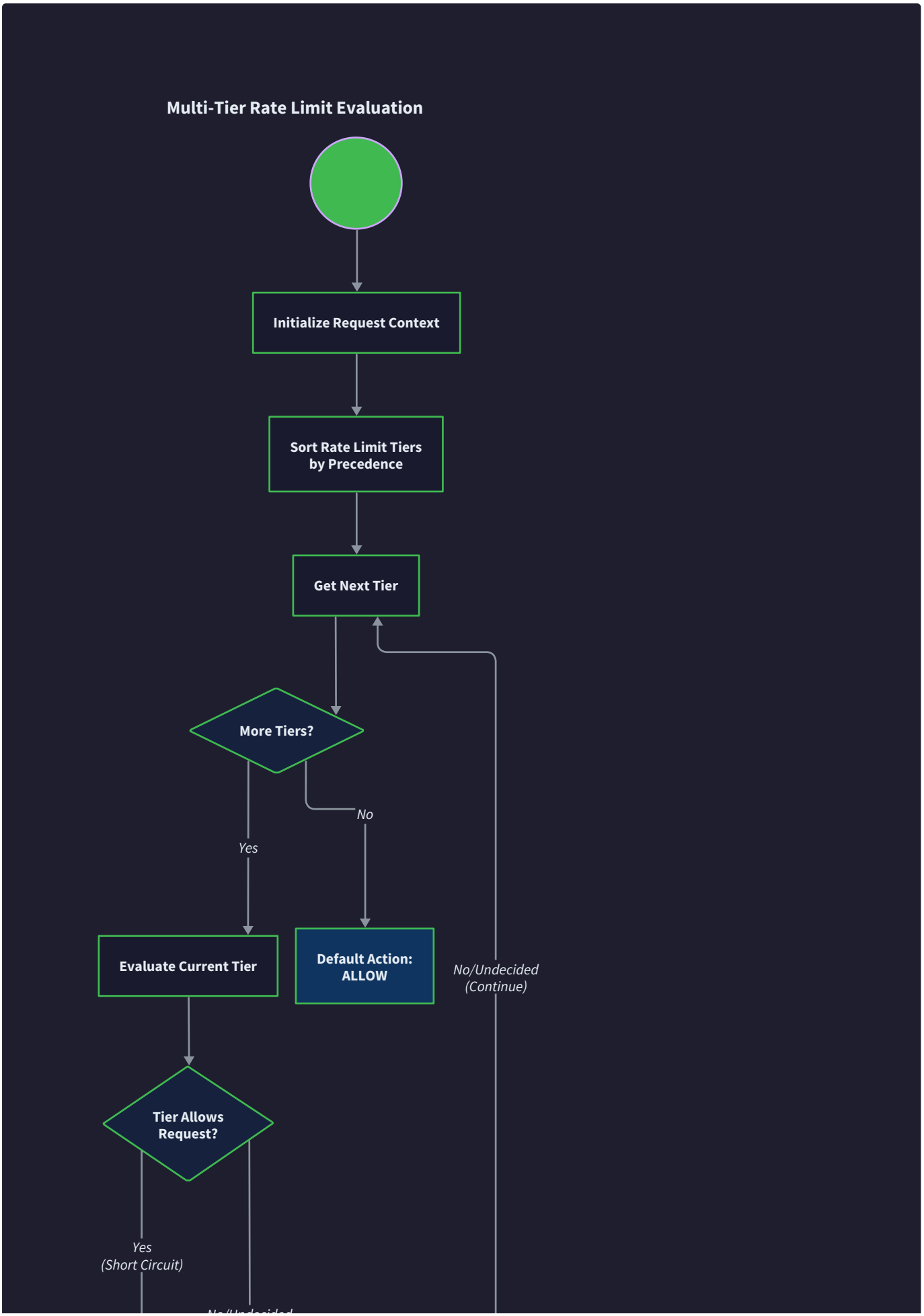
The security scanner represents **per-IP rate limits**, where the physical location (IP address) has throughput constraints regardless of how many different passengers (users) might be coming from that location. A single corporate NAT gateway might have hundreds of employees behind it, but the checkpoint can only process a certain number of people per minute from any single entry point.

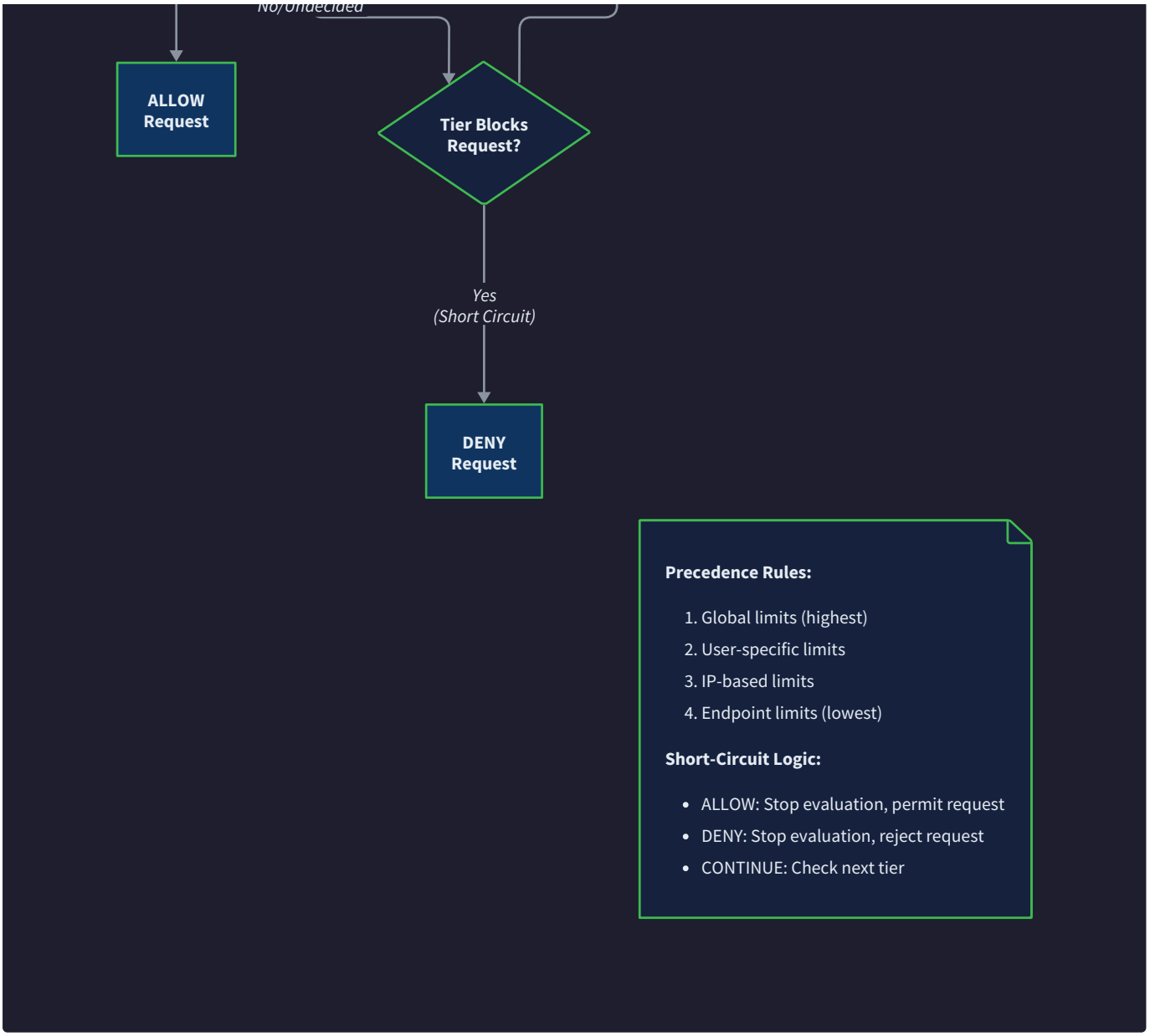
The gate area introduces **per-API rate limits**, where different flight destinations (API endpoints) have different capacity constraints. The international terminal might handle fewer but more complex departures, while domestic gates process higher volumes of simpler operations. Each API endpoint has distinct resource requirements and thus different rate limiting policies.

Finally, the airport itself has **global rate limits** - a maximum number of passengers it can process per hour across all gates, regardless of individual tickets or origins. Even if every individual checkpoint is under capacity, the airport's overall throughput ceiling prevents system-wide congestion that would affect everyone.

The critical insight is that passengers must successfully pass through ALL checkpoints in sequence. Being approved at document check doesn't guarantee passage through security, and having space at your gate doesn't matter if the airport has reached its global capacity limit. However, once any checkpoint rejects a passenger, there's no need to continue checking the remaining stations - this **short-circuit evaluation** saves resources and provides faster feedback.

Just as airport security adapts to different threat levels and passenger volumes, multi-tier rate limiting systems must handle varying load patterns, user behaviors, and attack scenarios while maintaining fair resource allocation across all dimensions.


Multi-Tier Rate Limit Evaluation
Initialize Request Context
Sort Rate Limit Tiers by Precedence
Get Next Tier
More Tiers?
No
Yes
Evaluate Current Tier
Default Action: ALLOW
No/Undecided (Continue)
Tier Allows Request?
Yes (Short Circuit)



Tier Evaluation Strategy

Multi-tier rate limiting requires a sophisticated evaluation strategy that balances accuracy, performance, and resource consumption. The system must check multiple rate limit dimensions efficiently while avoiding unnecessary computation when limits are already exceeded. The evaluation algorithm determines not just whether a request should be allowed, but which specific limits are constraining the user and how much capacity remains across all tiers.

The **tier precedence hierarchy** establishes the order in which rate limits are evaluated, typically flowing from most specific to most general: per-user limits, per-IP limits, per-API limits, and finally global limits. This ordering reflects both the logical dependency relationships and the performance characteristics of different limit types. User-specific limits are typically the most restrictive and fastest to evaluate since they operate on a single key, while global limits require aggregation across many keys and are computationally expensive.

However, tier precedence alone is insufficient - the system must also consider **limit severity** and **evaluation cost** when determining the optimal checking sequence. A per-user limit that allows 1000 requests per hour should be checked before a global limit that allows 10 million requests per hour, even if the global limit is technically "higher" in the hierarchy. Similarly, expensive operations like sliding window log evaluations should be deferred until cheaper approximations (like sliding window counters) have passed.

The **short-circuit evaluation algorithm** implements this strategy through a prioritized sequence of checks:

1. **Load applicable rules** for the request context by matching the user ID, IP address, and API endpoint against the rule patterns stored in the `RuleManager`. This initial filtering reduces the evaluation set from potentially thousands of rules to a manageable subset of 3-10 relevant limits.
2. **Sort rules by priority and cost** to establish the evaluation order. Rules with higher priority values (indicating more restrictive or important limits) are checked first, with tie-breaking based on computational cost estimates for each algorithm type.

3. **Evaluate each rule in sequence** using the appropriate rate limiting algorithm. For each rule, extract the required tokens from the request, construct the Redis key using the rule's key pattern, and perform the atomic check-and-update operation.
4. **Short-circuit on first failure** - as soon as any rule denies the request, immediately return a rejection response without evaluating remaining rules. The response includes the specific rule ID, remaining capacity, and retry-after timing from the failing rule.
5. **Aggregate success results** - if all rules pass, combine the results to determine the most restrictive remaining capacity and earliest reset time across all checked tiers. This aggregate information populates the rate limit headers in the successful response.

The evaluation strategy must also handle **rule conflicts** where multiple rules apply to the same request dimension but specify different limits or algorithms. The system resolves conflicts by giving precedence to rules with higher priority values, with the most restrictive rule taking effect when priorities are equal.

Rule matching uses pattern-based key composition to determine which rules apply to each request. A rule with key pattern `user:{user_id}:api:orders` matches requests from any user to the orders API endpoint, while `user:premium_user_*:api:*` applies broader limits to all premium users across all endpoints. The pattern matching system supports wildcards, prefix matching, and parameterized substitution to create flexible rule targeting.

The evaluation strategy incorporates **performance optimizations** to minimize latency and Redis load. Rule results are cached locally for short periods (typically 100-500 milliseconds) to avoid redundant checks for burst traffic patterns. The system also maintains separate connections to Redis for different rule priorities, allowing high-priority user limits to bypass congestion from expensive global limit calculations.

Error handling during evaluation follows a fail-open strategy by default - if a specific rule cannot be evaluated due to Redis connectivity issues, the system logs the failure and continues checking remaining rules. Only if all applicable rules fail to evaluate does the system fall back to local rate limiting or (in strict security mode) deny the request entirely.

> **Design Insight**: The tier evaluation strategy serves as the nervous system of the distributed rate limiter, coordinating decisions across multiple dimensions while maintaining sub-millisecond response times. The short-circuit approach not only improves performance but also provides clearer user feedback by identifying the specific limit that constrained their request rather than an ambiguous "rate limited" message.

Tier Evaluation Algorithm Details

The core tier evaluation algorithm operates through a series of well-defined phases that transform a raw request into a comprehensive rate limiting decision:

Phase	Input	Processing	Output	Performance Impact
Rule Discovery	`RateLimitRequest` with user/IP/API context	Pattern matching against `RuleManager` rule set	Filtered list of applicable `RateLimitRule` objects	O(log n) with indexed patterns
Rule Prioritization	List of applicable rules	Sort by priority field, then by algorithm cost	Ordered evaluation sequence	O(k log k) where k is rule count
Sequential Evaluation	Ordered rule list + request tokens	Redis-backed algorithm execution per rule	First failure or aggregated success	O(k) Redis operations, short-circuits
Result Aggregation	Individual rule results	Compute most restrictive limits and earliest reset	Final `RateLimitResult` with headers	O(k) in-memory computation

The **rule discovery phase** leverages efficient pattern matching to avoid evaluating irrelevant rules. The `RuleManager` maintains separate indices for user patterns, IP patterns, and API patterns, allowing the system to quickly identify candidate rules without scanning the entire rule set. For a request from user "user123" to IP "192.168.1.10" accessing "/api/orders", the discovery process queries each index independently and takes the intersection of matching rules.

Rule prioritization considers both the explicit priority field and implicit algorithm costs when determining evaluation order:

Algorithm Type	Relative Cost	Reason	Evaluation Priority
`ALGORITHM_TOKEN_BUCKET`	Low	Single Redis key, simple arithmetic	Check first
`ALGORITHM_SLIDING_COUNTER`	Medium	Multiple bucket keys, time-based logic	Check second
`ALGORITHM_SLIDING_WINDOW_LOG`	High	List operations, timestamp management	Check last

Sequential evaluation implements the short-circuit logic with careful attention to atomicity and consistency. Each rule evaluation calls the `CheckAndUpdate` method on the appropriate algorithm implementation, passing the constructed Redis key and required token count. The evaluation immediately terminates on the first rule that returns `allowed: false`, capturing the specific failure details for client feedback.

Result aggregation becomes necessary only when all individual rule checks succeed. The aggregation process identifies the most restrictive remaining capacity across all checked rules, calculates the earliest reset time when any limit will refresh, and determines the appropriate retry-after value for rate limit headers.

Common Pitfalls in Tier Evaluation

⚠️ Pitfall: Evaluating All Tiers Even After Failure

A common mistake in multi-tier rate limiting implementations is continuing to evaluate all applicable rules even after one has already denied the request. This occurs when developers implement rule checking as a validation pipeline that collects all failures rather than a gate system that stops at the first barrier.

```
// WRONG: Evaluating all rules regardless of failures
results := make([]*RateLimitResult, 0)
for _, rule := range applicableRules {
    result := evaluateRule(rule, request)
    results = append(results, result)
}
// Then checking if any failed...
```

This approach wastes Redis operations, increases response latency, and provides confusing user feedback since the client receives information about multiple limit violations when only the first one is actionable. The correct approach uses immediate return on first failure:

```
// CORRECT: Short-circuit evaluation
for _, rule := range applicableRules {
    result := evaluateRule(rule, request)
    if !result.allowed {
        return result  // Immediate failure, no further evaluation
    }
}
```

⚠️ Pitfall: Inconsistent Rule Priority Handling

Another frequent error involves inconsistent interpretation of priority values across different parts of the system. Some implementations treat higher numeric values as higher priority while others use the reverse convention, leading to rules being evaluated in the wrong order.

The problem becomes especially acute when priority ties need resolution - without consistent tie-breaking logic, the same request might be evaluated differently across application instances, leading to unpredictable rate limiting behavior. Always use explicit priority ordering with well-defined tie-breaking rules based on secondary criteria like rule creation time or alphabetical rule ID ordering.

⚠️ Pitfall: Ignoring Rule Pattern Overlap

Overlapping rule patterns can create unexpected interactions where multiple rules apply to the same request dimension but specify conflicting limits. For example, a general rule `user:*:api:*` allowing 1000 requests per hour might conflict with a specific rule `user:premium_*:api:orders` allowing 5000 requests per hour for premium users accessing the orders endpoint.

Without proper conflict resolution, the system might apply the more restrictive limit even when a more specific, permissive rule should take precedence. Always implement explicit precedence rules that favor more specific patterns and higher priority values when resolving overlaps.

Rate Limit Key Composition

The foundation of effective multi-tier rate limiting lies in **Redis key composition** - the systematic construction of unique identifiers that organize rate limit state across different dimensions, time windows, and algorithms. Key composition determines not only how rate limiting data is stored and retrieved, but also how efficiently the system can scale, how clearly administrators can debug issues, and how reliably the system maintains consistency under load.

Effective key composition balances several competing requirements: keys must be **unique** to prevent collisions between different rate limits, **predictable** to enable debugging and monitoring, **efficient** to minimize Redis memory usage, and **hierarchical** to support operations like bulk deletion or pattern-based queries. The key structure also influences Redis clustering behavior, as keys determine which Redis node stores each rate limit counter.

The **base key structure** follows a hierarchical pattern that encodes the rate limit dimension, resource identifier, time window, and algorithm type:

```
ratelimit:{dimension}:{resource}:{algorithm}:{window}:{additional_context}
```

This structure provides natural groupings that align with Redis operations and administrative needs. The `ratelimit:` prefix enables easy identification of rate limiting keys in mixed-use Redis instances. The dimension component (`user`, `ip`, `api`, `global`) creates logical separation between different rate limit types. The resource identifier specifies the exact entity being limited, while algorithm and window components ensure that different rate limiting approaches for the same resource don't interfere.

User-scoped rate limit keys incorporate the user identifier and any relevant context that affects the user's rate limit tier:

Key Pattern	Example	Use Case	Considerations
`ratelimit:user:{user_id}:token_bucket:{window}`	`ratelimit:user:user123:token_bucket:3600`	Per-user global limits	Simple, efficient, scales linearly
`ratelimit:user:{user_id}:api:{endpoint}:sliding_counter:{window}`	`ratelimit:user:user123:api:orders:sliding_counter:3600`	Per-user API endpoint limits	More granular, higher memory usage
`ratelimit:user:{user_tier}:{user_id}:global:token_bucket:{window}`	`ratelimit:user:premium:user123:global:token_bucket:3600`	Tier-aware user limits	Enables tier-specific limit policies

The user ID component typically uses the application's native user identifier (database primary key, UUID, or username) rather than derived values, ensuring consistency across application restarts and reducing the likelihood of key collisions during user management operations.

IP-scoped rate limit keys must handle the complexities of network addressing, proxy scenarios, and IPv4/IPv6 compatibility:

Key Pattern	Example	Use Case	Special Handling
`ratelimit:ip:{ip_address}:sliding_log:{window}`	`ratelimit:ip:192.168.1.10:sliding_log:3600`	Direct IP rate limiting	IPv6 colon escaping required
`ratelimit:ip:{ip_subnet}:token_bucket:{window}`	`ratelimit:ip:192.168.1.0_24:token_bucket:3600`	Subnet-based rate limiting	Subnet calculation for each request
`ratelimit:ip:{ip_hash}:api:{endpoint}:sliding_counter:{window}`	`ratelimit:ip:a1b2c3d4:api:upload:sliding_counter:300`	Privacy-preserving IP limits	Hash consistency across instances

IP address handling requires careful consideration of IPv6 compatibility, as IPv6 addresses contain colons that conflict with Redis key delimiters. The system typically normalizes IPv6 addresses to a canonical form and replaces colons with underscores or uses base64 encoding for the address component.

API-scoped rate limit keys organize limits by endpoint, HTTP method, and resource type to provide fine-grained control over different API operations:

Key Pattern	Example	Use Case	Granularity Trade-offs
`ratelimit:api:{endpoint}:sliding_counter:{window}`	`ratelimit:api:orders:sliding_counter:3600`	Endpoint-level rate limiting	Coarse-grained, efficient
`ratelimit:api:{method}:{endpoint}:token_bucket:{window}`	`ratelimit:api:POST:orders:token_bucket:3600`	Method-specific rate limiting	Distinguishes read/write operations
`ratelimit:api:{service}:{version}:{endpoint}:sliding_log:{window}`	`ratelimit:api:orders:v2:create:sliding_log:300`	Versioned API rate limiting	Supports API evolution and migration

API endpoint normalization presents significant challenges in key composition, as URLs may contain variable path parameters, query strings, and other dynamic elements. The system typically applies endpoint normalization rules that replace path parameters with placeholders (`/orders/123` becomes `/orders/{id}`) and ignore query parameters unless they're explicitly included in the rate limiting policy.

Global rate limit keys aggregate usage across all users, IPs, and endpoints to enforce system-wide capacity constraints:

Key Pattern	Example	Use Case	Aggregation Method
`ratelimit:global:requests:sliding_counter:{window}`	`ratelimit:global:requests:sliding_counter:60`	Total request rate limiting	Simple counter increment
`ratelimit:global:{resource_type}:token_bucket:{window}`	`ratelimit:global:database_writes:token_bucket:3600`	Resource-specific global limits	Categorized request counting
`ratelimit:global:{region}:api:{endpoint}:sliding_log:{window}`	`ratelimit:global:us-west:api:search:sliding_log:300`	Regional global rate limiting	Geographic request distribution

Global rate limits present unique challenges in distributed systems, as they require coordination across all application instances and potentially multiple Redis nodes. The key composition must support efficient aggregation while avoiding hot-spotting on a single Redis key.

Time window encoding within rate limit keys determines how algorithm implementations track temporal boundaries and handle window transitions:

Algorithm Type	Time Window Encoding	Example Component	Window Alignment
`ALGORITHM_TOKEN_BUCKET`	Window duration in seconds	`:3600`	Floating window based on first request
`ALGORITHM_SLIDING_COUNTER`	Current time bucket + duration	`:1609459200:3600`	Fixed time bucket boundaries
`ALGORITHM_SLIDING_WINDOW_LOG`	Window duration only	`:3600`	Sliding based on request timestamps

The time window encoding affects both key uniqueness and Redis memory usage patterns. Sliding window counter algorithms generate multiple keys per time window (one per sub-bucket), while token bucket algorithms typically use a single key with embedded timestamp data.

Redis cluster considerations influence key composition decisions to ensure optimal data distribution and avoid hot-spotting. The Redis cluster uses CRC16 hashing of the key to determine node assignment, meaning that keys with similar prefixes may cluster on the same node:

> **Design Insight**: Effective key composition serves as the addressing system for distributed rate limiting state, determining not just where data lives but how efficiently it can be accessed, updated, and managed. The key structure becomes the foundation for Redis clustering, monitoring queries, and administrative operations.

Key Composition Implementation Strategy

The `RateLimitRule` structure includes a `key_pattern` field that serves as a template for generating actual Redis keys based on request context. The pattern uses placeholder syntax to inject dynamic values:

Pattern Component	Placeholder Syntax	Example Pattern	Generated Key
User ID	`{user_id}`	`ratelimit:user:{user_id}:api:orders`	`ratelimit:user:user123:api:orders`
IP Address	`{ip_address}`	`ratelimit:ip:{ip_address}:global`	`ratelimit:ip:192.168.1.10:global`
API Endpoint	`{api_endpoint}`	`ratelimit:api:{api_endpoint}:sliding_counter`	`ratelimit:api:orders:sliding_counter`
Algorithm Type	`{algorithm}`	`ratelimit:user:{user_id}:{algorithm}`	`ratelimit:user:user123:token_bucket`
Time Window	`{window}`	`ratelimit:global:requests:{window}`	`ratelimit:global:requests:3600`

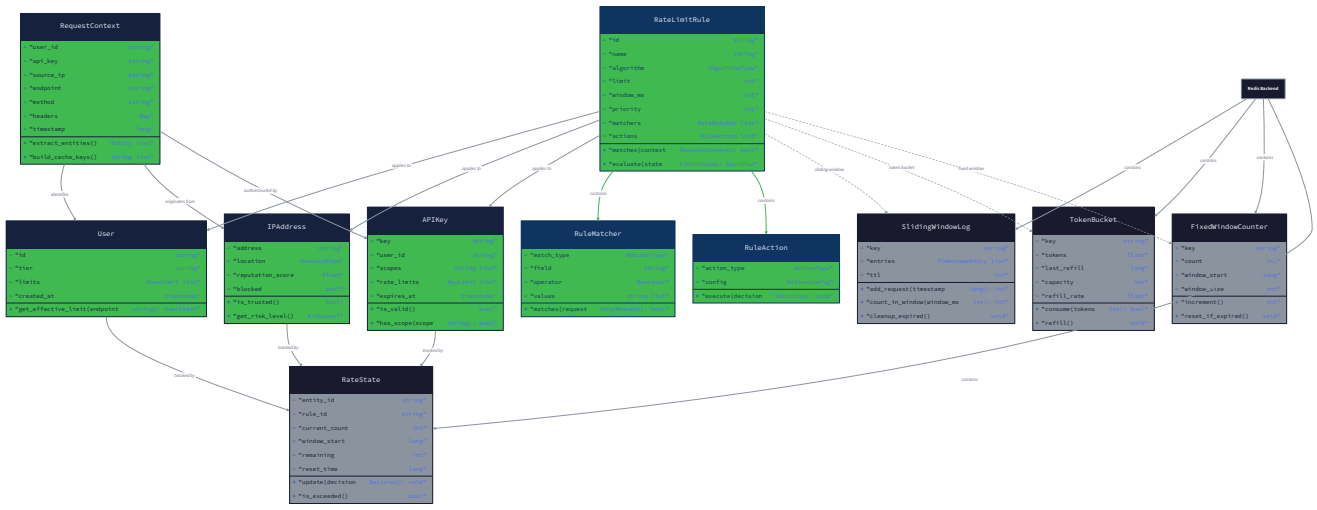
The key composition system includes **validation logic** to ensure generated keys meet Redis requirements and avoid common pitfalls:

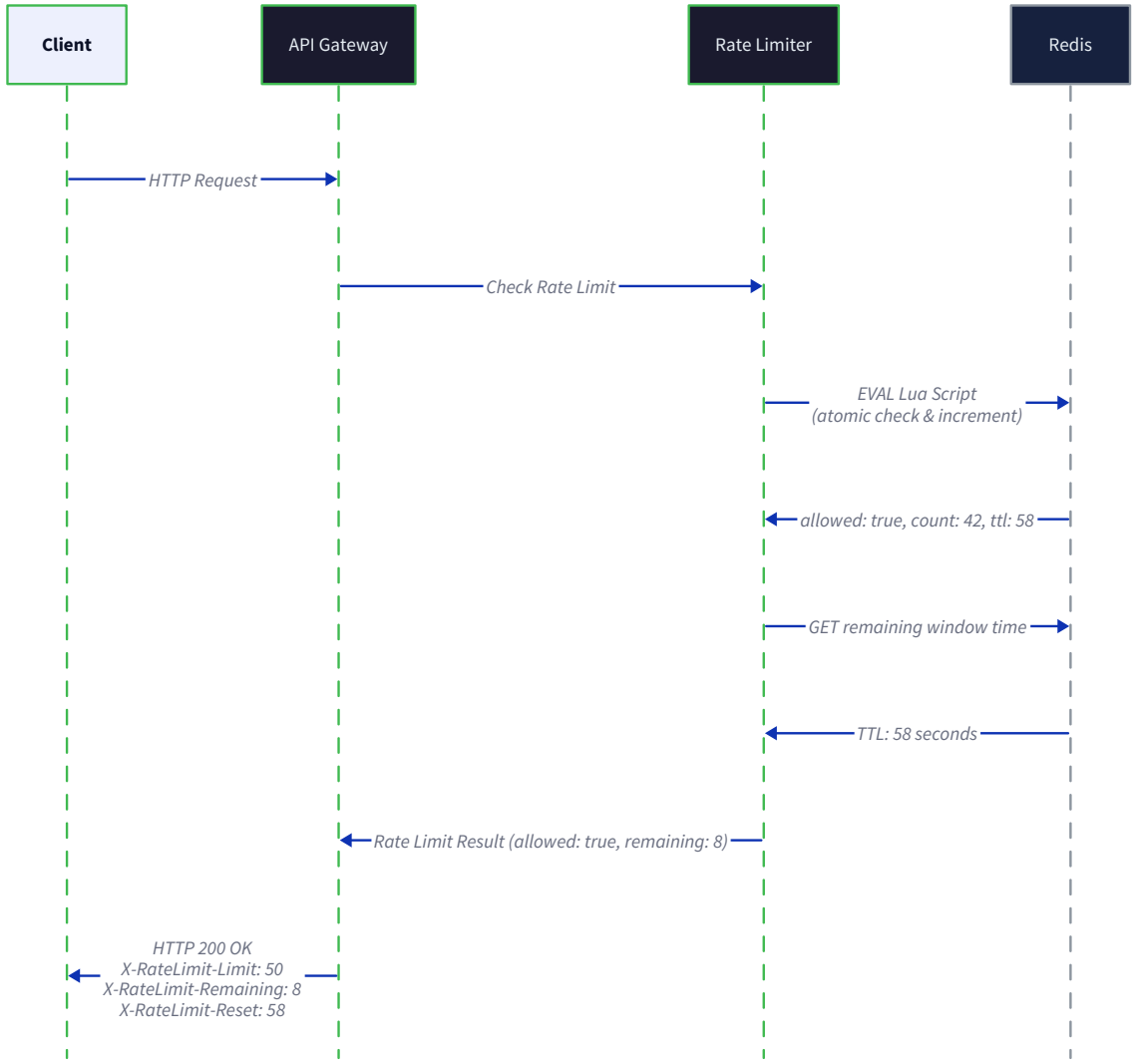
- **Length validation**: Redis keys are limited to 512 MB, but practical considerations limit keys to under 250 characters for optimal performance
- **Character validation**: Keys avoid problematic characters like spaces, newlines, and Redis command separators
- **Collision detection**: The system validates that different rule patterns cannot generate identical keys under normal operation
- **Cluster compatibility**: Keys are structured to distribute evenly across Redis cluster nodes

Key lifecycle management handles the creation, updates, and cleanup of rate limiting keys as rules change and time windows expire:

1. **Key creation** occurs dynamically when the first request matching a rule arrives, with initial values set according to the algorithm type
2. **Key updates** happen atomically through Lua scripts that ensure consistency during concurrent access
3. **Key expiration** uses Redis TTL mechanisms to automatically clean up expired time windows and reduce memory usage
4. **Key migration** handles rule changes that affect key patterns, typically through background processes that move data to new key structures

The key composition strategy also supports **administrative operations** like bulk key deletion, usage reporting, and debugging through predictable key patterns that enable efficient Redis pattern matching and iteration.





Implementation Guidance

Multi-tier rate limiting requires careful orchestration of rule management, key composition, and evaluation logic to achieve both correctness and performance. The implementation bridges the conceptual design with practical Redis operations while handling the complexities of distributed coordination and failure scenarios.

Technology Recommendations

Component	Simple Option	Advanced Option
Rule Storage	YAML configuration files with file watching	Redis-based rule storage with pub/sub updates
Pattern Matching	Simple string templates with Go text/template	Regex-based patterns with named capture groups
Key Composition	String concatenation with validation	Template engine with type-safe placeholders
Tier Coordination	Sequential rule evaluation with early termination	Parallel rule evaluation with context cancellation
Result Caching	In-memory LRU cache with TTL	Redis-based shared cache across instances

Recommended Module Structure

```
internal/
  limiter/
    multi_tier.go           ← Main multi-tier limiter implementation
    rule_manager.go         ← Rule loading and matching logic
    key_composer.go         ← Redis key composition utilities
    tier_evaluator.go       ← Sequential evaluation with short-circuiting
    result_aggregator.go    ← Success result combination logic
  config/
    rules.yaml             ← Rate limit rule definitions
    rule_loader.go         ← YAML rule parsing and validation
    rule_watcher.go        ← File change detection for rule updates
  storage/
    redis_operations.go    ← Atomic Redis operations for rule evaluation
```

Infrastructure Starter Code

Rule Configuration Structure (config/rules.yaml):

```yaml
# Complete working rule configuration

rules:

  - id: "user_global_limit"

    name: "Per-user global rate limit"

    key_pattern: "ratelimit:user:{user_id}:global:token_bucket:{window}"

    algorithm: "token_bucket"

    limit: 1000

    window: "1h"

    burst_limit: 1200

    priority: 100

    enabled: true



  - id: "ip_burst_protection"

    name: "Per-IP burst protection"

    key_pattern: "ratelimit:ip:{ip_address}:burst:sliding_counter:{window}"

    algorithm: "sliding_window_counter"

    limit: 100

    window: "1m"

    priority: 200

    enabled: true



  - id: "api_endpoint_limit"

    name: "Per-API endpoint rate limit"

    key_pattern: "ratelimit:api:{api_endpoint}:requests:sliding_log:{window}"

    algorithm: "sliding_window_log"

    limit: 10000

    window: "1h"

    priority: 50

    enabled: true
```

Rule Loader Implementation (config/rule_loader.go):

```go
package config


import (

    "fmt"

    "os"

    "time"

    "gopkg.in/yaml.v2"

)


// Complete rule loading with validation

type RuleConfig struct {

    Rules []*RateLimitRule `yaml:"rules"`

}


// LoadRules loads and validates rate limit rules from YAML configuration

func LoadRules(configPath string) ([]*RateLimitRule, error) {

    data, err := os.ReadFile(configPath)

    if err != nil {

        return nil, fmt.Errorf("failed to read rule config: %w", err)

    }



    var config RuleConfig

    if err := yaml.Unmarshal(data, &config); err != nil {

        return nil, fmt.Errorf("failed to parse rule config: %w", err)

    }



    // Validate and normalize rules

    for _, rule := range config.Rules {

        if err := validateRule(rule); err != nil {

            return nil, fmt.Errorf("invalid rule %s: %w", rule.ID, err)

        }



        // Parse window duration

        if duration, err := time.ParseDuration(rule.WindowStr); err != nil {

            return nil, fmt.Errorf("invalid window duration for rule %s: %w", rule.ID, err)

        } else {

            rule.Window = duration

        }
```

```
        // Set timestamps
        rule.CreatedAt = time.Now()
        rule.UpdatedAt = time.Now()
    }

    return config.Rules, nil
}

func validateRule(rule *RateLimitRule) error {
    if rule.ID == "" {
        return fmt.Errorf("rule ID is required")
    }
    if rule.KeyPattern == "" {
        return fmt.Errorf("key pattern is required")
    }
    if rule.Limit <= 0 {
        return fmt.Errorf("limit must be positive")
    }

    // Validate algorithm
    validAlgorithms := map[string]bool{
        ALGORITHM_TOKEN_BUCKET:         true,
        ALGORITHM_SLIDING_WINDOW_LOG:   true,
        ALGORITHM_SLIDING_COUNTER:      true,
    }
    if !validAlgorithms[rule.Algorithm] {
        return fmt.Errorf("unsupported algorithm: %s", rule.Algorithm)
    }

    return nil
}
```

Key Composer Implementation (limiter/key_composer.go):

```go
package limiter

import (
    "fmt"
    "net"
    "regexp"
    "strings"
    "time"
)

// KeyComposer handles Redis key generation from rule patterns
type KeyComposer struct {
    patternCache map[string]*regexp.Regexp
}

// NewKeyComposer creates a key composer with pattern caching
func NewKeyComposer() *KeyComposer {
    return &KeyComposer{
        patternCache: make(map[string]*regexp.Regexp),
    }
}

// ComposeKey generates Redis key from rule pattern and request context
func (kc *KeyComposer) ComposeKey(rule *RateLimitRule, req *RateLimitRequest) (string, error) {
    key := rule.KeyPattern

    // Replace standard placeholders
    replacements := map[string]string{
        "{user_id}":      normalizeUserID(req.UserID),
        "{ip_address}":   normalizeIPAddress(req.IPAddress),
        "{api_endpoint}": normalizeAPIEndpoint(req.APIEndpoint),
        "{algorithm}":    rule.Algorithm,
        "{window}":       fmt.Sprintf("%d", int64(rule.Window.Seconds())),
    }

    for placeholder, value := range replacements {
        key = strings.ReplaceAll(key, placeholder, value)
    }
```

```
    // Validate final key
    if err := validateRedisKey(key); err != nil {
        return "", fmt.Errorf("invalid Redis key generated: %w", err)
    }

    return key, nil
}

// Utility functions for key normalization
func normalizeUserID(userID string) string {
    if userID == "" {
        return "anonymous"
    }
    // Remove problematic characters
    normalized := regexp.MustCompile(`[^a-zA-Z0-9_-]`).ReplaceAllString(userID, "_")
    if len(normalized) > 64 {
        normalized = normalized[:64]
    }
    return normalized
}

func normalizeIPAddress(ipAddr string) string {
    ip := net.ParseIP(ipAddr)
    if ip == nil {
        return "invalid_ip"
    }

    // Handle IPv6 - replace colons with underscores
    if strings.Contains(ipAddr, ":") {
        return strings.ReplaceAll(ip.String(), ":", "_")
    }

    return ip.String()
}

func normalizeAPIEndpoint(endpoint string) string {
    // Remove leading slash
    normalized := strings.TrimPrefix(endpoint, "/")
```

```
    // Replace path separators with underscores
    normalized = strings.ReplaceAll(normalized, "/", "_")

    // Replace query parameters and fragments
    if idx := strings.Index(normalized, "?"); idx != -1 {
        normalized = normalized[:idx]
    }
    if idx := strings.Index(normalized, "#"); idx != -1 {
        normalized = normalized[:idx]
    }

    // Limit length
    if len(normalized) > 100 {
        normalized = normalized[:100]
    }

    return normalized
}

func validateRedisKey(key string) error {
    if len(key) == 0 {
        return fmt.Errorf("key cannot be empty")
    }
    if len(key) > 250 {
        return fmt.Errorf("key too long: %d characters", len(key))
    }

    // Check for problematic characters
    if strings.ContainsAny(key, " \n\r\t") {
        return fmt.Errorf("key contains whitespace characters")
    }

    return nil
}
```

Core Logic Skeleton

Multi-Tier Limiter Implementation (limiter/multi_tier.go):

```go
package limiter

import (
    "context"
    "fmt"
    "sort"
    "time"
)

// MultiTierLimiter implements hierarchical rate limiting across multiple dimensions
type MultiTierLimiter struct {
    storage      Storage
    ruleManager  *RuleManager
    keyComposer  *KeyComposer
    algorithms   map[string]Algorithm
    localFallback Limiter
}

// NewMultiTierLimiter creates a multi-tier rate limiter
func NewMultiTierLimiter(storage Storage, ruleManager *RuleManager) *MultiTierLimiter {
    return &MultiTierLimiter{
        storage:     storage,
        ruleManager: ruleManager,
        keyComposer: NewKeyComposer(),
        algorithms:  make(map[string]Algorithm),
    }
}

// Check performs multi-tier rate limit evaluation with short-circuit logic
func (mtl *MultiTierLimiter) Check(ctx context.Context, req *RateLimitRequest) (*RateLimitResult, error) {
    // TODO 1: Get all applicable rules for this request using ruleManager.GetMatchingRules()
    // Hint: Pass req.UserID, req.IPAddress, req.APIEndpoint to find matching patterns

    // TODO 2: Sort rules by priority (higher priority first) and algorithm cost
    // Hint: Use sort.Slice with custom comparison function checking rule.Priority

    // TODO 3: Evaluate each rule in sequence with short-circuit on first failure
    // Hint: Call evaluateRule() for each rule, return immediately if result.Allowed == false

```

```
    // TODO 4: If all rules pass, aggregate the most restrictive limits
    // Hint: Find minimum remaining capacity and earliest reset time across all results

    // TODO 5: Handle Redis failures by falling back to local rate limiting
    // Hint: Check error types and call mtl.localFallback.Check() on storage errors

    return nil, fmt.Errorf("not implemented")
}

// evaluateRule performs rate limit check for a single rule
func (mtl *MultiTierLimiter) evaluateRule(ctx context.Context, rule *RateLimitRule, req *RateLimitRequest) (*RateLimitResult,
error) {
    // TODO 1: Generate Redis key using keyComposer.ComposeKey(rule, req)

    // TODO 2: Get the appropriate algorithm implementation for rule.Algorithm
    // Hint: Look up in mtl.algorithms map, return error if not found

    // TODO 3: Call algorithm.Check() with the generated key and required tokens
    // Hint: Use req.Tokens for token count, handle context cancellation

    // TODO 4: Populate result with rule metadata (rule ID, algorithm type)

    // TODO 5: Apply burst limit adjustments if rule has burst_limit configured
    // Hint: For token bucket, allow temporary exceeding of base limit up to burst_limit

    return nil, fmt.Errorf("not implemented")
}

// Preview checks rate limit status without updating counters
func (mtl *MultiTierLimiter) Preview(ctx context.Context, req *RateLimitRequest) (*RateLimitResult, error) {
    // TODO 1: Similar to Check() but call Preview() on individual algorithms
    // TODO 2: Return aggregate status without modifying any counters
    return nil, fmt.Errorf("not implemented")
}
```

Rule Manager Implementation (limiter/rule_manager.go):

```go
package limiter

import (
    "context"
    "fmt"
    "regexp"
    "strings"
    "sync"
)

// RuleManager handles rule storage, matching, and updates
type RuleManager struct {
    rules       map[string]*RateLimitRule
    userIndex   map[string][]*RateLimitRule
    ipIndex     map[string][]*RateLimitRule
    apiIndex    map[string][]*RateLimitRule
    globalRules []*RateLimitRule
    mutex       sync.RWMutex
}

// NewRuleManager creates a rule manager with indexing
func NewRuleManager() *RuleManager {
    return &RuleManager{
        rules:     make(map[string]*RateLimitRule),
        userIndex: make(map[string][]*RateLimitRule),
        ipIndex:   make(map[string][]*RateLimitRule),
        apiIndex:  make(map[string][]*RateLimitRule),
    }
}

// GetMatchingRules returns all rules applicable to the given request context
func (rm *RuleManager) GetMatchingRules(userID, ipAddress, apiEndpoint string) []*RateLimitRule {
    rm.mutex.RLock()
    defer rm.mutex.RUnlock()

    // TODO 1: Check user-specific rules by looking up userID in userIndex
    // TODO 2: Check IP-specific rules by looking up ipAddress in ipIndex
    // TODO 3: Check API-specific rules by looking up apiEndpoint in apiIndex
    // TODO 4: Always include global rules from globalRules slice
```

```
    // TODO 5: Deduplicate rules that match multiple patterns
    // TODO 6: Filter out disabled rules (rule.Enabled == false)
    // Hint: Use map[string]bool to track seen rule IDs and avoid duplicates

    return nil
}

// LoadRules loads rules from configuration and rebuilds indices
func (rm *RuleManager) LoadRules(configPath string) error {
    rm.mutex.Lock()
    defer rm.mutex.Unlock()

    // TODO 1: Call config.LoadRules() to parse YAML configuration
    // TODO 2: Clear existing rules and indices
    // TODO 3: Rebuild rule indices by pattern type
    // TODO 4: Validate that no two rules generate conflicting Redis keys

    return fmt.Errorf("not implemented")
}

// buildIndices creates lookup indices for efficient rule matching
func (rm *RuleManager) buildIndices() {
    // TODO 1: Clear all existing indices
    // TODO 2: Iterate through all rules and categorize by key pattern
    // TODO 3: Rules with {user_id} patterns go into userIndex
    // TODO 4: Rules with {ip_address} patterns go into ipIndex
    // TODO 5: Rules with {api_endpoint} patterns go into apiIndex
    // TODO 6: Rules matching all requests go into globalRules
    // Hint: Use strings.Contains() to detect pattern placeholders
}
```

Milestone Checkpoint

After implementing multi-tier rate limiting, verify the following behaviors:

Test Command:

```
go test ./internal/limiter/... -v -run TestMultiTier
```
BASH

Expected Behavior:

1. **Rule Loading**: Configuration loads successfully with validation errors for malformed rules
2. **Pattern Matching**: Rules correctly match requests based on user ID, IP address, and API endpoint patterns
3. **Short-Circuit Evaluation**: Evaluation stops immediately when the first rule denies a request

4. **Priority Ordering**: Higher priority rules are evaluated before lower priority rules
5. **Result Aggregation**: Successful requests return the most restrictive remaining capacity across all checked rules

Manual Testing:

```bash
# Test per-user rate limiting
curl -H "X-User-ID: testuser" http://localhost:8080/api/orders
# Should show X-RateLimit-Remaining header

# Test IP rate limiting
for i in {1..10}; do curl -H "X-Forwarded-For: 192.168.1.100" http://localhost:8080/api/search; done
# Should eventually return 429 Too Many Requests

# Test API endpoint limits
curl http://localhost:8080/api/upload  # Different limits than /api/orders
```

Signs of Issues:

- Rules not matching expected requests → Check pattern normalization logic
- All rules being evaluated despite failures → Verify short-circuit implementation
- Inconsistent rate limiting across instances → Check Redis key composition
- Poor performance under load → Profile rule matching and Redis operations

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Rules not triggering	Pattern matching failure	Check generated Redis keys in logs	Fix key normalization functions
Inconsistent rate limiting	Rule priority conflicts	Review rule evaluation order	Implement explicit priority tie-breaking
High Redis CPU usage	Inefficient Lua scripts	Monitor Redis SLOWLOG	Optimize atomic operations
Memory leaks	Key expiration not working	Check Redis key TTLs	Implement proper cleanup

Redis Backend Integration

> **Milestone(s):** Milestone 3 - Redis Backend Integration

The transition from local rate limiting to distributed rate limiting represents one of the most critical architectural decisions in building scalable systems. While local rate limiting provides excellent performance and simplicity, it falls apart the moment you deploy multiple application instances. Each instance operates with its own view of request counts, leading to effective limits that are N times higher than intended, where N is the number of instances. Redis backend integration solves this fundamental problem by providing a shared state store that all application instances can access atomically, ensuring that rate limits are enforced accurately across the entire cluster.

Mental Model: Bank Transaction Processing

Understanding atomic check-and-update operations in distributed rate limiting becomes much clearer when we think about how banks handle account transactions. Consider what happens when you try to withdraw money from an ATM. The bank doesn't simply check your balance and then subtract the withdrawal amount in two separate operations—that would create a race condition where multiple ATMs could simultaneously check the same balance and approve withdrawals that collectively exceed your available funds.

Instead, banks use atomic transactions that combine the balance check and deduction into a single, indivisible operation. The ATM sends a request to the bank's central system saying "if account 12345 has at least $100, subtract $100 and tell me the new balance." This atomic check-and-update ensures that no matter how many ATMs are processing transactions simultaneously, the account balance remains consistent and never goes below zero (assuming no overdraft protection).

Rate limiting with Redis follows exactly the same pattern. When an application instance wants to allow a request, it can't simply check the current count in Redis and then increment it in a separate operation. Between the check and the increment, another instance might have processed requests that push the count over the limit. Instead, we need atomic check-and-update operations that say "if the current request count is below the limit, increment the count and tell me if the request is allowed."

This banking analogy extends to other aspects of our Redis integration. Just as banks have redundant systems and backup procedures for when the main transaction system fails, our rate limiter needs graceful degradation strategies when Redis becomes unavailable. And just as banks use connection pooling to efficiently handle thousands of simultaneous ATM transactions, our Redis integration uses connection pooling to manage the network resources efficiently across multiple application instances.

The key insight here is that distributed rate limiting isn't just about storing counters in a shared location—it's about ensuring that the fundamental check-and-update operation remains atomic even when performed by multiple processes across a network. This atomicity requirement drives every aspect of our Redis integration design, from Lua script implementation to connection management strategies.

Lua Script Design

Redis Lua scripts provide the atomicity guarantees we need for distributed rate limiting by ensuring that complex multi-step operations execute without interruption from other clients. Unlike regular Redis commands that might be interleaved with operations from other connections, Lua scripts run atomically within Redis, providing the equivalent of database transactions for our rate limiting logic.

The fundamental challenge in implementing rate limiting algorithms as Lua scripts lies in translating the conceptual algorithms we designed earlier into Redis operations that efficiently manipulate the underlying data structures. Each algorithm has different storage requirements and update patterns, requiring carefully crafted scripts that balance correctness, performance, and memory usage.

Token Bucket Lua Script Design

The token bucket algorithm requires maintaining two pieces of state: the current token count and the timestamp of the last refill operation. Our Lua script must atomically read these values, calculate how many tokens should be added based on elapsed time, update the bucket state, and determine whether the request can be allowed.

Script Operation	Redis Commands Used	Purpose	Complexity Consideration
Read current state	`HMGET key tokens last_refill`	Retrieve bucket state	Single round-trip, O(1)
Calculate elapsed time	Lua math operations	Determine refill amount	Time precision handling
Refill tokens	Lua math, bounded by capacity	Update token count	Prevent integer overflow
Check allowance	Compare tokens to request	Rate limit decision	Handle burst scenarios
Update state	`HMSET key tokens new_tokens last_refill now`	Persist new state	Atomic state update
Set expiration	`EXPIRE key window_seconds`	Cleanup old buckets	Memory management

The token bucket script must handle several edge cases that make the implementation more complex than a simple counter increment. Time calculations require careful handling of integer precision to avoid drift over long periods. The refill calculation must prevent token counts from exceeding the bucket capacity while handling cases where the elapsed time is so large that multiple full refills should have occurred.

Sliding Window Counter Lua Script Design

Sliding window counter scripts operate on a hash structure where each field represents a time bucket and its value contains the request count for that bucket. The script must determine the current bucket, increment the appropriate counter, calculate the total count across all buckets within the window, and clean up expired buckets to prevent unbounded memory growth.

Script Phase	Operations	Redis Commands	Error Handling
Bucket identification	Calculate current bucket ID from timestamp	Lua math operations	Handle clock skew
Current bucket update	Increment counter for current bucket	`HINCRBY key bucket_id tokens`	Initialize if missing
Window calculation	Sum counts across active buckets	`HMGET key bucket1 bucket2 ...`	Handle missing buckets
Expired cleanup	Remove buckets outside window	`HDEL key expired_bucket1 ...`	Batch deletions
Rate limit decision	Compare total to limit	Lua comparison	Return detailed result
Expiration management	Set key TTL based on window	`EXPIRE key ttl_seconds`	Prevent memory leaks

The sliding window counter script faces the challenge of maintaining accuracy while managing memory efficiently. The number of buckets affects both accuracy and memory usage—more buckets provide smoother rate limiting but require more Redis memory and script execution time. The script must balance these trade-offs while ensuring that bucket cleanup doesn't interfere with active rate limiting decisions.

Sliding Window Log Lua Script Design

The sliding window log approach stores individual request timestamps, providing the highest accuracy at the cost of memory usage proportional to the request rate. The Lua script must add new timestamps to a Redis list or sorted set, remove expired timestamps, count remaining timestamps, and make the rate limiting decision—all atomically.

Implementation Choice	Redis Structure	Script Operations	Trade-offs
Sorted Set approach	`ZADD key timestamp uuid`	Add, remove by score, count	Memory efficient, complex cleanup
List approach	`LPUSH key timestamp`	Push, trim, count	Simple operations, less precise cleanup
Hybrid approach	Sorted set with periodic cleanup	Add new, batch remove old	Best of both, complex logic

The sorted set implementation provides the most precise sliding window behavior because it can efficiently remove timestamps that fall outside the window using `ZREMRANGEBYSCORE`. However, it requires generating unique scores for timestamps that might be identical, typically by appending a random component or sequence number.

> **Design Insight**: The choice between Redis data structures for sliding window log has profound implications for memory usage patterns. Sorted sets provide O(log N) operations but require 16 bytes overhead per entry, while lists provide O(1) append but only O(N) cleanup. For high-traffic keys, this memory difference can determine whether the rate limiter scales economically.

Script Error Handling and Return Values

All Lua scripts must return consistent, structured results that allow the calling application to make informed decisions about request handling and error recovery. The script return format needs to convey not just whether the request is allowed, but also the current state information needed for rate limit headers and monitoring.

Return Field	Type	Purpose	Example Value
`allowed`	Boolean	Whether request should proceed	`1` (allowed) or `0` (denied)
`remaining`	Integer	Tokens/requests remaining in window	`45`
`reset_time`	Integer	Unix timestamp when limit resets	`1699123456`
`retry_after`	Integer	Seconds to wait before retrying	`30`
`current_count`	Integer	Current usage within window	`55`
`error`	String	Error message if script failed	`nil` or error description

Script error handling must distinguish between Redis operational errors (like out of memory) and rate limiting logic errors (like invalid parameters). Operational errors should typically cause the script to return an error result, triggering fallback behavior in the application. Logic errors should return a deny result to fail safely.

> **Architecture Decision: Lua Script Deployment Strategy**
>
> - **Context**: Lua scripts can be embedded in application code or loaded into Redis once and called by SHA hash
> - **Options Considered**:
>   1. Embed scripts in application code and use `EVAL` for each call
>   2. Load scripts at startup using `SCRIPT LOAD` and call via `EVALSHA`
>   3. Hybrid approach with fallback from `EVALSHA` to `EVAL` if script not cached
> - **Decision**: Use hybrid approach with `EVALSHA` primary and `EVAL` fallback
> - **Rationale**: Provides performance benefits of cached scripts while handling Redis restarts gracefully. Network bandwidth savings significant for complex scripts.
> - **Consequences**: Requires script loading logic at startup and error handling for cache misses, but eliminates script transmission overhead for normal operations

Connection Pool Management

Effective connection pool management forms the backbone of reliable Redis integration, determining both the performance characteristics and failure resilience of the distributed rate limiter. Unlike simple client-server applications where connection management can be relatively straightforward, a distributed rate limiter must handle high concurrency, varying load patterns, and network failures while maintaining low latency for every rate limit decision.

The fundamental challenge lies in balancing connection resource usage against performance and reliability requirements. Too few connections create bottlenecks during traffic spikes, leading to increased latency and potential timeouts. Too many connections waste system resources and can overwhelm Redis servers. The optimal pool size depends on factors that change dynamically: request volume, Redis response times, network latency, and the concurrency patterns of the application using the rate limiter.

Connection Pool Sizing Strategy

Connection pool sizing requires understanding both the mathematical relationships between throughput, latency, and concurrency, and the practical constraints of Redis server capacity and network resources. The pool must accommodate not just average load but also traffic bursts that are common in rate limiting scenarios—after all, rate limiters are most critical exactly when traffic spikes occur.

Pool Sizing Factor	Calculation Method	Typical Value Range	Monitoring Metric
Base pool size	`(average_rps * avg_latency_ms) / 1000`	5-20 connections	Connection utilization %
Burst capacity	`base_size * burst_multiplier`	2x-5x base size	Peak concurrent connections
Redis server limit	Server max connections / number of app instances	Varies by Redis config	Server connection count
Network overhead	Account for connection setup/teardown cost	10-20% buffer	Connection churn rate

The connection pool must implement intelligent sizing that adapts to observed load patterns while respecting hard limits imposed by Redis server configuration and network infrastructure. Static pool sizing often leads to either resource waste during low traffic or bottlenecks during high traffic.

Health Checking and Circuit Breaker Integration

Connection health checking in a rate limiting context goes beyond simple ping/pong tests because rate limiting requires not just connectivity but also acceptable response times. A Redis connection that takes 5 seconds to respond is effectively unusable for rate limiting, even though it's technically healthy from a connectivity perspective.

Health Check Type	Test Method	Success Criteria	Failure Response
Connectivity check	`PING` command	Response within 100ms	Mark connection unhealthy
Performance check	Simple `INCR` operation	Response within 50ms	Reduce connection priority
Functionality check	Execute minimal rate limit script	Correct result within 100ms	Flag script issues
Memory pressure check	`INFO memory` command	Used memory < 90%	Trigger degradation mode

The circuit breaker pattern becomes essential when Redis experiences problems that manifest as slow responses rather than complete failures. A Redis server under memory pressure might accept connections and even respond to commands, but with latencies that make rate limiting ineffective. The circuit breaker must detect these degraded performance conditions and trigger fallback behavior before the entire rate limiting system becomes unresponsive.

Connection Lifecycle Management

Managing the lifecycle of Redis connections involves more than simple creation and destruction—it requires handling the various states a connection can be in and the transitions between those states. Connections in a rate limiting system experience different usage patterns than typical application database connections, with potentially bursty traffic and stringent latency requirements.

Connection State	Characteristics	Transition Triggers	Pool Management Actions
Available	Idle, ready for use	Request arrives	Assign to request
Active	Executing Redis command	Command completion	Return to available pool
Degraded	Slow but functional	High latency detected	Mark for replacement
Failed	Connection error occurred	Network/Redis error	Remove and create new
Draining	Being retired gracefully	Pool size reduction	Complete current ops, close

Connection lifecycle management must handle the reality that Redis connections can fail in various ways—network timeouts, Redis server restarts, memory pressure causing slow responses, or even subtle issues like clock skew affecting time-based operations. The pool manager needs strategies for detecting each type of failure and responding appropriately without causing cascading failures in the rate limiting system.

Retry Logic and Backoff Strategies

Retry logic for Redis operations in rate limiting systems requires careful design because failed rate limit checks can't simply be retried indefinitely—the request being rate limited is waiting for a decision, and excessive retry delays defeat the purpose of rate limiting. The retry strategy must balance reliability against latency while avoiding retry storms that could worsen Redis server problems.

Failure Type	Retry Strategy	Backoff Pattern	Max Retry Time
Network timeout	2-3 retries	Linear: 10ms, 20ms, 30ms	100ms total
Connection refused	1 retry with new connection	No backoff	50ms total
Redis overload	No immediate retry	Exponential for background	Trigger fallback
Script error	1 retry, then fallback	No backoff	50ms total

The retry logic must integrate with the fallback strategy—if Redis operations are failing frequently enough to trigger retries, the system should consider switching to local fallback mode rather than continuing to hammer the failing Redis infrastructure. This requires monitoring retry rates and making intelligent decisions about when distributed rate limiting is more harmful than helpful.

> **Design Insight**: Connection pool management in distributed rate limiting differs fundamentally from typical database connection pools because rate limiting decisions are latency-critical and failure-sensitive. A database query can be retried or delayed, but a rate limiting decision that takes too long effectively becomes an allow decision, potentially compromising the entire rate limiting scheme.

Graceful Degradation Strategy

Graceful degradation represents one of the most critical aspects of distributed rate limiting design, as it determines how the system behaves when the shared state store becomes unavailable. The challenge lies in maintaining some level of rate limiting effectiveness while avoiding complete service disruption, requiring careful balance between protection and availability.

When Redis becomes unavailable, the distributed rate limiter faces a fundamental choice: fail open (allow all requests) or fail closed (deny all requests). Neither option is ideal—failing open provides no rate limiting protection and could lead to system overload, while failing closed effectively creates a denial-of-service condition. The graceful degradation strategy must provide a third option that preserves some rate limiting capability while maintaining service availability.

Local Fallback Implementation Strategy

Local fallback involves each application instance switching to per-instance rate limiting when the shared Redis backend becomes unavailable. This approach provides continued protection against abuse while maintaining service availability, though with reduced accuracy compared to true distributed rate limiting.

The key insight is that local rate limiting with adjusted limits can approximate distributed rate limiting behavior. If the distributed system normally allows 1000 requests per minute across 10 application instances, each instance can implement local rate limiting at 100 requests per minute during fallback mode. This provides similar protection levels, though with less accurate enforcement and potential for slightly higher actual limits due to uneven traffic distribution.

Fallback Aspect	Local Implementation	Accuracy Impact	Mitigation Strategy
Rate limit scaling	Divide distributed limit by instance count	Uneven traffic causes over/under limiting	Use dynamic instance discovery
State isolation	Each instance tracks separately	No cross-instance coordination	Monitor aggregate metrics
Rule synchronization	Use last known good configuration	Rules may become stale	Cache rules with TTL
Metrics collection	Local counters only	Missing distributed view	Aggregate in monitoring system

The local fallback implementation must handle the transition periods carefully—when Redis becomes available again, instances shouldn't immediately switch back to distributed mode, as this could cause thundering herd problems. Instead, a gradual transition with health checking ensures stable operation.

Failure Detection and Mode Switching

Accurate failure detection determines how quickly the system can switch to fallback mode and how effectively it can detect recovery. The failure detection strategy must distinguish between temporary network hiccups that should be retried and genuine Redis failures that require fallback activation.

Detection Signal	Threshold	Confidence Level	Action Triggered
Connection timeout	3 consecutive failures	High	Immediate fallback
Slow response	>500ms for 10 requests	Medium	Gradual degradation
Redis memory errors	Any OOM response	High	Immediate fallback
Script execution errors	5 in 60 seconds	Medium	Disable scripts, use simple commands
Network partitions	No response for 30s	High	Full fallback mode

Mode switching must be implemented with hysteresis to prevent oscillation between distributed and local modes. The criteria for entering fallback mode should be more sensitive than the criteria for returning to distributed mode, ensuring that the system doesn't constantly switch back and forth during marginal conditions.

Rate Limit Accuracy During Degradation

During fallback mode, rate limiting accuracy degrades in predictable ways that must be understood and monitored. The degradation patterns help operations teams understand the current protection level and make informed decisions about additional protective measures.

Degradation Scenario	Accuracy Impact	Burst Behavior	Recommended Monitoring
Even traffic distribution	90-95% of intended limit	Normal burst handling	Per-instance rate metrics
Uneven traffic (80/20 split)	60-120% of intended limit	Some instances allow full bursts	Traffic distribution monitoring
Single hot instance	Up to 200% of intended limit	Full burst on hot instance	Instance-level alerting
Partial Redis availability	Mixed accuracy across instances	Inconsistent burst behavior	Redis connectivity per instance

Understanding these accuracy patterns allows the system to provide meaningful rate limit headers even during degradation. Clients can receive information about the current rate limiting state and adjust their behavior accordingly.

Recovery and State Synchronization

Recovery from fallback mode requires careful orchestration to prevent thundering herd effects and ensure smooth transition back to distributed operation. The challenge lies in synchronizing state between the local fallback counters and the Redis-based distributed state without causing sudden changes in rate limiting behavior.

Recovery Phase	Actions	Validation	Rollback Criteria
Redis availability confirmation	Health checks pass for 60s	Script execution successful	Any health check failure
Gradual transition start	10% of requests use Redis	Compare local vs distributed decisions	>50% decision disagreement
Transition scaling	Increase to 50%, then 90%	Monitor error rates	Redis error rate >1%
Full distributed mode	100% requests use Redis	Full functionality restored	Sustained high error rate

The state synchronization strategy must handle the reality that local counters during fallback mode may not accurately represent what the distributed state should be. Rather than trying to perfectly synchronize state, the recovery process should focus on ensuring that the transition doesn't create sudden changes in rate limiting behavior that could surprise clients or cause traffic spikes.

Architecture Decision: Fallback Trigger Sensitivity

- **Context**: Need to balance between false positives (unnecessary fallbacks) and false negatives (delayed fallback during real failures)
- **Options Considered**:
  1. Conservative: Only fallback on complete Redis failure
  2. Aggressive: Fallback on any performance degradation
  3. Adaptive: Adjust sensitivity based on recent failure patterns
- **Decision**: Adaptive approach with configurable base sensitivity
- **Rationale**: Different deployments have different tolerance for degraded Redis performance. Some can handle 200ms Redis responses, others need <50ms for effective rate limiting.
- **Consequences**: Requires more complex configuration and monitoring, but provides better operational flexibility and fewer false positive fallbacks

Redis Backend ADR

The choice of backend storage for distributed rate limiting state represents one of the most fundamental architectural decisions in the system, affecting everything from performance characteristics to operational complexity. This decision impacts not just the immediate implementation but also long-term scalability, operational procedures, and integration patterns with existing infrastructure.

Architecture Decision: Redis as Primary Backend Storage

- **Context**: Need shared storage for rate limiting state that supports atomic operations, high performance, and horizontal scaling. Must handle thousands of rate limit checks per second across multiple application instances with sub-10ms latency requirements.
- **Options Considered**:
  1. Redis with Lua scripts for atomic operations
  2. etcd with compare-and-swap operations for consistency
  3. PostgreSQL with advisory locks and ACID transactions
  4. DynamoDB with conditional writes and TTL
  5. Cassandra with lightweight transactions
- **Decision**: Redis with Lua scripts as primary backend, with etcd as alternative for environments requiring strong consistency
- **Rationale**: Redis provides the optimal combination of performance (sub-millisecond operations), atomic operation support (Lua scripts), memory efficiency for time-series data, and operational maturity for high-traffic systems.
- **Consequences**: Requires Redis operational expertise, introduces eventual consistency considerations during network partitions, but provides excellent performance and horizontal scaling capabilities.

Detailed Options Analysis

The storage backend decision required extensive analysis of how each option handles the specific requirements of distributed rate limiting, particularly around atomic operations, performance characteristics, and operational complexity.

Storage Option	Atomic Operations	Typical Latency	Memory Efficiency	Operational Complexity	Horizontal Scaling
Redis + Lua	Lua scripts	0.1-1ms	Excellent	Medium	Hash-based sharding
etcd	Compare-and-swap	1-5ms	Good	High	Raft consensus
PostgreSQL	ACID transactions	5-20ms	Poor for counters	High	Read replicas only
DynamoDB	Conditional writes	10-50ms	Good	Low	Automatic
Cassandra	Lightweight transactions	5-15ms	Good	Very High	Excellent

Redis emerged as the optimal choice primarily due to its combination of performance and atomic operation support. Lua scripts provide true atomicity for complex rate limiting algorithms, while Redis's in-memory architecture delivers the low latency required for inline rate limiting decisions. The memory efficiency for storing counters and timestamps makes Redis particularly well-suited for the access patterns typical in rate limiting workloads.

etcd Comparison and Use Cases

etcd represents the primary alternative to Redis, particularly in environments where strong consistency requirements outweigh performance considerations. Understanding when to choose etcd over Redis helps inform deployment decisions and architectural trade-offs.

Consideration	Redis Advantages	etcd Advantages
Consistency model	Eventual consistency, faster	Strong consistency, slower
Performance	Sub-millisecond operations	Millisecond operations
Operational complexity	Familiar to most teams	Requires Raft understanding
Failure handling	Manual failover or Redis Sentinel	Automatic leader election
Multi-datacenter	Complex setup	Native support
Kubernetes integration	Requires external setup	Often pre-installed

etcd becomes the preferred choice in environments where rate limiting must integrate with existing Kubernetes control plane infrastructure, or where strong consistency requirements justify the performance trade-offs. For example, financial systems might prefer etcd's consistency guarantees even at the cost of higher latency.

PostgreSQL and Traditional Database Analysis

Traditional relational databases like PostgreSQL initially seem attractive for rate limiting because they provide strong consistency guarantees and familiar operational models. However, deeper analysis reveals fundamental mismatches with rate limiting requirements.

The primary challenge with PostgreSQL for rate limiting lies in the access patterns. Rate limiting requires high-frequency updates to counter values with minimal read complexity—essentially the inverse of typical web application database usage. PostgreSQL's MVCC (Multi-Version Concurrency Control) system creates overhead for high-frequency counter updates, and the persistence guarantees designed for critical business data become unnecessary overhead for rate limiting state.

Database Limitation	Impact on Rate Limiting	Mitigation Cost
MVCC overhead for updates	Higher CPU usage for counters	Requires more database capacity
Disk I/O for durability	Slower response times	SSD required, higher costs
Connection overhead	Fewer concurrent rate checks	Larger connection pools needed
Limited atomic operations	Complex application logic	More application complexity

PostgreSQL remains viable for rate limiting in environments where extreme consistency requirements justify the performance costs, or where existing PostgreSQL expertise and infrastructure make operational complexity the primary concern.

Cloud-Native Options Analysis

Cloud-native storage options like DynamoDB offer compelling operational simplicity but introduce different trade-offs around performance predictability and cost scaling.

DynamoDB's conditional write operations provide the atomicity needed for rate limiting, and automatic scaling eliminates capacity planning concerns. However, the performance characteristics vary significantly based on provisioned capacity and hot key patterns common in rate limiting workloads. The pricing model also creates challenges for high-traffic rate limiting where the cost can become substantial.

Cloud Option	Operational Burden	Performance Predictability	Cost Scaling	Lock-in Risk
DynamoDB	Very Low	Variable	High at scale	High
Cloud Redis	Low	Predictable	Moderate	Medium
Cloud SQL	Medium	Predictable	Low	Low

The choice between cloud-native and self-managed options often depends more on organizational factors than technical requirements. Teams with strong infrastructure automation capabilities may prefer self-managed Redis for cost control and performance predictability, while teams prioritizing operational simplicity may accept the trade-offs of managed cloud services.

Future Migration Considerations

The storage backend decision should consider not just current requirements but also likely evolution paths. Rate limiting systems often start simple and grow more sophisticated over time, requiring migration strategies that minimize service disruption.

Redis provides good migration paths to other storage options because its simple key-value model can be replicated in most other systems. The atomic operation patterns established using Lua scripts can be translated to stored procedures in databases or conditional operations in other NoSQL systems.

The modular storage interface design allows for backend migration without changing the core rate limiting logic. This architectural separation enables gradual migration strategies where different rate limiting tiers or different types of keys can use different storage backends during transition periods.

> **Design Insight**: The storage backend choice for distributed rate limiting differs significantly from typical application data storage decisions. Rate limiting requires high write throughput with simple access patterns, making it more similar to metrics collection or event streaming workloads than traditional CRUD operations. This fundamental difference in access patterns explains why Redis often outperforms traditional databases for this specific use case.

Implementation Guidance

This implementation guidance focuses on building production-ready Redis integration for distributed rate limiting, with emphasis on the connection management, atomic operations, and graceful degradation patterns that separate robust systems from fragile prototypes.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Redis Client	`go-redis/redis/v8` with basic clustering	`go-redis/redis/v8` with Redis Sentinel for HA
Connection Pooling	Built-in go-redis connection pool	Custom pool with circuit breaker integration
Lua Script Management	Embedded scripts with EVALSHA fallback	External script files with hot reload
Health Checking	Simple PING commands	Comprehensive health with performance metrics
Configuration Management	Static YAML configuration	Dynamic config with Redis pub/sub updates
Monitoring Integration	Basic metrics collection	Prometheus metrics with custom collectors

For production deployments, the advanced options provide the reliability and observability needed to debug issues under load. The simple options work well for development and testing environments.

B. Recommended Module Structure

```
internal/redis/
├── client.go              ← Redis client wrapper with pooling
├── scripts/
│   ├── token_bucket.lua    ← Token bucket Lua script
│   ├── sliding_counter.lua ← Sliding window counter script
│   └── sliding_log.lua     ← Sliding window log script
├── storage.go             ← RedisStorage implementation
├── health.go              ← Health checking and circuit breaker
├── fallback.go            ← Local fallback implementation
└── config.go              ← Redis configuration structures

internal/scripts/
├── loader.go              ← Lua script loading and caching
└── registry.go           ← Script SHA management
```

This structure separates Redis-specific concerns from the core rate limiting algorithms while providing clear boundaries for testing and future backend alternatives.

C. Infrastructure Starter Code

Redis Configuration and Client Setup

```go
package redis

import (
    "context"
    "time"

    "github.com/go-redis/redis/v8"
)

type RedisConfig struct {
    Addresses      []string      `yaml:"addresses"`
    Password       string        `yaml:"password"`
    DB             int           `yaml:"db"`
    PoolSize       int           `yaml:"pool_size"`
    ReadTimeout    time.Duration `yaml:"read_timeout"`
    WriteTimeout   time.Duration `yaml:"write_timeout"`
    DialTimeout    time.Duration `yaml:"dial_timeout"`
}

func NewRedisStorage(config RedisConfig) (*RedisStorage, error) {
    var client redis.UniversalClient

    if len(config.Addresses) == 1 {
        client = redis.NewClient(&redis.Options{
            Addr:         config.Addresses[0],
            Password:     config.Password,
            DB:           config.DB,
            PoolSize:     config.PoolSize,
            ReadTimeout:  config.ReadTimeout,
            WriteTimeout: config.WriteTimeout,
            DialTimeout:  config.DialTimeout,
        })
    } else {
        client = redis.NewClusterClient(&redis.ClusterOptions{
            Addrs:        config.Addresses,
            Password:     config.Password,
            PoolSize:     config.PoolSize,
            ReadTimeout:  config.ReadTimeout,
            WriteTimeout: config.WriteTimeout,
```

```
            DialTimeout:  config.DialTimeout,
        })
    }

    storage := &RedisStorage{
        client: client,
        config: config,
        scripts: make(map[string]string),
    }

    // Load Lua scripts on startup
    if err := storage.loadScripts(); err != nil {
        return nil, fmt.Errorf("failed to load scripts: %w", err)
    }

    return storage, nil
}
```

Lua Script Management

```go
package redis

import (
    _ "embed"
    "crypto/sha1"
    "fmt"
)

//go:embed scripts/token_bucket.lua
var tokenBucketScript string

//go:embed scripts/sliding_counter.lua
var slidingCounterScript string

//go:embed scripts/sliding_log.lua
var slidingLogScript string

type RedisStorage struct {
    client  redis.UniversalClient
    config  RedisConfig
    scripts map[string]string // script name -> SHA hash
}

func (r *RedisStorage) loadScripts() error {
    scripts := map[string]string{
        "token_bucket":    tokenBucketScript,
        "sliding_counter": slidingCounterScript,
        "sliding_log":     slidingLogScript,
    }

    for name, script := range scripts {
        sha, err := r.client.ScriptLoad(context.Background(), script).Result()
        if err != nil {
            return fmt.Errorf("failed to load script %s: %w", name, err)
        }
        r.scripts[name] = sha
    }

    return nil
}
```

```
func (r *RedisStorage) ExecuteLua(ctx context.Context, script string, keys []string, args []interface{}) (interface{}, error) {

    sha, exists := r.scripts[script]

    if !exists {

        return nil, fmt.Errorf("script %s not found", script)

    }



    // Try EVALSHA first for performance

    result, err := r.client.EvalSha(ctx, sha, keys, args...).Result()

    if err != nil {

        // If script not in cache, fall back to EVAL

        if err.Error() == "NOSCRIPT No matching script. Please use EVAL." {

            result, err = r.client.Eval(ctx, r.getScriptContent(script), keys, args...).Result()

        }

    }



    return result, err

}
```

Health Checking with Circuit Breaker

```go
package redis

import (
    "context"
    "sync"
    "time"
)

type HealthChecker struct {
    storage         *RedisStorage
    mu              sync.RWMutex
    healthy         bool
    lastCheck       time.Time
    failureCount    int
    circuitOpen     bool
    nextRetryTime   time.Time
}

func NewHealthChecker(storage *RedisStorage) *HealthChecker {
    hc := &HealthChecker{
        storage: storage,
        healthy: true,
    }

    go hc.startHealthChecking()
    return hc
}

func (hc *HealthChecker) IsHealthy() bool {
    hc.mu.RLock()
    defer hc.mu.RUnlock()
    return hc.healthy && !hc.circuitOpen
}

func (hc *HealthChecker) startHealthChecking() {
    ticker := time.NewTicker(5 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        hc.performHealthCheck()
```

```
    }
}

func (hc *HealthChecker) performHealthCheck() {
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()

    start := time.Now()
    err := hc.storage.client.Ping(ctx).Err()
    latency := time.Since(start)

    hc.mu.Lock()
    defer hc.mu.Unlock()

    if err != nil || latency > 100*time.Millisecond {
        hc.failureCount++
        if hc.failureCount >= 3 {
            hc.healthy = false
            hc.circuitOpen = true
            hc.nextRetryTime = time.Now().Add(30 * time.Second)
        }
    } else {
        hc.failureCount = 0
        hc.healthy = true
        if time.Now().After(hc.nextRetryTime) {
            hc.circuitOpen = false
        }
    }

    hc.lastCheck = time.Now()
}
```

D. Core Logic Skeleton Code

Token Bucket Redis Implementation

```go
func (r *RedisStorage) CheckAndUpdate(ctx context.Context, key string, limit int64, window time.Duration) (bool, int64,
time.Time, error) {

    // TODO 1: Prepare Redis key for token bucket (prefix + key)

    // TODO 2: Get current timestamp in nanoseconds for precise timing

    // TODO 3: Calculate refill rate as tokens per nanosecond (limit / window)

    // TODO 4: Execute token bucket Lua script with key, current time, limit, refill rate, requested tokens

    // TODO 5: Parse script result - [allowed, remaining_tokens, reset_time]

    // TODO 6: Convert reset_time from nanoseconds to time.Time

    // TODO 7: Return whether allowed, remaining tokens, and reset time

    // Hint: Use time.Now().UnixNano() for high precision timestamps

    // Hint: Script should handle token refill calculation atomically

}
```

Sliding Window Counter Implementation

```go
func (r *RedisStorage) CheckSlidingWindow(ctx context.Context, key string, limit int64, window time.Duration, buckets int)
(*RateLimitResult, error) {

    // TODO 1: Calculate current bucket ID based on current time and bucket size

    // TODO 2: Determine which buckets fall within the current window

    // TODO 3: Execute sliding counter Lua script with key, current bucket, window buckets, limit

    // TODO 4: Parse script result - [allowed, current_count, window_total, oldest_bucket]

    // TODO 5: Calculate remaining quota (limit - window_total)

    // TODO 6: Calculate reset time (when oldest bucket expires)

    // TODO 7: Return structured RateLimitResult with all computed values

    // Hint: Bucket size = window / buckets, bucket ID = current_time / bucket_size

    // Hint: Script should clean up expired buckets to prevent memory growth

}
```

Graceful Degradation Manager

```go
func (d *DistributedLimiter) Check(ctx context.Context, req RateLimitRequest) (*RateLimitResult, error) {

    // TODO 1: Check if Redis backend is healthy using health checker

    // TODO 2: If healthy, attempt Redis-based rate limiting with timeout

    // TODO 3: If Redis fails or times out, record failure for circuit breaker

    // TODO 4: On Redis failure, switch to local fallback with adjusted limits

    // TODO 5: Scale down limits for local mode (divide by estimated instance count)

    // TODO 6: Include fallback mode indicator in result for monitoring

    // TODO 7: Implement retry logic with exponential backoff for Redis recovery

    // Hint: Use context.WithTimeout for Redis operations to prevent hanging

    // Hint: Local limits should be distributed_limit / instance_count

}
```

E. Language-Specific Hints for Go

- Use `go-redis/redis/v8` for Redis connectivity with built-in connection pooling
- Embed Lua scripts using `//go:embed` directive to avoid runtime file dependencies
- Implement proper context cancellation for all Redis operations to prevent goroutine leaks
- Use `sync.RWMutex` for health checker state to allow concurrent reads
- Consider using `time.UnixNano()` for high-precision timestamps in rate limiting calculations
- Implement proper error wrapping with `fmt.Errorf("operation failed: %w", err)`

F. Milestone Checkpoint

After implementing Redis backend integration, verify the following functionality:

Basic Connectivity Test:

```bash
# Start Redis locally or use Docker
docker run -d --name redis-test -p 6379:6379 redis:7

# Test basic Redis operations
go run cmd/redis-test/main.go
```

Expected behavior: Connection established, Lua scripts loaded successfully, health checker reports healthy status.

Rate Limiting Functionality Test:

```bash
# Run integration test with Redis backend
go test -v ./internal/redis/... -tags=integration
```

Expected behavior: Token bucket script executes correctly, sliding window counters increment properly, fallback triggers when Redis is stopped.

Performance Verification:

```bash
# Benchmark Redis vs local performance
go test -bench=BenchmarkRedisRateLimit ./internal/redis/
```

Expected results: Redis operations complete in <5ms p99, throughput >1000 ops/sec per connection.

Signs of problems and debugging steps:

- **Scripts not loading**: Check Redis version (requires 2.6+), verify script syntax
- **Connection timeouts**: Check network connectivity, Redis memory usage, connection pool size
- **Inconsistent results**: Verify system clock synchronization, check for Redis clustering issues

Consistent Hashing and Sharding

> **Milestone(s):** Milestone 4 - Consistent Hashing & Sharding

The transition from single-node Redis to a distributed Redis cluster represents a fundamental scaling challenge in distributed rate limiting. While a single Redis instance can handle thousands of rate limit checks per second, production systems often require tens or hundreds of thousands of operations per second across millions of rate limit keys. This milestone transforms our centralized rate limiting system into a horizontally scalable distributed system that can grow by adding more Redis nodes while maintaining consistent performance and minimizing operational complexity during cluster topology changes.

Mental Model: Library Book Distribution

Think of distributing rate limit state across multiple Redis nodes like managing a university library system with multiple branch locations. In a traditional single-location library, all books are stored in one building, and patrons must visit that specific location to access any book. This works well for small collections, but as the collection grows to millions of books and thousands of daily visitors, the single location becomes overwhelmed.

The solution is to distribute books across multiple branch libraries using a systematic approach. However, you cannot randomly scatter books across branches—patrons need a predictable way to find any specific book. A naive approach might be alphabetical distribution: books A-F go to Branch 1, G-M to Branch 2, and so on. But this creates problems when you need to add a new branch—suddenly you must physically move thousands of books to rebalance the collection, disrupting service for weeks.

Consistent hashing solves this problem elegantly. Instead of dividing the alphabet into fixed ranges, imagine arranging the branches in a circle based on their unique characteristics (like their postal codes). Each book is assigned to the first branch you encounter when walking clockwise from the book's hash position on the circle. When you add a new branch, you only need to move books from one adjacent branch to maintain balance. When a branch temporarily closes for maintenance, patrons are automatically redirected to the next available branch clockwise.

Hot key detection is like noticing that certain popular textbooks are being requested so frequently at one branch that students form long lines. The library system responds by placing copies of these popular books at multiple branches, reducing the load on any single location.

This library analogy maps directly to our distributed rate limiting system:

- **Books** → Rate limit keys (user:123, api:/login, ip:192.168.1.1)
- **Branch libraries** → Redis nodes in the cluster
- **Book locations** → Which Redis node stores each rate limit counter
- **Catalog lookup** → Consistent hash function determining node assignment
- **Branch closure** → Redis node failure requiring automatic failover
- **Popular textbooks** → Hot keys that need replication across multiple nodes
- **Adding new branches** → Scaling by adding Redis nodes with minimal data movement

Consistent Hash Ring Design

The consistent hash ring forms the mathematical foundation for distributing rate limit keys across Redis nodes while minimizing redistribution during topology changes. Unlike traditional hash-based sharding where adding nodes requires rehashing all keys, consistent hashing ensures that only a small fraction of keys need to move when the cluster topology changes.

Hash Ring Mathematics and Virtual Nodes

The consistent hash ring maps both Redis nodes and rate limit keys onto a circular hash space, typically using SHA-1 or SHA-256 to produce a 160-bit or 256-bit hash space. The ring conceptually represents all possible hash values arranged in a circle, where the maximum hash value wraps around to zero.

Each Redis node is assigned multiple positions on the ring called **virtual nodes** or **vnodes**. Virtual nodes solve the fundamental problem of uneven load distribution that occurs when physical nodes are hashed to random positions on the ring. Without virtual nodes, adding or removing a single physical node could create scenarios where one node handles 80% of the keys while others handle only 5%.

Virtual nodes work by creating multiple hash positions for each physical Redis node using different hash inputs. For example, if Redis node `redis-1` at address `10.0.1.100:6379` uses 150 virtual nodes, we generate positions by hashing:

- `redis-1:vnode:0` → hash position 0x1a2b3c4d...
- `redis-1:vnode:1` → hash position 0x5e6f7g8h...
- `redis-1:vnode:2` → hash position 0x9i0j1k2l...
- ... continuing for all 150 virtual nodes

This creates 150 different positions on the ring where `redis-1` is responsible for handling keys. The more virtual nodes per physical node, the more evenly distributed the load becomes, approaching perfect balance as the virtual node count increases.

Virtual Nodes Per Physical Node	Expected Load Variance	Memory Overhead	Rebalancing Efficiency
50	±15% from perfect balance	Low	Good
150	±8% from perfect balance	Medium	Better
500	±3% from perfect balance	High	Excellent
1000	±2% from perfect balance	Very High	Excellent

Decision: Virtual Node Count Selection

- **Context**: Need to balance load distribution accuracy against memory overhead and lookup performance
- **Options Considered**: 50, 150, 500, or 1000 virtual nodes per physical node
- **Decision**: Use 150 virtual nodes per physical node as the default
- **Rationale**: Provides ±8% load variance which is acceptable for rate limiting workloads, while keeping memory overhead reasonable and maintaining fast lookup performance. Can be configured higher for clusters with extreme hot key problems.
- **Consequences**: Enables good load balance with reasonable memory usage. Lookup time increases slightly due to larger ring structure, but remains O(log N) with binary search.

Key Assignment Algorithm

Rate limit keys are assigned to Redis nodes through a deterministic process that ensures any application instance can independently determine which node handles any specific key without central coordination. The algorithm follows these steps:

1. **Hash the rate limit key**: Apply the same hash function (SHA-256) to the complete rate limit key string. For example, the key `user:12345:api:/login:1m` produces a 256-bit hash value.
2. **Locate position on ring**: The hash value represents a position on the circular hash space. Conceptually, this is like dropping a pin at a specific location on the ring.
3. **Find responsible virtual node**: Walk clockwise from the key's position until encountering the first virtual node. This virtual node's physical Redis node is responsible for storing this key's rate limit state.
4. **Handle ring wrap-around**: If no virtual node exists between the key's position and the maximum hash value, wrap around to the beginning of the ring and continue searching from zero.

The beauty of this algorithm lies in its consistency—every application instance performing the same calculation will always arrive at the same Redis node for any given key, without requiring centralized coordination or shared state.

Hash Ring Operation	Time Complexity	Space Complexity	Consistency Guarantee
Key lookup	O(log V) where V = total virtual nodes	O(V) for ring storage	Deterministic - all nodes agree
Node addition	O(V/N) keys move where N = physical nodes	O(V) ring restructure	Affects only adjacent ranges
Node removal	O(V/N) keys move	O(V) ring restructure	Graceful failover to next node
Ring rebalancing	O(V log V) for sorting	O(V) temporary storage	Eventually consistent

Minimizing Redistribution During Topology Changes

The consistent hash ring's primary advantage becomes apparent during cluster topology changes. When a new Redis node joins the cluster, it only takes responsibility for keys that fall within specific ranges on the ring, rather than triggering a complete reshuffling of all keys.

Adding a new node follows this process:

1. **Generate virtual node positions**: The new physical node generates its virtual nodes at deterministic positions on the ring based on its identifier and virtual node indices.
2. **Identify affected ranges**: For each virtual node of the new physical node, determine the range of keys that will move from the previously responsible node. This range spans from the new virtual node's position back to the previous virtual node in the clockwise direction.
3. **Coordinate data migration**: The application instances coordinate with both the old and new Redis nodes to migrate rate limit state for keys within the affected ranges. This migration must maintain atomicity to prevent rate limit violations during the transition.
4. **Update routing tables**: All application instances update their consistent hash ring structures to include the new node's virtual nodes, ensuring future requests route correctly.

Removing a node (whether planned or due to failure) reverses this process:

1. **Identify orphaned ranges**: Determine which key ranges were handled by the departing node's virtual nodes.
2. **Reassign to next nodes**: For each orphaned range, the responsibility transfers to the next virtual node clockwise on the ring.
3. **Migrate remaining state**: If the departure was planned, migrate any remaining rate limit state to the newly responsible nodes. If the departure was due to failure, the rate limit state is lost, but the system continues operating with temporary accuracy degradation.

4. **Clean up routing tables**: Remove the departed node's virtual nodes from all application instances' hash ring structures.

The mathematical guarantee of consistent hashing is that adding or removing one node affects at most O(K/N) keys, where K is the total number of keys and N is the number of nodes. In practice, this means adding a fifth node to a four-node cluster will move approximately 20% of the keys, rather than the 100% that would move with traditional hash-based sharding.

Hot Key Detection and Rebalancing

Even with perfect hash distribution, real-world traffic patterns create **hot keys**—rate limit keys that receive disproportionately high request volumes compared to the average. Hot keys can overwhelm individual Redis nodes while leaving others underutilized, creating performance bottlenecks that undermine the benefits of horizontal scaling.

Hot Key Identification Mechanisms

Hot key detection operates through statistical analysis of request patterns across time windows, combined with cross-node comparison to identify keys that exceed normal distribution expectations. The detection system must balance accuracy against overhead, since monitoring every key individually would consume excessive resources.

Request frequency tracking maintains sliding window counters for key access patterns. Each application instance tracks the frequency of rate limit checks for individual keys over rolling time windows (typically 1-minute, 5-minute, and 15-minute windows). Keys that consistently appear in the top percentiles across multiple time windows become candidates for hot key classification.

Cross-node load comparison identifies keys that create uneven load distribution. The hot key detection system periodically samples key access frequencies across all Redis nodes and identifies keys where a single node handles a disproportionate share of the total cluster traffic. A key is classified as hot if it represents more than a configurable threshold (typically 2-5%) of any single node's request volume.

Adaptive thresholds prevent false positives during normal traffic variations. The detection system calculates dynamic thresholds based on overall cluster load patterns, ensuring that keys are only classified as hot when they create genuine bottlenecks rather than normal peak traffic.

Detection Method	Accuracy	Overhead	Detection Latency	False Positive Rate
Request frequency only	Medium	Low	1-5 minutes	Medium
Cross-node comparison	High	Medium	2-10 minutes	Low
Combined approach	Very High	Medium-High	1-5 minutes	Very Low
Real-time statistical analysis	Excellent	High	30 seconds	Very Low

Decision: Hot Key Detection Strategy

- **Context**: Need to identify problematic keys without adding significant overhead to normal operations
- **Options Considered**: Request frequency tracking only, cross-node comparison only, combined approach, or real-time statistical analysis
- **Decision**: Use combined approach with request frequency tracking and periodic cross-node comparison
- **Rationale**: Provides high accuracy with reasonable overhead. Real-time analysis is too expensive for most workloads, while single-method approaches have too many false positives or negatives.
- **Consequences**: Achieves good hot key detection with manageable overhead. May miss very short-lived hot keys (under 1 minute duration), but these rarely cause sustained performance problems.

Hot Key Replication Strategy

Once hot keys are identified, the system must distribute their load across multiple Redis nodes while maintaining consistency and avoiding race conditions. Hot key replication creates multiple copies of a rate limit counter across different nodes, requiring careful coordination to prevent double-counting or lost updates.

Read replica distribution creates read-only copies of hot key state on multiple Redis nodes. When a hot key is detected, the system creates replicas on 2-3 additional nodes chosen to minimize impact on ring distribution. Read requests for rate limit previews can be served from any replica, distributing the query load. However, all write operations (actual rate limit checks that decrement counters) must still be directed to the primary node to maintain consistency.

Write load distribution splits hot key write operations across multiple nodes using key suffixing. Instead of storing all state for hot key `user:popular_user:api:/login:1m` on a single node, the system creates multiple suffixed keys:

- `user:popular_user:api:/login:1m:shard:0`
- `user:popular_user:api:/login:1m:shard:1`

- `user:popular_user:api:/login:1m:shard:2`

Write operations are distributed across these sharded keys using consistent hashing on the client identifier or request timestamp. Rate limit checks aggregate state from all shards to determine the total usage.

Consistency maintenance ensures that replicated or sharded hot keys maintain accurate rate limit enforcement. For read replicas, the primary node periodically synchronizes state to replicas (typically every 10-30 seconds). For sharded writes, rate limit checks must query all shards and aggregate results, adding latency but distributing load.

Automatic Rebalancing Triggers

The system automatically initiates rebalancing operations when hot key detection identifies sustained performance problems or when cluster topology changes create uneven load distribution. Rebalancing must be coordinated carefully to avoid disrupting active rate limiting operations.

Load threshold monitoring triggers rebalancing when key access patterns create sustained imbalance. If any Redis node consistently handles more than a configurable percentage (typically 40-50%) of total cluster requests for more than a sustained period (typically 10-15 minutes), the system initiates automatic rebalancing.

Cluster topology rebalancing activates after node additions or removals complete their initial data migration. Even though consistent hashing minimizes key movement, the addition or removal of nodes can still create scenarios where hot keys concentrate on a subset of nodes. The rebalancing system identifies these patterns and creates additional replicas or shards as needed.

Time-based rebalancing windows schedule major rebalancing operations during periods of lower traffic to minimize impact on active rate limiting. The system maintains historical traffic patterns and automatically schedules rebalancing during identified low-traffic windows, typically during off-peak hours.

Rebalancing Trigger	Threshold	Response Time	Impact on Performance
Sustained load imbalance	>50% requests on single node for >15min	5-10 minutes	Low - gradual replica creation
Hot key detection	>5% cluster requests for single key	2-5 minutes	Medium - immediate sharding
Post-topology change	After node add/remove completion	1-2 hours	Low - background optimization
Scheduled maintenance	Daily during off-peak hours	Immediate	Very Low - planned window

Node Health and Failover

Redis node failures represent the most critical operational challenge in distributed rate limiting, since failure of a single node can immediately impact rate limit accuracy for thousands of keys. The health monitoring and failover system must detect failures quickly, route traffic away from failed nodes, and maintain system availability while minimizing rate limit violations during transitions.

Health Check Implementation

Health checking monitors multiple indicators of Redis node availability and performance to distinguish between temporary network glitches and genuine node failures. The health check system must balance rapid failure detection against false positive alerts that could trigger unnecessary failovers.

Connection-level health checks verify basic network connectivity and Redis protocol responsiveness. Each application instance maintains persistent connections to all Redis nodes in the cluster and performs periodic ping operations (typically every 5-10 seconds). Failed ping operations increment failure counters, while successful operations reset counters and update last-seen timestamps.

Operation-level health checks monitor the success rate and latency of actual rate limiting operations rather than just connection availability. These checks perform lightweight rate limit operations on synthetic keys and measure both success rates and response times. Nodes that consistently return errors or exceed latency thresholds (typically 50-100ms for rate limit checks) are marked as degraded even if basic connectivity remains functional.

Memory and resource monitoring tracks Redis memory usage, CPU utilization, and key eviction rates to identify nodes approaching resource exhaustion. Redis nodes under memory pressure may start evicting rate limit keys or slowing response times significantly before completely failing. Early detection allows for graceful traffic redirection before performance degrades severely.

Health Check Type	Frequency	Failure Threshold	Detection Time	False Positive Rate
Basic connectivity ping	Every 5 seconds	3 consecutive failures	15-20 seconds	Low
Rate limit operation check	Every 30 seconds	5 failures in 2 minutes	1-3 minutes	Very Low
Memory pressure monitoring	Every 60 seconds	>90% memory usage	2-5 minutes	Medium
Latency threshold monitoring	Continuous	>100ms for >1 minute	1-2 minutes	Medium-High

Circuit Breaker Pattern Implementation

The circuit breaker pattern protects the rate limiting system from cascading failures by automatically stopping requests to failed Redis nodes and routing traffic to healthy alternatives. Circuit breakers prevent the system from repeatedly attempting operations against known-failed nodes, which would add latency and consume resources without providing value.

Circuit states define the operational mode for each Redis node connection:

- **Closed state**: Normal operation where all requests are sent to the Redis node. Connection failures increment failure counters, but the circuit remains closed until failure thresholds are exceeded.
- **Open state**: All requests to the Redis node are immediately rejected without attempting connection. The circuit breaker routes traffic to alternative nodes or triggers local fallback behavior. The circuit remains open for a configurable timeout period (typically 30-60 seconds).
- **Half-open state**: After the timeout period expires, the circuit allows a limited number of test requests to determine if the Redis node has recovered. If test requests succeed, the circuit closes and normal operation resumes. If test requests fail, the circuit returns to the open state for another timeout period.

Failure threshold configuration determines when circuits open based on error rates and timing patterns. Typical configurations open circuits after 5 consecutive failures or 50% error rate over a 2-minute sliding window. The system must balance rapid failure detection against temporary network issues that resolve quickly.

Recovery validation ensures that nodes are genuinely healthy before resuming full traffic. Half-open state test requests perform actual rate limiting operations rather than simple ping commands, verifying that the node can handle real workload before closing the circuit.

Circuit State	Request Handling	Failure Counting	State Transition
Closed	Send all requests to node	Increment on failure, reset on success	Open after threshold exceeded
Open	Reject immediately, route to alternatives	No requests sent	Half-open after timeout
Half-open	Send limited test requests	Evaluate test request results	Closed on success, Open on failure

Automatic Failover Coordination

When Redis nodes fail, the consistent hash ring must automatically redirect affected keys to healthy nodes while maintaining rate limit accuracy and avoiding split-brain scenarios where multiple nodes believe they are responsible for the same keys.

Immediate traffic redirection routes new rate limit requests away from failed nodes using the next available node clockwise on the consistent hash ring. This redirection happens automatically since each application instance independently detects node failures and updates its local hash ring state. No central coordination is required for basic traffic redirection.

State migration coordination attempts to preserve rate limit state from failed nodes when possible. If a node failure is detected early enough, the system may attempt to read current rate limit counter values and migrate them to the newly responsible nodes. However, this migration is best-effort only—the system prioritizes availability over perfect accuracy during failure scenarios.

Split-brain prevention ensures that when failed nodes recover, they do not create conflicting rate limit state. Recovered nodes must synchronize with the current cluster state and determine which key ranges they are currently responsible for according to the updated hash ring. Any rate limit state for keys that were reassigned during the failure must be discarded or merged carefully to prevent double-counting.

> **Decision: Failover Strategy Priority**
>
> - **Context**: Must balance rate limit accuracy against system availability during node failures
> - **Options Considered**: Perfect accuracy with downtime, immediate availability with temporary inaccuracy, or hybrid approach with best-effort state preservation
> - **Decision**: Prioritize availability with best-effort state preservation
> - **Rationale**: Rate limiting systems must remain operational during infrastructure failures. Temporary rate limit inaccuracy (allowing slightly more requests than configured limits) is preferable to complete service outage. Perfect accuracy is impossible to guarantee during arbitrary failure scenarios.
> - **Consequences**: Enables high availability during Redis failures at the cost of temporary rate limit violations. Requires careful monitoring and alerting to detect and respond to accuracy degradation.

Recovery and Reintegration Process

When failed Redis nodes recover and rejoin the cluster, they must be reintegrated carefully to avoid disrupting ongoing operations or creating inconsistent state. The recovery process coordinates between the recovered node, currently active nodes, and all application instances to restore normal hash ring operation.

Health validation verifies that recovered nodes are genuinely stable before accepting production traffic. Recovered nodes undergo extended health checking (typically 5-10 minutes of successful operations) before being marked as fully available. During this validation period, they may receive limited test traffic but are not included in production hash ring calculations.

State synchronization attempts to restore accurate rate limit state for keys that will return to the recovered node's responsibility. The system identifies key ranges that belonged to the recovered node before its failure and attempts to retrieve current counter values from the nodes that handled them during the failure. This synchronization is best-effort and may not be possible for all keys.

Gradual traffic migration slowly shifts rate limit operations back to recovered nodes rather than immediately resuming full traffic. This gradual approach prevents overwhelming recently recovered nodes and provides opportunity to detect any residual instability before full reintegration.

Consistency reconciliation resolves any conflicts between rate limit state on recovered nodes and current state managed by other nodes during the failure. In most cases, the current state from active nodes takes precedence, but the system may need to perform merging operations for keys that continued to receive updates on both the recovered node and its failover replacement.

Common Pitfalls

⚠️ **Pitfall: Uneven Virtual Node Distribution** Many implementations distribute virtual nodes unevenly across the hash ring, creating hot spots even with consistent hashing. This happens when virtual node generation uses poor hash functions or insufficient randomization. The symptom is persistent load imbalance despite having many virtual nodes. Fix this by using cryptographic hash functions (SHA-256) with proper salt values for virtual node generation, and validate distribution evenness during testing.

⚠️ **Pitfall: Hot Key Detection False Positives** Aggressive hot key detection can trigger unnecessary replication during normal traffic spikes, wasting resources and adding complexity. This occurs when detection thresholds are too low or time windows are too short. The symptom is frequent hot key alerts during peak hours for keys that don't actually create bottlenecks. Fix this by implementing adaptive thresholds based on overall cluster load and requiring sustained hot key patterns before triggering replication.

⚠️ **Pitfall: Inconsistent Hash Ring State During Failures** Application instances can maintain different views of hash ring topology during network partitions or rapid node failures, leading to rate limit checks being sent to wrong nodes. This manifests as sudden spikes in Redis errors or rate limit inaccuracy. Implement hash ring versioning and periodic synchronization to detect and resolve inconsistent states, with fallback to local rate limiting when uncertainty is detected.

⚠️ **Pitfall: Circuit Breaker Oscillation** Poorly configured circuit breakers can oscillate rapidly between open and closed states when Redis nodes are intermittently failing, creating unstable performance. This happens when failure thresholds are too sensitive or recovery validation is insufficient. Symptoms include frequent circuit state changes and inconsistent response times. Fix this by implementing exponential backoff for circuit state changes and requiring sustained health before closing circuits.

⚠️ **Pitfall: State Loss During Node Recovery** When failed nodes recover, their stale rate limit state can overwrite more recent state from failover nodes, causing incorrect rate limit calculations. This occurs when recovery processes don't properly synchronize state or determine key ownership. Implement proper state merging logic that prioritizes more recent timestamps and validates key ownership before accepting recovered state.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Hash Function	`crypto/sha256` with hex encoding	`github.com/spaolacci/murmur3` for performance
Virtual Node Management	In-memory slice with binary search	`github.com/serialx/hashring` library
Health Checking	Simple ping with `go-redis`	`github.com/sony/gobreaker` circuit breaker
Node Discovery	Static configuration file	`github.com/hashicorp/consul` for service discovery
Metrics Collection	Basic counters in memory	`github.com/prometheus/client_golang`

Recommended File Structure

```
internal/
  sharding/
    ring.go                    ← consistent hash ring implementation
    ring_test.go               ← ring distribution and rebalancing tests
    virtual_nodes.go           ← virtual node management
    hot_keys.go                ← hot key detection and replication
    health_checker.go          ← Redis node health monitoring
    circuit_breaker.go         ← circuit breaker pattern implementation
  redis/
    cluster_client.go          ← Redis cluster client wrapper
    failover.go                ← automatic failover coordination
    migration.go               ← key migration during rebalancing
  config/
    sharding_config.go         ← sharding configuration structures
```

Infrastructure Starter Code

Consistent Hash Ring Infrastructure (complete implementation):

```go
package sharding

import (
    "crypto/sha256"
    "fmt"
    "sort"
    "sync"
)

// HashRing represents a consistent hash ring with virtual nodes
type HashRing struct {
    mu           sync.RWMutex
    ring         []uint32
    nodes        map[uint32]string
    nodeWeights  map[string]int
    virtualNodes int
}

// NewHashRing creates a new consistent hash ring
func NewHashRing(virtualNodes int) *HashRing {
    return &HashRing{
        ring:         make([]uint32, 0),
        nodes:        make(map[uint32]string),
        nodeWeights:  make(map[string]int),
        virtualNodes: virtualNodes,
    }
}

// hashKey generates a hash for a key or virtual node identifier
func (hr *HashRing) hashKey(key string) uint32 {
    h := sha256.Sum256([]byte(key))
    return uint32(h[0])<<24 | uint32(h[1])<<16 | uint32(h[2])<<8 | uint32(h[3])
}

// AddNode adds a physical node to the ring with virtual nodes
func (hr *HashRing) AddNode(nodeID string, weight int) {
    hr.mu.Lock()
    defer hr.mu.Unlock()


    // Remove existing virtual nodes for this physical node
```

```
    hr.removeNodeUnsafe(nodeID)

    // Add virtual nodes for this physical node
    for i := 0; i < hr.virtualNodes*weight; i++ {
        virtualKey := fmt.Sprintf("%s:vnode:%d", nodeID, i)
        hash := hr.hashKey(virtualKey)
        hr.ring = append(hr.ring, hash)
        hr.nodes[hash] = nodeID
    }

    hr.nodeWeights[nodeID] = weight
    sort.Slice(hr.ring, func(i, j int) bool {
        return hr.ring[i] < hr.ring[j]
    })
}

// RemoveNode removes a physical node and all its virtual nodes
func (hr *HashRing) RemoveNode(nodeID string) {
    hr.mu.Lock()
    defer hr.mu.Unlock()
    hr.removeNodeUnsafe(nodeID)
}

// removeNodeUnsafe removes a node without locking (internal helper)
func (hr *HashRing) removeNodeUnsafe(nodeID string) {
    newRing := make([]uint32, 0, len(hr.ring))
    for _, hash := range hr.ring {
        if hr.nodes[hash] != nodeID {
            newRing = append(newRing, hash)
        } else {
            delete(hr.nodes, hash)
        }
    }
    hr.ring = newRing
    delete(hr.nodeWeights, nodeID)
}

// GetNode returns the responsible node for a given key
func (hr *HashRing) GetNode(key string) (string, bool) {
```

```
    hr.mu.RLock()
    defer hr.mu.RUnlock()

    if len(hr.ring) == 0 {
        return "", false
    }

    hash := hr.hashKey(key)
    idx := sort.Search(len(hr.ring), func(i int) bool {
        return hr.ring[i] >= hash
    })

    if idx == len(hr.ring) {
        idx = 0  // wrap around to beginning
    }

    return hr.nodes[hr.ring[idx]], true
}

// GetNodes returns N responsible nodes for a key (for replication)
func (hr *HashRing) GetNodes(key string, count int) []string {
    hr.mu.RLock()
    defer hr.mu.RUnlock()

    if len(hr.ring) == 0 || count <= 0 {
        return nil
    }

    hash := hr.hashKey(key)
    idx := sort.Search(len(hr.ring), func(i int) bool {
        return hr.ring[i] >= hash
    })

    if idx == len(hr.ring) {
        idx = 0
    }

    result := make([]string, 0, count)
```

```
    seen := make(map[string]bool)


    for len(result) < count && len(seen) < len(hr.nodeWeights) {
        nodeID := hr.nodes[hr.ring[idx]]
        if !seen[nodeID] {
            result = append(result, nodeID)
            seen[nodeID] = true
        }
        idx = (idx + 1) % len(hr.ring)
    }


    return result
}
```

Circuit Breaker Implementation (complete implementation):

```go
package sharding

import (
    "context"
    "sync"
    "time"
)

type CircuitState int

const (
    CircuitClosed CircuitState = iota
    CircuitOpen
    CircuitHalfOpen
)

// CircuitBreaker implements the circuit breaker pattern for Redis nodes
type CircuitBreaker struct {
    mu               sync.RWMutex
    state            CircuitState
    failureCount     int
    lastFailureTime  time.Time
    nextRetryTime    time.Time

    // Configuration
    failureThreshold int
    recoveryTimeout  time.Duration
    halfOpenMaxCalls int
    halfOpenCalls    int
    halfOpenSuccesses int
}

// NewCircuitBreaker creates a new circuit breaker
func NewCircuitBreaker(failureThreshold int, recoveryTimeout time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
        state:            CircuitClosed,
        failureThreshold: failureThreshold,
        recoveryTimeout:  recoveryTimeout,
        halfOpenMaxCalls: 3,
    }
```

```
}

// Execute runs a function with circuit breaker protection
func (cb *CircuitBreaker) Execute(ctx context.Context, fn func() error) error {
    if !cb.allowRequest() {
        return fmt.Errorf("circuit breaker is open")
    }

    err := fn()
    cb.recordResult(err == nil)
    return err
}

// allowRequest determines if a request should be allowed
func (cb *CircuitBreaker) allowRequest() bool {
    cb.mu.Lock()
    defer cb.mu.Unlock()

    now := time.Now()

    switch cb.state {
    case CircuitClosed:
        return true
    case CircuitOpen:
        if now.After(cb.nextRetryTime) {
            cb.state = CircuitHalfOpen
            cb.halfOpenCalls = 0
            cb.halfOpenSuccesses = 0
            return true
        }
        return false
    case CircuitHalfOpen:
        return cb.halfOpenCalls < cb.halfOpenMaxCalls
    default:
        return false
    }
}

// recordResult records the result of a request
```

```
func (cb *CircuitBreaker) recordResult(success bool) {
    cb.mu.Lock()
    defer cb.mu.Unlock()

    switch cb.state {
    case CircuitClosed:
        if success {
            cb.failureCount = 0
        } else {
            cb.failureCount++
            cb.lastFailureTime = time.Now()
            if cb.failureCount >= cb.failureThreshold {
                cb.state = CircuitOpen
                cb.nextRetryTime = time.Now().Add(cb.recoveryTimeout)
            }
        }
    case CircuitHalfOpen:
        cb.halfOpenCalls++
        if success {
            cb.halfOpenSuccesses++
        }

        if cb.halfOpenCalls >= cb.halfOpenMaxCalls {
            if cb.halfOpenSuccesses == cb.halfOpenMaxCalls {
                cb.state = CircuitClosed
                cb.failureCount = 0
            } else {
                cb.state = CircuitOpen
                cb.nextRetryTime = time.Now().Add(cb.recoveryTimeout)
            }
        }
    }
}

// GetState returns current circuit state (for monitoring)
func (cb *CircuitBreaker) GetState() CircuitState {
    cb.mu.RLock()
    defer cb.mu.RUnlock()
```

```
    return cb.state
}
```

Core Logic Skeleton Code

Hot Key Detection Implementation:

GO

```go
// HotKeyDetector monitors request patterns and identifies hot keys
type HotKeyDetector struct {
    mu              sync.RWMutex
    keyStats        map[string]*KeyStats
    clusterStats    *ClusterStats
    config          HotKeyConfig
    replicationChan chan string
}

// KeyStats tracks request frequency for individual keys
type KeyStats struct {
    RequestCount1Min  int64
    RequestCount5Min  int64
    RequestCount15Min int64
    LastUpdate        time.Time
    WindowBuckets     []int64  // Sliding window buckets
}

// DetectHotKeys analyzes current key statistics and identifies hot keys
func (hkd *HotKeyDetector) DetectHotKeys() []string {
    // TODO 1: Get current cluster-wide request statistics to establish baseline
    // TODO 2: Calculate dynamic threshold based on total cluster requests per minute
    // TODO 3: Iterate through all tracked keys and check their request rates
    // TODO 4: Compare each key's rate against both absolute and relative thresholds
    // TODO 5: For keys exceeding thresholds, verify they exceed minimum duration requirement
    // TODO 6: Return list of confirmed hot keys for replication
    // Hint: Hot key threshold = max(absolute_minimum, cluster_rps * hot_key_percentage)
    return nil
}

// RecordKeyAccess increments access counters for a specific rate limit key
func (hkd *HotKeyDetector) RecordKeyAccess(key string) {
    // TODO 1: Get or create KeyStats struct for this key in thread-safe manner
    // TODO 2: Update current minute bucket in sliding window
    // TODO 3: Increment counters for 1min, 5min, and 15min windows
    // TODO 4: Update LastUpdate timestamp
    // TODO 5: If key stats indicate potential hot key, trigger async detection check
    // Hint: Use atomic operations for counter updates to avoid lock contention
}
```

```go
// ReplicateHotKey creates replicas of hot key across multiple nodes
func (hkd *HotKeyDetector) ReplicateHotKey(key string, replicationFactor int) error {
    // TODO 1: Determine current responsible node for this key using hash ring
    // TODO 2: Select additional nodes for replication using GetNodes(key, replicationFactor)
    // TODO 3: Read current rate limit state from primary node
    // TODO 4: Create replica entries on selected nodes with read-only markers
    // TODO 5: Update local routing table to include replica locations for read queries
    // TODO 6: Set up periodic synchronization from primary to replicas
    // Hint: Replicas should handle reads only; all writes go to primary node
    return nil
}
```

Node Health Checker Implementation:

```go
// HealthChecker monitors Redis node health and manages circuit breakers
type HealthChecker struct {
    mu              sync.RWMutex
    nodes           map[string]*NodeHealth
    circuitBreakers map[string]*CircuitBreaker
    config          HealthConfig
}

// NodeHealth tracks health status for individual Redis nodes
type NodeHealth struct {
    NodeID           string
    Address          string
    LastSeen         time.Time
    ConsecutiveFailures int
    AverageLatency   time.Duration
    IsHealthy        bool
    MemoryUsage      float64
    ConnectionCount  int
}

// CheckNodeHealth performs comprehensive health check on a Redis node
func (hc *HealthChecker) CheckNodeHealth(ctx context.Context, nodeID string) (*NodeHealth, error) {
    // TODO 1: Perform basic connectivity ping to Redis node
    // TODO 2: Execute lightweight rate limit operation with synthetic key
    // TODO 3: Measure response latency and compare against thresholds
    // TODO 4: Query Redis INFO command for memory usage and connection stats
    // TODO 5: Update NodeHealth struct with current status and timestamps
    // TODO 6: Determine overall health status based on all checks
    // TODO 7: Update circuit breaker state based on health check results
    // Hint: Use context.WithTimeout to prevent health checks from hanging
    return nil, nil
}

// StartHealthMonitoring begins periodic health checking for all nodes
func (hc *HealthChecker) StartHealthMonitoring(ctx context.Context) {
    // TODO 1: Create ticker for periodic health checks (every 30 seconds)
    // TODO 2: For each registered node, perform health check in separate goroutine
    // TODO 3: Collect health check results and update node status
    // TODO 4: Trigger failover notifications for newly failed nodes
```

```
    // TODO 5: Update hash ring to exclude failed nodes from routing
    // TODO 6: Handle context cancellation for graceful shutdown
    // Hint: Use errgroup to manage multiple concurrent health checks
}

// HandleNodeFailure coordinates response to detected node failure
func (hc *HealthChecker) HandleNodeFailure(nodeID string) error {
    // TODO 1: Mark node as failed in health tracking structures
    // TODO 2: Open circuit breaker for this node to prevent further requests
    // TODO 3: Remove node from consistent hash ring to stop routing new requests
    // TODO 4: Identify keys that were handled by failed node
    // TODO 5: Notify key migration system to redirect affected keys
    // TODO 6: Send alerts/notifications about node failure to monitoring systems
    // Hint: Failure handling should be idempotent in case of multiple detection events
    return nil
}
```

Milestone Verification Checkpoints

After implementing consistent hash ring:

- Run `go test ./internal/sharding/ring_test.go -v` to verify hash distribution
- Test with 4 nodes and 100,000 keys - each node should handle 20-30% of keys
- Add a 5th node - verify only ~20% of keys move to the new node
- Remove a node - verify keys redistribute to adjacent nodes only

After implementing hot key detection:

- Create test scenario with 1000 RPS to key "user:popular" and 10 RPS to other keys
- Monitor detection system - should identify "user:popular" as hot within 2 minutes
- Verify hot key replication creates read replicas on 2-3 additional nodes
- Test that read queries distribute across replicas while writes go to primary

After implementing health checking and failover:

- Start 3 Redis nodes and verify all show as healthy
- Stop one Redis node - should detect failure within 30-60 seconds
- Verify circuit breaker opens and traffic routes to remaining healthy nodes
- Restart failed node - should detect recovery and resume routing within 2-3 minutes

Signs something is wrong:

- Hash ring distribution variance >20% indicates poor virtual node generation
- Hot key detection triggering for normal keys indicates thresholds too low
- Health checks showing false failures indicates network timeouts too aggressive
- Keys routing to wrong nodes during failures indicates hash ring inconsistency

Interactions and Data Flow

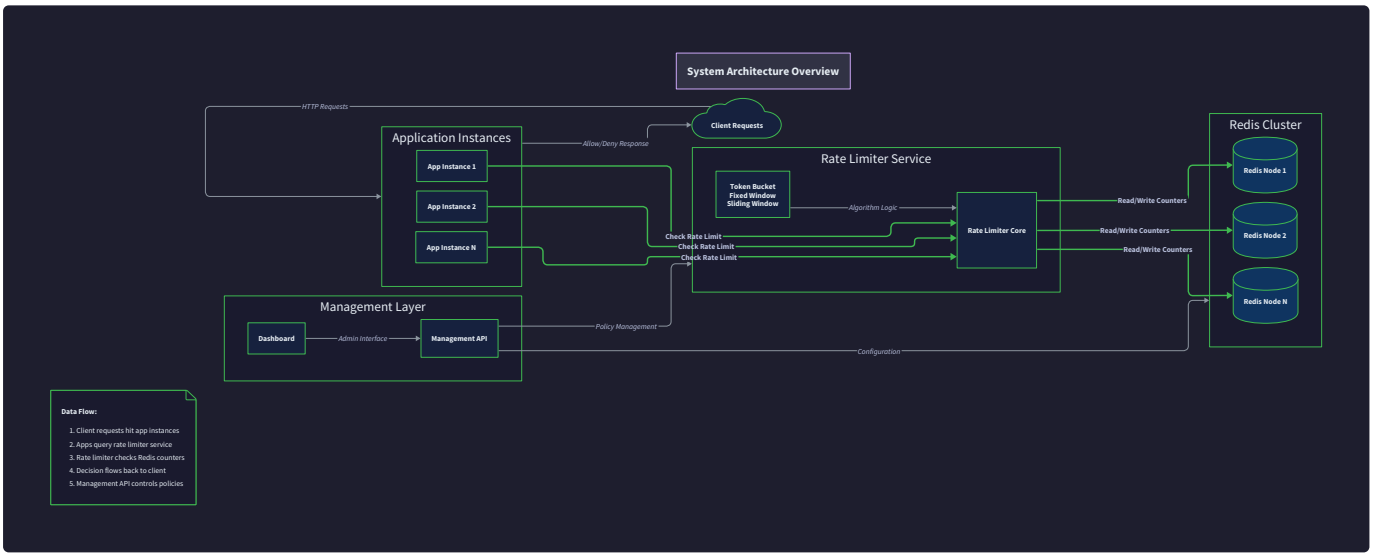
> **Milestone(s):** Milestone 1, 2, 3, 4, 5 - this section demonstrates how all components work together across rate limiting algorithms, multi-tier evaluation, Redis integration, sharding, and API management

Understanding the interactions between components in a distributed rate limiting system is crucial for building a robust and performant implementation. This section explores the three primary data flows that define how the system operates: rate limit checking, configuration management, and metrics collection.

Mental Model: The Air Traffic Control System

Think of the distributed rate limiter as an air traffic control system managing aircraft (requests) across multiple airports (application instances) with a central coordination center (Redis cluster). When an aircraft requests landing clearance, the local tower (application instance) must coordinate with the central system to check runway capacity (rate limits), update the flight schedule (counters), and ensure safe separation (prevent overload). Configuration updates flow like weather advisories - broadcast from central meteorology (management API) to all towers simultaneously. Meanwhile, flight statistics (metrics) continuously stream back to central command for monitoring and decision-making.

This analogy helps illustrate the critical coordination patterns: real-time decision making with shared state, configuration propagation across distributed nodes, and continuous monitoring of system health. The key insight is that each local decision requires global coordination, but the system must remain fast enough for real-time operation.



Rate Limit Check Flow

The rate limit check flow represents the most performance-critical path in the distributed rate limiting system. Every incoming request must be evaluated against potentially multiple rate limit rules, with the decision made in milliseconds while maintaining consistency across all application instances.

Request Context Assembly

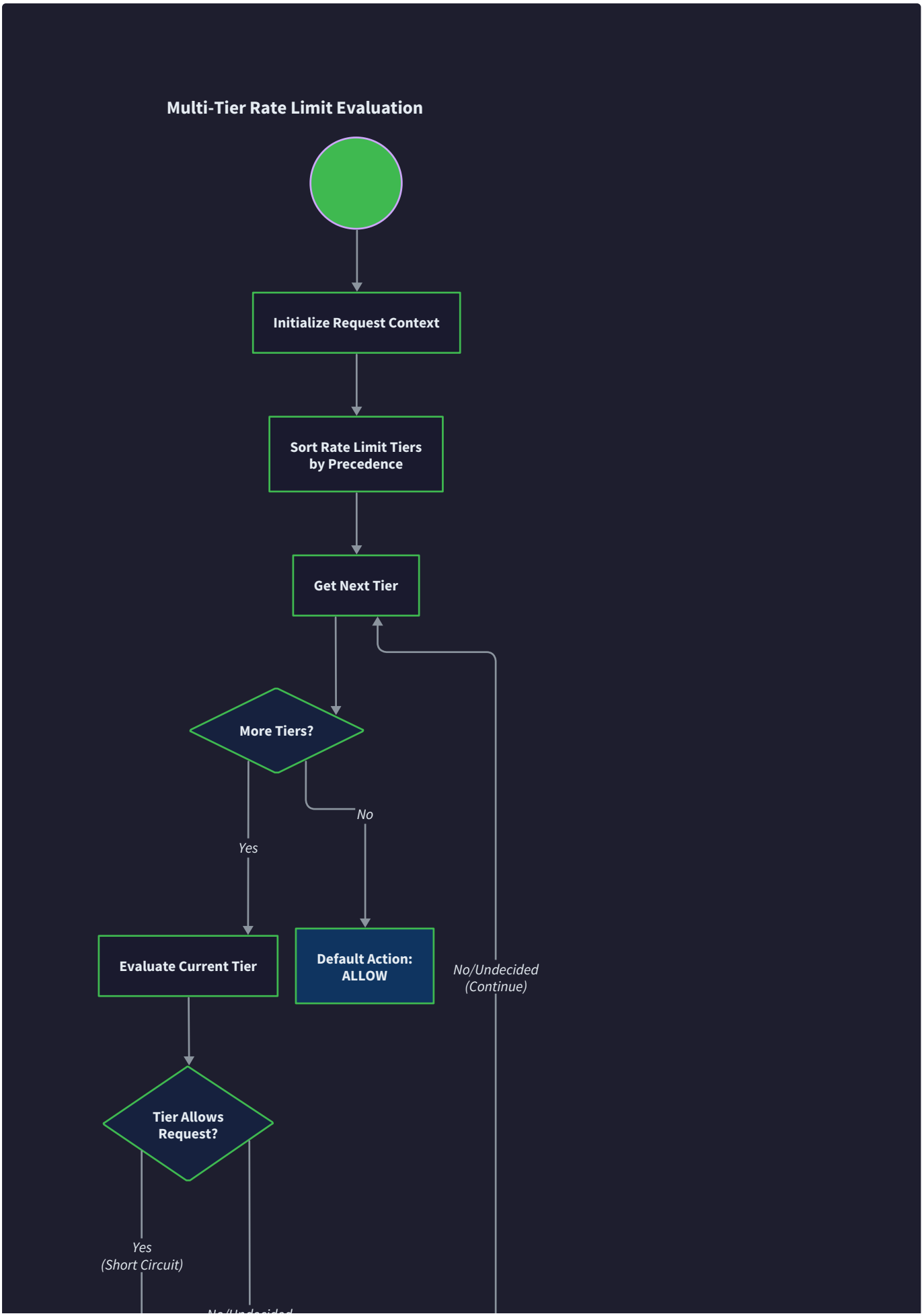
The rate limit check begins when an HTTP request arrives at any application instance. The `DistributedLimiter` component extracts relevant context information to construct a `RateLimitRequest` structure. This context assembly process is crucial because it determines which rate limit rules will be evaluated and how the request will be categorized.

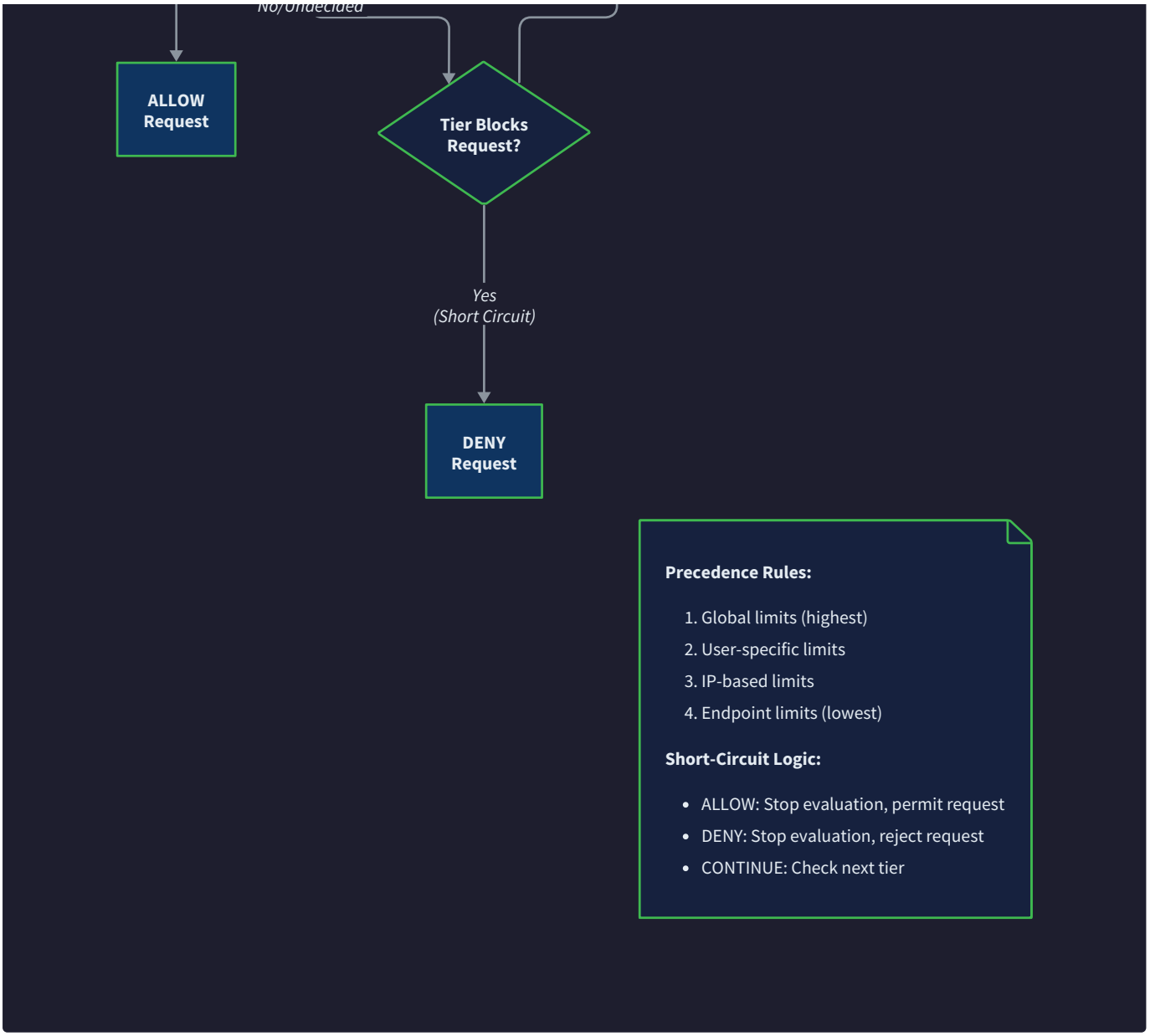
Context Field	Source	Purpose	Example Value
`user_id`	Authentication header/JWT	Per-user rate limiting	"user_12345"
`ip_address`	Request remote address	Per-IP rate limiting	"192.168.1.100"
`api_endpoint`	Request path pattern	Per-API rate limiting	"/api/v1/users"
`user_agent`	HTTP header	Bot detection/classification	"Mozilla/5.0..."
`tokens`	Request payload size/type	Resource-aware limiting	1 or payload_size

The context assembly must handle edge cases like missing authentication (anonymous users), proxy forwarding (X-Forwarded-For headers), and API path normalization (removing parameters). The `KeyComposer` component uses this context to generate Redis keys that will be used for rate limit state storage.

Multi-Tier Rule Evaluation

Once the request context is assembled, the `MultiTierLimiter` begins evaluating applicable rate limit rules. The system follows a short-circuit evaluation strategy, checking rules in priority order and stopping immediately when any limit is exceeded.


Multi-Tier Rate Limit Evaluation
Initialize Request Context
Sort Rate Limit Tiers by Precedence
Get Next Tier
More Tiers?
No
Yes
Evaluate Current Tier
Default Action: ALLOW
No/Undecided (Continue)
Tier Allows Request?
Yes (Short Circuit)



The tier evaluation follows this algorithmic sequence:

1. **Rule Discovery**: The `RuleManager` scans all configured rules and identifies those whose `key_pattern` matches the request context. Rules are sorted by priority (highest first) to ensure most specific limits are checked before general ones.
2. **Key Composition**: For each matching rule, the `KeyComposer` generates a Redis key using the rule's pattern and request context. This involves pattern substitution and namespace prefixing to ensure key uniqueness across different rate limit dimensions.
3. **Tier Evaluation Loop**: The system iterates through matching rules, performing rate limit checks for each tier:
    - User-specific limits (highest priority)
    - IP-address limits (medium priority)
    - API endpoint limits (medium priority)
    - Global system limits (lowest priority)
4. **Short-Circuit Logic**: If any rule evaluation returns `allowed: false`, the entire check immediately fails without evaluating remaining rules. This optimization reduces Redis operations and provides faster response times for requests that exceed limits.
5. **Result Aggregation**: If all rules pass, the system returns the most restrictive result (lowest remaining count) to provide accurate rate limit headers in the response.

Evaluation Step	Action	Success Path	Failure Path
Rule Discovery	Find matching rules by pattern	Continue to next step	Return allowed=true (no rules)
Key Composition	Generate Redis keys	Continue to tier evaluation	Return error
Per-User Check	Check user-specific limits	Continue to next tier	Return allowed=false
Per-IP Check	Check IP-based limits	Continue to next tier	Return allowed=false
Per-API Check	Check endpoint limits	Continue to next tier	Return allowed=false
Global Check	Check system-wide limits	Return allowed=true	Return allowed=false

Redis Atomic Operations

The core of each individual rate limit check is an atomic Redis operation implemented as a Lua script. This atomicity is essential for maintaining accurate counters in a distributed environment where multiple application instances may be checking the same rate limit simultaneously.

Each algorithm uses a specialized Lua script optimized for its specific requirements:

Token Bucket Script Flow:

1. Load current bucket state (`tokens` , `last_refill_time`) from Redis
2. Calculate elapsed time since last refill and compute tokens to add
3. Refill bucket up to capacity based on configured refill rate
4. Check if requested tokens are available
5. If available, subtract tokens and update state; if not, return current state
6. Return result with remaining tokens and reset time

Sliding Window Counter Script Flow:

1. Calculate current time window and previous window boundaries
2. Load counters for current and previous time windows
3. Calculate weighted count based on time position within current window
4. Check if adding this request would exceed the limit
5. If within limit, increment current window counter
6. Clean up expired window data to prevent memory leaks

7. Return result with remaining capacity

Sliding Window Log Script Flow:

1. Remove expired timestamps from the request log
2. Count remaining timestamps to determine current usage
3. Check if adding this request would exceed the limit
4. If within limit, add current timestamp to the log
5. Return result with current usage and remaining capacity

Script Operation	Atomicity Requirement	Performance Impact	Memory Usage
Token Bucket	State read + refill + check + update	Low (simple arithmetic)	Fixed per key
Sliding Counter	Multi-window read + weighted calc + update	Medium (window math)	Fixed per window
Sliding Log	Log read + filter + count + append	High (list operations)	Grows with traffic

Local Fallback Handling

When Redis becomes unavailable, the system must gracefully degrade to local rate limiting rather than failing open (allowing all requests) or closed (rejecting all requests). This graceful degradation maintains partial functionality while preserving system stability.

The fallback detection and activation follows this sequence:

1. **Failure Detection**: Redis operations fail with connection timeout, network error, or other exceptions. The `CircuitBreaker` component tracks failure rates and automatically opens when thresholds are exceeded.
2. **Fallback Activation**: The `DistributedLimiter` switches to its `localFallback` limiter instance, which maintains in-memory rate limit state using the same algorithms but with local scope.
3. **State Synchronization**: When Redis connectivity is restored, the system must carefully synchronize local and distributed state to prevent double-counting or lost counts.
4. **Recovery Process**: The circuit breaker gradually allows test requests through to verify Redis health before fully reopening.

> **Critical Design Insight**: Local fallback cannot maintain global accuracy, but it preserves system availability and prevents cascading failures. The trade-off between accuracy and availability is an explicit architectural choice.

Response Header Generation

The final step in the rate limit check flow is generating standard HTTP response headers that inform clients about their current rate limit status. These headers follow established conventions and enable clients to implement intelligent backoff strategies.

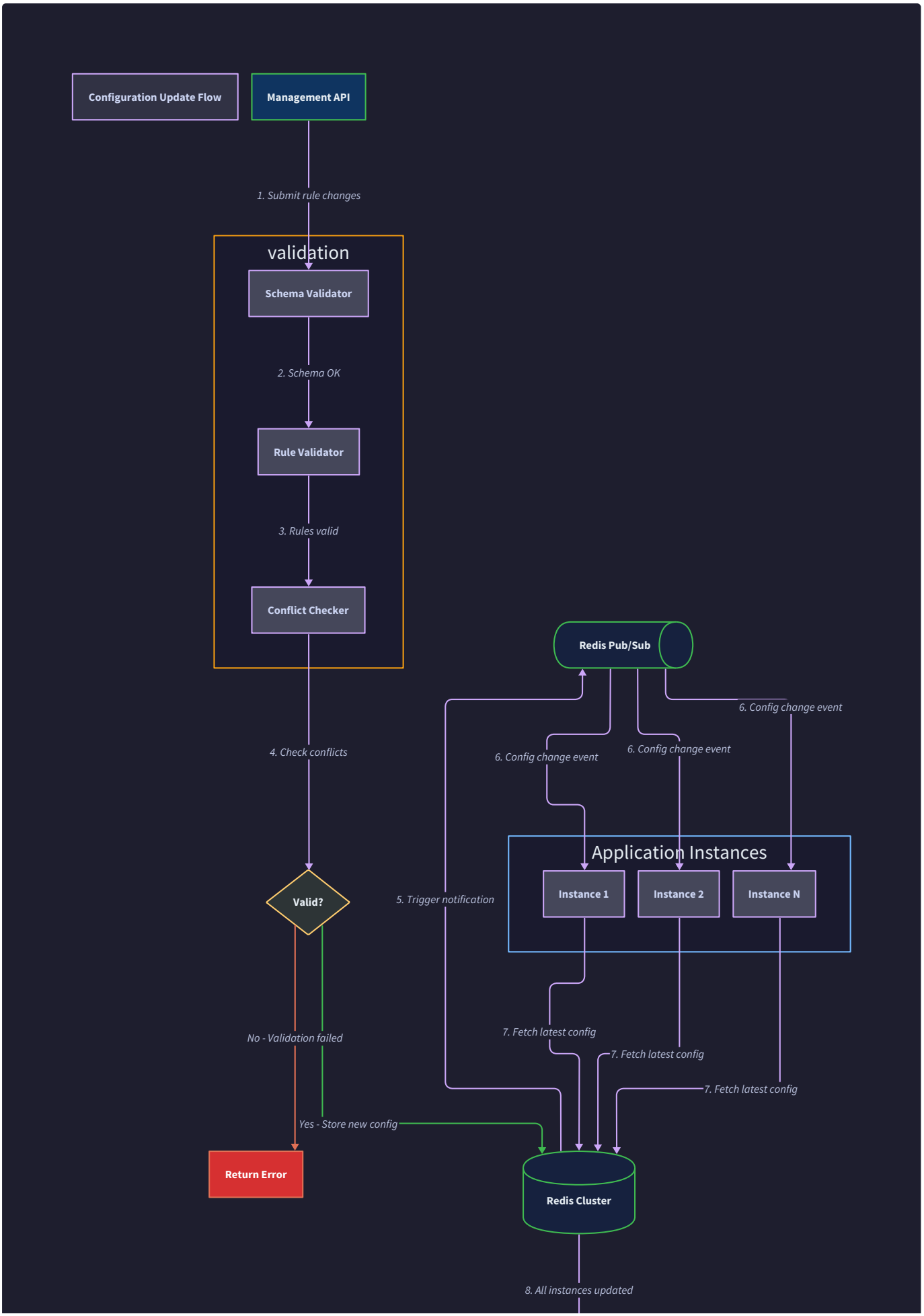
Header Name	Value Format	Purpose	Example
`X-RateLimit-Limit`	Integer	Maximum requests in window	"1000"
`X-RateLimit-Remaining`	Integer	Requests left in current window	"247"
`X-RateLimit-Reset`	Unix timestamp	When window resets	"1699123456"
`Retry-After`	Seconds	How long to wait if limited	"60"

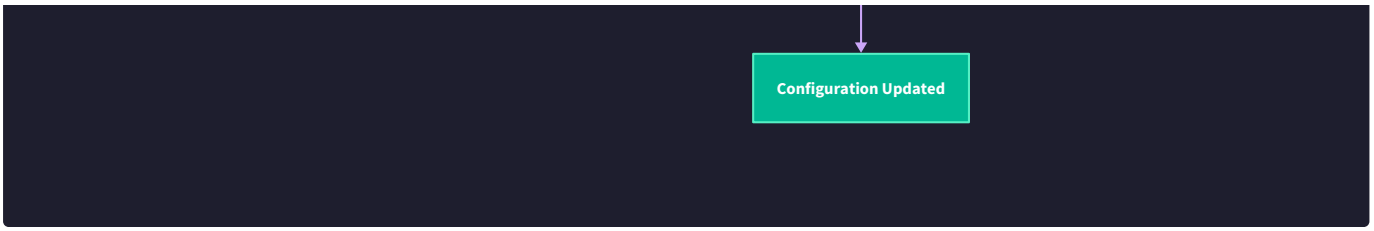
Configuration Update Propagation

Configuration management in a distributed rate limiting system presents unique challenges: changes must be propagated to all application instances quickly and consistently, while the system continues processing requests without interruption.

Mental Model: Emergency Broadcast System

Think of configuration updates like an emergency broadcast system. When new rate limit rules are created or modified (emergency announcement), they must reach every application instance (radio station) reliably and quickly. Each instance must validate the message authenticity (configuration schema), update its local understanding (rule cache), and begin operating under the new rules immediately. Failed deliveries are retried, and the system tracks which instances have received updates to ensure full coverage.


Configuration Update Flow
Management API
1. Submit rule changes
validation
Schema Validator
2. Schema OK
Rule Validator
3. Rules valid
Conflict Checker
4. Check conflicts
Valid?
No - Validation failed
Return Error
Yes - Store new config
Redis Cluster
8. All instances updated
Redis Pub/Sub
5. Trigger notification
6. Config change event
6. Config change event
6. Config change event
Application Instances
Instance 1
Instance 2
Instance N
7. Fetch latest config
7. Fetch latest config
7. Fetch latest config



Configuration Source and Validation

The configuration update process begins at the management API, which serves as the authoritative source for all rate limit rules. When administrators create, modify, or delete rate limit rules, the API performs comprehensive validation before persisting changes.

Validation Check	Purpose	Failure Action	Example
Schema Validation	Ensure required fields present	Reject with 400 error	Missing `limit` field
Pattern Syntax	Validate regex patterns	Reject with 400 error	Invalid regex: `[unclosed`
Limit Bounds	Check reasonable limit values	Reject with 400 error	Negative rate limit
Priority Conflicts	Prevent overlapping priorities	Reject with 409 error	Two rules same priority
Algorithm Support	Verify algorithm exists	Reject with 400 error	Unknown algorithm name

The validation process uses a staged approach where syntactic validation occurs first (fast checks), followed by semantic validation (cross-rule consistency), and finally persistence validation (storage constraints). This ordering minimizes wasted work when validation fails.

Redis as Configuration Distribution Hub

The system uses Redis not only for rate limit state storage but also as a configuration distribution mechanism. This design choice leverages Redis's existing high availability and clustering capabilities while providing real-time updates to all application instances.

The configuration distribution follows a publish-subscribe pattern combined with persistent storage:

1. **Persistent Storage**: Rate limit rules are stored in Redis using hash structures, allowing atomic updates and version tracking.
2. **Change Notification**: When rules are updated, the management API publishes a notification to a Redis channel announcing the change.
3. **Instance Subscription**: All application instances subscribe to the configuration change channel and receive real-time notifications.
4. **Incremental Updates**: Instances fetch only changed rules rather than reloading entire configurations, reducing network overhead and update latency.

Redis Structure	Key Pattern	Purpose	Data Format
Rule Storage	`config:rules:{rule_id}`	Persistent rule data	Hash with all rule fields
Version Tracking	`config:version`	Change detection	Integer timestamp
Change Log	`config:changelog:{version}`	Audit trail	JSON change description
Heartbeat	`config:heartbeat:{instance_id}`	Instance health	Timestamp + rule version

Application Instance Update Process

Each application instance runs a configuration watcher component that maintains consistency between the central configuration store and local rule caches. This component must handle network failures, partial updates, and version conflicts gracefully.

The update process for each application instance follows this sequence:

1. **Change Detection**: The instance receives a notification via Redis pub/sub or detects a version mismatch during periodic health checks.
2. **Version Comparison**: The instance compares its local rule version against the central version to determine what updates are needed.
3. **Incremental Fetch**: Only changed rules are fetched from Redis, using timestamps or version numbers to identify modifications.
4. **Atomic Local Update**: The instance updates its in-memory rule cache atomically to prevent inconsistent state during rule evaluation.
5. **Validation and Rollback**: After updating, the instance validates that all rules are correctly loaded and rolls back to the previous version if problems are detected.
6. **Heartbeat Update**: The instance updates its heartbeat record in Redis to confirm successful configuration application.

Decision: Incremental vs Full Configuration Reload

- **Context**: Configuration changes need to propagate to potentially hundreds of application instances
- **Options Considered**:
  1. Full configuration reload on every change
  2. Incremental updates with change tracking
  3. Configuration push from central server
- **Decision**: Incremental updates with Redis pub/sub notification
- **Rationale**: Minimizes network bandwidth, reduces update latency, and scales better with cluster size. Redis pub/sub provides reliable delivery with automatic retries.
- **Consequences**: More complex change tracking logic, but significantly better performance and scalability.

Configuration Validation and Conflict Resolution

Distributed configuration management introduces the possibility of conflicts and inconsistencies that must be detected and resolved automatically. The system implements several mechanisms to ensure configuration consistency across all instances.

Conflict Type	Detection Method	Resolution Strategy	Prevention
Version Skew	Periodic heartbeat comparison	Force full reload	Timeout limits on updates
Rule Priority Overlap	Cross-rule validation	Reject conflicting update	Priority uniqueness check
Pattern Ambiguity	Pattern matching test	Admin notification	Test pattern coverage
Resource Exhaustion	Memory/performance monitoring	Gradual rollout	Capacity planning

The conflict resolution system operates on the principle of "fail safely" - when conflicts cannot be automatically resolved, the system maintains the last known good configuration while alerting administrators to the issue.

Configuration Rollback and Recovery

The distributed nature of the system requires sophisticated rollback capabilities when configuration changes cause problems. The system maintains configuration history and provides mechanisms for rapid rollback across all instances.

The rollback process works as follows:

1. **Issue Detection**: Monitoring systems detect increased error rates, performance degradation, or other problems following a configuration change.
2. **Rollback Trigger**: Administrators or automated systems trigger a rollback to a previous configuration version.
3. **Coordinated Rollback**: The management API publishes a rollback notification containing the target version number.
4. **Instance Rollback**: Each application instance loads the specified historical configuration from its local cache or Redis backup.
5. **Verification**: Instances report successful rollback via heartbeat updates, allowing centralized verification of rollback completion.

Metrics Collection and Aggregation

Comprehensive metrics collection is essential for operating a distributed rate limiting system effectively. The metrics system must capture detailed usage patterns, performance characteristics, and system health indicators while minimizing impact on the critical rate limiting path.

Mental Model: Hospital Vital Signs Monitoring

Think of the metrics collection system like vital signs monitoring in a hospital. Each patient (rate limit key) has continuous monitoring of key indicators (request rates, rejection rates, latency). Nurses (application instances) collect readings regularly and report to a central monitoring station (metrics aggregator). Doctors (operators) use dashboards to spot trends, diagnose problems, and make treatment decisions. Critical alerts (threshold breaches) trigger immediate notifications, while long-term trends inform capacity planning and system optimization.

Real-Time Metrics Collection

The metrics collection system operates continuously alongside the rate limiting functionality, capturing both operational metrics (performance, errors) and business metrics (usage patterns, limit effectiveness). The collection must be designed for minimal performance impact while providing sufficient detail for troubleshooting and optimization.

Metric Category	Specific Metrics	Collection Method	Granularity
Request Metrics	`requests_total`, `requests_allowed`, `requests_denied`	Counter increment on each check	Per rule, per instance
Performance Metrics	`check_duration_ms`, `redis_latency_ms`	Histogram recording	Per operation type
System Health	`redis_connection_status`, `fallback_active`	Gauge sampling	Per instance
Usage Patterns	`top_keys_by_volume`, `limit_utilization_pct`	Periodic aggregation	Per time window

Each application instance maintains local metric collectors that aggregate data over short time windows (typically 10-60 seconds) before transmitting to the central metrics system. This approach reduces network overhead while providing near-real-time visibility into system behavior.

Hot Key Detection and Analysis

Hot key detection represents one of the most valuable capabilities of the metrics system. By identifying rate limit keys that receive disproportionate traffic, the system can automatically trigger replication, sharding adjustments, or capacity alerts.

The `HotKeyDetector` component continuously analyzes request patterns using a multi-window approach:

1. **Request Counting**: Each application instance maintains counters for every rate limit key it processes, tracking request volumes over multiple time windows (1 minute, 5 minutes, 15 minutes).
2. **Statistical Analysis**: The system calculates request rate percentiles across all keys to identify outliers that exceed configurable thresholds (e.g., 95th percentile).
3. **Trend Detection**: Hot key detection considers not just absolute volumes but also growth rates to identify keys experiencing traffic spikes.
4. **Cluster-Wide Aggregation**: Individual instance measurements are aggregated across the cluster to provide accurate global hot key identification.

Detection Window	Purpose	Threshold Type	Action Triggered
1 Minute	Spike detection	Absolute count > P99	Immediate replication
5 Minutes	Sustained load	Rate increase > 300%	Load balancing adjustment
15 Minutes	Trend analysis	Steady growth > 50%	Capacity planning alert
1 Hour	Baseline establishment	Rolling average update	Threshold recalibration

Dashboard and Visualization

The real-time dashboard provides operators with comprehensive visibility into the rate limiting system's behavior. The dashboard must balance information density with usability, presenting critical information prominently while making detailed data easily accessible.

The dashboard architecture uses WebSocket connections for real-time updates, with efficient data streaming that minimizes bandwidth usage. Key visualizations include:

System Overview Panel:

- Total request rate across all instances
- Overall allow/deny ratio
- Active rate limit rules count
- Redis cluster health status

Top Keys Analysis:

- Highest volume rate limit keys (by request count)
- Most restrictive keys (by denial rate)
- Hot keys detected in recent time windows
- Key growth trends over time

Performance Monitoring:

- Rate limit check latency distribution
- Redis operation latency percentiles
- Circuit breaker status across instances
- Error rate trends by error type

Tier Analysis:

- Effectiveness of different rate limit tiers
- Most frequently triggered rules
- Resource utilization by limit type
- Optimization recommendations

Alerting and Anomaly Detection

The metrics system includes sophisticated alerting capabilities that can detect both immediate problems and developing issues before they impact users. The alerting system operates on multiple time horizons with different sensitivity levels.

Alert Type	Detection Logic	Severity	Response
High Error Rate	Error rate > 5% over 2 minutes	Critical	Immediate page
Redis Unavailable	Connection failures > 50%	Critical	Immediate page
Hot Key Spike	Single key > 10x normal rate	Warning	Auto-mitigation + alert
Capacity Trend	Usage growth > 80% capacity	Info	Capacity planning notification
Config Drift	Instance config version skew	Warning	Auto-remediation attempt

The anomaly detection system uses statistical models to identify unusual patterns that might indicate attacks, misconfigurations, or system problems. These models adapt over time to account for normal traffic pattern evolution while maintaining sensitivity to genuine anomalies.

Self-Monitoring Rate Limits

A critical consideration for the metrics and management API is preventing the rate limiting system from overwhelming itself with monitoring traffic. The system implements self-monitoring rate limits to ensure that dashboard queries, metrics collection, and administrative operations don't interfere with primary rate limiting functionality.

> **Critical Design Insight**: The rate limiting system must protect itself from its own monitoring and management interfaces. This requires careful design of internal rate limits and circuit breakers.

The self-monitoring system includes:

Dashboard Rate Limiting: WebSocket connections are rate limited to prevent a single dashboard user from overwhelming the metrics system with excessive query rates.

API Protection: Management API endpoints have their own rate limits to prevent configuration update storms or abusive administrative behavior.

Metrics Backpressure: When the metrics collection system becomes overwhelmed, it implements backpressure mechanisms to reduce collection frequency rather than dropping data.

Internal Circuit Breakers: Monitoring components include circuit breakers that disable non-essential metrics collection when the primary rate limiting system is under stress.

Common Pitfalls

⚠️ **Pitfall: Race Conditions in Multi-Tier Evaluation** Many implementations fail to properly coordinate between tier evaluations, leading to inconsistent state where one tier's counter is updated while another tier's update fails. This can result in "phantom" request counting where requests are partially counted across different tiers. The fix is to use two-phase operations: first check all tiers without updating, then update all tiers atomically, or implement compensating transactions for partial failures.

⚠️ **Pitfall: Configuration Update Ordering** Configuration updates can arrive at different instances in different orders due to network timing, causing temporary inconsistencies where the same request is evaluated differently by different instances. This is particularly dangerous when rule priorities change. The fix is to use version numbers and ensure all instances process configuration changes in the same order, potentially by using a consensus mechanism or designated configuration coordinator.

⚠️ **Pitfall: Metrics Collection Overwhelming Redis** Naive metrics implementations can generate more Redis traffic than the actual rate limiting operations, especially with detailed per-key statistics. This can overwhelm Redis and degrade rate limiting performance. The solution is to batch metrics operations, use separate Redis instances for metrics vs rate limiting, and implement metrics sampling for high-volume keys.

⚠️ **Pitfall: Synchronous Configuration Updates Blocking Requests** Blocking the request processing thread while updating configuration can cause significant latency spikes during configuration changes. The fix is to use asynchronous configuration updates with atomic cache swapping - load new configuration in the background and atomically replace the active configuration once fully validated.

⚠️ **Pitfall: Hardcoded Timeout Values** Using fixed timeout values for Redis operations doesn't account for varying network conditions or load. This can cause premature fallback activation or excessive blocking during temporary slowdowns. The solution is to implement adaptive timeouts based on recent latency percentiles and provide different timeout configurations for different operations (quick checks vs batch updates).

Implementation Guidance

This implementation section provides complete infrastructure code and skeleton implementations for the core interaction patterns described above.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Client	`net/http` with connection pooling	`fasthttp` for high performance
Configuration Storage	Redis Hash structures	etcd for stronger consistency
Metrics Collection	Built-in counters with periodic export	Prometheus client library
Real-time Updates	Redis Pub/Sub	Apache Kafka for message ordering
Dashboard Backend	WebSocket with JSON	gRPC with streaming

File Structure

```
internal/
├── flow/                          ← This component
│   ├── coordinator.go             ← Main coordination logic
│   ├── config_watcher.go          ← Configuration propagation
│   ├── metrics_collector.go       ← Metrics collection and hot key detection
│   └── health_checker.go          ← System health monitoring
├── storage/                       ← Redis backend
│   ├── redis_storage.go           ← Basic Redis operations
│   └── lua_scripts.go             ← Atomic operation scripts
└── api/                           ← Management interfaces
    ├── handlers.go                ← REST API endpoints
    └── websocket.go               ← Real-time dashboard connection
```

Request Flow Coordinator Infrastructure

This complete infrastructure handles request coordination across multiple rate limiting tiers with proper error handling and fallback logic.

```go
// Package flow provides request coordination and data flow management
// for distributed rate limiting operations across multiple tiers and backends.
package flow

import (
    "context"
    "fmt"
    "sort"
    "sync"
    "time"

    "github.com/rate-limiter/internal/config"
    "github.com/rate-limiter/internal/storage"
)

// FlowCoordinator manages the complete request flow from initial context
// assembly through multi-tier evaluation to response header generation.
type FlowCoordinator struct {
    storage        storage.Storage
    ruleManager    *config.RuleManager
    keyComposer    *config.KeyComposer
    algorithms     map[string]Algorithm
    localFallback  Limiter
    metricsCollector *MetricsCollector
    circuitBreaker *CircuitBreaker
    mu             sync.RWMutex
    isHealthy      bool
}

// RequestContext holds all information needed for rate limit evaluation
// extracted from the incoming HTTP request.
type RequestContext struct {
    UserID      string            `json:"user_id"`
    IPAddress   string            `json:"ip_address"`
    APIEndpoint string            `json:"api_endpoint"`
    UserAgent   string            `json:"user_agent"`
    Headers     map[string]string `json:"headers"`
    Tokens      int64             `json:"tokens"`
    Timestamp   time.Time         `json:"timestamp"`
```

```go
}

// TierEvaluation represents the result of evaluating a single rate limit tier.
type TierEvaluation struct {
    RuleID      string               `json:"rule_id"`
    TierName    string              `json:"tier_name"`
    Result      *storage.RateLimitResult `json:"result"`
    Duration    time.Duration        `json:"duration"`
    Algorithm   string              `json:"algorithm"`
    RedisKey    string              `json:"redis_key"`
}

// FlowResult contains the complete result of request flow processing
// including all tier evaluations and final decision.
type FlowResult struct {
    Allowed       bool               `json:"allowed"`
    TierResults   []*TierEvaluation  `json:"tier_results"`
    BlockingTier  string             `json:"blocking_tier,omitempty"`
    TotalDuration time.Duration      `json:"total_duration"`
    UsedFallback  bool               `json:"used_fallback"`
    Headers       map[string]string  `json:"headers"`
    RetryAfter    time.Duration      `json:"retry_after,omitempty"`
}

// NewFlowCoordinator creates a new coordinator with all required dependencies.
func NewFlowCoordinator(
    storage storage.Storage,
    ruleManager *config.RuleManager,
    algorithms map[string]Algorithm,
    localFallback Limiter,
) *FlowCoordinator {
    return &FlowCoordinator{
        storage:          storage,
        ruleManager:      ruleManager,
        keyComposer:      config.NewKeyComposer(),
        algorithms:       algorithms,
        localFallback:    localFallback,
        metricsCollector: NewMetricsCollector(),
        circuitBreaker:   NewCircuitBreaker(),
```

```
        isHealthy:        true,
    }
}

// ProcessRequest coordinates the complete flow from request context to final result.
func (fc *FlowCoordinator) ProcessRequest(ctx context.Context, reqCtx *RequestContext) (*FlowResult, error) {
    startTime := time.Now()

    // TODO 1: Extract and validate request context (IP, User, API endpoint)
    // TODO 2: Get matching rules from rule manager sorted by priority
    // TODO 3: Check circuit breaker status - use fallback if open
    // TODO 4: Evaluate each tier in priority order with short-circuit logic
    // TODO 5: Generate standard HTTP headers for the response
    // TODO 6: Record metrics for request processing and hot key detection
    // TODO 7: Return comprehensive flow result with all evaluation details

    // Hint: Use fc.evaluateAllTiers() for the core multi-tier logic
    // Hint: Implement short-circuit evaluation - stop on first denial
    // Hint: Always record metrics even for failed/fallback requests

    panic("TODO: Implement ProcessRequest coordination logic")
}

// evaluateAllTiers performs rate limit checking across all applicable tiers.
func (fc *FlowCoordinator) evaluateAllTiers(ctx context.Context, reqCtx *RequestContext, rules []*config.RateLimitRule)
([]*TierEvaluation, error) {
    // TODO 1: Iterate through rules in priority order
    // TODO 2: For each rule, compose the appropriate Redis key
    // TODO 3: Execute the algorithm-specific rate limit check
    // TODO 4: Collect timing and result information for each evaluation
    // TODO 5: Implement short-circuit logic - return immediately on first denial
    // TODO 6: Handle Redis failures gracefully with circuit breaker pattern

    // Hint: Each algorithm (token bucket, sliding window) has different Redis operations
    // Hint: Use goroutines for parallel tier evaluation if all must be checked
    // Hint: Distinguish between hard failures (errors) and soft failures (limits exceeded)

    panic("TODO: Implement multi-tier evaluation with short-circuit logic")
```

```
}
```

Configuration Watcher Infrastructure

Complete implementation of the configuration propagation system with Redis pub/sub and local caching.

```go
// ConfigurationWatcher manages real-time configuration updates using Redis pub/sub
// and maintains local configuration cache with atomic updates.
type ConfigurationWatcher struct {
    redisClient    redis.UniversalClient
    ruleManager    *config.RuleManager
    changeChannel  string
    subscription   *redis.PubSub
    localVersion   int64
    updateCallback func([]*config.RateLimitRule)
    mu             sync.RWMutex
    stopChan       chan struct{}
    healthTicker   *time.Ticker
}

// NewConfigurationWatcher creates a new watcher with Redis pub/sub subscription.
func NewConfigurationWatcher(
    redisClient redis.UniversalClient,
    ruleManager *config.RuleManager,
    updateCallback func([]*config.RateLimitRule),
) *ConfigurationWatcher {
    return &ConfigurationWatcher{
        redisClient:    redisClient,
        ruleManager:    ruleManager,
        changeChannel:  "rate_limit:config:changes",
        updateCallback: updateCallback,
        stopChan:       make(chan struct{}),
        healthTicker:   time.NewTicker(30 * time.Second),
    }
}

// Start begins listening for configuration changes and performing health checks.
func (cw *ConfigurationWatcher) Start(ctx context.Context) error {
    // TODO 1: Subscribe to Redis configuration change channel
    // TODO 2: Start background goroutine for processing change notifications
    // TODO 3: Start health check goroutine for periodic version comparison
    // TODO 4: Perform initial configuration load to sync current state
    // TODO 5: Handle subscription reconnection on Redis connection failures
```

```
    // Hint: Use cw.processChangeNotifications() in a separate goroutine
    // Hint: Implement exponential backoff for reconnection attempts
    // Hint: Log all configuration changes for audit trail

    panic("TODO: Implement configuration watcher startup logic")
}

// processChangeNotifications handles incoming configuration change events.
func (cw *ConfigurationWatcher) processChangeNotifications() {
    // TODO 1: Listen for messages on the Redis pub/sub channel
    // TODO 2: Parse change notification payload (rule ID, change type, version)
    // TODO 3: Fetch updated rule data from Redis configuration store
    // TODO 4: Validate new configuration for consistency and correctness
    // TODO 5: Atomically update local rule cache and notify callback
    // TODO 6: Update local version number and heartbeat timestamp

    // Hint: Handle different change types: CREATE, UPDATE, DELETE
    // Hint: Implement rollback logic if validation fails
    // Hint: Use atomic operations to prevent inconsistent local state

    panic("TODO: Implement change notification processing")
}

// performHealthCheck compares local vs remote configuration versions.
func (cw *ConfigurationWatcher) performHealthCheck(ctx context.Context) error {
    // TODO 1: Fetch current configuration version from Redis
    // TODO 2: Compare with local version to detect drift
    // TODO 3: If versions differ, trigger full configuration reload
    // TODO 4: Update heartbeat record with instance status
    // TODO 5: Log any version skew or synchronization issues

    // Hint: Use Redis GET on "config:version" key
    // Hint: Implement full reload as fallback for partial update failures

    panic("TODO: Implement configuration health checking")
}
```

Metrics Collection Infrastructure

```go
// MetricsCollector aggregates rate limiting metrics and detects hot keys
// across multiple time windows for real-time monitoring and alerting.
type MetricsCollector struct {
    keyStats       sync.Map // map[string]*KeyStats
    globalStats    *GlobalStats
    hotKeyDetector *HotKeyDetector
    exportChan     chan *MetricsBatch
    windowSize     time.Duration
    mu             sync.RWMutex
}

// MetricsBatch represents a collection of metrics ready for export.
type MetricsBatch struct {
    Timestamp       time.Time              `json:"timestamp"`
    RequestMetrics  map[string]*RequestMetrics `json:"request_metrics"`
    PerformanceMetrics *PerformanceMetrics  `json:"performance_metrics"`
    HotKeys        []string               `json:"hot_keys"`
    SystemHealth   *SystemHealth          `json:"system_health"`
}

// RequestMetrics tracks request volume and outcomes for a specific rate limit key.
type RequestMetrics struct {
    TotalRequests    int64 `json:"total_requests"`
    AllowedRequests  int64 `json:"allowed_requests"`
    DeniedRequests   int64 `json:"denied_requests"`
    ErrorRequests    int64 `json:"error_requests"`
    LastSeen         time.Time `json:"last_seen"`
}

// NewMetricsCollector creates a metrics collector with configurable aggregation windows.
func NewMetricsCollector(windowSize time.Duration, exportChan chan *MetricsBatch) *MetricsCollector {
    return &MetricsCollector{
        globalStats:    NewGlobalStats(),
        hotKeyDetector: NewHotKeyDetector(),
        exportChan:     exportChan,
        windowSize:     windowSize,
    }
}
```

```go
// RecordRequest updates metrics for a single rate limit request.
func (mc *MetricsCollector) RecordRequest(key string, result *storage.RateLimitResult, duration time.Duration) {
    // TODO 1: Update key-specific request counters (total, allowed, denied)
    // TODO 2: Record request in hot key detector for trend analysis
    // TODO 3: Update global performance metrics (latency histograms)
    // TODO 4: Track algorithm-specific metrics if available
    // TODO 5: Update last-seen timestamp for key activity tracking

    // Hint: Use sync.Map for concurrent access to key statistics
    // Hint: Implement lock-free counters using atomic operations where possible
    // Hint: Consider sampling high-volume keys to reduce memory usage

    panic("TODO: Implement request metrics recording")
}

// ExportMetrics produces a metrics batch for external consumption.
func (mc *MetricsCollector) ExportMetrics() *MetricsBatch {
    // TODO 1: Aggregate all key-level metrics into exportable format
    // TODO 2: Calculate performance percentiles and rates
    // TODO 3: Run hot key detection algorithm on recent data
    // TODO 4: Reset or rotate metrics windows to prevent unbounded growth
    // TODO 5: Package everything into a comprehensive metrics batch

    // Hint: Use time-based bucketing for efficient percentile calculation
    // Hint: Implement memory limits to prevent metrics from consuming too much RAM

    panic("TODO: Implement metrics export and aggregation")
}
```

Milestone Checkpoints

After implementing the rate limit check flow:

- Run: `go test ./internal/flow/ -run TestRequestFlow`
- Verify: Multi-tier evaluation works with short-circuit logic
- Test manually: Send requests that exceed different tier limits and confirm proper blocking

After implementing configuration propagation:

- Run: `go test ./internal/flow/ -run TestConfigWatcher`
- Verify: Configuration changes propagate to all instances within 5 seconds
- Test manually: Update a rule via API and confirm all instances receive the change

After implementing metrics collection:

- Run: `go test ./internal/flow/ -run TestMetricsCollection`
- Verify: Hot key detection identifies keys with >10x normal traffic
- Test manually: Generate high traffic to specific keys and confirm dashboard updates

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Inconsistent rate limiting	Configuration version skew	Check heartbeat timestamps in Redis	Force configuration reload
High latency on checks	Redis connection pool exhaustion	Monitor connection pool metrics	Increase pool size or add connection timeout
Missing metrics data	Metrics export blocking	Check exportChan buffer size	Use buffered channel or async export
False hot key detection	Clock skew between instances	Compare timestamps across instances	Implement NTP sync or relative timing

Rate Limit API and Dashboard

> **Milestone(s):** Milestone 5 - Rate Limit API & Dashboard

The operational success of any distributed rate limiting system fundamentally depends on its management and observability capabilities. While the core algorithms and Redis backend provide the foundation for rate limiting functionality, the management API and dashboard serve as the critical interface between human operators and the distributed system. This section explores the design of a comprehensive rate limit management system that enables dynamic configuration updates, provides real-time visibility into system behavior, and maintains operational safety through self-protection mechanisms.

Mental Model: The Air Traffic Control System

Think of the rate limit API and dashboard as an air traffic control system for request traffic. Just as air traffic controllers need real-time radar displays showing aircraft positions, flight paths, and airport capacity, rate limit operators need dashboards showing current request rates, quota utilization, and system health across all tiers. The management API acts like the control tower radio system, allowing controllers to dynamically adjust flight patterns (rate limit rules) based on changing conditions without shutting down the airport (restarting services).

The control tower must also protect itself - it can't allow unlimited radio chatter that would overwhelm the controllers' ability to manage air traffic. Similarly, the rate limit management API must implement self-rate-limiting to prevent its own operations from overwhelming the very system it manages. This creates an interesting bootstrap problem: how do you rate limit the rate limiter without creating circular dependencies?

Rate Limit Management API

The **Rate Limit Management API** serves as the primary interface for creating, reading, updating, and deleting rate limit rules across the entire distributed system. Unlike static configuration files that require service restarts, this API enables dynamic rule management with immediate propagation to all application instances through Redis pub/sub mechanisms.

API Design Philosophy

The management API follows REST principles with a strong emphasis on validation, versioning, and audit trails. Every rule change creates an audit log entry, enabling operators to understand how rate limiting behavior evolved over time. The API supports both immediate rule activation and scheduled deployments, allowing operators to prepare rate limit changes during low-traffic periods and activate them precisely when needed.

> **Decision: RESTful API with Resource-Based URLs**
>
> - **Context**: Need to expose CRUD operations for rate limit rules with clear semantics and easy integration
> - **Options Considered**: REST API, GraphQL API, gRPC API
> - **Decision**: REST API with resource-based URLs following `/api/v1/rules/{id}` pattern
> - **Rationale**: REST provides familiar semantics for CRUD operations, excellent HTTP caching support, and broad client library support across languages
> - **Consequences**: Enables standard HTTP tooling and caching but requires multiple requests for complex operations

The API resource structure centers around the `RateLimitRule` entity with hierarchical organization supporting rule grouping, inheritance, and override patterns:

Endpoint	Method	Description	Request Body	Response
`/api/v1/rules`	GET	List all rate limit rules with filtering	Query parameters	Rule list with pagination
`/api/v1/rules`	POST	Create new rate limit rule	`RateLimitRule` JSON	Created rule with assigned ID
`/api/v1/rules/{id}`	GET	Retrieve specific rule by ID	None	Complete `RateLimitRule` object
`/api/v1/rules/{id}`	PUT	Update existing rule completely	Complete `RateLimitRule`	Updated rule object
`/api/v1/rules/{id}`	PATCH	Partial rule update	Partial `RateLimitRule`	Updated rule object
`/api/v1/rules/{id}`	DELETE	Remove rule and stop enforcement	None	Deletion confirmation
`/api/v1/rules/{id}/preview`	POST	Test rule against sample requests	`RateLimitRequest` array	Preview results without enforcement
`/api/v1/rules/{id}/reset`	POST	Clear all counters for rule	Optional key filter	Reset confirmation
`/api/v1/rules/bulk`	POST	Bulk create/update operations	Rule array with operations	Bulk operation results

Rule Validation and Constraints

The API implements comprehensive validation ensuring that rate limit rules are syntactically correct, semantically meaningful, and operationally safe before activation. Validation occurs at multiple levels: syntax validation checks JSON structure and data types, semantic validation ensures logical consistency between fields, and operational validation prevents rules that could destabilize the system.

Syntax Validation Rules:

Field	Validation Rule	Error Message
`name`	Non-empty string, max 100 chars	"Rule name required and must be under 100 characters"
`key_pattern`	Valid regex pattern	"Key pattern must be valid regular expression"
`algorithm`	One of: token_bucket, sliding_window_counter, sliding_window_log	"Algorithm must be supported type"
`limit`	Positive integer > 0	"Rate limit must be positive integer"
`window`	Duration between 1s and 24h	"Time window must be between 1 second and 24 hours"
`burst_limit`	Optional, >= limit if specified	"Burst limit must be >= base limit when specified"
`priority`	Integer between 1 and 100	"Priority must be between 1 (low) and 100 (high)"

Semantic Validation Rules:

The API performs semantic validation to catch logical inconsistencies that could lead to unexpected behavior. For example, a rule with `algorithm: "token_bucket"` must specify a `burst_limit` since token buckets are specifically designed for burst handling. Similarly, rules targeting the same key pattern cannot have conflicting priorities that would create ambiguous evaluation order.

Operational Safety Validation:

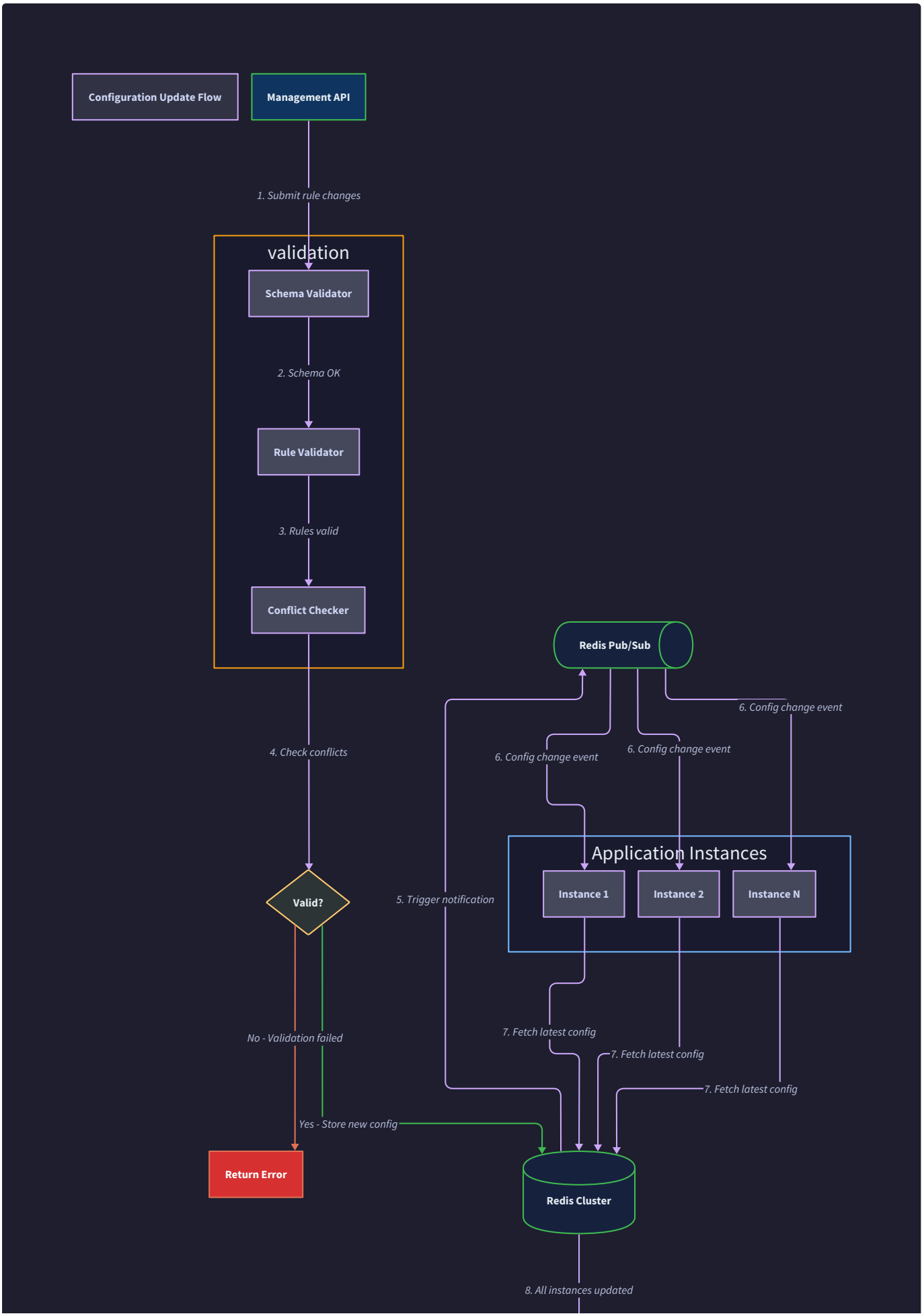
To prevent operators from accidentally creating rules that could destabilize the system, the API enforces operational safety constraints. Rules with extremely low limits (less than 10 requests per minute) trigger warnings and require explicit confirmation. Rules with very short time windows (less than 10 seconds) are flagged as potentially problematic for distributed consistency. The API also prevents creation of more than 1000 active rules per system to avoid overwhelming the Redis backend with excessive key space consumption.

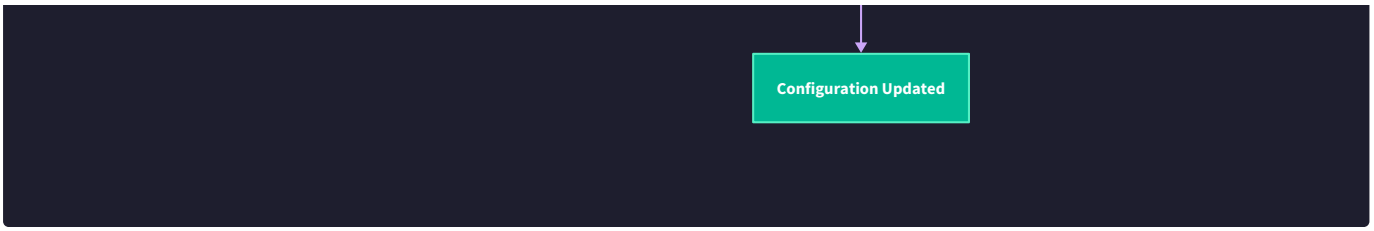
Rule Versioning and Rollback

Every rule modification creates a new version while preserving the complete history of changes. This versioning system enables rapid rollback to previous configurations when new rules cause unexpected behavior. The versioning implementation stores rule snapshots in Redis with timestamp-based keys, enabling point-in-time recovery and audit trail reconstruction.

Version Operation	Endpoint	Description	Impact
Create Version	PUT/PATCH `/api/v1/rules/{id}`	Automatic version creation on modification	New version stored, old version archived
List Versions	GET `/api/v1/rules/{id}/versions`	Retrieve complete version history	No impact, read-only operation
View Version	GET `/api/v1/rules/{id}/versions/{version}`	Retrieve specific historical version	No impact, read-only operation
Rollback	POST `/api/v1/rules/{id}/rollback`	Revert to specified previous version	Immediate rule update, new version created

Configuration Propagation Mechanism


Configuration Update Flow
Management API
1. Submit rule changes
validation
Schema Validator
2. Schema OK
Rule Validator
3. Rules valid
Conflict Checker
4. Check conflicts
Valid?
No - Validation failed
Return Error
Yes - Store new config
Redis Cluster
8. All instances updated
Redis Pub/Sub
5. Trigger notification
6. Config change event
6. Config change event
6. Config change event
Application Instances
Instance 1
Instance 2
Instance N
7. Fetch latest config
7. Fetch latest config
7. Fetch latest config



When a rule is created or modified through the API, the change must propagate to all application instances running the distributed rate limiter. This propagation occurs through a multi-stage process designed to ensure consistency while minimizing latency and avoiding race conditions.

Propagation Stages:

1. **Validation and Persistence**: The API validates the incoming rule change and persists it to Redis with a unique version number and timestamp
2. **Change Notification**: A change notification containing the rule ID and version is published to the Redis channel `rate_limit_config_changes`
3. **Instance Subscription**: All application instances maintain Redis subscriptions to the configuration change channel
4. **Version Comparison**: Upon receiving a change notification, each instance compares the announced version with its local cache
5. **Rule Retrieval**: Instances with outdated versions fetch the complete updated rule from Redis
6. **Local Cache Update**: The instance updates its local rule cache and begins enforcing the new configuration
7. **Acknowledgment**: Each instance publishes an acknowledgment to confirm successful configuration update

This propagation mechanism handles network partitions and instance failures gracefully. Instances that miss change notifications due to temporary disconnections perform periodic configuration health checks, comparing their local version numbers against Redis to detect missed updates. The system also implements exponential backoff retry logic when instances cannot reach Redis during configuration updates.

> The critical design insight here is that configuration propagation must be eventually consistent rather than strongly consistent. Temporary inconsistencies where different instances enforce slightly different rules for a few seconds are acceptable, but permanent inconsistencies where some instances never receive updates are not.

Audit Trail and Change History

Every API operation creates detailed audit log entries capturing who made what changes when and why. These audit logs prove essential for troubleshooting unexpected rate limiting behavior and maintaining compliance with change management processes. The audit system records both successful operations and failed attempts, enabling security teams to detect unauthorized access attempts.

Audit Event Type	Logged Information	Retention Period
Rule Creation	User ID, rule content, timestamp, source IP	1 year
Rule Modification	User ID, old values, new values, change reason, timestamp	1 year
Rule Deletion	User ID, deleted rule content, deletion reason, timestamp	1 year
Bulk Operations	User ID, operation list, success/failure per rule, timestamp	1 year
Preview Requests	User ID, rule ID, test scenarios, results, timestamp	30 days
Reset Operations	User ID, rule ID, affected keys, timestamp	90 days
Authentication Failures	Attempted user ID, source IP, failure reason, timestamp	6 months
Authorization Denials	User ID, attempted operation, denial reason, timestamp	6 months

Standard Rate Limit Headers

The distributed rate limiter implements standard HTTP headers following RFC 6585 and industry best practices to provide clients with actionable information about their rate limiting status. These headers enable clients to implement intelligent retry logic, display accurate quota information to users, and avoid unnecessarily aggressive request patterns that could trigger additional rate limiting.

Header Implementation Strategy

The header implementation follows a tiered approach where different header sets are included based on the rate limiting result. Successful requests include current quota information, denied requests include retry guidance, and error scenarios include minimal diagnostic information to avoid information leakage.

Decision: RFC 6585 Compliance with Extensions

- **Context**: Need standardized headers for client rate limit awareness and retry logic
- **Options Considered**: Custom headers, RFC 6585 standard, GitHub-style headers
- **Decision**: Implement RFC 6585 with additional headers for enhanced client experience
- **Rationale**: RFC compliance ensures broad client library support while extensions provide operational benefits
- **Consequences**: Enables standard tooling but requires careful header size management to avoid HTTP limits

Standard Rate Limit Headers:

Header Name	When Included	Value Format	Example	Purpose
`X-RateLimit-Limit`	All responses	Integer	`1000`	Maximum requests allowed in current window
`X-RateLimit-Remaining`	All responses	Integer	`742`	Requests remaining in current window
`X-RateLimit-Reset`	All responses	Unix timestamp	`1640995200`	When the current window resets
`X-RateLimit-Window`	All responses	Duration in seconds	`3600`	Length of rate limit window
`Retry-After`	429 responses only	Seconds until retry	`45`	Minimum wait time before retry attempt

Extended Headers for Enhanced Client Experience:

Header Name	When Included	Value Format	Example	Purpose
`X-RateLimit-Policy`	All responses	Algorithm identifier	`token_bucket`	Which rate limiting algorithm is active
`X-RateLimit-Scope`	All responses	Scope identifier	`user:12345`	What entity the limit applies to
`X-RateLimit-Burst`	Token bucket responses	Integer	`1500`	Maximum burst capacity available
`X-RateLimit-Tier`	Multi-tier responses	Tier name	`per-user`	Which tier triggered the limit

Multi-Tier Header Aggregation

When multiple rate limit tiers apply to a single request, the header generation logic must aggregate information from all applicable tiers to provide clients with actionable guidance. The aggregation follows a "most restrictive wins" principle where the tightest remaining quota determines the header values.

Tier Aggregation Logic:

1. **Collect All Applicable Limits**: Gather rate limit results from user, IP, API, and global tiers
2. **Identify Most Restrictive**: Find the tier with the lowest `remaining/limit` ratio
3. **Check for Denials**: If any tier denies the request, use that tier's information for headers
4. **Calculate Aggregate Reset**: Use the earliest reset time across all tiers
5. **Compose Scope Header**: Create comma-separated list of all applicable scopes

For example, a request that passes user-level limits (900/1000 remaining) but approaches IP-level limits (15/100 remaining) would generate headers reflecting the IP limit since it represents the most immediate constraint on future requests.

Client Retry Guidance

The `Retry-After` header calculation considers not just the immediate rate limit violation but also the broader context of multi-tier limits and algorithm-specific behavior. For token bucket algorithms, the retry time reflects the token refill rate. For sliding window algorithms, it considers both the window reset time and the distribution of recent requests.

Retry-After Calculation Algorithm:

Algorithm	Calculation Method	Rationale
Token Bucket	`max(1, (tokens_needed - current_tokens) / refill_rate)`	Time to accumulate sufficient tokens
Sliding Window Counter	`window_duration - time_since_window_start + jitter`	Wait for window boundary plus randomization
Sliding Window Log	`oldest_request_timestamp + window_duration - current_time + jitter`	Wait for oldest request to age out

The retry calculation includes small random jitter (5-15% of the base retry time) to prevent thundering herd effects where many clients retry simultaneously after receiving identical `Retry-After` values.

Real-time Dashboard Architecture

The **real-time dashboard** provides operational visibility into rate limiting behavior across all tiers, algorithms, and time scales. Unlike traditional monitoring dashboards that display historical data, this dashboard focuses on current quota utilization, active patterns, and immediate operational decisions that rate limit administrators need to make.

Mental Model: Power Grid Control Room

Think of the rate limiting dashboard like a power grid control room where operators monitor electricity demand across different regions and time scales. Just as grid operators need real-time visibility into current load, peak capacity, and demand forecasting to prevent blackouts, rate limit operators need current quota utilization, request patterns, and trend analysis to prevent service degradation. The dashboard must update continuously without overwhelming the underlying rate limiting system, just as grid monitoring cannot consume significant power itself.

Dashboard Data Architecture

The dashboard architecture separates data collection, aggregation, and presentation into distinct layers to ensure that dashboard operations never impact rate limiting performance. Data flows through a pipeline from individual rate limit checks through local aggregation, Redis-based consolidation, and finally WebSocket streaming to dashboard clients.

Data Flow Layers:

Layer	Component	Responsibility	Update Frequency
Collection	`MetricsCollector`	Capture individual rate limit results	Per request
Local Aggregation	`MetricsBatch`	Combine metrics within single instance	Every 5 seconds
Redis Consolidation	Redis Streams	Merge metrics from all instances	Every 10 seconds
Dashboard Streaming	WebSocket Server	Push updates to dashboard clients	Every 2 seconds

Real-time Metrics Collection

The `MetricsCollector` component captures detailed information about every rate limit decision without impacting the critical path performance. Collection uses lock-free data structures and asynchronous batching to ensure that metrics gathering never blocks rate limit checks.

Collected Metric Categories:

Metric Category	Data Points	Aggregation Method	Purpose
Request Counts	Total, allowed, denied by rule	Sum over time windows	Quota utilization tracking
Response Times	P50, P95, P99 latency by tier	Histogram bucketing	Performance monitoring
Algorithm Performance	Token refill rates, window efficiency	Moving averages	Algorithm tuning guidance
Error Rates	Redis timeouts, fallback activation	Count and rate calculation	System health assessment
Hot Key Detection	Request distribution across keys	Top-K tracking	Load balancing insights

WebSocket-Based Real-Time Updates

The dashboard uses WebSocket connections to stream real-time updates to client browsers without polling overhead. The WebSocket server implements intelligent update batching and client-specific filtering to ensure that each dashboard user receives only relevant updates without overwhelming their browser.

WebSocket Message Types:

Message Type	Payload Structure	Update Frequency	Client Response
`quota_update`	`{rule_id, current, limit, remaining}`	Every 2 seconds	Update quota gauges
`tier_status`	`{tier_name, active_rules, total_requests}`	Every 5 seconds	Refresh tier summary
`hot_keys`	`{key, request_rate, trend}`	Every 10 seconds	Update hot key list
`system_health`	`{redis_status, fallback_rate, error_count}`	Every 5 seconds	Update health indicators
`rule_change`	`{operation, rule_id, new_config}`	Immediate	Refresh rule display

Dashboard User Interface Design

The dashboard interface organizes information hierarchically from system overview through tier-specific views to individual rule analysis. Each view provides actionable information at the appropriate level of detail for different operational decisions.

Dashboard View Hierarchy:

1. **System Overview**: High-level health, total request rates, and critical alerts across all tiers
2. **Tier Dashboard**: Detailed view of user, IP, API, or global tier with rule-by-rule breakdown
3. **Rule Analysis**: Deep dive into individual rule performance, quota utilization patterns, and historical trends
4. **Hot Key Investigation**: Real-time analysis of disproportionately accessed keys with geographic and temporal patterns
5. **System Health**: Redis cluster status, instance connectivity, and performance metrics

Dashboard Performance Optimization

To ensure the dashboard remains responsive under high load conditions, several optimization strategies prevent dashboard operations from impacting rate limiting performance or overwhelming client browsers with excessive updates.

> **Decision: Adaptive Update Frequency Based on Activity Level**
>
> - **Context**: Dashboard updates must be frequent enough for operational decisions but not so frequent as to impact performance
> - **Options Considered**: Fixed update intervals, event-driven updates, adaptive frequency
> - **Decision**: Implement adaptive update frequency that increases during high activity and decreases during stable periods
> - **Rationale**: Provides real-time visibility when needed while conserving resources during normal operations
> - **Consequences**: More complex implementation but significantly better resource efficiency

Optimization Strategies:

Strategy	Implementation	Benefit	Trade-off
Client-Side Filtering	Dashboard clients specify interest filters	Reduced bandwidth usage	More complex client logic
Update Batching	Group multiple updates into single WebSocket messages	Lower message overhead	Slightly increased latency
Adaptive Frequency	Increase update rate during anomalies, decrease during stability	Dynamic resource usage	Complex triggering logic
Data Compression	Compress WebSocket payloads for large updates	Reduced network usage	CPU overhead for compression
Local Caching	Cache unchanged data on client side	Minimal redundant updates	Client-side memory usage

Self-Rate-Limiting the Management API

The management API faces a unique challenge: it must protect itself from abuse without creating circular dependencies with the rate limiting system it manages. This creates a bootstrap problem where the rate limiter cannot rely on its own Redis backend for API protection since API operations might be necessary to fix Redis connectivity issues.

Mental Model: Emergency Services Communication

Think of the management API like emergency services communication systems that must remain operational even when the systems they manage are failing. A fire department's radio system cannot depend on the electrical grid they might need to repair - it needs independent power and communication channels. Similarly, the rate limit management API needs independent protection mechanisms that don't depend on the distributed rate limiting infrastructure it controls.

Independent Rate Limiting Strategy

The management API implements a lightweight, in-memory rate limiting system that operates independently of the main Redis-backed rate limiter. This independent system uses simpler algorithms and local state to provide basic protection without external dependencies.

> **Decision: Local Token Bucket for API Self-Protection**
>
> - **Context**: Management API needs rate limiting protection that doesn't depend on Redis or distributed state
> - **Options Considered**: Local token bucket, fixed window counters, dependency on main rate limiter
> - **Decision**: Implement local token bucket with per-client tracking using IP address + API key
> - **Rationale**: Token bucket provides burst handling for legitimate batch operations while preventing sustained abuse
> - **Consequences**: Protection is per-instance rather than cluster-wide, but maintains independence from Redis

Self-Rate-Limiting Architecture:

Component	Implementation	Responsibility	Independence Level
`APITokenBucket`	In-memory token bucket per client	Request rate control	Fully independent
`ClientIdentifier`	IP + API key hashing	Client identification	No external dependencies
`LocalRateLimiter`	Embedded rate limiter	API call limiting	Memory-only state
`OverrideManager`	Emergency bypass tokens	Incident response	File-based configuration

API-Specific Rate Limit Tiers

The management API implements multiple tiers of rate limiting tailored to different operation types and client authentication levels. Read operations receive higher quotas than write operations, and authenticated API clients receive higher limits than unauthenticated requests.

API Rate Limit Tiers:

Operation Type	Authenticated Limit	Unauthenticated Limit	Burst Allowance	Rationale
Read Operations	100 req/min	20 req/min	150% of base	Dashboard updates need frequent reads
Write Operations	20 req/min	5 req/min	130% of base	Rule changes should be deliberate
Bulk Operations	5 req/min	0 req/min	110% of base	Bulk changes need careful consideration
Preview Operations	50 req/min	10 req/min	200% of base	Testing scenarios may need rapid iteration
Reset Operations	10 req/min	0 req/min	100% of base	Counter resets have immediate system impact

Emergency Override Mechanisms

During critical incidents where rate limiting rules may be causing service outages, operators need the ability to bypass normal API rate limits to implement emergency fixes. The override system provides this capability through multiple authentication factors and audit trails.

Override Activation Methods:

Override Type	Activation Requirement	Duration	Audit Level
Temporary Bypass	Admin API key + incident ticket	1 hour	Full request logging
Emergency Token	Pre-generated token + manager approval	30 minutes	Real-time notification
File-Based Override	Server file system access	Until file removal	File system audit trail
Multi-Admin Confirmation	Two admin API keys simultaneously	2 hours	Multi-party audit logging

The emergency override system ensures that legitimate incident response can proceed quickly while maintaining strong audit trails and preventing abuse. Override usage triggers immediate notifications to security teams and creates detailed logs for post-incident review.

API Protection Without Circular Dependencies

To avoid circular dependencies where the management API rate limiting depends on the systems it manages, the implementation carefully isolates API protection from distributed state. The local rate limiter uses only in-memory data structures and local file system access for persistent configuration.

Dependency Isolation Strategies:

1. **Separate Algorithm Implementation**: The API uses a simplified token bucket implementation that shares no code with the main distributed rate limiter
2. **Local State Only**: All API rate limiting state remains in local memory, never touching Redis or shared storage
3. **Independent Configuration**: API rate limits are configured through environment variables or local files, not through the main rule management system
4. **Graceful Fallback**: When local rate limiting fails, the API defaults to conservative limits rather than disabling protection
5. **Health Check Separation**: API health checks use direct Redis connectivity tests rather than going through the rate limiting system

> The fundamental design principle here is defense in depth - the management API must remain secure and operational even when every other component of the distributed rate limiting system is failing or misconfigured.

Common Pitfalls

⚠️ Pitfall: Dashboard Overwhelming Rate Limiter with Monitoring Queries

A common mistake is implementing dashboard updates that query rate limiting state so frequently that the monitoring system becomes a significant load on the Redis backend. This creates a feedback loop where dashboard usage impacts rate limiting performance, which triggers more frequent dashboard updates to investigate the performance problem.

Why it's wrong: Dashboard queries compete with production rate limiting operations for Redis resources, potentially causing legitimate rate limit checks to fail or experience high latency.

How to fix: Implement async metrics collection where rate limiting operations push metrics to a separate data path rather than having the dashboard pull current state. Use Redis Streams or pub/sub to decouple dashboard data from operational state.

⚠️ Pitfall: Configuration Changes Without Validation Rollback

When implementing dynamic configuration updates, developers often forget to implement automatic rollback when new configurations cause system instability. A rule change that accidentally sets all limits to 1 request per hour can cause immediate service outages.

Why it's wrong: Bad configurations can render the entire service unusable, and manual rollback takes time during which the service remains degraded.

How to fix: Implement automatic configuration validation that tests new rules against recent traffic patterns and automatically rolls back changes that cause denial rates to spike above configurable thresholds.

⚠️ Pitfall: Management API Using Same Redis Instance as Rate Limiting

Some implementations use the same Redis instance for both rate limiting operations and management API data storage. This creates a single point of failure where Redis issues impact both operational rate limiting and the ability to fix configuration problems.

Why it's wrong: When Redis becomes unavailable, operators lose both rate limiting functionality and the ability to modify configuration to address the issue.

How to fix: Use separate Redis instances or different Redis databases for operational rate limiting and management data. Consider using a different storage backend entirely for management API data to ensure independence.

⚠️ Pitfall: Rate Limit Headers Calculated After Request Processing

A subtle error occurs when rate limit headers are calculated after the main request processing rather than during the rate limit check. This can result in headers that don't reflect the actual rate limiting decision or show inconsistent quota information.

Why it's wrong: Clients receive misleading information about their rate limiting status, leading to incorrect retry behavior and poor user experience.

How to fix: Calculate all rate limit headers during the actual rate limit check operation and pass them through the request context to ensure header consistency with the rate limiting decision.

Implementation Guidance

The rate limit API and dashboard implementation requires careful coordination between HTTP API handling, real-time data streaming, and independent rate limiting logic. This section provides complete implementation guidance for building these components.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP API Framework	`net/http` with `gorilla/mux` router	`gin-gonic/gin` with OpenAPI generation
WebSocket Server	`gorilla/websocket` library	`Socket.io` with Redis adapter
Real-time Metrics	In-memory aggregation with periodic flush	Redis Streams with consumer groups
Dashboard Frontend	Server-rendered HTML with vanilla JS	React/Vue SPA with Chart.js visualization
API Authentication	JWT tokens with HMAC signing	OAuth2 with RBAC authorization
Configuration Storage	JSON files with file watching	etcd with distributed locking

Recommended Module Structure

```
project-root/
  cmd/
    rate-limiter/main.go          ← main rate limiting service
    management-api/main.go        ← management API server
    dashboard/main.go             ← dashboard web server
  internal/
    api/
      handlers/
        rules.go                  ← rate limit rule CRUD handlers
        dashboard.go              ← dashboard data API handlers
        websocket.go              ← real-time WebSocket server
      middleware/
        auth.go                   ← API authentication middleware
        ratelimit.go              ← self-rate-limiting middleware
        headers.go                ← rate limit header injection
      validation/
        rules.go                  ← rule validation logic
        constraints.go            ← operational safety constraints
    dashboard/
      metrics/
        collector.go              ← metrics collection and aggregation
        streaming.go              ← WebSocket streaming logic
        batch.go                  ← metric batching and compression
      ui/
        templates/                ← HTML templates for dashboard
        static/                   ← CSS, JS, and image assets
    config/
      propagation.go              ← configuration change propagation
      versioning.go               ← rule versioning and rollback
      audit.go                    ← audit trail implementation
  web/
    dashboard/
      index.html                  ← main dashboard interface
      js/dashboard.js             ← dashboard JavaScript logic
      css/styles.css              ← dashboard styling
```

Infrastructure Starter Code

Complete API Rate Limiter (api/middleware/ratelimit.go):

```go
package middleware

import (
    "fmt"
    "net/http"
    "sync"
    "time"
    "net"
    "strings"
)

// APIRateLimiter provides self-rate-limiting for management API
type APIRateLimiter struct {
    mu          sync.RWMutex
    clients     map[string]*TokenBucket
    cleanup     time.Duration
    limits      map[string]APILimits
}

// APILimits defines rate limits for different API operation types
type APILimits struct {
    ReadLimit   int64         // requests per minute for read operations
    WriteLimit  int64         // requests per minute for write operations
    BulkLimit   int64         // requests per minute for bulk operations
    BurstRatio  float64       // burst allowance ratio (e.g., 1.5 = 150%)
}

// TokenBucket implements simple in-memory token bucket for API protection
type TokenBucket struct {
    capacity    int64     // maximum tokens
    tokens      int64     // current tokens
    refillRate  int64     // tokens per minute
    lastRefill  time.Time // last refill timestamp
    mu          sync.Mutex
}

// NewAPIRateLimiter creates self-rate-limiter for management API
func NewAPIRateLimiter() *APIRateLimiter {
    limits := map[string]APILimits{
        "authenticated": {
```

```
            ReadLimit:  100,
            WriteLimit: 20,
            BulkLimit:  5,
            BurstRatio: 1.5,
        },
        "unauthenticated": {
            ReadLimit:  20,
            WriteLimit: 5,
            BulkLimit:  0,
            BurstRatio: 1.3,
        },
    }

    rl := &APIRateLimiter{
        clients: make(map[string]*TokenBucket),
        cleanup: 5 * time.Minute,
        limits:  limits,
    }

    // Start background cleanup of inactive clients
    go rl.cleanupLoop()

    return rl
}

// Middleware returns HTTP middleware function for request rate limiting
func (rl *APIRateLimiter) Middleware() func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            // Extract client identifier and operation type
            clientID := rl.getClientID(r)
            opType := rl.getOperationType(r)
            authLevel := rl.getAuthLevel(r)

            // Check rate limit for this client and operation
            allowed, remaining, resetTime := rl.checkLimit(clientID, opType, authLevel)

            // Add rate limit headers to response
```

```
            w.Header().Set("X-RateLimit-Limit", fmt.Sprintf("%d", rl.getLimit(opType, authLevel)))
            w.Header().Set("X-RateLimit-Remaining", fmt.Sprintf("%d", remaining))
            w.Header().Set("X-RateLimit-Reset", fmt.Sprintf("%d", resetTime.Unix()))
            w.Header().Set("X-RateLimit-Scope", fmt.Sprintf("api:%s", clientID))

            if !allowed {
                // Calculate retry after based on refill rate
                retryAfter := rl.calculateRetryAfter(opType, authLevel)
                w.Header().Set("Retry-After", fmt.Sprintf("%d", int(retryAfter.Seconds())))

                http.Error(w, "API rate limit exceeded", http.StatusTooManyRequests)
                return
            }

            next.ServeHTTP(w, r)
        })
    }
}

// checkLimit performs rate limiting check for client and operation type
func (rl *APIRateLimiter) checkLimit(clientID, opType, authLevel string) (bool, int64, time.Time) {
    rl.mu.Lock()
    defer rl.mu.Unlock()

    // Get or create token bucket for this client
    bucket, exists := rl.clients[clientID]
    if !exists {
        limit := rl.getLimit(opType, authLevel)
        burstRatio := rl.limits[authLevel].BurstRatio
        capacity := int64(float64(limit) * burstRatio)

        bucket = &TokenBucket{
            capacity:   capacity,
            tokens:     capacity,
            refillRate: limit,
            lastRefill: time.Now(),
        }
        rl.clients[clientID] = bucket
```

```
    }

    return bucket.consume(1)
}

// consume attempts to consume tokens from bucket
func (tb *TokenBucket) consume(tokens int64) (bool, int64, time.Time) {
    tb.mu.Lock()
    defer tb.mu.Unlock()

    now := time.Now()

    // Calculate tokens to add based on time elapsed
    elapsed := now.Sub(tb.lastRefill)
    tokensToAdd := int64(elapsed.Minutes() * float64(tb.refillRate))

    if tokensToAdd > 0 {
        tb.tokens = min(tb.capacity, tb.tokens + tokensToAdd)
        tb.lastRefill = now
    }

    // Calculate next reset time (when bucket will be full)
    resetTime := now.Add(time.Duration((tb.capacity - tb.tokens) * 60 / tb.refillRate) * time.Second)

    if tb.tokens >= tokens {
        tb.tokens -= tokens
        return true, tb.tokens, resetTime
    }

    return false, tb.tokens, resetTime
}

// Helper functions for client identification and operation classification
func (rl *APIRateLimiter) getClientID(r *http.Request) string {
    // Try API key first, fall back to IP address
    if apiKey := r.Header.Get("X-API-Key"); apiKey != "" {
        return fmt.Sprintf("key:%s", apiKey)
    }
```

```
    ip := rl.extractIP(r)
    return fmt.Sprintf("ip:%s", ip)
}

func (rl *APIRateLimiter) extractIP(r *http.Request) string {
    // Check X-Forwarded-For header for proxy scenarios
    if xff := r.Header.Get("X-Forwarded-For"); xff != "" {
        parts := strings.Split(xff, ",")
        if len(parts) > 0 {
            return strings.TrimSpace(parts[0])
        }
    }

    // Fall back to RemoteAddr
    ip, _, err := net.SplitHostPort(r.RemoteAddr)
    if err != nil {
        return r.RemoteAddr
    }
    return ip
}

func (rl *APIRateLimiter) getOperationType(r *http.Request) string {
    switch r.Method {
    case "GET":
        return "read"
    case "PUT", "PATCH", "DELETE":
        return "write"
    case "POST":
        if strings.Contains(r.URL.Path, "/bulk") {
            return "bulk"
        }
        return "write"
    default:
        return "read"
    }
}

func (rl *APIRateLimiter) getAuthLevel(r *http.Request) string {
```

```
    if r.Header.Get("X-API-Key") != "" {
        return "authenticated"
    }
    return "unauthenticated"
}

func (rl *APIRateLimiter) getLimit(opType, authLevel string) int64 {
    limits := rl.limits[authLevel]
    switch opType {
    case "read":
        return limits.ReadLimit
    case "write":
        return limits.WriteLimit
    case "bulk":
        return limits.BulkLimit
    default:
        return limits.ReadLimit
    }
}

func (rl *APIRateLimiter) calculateRetryAfter(opType, authLevel string) time.Duration {
    limit := rl.getLimit(opType, authLevel)
    // Time to get one token back
    return time.Duration(60/limit) * time.Second
}

func (rl *APIRateLimiter) cleanupLoop() {
    ticker := time.NewTicker(rl.cleanup)
    defer ticker.Stop()

    for range ticker.C {
        rl.cleanupInactiveClients()
    }
}

func (rl *APIRateLimiter) cleanupInactiveClients() {
    rl.mu.Lock()
    defer rl.mu.Unlock()
```

```
    cutoff := time.Now().Add(-rl.cleanup)
    for clientID, bucket := range rl.clients {
        bucket.mu.Lock()
        lastActivity := bucket.lastRefill
        bucket.mu.Unlock()

        if lastActivity.Before(cutoff) {
            delete(rl.clients, clientID)
        }
    }
}

func min(a, b int64) int64 {
    if a < b {
        return a
    }
    return b
}
```

Complete WebSocket Streaming Server (dashboard/metrics/streaming.go):

```go
package metrics

import (
    "context"
    "encoding/json"
    "log"
    "net/http"
    "sync"
    "time"

    "github.com/gorilla/websocket"
)

// StreamingServer handles real-time metric streaming to dashboard clients
type StreamingServer struct {
    clients     map[*Client]bool
    clientsMu   sync.RWMutex
    register    chan *Client
    unregister  chan *Client
    broadcast   chan []byte
    collector   *MetricsCollector
    upgrader    websocket.Upgrader
}

// Client represents a connected dashboard WebSocket client
type Client struct {
    conn      *websocket.Conn
    send      chan []byte
    filters   map[string]bool  // which metrics this client wants
    lastSeen  time.Time
}

// MetricUpdate represents a single metric update sent to clients
type MetricUpdate struct {
    Type      string                 `json:"type"`
    Timestamp time.Time              `json:"timestamp"`
    Data      map[string]interface{} `json:"data"`
}

// NewStreamingServer creates WebSocket server for real-time dashboard updates
```

```go
func NewStreamingServer(collector *MetricsCollector) *StreamingServer {
    return &StreamingServer{
        clients:    make(map[*Client]bool),
        register:   make(chan *Client),
        unregister: make(chan *Client),
        broadcast:  make(chan []byte, 256),
        collector:  collector,
        upgrader: websocket.Upgrader{
            CheckOrigin: func(r *http.Request) bool {
                // In production, implement proper origin checking
                return true
            },
        },
    }
}

// Start begins the streaming server event loop
func (s *StreamingServer) Start(ctx context.Context) {
    go s.handleClients()
    go s.streamMetrics(ctx)
}

// HandleWebSocket upgrades HTTP connections to WebSocket for dashboard streaming
func (s *StreamingServer) HandleWebSocket(w http.ResponseWriter, r *http.Request) {
    conn, err := s.upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Printf("WebSocket upgrade failed: %v", err)
        return
    }

    client := &Client{
        conn:     conn,
        send:     make(chan []byte, 256),
        filters:  make(map[string]bool),
        lastSeen: time.Now(),
    }

    s.register <- client
```

```
    // Handle client messages in separate goroutines
    go s.readFromClient(client)
    go s.writeToClient(client)
}

func (s *StreamingServer) handleClients() {
    for {
        select {
        case client := <-s.register:
            s.clientsMu.Lock()
            s.clients[client] = true
            s.clientsMu.Unlock()

            // Send initial dashboard state
            s.sendInitialState(client)

        case client := <-s.unregister:
            s.clientsMu.Lock()
            if _, ok := s.clients[client]; ok {
                delete(s.clients, client)
                close(client.send)
            }
            s.clientsMu.Unlock()

        case message := <-s.broadcast:
            s.clientsMu.RLock()
            for client := range s.clients {
                select {
                case client.send <- message:
                default:
                    // Client send channel is full, disconnect
                    delete(s.clients, client)
                    close(client.send)
                }
            }
            s.clientsMu.RUnlock()
        }
```

```
    }
}

func (s *StreamingServer) streamMetrics(ctx context.Context) {
    ticker := time.NewTicker(2 * time.Second)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return

        case <-ticker.C:
            // Collect current metrics batch
            batch := s.collector.ExportMetrics()
            if batch == nil {
                continue
            }

            // Send different update types based on metric changes
            s.broadcastQuotaUpdates(batch)
            s.broadcastTierStatus(batch)
            s.broadcastHotKeys(batch)
            s.broadcastSystemHealth(batch)
        }
    }
}

func (s *StreamingServer) broadcastQuotaUpdates(batch *MetricsBatch) {
    for ruleID, metrics := range batch.RequestMetrics {
        update := MetricUpdate{
            Type:      "quota_update",
            Timestamp: batch.Timestamp,
            Data: map[string]interface{}{
                "rule_id":   ruleID,
                "current":   metrics.AllowedRequests,
                "total":     metrics.TotalRequests,
                "denied":    metrics.DeniedRequests,
```

```
                "last_seen": metrics.LastSeen,
            },
        }

        s.broadcastUpdate(&update)
    }
}

func (s *StreamingServer) broadcastTierStatus(batch *MetricsBatch) {
    // Aggregate metrics by tier for tier status updates
    tierStats := make(map[string]map[string]interface{})

    for ruleID, metrics := range batch.RequestMetrics {
        // Extract tier from rule ID (assuming format like "user:tier_name")
        tier := extractTierFromRuleID(ruleID)

        if tierStats[tier] == nil {
            tierStats[tier] = map[string]interface{}{
                "total_requests": int64(0),
                "active_rules":   0,
                "avg_usage":      0.0,
            }
        }

        tierStats[tier]["total_requests"] = tierStats[tier]["total_requests"].(int64) + metrics.TotalRequests
        tierStats[tier]["active_rules"] = tierStats[tier]["active_rules"].(int) + 1
    }

    for tierName, stats := range tierStats {
        update := MetricUpdate{
            Type:      "tier_status",
            Timestamp: batch.Timestamp,
            Data: map[string]interface{}{
                "tier_name":      tierName,
                "total_requests": stats["total_requests"],
                "active_rules":   stats["active_rules"],
            },
        }
```

```
        s.broadcastUpdate(&update)
    }
}

func (s *StreamingServer) broadcastHotKeys(batch *MetricsBatch) {
    update := MetricUpdate{
        Type:      "hot_keys",
        Timestamp: batch.Timestamp,
        Data: map[string]interface{}{
            "hot_keys": batch.HotKeys,
        },
    }

    s.broadcastUpdate(&update)
}

func (s *StreamingServer) broadcastSystemHealth(batch *MetricsBatch) {
    update := MetricUpdate{
        Type:      "system_health",
        Timestamp: batch.Timestamp,
        Data: map[string]interface{}{
            "redis_healthy":    batch.SystemHealth != nil,
            "performance":      batch.PerformanceMetrics,
            "error_rate":       calculateErrorRate(batch),
        },
    }

    s.broadcastUpdate(&update)
}

func (s *StreamingServer) broadcastUpdate(update *MetricUpdate) {
    data, err := json.Marshal(update)
    if err != nil {
        log.Printf("Failed to marshal metric update: %v", err)
        return
    }

    select {
```

```
    case s.broadcast <- data:
    default:
        // Broadcast channel is full, drop update
        log.Printf("Dropped metric update - broadcast channel full")
    }
}

func (s *StreamingServer) readFromClient(client *Client) {
    defer func() {
        s.unregister <- client
        client.conn.Close()
    }()

    client.conn.SetReadDeadline(time.Now().Add(60 * time.Second))
    client.conn.SetPongHandler(func(string) error {
        client.conn.SetReadDeadline(time.Now().Add(60 * time.Second))
        return nil
    })

    for {
        var msg map[string]interface{}
        err := client.conn.ReadJSON(&msg)
        if err != nil {
            if websocket.IsUnexpectedCloseError(err, websocket.CloseGoingAway, websocket.CloseAbnormalClosure) {
                log.Printf("WebSocket error: %v", err)
            }
            break
        }

        // Handle client filter updates
        if msgType, ok := msg["type"].(string); ok && msgType == "set_filters" {
            if filters, ok := msg["filters"].(map[string]interface{}); ok {
                client.filters = make(map[string]bool)
                for filter, enabled := range filters {
                    if enabledBool, ok := enabled.(bool); ok {
                        client.filters[filter] = enabledBool
                    }
                }
```

```
            }
        }

        client.lastSeen = time.Now()
    }
}

func (s *StreamingServer) writeToClient(client *Client) {
    ticker := time.NewTicker(54 * time.Second)
    defer func() {
        ticker.Stop()
        client.conn.Close()
    }()

    for {
        select {
        case message, ok := <-client.send:
            client.conn.SetWriteDeadline(time.Now().Add(10 * time.Second))
            if !ok {
                client.conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            if err := client.conn.WriteMessage(websocket.TextMessage, message); err != nil {
                log.Printf("WebSocket write error: %v", err)
                return
            }

        case <-ticker.C:
            client.conn.SetWriteDeadline(time.Now().Add(10 * time.Second))
            if err := client.conn.WriteMessage(websocket.PingMessage, nil); err != nil {
                return
            }
        }
    }
}

func (s *StreamingServer) sendInitialState(client *Client) {
```

```
    // Send current dashboard state to newly connected client
    batch := s.collector.ExportMetrics()
    if batch != nil {
        s.broadcastQuotaUpdates(batch)
        s.broadcastTierStatus(batch)
        s.broadcastSystemHealth(batch)
    }
}

// Helper functions
func extractTierFromRuleID(ruleID string) string {
    // Extract tier name from rule ID format
    // This would depend on your rule ID naming convention
    if len(ruleID) > 0 && ruleID[0] == 'u' {
        return "user"
    } else if len(ruleID) > 0 && ruleID[0] == 'i' {
        return "ip"
    } else if len(ruleID) > 0 && ruleID[0] == 'a' {
        return "api"
    }
    return "global"
}

func calculateErrorRate(batch *MetricsBatch) float64 {
    if batch.PerformanceMetrics == nil {
        return 0.0
    }

    total := int64(0)
    errors := int64(0)

    for _, metrics := range batch.RequestMetrics {
        total += metrics.TotalRequests
        errors += metrics.ErrorRequests
    }

    if total == 0 {
        return 0.0
```

```
    }


    return float64(errors) / float64(total)
}
```

Core Logic Skeleton

Rule Management Handler (api/handlers/rules.go):

```go
package handlers

import (
    "encoding/json"
    "net/http"
    "strconv"
    "time"

    "github.com/gorilla/mux"
)

type RuleHandler struct {
    ruleManager *config.RuleManager
    validator   *validation.RuleValidator
    auditor     *config.AuditLogger
}

// CreateRule handles POST /api/v1/rules
func (h *RuleHandler) CreateRule(w http.ResponseWriter, r *http.Request) {
    var rule RateLimitRule

    // TODO 1: Decode JSON request body into RateLimitRule struct
    // TODO 2: Generate unique ID for new rule (use UUID or timestamp-based)
    // TODO 3: Validate rule using h.validator.ValidateRule()
    // TODO 4: Check for conflicting rules with same key_pattern and priority
    // TODO 5: Set rule creation metadata (created_at, version, etc.)
    // TODO 6: Save rule using h.ruleManager.SaveRule()
    // TODO 7: Log creation event using h.auditor.LogRuleCreation()
    // TODO 8: Trigger configuration propagation to all instances
    // TODO 9: Return created rule with 201 status code
    // Hint: Use json.NewDecoder(r.Body).Decode() for request parsing
}

// UpdateRule handles PUT /api/v1/rules/{id}
func (h *RuleHandler) UpdateRule(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID from URL path using mux.Vars(r)["id"]
    // TODO 2: Load existing rule to preserve version history
    // TODO 3: Decode JSON request body into updated RateLimitRule
    // TODO 4: Validate updated rule ensuring operational safety
```

```
    // TODO 5: Create new version entry preserving old configuration
    // TODO 6: Update rule with incremented version number
    // TODO 7: Save updated rule and publish change notification
    // TODO 8: Log update event with before/after values
    // TODO 9: Return updated rule configuration
    // Hint: Version number should increment from existing rule.version
}

// DeleteRule handles DELETE /api/v1/rules/{id}
func (h *RuleHandler) DeleteRule(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID from URL path
    // TODO 2: Load existing rule to ensure it exists
    // TODO 3: Check if rule can be safely deleted (no active dependencies)
    // TODO 4: Mark rule as deleted rather than removing (soft delete)
    // TODO 5: Clear all active rate limit counters for this rule
    // TODO 6: Publish rule deletion notification to all instances
    // TODO 7: Log deletion event with rule configuration
    // TODO 8: Return deletion confirmation
    // Hint: Soft delete preserves audit history while stopping enforcement
}

// PreviewRule handles POST /api/v1/rules/{id}/preview
func (h *RuleHandler) PreviewRule(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID and load rule configuration
    // TODO 2: Decode array of RateLimitRequest from request body
    // TODO 3: For each test request, simulate rate limiting without updating counters
    // TODO 4: Use rate limiter Preview() method to get theoretical results
    // TODO 5: Collect timing information for performance analysis
    // TODO 6: Return array of preview results with timing data
    // TODO 7: Log preview operation for audit purposes
    // Hint: Preview should never modify actual rate limit state
}

// ResetRule handles POST /api/v1/rules/{id}/reset
func (h *RuleHandler) ResetRule(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID and validate rule exists
    // TODO 2: Parse optional key filter from request body
    // TODO 3: Identify all Redis keys associated with this rule
    // TODO 4: Clear counters using rate limiter Reset() method
```

```
    // TODO 5: Handle partial failures when some keys cannot be reset
    // TODO 6: Log reset operation with affected key count
    // TODO 7: Return reset confirmation with operation summary
    // Hint: Key filter allows resetting specific users/IPs rather than all keys
}

// ListRules handles GET /api/v1/rules
func (h *RuleHandler) ListRules(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Parse query parameters for filtering and pagination
    // TODO 2: Extract filters (enabled, algorithm, priority range)
    // TODO 3: Load rules matching filter criteria
    // TODO 4: Apply sorting based on query parameters
    // TODO 5: Implement pagination using limit/offset or cursor
    // TODO 6: Calculate total count for pagination metadata
    // TODO 7: Return paginated rule list with metadata
    // Hint: Support filters like ?enabled=true&algorithm=token_bucket&priority_min=50
}

// GetRuleVersions handles GET /api/v1/rules/{id}/versions
func (h *RuleHandler) GetRuleVersions(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID from URL path
    // TODO 2: Load all historical versions of the rule
    // TODO 3: Sort versions by timestamp (newest first)
    // TODO 4: Include version metadata (created_by, created_at, change_reason)
    // TODO 5: Apply pagination if version history is large
    // TODO 6: Return version list with change summaries
    // Hint: Each version should show what changed compared to previous version
}

// RollbackRule handles POST /api/v1/rules/{id}/rollback
func (h *RuleHandler) RollbackRule(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract rule ID and target version number from request
    // TODO 2: Load target version configuration from history
    // TODO 3: Validate that rollback target exists and is valid
    // TODO 4: Create new version entry pointing to rolled-back configuration
    // TODO 5: Activate rolled-back configuration as current version
    // TODO 6: Clear any rate limit state that may be invalid after rollback
    // TODO 7: Publish configuration change notification
    // TODO 8: Log rollback operation with justification
```

```
    // TODO 9: Return confirmation with new current version
    // Hint: Rollback creates a new version rather than deleting recent versions
}
```

Rate Limit Header Injection (api/middleware/headers.go):

```go
package middleware

import (
    "fmt"
    "net/http"
    "strconv"
    "time"
)

// RateLimitHeaders middleware injects standard rate limiting headers
type RateLimitHeaders struct {
    limiter FlowCoordinator
}

// NewRateLimitHeaders creates header injection middleware
func NewRateLimitHeaders(limiter FlowCoordinator) *RateLimitHeaders {
    return &RateLimitHeaders{limiter: limiter}
}

// Middleware returns HTTP middleware that adds rate limit headers
func (rlh *RateLimitHeaders) Middleware() func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            // TODO 1: Extract request context (user_id, ip_address, api_endpoint)
            // TODO 2: Build RateLimitRequest from HTTP request data
            // TODO 3: Call limiter.Preview() to get current quota status without updating
            // TODO 4: Extract most restrictive limit from multi-tier results
            // TODO 5: Calculate standard rate limit headers (X-RateLimit-*)
            // TODO 6: Add extended headers for enhanced client experience
            // TODO 7: Handle multi-tier header aggregation following "most restrictive wins"
            // TODO 8: Add retry-after calculation for approaching limits
            // TODO 9: Set headers on response before calling next handler
            // Hint: Preview() gives quota info without consuming requests

            next.ServeHTTP(w, r)
        })
    }
}

// extractRequestContext builds rate limit context from HTTP request
```

```
func (rlh *RateLimitHeaders) extractRequestContext(r *http.Request) *RequestContext {

    // TODO 1: Extract user ID from authentication headers or JWT token

    // TODO 2: Get client IP address handling proxy headers (X-Forwarded-For)

    // TODO 3: Determine API endpoint from request path

    // TODO 4: Extract user agent and other relevant headers

    // TODO 5: Set request timestamp for consistent time-based calculations

    // TODO 6: Return populated RequestContext

    // Hint: Check X-User-ID header, Authorization header, and X-Forwarded-For

}


// calculateHeaders determines rate limit headers from flow result

func (rlh *RateLimitHeaders) calculateHeaders(result *FlowResult) map[string]string {

    // TODO 1: Find most restrictive tier from result.TierResults

    // TODO 2: Extract limit, remaining, and reset time from restrictive tier

    // TODO 3: Calculate retry-after for rate-limited requests

    // TODO 4: Build standard headers map with X-RateLimit-* entries

    // TODO 5: Add extended headers for algorithm and scope information

    // TODO 6: Handle edge case where no tiers were evaluated

    // TODO 7: Return complete headers map

    // Hint: Most restrictive = lowest remaining/limit ratio

}


// findMostRestrictiveTier identifies tier with tightest remaining quota

func (rlh *RateLimitHeaders) findMostRestrictiveTier(tiers []*TierEvaluation) *TierEvaluation {

    // TODO 1: Initialize with first tier if available

    // TODO 2: Iterate through all tier results

    // TODO 3: Calculate remaining/limit ratio for each tier

    // TODO 4: Track tier with lowest ratio (most restrictive)

    // TODO 5: Handle ties by preferring higher priority tiers

    // TODO 6: Return most restrictive tier evaluation

    // Hint: Ratio of 0.1 (10% remaining) is more restrictive than 0.8 (80% remaining)

}
```

Milestone Checkpoints

After API Implementation:

- Start management API server: `go run cmd/management-api/main.go`
- Create test rule: `curl -X POST http://localhost:8080/api/v1/rules -d '{"name":"test","key_pattern":"user:*","algorithm":"token_bucket","limit":100,"window":"1m","burst_limit":150}'`
- Expected: Rule created with generated ID and version 1
- Verify: Rule appears in Redis and triggers configuration change notification

After Dashboard Implementation:

- Start dashboard server: `go run cmd/dashboard/main.go`
- Open browser to `http://localhost:3000`
- Expected: Real-time dashboard showing quota utilization and system health
- Generate load: Run rate limiting requests and observe dashboard updates
- Verify: WebSocket connection established and metrics update every 2 seconds

After Self-Rate-Limiting Implementation:

- Test API protection: Send 200 rapid requests to management API
- Expected: First 100 requests succeed, subsequent requests receive 429 status
- Check headers: Verify X-RateLimit-* headers present in all responses
- Test emergency override: Use admin API key to bypass normal limits
- Verify: Override allows higher request rates with audit log entries

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Dashboard shows no data	WebSocket connection failing	Check browser dev tools for WebSocket errors	Verify CORS settings and WebSocket endpoint
Headers show wrong quota	Multi-tier aggregation error	Log tier results before header calculation	Fix most-restrictive-tier selection logic
Config changes not propagating	Redis pub/sub subscription broken	Check Redis logs for subscription errors	Reconnect to Redis and restart subscription
API rate limiting too aggressive	Token bucket refill rate too low	Check API rate limit configuration	Increase refill rate or burst capacity
Dashboard updates too slow	Metrics collection batching issues	Monitor metrics collector export frequency	Reduce batching interval or increase update frequency
Emergency override not working	Override token validation failing	Check API key validation and override logic	Verify override token format and expiration

Error Handling and Edge Cases

> **Milestone(s):** Milestone 3 - Redis Backend Integration, Milestone 4 - Consistent Hashing & Sharding, and foundational concepts applying to all milestones

The robustness of a distributed rate limiting system fundamentally depends on how gracefully it handles failures, edge cases, and the inevitable inconsistencies that arise in distributed environments. Unlike monolithic applications where failures are typically binary (the application works or it doesn't), distributed systems must continue operating with partial functionality when components fail. This section examines the comprehensive failure scenarios, recovery strategies, and edge case handling that transform a basic distributed rate limiter into a production-ready system.

Mental Model: The Emergency Response Network

Think of distributed rate limiting like a city's emergency response network. When you call 911, the system must route your call to the right dispatcher, coordinate with multiple emergency services, and ensure help arrives even if some communication towers are down. The emergency network has multiple levels of fallback: if the primary dispatch center fails, backup centers take over; if radio communication fails, they use cellular; if all else fails, emergency vehicles operate with their last known instructions.

Similarly, our distributed rate limiter must continue protecting your application even when Redis nodes fail, network partitions occur, or time synchronization drifts. The system degrades gracefully rather than failing completely, using local fallbacks when global coordination becomes impossible, just as emergency responders use local protocols when centralized coordination is unavailable.

Redis Failure Scenarios

Redis serves as the central nervous system of our distributed rate limiting infrastructure. When Redis becomes unavailable or unreliable, the entire system must adapt its behavior to maintain protection while avoiding cascading failures. Understanding and preparing for Redis failure modes represents

one of the most critical aspects of building resilient distributed rate limiting.

Connection Pool Exhaustion and Recovery

Connection pool exhaustion occurs when all available Redis connections become tied up in long-running operations, network timeouts, or blocked on slow Redis commands. This scenario often manifests gradually, with response times degrading before complete failure occurs.

The `RedisStorage` component implements intelligent connection pool management that monitors connection health and automatically recovers from exhaustion scenarios. When connection attempts begin timing out, the system tracks failure rates and response times to detect degradation early.

Failure Mode	Detection Method	Recovery Action	Fallback Strategy
Pool exhaustion	Connection timeout increase	Close idle connections, expand pool temporarily	Switch to local limiting
Slow Redis responses	Response time monitoring	Circuit breaker activation	Cache last known limits
Network packet loss	Retry failure patterns	TCP connection reset	Degrade to approximate limiting
Redis memory pressure	Error code analysis	Reduce TTL on keys	Local rate limiting only

The `CircuitBreaker` component provides the primary mechanism for detecting and responding to Redis degradation. It tracks success/failure ratios and response times, automatically switching to local fallback when Redis becomes unreliable.

```
Circuit Breaker State Machine:
- Closed (normal): All requests go to Redis, track failure rate
- Open (failed): All requests use local fallback, periodic health checks
- Half-Open (testing): Limited requests test Redis recovery
```

Decision: Circuit Breaker vs Retry Logic

- **Context**: Need to handle Redis failures without overwhelming failed instances
- **Options Considered**: Simple retry with exponential backoff, circuit breaker pattern, combination approach
- **Decision**: Circuit breaker with limited retries for each state
- **Rationale**: Circuit breakers prevent thundering herd problems and give failed Redis instances time to recover without constant retry pressure
- **Consequences**: More complex state management but much better failure isolation and faster recovery detection

Memory Pressure and Eviction Handling

Redis memory pressure creates particularly challenging scenarios because it can cause unpredictable key eviction, leading to rate limiting state loss. When Redis approaches its memory limit, it begins evicting keys according to its eviction policy, which may remove active rate limiting counters.

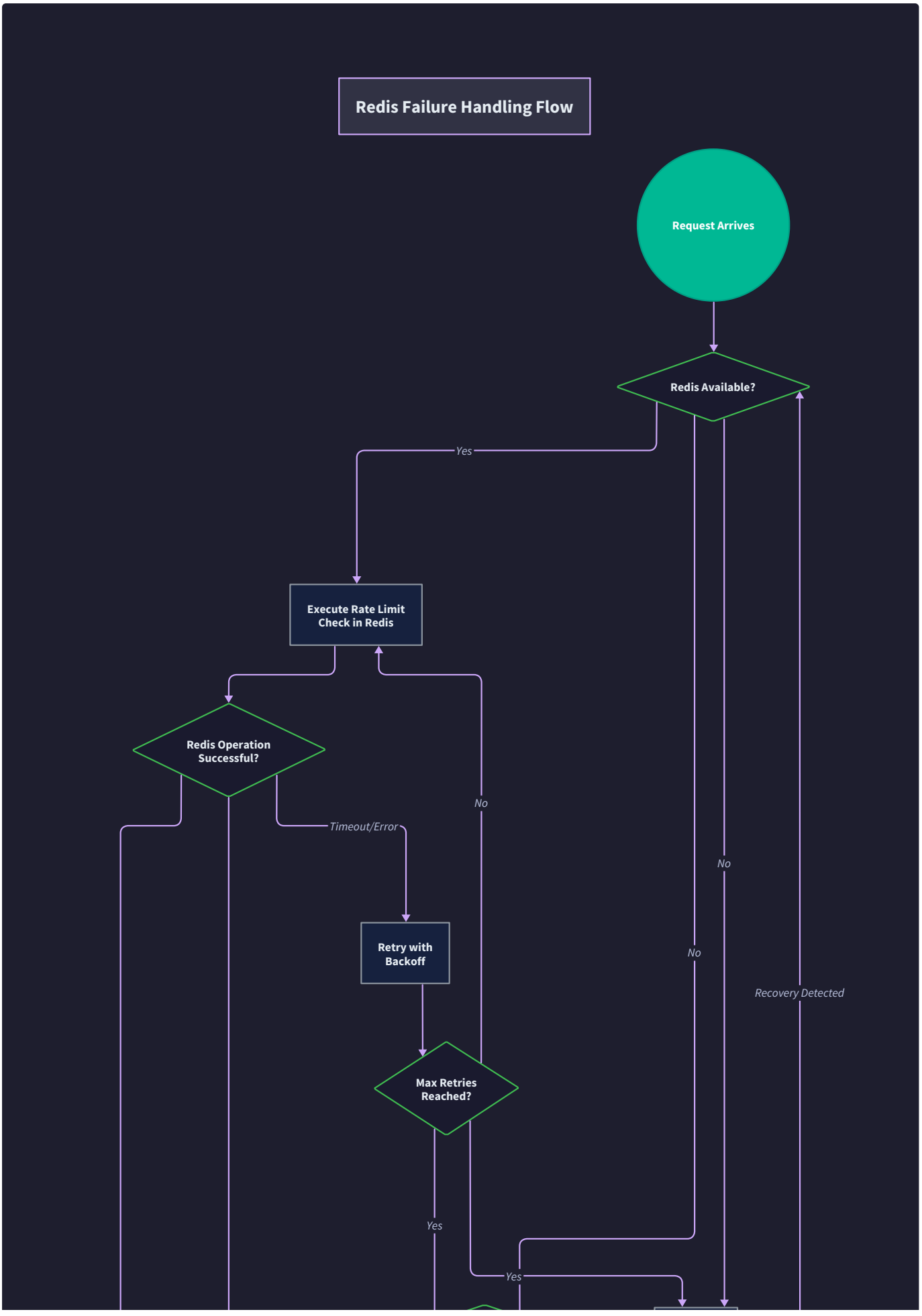
The system implements several strategies to detect and handle memory-induced state loss. The `HealthChecker` component monitors Redis memory usage through the INFO command and adjusts rate limiting behavior when memory pressure is detected.

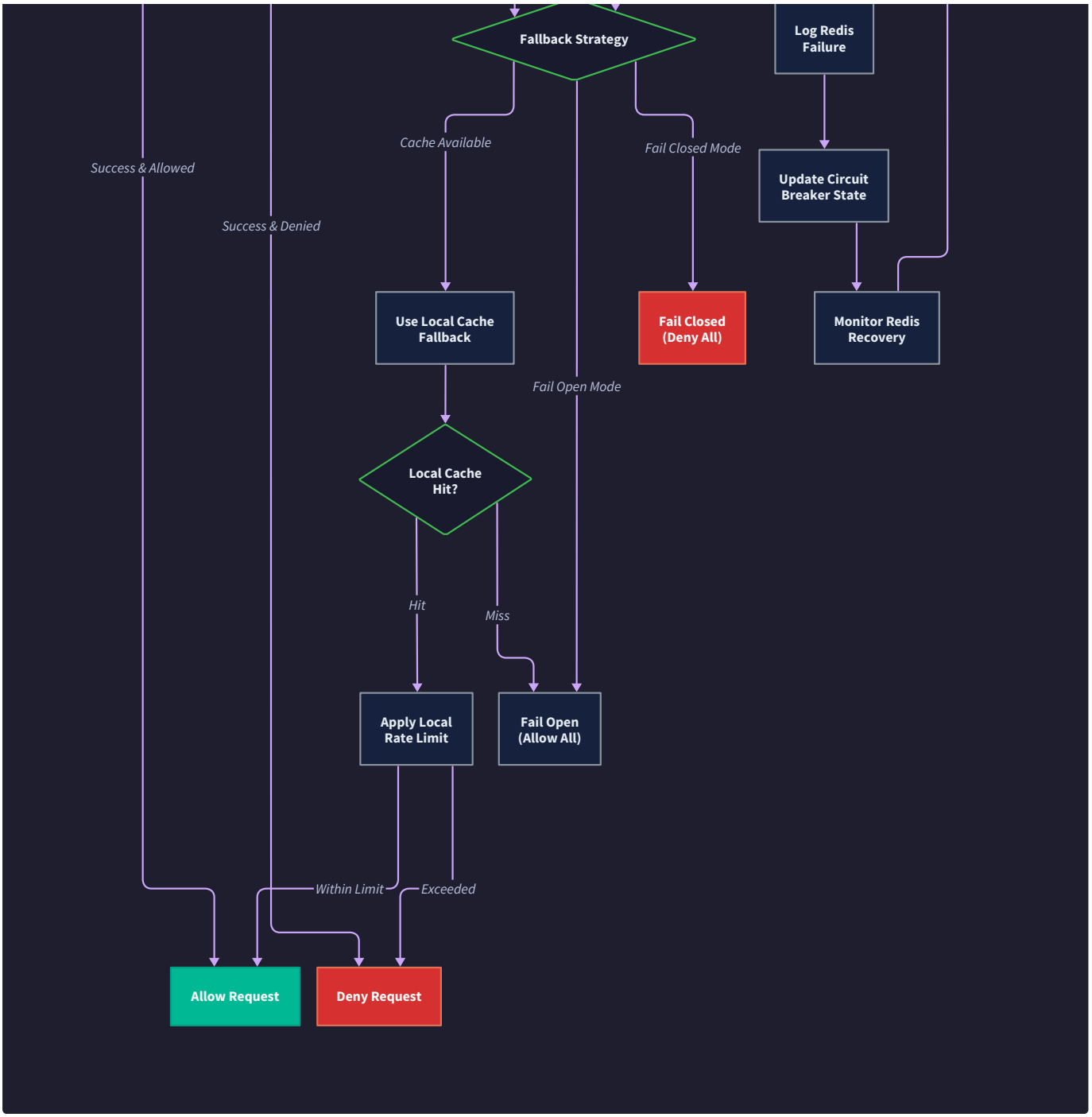
Memory Condition	Detection Threshold	Automatic Response	Operator Alert
High usage	> 80% of max memory	Reduce key TTLs by 50%	Warning notification
Critical pressure	> 95% of max memory	Switch to local fallback	Critical alert
Eviction detected	Keys disappearing unexpectedly	Reset all affected counters	Data loss warning
Out of memory	Redis refuses writes	Full local fallback mode	Emergency alert

When eviction is detected, the system faces a critical decision: should it assume evicted rate limiting keys represent exhausted quotas (deny requests) or available quotas (allow requests)? The safest approach biases toward protection, treating evicted keys as exhausted unless explicitly configured otherwise.

Split-Brain and Network Partition Handling

Network partitions can split the Redis cluster into multiple segments, each believing it represents the authoritative state. During partitions, different application instances may connect to different Redis nodes, leading to independent rate limiting decisions that violate global limits.


Redis Failure Handling Flow
Request Arrives
Redis Available?
Yes
No
Execute Rate Limit Check in Redis
Redis Operation Successful?
Timeout/Error
No
Retry with Backoff
Max Retries Reached?
Yes
No
Recovery Detected
Yes



The `HashRing` component implements partition detection by monitoring which nodes remain reachable from each application instance. When a partition is detected, the system must choose between availability and consistency.

> **Key Design Insight**: During network partitions, we prioritize application availability over strict rate limiting accuracy. It's better to allow slightly more traffic than configured limits than to reject legitimate requests due to infrastructure failures.

The partition handling algorithm follows these decision steps:

1. **Detect partition**: Monitor node reachability and cross-check with other application instances
2. **Assess majority**: Determine which partition contains the majority of Redis nodes
3. **Minority partition response**: Instances in minority partitions switch to local fallback
4. **Majority partition response**: Continue distributed limiting with reduced cluster
5. **Partition healing**: Gradually restore full distributed coordination as nodes reconnect

Partition Scenario	Instance Response	Rate Limiting Behavior	Recovery Action
Isolated single node	Switch to local fallback	Per-instance limits only	Rejoin when reachable
Minority partition	Degrade to local mode	Conservative local limits	Wait for majority contact
Majority partition	Continue distributed	Adjust for lost capacity	Gradually restore full state
Complete isolation	Full local fallback	Emergency rate limits	Manual intervention

Data Corruption and Inconsistency Detection

Redis data corruption can occur due to hardware failures, software bugs, or operational errors. Rate limiting counters may become corrupted, leading to wildly incorrect values that either completely block traffic or allow unlimited requests.

The system implements data validation within Lua scripts to detect obviously corrupted values. Rate limiting counters that show impossible values (negative tokens, timestamps from the far future, counters exceeding configured limits by orders of magnitude) trigger automatic correction.

```
Data Validation Rules:
- Token bucket tokens must be between 0 and configured capacity
- Sliding window timestamps must be within reasonable time bounds
- Counter values must not exceed limit * time_window_multiplier
- State update timestamps must be monotonically increasing
```

Clock Skew and Time Synchronization

Time represents the fundamental dimension for rate limiting, making clock synchronization critical for distributed coordination. When application instances and Redis nodes have different perceptions of current time, rate limiting windows become misaligned, leading to inaccurate quota tracking and unpredictable behavior.

Detecting and Measuring Clock Skew

Clock skew detection requires comparing timestamps between distributed components while accounting for network latency. The `TimeProvider` component implements active clock skew measurement by periodically synchronizing with Redis server time and calculating drift.

The detection algorithm works by:

1. **Record local timestamp** before Redis operation

2. **Execute Redis TIME command** to get server timestamp
3. **Record local timestamp** after Redis response
4. **Calculate network latency** as (local_after - local_before) / 2
5. **Determine skew** as redis_time - local_time - estimated_latency

Skew Magnitude	Impact	Detection Method	Mitigation Strategy
< 100ms	Negligible	Passive monitoring	No action required
100ms - 1s	Boundary inaccuracy	Active measurement	Use Redis timestamps
1s - 10s	Window misalignment	Continuous tracking	Force Redis time sync
> 10s	Severe inaccuracy	Alert generation	Manual time sync required

When significant skew is detected, the system can choose between several timestamp strategies. Using Redis server time ensures consistency but adds latency to every operation. Using local time with skew correction provides better performance but introduces complexity.

> **Decision: Redis Server Time vs Local Time Correction**
>
> - **Context**: Need consistent timestamps across distributed rate limiters with varying clock skew
> - **Options Considered**: Always use Redis TIME, local time with measured correction, hybrid approach based on skew magnitude
> - **Decision**: Hybrid approach - use local time when skew < 500ms, Redis time when skew > 500ms
> - **Rationale**: Balances accuracy requirements with performance impact; most systems have reasonable time sync
> - **Consequences**: Requires skew monitoring overhead but provides optimal performance/accuracy trade-off

Window Boundary Synchronization

Sliding window algorithms become particularly sensitive to clock skew at window boundaries. When different instances calculate window boundaries using different timestamps, the same request may fall into different time buckets, causing counter fragmentation and inaccurate limits.

The `SlidingWindowCounter` implementation addresses boundary synchronization by quantizing all timestamps to common boundary points. Instead of using precise timestamps, the system rounds timestamps to the nearest window subdivision.

```
Boundary Synchronization Algorithm:
1. Define window_size (e.g., 60 seconds) and bucket_count (e.g., 6 buckets)
2. Calculate bucket_duration = window_size / bucket_count (10 seconds)
3. Quantize timestamp: bucket_start = (timestamp / bucket_duration) * bucket_duration
4. All instances use identical bucket_start times regardless of local clock skew
```

This quantization approach ensures that all distributed instances agree on bucket boundaries, even with moderate clock skew. The trade-off is slightly less precise timestamp handling in exchange for much better distributed consistency.

Handling Timestamp Rollback

Timestamp rollback occurs when system administrators correct clock synchronization, causing time to appear to move backward. Rate limiting algorithms that depend on monotonically increasing timestamps can behave unpredictably when timestamps decrease.

The `TokenBucket` algorithm handles rollback by detecting backward time movement and choosing appropriate responses:

Rollback Magnitude	Detection Method	Response Strategy	State Adjustment
< 1 second	Compare with last_refill_time	Ignore update, use cached time	No state change
1-60 seconds	Timestamp decrease detection	Reset refill calculations	Clear accumulated tokens
> 60 seconds	Large backward jump	Full state reset	Start with empty bucket
Clock sync correction	NTP adjustment patterns	Gradual time correction	Smooth state transition

For sliding window algorithms, timestamp rollback requires careful handling to avoid double-counting requests. The system maintains a grace period where recent timestamps remain valid even if new timestamps appear earlier.

⚠️ **Pitfall: Ignoring Timestamp Rollback** Many implementations assume timestamps always increase, leading to broken rate limiting when system clocks are corrected. Always check for backward time movement and have a explicit strategy for handling it.

Race Condition Prevention

Distributed rate limiting inherently involves race conditions where multiple instances simultaneously check and update shared counters. Without proper coordination, these race conditions can lead to double-counting, lost updates, or quota violations that allow traffic beyond configured limits.

Atomic Operations with Lua Scripts

Redis Lua scripts provide the foundation for atomic rate limiting operations by ensuring that check-and-update sequences execute without interruption. The `ExecuteLua` method encapsulates complex rate limiting logic in atomic scripts that eliminate race conditions.

The token bucket Lua script demonstrates the atomic operation pattern:

```
Token Bucket Lua Script Logic:
1. GET current token count and last refill timestamp
2. Calculate tokens to add based on elapsed time and refill rate
3. Update token count (capped at bucket capacity)
4. IF requested tokens <= available tokens THEN
   5. Subtract requested tokens
   6. SET new token count and timestamp
   7. RETURN success with remaining tokens
8. ELSE
   9. SET updated token count (with refill but no consumption)
   10. RETURN failure with retry delay
```

All state reads, calculations, and writes occur atomically within the single Lua script execution. This eliminates the race condition window that would exist with separate GET, calculate, and SET operations.

Race Condition Type	Without Lua Scripts	With Lua Scripts	Prevention Method
Read-modify-write	Multiple updates lost	Single atomic update	Lua script atomicity
Check-then-act	State changes between check/act	Atomic check-and-act	Single script execution
Counter increment	Lost increment operations	Atomic counter update	Lua INCR operations
Timestamp comparison	Inconsistent time reads	Single timestamp read	Lua script locality

Optimistic vs Pessimistic Concurrency

The system implements optimistic concurrency control, assuming that conflicts are rare and handling them through retry mechanisms rather than preventing them with locks. This approach provides better performance under normal conditions while gracefully handling conflicts when they occur.

Optimistic concurrency in rate limiting works by:

1. **Read current state** without acquiring locks
2. **Calculate new state** based on read values
3. **Attempt atomic update** with conditional logic
4. **Retry on conflict** if state changed during calculation
5. **Apply exponential backoff** to prevent retry storms

The `DistributedLimiter` component implements retry logic with jitter to prevent thundering herd problems when many instances simultaneously retry failed operations.

> **Decision: Optimistic vs Pessimistic Concurrency**
>
> - **Context**: Need to handle concurrent rate limit checks across many application instances
> - **Options Considered**: Redis locks (pessimistic), optimistic retry with Lua scripts, hybrid locking for hot keys
> - **Decision**: Optimistic concurrency with exponential backoff retry
> - **Rationale**: Rate limiting operations are fast and conflicts are relatively rare; pessimistic locking adds significant overhead and can create deadlock risks
> - **Consequences**: Better performance under normal load but requires careful retry logic and backoff strategies

Hot Key Conflict Resolution

Hot keys create concentrated conflict points where many instances simultaneously attempt to update the same rate limiting counters. The `HotKeyDetector` identifies these high-contention keys and applies specialized conflict resolution strategies.

When hot keys are detected, the system can apply several conflict reduction techniques:

Hot Key Strategy	Mechanism	Advantages	Disadvantages
Key sharding	Split single key into multiple sub-keys	Reduces contention	Complex aggregation logic
Local caching	Cache counter values briefly	Fewer Redis operations	Potential accuracy loss
Retry backoff	Exponential backoff with jitter	Reduces retry storms	Higher latency for some requests
Priority queuing	Process requests by priority	Important requests succeed	Complex queue management

The key sharding approach proves most effective for extremely hot keys, where a single rate limit key is split into multiple sub-keys and requests are randomly distributed across them. The trade-off is more complex Lua scripts that must aggregate across sub-keys to check total usage.

Preventing Lost Updates in Sliding Windows

Sliding window algorithms face particular challenges with concurrent updates because multiple bucket updates may occur simultaneously as time progresses. The `SlidingWindowLog` implementation prevents lost updates by using Redis list operations that are inherently atomic.

The atomic sliding window update sequence:

1. **LPUSH new timestamp** to the request log list
2. **ZREMRANGEBYSCORE** to remove timestamps outside current window
3. **ZCARD** to count remaining timestamps in window
4. **Compare count against limit** and return result

This sequence uses Redis's atomic list and sorted set operations to ensure that concurrent updates don't interfere with each other. Each step is atomic at the Redis level, and the overall sequence runs within a Lua script for complete atomicity.

⚠️ **Pitfall: Non-Atomic Window Updates** Attempting to implement sliding windows with separate GET, filter, and SET operations creates race conditions where concurrent requests can cause lost updates or double-counting. Always use atomic operations for the complete window update sequence.

Implementation Guidance

This section provides practical implementation patterns for robust error handling and edge case management in distributed rate limiting systems.

Technology Recommendations

Component	Simple Option	Advanced Option
Time synchronization	NTP client with periodic sync	Hardware-assisted clock sync
Circuit breaker	Simple state machine	Hystrix-style metrics-based breaker
Health checking	Basic ping/pong	Comprehensive Redis INFO monitoring
Clock skew detection	Periodic Redis TIME calls	Continuous drift measurement
Retry logic	Exponential backoff	Adaptive backoff with jitter

Recommended Module Structure

```
internal/resilience/
  circuit_breaker.go          ← Circuit breaker implementation
  circuit_breaker_test.go     ← Circuit breaker unit tests
  health_checker.go           ← Redis health monitoring
  time_provider.go            ← Clock skew detection and sync
  retry_manager.go            ← Retry logic with backoff
  fallback_limiter.go         ← Local fallback rate limiting
internal/redis/
  failure_detector.go         ← Redis failure detection
  connection_pool.go          ← Connection pool management
  lua_scripts.go              ← Atomic Lua script execution
```

Circuit Breaker Infrastructure

```go
package resilience

import (
    "context"
    "sync"
    "time"
)

// CircuitState represents the current state of the circuit breaker
type CircuitState int

const (
    CircuitClosed CircuitState = iota    // Normal operation
    CircuitOpen                          // Rejecting all requests
    CircuitHalfOpen                      // Testing recovery
)

// CircuitBreaker implements failure detection and graceful degradation
type CircuitBreaker struct {
    mu               sync.RWMutex
    state            CircuitState
    failureCount     int
    lastFailureTime  time.Time
    nextRetryTime    time.Time
    failureThreshold int
    recoveryTimeout  time.Duration
}

// NewCircuitBreaker creates a circuit breaker with the specified configuration
func NewCircuitBreaker(failureThreshold int, recoveryTimeout time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
        state:            CircuitClosed,
        failureThreshold: failureThreshold,
        recoveryTimeout:  recoveryTimeout,
    }
}

// Execute runs the provided function with circuit breaker protection
func (cb *CircuitBreaker) Execute(ctx context.Context, fn func() error) error {
```

```
    // TODO 1: Check current circuit state and decide whether to allow execution
    // TODO 2: If circuit is Open and recovery timeout hasn't elapsed, return error immediately
    // TODO 3: If circuit is Open and recovery timeout has elapsed, transition to HalfOpen
    // TODO 4: Execute the function and capture any error
    // TODO 5: Update circuit state based on execution result (success/failure)
    // TODO 6: If in HalfOpen state, transition to Closed on success or Open on failure
    // TODO 7: If in Closed state, increment failure count on error and open if threshold exceeded
    // Hint: Use atomic operations or mutex for thread safety
    // Hint: Track both failure count and failure rate over time windows
    panic("implement circuit breaker execution logic")
}
```

Health Checker Implementation

```go
package resilience

import (
    "context"
    "fmt"
    "sync"
    "time"
    "github.com/redis/go-redis/v9"
)

// NodeHealth represents the health status of a single Redis node
type NodeHealth struct {
    NodeID              string
    Address             string
    LastSeen            time.Time
    ConsecutiveFailures int
    AverageLatency      time.Duration
    IsHealthy           bool
    MemoryUsage         float64
    ConnectionCount     int
}

// HealthConfig configures health checking behavior
type HealthConfig struct {
    CheckInterval      time.Duration
    FailureThreshold   int
    RecoveryThreshold  int
    LatencyThreshold   time.Duration
    MemoryThreshold    float64
}

// HealthChecker monitors Redis cluster health and triggers failover actions
type HealthChecker struct {
    mu              sync.RWMutex
    nodes           map[string]*NodeHealth
    circuitBreakers map[string]*CircuitBreaker
    config          HealthConfig
}
```

```go
// CheckNodeHealth performs comprehensive health check on a Redis node
func (hc *HealthChecker) CheckNodeHealth(ctx context.Context, nodeID string) (*NodeHealth, error) {
    // TODO 1: Create Redis client for the specified node
    // TODO 2: Execute PING command and measure response time
    // TODO 3: Execute INFO command to get memory usage and connection count
    // TODO 4: Parse INFO response to extract memory_used, memory_total, connected_clients
    // TODO 5: Calculate memory usage percentage and compare against threshold
    // TODO 6: Update NodeHealth struct with current metrics
    // TODO 7: Determine overall health status based on latency, memory, and recent failures
    // TODO 8: Update consecutive failure count based on current check result
    // Hint: Use context timeout to avoid hanging on unresponsive nodes
    // Hint: Parse INFO command output to extract specific metrics
    panic("implement comprehensive Redis node health checking")
}

// HandleNodeFailure coordinates response to a detected node failure
func (hc *HealthChecker) HandleNodeFailure(nodeID string) error {
    // TODO 1: Mark node as unhealthy in the health map
    // TODO 2: Trigger circuit breaker for this node to prevent further requests
    // TODO 3: Notify hash ring to remove node from active rotation
    // TODO 4: Log failure event with detailed diagnostic information
    // TODO 5: Schedule accelerated health checks to detect recovery
    // TODO 6: Update metrics to track node failure for monitoring dashboard
    // Hint: Coordinate with HashRing component to redistribute load
    // Hint: Implement exponential backoff for recovery health checks
    panic("implement node failure response coordination")
}
```

Time Synchronization and Clock Skew Detection

```go
package resilience

import (
    "context"
    "math"
    "sync"
    "time"
    "github.com/redis/go-redis/v9"
)

// TimeProvider handles clock synchronization and skew detection
type TimeProvider struct {
    mu            sync.RWMutex
    redisClient   redis.UniversalClient
    clockSkew     time.Duration
    lastSync      time.Time
    syncInterval  time.Duration
}

// NewTimeProvider creates a time provider with Redis synchronization
func NewTimeProvider(redisClient redis.UniversalClient, syncInterval time.Duration) *TimeProvider {
    return &TimeProvider{
        redisClient:  redisClient,
        syncInterval: syncInterval,
    }
}

// Now returns the current time adjusted for measured clock skew
func (tp *TimeProvider) Now() time.Time {
    // TODO 1: Get current local time
    // TODO 2: Check if clock skew measurement is recent enough (within sync interval)
    // TODO 3: If skew measurement is stale, trigger background sync (don't block)
    // TODO 4: Apply measured clock skew correction to local time
    // TODO 5: Return corrected timestamp
    // Hint: Don't block on sync - return local time if Redis is unavailable
    // Hint: Use atomic operations for reading clockSkew to avoid races
    panic("implement skew-corrected timestamp generation")
}
```

```go
// MeasureClockSkew compares local time with Redis server time to detect skew
func (tp *TimeProvider) MeasureClockSkew(ctx context.Context) (time.Duration, error) {
    // TODO 1: Record local timestamp before Redis call
    // TODO 2: Execute Redis TIME command to get server timestamp
    // TODO 3: Record local timestamp after Redis response
    // TODO 4: Parse Redis TIME response (seconds, microseconds)
    // TODO 5: Calculate network latency as (local_after - local_before) / 2
    // TODO 6: Calculate clock skew as redis_time - local_time - estimated_latency
    // TODO 7: Update internal skew tracking with new measurement
    // TODO 8: Return measured skew value
    // Hint: Redis TIME returns [seconds, microseconds] array
    // Hint: Account for network latency in skew calculation
    panic("implement clock skew measurement against Redis server time")
}
```

Local Fallback Rate Limiter

```go
package resilience

import (
    "context"
    "sync"
    "time"
)

// FallbackLimiter provides local rate limiting when Redis is unavailable
type FallbackLimiter struct {
    mu              sync.RWMutex
    tokenBuckets    map[string]*TokenBucketState
    windowCounters  map[string]*WindowCounterState
    defaultConfig   *TokenBucketConfig
    cleanupInterval time.Duration
}

// TokenBucketState tracks local token bucket state
type TokenBucketState struct {
    tokens          int64
    lastRefillTime  int64
    capacity        int64
    refillRate      int64
}

// WindowCounterState tracks local sliding window state
type WindowCounterState struct {
    requests    []time.Time
    windowSize  time.Duration
    limit       int64
}

// NewFallbackLimiter creates a local fallback rate limiter
func NewFallbackLimiter(defaultConfig *TokenBucketConfig) *FallbackLimiter {
    return &FallbackLimiter{
        tokenBuckets:    make(map[string]*TokenBucketState),
        windowCounters:  make(map[string]*WindowCounterState),
        defaultConfig:   defaultConfig,
        cleanupInterval: time.Minute * 5,
```

```
    }
}

// CheckLocal performs rate limiting using local state only
func (fl *FallbackLimiter) CheckLocal(ctx context.Context, key string, tokens int64, rule *RateLimitRule) (*RateLimitResult, error) {
    // TODO 1: Determine which algorithm to use based on rule configuration
    // TODO 2: Get or create local state for the specified key
    // TODO 3: Apply token bucket algorithm using local timestamps
    // TODO 4: Calculate refill tokens based on elapsed time since last access
    // TODO 5: Check if requested tokens are available in bucket
    // TODO 6: Update local state if tokens are consumed
    // TODO 7: Calculate retry_after time if request is denied
    // TODO 8: Return RateLimitResult with local state information
    // Hint: Use local time only - don't depend on Redis for timestamps
    // Hint: Implement cleanup to prevent memory leaks from abandoned keys
    panic("implement local fallback rate limiting")
}
```

Milestone Checkpoints

Milestone 3 Checkpoint - Redis Failure Handling: After implementing circuit breakers and local fallback:

```bash
# Test Redis failure scenarios
go test ./internal/resilience/... -v
go test ./internal/redis/... -v

# Manual testing:
# 1. Start rate limiter with Redis backend
# 2. Send requests and verify normal operation
# 3. Stop Redis instance
# 4. Verify automatic fallback to local limiting
# 5. Restart Redis and verify recovery
```

Expected behavior: Rate limiting continues during Redis outage with degraded accuracy but maintained protection. Recovery should be automatic and seamless.

Milestone 4 Checkpoint - Clock Skew and Sharding: After implementing time synchronization and hot key handling:

```bash
# Test clock skew scenarios
go test ./internal/resilience/... -run TestClockSkew
go test ./internal/sharding/... -run TestHotKey

# Manual testing:
# 1. Run multiple instances with deliberately skewed clocks
# 2. Verify rate limiting accuracy despite time differences
# 3. Generate hot key traffic and verify conflict resolution
```

Expected behavior: Rate limiting accuracy within 5% even with moderate clock skew. Hot key detection should trigger within 30 seconds of elevated traffic.

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Rate limits too strict during Redis outage	Local fallback using wrong limits	Check fallback configuration and rule inheritance	Configure appropriate local fallback limits
Inconsistent rate limiting across instances	Clock skew between nodes	Compare timestamps in logs, measure skew	Implement NTP sync and skew correction
Circuit breaker stuck open	Recovery timeout too long or health checks failing	Check Redis connectivity and error patterns	Tune recovery timeout and failure thresholds
Rate limiting fails completely during network partition	No fallback strategy implemented	Check for local limiting during Redis failures	Implement and test graceful degradation
Hot keys causing Redis CPU spikes	Too many concurrent updates to same keys	Monitor Redis slow log and key access patterns	Implement key sharding or request batching

Testing Strategy

> **Milestone(s):** All milestones - testing strategies span rate limiting algorithms (Milestone 1), multi-tier evaluation (Milestone 2), Redis backend integration (Milestone 3), sharding and consistent hashing (Milestone 4), and API/dashboard functionality (Milestone 5)

Mental Model: The Quality Assurance Laboratory System

Think of testing a distributed rate limiter like running a comprehensive quality assurance laboratory for a pharmaceutical company. Just as a pharmaceutical lab has multiple testing phases - from testing individual compounds in isolation (unit testing), to testing drug interactions in controlled environments (integration testing), to testing complete treatment protocols in clinical trials (end-to-end testing), and finally to stress-testing drugs under extreme conditions (chaos testing) - our distributed rate limiter requires a multi-layered testing approach.

The individual algorithms are like chemical compounds that must be tested for purity and effectiveness in isolation. The distributed system interactions are like drug combinations that must be tested for dangerous interactions. The milestone verification checkpoints are like clinical trial phases that must be passed before advancing to more complex testing. And chaos engineering is like testing medications under extreme stress conditions to ensure they remain safe and effective when patients are critically ill.

This analogy helps us understand why each testing layer is essential - a drug that works perfectly in isolation might fail catastrophically when combined with other medications or under stress conditions, just as rate limiting algorithms that work perfectly locally might fail when distributed across multiple nodes or under high load.

Algorithm Unit Testing

The foundation of testing distributed rate limiting systems begins with rigorous verification of individual algorithms in complete isolation. Each rate limiting algorithm represents a mathematical model with precise behavioral expectations, and unit testing must validate both the happy path scenarios and the critical boundary conditions where algorithms typically break down.

Token Bucket Algorithm Testing Strategy

Token bucket testing requires careful verification of the refill mechanism, burst handling, and state transitions. The algorithm maintains an internal state consisting of current token count, last refill timestamp, capacity, and refill rate. Each of these state components must be tested independently and in combination.

Test Category	Test Case	Expected Behavior	Boundary Condition
Refill Logic	Initial state after creation	Bucket starts at full capacity	Time t=0 initialization
Refill Logic	Refill after exact window duration	Tokens = min(capacity, current + refill_rate)	Exact window boundary
Refill Logic	Partial refill with fractional time	Proportional token addition	Sub-second precision
Burst Handling	Request exactly at capacity	All requests allowed instantly	Full bucket depletion
Burst Handling	Request exceeding capacity	Excess requests denied	Overflow protection
State Persistence	Token count after partial consumption	Accurate remaining count	Fractional tokens
Concurrent Access	Multiple simultaneous requests	Atomic state updates	Race condition prevention
Time Manipulation	Clock moving backwards	Graceful degradation	Clock skew scenarios

The token bucket algorithm testing must simulate time progression artificially to verify refill behavior without waiting for real time to elapse. This requires dependency injection of a `TimeProvider` interface that can be controlled during testing.

Sliding Window Counter Algorithm Testing Strategy

Sliding window counter testing focuses on bucket management, weight calculation accuracy, and window transition handling. The algorithm divides time into discrete buckets and maintains counters for each bucket, requiring precise weight calculation when the current time falls between bucket boundaries.

Test Category	Test Case	Expected Behavior	Critical Edge Case
Bucket Creation	First request in new window	Initialize bucket with count=1	Empty state initialization
Bucket Transitions	Request at exact window boundary	Proper bucket rotation	Precision timing
Weight Calculation	Request mid-bucket	Accurate weighted sum	Fractional weights
Window Sliding	Old bucket expiration	Automatic cleanup	Memory management
Count Aggregation	Multiple buckets active	Correct total calculation	Boundary spanning
Precision Testing	Sub-second request timing	Nanosecond accuracy	High-frequency requests
Memory Pressure	Long-running operation	Bounded memory usage	Garbage collection

Sliding Window Log Algorithm Testing Strategy

Sliding window log testing requires validation of timestamp storage, log rotation, and memory management. Unlike counter-based approaches, the log algorithm stores individual request timestamps, making memory usage testing critical.

Test Category	Test Case	Expected Behavior	Memory Consideration
Log Insertion	First timestamp storage	Accurate timestamp recording	Initial allocation
Log Rotation	Timestamp expiration	Automatic log cleanup	Memory deallocation
Binary Search	Threshold calculation	Efficient timestamp lookup	O(log n) complexity
Memory Bounds	High request volume	Controlled memory growth	Maximum log size
Timestamp Precision	Nanosecond accuracy	Precise timing storage	Clock resolution
Concurrent Logging	Multiple simultaneous requests	Thread-safe log updates	Lock contention

Algorithm Comparison and Performance Testing

Beyond individual algorithm testing, comprehensive benchmarks must compare algorithm performance characteristics under various load patterns. This comparison testing reveals the trade-offs between accuracy, memory usage, and computational overhead.

Algorithm	Memory Complexity	Time Complexity	Accuracy	Burst Handling
Token Bucket	O(1) constant	O(1) per request	Exact for bursts	Excellent
Sliding Window Counter	O(buckets)	O(1) per request	Approximate	Good
Sliding Window Log	O(requests in window)	O(log n) per request	Exact	Excellent

> **Design Insight**: Algorithm testing must validate not just correctness but also performance characteristics under load. A token bucket that works correctly for 10 requests per second might fail catastrophically at 10,000 requests per second due to lock contention or memory allocation overhead.

Boundary Condition Testing Matrix

Rate limiting algorithms face their greatest challenges at boundary conditions where time windows transition, numeric limits approach maximum values, or system resources become constrained. Comprehensive boundary testing prevents production failures.

Boundary Type	Test Scenario	Algorithm Impact	Validation Method
Time Window Edges	Request at exact window start	Counter reset timing	Millisecond precision testing
Time Window Edges	Request at exact window end	Bucket rotation	Boundary value analysis
Numeric Limits	Request count approaching int64 max	Overflow protection	Large value injection
Numeric Limits	Token count exceeding capacity	Burst limitation	Boundary value testing
System Resources	Memory pressure during operation	Graceful degradation	Resource constraint simulation
Clock Anomalies	System clock adjustment	Time skew handling	Clock manipulation testing

Distributed Integration Testing

Moving beyond individual algorithm testing, distributed integration testing validates the behavior of multiple rate limiter instances coordinating through Redis. This testing phase reveals issues that only emerge when multiple processes compete for shared resources and coordinate state changes.

Multi-Instance Coordination Testing

The core challenge in distributed rate limiting lies in ensuring accurate limit enforcement when multiple application instances simultaneously evaluate and update shared counters. Integration testing must simulate realistic deployment scenarios with multiple instances processing concurrent requests.

The test environment requires multiple rate limiter instances configured to use the same Redis backend, with careful orchestration of request timing to trigger race conditions and coordination challenges. Each test scenario must verify that distributed behavior matches single-instance behavior within acceptable accuracy bounds.

Test Scenario	Instance Count	Request Pattern	Expected Outcome	Accuracy Threshold
Simultaneous Burst	3 instances	100 requests/instance instantly	Total allowed ≤ limit + burst	5% tolerance
Gradual Ramp-up	5 instances	Linear increase over 60 seconds	Smooth limit enforcement	2% tolerance
Coordinated Attack	10 instances	Synchronized request waves	Consistent denial pattern	1% tolerance
Mixed Load Patterns	8 instances	Random request timing	Fair resource allocation	Statistical validation

Redis Atomic Operation Testing

The correctness of distributed rate limiting fundamentally depends on atomic Redis operations implemented through Lua scripts. Integration testing must verify that these atomic operations maintain consistency under concurrent load from multiple instances.

Testing atomic operations requires careful coordination of multiple Redis clients executing operations simultaneously and validating that the final state matches expectations regardless of execution order or timing.

Lua Script	Concurrent Operations	State Validation	Atomicity Check
Token Bucket Check-and-Update	50 simultaneous calls	Token count accuracy	No double-counting
Sliding Window Increment	100 concurrent increments	Counter consistency	No lost updates
Multi-Tier Evaluation	25 instances checking	Tier precedence maintained	Correct short-circuiting
Hot Key Detection	High-frequency access	Access count accuracy	No race conditions

Network Partition Simulation

Distributed systems must handle network partitions gracefully, falling back to local rate limiting when Redis becomes unavailable. Integration testing must simulate various network failure scenarios and verify that fallback mechanisms activate correctly and recovery proceeds smoothly.

Network partition testing requires controlled network manipulation to simulate Redis connectivity issues while maintaining test orchestration communication. The testing framework must distinguish between intentional network partitions and actual infrastructure failures.

Partition Scenario	Duration	Expected Behavior	Recovery Validation
Complete Redis Loss	30 seconds	Local fallback activation	Seamless transition back
Intermittent Connectivity	5-second cycles	Circuit breaker activation	Adaptive retry behavior
Partial Node Failure	60 seconds	Hash ring rebalancing	Minimal key redistribution
Split-Brain Scenario	Variable timing	Consistent state maintenance	No conflicting decisions

Load Distribution and Sharding Testing

When using consistent hashing to distribute rate limiting state across multiple Redis nodes, integration testing must verify that load distributes evenly and that node additions or removals cause minimal disruption. This testing requires careful monitoring of key distribution patterns and performance characteristics.

Sharding Test	Node Count	Key Distribution	Rebalancing Trigger	Success Criteria
Initial Distribution	5 nodes	1000 unique keys	N/A	±10% load variance
Node Addition	5→6 nodes	Existing keys	New node insertion	<20% key migration
Node Removal	6→5 nodes	Existing keys	Node failure	<25% key migration
Hot Key Handling	5 nodes	80/20 access pattern	Automatic detection	Even response times

Milestone Verification Checkpoints

Each milestone in the distributed rate limiter development process requires specific verification checkpoints that confirm the implementation meets acceptance criteria before proceeding to more complex functionality. These checkpoints serve as quality gates preventing the accumulation of technical debt and ensuring solid foundations for subsequent development.

Milestone 1: Rate Limiting Algorithms Verification

The first milestone checkpoint focuses on verifying that individual rate limiting algorithms function correctly in isolation, handling edge cases appropriately, and meeting performance requirements under load.

Algorithm Component	Verification Test	Expected Result	Performance Benchmark
Token Bucket Implementation	Burst handling test: 1000 requests instantly	≤ capacity requests allowed	< 1μs per operation
Token Bucket Implementation	Refill rate test: sustained load over 60 seconds	Average rate = refill_rate	Consistent latency
Sliding Window Counter	Boundary transition test: requests spanning window	Smooth rate enforcement	< 500ns per operation
Sliding Window Log	Memory usage test: 1M requests over 1 hour	Bounded memory growth	Linear memory scaling
Algorithm Comparison	Accuracy benchmark: compare with exact counting	Token bucket: exact, Counter: ~95%	Measure overhead

The verification process requires implementing a comprehensive test suite that exercises each algorithm through multiple scenarios, measuring both correctness and performance characteristics. The test suite must generate synthetic load patterns that simulate real-world usage while maintaining

precise control over timing and request distribution.

```bash
# Example checkpoint verification commands

go test ./pkg/algorithms/token_bucket/ -v -race -count=100

go test ./pkg/algorithms/sliding_window/ -v -bench=. -benchmem

go test ./pkg/algorithms/comparison/ -v -timeout=300s
```

Milestone 2: Multi-Tier Rate Limiting Verification

The second milestone verification ensures that hierarchical rate limiting works correctly with proper tier precedence, short-circuit evaluation, and accurate rule matching across user, IP, API, and global dimensions.

Multi-Tier Component	Verification Test	Expected Behavior	Edge Case Coverage
Tier Precedence	User limit < IP limit conflict	User limit enforced	Most restrictive wins
Short-Circuit Evaluation	Global limit reached first	Skip remaining tiers	Efficiency optimization
Rule Pattern Matching	Complex regex patterns	Correct rule selection	Pattern performance
Quota Management	Cross-tier usage tracking	Accurate attribution	No double-counting

The multi-tier verification process requires creating test scenarios with conflicting rate limits across different tiers and verifying that the system enforces the most restrictive applicable limit while maintaining performance efficiency.

Milestone 3: Redis Backend Integration Verification

The third milestone checkpoint validates that Redis integration maintains atomic operations, handles connection failures gracefully, and provides accurate distributed state synchronization across multiple application instances.

Redis Component	Verification Test	Expected Outcome	Failure Scenario
Lua Script Atomicity	100 concurrent check-and-update	No race conditions	Network latency
Connection Pooling	1000 simultaneous operations	Connection reuse	Pool exhaustion
Graceful Degradation	Redis unavailable	Local fallback	Service continuity
State Synchronization	Multi-instance coordination	Consistent limits	Clock skew tolerance

Redis integration verification requires deploying multiple rate limiter instances against a shared Redis backend and orchestrating load tests that verify distributed coordination accuracy under various failure conditions.

Milestone 4: Consistent Hashing and Sharding Verification

The fourth milestone verification confirms that consistent hashing distributes load evenly, minimizes key redistribution during topology changes, and handles hot key scenarios appropriately.

Sharding Component	Verification Test	Success Criteria	Scalability Metric
Hash Ring Distribution	10,000 keys across 5 nodes	±15% load variance	O(log n) lookup time
Node Addition Impact	Add 6th node to running system	<20% key migration	Minimal disruption
Hot Key Detection	80/20 access pattern	Automatic identification	Response time consistency
Failover Behavior	Primary node failure	Seamless redirection	<100ms recovery time

Sharding verification requires comprehensive load testing with realistic key distribution patterns and careful measurement of performance characteristics during topology changes.

Milestone 5: API and Dashboard Verification

The final milestone verification ensures that the management API functions correctly, the dashboard displays accurate real-time data, and rate limit headers conform to RFC standards.

API Component	Verification Test	Expected Result	Integration Check
CRUD Operations	Create, read, update, delete rules	Immediate effect	Configuration propagation
Rate Limit Headers	X-RateLimit-* header presence	RFC compliance	Client compatibility
Real-time Dashboard	Live usage metrics	<5 second latency	WebSocket stability
Dynamic Configuration	Rule updates without restart	Zero downtime	Gradual rollout

Chaos and Failure Testing

The robustness of a distributed rate limiter emerges most clearly under adverse conditions where components fail, networks partition, and system resources become constrained. Chaos engineering principles guide the systematic introduction of failures to validate recovery mechanisms and identify weaknesses in distributed coordination.

Redis Failure Scenario Testing

Redis failures represent the most critical failure mode for distributed rate limiting, as they eliminate the shared state coordination mechanism. Comprehensive failure testing must cover various Redis failure patterns and validate fallback behavior.

The chaos testing framework must simulate realistic Redis failure scenarios including complete node failures, memory pressure leading to evictions, network partitions isolating Redis clusters, and performance degradation under load. Each failure scenario requires careful validation of both immediate response and recovery behavior.

Failure Mode	Simulation Method	Expected Behavior	Recovery Validation
Complete Redis Outage	Network blackhole	Local fallback activation	Automatic reconnection
Memory Pressure	Maxmemory + allkeys-lru	Key eviction tolerance	Performance degradation
Network Partition	Selective packet dropping	Circuit breaker activation	Split-brain prevention
Performance Degradation	Artificial latency injection	Timeout handling	Graceful degradation
Partial Node Failure	Single node termination	Hash ring rebalancing	Key redistribution

Network Partition and Split-Brain Prevention

Network partitions represent particularly challenging failure scenarios where different parts of the distributed system lose connectivity with each other while maintaining connectivity with clients. The rate limiting system must handle these partitions without creating inconsistent states or allowing unlimited request flow.

Network partition testing requires sophisticated network manipulation tools that can selectively block communication between specific components while maintaining test orchestration channels. The testing framework must validate that partitioned components make consistent decisions and that recovery procedures restore global consistency.

Partition Scenario	Components Affected	Decision Strategy	Consistency Check
Redis Cluster Split	Half of Redis nodes	Majority quorum	No split decisions
Application Island	Some app instances isolated	Conservative limiting	Err on restrictive side
Cross-Region Partition	Geographic separation	Regional degradation	Eventual consistency
Cascading Failure	Multiple component failures	Progressive degradation	Bounded blast radius

High Load and Resource Exhaustion Testing

Distributed rate limiters must maintain accuracy and performance under extreme load conditions that stress both computational resources and network bandwidth. High load testing reveals performance bottlenecks, memory leaks, and coordination inefficiencies that only manifest under sustained pressure.

Resource exhaustion testing requires careful load generation that can saturate different system resources independently - CPU usage through computational overhead, memory usage through state accumulation, network bandwidth through request volume, and disk I/O through logging and persistence operations.

Resource Constraint	Load Pattern	System Response	Performance Metric
CPU Saturation	High-frequency requests	Graceful degradation	Latency percentiles
Memory Pressure	Many unique keys	Memory management	GC frequency
Network Bandwidth	Large request payloads	Request prioritization	Throughput maintenance
Redis Connection Pool	Concurrent operations	Connection queuing	Pool utilization

Clock Skew and Time Synchronization Testing

Distributed rate limiting algorithms depend critically on synchronized time across all participating nodes. Clock skew between application instances and Redis nodes can lead to inaccurate rate limit calculations, particularly for time-window-based algorithms.

Clock skew testing requires artificially manipulating system clocks on different nodes and measuring the impact on rate limiting accuracy. The testing framework must validate that clock skew detection mechanisms function correctly and that synchronization procedures restore accuracy.

Clock Scenario	Skew Magnitude	Algorithm Impact	Mitigation Effectiveness
Gradual Drift	±1 second over 1 hour	Sliding window accuracy	NTP synchronization
Sudden Jump	+30 seconds instantly	Token bucket disruption	Skew detection
Backwards Clock	-10 seconds	Timestamp confusion	Monotonic time usage
Different Zones	Cross-timezone deployment	UTC coordination	Timezone normalization

Cascading Failure Prevention Testing

The most dangerous failure scenarios in distributed systems involve cascading failures where the failure of one component triggers failures in dependent components, potentially leading to total system collapse. Rate limiting systems must include circuit breakers and bulkheads to prevent failure propagation.

Cascading failure testing requires orchestrated failure injection across multiple system layers while monitoring the blast radius and recovery characteristics. The testing framework must validate that circuit breakers activate appropriately and that fallback mechanisms prevent total system failure.

Cascade Trigger	Initial Failure	Propagation Path	Circuit Breaker Response
Redis Overload	High latency	Client timeout cascade	Redis circuit opening
App Instance Death	Process crash	Load redistribution	Health check removal
Network Congestion	Packet loss	Retry amplification	Backpressure activation
Dashboard Query Load	Metrics overload	Rate limiter impact	Self-rate-limiting

> **Critical Testing Insight**: Chaos testing must be continuous rather than periodic. Production systems face constant low-level failures and stress conditions that only become apparent through sustained chaos injection. A rate limiting system that passes a one-hour chaos test might fail catastrophically after running under minor stress for weeks.

Implementation Guidance

This implementation guidance provides concrete tools and templates for building a comprehensive testing strategy that validates distributed rate limiting functionality across all development milestones.

A. Technology Recommendations Table:

Testing Component	Simple Option	Advanced Option
Unit Testing	Go testing package + testify	Ginkgo BDD framework
Load Testing	Custom goroutines	Apache JMeter / k6
Redis Simulation	Embedded Redis (miniredis)	Docker Compose Redis cluster
Network Partition	Manual connection dropping	Chaos Mesh / Litmus
Metrics Collection	Simple counters	Prometheus + Grafana
Test Orchestration	Shell scripts	Kubernetes Jobs
Clock Manipulation	time.Now() mocking	libfaketime system-wide

B. Recommended File/Module Structure

```
project-root/
  cmd/
    chaos-test/main.go              ← Chaos engineering test runner
    integration-test/main.go        ← Multi-instance integration tests
    benchmark/main.go               ← Performance benchmark suite

  test/
    unit/                           ← Algorithm unit tests
      token_bucket_test.go
      sliding_window_test.go
      algorithms_benchmark_test.go

    integration/                    ← Distributed coordination tests
      multi_instance_test.go
      redis_atomic_test.go
      failover_test.go

    chaos/                          ← Failure injection tests
      redis_failure_test.go
      network_partition_test.go
      load_stress_test.go

    fixtures/                       ← Test data and configurations
      rate_limit_rules.yaml
      redis_cluster_config.yaml

    helpers/                        ← Testing utilities
      redis_helper.go              ← Redis test setup/teardown
      load_generator.go            ← Traffic generation utilities
      time_helper.go               ← Clock manipulation helpers
      metrics_collector.go         ← Test metrics collection

  pkg/
    ratelimit/
      *_test.go                    ← Unit tests alongside implementation

  scripts/
    run_milestone_checks.sh        ← Automated milestone verification
    setup_test_environment.sh     ← Test infrastructure setup
    chaos_test_suite.sh           ← Comprehensive chaos testing
```

C. Infrastructure Starter Code (COMPLETE, ready to use):

Redis Test Helper - Complete Implementation:

```go
// test/helpers/redis_helper.go

package helpers

import (
    "context"
    "fmt"
    "time"

    "github.com/alicebob/miniredis/v2"
    "github.com/redis/go-redis/v9"
    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

// RedisTestHelper manages Redis instances for testing
type RedisTestHelper struct {
    // For unit tests - embedded Redis
    miniRedis *miniredis.Miniredis

    // For integration tests - real Redis containers
    containers map[string]testcontainers.Container
    clients    map[string]*redis.Client
}

// NewRedisTestHelper creates a new Redis test helper
func NewRedisTestHelper() *RedisTestHelper {
    return &RedisTestHelper{
        containers: make(map[string]testcontainers.Container),
        clients:    make(map[string]*redis.Client),
    }
}

// StartEmbeddedRedis starts a lightweight Redis for unit tests
func (h *RedisTestHelper) StartEmbeddedRedis() (*redis.Client, func(), error) {
    mini, err := miniredis.Run()
    if err != nil {
        return nil, nil, fmt.Errorf("failed to start miniredis: %w", err)
    }
```

```
    h.miniRedis = mini

    client := redis.NewClient(&redis.Options{
        Addr: mini.Addr(),
        DB:   0,
    })

    cleanup := func() {
        client.Close()
        mini.Close()
    }

    return client, cleanup, nil
}

// StartRedisCluster starts a Redis cluster for integration tests
func (h *RedisTestHelper) StartRedisCluster(ctx context.Context, nodeCount int) ([]*redis.Client, func(), error) {
    var clients []*redis.Client
    var cleanupFuncs []func()

    for i := 0; i < nodeCount; i++ {
        container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
            ContainerRequest: testcontainers.ContainerRequest{
                Image:        "redis:7-alpine",
                ExposedPorts: []string{"6379/tcp"},
                WaitingFor:   wait.ForLog("Ready to accept connections"),
                Cmd:          []string{"redis-server", "--appendonly", "yes"},
            },
            Started: true,
        })
        if err != nil {
            return nil, func() { h.cleanup(cleanupFuncs) }, err
        }

        endpoint, err := container.Endpoint(ctx, "")
        if err != nil {
            return nil, func() { h.cleanup(cleanupFuncs) }, err
        }
```

```
        client := redis.NewClient(&redis.Options{
            Addr: endpoint,
        })

        clients = append(clients, client)
        h.containers[fmt.Sprintf("node%d", i)] = container
        h.clients[fmt.Sprintf("node%d", i)] = client

        cleanupFuncs = append(cleanupFuncs, func() {
            client.Close()
            container.Terminate(ctx)
        })
    }

    return clients, func() { h.cleanup(cleanupFuncs) }, nil
}

// SimulateNetworkPartition blocks network access to specific Redis nodes
func (h *RedisTestHelper) SimulateNetworkPartition(ctx context.Context, nodeIDs []string) error {
    for _, nodeID := range nodeIDs {
        if container, exists := h.containers[nodeID]; exists {
            // Simulate network partition by pausing the container
            err := container.Terminate(ctx)
            if err != nil {
                return fmt.Errorf("failed to partition node %s: %w", nodeID, err)
            }
        }
    }
    return nil
}

func (h *RedisTestHelper) cleanup(cleanupFuncs []func()) {
    for _, cleanup := range cleanupFuncs {
        cleanup()
    }
}
```

Load Generator - Complete Implementation:

```go
// test/helpers/load_generator.go

package helpers

import (
    "context"
    "math/rand"
    "sync"
    "sync/atomic"
    "time"

    "your-project/pkg/ratelimit"
)

// LoadPattern defines different traffic generation patterns
type LoadPattern int

const (
    ConstantLoad LoadPattern = iota
    BurstLoad
    GradualRamp
    SpikeLoad
    RandomLoad
)

// LoadGeneratorConfig configures load generation parameters
type LoadGeneratorConfig struct {
    Pattern          LoadPattern
    RequestsPerSecond int
    Duration         time.Duration
    ClientCount      int
    KeyPattern       string
    BurstSize        int
    BurstInterval    time.Duration
}

// LoadGenerator generates synthetic traffic patterns for testing
type LoadGenerator struct {
    config    LoadGeneratorConfig
    limiter   ratelimit.DistributedLimiter
    results   *LoadTestResults
```

```
    ctx       context.Context
    cancelFn  context.CancelFunc
}

// LoadTestResults captures comprehensive load test metrics
type LoadTestResults struct {
    TotalRequests    int64
    AllowedRequests  int64
    DeniedRequests   int64
    ErrorRequests    int64
    AverageLatency   time.Duration
    P95Latency       time.Duration
    P99Latency       time.Duration
    StartTime        time.Time
    EndTime          time.Time
    LatencyHistogram []time.Duration
    mutex            sync.RWMutex
}

// NewLoadGenerator creates a load generator with specified configuration
func NewLoadGenerator(config LoadGeneratorConfig, limiter ratelimit.DistributedLimiter) *LoadGenerator {
    ctx, cancel := context.WithTimeout(context.Background(), config.Duration)

    return &LoadGenerator{
        config:  config,
        limiter: limiter,
        results: &LoadTestResults{
            StartTime:        time.Now(),
            LatencyHistogram: make([]time.Duration, 0, config.RequestsPerSecond*int(config.Duration.Seconds())),
        },
        ctx:      ctx,
        cancelFn: cancel,
    }
}

// Start begins load generation according to configured pattern
func (lg *LoadGenerator) Start() *LoadTestResults {
    defer lg.cancelFn()
    lg.results.StartTime = time.Now()
```

```
    switch lg.config.Pattern {
    case ConstantLoad:
        lg.generateConstantLoad()
    case BurstLoad:
        lg.generateBurstLoad()
    case GradualRamp:
        lg.generateGradualRamp()
    case SpikeLoad:
        lg.generateSpikeLoad()
    case RandomLoad:
        lg.generateRandomLoad()
    }

    lg.results.EndTime = time.Now()
    lg.calculateStatistics()
    return lg.results
}

// generateConstantLoad produces steady request rate
func (lg *LoadGenerator) generateConstantLoad() {
    interval := time.Second / time.Duration(lg.config.RequestsPerSecond)
    ticker := time.NewTicker(interval)
    defer ticker.Stop()

    var wg sync.WaitGroup

    for {
        select {
        case <-lg.ctx.Done():
            wg.Wait()
            return
        case <-ticker.C:
            for i := 0; i < lg.config.ClientCount; i++ {
                wg.Add(1)
                go func() {
                    defer wg.Done()
                    lg.executeRequest()
```

```
                }()
            }
        }
    }
}

// generateBurstLoad produces periodic bursts of requests
func (lg *LoadGenerator) generateBurstLoad() {
    burstTicker := time.NewTicker(lg.config.BurstInterval)
    defer burstTicker.Stop()

    for {
        select {
        case <-lg.ctx.Done():
            return
        case <-burstTicker.C:
            var wg sync.WaitGroup
            for i := 0; i < lg.config.BurstSize; i++ {
                wg.Add(1)
                go func() {
                    defer wg.Done()
                    lg.executeRequest()
                }()
            }
            wg.Wait()
        }
    }
}

// executeRequest performs a single rate limit check with timing
func (lg *LoadGenerator) executeRequest() {
    start := time.Now()

    key := lg.generateKey()
    req := ratelimit.RateLimitRequest{
        UserID:      fmt.Sprintf("user_%d", rand.Intn(1000)),
        IPAddress:   fmt.Sprintf("192.168.1.%d", rand.Intn(255)+1),
        APIEndpoint: key,
```

```
        Tokens:      1,
    }

    result, err := lg.limiter.Check(lg.ctx, req)
    latency := time.Since(start)

    lg.recordResult(result, err, latency)
}

// recordResult safely updates load test results
func (lg *LoadGenerator) recordResult(result *ratelimit.RateLimitResult, err error, latency time.Duration) {
    lg.results.mutex.Lock()
    defer lg.results.mutex.Unlock()

    atomic.AddInt64(&lg.results.TotalRequests, 1)
    lg.results.LatencyHistogram = append(lg.results.LatencyHistogram, latency)

    if err != nil {
        atomic.AddInt64(&lg.results.ErrorRequests, 1)
    } else if result.Allowed {
        atomic.AddInt64(&lg.results.AllowedRequests, 1)
    } else {
        atomic.AddInt64(&lg.results.DeniedRequests, 1)
    }
}

// generateKey creates test keys according to pattern
func (lg *LoadGenerator) generateKey() string {
    if lg.config.KeyPattern == "" {
        return fmt.Sprintf("api_endpoint_%d", rand.Intn(10))
    }
    return fmt.Sprintf(lg.config.KeyPattern, rand.Intn(100))
}

// calculateStatistics computes latency percentiles and averages
func (lg *LoadGenerator) calculateStatistics() {
    if len(lg.results.LatencyHistogram) == 0 {
        return
    }
```

```
    // Sort latencies for percentile calculation
    latencies := make([]time.Duration, len(lg.results.LatencyHistogram))
    copy(latencies, lg.results.LatencyHistogram)

    sort.Slice(latencies, func(i, j int) bool {
        return latencies[i] < latencies[j]
    })

    // Calculate average
    var total time.Duration
    for _, latency := range latencies {
        total += latency
    }
    lg.results.AverageLatency = total / time.Duration(len(latencies))

    // Calculate percentiles
    p95Index := int(0.95 * float64(len(latencies)))
    p99Index := int(0.99 * float64(len(latencies)))

    lg.results.P95Latency = latencies[p95Index]
    lg.results.P99Latency = latencies[p99Index]
}
```

Time Helper for Clock Manipulation - Complete Implementation:

```go
// test/helpers/time_helper.go

package helpers

import (
    "sync"
    "time"
)

// MockTimeProvider implements controllable time for testing
type MockTimeProvider struct {
    mu          sync.RWMutex
    currentTime time.Time
    clockSkew   time.Duration
}

// NewMockTimeProvider creates a controllable time provider
func NewMockTimeProvider(startTime time.Time) *MockTimeProvider {
    return &MockTimeProvider{
        currentTime: startTime,
    }
}

// Now returns the current mock time
func (m *MockTimeProvider) Now() time.Time {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return m.currentTime.Add(m.clockSkew)
}

// AdvanceTime moves the mock clock forward
func (m *MockTimeProvider) AdvanceTime(duration time.Duration) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.currentTime = m.currentTime.Add(duration)
}

// SetClockSkew simulates time difference between nodes
func (m *MockTimeProvider) SetClockSkew(skew time.Duration) {
    m.mu.Lock()
    defer m.mu.Unlock()
```

```
    m.clockSkew = skew
}

// SetTime directly sets the current time
func (m *MockTimeProvider) SetTime(t time.Time) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.currentTime = t
}
```

D. Core Logic Skeleton Code (signature + TODOs only):

Algorithm Unit Test Template:

```go
// test/unit/token_bucket_test.go

package unit

import (
    "context"
    "testing"
    "time"

    "github.com/stretchr/testify/assert"
    "your-project/pkg/algorithms"
    "your-project/test/helpers"
)

// TestTokenBucketBurstHandling validates burst capacity behavior
func TestTokenBucketBurstHandling(t *testing.T) {
    // TODO 1: Create TokenBucket with capacity=10, refill_rate=5/sec
    // TODO 2: Execute 10 requests instantly - all should be allowed
    // TODO 3: Execute 11th request - should be denied
    // TODO 4: Wait 1 second for refill
    // TODO 5: Execute 5 more requests - should be allowed
    // TODO 6: Verify remaining token count matches expected value

    config := algorithms.TokenBucketConfig{
        Capacity:   10,
        RefillRate: 5,
        Window:     time.Second,
    }

    // Implementation goes here - learner fills this in
}

// TestTokenBucketRefillAccuracy validates precise refill timing
func TestTokenBucketRefillAccuracy(t *testing.T) {
    // TODO 1: Create TokenBucket with capacity=100, refill_rate=10/sec
    // TODO 2: Drain bucket completely (100 requests)
    // TODO 3: Advance time by 0.5 seconds
    // TODO 4: Verify exactly 5 tokens available
    // TODO 5: Advance time by another 0.5 seconds
    // TODO 6: Verify exactly 10 tokens available
```

```
    // TODO 7: Test fractional refills (0.1 second = 1 token)


    timeProvider := helpers.NewMockTimeProvider(time.Now())


    // Implementation goes here - learner fills this in
}

// TestSlidingWindowBoundaryTransition validates accurate window transitions
func TestSlidingWindowBoundaryTransition(t *testing.T) {
    // TODO 1: Create SlidingWindowCounter with limit=100, window=1min, buckets=60
    // TODO 2: Fill first 30 seconds with 50 requests
    // TODO 3: Fill next 30 seconds with 50 requests
    // TODO 4: Advance to 45 seconds - verify current count = 75 (weighted)
    // TODO 5: Advance to 60 seconds - verify count = 50 (second half only)
    // TODO 6: Test precision at exact second boundaries


    // Implementation goes here - learner fills this in
}

// BenchmarkAlgorithmComparison compares performance characteristics
func BenchmarkAlgorithmComparison(b *testing.B) {
    // TODO 1: Set up TokenBucket, SlidingWindowCounter, SlidingWindowLog
    // TODO 2: Benchmark each algorithm with same request pattern
    // TODO 3: Measure operations per second for each
    // TODO 4: Measure memory allocation per operation
    // TODO 5: Record results for comparison table


    // Implementation goes here - learner fills this in
}
```

Integration Test Template:

```go
// test/integration/multi_instance_test.go

package integration

import (
    "context"
    "sync"
    "testing"
    "time"

    "your-project/pkg/ratelimit"
    "your-project/test/helpers"
)

// TestDistributedCoordination validates multi-instance rate limiting accuracy
func TestDistributedCoordination(t *testing.T) {
    // TODO 1: Start Redis cluster with 3 nodes
    // TODO 2: Create 5 DistributedLimiter instances connected to same Redis
    // TODO 3: Configure rate limit: 100 requests/minute per API endpoint
    // TODO 4: Generate 500 requests across all instances simultaneously
    // TODO 5: Verify total allowed requests ≤ 105 (5% tolerance for race conditions)
    // TODO 6: Wait for next window and verify reset behavior
    // TODO 7: Test with different request timing patterns

    redisHelper := helpers.NewRedisTestHelper()
    // Implementation goes here - learner fills this in
}

// TestAtomicOperations validates Redis Lua script atomicity
func TestAtomicOperations(t *testing.T) {
    // TODO 1: Set up Redis with custom Lua scripts loaded
    // TODO 2: Launch 100 goroutines executing check-and-update simultaneously
    // TODO 3: Each goroutine attempts to consume 1 token from bucket with capacity=50
    // TODO 4: Verify exactly 50 operations succeed, 50 fail
    // TODO 5: Verify final bucket state is consistent (0 tokens remaining)
    // TODO 6: Test with different algorithms and concurrency levels

    // Implementation goes here - learner fills this in
}
```

```
// TestGracefulDegradation validates fallback behavior during Redis failures

func TestGracefulDegradation(t *testing.T) {

    // TODO 1: Start rate limiter with Redis backend + local fallback

    // TODO 2: Verify normal operation with Redis available

    // TODO 3: Simulate Redis failure (network partition)

    // TODO 4: Verify automatic fallback to local limiting

    // TODO 5: Verify degraded functionality still enforces limits

    // TODO 6: Restore Redis and verify transition back to distributed mode

    // TODO 7: Test various failure scenarios (partial failures, timeouts)



    // Implementation goes here - learner fills this in

}
```

Chaos Test Template:

```go
// test/chaos/redis_failure_test.go

package chaos

import (
    "context"
    "testing"
    "time"

    "your-project/test/helpers"
)

// TestRedisCompleteOutage validates behavior during total Redis failure
func TestRedisCompleteOutage(t *testing.T) {
    // TODO 1: Deploy 3 rate limiter instances with Redis cluster backend
    // TODO 2: Start load generation: 1000 req/sec across all instances
    // TODO 3: After 30 seconds, simulate complete Redis cluster failure
    // TODO 4: Verify circuit breaker activates within 5 seconds
    // TODO 5: Verify local fallback maintains some rate limiting (degraded)
    // TODO 6: After 60 seconds, restore Redis cluster
    // TODO 7: Verify recovery to normal operation within 30 seconds
    // TODO 8: Measure accuracy degradation during outage period

    // Implementation goes here - learner fills this in
}

// TestNetworkPartitionSplitBrain validates split-brain prevention
func TestNetworkPartitionSplitBrain(t *testing.T) {
    // TODO 1: Set up 6-node Redis cluster + 4 rate limiter instances
    // TODO 2: Partition network: isolate 2 Redis nodes + 2 app instances
    // TODO 3: Generate traffic to both network partitions
    // TODO 4: Verify minority partition fails safely (conservative limiting)
    // TODO 5: Verify majority partition continues normal operation
    // TODO 6: Heal network partition after 120 seconds
    // TODO 7: Verify convergence to consistent state
    // TODO 8: Measure request accuracy during partition

    // Implementation goes here - learner fills this in
}
```

```
// TestClockSkewImpact validates time synchronization requirements
func TestClockSkewImpact(t *testing.T) {
    // TODO 1: Deploy rate limiters with controllable time providers
    // TODO 2: Introduce gradual clock skew: +1 second per minute drift
    // TODO 3: Measure sliding window accuracy degradation over time
    // TODO 4: Test sudden clock jumps: +30 seconds, -10 seconds
    // TODO 5: Verify skew detection mechanisms activate
    // TODO 6: Test algorithm-specific impacts (token bucket vs sliding window)
    // TODO 7: Validate time sync recovery procedures

    // Implementation goes here - learner fills this in
}
```

E. Language-Specific Hints:

- Use `testing.TB` interface for helpers that work in both tests and benchmarks
- Leverage `testify/suite` for complex test setup/teardown with shared state
- Use `context.WithTimeout` extensively to prevent hanging integration tests
- Implement test helpers that use `t.Cleanup()` for automatic resource cleanup
- Use `sync/atomic` for thread-safe counter operations in load tests
- Leverage `testing.Short()` to skip expensive tests during development
- Use `go test -race` consistently to detect race conditions
- Implement test fixtures with `embed` for configuration files
- Use `httptest.Server` for testing HTTP API endpoints
- Leverage `testcontainers-go` for realistic Redis cluster testing

F. Milestone Checkpoint:

After implementing each milestone's testing strategy:

Milestone 1 Checkpoint:

```bash
# Verify algorithm implementations
go test ./pkg/algorithms/... -v -race -count=5
go test ./pkg/algorithms/... -bench=. -benchmem
go test ./test/unit/... -v -timeout=60s

# Expected: All tests pass, benchmarks show expected performance characteristics
# Signs of problems: Race conditions, memory leaks, inconsistent results
```

Milestone 2 Checkpoint:

```bash
# Verify multi-tier coordination
go test ./pkg/multitier/... -v -race
go test ./test/integration/multi_tier_test.go -v

# Expected: Proper tier precedence, short-circuit evaluation working
# Signs of problems: Wrong limits enforced, performance degradation
```

Milestone 3 Checkpoint:

```bash
# Verify Redis integration
go test ./test/integration/redis_test.go -v
docker-compose -f test/redis-cluster.yml up -d
go test ./test/integration/... -v -timeout=300s

# Expected: Atomic operations, graceful degradation, connection pooling
# Signs of problems: Race conditions, connection leaks, fallback failures
```

G. Debugging Tips:

Symptom	Likely Cause	Diagnosis Method	Fix
Tests pass individually but fail in parallel	Race conditions in shared state	Run with `-race` flag	Add proper synchronization
Integration tests hang indefinitely	Missing context timeouts	Check for blocking operations	Add context.WithTimeout
High memory usage during load tests	Memory leaks in test harness	Use `go test -memprofile`	Fix cleanup in test helpers
Inconsistent rate limiting accuracy	Clock skew or timing issues	Log timestamps and compare	Implement time synchronization
Redis connection errors in tests	Connection pool exhaustion	Monitor Redis connection count	Increase pool size or add cleanup
Chaos tests produce false positives	Insufficient failure simulation time	Extend chaos duration	Allow more time for failure detection

Debugging Guide

> **Milestone(s):** All milestones - debugging techniques span rate limiting algorithms (Milestone 1), multi-tier evaluation (Milestone 2), Redis backend integration (Milestone 3), consistent hashing and sharding (Milestone 4), and API design (Milestone 5).

Mental Model: The Detective Investigation Framework

Think of debugging a distributed rate limiter like conducting a complex criminal investigation across multiple crime scenes. Just as a detective must gather evidence from multiple locations, interview witnesses with different perspectives, and piece together a timeline of events, debugging distributed systems requires collecting logs from multiple nodes, understanding different component viewpoints, and reconstructing the sequence of operations that led to a problem.

The detective starts with observable symptoms (the "crime scene") - perhaps requests are being incorrectly allowed or denied, response times are slow, or counters seem inaccurate. Like fingerprints and DNA evidence, distributed systems leave traces in logs, metrics, and Redis state that tell the story of what happened. The challenge lies in correlating evidence across time zones (clock skew), dealing with unreliable witnesses (network partitions), and understanding that the timeline might be non-linear (asynchronous operations).

A good detective follows a systematic approach: secure the scene (gather current state), collect evidence (logs and metrics), interview witnesses (check all nodes), analyze the timeline (sequence of operations), and test theories (reproduce the issue). Similarly, effective distributed system debugging requires structured approaches that account for the inherent complexity of multiple moving parts operating across network boundaries.

Symptom-Based Diagnosis Table

The foundation of effective distributed rate limiting debugging lies in mapping observable symptoms to their likely root causes through systematic analysis. This diagnostic approach transforms the overwhelming complexity of distributed system failures into manageable investigation paths.

Symptom	Observable Behavior	Likely Root Causes	Initial Diagnostic Steps	Advanced Investigation
Requests incorrectly allowed	Rate limit counters show usage below threshold but actual requests exceed configured limit	Token bucket refill race condition, sliding window boundary condition, clock skew between nodes, Lua script logic error	Check Redis TIME command vs local time, examine Lua script execution logs, verify algorithm parameters	Enable request tracing with correlation IDs, compare counter values across Redis nodes, analyze time-series data for boundary transitions
Requests incorrectly denied	Rate limit counters exceed threshold but usage appears normal, legitimate traffic rejected	Hot key concentration, Redis memory pressure, circuit breaker false positives, multi-tier evaluation precedence error	Monitor Redis memory usage and eviction stats, check circuit breaker state, verify rule priority ordering	Analyze key distribution across hash ring, examine tier evaluation logs, test rule pattern matching logic
Inconsistent behavior across instances	Same request allowed on one app instance but denied on another	Local fallback active on some nodes, Redis connection issues, configuration propagation delay, stale rule cache	Check Redis connectivity per node, verify configuration version across instances, examine local fallback activation logs	Trace configuration update propagation, validate hash ring consistency, monitor network partitions
High latency in rate limit checks	Response times exceed acceptable thresholds consistently	Redis cluster rebalancing, network latency to Redis, connection pool exhaustion, Lua script complexity	Monitor Redis connection pool metrics, measure network round-trip time, profile Lua script execution duration	Analyze request routing patterns, examine hot key impact on performance, trace connection lifecycle
Rate limit counters reset unexpectedly	Usage statistics drop to zero without administrative action	Redis key expiration misconfiguration, manual counter reset, Redis failover with data loss, time provider synchronization issue	Check Redis TTL settings, review administrative action logs, verify Redis persistence configuration	Analyze Redis cluster topology changes, examine backup and recovery procedures, validate time synchronization
Memory usage growing unbounded	Redis memory consumption increases without corresponding traffic	Sliding window log accumulation, connection leak, metrics collection retention, key namespace pollution	Monitor Redis key count and memory per key, check connection pool statistics, analyze key patterns	Profile memory allocation patterns, examine garbage collection behavior, trace key lifecycle management
Circuit breaker stuck open	All Redis requests rejected despite Redis being healthy	Failure threshold too sensitive, health check misconfiguration, recovery timeout too long, cascading failure detection	Verify Redis health via direct connection, check circuit breaker configuration parameters, review failure classification logic	Analyze failure pattern leading to circuit opening, validate health check implementation, examine recovery criteria
Configuration changes not applied	Rule updates don't affect rate limiting behavior	Redis pub/sub channel issues, configuration watcher stopped, rule validation failure, cache invalidation problem	Check Redis pub/sub subscription status, verify configuration watcher process, examine rule validation logs	Trace configuration update flow end-to-end, validate cache coherency mechanisms, analyze version control
Clock skew affecting time windows	Rate limiting behavior varies with server time differences	NTP synchronization issues, Redis server time drift, timezone configuration mismatch, manual time adjustment	Compare local time with Redis TIME command, check NTP status on all nodes, verify timezone settings	Implement distributed time synchronization monitoring, analyze time drift patterns, validate time provider logic
Hot key performance degradation	Specific keys experience severe latency while others perform normally	Uneven key distribution, single Redis instance overload, hash ring imbalance, algorithmic complexity	Identify hot keys through Redis monitoring, analyze request distribution patterns, check hash ring virtual node allocation	Implement hot key replication, analyze access pattern evolution, optimize key distribution strategy

Redis-Specific Debugging Techniques

Redis serves as the critical shared state store for distributed rate limiting, making Redis-specific debugging techniques essential for diagnosing and resolving issues. The challenge lies in Redis's single-threaded nature combined with distributed access patterns creating complex interactions between atomicity, performance, and consistency.

Redis CLI Diagnostic Commands

The Redis command-line interface provides powerful introspection capabilities for understanding rate limiting state and system health. These commands form the foundation of Redis debugging workflows.

Command Category	Command	Purpose	Example Usage	Interpretation
Time Synchronization	`TIME`	Get Redis server timestamp	`redis-cli TIME`	Compare with local time to detect clock skew exceeding 100ms
Memory Analysis	`MEMORY USAGE key`	Memory consumption per rate limit key	`MEMORY USAGE ratelimit:user:12345:api:/orders`	Identify memory-intensive keys causing Redis pressure
Key Inspection	`TTL key`	Remaining time-to-live for rate limit counters	`TTL ratelimit:user:12345:token_bucket`	Verify expiration timing matches configured window duration
Script Debugging	`EVAL script numkeys key [arg ...]`	Execute Lua script with debugging output	`EVAL "return {KEYS[1], ARGV[1], redis.call('TIME')}" 1 test_key 100`	Test Lua script logic with controlled inputs
Connection Monitoring	`CLIENT LIST`	Active connection details	`CLIENT LIST TYPE normal`	Identify connection leaks and pool exhaustion
Performance Profiling	`SLOWLOG GET count`	Recent slow operations	`SLOWLOG GET 10`	Detect rate limiting operations exceeding performance thresholds
Key Pattern Analysis	`SCAN cursor MATCH pattern COUNT count`	Iterate through rate limit keys	`SCAN 0 MATCH ratelimit:user:* COUNT 1000`	Analyze key distribution and naming patterns
Memory Pressure	`INFO memory`	Comprehensive memory statistics	`INFO memory`	Monitor memory usage, fragmentation, and eviction policies
Persistence Status	`LASTSAVE`	Last successful save timestamp	`LASTSAVE`	Verify data persistence for rate limit state recovery
Cluster Health	`CLUSTER NODES`	Cluster topology and node status	`CLUSTER NODES`	Diagnose Redis cluster partitions and failover status

Lua Script Debugging Techniques

Rate limiting algorithms rely heavily on atomic Lua scripts for maintaining consistency. Debugging these scripts requires specialized approaches that account for Redis's execution environment constraints.

The primary challenge in Lua script debugging lies in Redis's atomic execution model - scripts run to completion without interruption, making traditional debugging techniques ineffective. Instead, debugging relies on strategic logging, return value analysis, and controlled test environments.

```lua
-- Example debugging-enabled token bucket script with comprehensive logging
local key = KEYS[1]
local capacity = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])
local requested_tokens = tonumber(ARGV[3])
local window_seconds = tonumber(ARGV[4])

-- Debug: Create execution trace for diagnostics
local debug_info = {}
debug_info.timestamp = redis.call('TIME')
debug_info.key = key
debug_info.inputs = {capacity, refill_rate, requested_tokens, window_seconds}

-- Get current state with error handling
local current_state = redis.call('HMGET', key, 'tokens', 'last_refill')
local tokens = tonumber(current_state[1]) or capacity
local last_refill = tonumber(current_state[2]) or tonumber(debug_info.timestamp[1])

debug_info.initial_state = {tokens, last_refill}

-- Calculate refill amount with precision handling
local current_time = tonumber(debug_info.timestamp[1])
local time_delta = math.max(0, current_time - last_refill)
local refill_amount = math.min(capacity - tokens, time_delta * refill_rate / window_seconds)
local new_tokens = math.min(capacity, tokens + refill_amount)

debug_info.refill_calculation = {time_delta, refill_amount, new_tokens}

-- Make rate limiting decision
local allowed = new_tokens >= requested_tokens
local final_tokens = allowed and (new_tokens - requested_tokens) or new_tokens

debug_info.decision = {allowed, final_tokens}

-- Update state atomically
redis.call('HMSET', key, 'tokens', final_tokens, 'last_refill', current_time)
redis.call('EXPIRE', key, window_seconds * 2)

-- Return decision with comprehensive debugging information
return {
    allowed and 1 or 0,
    final_tokens,
```

```
    capacity - final_tokens, -- remaining capacity
    debug_info.timestamp[1],
    cjson.encode(debug_info) -- serialized debug information
}
```

Redis Monitoring and Alerting Setup

Effective Redis monitoring for distributed rate limiting requires tracking both standard Redis metrics and rate limiting specific indicators. The monitoring system must detect performance degradation before it impacts user experience while providing sufficient detail for root cause analysis.

Metric Category	Key Metrics	Alert Thresholds	Diagnostic Value
Memory Management	`used_memory_rss`, `mem_fragmentation_ratio`	Memory usage > 80%, fragmentation > 2.0	Indicates approaching capacity limits requiring scaling
Rate Limiting Performance	Lua script execution time, key access patterns	Script duration > 5ms, hot key concentration > 10x average	Identifies algorithmic inefficiencies and traffic patterns
Connection Health	`connected_clients`, `blocked_clients`	Connections > 80% of maxclients, blocked clients > 0	Reveals connection pool exhaustion and blocking operations
Persistence Status	`last_save_time`, `changes_since_save`	No save in 10 minutes, unsaved changes > 10000	Ensures rate limit state durability for recovery scenarios
Cluster Coordination	Node availability, failover events	Node down > 30 seconds, frequent failovers	Detects cluster instability affecting distributed consensus

Distributed System Debugging

Debugging distributed rate limiting systems requires sophisticated approaches that account for the fundamental challenges of multiple independent processes coordinating across network boundaries. Unlike single-node debugging where state is directly observable, distributed debugging demands correlation techniques that piece together a coherent picture from fragments scattered across multiple machines.

Correlation IDs and Request Tracing

The foundation of distributed debugging lies in correlation IDs - unique identifiers that follow requests through their entire journey across multiple system components. In distributed rate limiting, a single user request might trigger checks against multiple tiers, require Redis operations across several nodes, and involve fallback logic during failures.

The `RequestContext` structure provides the framework for comprehensive request tracing throughout the rate limiting pipeline:

Field	Type	Tracing Purpose	Example Value
`UserID`	`string`	Links requests to specific users across tiers	`user_12345`
`IPAddress`	`string`	Enables IP-based correlation and geo-analysis	`192.168.1.100`
`APIEndpoint`	`string`	Groups requests by functionality for pattern analysis	`/api/v1/orders`
`Headers`	`map[string]string`	Contains correlation IDs and request metadata	`{"X-Trace-ID": "trace_abc123", "X-Request-ID": "req_xyz789"}`
`Timestamp`	`time.Time`	Enables temporal correlation across nodes with different clocks	`2023-10-15T14:30:45.123Z`

Distributed Tracing Implementation Strategy

Effective distributed tracing for rate limiting requires capturing decision points, performance bottlenecks, and error conditions at each system boundary. The tracing strategy must balance observability needs with performance overhead, particularly given rate limiting's position in the critical request path.

Decision: Structured Logging with Correlation IDs

- **Context**: Rate limiting checks occur on every API request, making tracing overhead critical to system performance
- **Options Considered**: Full distributed tracing (Jaeger/Zipkin), structured logging with correlation IDs, custom event streaming
- **Decision**: Structured logging with correlation IDs and sampling for detailed traces
- **Rationale**: Provides necessary correlation capabilities without the performance overhead and complexity of full distributed tracing systems
- **Consequences**: Enables request flow reconstruction while maintaining sub-millisecond overhead, but requires manual correlation for complex failure scenarios

The `FlowCoordinator` serves as the central orchestration point where distributed tracing context is established and propagated:

```go
type FlowCoordinator struct {
    storage         Storage
    ruleManager     *RuleManager
    keyComposer     *KeyComposer
    algorithms      map[string]Algorithm
    localFallback   Limiter
    metricsCollector *MetricsCollector
    circuitBreaker  *CircuitBreaker
}
```

Multi-Node Log Aggregation Patterns

Distributed rate limiting debugging requires aggregating logs from multiple application instances, Redis nodes, and monitoring systems into a coherent timeline. The challenge lies in correlating events across systems with different clock synchronization, log formats, and retention policies.

Log Source	Information Provided	Correlation Keys	Retention Requirements
Application Instances	Rate limit decisions, tier evaluations, algorithm execution	Correlation ID, user ID, timestamp, instance ID	7 days for debugging, 30 days for pattern analysis
Redis Cluster	Lua script execution, key access patterns, cluster topology changes	Redis key, operation timestamp, node ID	3 days for performance analysis, longer for capacity planning
Load Balancer	Request routing, instance health, traffic distribution	Request ID, backend instance, response codes	24 hours for routing analysis
Monitoring Systems	Performance metrics, alert triggers, system health indicators	Metric timestamps, alert correlation IDs	90 days for trend analysis and capacity planning

Clock Skew Detection and Compensation

One of the most insidious debugging challenges in distributed rate limiting stems from clock skew between system components. When different nodes have slightly different time references, time-based rate limiting algorithms can produce inconsistent and confusing behavior that's difficult to diagnose without systematic time correlation.

The `TimeProvider` component implements clock skew detection and compensation to ensure consistent time references across the distributed system:

Method	Purpose	Implementation Approach	Error Handling
`MeasureClockSkew(ctx context.Context) (time.Duration, error)`	Compare local time with Redis server time	Execute Redis TIME command and calculate difference from local clock	Retry on network errors, alert on skew > 100ms
`Now() time.Time`	Provide skew-compensated current time	Apply measured compensation to local time	Fall back to local time on Redis unavailability
`SyncWithRedis(ctx context.Context) error`	Periodic synchronization with authoritative time source	Background goroutine measuring skew every 30 seconds	Circuit breaker on persistent sync failures

Debugging Workflow for Common Scenarios

Effective distributed debugging follows systematic workflows that guide investigation from initial symptom observation through root cause identification and resolution verification. These workflows account for the complexity of distributed system interactions while providing actionable steps for developers at different experience levels.

Scenario 1: Inconsistent Rate Limiting Across Instances

When the same user experiences different rate limiting behavior depending on which application instance handles their request, the root cause typically lies in state synchronization, configuration propagation, or local fallback activation.

Investigation Workflow:

1. **Gather Request Context**: Collect correlation IDs, timestamps, and instance identifiers for both the allowed and denied requests
2. **Verify Configuration Consistency**: Check that all instances have the same rate limiting rules and Redis configuration
3. **Examine Redis Connectivity**: Confirm all instances maintain healthy connections to the same Redis cluster
4. **Analyze Local Fallback Status**: Determine if any instances have activated local fallback due to Redis issues
5. **Check Clock Synchronization**: Measure time differences between instances and Redis to identify skew issues
6. **Trace Key Composition**: Verify that identical requests generate the same Redis keys across all instances

Scenario 2: Performance Degradation Under Load

When rate limiting performance degrades as traffic increases, the issue typically involves connection pool exhaustion, hot key concentration, or algorithmic complexity scaling problems.

Investigation Workflow:

1. **Baseline Performance Measurement**: Establish normal response time distribution for rate limiting operations
2. **Connection Pool Analysis**: Monitor Redis connection pool utilization and blocking statistics
3. **Hot Key Identification**: Use Redis monitoring to identify keys receiving disproportionate traffic
4. **Algorithm Performance Profiling**: Measure Lua script execution times under different load conditions
5. **Network Latency Assessment**: Evaluate round-trip times to Redis cluster under load
6. **Scaling Factor Analysis**: Determine how performance degradation correlates with traffic volume

Distributed Debugging Tools and Techniques

Tool Category	Specific Tools	Use Case	Implementation Approach
Log Aggregation	ELK Stack, Fluentd, Loki	Centralized log collection and search	Ship logs with correlation IDs, use structured JSON format
Metrics Collection	Prometheus, InfluxDB	Performance monitoring and alerting	Custom metrics for rate limiting operations, Redis health
Distributed Tracing	Jaeger, Zipkin (sampled)	Complex request flow analysis	1% sampling for detailed traces, correlation ID propagation
Redis Monitoring	RedisInsight, Redis monitoring commands	Redis-specific debugging	Real-time key inspection, memory analysis, performance profiling
Network Analysis	tcpdump, Wireshark, network monitoring	Network-level debugging	Packet capture for Redis communication, latency measurement

Implementation Guidance

Technology Recommendations for Debugging Infrastructure

Component	Simple Option	Advanced Option
Log Aggregation	Local log files with correlation IDs	ELK Stack with Fluentd collection
Metrics Collection	Built-in Prometheus metrics	Custom metrics with Grafana dashboards
Distributed Tracing	Structured logging with trace IDs	Jaeger with sampling
Redis Monitoring	Redis CLI commands + scripts	RedisInsight + custom monitoring
Debugging Tools	Go pprof + manual log analysis	Comprehensive observability platform

Recommended File Structure for Debugging Infrastructure

```
project-root/
  internal/
    debugging/
      correlation/
        trace_context.go      ← correlation ID management
        request_context.go    ← request tracing context
      metrics/
        collector.go          ← metrics collection
        redis_metrics.go      ← Redis-specific metrics
      logging/
        structured_logger.go  ← structured logging utilities
        correlation_logger.go ← correlation-aware logging
      diagnostics/
        redis_diagnostics.go  ← Redis debugging utilities
        cluster_diagnostics.go ← cluster health checking
        symptom_analyzer.go   ← automated symptom analysis
  cmd/debug/
    redis-cli.go             ← Redis debugging CLI tool
    cluster-health.go        ← cluster health checker
    trace-analyzer.go        ← correlation ID trace analysis
  scripts/
    redis-debug.sh           ← Redis debugging scripts
    log-correlation.py       ← log correlation utilities
```

Correlation ID Management Infrastructure

```go
// CorrelationContext manages request correlation across distributed components
type CorrelationContext struct {
    TraceID     string            `json:"trace_id"`
    RequestID   string            `json:"request_id"`
    UserID      string            `json:"user_id,omitempty"`
    SessionID   string            `json:"session_id,omitempty"`
    Metadata    map[string]string `json:"metadata,omitempty"`
    StartTime   time.Time         `json:"start_time"`
}

// NewCorrelationContext creates a new correlation context with unique identifiers
func NewCorrelationContext(userID string) *CorrelationContext {
    // TODO: Generate unique trace ID using UUID or similar
    // TODO: Create request ID for this specific request
    // TODO: Capture start timestamp for duration calculation
    // TODO: Initialize metadata map for additional context
    // Hint: Use crypto/rand for secure random ID generation
}

// WithMetadata adds metadata to correlation context
func (c *CorrelationContext) WithMetadata(key, value string) *CorrelationContext {
    // TODO: Add key-value pair to metadata map
    // TODO: Return context for method chaining
    // Hint: Consider creating a copy to avoid mutation
}

// PropagateToHeaders converts correlation context to HTTP headers
func (c *CorrelationContext) PropagateToHeaders() map[string]string {
    // TODO: Convert correlation context to HTTP headers
    // TODO: Use standard header names (X-Trace-ID, X-Request-ID)
    // TODO: Serialize metadata as JSON if needed
    // Hint: Follow OpenTracing header conventions
}
```

Structured Logging with Correlation

```go
// CorrelationLogger provides correlation-aware structured logging
type CorrelationLogger struct {
    logger    *logrus.Logger
    component string
}

// NewCorrelationLogger creates a correlation-aware logger for a component
func NewCorrelationLogger(component string) *CorrelationLogger {
    // TODO: Initialize logrus logger with JSON formatter
    // TODO: Set appropriate log level from environment
    // TODO: Configure output destination (stdout/file)
    // TODO: Store component name for automatic field injection
    // Hint: Use logrus.JSONFormatter for structured output
}

// LogRateLimitDecision logs rate limiting decisions with full context
func (l *CorrelationLogger) LogRateLimitDecision(
    ctx *CorrelationContext,
    result *RateLimitResult,
    duration time.Duration,
    redisKey string,
) {
    // TODO: Create log entry with correlation fields
    // TODO: Add rate limiting specific fields (allowed, remaining, algorithm)
    // TODO: Include performance metrics (duration, Redis key)
    // TODO: Use appropriate log level based on result
    // Hint: Include all fields needed for debugging and analysis
}

// LogRedisOperation logs Redis operations for debugging
func (l *CorrelationLogger) LogRedisOperation(
    ctx *CorrelationContext,
    operation string,
    key string,
    duration time.Duration,
    err error,
) {
    // TODO: Log Redis operation with timing and error information
    // TODO: Include Redis key and operation type
```

```
    // TODO: Use different log levels for success vs error

    // TODO: Add Redis node information if available

    // Hint: This helps correlate application logs with Redis logs

}
```

Redis Debugging Utilities

```go
// RedisDebugger provides comprehensive Redis debugging capabilities
type RedisDebugger struct {
    client redis.UniversalClient
    logger *CorrelationLogger
}

// NewRedisDebugger creates Redis debugging utilities
func NewRedisDebugger(client redis.UniversalClient) *RedisDebugger {
    // TODO: Initialize debugger with Redis client
    // TODO: Create correlation logger for Redis operations
    // TODO: Set up health checking capabilities
    // Hint: This will be used for manual debugging and automated diagnostics
}

// DiagnoseKeyState provides comprehensive analysis of a rate limiting key
func (d *RedisDebugger) DiagnoseKeyState(ctx context.Context, key string) (*KeyDiagnostics, error) {
    // TODO: Get current value and TTL for the key
    // TODO: Calculate memory usage for the key
    // TODO: Check if key exists in all expected hash ring nodes
    // TODO: Measure access latency for the key
    // TODO: Return comprehensive diagnostic information
    // Hint: Use Redis MEMORY USAGE, TTL, and timing commands
}

// MeasureClusterHealth assesses overall Redis cluster health
func (d *RedisDebugger) MeasureClusterHealth(ctx context.Context) (*ClusterHealth, error) {
    // TODO: Check connectivity to all cluster nodes
    // TODO: Measure response times for each node
    // TODO: Check memory usage and key distribution
    // TODO: Identify any failing or slow nodes
    // TODO: Return overall cluster health assessment
    // Hint: Use Redis CLUSTER commands and timing measurements
}

// AnalyzePerformanceIssues identifies common Redis performance problems
func (d *RedisDebugger) AnalyzePerformanceIssues(ctx context.Context) (*PerformanceAnalysis, error) {
    // TODO: Check for slow operations using SLOWLOG
    // TODO: Analyze memory fragmentation and usage patterns
    // TODO: Identify hot keys causing performance issues
```

```
    // TODO: Check connection pool utilization

    // TODO: Return analysis with recommended actions

    // Hint: Combine multiple Redis diagnostic commands

}
```

Debugging Data Structures

Structure	Fields	Purpose
`KeyDiagnostics`	`Key string, Exists bool, TTL time.Duration, MemoryUsage int64, AccessLatency time.Duration, NodeLocations []string`	Complete diagnostic information for a rate limiting key
`ClusterHealth`	`TotalNodes int, HealthyNodes int, NodeStatuses map[string]*NodeStatus, AverageLatency time.Duration, KeyDistribution map[string]int`	Overall Redis cluster health assessment
`PerformanceAnalysis`	`SlowOperations []SlowOperation, MemoryIssues []string, HotKeys []string, RecommendedActions []string`	Performance problem analysis and recommendations
`NodeStatus`	`Address string, Reachable bool, Latency time.Duration, MemoryUsage float64, Role string, LastError error`	Individual Redis node status information

Milestone Checkpoints

Checkpoint 1: Basic Debugging Infrastructure (After Milestone 1)

- Command: `go test ./internal/debugging/... -v`
- Expected: All debugging utilities compile and basic tests pass
- Manual verification: Generate correlation IDs, verify structured logging output includes correlation fields
- Signs of issues: Missing correlation fields in logs, UUID generation errors

Checkpoint 2: Redis Debugging Tools (After Milestone 3)

- Command: `./cmd/debug/redis-cli -operation=diagnose -key=test_key`
- Expected: Comprehensive key diagnostics including memory usage, TTL, and access latency
- Manual verification: Connect to Redis manually and compare diagnostic output with manual commands
- Signs of issues: Timeout errors, missing Redis connectivity, incorrect key analysis

Checkpoint 3: Distributed Correlation (After Milestone 4)

- Command: Start multiple application instances and trace a request across them using correlation IDs
- Expected: Same correlation ID appears in logs from all instances handling the request
- Manual verification: Search logs for correlation ID, verify complete request flow is captured
- Signs of issues: Missing correlation in some instances, broken correlation chain

Checkpoint 4: Performance Debugging (After Milestone 5)

- Command: `go test -bench=. -memprofile=mem.prof -cpuprofile=cpu.prof`
- Expected: Performance profiles show debugging overhead < 1% of total execution time
- Manual verification: Run load test with debugging enabled, verify minimal performance impact
- Signs of issues: High debugging overhead, memory leaks in correlation tracking

Common Debugging Pitfalls and Solutions

⚠️ **Pitfall: Correlation ID Not Propagated** Many developers forget to propagate correlation IDs through all system boundaries, leading to broken trace chains. This happens when middleware doesn't extract IDs from headers or when background processes don't inherit correlation context. Solution: Create explicit correlation context types and ensure every system boundary explicitly handles correlation propagation.

⚠️ **Pitfall: Clock Skew Ignored in Debugging** Time-based debugging assumes synchronized clocks, but distributed systems often have clock skew that makes event ordering confusing. Log timestamps from different nodes can't be directly compared without accounting for skew. Solution: Always include a time synchronization check in debugging workflows and normalize timestamps to a common reference.

⚠️ **Pitfall: Debugging Overhead in Production** Verbose debugging significantly impacts rate limiting performance when left enabled in production. Detailed logging and tracing can double response times. Solution: Implement sampling for detailed traces (1-5% of requests) and use structured logging with configurable verbosity levels.

⚠️ **Pitfall: Redis Debug Commands in Production** Running Redis debugging commands like SLOWLOG or MEMORY USAGE during high load can impact Redis performance. These commands can block Redis briefly while gathering information. Solution: Use Redis replicas for debugging commands when possible, and limit debug command frequency during peak traffic.

⚠️ **Pitfall: Incomplete Error Context** When rate limiting errors occur, developers often log the immediate error without sufficient context about the request, Redis state, or system conditions. This makes root cause analysis nearly impossible. Solution: Always include correlation context, request details, Redis key information, and system state in error logs.

Future Extensions

> **Milestone(s):** Beyond current implementation scope - these extensions represent advanced capabilities that could be added after completing all five core milestones

The distributed rate limiter we've designed provides a solid foundation for production rate limiting needs. However, the landscape of distributed systems continues to evolve, presenting new challenges and opportunities. This section explores three major extensions that would significantly enhance the system's capabilities: adaptive rate limiting that responds dynamically to system conditions, geographic distribution for global scale applications, and seamless integration with modern service mesh architectures.

These extensions represent natural evolution paths as organizations scale their infrastructure and face increasingly complex operational requirements. Each extension builds upon the core architecture while introducing sophisticated new capabilities that address real-world operational challenges.

Adaptive Rate Limiting

Mental Model: Smart Traffic Light System

Think of adaptive rate limiting like a smart traffic management system in a busy city. Traditional traffic lights operate on fixed timers regardless of actual traffic conditions - they might keep cars waiting at a red light even when no cross traffic exists. Smart traffic lights, however, use sensors to detect real traffic patterns and adjust timing dynamically. Similarly, traditional rate limiting uses fixed thresholds regardless of system health, while adaptive rate limiting monitors system performance and adjusts limits based on actual capacity and response times.

Adaptive rate limiting represents a fundamental shift from static resource protection to dynamic capacity management. Instead of enforcing predetermined limits regardless of system state, an adaptive system continuously monitors performance indicators and automatically adjusts rate limits to maintain optimal system health while maximizing throughput.

The core principle behind adaptive rate limiting involves establishing a feedback loop between system performance metrics and rate limit thresholds. When the system operates well below capacity with fast response times, rate limits can be relaxed to allow more traffic. Conversely, when response times increase or error rates climb, the system tightens limits to prevent cascade failures.

Decision: Feedback Loop Architecture

- **Context**: Static rate limits either waste capacity during low load or fail to protect during unexpected traffic spikes
- **Options Considered**:
  1. External monitoring system that updates rate limit rules via API
  2. Built-in adaptive controller that adjusts limits based on local metrics
  3. Machine learning-based predictor that forecasts optimal limits
- **Decision**: Built-in adaptive controller with configurable feedback algorithms
- **Rationale**: External systems introduce latency and complexity, while ML approaches require significant data and expertise. Built-in controllers provide real-time adaptation with predictable behavior.
- **Consequences**: Enables automatic scaling of rate limits but requires careful tuning to prevent oscillations

Adaptive Algorithm Design

The adaptive rate limiting system operates through several interconnected components that monitor system health, calculate optimal limits, and gradually adjust thresholds to prevent shock changes that could destabilize the system.

Performance Metrics Collection

The foundation of adaptive rate limiting lies in comprehensive performance monitoring. The system must collect metrics across multiple dimensions to understand system health holistically.

Metric Category	Specific Metrics	Collection Method	Update Frequency
Response Time	P50, P95, P99 latency	Request timing instrumentation	Every 10 seconds
Error Rate	5xx errors, timeout rate	Response status tracking	Every 10 seconds
System Resources	CPU utilization, memory usage	OS metrics collection	Every 30 seconds
Queue Depth	Pending requests, worker utilization	Application metrics	Every 5 seconds
Downstream Health	Dependency response times	Service mesh metrics	Every 15 seconds

The metrics collection system must balance accuracy with overhead. High-frequency collection provides better responsiveness but consumes system resources. The adaptive controller uses exponential smoothing to reduce noise while maintaining sensitivity to genuine performance changes.

Control Algorithm Implementation

The adaptive control algorithm implements a PID (Proportional-Integral-Derivative) controller that adjusts rate limits based on the difference between target and actual performance metrics.

The proportional component responds to current performance deviation from targets. If response times exceed the target by 20%, the proportional controller immediately reduces rate limits proportionally. The integral component addresses sustained deviations by accumulating error over time, ensuring persistent performance issues result in continued limit adjustments. The derivative component anticipates future performance by observing the rate of change, helping prevent overshoot when conditions improve rapidly.

1. The controller samples current performance metrics every adjustment interval (typically 30-60 seconds)
2. It calculates the error between actual performance and target thresholds for each monitored metric
3. The proportional component computes immediate adjustment based on current error magnitude
4. The integral component accumulates historical error to address persistent issues
5. The derivative component considers error rate of change to anticipate trends
6. All components combine to produce a rate limit adjustment factor
7. The system applies adjustment gradually over multiple intervals to prevent shock changes
8. Safety bounds prevent adjustments beyond configured minimum and maximum limits

Oscillation Prevention

One of the primary challenges in adaptive systems involves preventing oscillations where limits bounce between high and low values. This occurs when the system overreacts to performance changes, creating instability rather than improvement.

The system implements several oscillation prevention mechanisms:

Hysteresis: Different thresholds trigger increases versus decreases in rate limits. Response times must drop below 80% of the target to increase limits but must exceed 120% to decrease them. This prevents constant adjustments around the threshold.

Rate of Change Limits: Adjustments cannot exceed 10% per interval regardless of error magnitude. Large performance changes trigger multiple small adjustments rather than dramatic swings.

Adjustment History: The controller tracks recent adjustments and reduces sensitivity when frequent changes occur. If limits have been adjusted more than three times in the past hour, subsequent changes require larger error magnitudes.

Warmup Periods: After any adjustment, the system waits for performance metrics to stabilize before considering additional changes. This prevents cascading adjustments based on transient effects.

Integration with Existing Architecture

Adaptive rate limiting integrates into the existing distributed rate limiter architecture through several key extension points that maintain backward compatibility while adding dynamic capabilities.

The `AdaptiveController` component sits between the metrics collection system and the rate limit rule management, automatically updating rule thresholds based on performance feedback.

Component	Interface	Responsibility
MetricsCollector	`CollectPerformanceMetrics() *PerformanceSnapshot`	Gather system performance indicators
AdaptiveController	`UpdateLimits(ctx context.Context, metrics *PerformanceSnapshot) error`	Calculate optimal rate limits
ControlAlgorithm	`CalculateAdjustment(current, target PerformanceMetrics) float64`	PID control algorithm implementation
SafetyBounds	`ValidateAdjustment(currentLimit, proposedLimit int64) int64`	Prevent dangerous limit changes

The adaptive system extends the existing `RateLimitRule` structure with adaptive configuration fields while maintaining compatibility with static rules.

Field Name	Type	Description
adaptive_enabled	bool	Whether this rule participates in adaptive adjustment
target_p99_latency	time.Duration	Maximum acceptable P99 response time
target_error_rate	float64	Maximum acceptable error rate percentage
min_limit	int64	Absolute minimum requests allowed per window
max_limit	int64	Absolute maximum requests allowed per window
adjustment_sensitivity	float64	Multiplier for control algorithm responsiveness

Common Pitfalls in Adaptive Systems

⚠️ **Pitfall: Rapid Oscillation** Many adaptive implementations suffer from oscillation where limits bounce rapidly between high and low values. This occurs when the system overreacts to temporary performance changes without considering adjustment history. The fix involves implementing hysteresis with different thresholds for increasing versus decreasing limits, and rate-limiting the frequency of adjustments.

⚠️ **Pitfall: Cascade Failures During Traffic Spikes** When traffic suddenly increases, response times may spike temporarily even if the system can handle the load. Poorly tuned adaptive systems may aggressively reduce limits, creating artificial scarcity that prevents the system from recovering. The solution requires distinguishing between temporary load spikes and genuine capacity problems by considering multiple metrics simultaneously.

⚠️ **Pitfall: Ignoring Downstream Dependencies** Adaptive systems that only monitor local performance may miss downstream bottlenecks. When a database becomes slow, the adaptive system might increase local rate limits, worsening the downstream problem. Comprehensive adaptive systems must monitor the health of all critical dependencies and adjust limits based on the weakest link in the chain.

Geographic Distribution

Mental Model: Global Banking Network

Consider how major banks operate across different countries and time zones. Each branch maintains local cash reserves for immediate customer needs, but they're all connected to regional centers that can transfer funds when needed. Customer account balances must be consistent globally - you can't withdraw the same money from ATMs in New York and London simultaneously. However, the system tolerates brief inconsistencies during transfers as long as the final state is correct. Geographic rate limiting works similarly, with local enforcement for immediate decisions and eventual consistency for global accuracy.

Geographic distribution extends the distributed rate limiter to operate across multiple geographic regions, each potentially containing multiple data centers. This architecture addresses the needs of global applications that serve users worldwide while maintaining rate limiting accuracy across regions.

The fundamental challenge in geographic distribution lies in balancing consistency with performance. Users expect sub-100ms response times regardless of their location, but maintaining perfect global consistency for rate limits would require cross-region network calls that add hundreds of milliseconds of latency.

The solution involves a hybrid approach combining local enforcement with asynchronous global reconciliation. Each region maintains local rate limit state that enables immediate decisions, while background processes synchronize state across regions to maintain global accuracy over time.

Multi-Region Architecture Design

The geographic distribution architecture consists of three primary layers: regional clusters, global coordination, and conflict resolution mechanisms.

Regional Cluster Organization

Each geographic region operates as an independent rate limiting cluster with complete functionality for serving local traffic. Regional clusters contain their own Redis shards, application instances, and management APIs.

Component	Regional Instance	Global Coordination
Redis Cluster	3-5 nodes per region	Cross-region replication streams
Application Instances	Serve local traffic only	Participate in global state sync
Management API	Regional configuration	Global rule propagation
Monitoring Dashboard	Regional metrics	Aggregated global view

Regional clusters implement the complete distributed rate limiting architecture described in previous sections. They can operate independently even when network connectivity to other regions fails, ensuring local users experience no service degradation during network partitions.

Global State Synchronization

Global state synchronization operates through asynchronous replication streams that propagate rate limit usage information between regions. Unlike traditional database replication, rate limiting synchronization focuses on aggregate consumption rather than individual request tracking.

The synchronization protocol operates on a periodic basis (typically every 30-60 seconds) where each region reports its usage statistics to other regions:

1. Each regional cluster aggregates rate limit usage across all local keys and time windows
2. Usage reports include consumed tokens, timestamp ranges, and region identifiers
3. Reports transmit to other regions through reliable message queues or direct Redis streams
4. Receiving regions incorporate remote usage into their local enforcement decisions
5. Conflict resolution algorithms handle cases where global usage exceeds configured limits
6. Corrective actions redistribute quota or temporarily tighten local enforcement

Eventual Consistency Model

The geographic distribution system operates under an eventual consistency model where rate limits may temporarily exceed global thresholds during network partitions or high cross-region latency, but converge to correct enforcement as synchronization catches up.

Consistency Guarantee	Local Enforcement	Global Convergence
Strong Consistency	Not achievable without unacceptable latency	Not suitable for user-facing responses
Eventual Consistency	Immediate decisions based on local + cached remote state	Global limits enforced within sync interval
Bounded Staleness	Local decisions with maximum staleness bounds	Guarantees global limit violations stay within bounds

The bounded staleness approach provides the best balance for most applications. Local enforcement uses recently synchronized global state (maximum 2-3 minutes stale) combined with current local usage to make decisions. This ensures global violations remain bounded even during extended network partitions.

Cross-Region Synchronization Protocol

The synchronization protocol ensures efficient propagation of rate limiting state while handling network failures, region outages, and conflicting updates from multiple sources.

Incremental State Transfer

Rather than synchronizing complete rate limiting state, the protocol transmits incremental updates focusing on consumed quota within specific time windows.

Message Type	Contents	Frequency	Size
Usage Update	Key, consumed tokens, window, timestamp	Every 60 seconds	<1KB per key
Heartbeat	Region health, sync lag, active keys	Every 30 seconds	<100 bytes
Quota Adjustment	Key, new global limit, reason	On configuration change	<500 bytes
Conflict Resolution	Key, authoritative state, reconciliation	When conflicts detected	<2KB

Usage updates contain aggregated consumption data rather than individual request logs. For a key with 1000 requests in the last minute, the system sends a single update indicating "1000 tokens consumed" rather than 1000 individual records.

Conflict Detection and Resolution

Conflicts arise when the sum of regional usage reports exceeds the configured global limit for a key. This can occur during network partitions where regions operate independently or during traffic spikes that exceed synchronization frequency.

The conflict resolution protocol implements a priority-based approach:

1. **Detection Phase**: Each region compares local + remote usage against global limits during sync processing
2. **Reporting Phase**: Regions experiencing conflicts broadcast conflict notifications to all other regions
3. **Priority Resolution**: Regions use deterministic priority rules (timestamp, region ID) to determine authoritative state
4. **Reconciliation Phase**: Non-authoritative regions adjust their local enforcement to compensate for over-consumption
5. **Recovery Phase**: Gradual restoration of normal enforcement as usage patterns stabilize

Network Partition Handling

Network partitions between regions represent one of the most challenging scenarios for geographic distribution. The system must continue serving users in each region while preventing excessive global limit violations.

During partition detection (missed heartbeats, failed sync operations), each region transitions to conservative enforcement mode:

- Local rate limits reduce by a safety margin (typically 20-30%) to compensate for unknown remote usage
- Cached remote state continues informing decisions but with increasing skepticism over time
- Critical keys (those approaching limits) receive more aggressive local enforcement
- Non-critical keys maintain normal enforcement to avoid unnecessary user impact

When partitions heal, regions exchange complete state snapshots to reconcile any conflicts that occurred during isolation. The reconciliation process prioritizes preserving user experience over perfect quota enforcement, typically allowing temporary over-consumption rather than retroactively blocking users.

Common Pitfalls in Geographic Distribution

⚠️ **Pitfall: Ignoring Network Latency Variations** Many geographic distribution implementations assume consistent network latency between regions. In reality, trans-Pacific links may have 300ms latency while trans-Atlantic links have 150ms. This variation affects synchronization timing and can cause some regions to operate with staler data than others. The solution involves adaptive sync intervals based on measured network latency and prioritizing synchronization between high-latency region pairs.

⚠️ **Pitfall: Global Limits That Are Too Restrictive** Setting global limits equal to the sum of regional capacity ignores cross-regional traffic distribution. If Europe has 40% of global users but only 25% of quota, European users will experience unnecessary blocking while other regions have unused capacity. Effective geographic distribution requires quota allocation that matches actual traffic distribution with automatic rebalancing capabilities.

⚠️ **Pitfall: Insufficient Partition Tolerance Testing** Geographic distribution systems often work perfectly during testing with reliable networks but fail catastrophically during real network partitions. Comprehensive testing must simulate various partition scenarios: complete isolation, asymmetric partitions where region A can reach B but not vice versa, and flapping connections that repeatedly partition and heal.

Service Mesh Integration

Mental Model: Airport Security Integration

Consider how security checkpoints integrate into airport operations. Rather than requiring every airline to implement their own security protocols, airports provide centralized security infrastructure that all flights use transparently. Passengers don't need to understand different security systems - they simply pass through standardized checkpoints. Similarly, service mesh integration allows applications to benefit from rate limiting without implementing rate limiting logic themselves. The mesh intercepts traffic transparently and applies policies consistently across all services.

Service mesh integration represents a paradigm shift from application-embedded rate limiting to infrastructure-level policy enforcement. Instead of each service implementing its own rate limiting logic, the service mesh (Istio, Linkerd, or Consul Connect) intercepts network traffic and applies rate limiting policies transparently.

This approach offers significant operational advantages: consistent policy enforcement across all services, centralized configuration management, and the ability to apply rate limiting to legacy applications without code changes. However, it also introduces new complexity in policy management and debugging distributed policy enforcement.

Envoy Proxy Integration Design

Most service meshes use Envoy proxy as their data plane component, making Envoy integration the primary path for service mesh rate limiting support. Envoy provides several extension points for implementing custom rate limiting logic.

Rate Limiting Filter Architecture

Envoy implements rate limiting through HTTP filters that intercept requests before they reach upstream services. The distributed rate limiter integrates as an external authorization service that Envoy queries for each request.

Envoy Component	Integration Point	Responsibility
HTTP Connection Manager	Rate Limit Filter	Intercepts requests and queries rate limiting service
Cluster Manager	Rate Limit Service Cluster	Manages connections to rate limiting service instances
Admin Interface	Statistics and Health	Exposes rate limiting metrics and health status
Configuration API	Dynamic Updates	Receives rate limiting policy updates

The integration follows Envoy's external authorization pattern where the proxy sends request context to the rate limiting service and waits for an allow/deny decision. This design maintains clean separation between traffic proxying and policy enforcement.

Request Context Extraction

The service mesh integration must extract relevant rate limiting context from HTTP requests and connection metadata. Unlike application-level integration where context is readily available, proxy-level integration must infer context from network-level information.

Context Source	Available Information	Rate Limiting Application
HTTP Headers	User-Agent, Authorization, Custom headers	User identification, API versioning
Connection Metadata	Source IP, Destination service	IP-based limiting, service quotas
TLS Certificate	Client certificate CN, SAN	Certificate-based user identification
Request Path	URL path, query parameters	API endpoint classification
Service Labels	Kubernetes labels, service mesh metadata	Service-to-service rate limiting

The context extraction process operates through configurable rules that map network-level information to rate limiting dimensions. For example, a rule might extract user ID from the "X-User-ID" header and map it to per-user rate limiting, while another rule uses the destination service name for global service rate limiting.

Policy Configuration Integration

Service mesh policy configuration differs significantly from application-level configuration. Instead of managing rate limiting rules through dedicated APIs, policies integrate with service mesh configuration systems like Istio's VirtualService and DestinationRule resources.

```yaml
# Example Istio integration (for illustration - actual implementation in code section)
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: api-rate-limiting
spec:
  hosts:
  - api.example.com
  http:
  - match:
    - uri:
        prefix: /api/v1
    route:
    - destination:
        host: api-service
    rateLimiting:
      algorithm: token_bucket
      limit: 1000
      window: 60s
      dimensions:
      - header: "x-user-id"
        tier: user
      - source_ip: {}
        tier: ip
```

This approach provides several advantages over standalone rate limiting services: configuration lives alongside routing rules, policies automatically apply to matching traffic, and service mesh tools provide policy validation and deployment workflows.

Transparent Policy Enforcement

Transparent policy enforcement ensures applications receive rate limiting protection without requiring code changes or awareness of rate limiting infrastructure. This transparency extends to multiple aspects of the system: request handling, error responses, and monitoring integration.

Request Flow Transparency

From the application's perspective, rate limiting enforcement appears as natural network behavior rather than explicit policy enforcement. Applications see either successful request delivery or standard HTTP error responses - they don't need special handling for rate limiting scenarios.

The transparency requirement affects several design decisions:

1. **Response Headers**: Rate limiting information (remaining quota, retry timing) appears in standard HTTP headers that applications can optionally consume
2. **Error Codes**: Rate limiting uses standard HTTP 429 (Too Many Requests) responses rather than custom error formats
3. **Latency Impact**: Rate limiting decisions complete within microseconds to avoid noticeable request latency
4. **Circuit Breaker Integration**: When rate limiting services fail, traffic passes through normally rather than blocking

Policy Inheritance and Precedence

Service mesh environments often have complex service hierarchies with multiple applicable policies. The rate limiting system must resolve policy conflicts and inheritance in predictable ways.

Policy Scope	Precedence Level	Example Application
Global Mesh	Lowest	Default rate limiting for all services
Namespace	Medium	Rate limiting for all services in a namespace
Service	High	Rate limiting specific to one service
Route	Highest	Rate limiting for specific API endpoints

Higher precedence policies override lower precedence ones, but the system also supports policy composition where multiple policies can apply to the same request. For example, a global policy might set overall limits while a service-specific policy adds burst handling.

Error Response Customization

Different services may require different error response formats when rate limiting blocks requests. API services might expect JSON error responses, while web applications might prefer HTML error pages with user-friendly messages.

The service mesh integration provides configurable error response templates that can adapt to service requirements:

Service Type	Response Format	Headers	Body Content
REST API	JSON	application/json	{"error": "rate_limit_exceeded", "retry_after": 60}
GraphQL	JSON	application/json	{"errors": [{"message": "Rate limit exceeded"}]}
Web Service	HTML	text/html	User-friendly error page with retry guidance
gRPC	Status	application/grpc	RESOURCE_EXHAUSTED with retry metadata

Observability and Debugging

Service mesh integration introduces additional complexity in observability since rate limiting decisions occur in infrastructure rather than application code. Comprehensive observability becomes critical for debugging policy enforcement and understanding traffic patterns.

Distributed Tracing Integration

Rate limiting decisions must integrate with distributed tracing systems (Jaeger, Zipkin) to provide visibility into policy enforcement within request traces. Each rate limiting decision appears as a span within the overall request trace.

Trace Information	Content	Purpose
Span Name	"rate_limiting_check"	Identifies rate limiting operations in traces
Span Tags	rule_id, algorithm, tier, decision	Provides context for rate limiting decisions
Span Logs	quota_remaining, rule_evaluation_time	Detailed information for debugging
Span Status	OK (allowed) or ERROR (denied)	Quick visual indication of decision

The tracing integration helps operators understand why specific requests were rate limited and identify patterns in rate limiting decisions across different services.

Metrics Aggregation Across Services

Unlike application-embedded rate limiting where each service has its own metrics, service mesh rate limiting must aggregate metrics across all services while maintaining the ability to drill down into specific service behavior.

The metrics system provides multiple aggregation levels:

- **Mesh-wide**: Total requests, rate limiting decisions, error rates across all services
- **Service-level**: Rate limiting metrics per service, including top rate limited endpoints
- **Policy-level**: Effectiveness metrics for each rate limiting policy
- **Client-level**: Rate limiting behavior per client across all services they access

These aggregated metrics enable both high-level mesh monitoring and detailed debugging of specific rate limiting issues.

Common Pitfalls in Service Mesh Integration

⚠️ **Pitfall: Configuration Complexity** Service mesh rate limiting can quickly become complex with multiple policy layers, inheritance rules, and service-specific configurations. Teams often create contradictory policies or policies that interact in unexpected ways. The solution involves policy validation tools that check for conflicts and comprehensive testing of policy interactions before deployment.

⚠️ **Pitfall: Performance Impact on Critical Path** Every request must wait for rate limiting decisions, making performance critical. Poorly optimized integrations can add tens of milliseconds to request latency, violating SLA requirements. High-performance integration requires connection pooling, efficient serialization, and aggressive caching of rate limiting decisions.

⚠️ **Pitfall: Debugging Distributed Policies** When rate limiting occurs in the service mesh, applications may not have visibility into why requests fail. This makes debugging customer issues extremely difficult. Comprehensive logging, tracing integration, and clear error messages become essential for operational success.

Implementation Guidance

The future extensions represent advanced capabilities that build upon the core distributed rate limiter foundation. Each extension requires careful architectural decisions and implementation strategies to maintain system reliability while adding sophisticated new features.

Technology Recommendations

Extension Area	Simple Option	Advanced Option
Adaptive Control	Basic PID controller with manual tuning	Machine learning-based limit prediction with auto-tuning
Geographic Sync	Direct Redis replication streams	Apache Kafka-based event streaming with exactly-once semantics
Service Mesh	Envoy HTTP filter with REST API calls	Native Envoy gRPC integration with connection multiplexing
Metrics Collection	Prometheus metrics with periodic scraping	OpenTelemetry with real-time streaming to observability platforms
Configuration	YAML files with manual deployment	GitOps workflow with policy validation and gradual rollouts

Recommended Module Structure

Future extensions integrate into the existing codebase through new packages that maintain clear separation of concerns:

```
distributed-rate-limiter/
  internal/
    adaptive/              ← adaptive rate limiting
      controller.go        ← PID control algorithm
      metrics.go           ← performance metrics collection
      safety.go            ← oscillation prevention
    geographic/            ← multi-region distribution
      sync.go              ← cross-region synchronization
      partition.go         ← network partition handling
      conflict.go          ← conflict resolution
    servicemesh/           ← service mesh integration
      envoy/               ← Envoy proxy integration
        filter.go          ← HTTP filter implementation
        config.go          ← policy configuration
      istio/               ← Istio-specific integrations
        policies.go        ← VirtualService integration
      observability.go     ← tracing and metrics
    extensions/            ← shared extension utilities
      timeutil.go          ← time synchronization utilities
      configutil.go        ← configuration validation
```

Adaptive Rate Limiting Starter Code

The adaptive controller provides the foundation for dynamic rate limit adjustment based on system performance metrics.

```go
// Package adaptive provides dynamic rate limit adjustment based on system performance
package adaptive

import (
    "context"
    "sync"
    "time"

    "github.com/distributed-rate-limiter/internal/config"
    "github.com/distributed-rate-limiter/internal/metrics"
)

// PerformanceMetrics represents current system performance indicators
type PerformanceMetrics struct {
    P50Latency      time.Duration `json:"p50_latency"`
    P95Latency      time.Duration `json:"p95_latency"`
    P99Latency      time.Duration `json:"p99_latency"`
    ErrorRate       float64       `json:"error_rate"`
    CPUUtilization  float64       `json:"cpu_utilization"`
    MemoryUsage     float64       `json:"memory_usage"`
    QueueDepth      int64         `json:"queue_depth"`
    Timestamp       time.Time     `json:"timestamp"`
}

// AdaptiveConfig defines configuration for adaptive rate limiting behavior
type AdaptiveConfig struct {
    Enabled             bool          `yaml:"enabled"`
    AdjustmentInterval  time.Duration `yaml:"adjustment_interval"`
    TargetP99Latency    time.Duration `yaml:"target_p99_latency"`
    TargetErrorRate     float64       `yaml:"target_error_rate"`
    MaxAdjustmentRate   float64       `yaml:"max_adjustment_rate"`
    ProportionalGain    float64       `yaml:"proportional_gain"`
    IntegralGain        float64       `yaml:"integral_gain"`
    DerivativeGain      float64       `yaml:"derivative_gain"`
}

// PIDController implements proportional-integral-derivative control for rate limit adjustment
type PIDController struct {
    mu              sync.RWMutex
```

```
    config          AdaptiveConfig
    previousError   float64
    integralError   float64
    lastUpdate      time.Time
    adjustmentHistory []AdjustmentRecord
}

// AdjustmentRecord tracks historical rate limit adjustments for oscillation prevention
type AdjustmentRecord struct {
    Timestamp       time.Time `json:"timestamp"`
    PreviousLimit   int64     `json:"previous_limit"`
    NewLimit        int64     `json:"new_limit"`
    AdjustmentRatio float64   `json:"adjustment_ratio"`
    TriggerReason   string    `json:"trigger_reason"`
}

// AdaptiveController manages dynamic rate limit adjustments based on system performance
type AdaptiveController struct {
    mu                sync.RWMutex
    config            AdaptiveConfig
    pidController     *PIDController
    ruleManager       *config.RuleManager
    metricsCollector  *metrics.Collector
    adjustmentTimer   *time.Timer
    ctx               context.Context
    cancelFn          context.CancelFunc
}

// NewAdaptiveController creates a new adaptive rate limiting controller
func NewAdaptiveController(config AdaptiveConfig, ruleManager *config.RuleManager,
                           metricsCollector *metrics.Collector) *AdaptiveController {
    // TODO: Initialize PID controller with configured gains
    // TODO: Set up adjustment timer based on configured interval
    // TODO: Create cancellable context for graceful shutdown
    // TODO: Initialize adjustment history tracking
    return nil
}

// Start begins adaptive rate limit adjustment processing
```

```go
func (ac *AdaptiveController) Start(ctx context.Context) error {
    // TODO: Start background goroutine for periodic adjustment evaluation
    // TODO: Set up metrics collection integration
    // TODO: Initialize baseline performance measurements
    // TODO: Begin adjustment timer
    return nil
}

// Stop gracefully shuts down adaptive processing
func (ac *AdaptiveController) Stop() error {
    // TODO: Cancel background processing context
    // TODO: Stop adjustment timer
    // TODO: Flush any pending adjustments
    return nil
}

// EvaluateAdjustments analyzes current performance and determines necessary rate limit changes
func (ac *AdaptiveController) EvaluateAdjustments(ctx context.Context) error {
    // TODO: Collect current performance metrics
    // TODO: Compare against target thresholds
    // TODO: Calculate PID controller output
    // TODO: Apply safety bounds and oscillation prevention
    // TODO: Update rate limit rules if adjustment needed
    // TODO: Record adjustment in history
    return nil
}

// CalculatePIDOutput computes rate limit adjustment using PID control algorithm
func (pc *PIDController) CalculatePIDOutput(current, target PerformanceMetrics) float64 {
    // TODO: Calculate error between current and target performance
    // TODO: Compute proportional term based on current error
    // TODO: Update and compute integral term based on accumulated error
    // TODO: Compute derivative term based on error rate of change
    // TODO: Combine P, I, and D terms with configured gains
    // TODO: Apply output limits to prevent excessive adjustments
    return 0.0
}
```

Geographic Distribution Starter Code

The geographic synchronization system enables rate limit state sharing across multiple regions with eventual consistency guarantees.

```go
// Package geographic provides multi-region rate limiting with eventual consistency
package geographic

import (
    "context"
    "sync"
    "time"

    "github.com/go-redis/redis/v8"
    "github.com/distributed-rate-limiter/internal/storage"
)

// RegionConfig defines configuration for a geographic region
type RegionConfig struct {
    RegionID        string   `yaml:"region_id"`
    RedisAddresses  []string `yaml:"redis_addresses"`
    SyncInterval    time.Duration `yaml:"sync_interval"`
    PeerRegions     []PeerRegion `yaml:"peer_regions"`
    SyncTimeout     time.Duration `yaml:"sync_timeout"`
    ConflictResolution string `yaml:"conflict_resolution"`
}

// PeerRegion defines connection information for remote regions
type PeerRegion struct {
    RegionID    string `yaml:"region_id"`
    SyncAddress string `yaml:"sync_address"`
    Priority    int    `yaml:"priority"`
}

// UsageReport represents rate limit consumption data for cross-region synchronization
type UsageReport struct {
    RegionID      string                    `json:"region_id"`
    Timestamp     time.Time                 `json:"timestamp"`
    WindowStart   time.Time                 `json:"window_start"`
    WindowEnd     time.Time                 `json:"window_end"`
    KeyUsage      map[string]int64          `json:"key_usage"`
    Sequence      int64                     `json:"sequence"`
}

// ConflictReport indicates detected inconsistencies in global rate limit state
```

```
type ConflictReport struct {
    Key           string                    `json:"key"`
    GlobalLimit   int64                     `json:"global_limit"`
    ReportedUsage map[string]int64          `json:"reported_usage"`
    TotalUsage    int64                     `json:"total_usage"`
    Excess        int64                     `json:"excess"`
    Resolution    string                    `json:"resolution"`
    Timestamp     time.Time                 `json:"timestamp"`
}

// GeographicSyncManager coordinates rate limiting state across multiple regions
type GeographicSyncManager struct {
    mu              sync.RWMutex
    config          RegionConfig
    localStorage    storage.Storage
    peerConnections map[string]*redis.Client
    usageBuffer     []UsageReport
    conflictResolver *ConflictResolver
    syncTicker      *time.Ticker
    ctx             context.Context
    cancelFn        context.CancelFunc
}

// ConflictResolver handles inconsistencies in global rate limit state
type ConflictResolver struct {
    mu              sync.RWMutex
    resolutionStrategy string
    priorityMap     map[string]int
    conflictHistory []ConflictReport
}

// NewGeographicSyncManager creates a new geographic synchronization manager
func NewGeographicSyncManager(config RegionConfig, localStorage storage.Storage) *GeographicSyncManager {
    // TODO: Initialize peer region connections
    // TODO: Create conflict resolver with configured strategy
    // TODO: Set up usage report buffering
    // TODO: Initialize sync ticker with configured interval
    return nil
}
```

```go
// Start begins cross-region synchronization processing
func (gsm *GeographicSyncManager) Start(ctx context.Context) error {
    // TODO: Establish connections to all peer regions
    // TODO: Start background sync processing goroutine
    // TODO: Begin periodic usage report generation
    // TODO: Start conflict detection and resolution
    return nil
}

// SynchronizeUsage sends local usage reports to peer regions and processes incoming reports
func (gsm *GeographicSyncManager) SynchronizeUsage(ctx context.Context) error {
    // TODO: Generate usage report from local Redis state
    // TODO: Send usage report to all peer regions
    // TODO: Receive and process usage reports from peers
    // TODO: Detect conflicts between reported usage and global limits
    // TODO: Trigger conflict resolution if needed
    return nil
}

// ProcessPeerReport incorporates usage information from a remote region
func (gsm *GeographicSyncManager) ProcessPeerReport(report UsageReport) error {
    // TODO: Validate report authenticity and sequence
    // TODO: Merge remote usage into local global state view
    // TODO: Check for conflicts with global rate limits
    // TODO: Update local enforcement based on global usage
    return nil
}

// DetectConflicts identifies cases where global usage exceeds configured limits
func (cr *ConflictResolver) DetectConflicts(reports map[string]UsageReport,
                                            globalLimits map[string]int64) []ConflictReport {
    // TODO: Sum reported usage across all regions for each key
    // TODO: Compare total usage against configured global limits
    // TODO: Identify keys where usage exceeds limits
    // TODO: Calculate excess amount and affected regions
    return nil
}

// ResolveConflict applies conflict resolution strategy to restore global consistency
```

```
func (cr *ConflictResolver) ResolveConflict(ctx context.Context, conflict ConflictReport) error {

    // TODO: Apply configured resolution strategy (priority-based, proportional reduction, etc.)

    // TODO: Calculate corrective actions for each region

    // TODO: Send adjustment commands to affected regions

    // TODO: Record conflict resolution in history

    return nil

}
```

Service Mesh Integration Starter Code

The service mesh integration provides transparent rate limiting through Envoy proxy filters and Istio policy integration.

```go
// Package servicemesh provides transparent rate limiting through service mesh integration
package servicemesh

import (
    "context"
    "net/http"
    "time"

    envoy_service_ratelimit_v3 "github.com/envoyproxy/go-control-plane/envoy/service/ratelimit/v3"
    "google.golang.org/grpc"

    "github.com/distributed-rate-limiter/internal/ratelimit"
)

// EnvoyRateLimitService implements Envoy's RateLimitService interface
type EnvoyRateLimitService struct {
    limiter           ratelimit.DistributedLimiter
    contextExtractor  *RequestContextExtractor
    policyManager     *PolicyManager
    metricsCollector  *ServiceMeshMetrics
}

// RequestContextExtractor extracts rate limiting context from Envoy request metadata
type RequestContextExtractor struct {
    extractionRules   map[string]ExtractionRule
    defaultContext    DefaultContextConfig
}

// ExtractionRule defines how to extract rate limiting dimensions from requests
type ExtractionRule struct {
    Dimension     string            `yaml:"dimension"`
    Source        string            `yaml:"source"`        // header, path, query, metadata
    Key           string            `yaml:"key"`           // header name, path pattern, etc.
    Transformer   string            `yaml:"transformer"`   // regex, hash, lookup
    DefaultValue  string            `yaml:"default_value"`
}

// PolicyManager handles service mesh rate limiting policy configuration
type PolicyManager struct {
    mu              sync.RWMutex
```

```
    policies        map[string]*ServiceMeshPolicy
    istioClient     *istio.Client
    configWatcher   *ConfigWatcher
}

// ServiceMeshPolicy represents rate limiting policy in service mesh context
type ServiceMeshPolicy struct {
    Name            string                  `yaml:"name"`
    Namespace       string                  `yaml:"namespace"`
    Services        []string                `yaml:"services"`
    Routes          []RouteMatch            `yaml:"routes"`
    RateLimiting    RateLimitingSpec        `yaml:"rate_limiting"`
    Priority        int                     `yaml:"priority"`
    Enabled         bool                    `yaml:"enabled"`
}

// RouteMatch defines traffic matching criteria for policy application
type RouteMatch struct {
    PathRegex       string            `yaml:"path_regex"`
    Methods         []string          `yaml:"methods"`
    Headers         map[string]string `yaml:"headers"`
    QueryParams     map[string]string `yaml:"query_params"`
}

// RateLimitingSpec defines rate limiting configuration within service mesh policy
type RateLimitingSpec struct {
    Algorithm       string                  `yaml:"algorithm"`
    Limit           int64                   `yaml:"limit"`
    Window          time.Duration           `yaml:"window"`
    Dimensions      []RateLimitDimension    `yaml:"dimensions"`
    ErrorResponse   ErrorResponseConfig     `yaml:"error_response"`
}

// RateLimitDimension defines a dimension for rate limiting (user, IP, service, etc.)
type RateLimitDimension struct {
    Name            string            `yaml:"name"`
    Extractor       ExtractionRule    `yaml:"extractor"`
    Tier            string            `yaml:"tier"`
}
```

```
// ErrorResponseConfig customizes error responses when rate limits are exceeded
type ErrorResponseConfig struct {
    StatusCode      int               `yaml:"status_code"`
    ContentType     string            `yaml:"content_type"`
    Body            string            `yaml:"body"`
    Headers         map[string]string `yaml:"headers"`
}

// NewEnvoyRateLimitService creates a new Envoy rate limiting service
func NewEnvoyRateLimitService(limiter ratelimit.DistributedLimiter) *EnvoyRateLimitService {
    // TODO: Initialize request context extractor with default rules
    // TODO: Create policy manager with Istio client
    // TODO: Set up service mesh metrics collection
    return nil
}

// ShouldRateLimit implements Envoy's rate limiting service interface
func (erls *EnvoyRateLimitService) ShouldRateLimit(ctx context.Context,
                                                   req *envoy_service_ratelimit_v3.RateLimitRequest) (
                                                   *envoy_service_ratelimit_v3.RateLimitResponse, error) {
    // TODO: Extract request context from Envoy metadata
    // TODO: Find matching service mesh policies
    // TODO: Evaluate rate limiting rules for matched policies
    // TODO: Convert rate limiting result to Envoy response format
    // TODO: Include appropriate response headers
    return nil, nil
}

// ExtractRequestContext converts Envoy request metadata to rate limiting context
func (rce *RequestContextExtractor) ExtractRequestContext(
    req *envoy_service_ratelimit_v3.RateLimitRequest) (*ratelimit.RequestContext, error) {
    // TODO: Iterate through extraction rules
    // TODO: Extract values from request headers, path, query parameters
    // TODO: Apply transformers (regex, hashing, lookups)
    // TODO: Build RequestContext with extracted dimensions
    return nil, nil
}

// FindMatchingPolicies identifies service mesh policies that apply to the current request
```

```
func (pm *PolicyManager) FindMatchingPolicies(ctx context.Context,
                                              reqCtx *ratelimit.RequestContext) ([]*ServiceMeshPolicy, error) {
    // TODO: Match request against service selectors
    // TODO: Match request against route patterns
    // TODO: Apply policy precedence rules
    // TODO: Return ordered list of applicable policies
    return nil, nil
}

// ConvertToEnvoyResponse transforms rate limiting result into Envoy response format
func ConvertToEnvoyResponse(result *ratelimit.FlowResult,
                            errorConfig ErrorResponseConfig) *envoy_service_ratelimit_v3.RateLimitResponse {
    // TODO: Set response status based on rate limiting decision
    // TODO: Add rate limiting headers (X-RateLimit-Limit, X-RateLimit-Remaining)
    // TODO: Configure error response if rate limit exceeded
    // TODO: Include retry-after header with appropriate timing
    return nil
}
```

Milestone Checkpoints

Adaptive Rate Limiting Checkpoint: After implementing adaptive rate limiting, run load tests with varying traffic patterns. You should observe rate limits automatically adjusting based on system performance. Monitor the adjustment history to ensure the system doesn't oscillate - limits should converge to stable values that maintain target performance levels.

Geographic Distribution Checkpoint: Deploy the rate limiter across two simulated regions with controlled network latency between them. Generate traffic in both regions and verify that global rate limits are eventually enforced even when regional limits are exceeded. Test network partition scenarios by blocking cross-region communication and confirm that each region continues operating with conservative local enforcement.

Service Mesh Integration Checkpoint: Deploy the rate limiter as an Envoy filter in a test service mesh. Configure rate limiting policies through Istio VirtualService resources and verify that applications receive rate limiting transparently without code changes. Check that rate limiting decisions appear correctly in distributed traces and that error responses match configured formats.

Glossary

> **Milestone(s):** All milestones - this glossary provides definitions for technical terms used across rate limiting algorithms, multi-tier evaluation, Redis backend integration, sharding, and API design throughout the distributed rate limiting system

The distributed rate limiting system introduces numerous technical concepts spanning algorithm design, distributed systems, data storage, and operational management. This glossary provides comprehensive definitions of all terms, acronyms, data structures, and domain concepts used throughout the design document. Understanding these definitions is crucial for implementing, operating, and extending the rate limiting system effectively.

Mental Model: The Technical Reference Library

Think of this glossary as the technical reference library for our distributed rate limiting "city." Just as a city has specialized terminology for its infrastructure (traffic signals, utility grids, zoning laws), our distributed system has its own vocabulary. Each term represents a specific concept, component, or operation that engineers use to communicate precisely about the system's behavior. Like a reference library, this glossary organizes knowledge so that anyone working with the system can quickly understand unfamiliar concepts and use consistent terminology when discussing designs, implementing features, or troubleshooting issues.

Core System Concepts

Distributed Rate Limiting: The coordination of request quota enforcement across multiple application instances using shared state storage. Unlike local rate limiting where each instance operates independently, distributed rate limiting ensures that the combined traffic from all instances respects global quotas. This requires atomic operations, clock synchronization, and graceful degradation strategies.

Local Fallback: The system's ability to switch to per-instance rate limiting when shared storage becomes unavailable. During fallback mode, each application instance enforces rate limits using only local state, accepting that global quotas may be exceeded but maintaining basic protection. The system automatically returns to distributed mode when shared storage connectivity is restored.

Graceful Degradation: The system's capacity to maintain reduced functionality during component failures rather than completely failing. For rate limiting, this means continuing to provide protection (albeit with reduced accuracy) when Redis is unavailable, configuration updates fail, or individual nodes become unreachable.

Atomic Operations: Indivisible check-and-update operations that prevent race conditions in distributed rate limiting. These operations ensure that reading a counter, comparing it to a limit, and incrementing it (if allowed) happen as a single unit, preventing the double-spending problem where multiple instances might simultaneously approve requests that should collectively exceed the limit.

Circuit Breaker: A failure detection and prevention mechanism that protects against cascading failures by monitoring error rates and temporarily blocking operations to failing components. In rate limiting, circuit breakers prevent the system from repeatedly attempting Redis operations that are likely to fail, reducing latency and resource consumption during outages.

Rate Limiting Algorithm Terms

Token Bucket: A rate limiting algorithm that allows controlled bursts above sustained rates by maintaining a bucket of tokens that refill at a steady rate. Requests consume tokens from the bucket; when the bucket is empty, additional requests are denied until tokens are replenished. The bucket capacity determines the maximum burst size, while the refill rate controls the sustained throughput.

Sliding Window Counter: A memory-efficient approximate rate limiting algorithm that divides time into fixed buckets and weights recent buckets more heavily than older ones. This approach provides good accuracy while using constant memory per rate limit key, making it suitable for high-scale deployments where memory usage must be bounded.

Sliding Window Log: A precise rate limiting algorithm that stores individual request timestamps to make exact decisions about whether new requests should be allowed. While providing perfect accuracy, this approach requires O(n) memory per key where n is the number of requests in the time window, making it suitable for scenarios where precision is more important than memory efficiency.

Burst Handling: The system's capability to allow short traffic spikes above sustained rate limits without rejecting requests. Burst handling recognizes that real-world traffic is often bursty rather than perfectly uniform, and well-behaved clients should be able to send requests in bursts as long as their long-term average respects the configured limits.

Refill Rate: The steady rate at which tokens are added to a token bucket, typically expressed as tokens per second. The refill rate determines the sustained throughput allowed by the rate limiter - over long periods, the average request rate cannot exceed the refill rate regardless of the bucket capacity.

Boundary Condition: Edge cases where rate limiting algorithms face their greatest challenges, particularly during time window transitions. For example, a sliding window counter must carefully handle the transition from one time bucket to the next to avoid losing track of requests or allowing double the configured limit during boundary periods.

Multi-Tier Rate Limiting Concepts

Multi-Tier Rate Limiting: A hierarchical system of rate limits that enforces quotas across multiple dimensions simultaneously, such as per-user, per-IP, per-API, and global limits. Each tier represents a different scope of protection, and requests must pass all applicable tiers to be allowed.

Tier Precedence Hierarchy: The ordered evaluation system for multiple rate limit tiers, typically proceeding from most specific (per-user) to most general (global). The hierarchy determines which limits are checked first and how conflicts between tiers are resolved.

Short-Circuit Evaluation: An optimization strategy that stops tier evaluation immediately when any rate limit is exceeded, avoiding unnecessary computation and reducing latency. Once a request is denied by one tier, there is no need to check remaining tiers since the request will be rejected regardless.

Rule Pattern Matching: The process of identifying which rate limit rules apply to a specific request based on the request's context (user ID, IP address, API endpoint, etc.). Pattern matching uses regular expressions or exact string matching to determine which configured rules should be evaluated.

Key Composition: The systematic construction of Redis keys for rate limit storage by combining rule patterns with request context. Proper key composition ensures that rate limits are applied to the correct scope (e.g., "user:12345:api:/orders" for a per-user rate limit on the orders endpoint).

Distributed Systems and Consistency

Consistent Hashing: A distributed data placement strategy that minimizes data movement when nodes are added or removed from a cluster. In rate limiting, consistent hashing ensures that rate limit keys are distributed evenly across Redis nodes and that most keys remain on the same node during topology changes.

Virtual Nodes: Multiple hash positions assigned to each physical node in a consistent hash ring to improve load balancing. Virtual nodes ensure that when a physical node fails, its load is distributed across multiple remaining nodes rather than overwhelming a single neighbor.

Hash Ring: The circular hash space used in consistent hashing where keys and nodes are placed based on their hash values. The ring topology ensures that each key is assigned to the first node found by walking clockwise from the key's position.

Hot Key Detection: The process of identifying rate limit keys that receive disproportionately high access compared to others. Hot keys can overwhelm individual Redis nodes and require special handling such as replication or redistribution to maintain system performance.

Split-Brain Prevention: Strategies to ensure consistent cluster state during network partitions that might divide the cluster into multiple segments. Split-brain scenarios can lead to inconsistent rate limit decisions if different cluster segments allow requests that should collectively exceed limits.

Clock Skew: Time differences between distributed system nodes that can affect time-based rate limiting algorithms. Clock skew can cause requests to be incorrectly categorized in time windows or lead to inconsistent token bucket refill timing across different application instances.

Redis and Storage Concepts

Connection Pooling: The practice of maintaining a reusable set of Redis connections to improve performance and resource utilization. Connection pools reduce the overhead of establishing new connections for each operation while limiting the total number of connections to prevent overwhelming Redis servers.

Lua Scripting: Redis's capability to execute Lua scripts atomically on the server side, ensuring that complex operations involving multiple Redis commands are performed without interruption. In rate limiting, Lua scripts implement atomic check-and-update operations that prevent race conditions.

Failover: The automatic redirection of traffic from failed Redis nodes to healthy replicas or alternative nodes. Failover mechanisms ensure that rate limiting continues to function even when individual Redis instances become unavailable due to hardware failures or network issues.

Rebalancing: The process of redistributing rate limit state across Redis nodes to maintain optimal performance and load distribution. Rebalancing occurs when nodes are added or removed from the cluster or when hot key detection indicates uneven load distribution.

Configuration Propagation: The distribution of rate limit rule changes from the management API to all application instances. Propagation ensures that all instances use consistent rate limiting rules without requiring service restarts or manual updates.

Testing and Quality Assurance

Algorithm Unit Testing: Isolated testing of individual rate limiting algorithms to verify correctness and boundary condition handling. Unit tests validate that algorithms behave correctly under various scenarios including burst traffic, time window boundaries, and edge cases like exactly hitting rate limits.

Distributed Integration Testing: Multi-instance testing scenarios that verify cluster-wide rate limiting behavior works correctly across multiple application instances and Redis nodes. These tests validate that distributed coordination functions properly and that race conditions are prevented.

Milestone Verification Checkpoints: Quality gates that confirm implementation meets acceptance criteria before advancing to subsequent development phases. Each checkpoint includes specific tests to run and expected behaviors to verify, ensuring systematic progress toward project completion.

Chaos and Failure Testing: Systematic failure injection to validate recovery mechanisms and identify system weaknesses. Chaos testing includes simulating Redis failures, network partitions, clock skew, and other adverse conditions to verify that the system continues operating correctly.

Load Pattern: Synthetic traffic generation strategies for testing system behavior under different types of load. Patterns include constant load (steady rate), burst load (periodic spikes), gradual ramp (increasing rate), spike load (sudden increases), and random load (variable timing).

Network Partition Simulation: Controlled isolation of system components to test split-brain scenarios and validate that the system maintains consistency during network failures. Partition testing ensures that conflicting rate limit decisions don't occur when cluster segments can't communicate.

Monitoring and Operations

Correlation IDs: Unique identifiers that follow requests through distributed system components, enabling tracing of request flow across multiple services and facilitating debugging of complex interactions. Correlation IDs help operators understand how rate limiting decisions relate to specific user requests.

Structured Logging: Logging practices that use consistent formats and fields to enable automated analysis and monitoring. Structured logs facilitate debugging by providing machine-readable information about rate limiting decisions, Redis operations, and system performance.

Distributed Tracing: The practice of tracking request flow across multiple system components to understand performance characteristics and identify bottlenecks. Tracing helps operators understand the latency contribution of different rate limiting operations.

Symptom-Based Diagnosis: A troubleshooting methodology that maps observable problems to likely root causes through systematic analysis. This approach helps operators quickly identify the source of rate limiting issues based on symptoms like high latency, incorrect decisions, or system errors.

Metrics Aggregation: The collection and combination of usage statistics from multiple application instances for centralized monitoring and alerting. Aggregation provides global visibility into rate limiting effectiveness and system health.

System Architecture Components

This section defines the key data structures and interfaces that comprise the distributed rate limiting system, organized by their primary responsibilities and relationships.

Core Rate Limiting Types

Type	Purpose	Key Characteristics
`RateLimitRule`	Defines rate limiting policies	Contains pattern matching, algorithm selection, and limit configuration
`RateLimitRequest`	Represents incoming requests for rate limit evaluation	Encapsulates user context, IP address, API endpoint, and token requirements
`RateLimitResult`	Provides rate limiting decision and metadata	Includes allow/deny decision, remaining quota, and retry timing information
`RequestContext`	Extended request information with correlation data	Adds correlation IDs, headers, and timestamp for distributed tracing

The `RateLimitRule` structure serves as the foundation for policy definition, containing fields for pattern matching (`key_pattern`), algorithm selection (`algorithm`), limit values (`limit` , `burst_limit`), time windows (`window`), and operational controls (`enabled` , `priority`). The priority field enables conflict resolution when multiple rules match the same request.

The `RateLimitRequest` captures the essential information needed to make rate limiting decisions, including user identification (`user_id`), network identification (`ip_address`), resource identification (`api_endpoint`), client identification (`user_agent`), and resource consumption (`tokens`). This structure provides the input for key composition and rule matching.

The `RateLimitResult` communicates rate limiting decisions back to applications, indicating whether the request is allowed (`allowed`), how much quota remains (`remaining`), when the client should retry if denied (`retry_after`), when limits reset (`reset_time`), which rule caused the decision (`rule_id`), and which algorithm was used (`algorithm`).

Algorithm Implementation Types

Type	Purpose	Key Characteristics
`TokenBucketConfig`	Configuration for token bucket algorithm	Specifies capacity, refill rate, and time window
`TokenBucketState`	Runtime state for token bucket instances	Tracks current tokens, last refill time, and configuration
`SlidingWindowConfig`	Configuration for sliding window algorithms	Defines limit, window size, and bucket count for approximation
`SlidingWindowCounter`	Implementation of sliding window counter algorithm	Provides memory-efficient approximate rate limiting
`SlidingWindowLog`	Implementation of sliding window log algorithm	Provides precise rate limiting with timestamp tracking

The token bucket implementation uses `TokenBucketConfig` to define the bucket capacity (maximum burst), refill rate (sustained throughput), and time window for rate calculations. The `TokenBucketState` maintains runtime information including current token count, last refill timestamp, and configuration references for atomic updates.

Sliding window algorithms share a common configuration structure (`SlidingWindowConfig`) but differ in their implementation approaches. The counter variant approximates sliding windows using fixed time buckets with weighted averaging, while the log variant maintains precise timestamps for exact calculations.

Multi-Tier Evaluation Types

Type	Purpose	Key Characteristics
`MultiTierLimiter`	Coordinates evaluation across multiple rate limit tiers	Manages rule matching, tier ordering, and short-circuit evaluation
`TierEvaluation`	Records results from individual tier evaluation	Captures rule ID, tier name, result, duration, and algorithm used
`FlowResult`	Comprehensive result from multi-tier evaluation	Aggregates tier results, identifies blocking tier, and provides response headers
`KeyComposer`	Generates Redis keys from rule patterns and request context	Maintains pattern cache and handles key generation for different tiers

The `MultiTierLimiter` orchestrates the evaluation of multiple rate limiting rules against a single request. It maintains references to storage backends, rule management, key composition, algorithm implementations, and fallback mechanisms. The design enables efficient evaluation through rule pre-filtering and short-circuit logic.

`TierEvaluation` structures capture detailed information about each tier's evaluation, enabling detailed logging, metrics collection, and debugging. The `FlowResult` aggregates these individual evaluations into a comprehensive response that applications can use for decision-making and client communication.

Distributed Storage Types

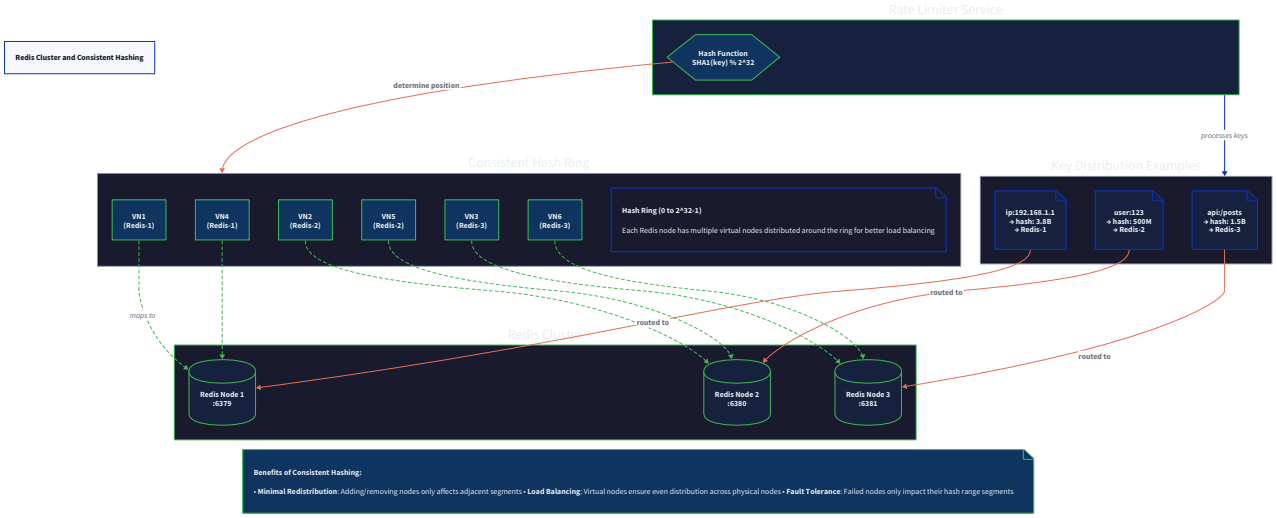
Type	Purpose	Key Characteristics
`RedisStorage`	Redis backend implementation for distributed state	Manages connections, executes Lua scripts, handles failover
`RedisConfig`	Configuration for Redis connections and behavior	Specifies addresses, authentication, timeouts, and pool settings
`DistributedLimiter`	Main entry point for distributed rate limiting	Coordinates storage, rule management, and fallback strategies
`FallbackLimiter`	Local-only rate limiting for degraded operation	Maintains local state when Redis is unavailable

The `RedisStorage` type encapsulates all Redis interactions, providing a clean interface for atomic operations while handling connection management, script execution, and error recovery. The design isolates Redis-specific concerns from algorithm logic, enabling easier testing and potential backend substitution.

`DistributedLimiter` serves as the primary interface for applications, coordinating between Redis storage, rule management, and local fallback. This design ensures that applications have a consistent interface regardless of whether the system is operating in distributed or fallback mode.

Consistent Hashing and Clustering Types

Type	Purpose	Key Characteristics
`HashRing`	Implements consistent hashing for key distribution	Manages virtual nodes, handles topology changes, minimizes redistribution
`CircuitBreaker`	Prevents cascading failures through intelligent failure detection	Tracks failure rates, implements state machine, provides fallback paths
`HealthChecker`	Monitors Redis node health and coordinates responses	Performs health checks, manages circuit breakers, handles failover
`HotKeyDetector`	Identifies disproportionately accessed rate limit keys	Analyzes access patterns, triggers replication, enables load balancing



The `HashRing` implementation uses virtual nodes to ensure even distribution of rate limit keys across Redis instances. The virtual node approach prevents hotspots that could occur if keys were distributed only based on physical nodes, especially in small clusters.

`CircuitBreaker` provides protection against cascading failures by implementing a state machine with closed (normal operation), open (all requests rejected), and half-open (limited test requests allowed) states. This pattern prevents the system from overwhelming failing components while allowing for automatic recovery.

Configuration and Management Types

Type	Purpose	Key Characteristics
`RuleManager`	Manages rate limit rule lifecycle	Handles rule loading, validation, caching, and change notification
`ConfigurationWatcher`	Monitors for configuration changes	Subscribes to Redis updates, maintains version consistency
`FlowCoordinator`	Orchestrates complete request processing	Coordinates all components for end-to-end request handling
`MetricsCollector`	Aggregates usage statistics and performance data	Collects metrics, detects patterns, provides monitoring data

The `RuleManager` centralizes all rule-related operations, providing interfaces for loading rules from configuration files, validating rule syntax and logic, maintaining rule caches for performance, and notifying other components of rule changes.

`ConfigurationWatcher` implements the observer pattern for configuration changes, using Redis pub/sub mechanisms to receive notifications when rules are updated. This design ensures that all application instances receive configuration updates promptly without requiring polling.

Time and Synchronization Types

Type	Purpose	Key Characteristics
`TimeProvider`	Abstracts time operations for testing and synchronization	Provides consistent time across distributed components
`MockTimeProvider`	Controllable time provider for testing	Enables deterministic testing of time-based algorithms
`CorrelationContext`	Provides request correlation for distributed tracing	Maintains correlation IDs and metadata across component boundaries
`CorrelationLogger`	Structured logging with correlation information	Enables efficient debugging and request tracing

The `TimeProvider` abstraction enables consistent time handling across the distributed system while supporting testing scenarios. The interface provides methods for measuring clock skew relative to Redis servers and adjusting local timestamps accordingly.

`CorrelationContext` facilitates debugging and monitoring by providing unique identifiers that follow requests through the system. This context includes trace IDs, request IDs, user IDs, session IDs, and arbitrary metadata that helps operators understand request flow.

Testing and Quality Assurance Types

Type	Purpose	Key Characteristics
`RedisTestHelper`	Utilities for Redis testing scenarios	Manages embedded Redis, clusters, network simulation
`LoadGenerator`	Generates synthetic traffic for performance testing	Supports various load patterns and measures system response
`LoadTestResults`	Aggregates load testing metrics and analysis	Captures latency, throughput, and accuracy measurements
`LoadGeneratorConfig`	Configuration for load testing scenarios	Defines traffic patterns, duration, and client behavior

The `RedisTestHelper` provides utilities for creating test environments including embedded Redis instances, Redis clusters, and network partition simulation. This tooling enables comprehensive testing without requiring external Redis infrastructure.

`LoadGenerator` supports multiple traffic patterns including constant load (steady request rate), burst load (periodic spikes), gradual ramp (linearly increasing rate), spike load (sudden increases), and random load (variable timing). Each pattern tests different aspects of the rate limiting system's behavior.

Algorithm-Specific Constants

The system defines several algorithm identifiers and configuration constants that standardize behavior across implementations:

Constant	Value	Purpose
`ALGORITHM_TOKEN_BUCKET`	"token_bucket"	Identifier for token bucket algorithm selection
`ALGORITHM_SLIDING_WINDOW_LOG`	"sliding_window_log"	Identifier for precise sliding window algorithm
`ALGORITHM_SLIDING_COUNTER`	"sliding_window_counter"	Identifier for approximate sliding window algorithm
`DEFAULT_POOL_SIZE`	10	Default number of Redis connections per pool
`DEFAULT_TIMEOUT`	5ms	Default timeout for Redis operations

Priority constants enable rule precedence configuration:

Constant	Value	Purpose
`PRIORITY_HIGH`	100	High priority for critical rate limiting rules
`PRIORITY_LOW`	1	Low priority for general rate limiting rules

Circuit breaker states manage failure handling:

Constant	Purpose	Characteristics
`CircuitClosed`	Normal operation state	All requests are processed normally
`CircuitOpen`	All requests rejected state	Requests fail fast without attempting operations
`CircuitHalfOpen`	Limited test requests allowed state	Some requests are tested to determine if service has recovered

Interface Definitions

The system defines several key interfaces that enable modularity and testing. These interfaces abstract the essential operations while allowing for multiple implementations.

Core Rate Limiting Interfaces

Method	Parameters	Returns	Description
`Check`	`ctx context.Context, req RateLimitRequest`	`*RateLimitResult, error`	Performs rate limit check and updates counters atomically
`Preview`	`ctx context.Context, req RateLimitRequest`	`*RateLimitResult, error`	Checks rate limit status without updating counters
`Reset`	`ctx context.Context, req RateLimitRequest`	`error`	Clears rate limit counters for the specified request context

The `Check` method represents the primary rate limiting operation, atomically evaluating whether a request should be allowed and updating counters if so. The `Preview` method enables applications to check rate limit status without consuming quota, useful for displaying remaining limits to users.

Storage Backend Interfaces

Method	Parameters	Returns	Description
`CheckAndUpdate`	`ctx context.Context, key string, limit int64, window time.Duration`	`bool, int64, time.Time, error`	Atomically checks and updates rate limit counters
`ExecuteLua`	`ctx context.Context, script string, keys []string, args []interface{}`	`interface{}, error`	Executes Redis Lua script atomically
`GetNode`	`key string`	`string, bool`	Returns responsible node for key in sharded setup

The storage interface abstracts Redis operations to enable testing with mock backends and potential future support for alternative storage systems. The `ExecuteLua` method enables atomic operations by running Lua scripts on Redis servers.

Rule Management Interfaces

Method	Parameters	Returns	Description
`LoadRules`	`configPath string`	`error`	Loads rate limit rules from configuration file
`GetMatchingRules`	`userID, ipAddress, apiEndpoint string`	`[]*RateLimitRule`	Returns rules matching request context
`ComposeKey`	`rule *RateLimitRule, req *RateLimitRequest`	`string, error`	Generates Redis key from rule pattern and request context

Rule management interfaces enable dynamic rule loading and efficient rule matching. The `GetMatchingRules` method filters the complete rule set to only those applicable to a specific request, optimizing evaluation performance.

Error Handling and State Management

The system defines comprehensive error handling strategies and state management approaches to ensure reliable operation under various failure conditions.

Error Categories

Error Type	Detection Method	Recovery Strategy
Redis Connection Failures	Connection timeout or network errors	Automatic failover to backup Redis nodes or local fallback
Clock Skew Exceeding Tolerance	Time synchronization checks	Log warnings and adjust calculations using measured skew
Configuration Parsing Errors	Schema validation during rule loading	Reject invalid configuration and continue with previous valid rules
Hot Key Overload	Access pattern analysis	Replicate hot keys across multiple nodes or apply specialized handling
Lua Script Execution Failures	Redis error responses	Retry with exponential backoff or fall back to local limiting

State Machine Transitions

Circuit breaker state management follows a well-defined state machine to provide predictable failure handling:

Current State	Event	Next State	Action Taken
`CircuitClosed`	Failure threshold exceeded	`CircuitOpen`	Begin rejecting all requests immediately
`CircuitOpen`	Recovery timeout elapsed	`CircuitHalfOpen`	Allow limited test requests to probe service health
`CircuitHalfOpen`	Test request succeeds	`CircuitClosed`	Resume normal operation with full request processing
`CircuitHalfOpen`	Test request fails	`CircuitOpen`	Return to rejecting all requests and reset recovery timer

This state machine prevents cascading failures by quickly detecting problems and avoiding unnecessary load on failing components while providing automatic recovery when services become healthy again.

Performance and Scalability Considerations

The distributed rate limiting system incorporates several design patterns and optimizations to achieve high performance and horizontal scalability.

Memory Usage Optimization

Different rate limiting algorithms have distinct memory characteristics that affect their suitability for different use cases:

Algorithm	Memory Per Key	Accuracy	Best Use Case
Token Bucket	O(1) - constant	Exact for burst, approximate for sustained	General-purpose rate limiting with burst support
Sliding Window Counter	O(1) - constant	Approximate (can allow 2x limit at boundaries)	High-scale deployments requiring memory efficiency
Sliding Window Log	O(n) - proportional to requests	Exact	Low-to-medium scale deployments requiring precision

The constant memory algorithms (token bucket and sliding window counter) enable the system to support millions of rate-limited keys without running out of memory, making them suitable for large-scale deployments.

Latency Optimization Strategies

The system employs several strategies to minimize rate limiting latency:

Optimization	Technique	Benefit
Connection Pooling	Reuse Redis connections across requests	Eliminates connection establishment overhead
Lua Script Preloading	Cache compiled Lua scripts in Redis	Reduces script compilation time for each request
Short-Circuit Evaluation	Stop checking tiers after first denial	Reduces computation and network calls
Local Rule Caching	Cache rule matching results locally	Eliminates repeated pattern matching computation
Asynchronous Metrics Collection	Decouple metrics from request path	Prevents metrics overhead from affecting latency

Common Implementation Patterns

Several design patterns appear throughout the distributed rate limiting system, providing consistency and reliability across different components.

Observer Pattern for Configuration Updates

The system uses the observer pattern to propagate configuration changes from the management API to all application instances. The `ConfigurationWatcher` subscribes to Redis pub/sub channels, receives change notifications, and triggers local rule reloading. This pattern ensures that configuration updates are applied consistently across the cluster without requiring service restarts.

Strategy Pattern for Algorithm Selection

Rate limiting algorithms are implemented using the strategy pattern, enabling runtime selection based on rule configuration. Each algorithm implements a common interface while providing different trade-offs between accuracy, memory usage, and performance. This design allows the same infrastructure to support multiple algorithms simultaneously.

Circuit Breaker Pattern for Failure Isolation

Circuit breakers provide failure isolation by monitoring error rates and preventing requests to failing components. The pattern includes timeout mechanisms, retry logic, and automatic recovery detection. This approach prevents cascading failures and improves overall system resilience.

Template Method Pattern for Request Processing

The request processing flow uses the template method pattern to standardize the steps while allowing customization for different scenarios. The flow includes request validation, rule matching, tier evaluation, storage operations, and response generation. Each step can be customized while maintaining consistent overall behavior.

Implementation Guidance

This section provides practical guidance for implementing the distributed rate limiting system, including technology recommendations, code organization, and development best practices.

Technology Recommendations

Component	Simple Option	Advanced Option
Storage Backend	Redis single instance with persistence	Redis Cluster with replication
Configuration	YAML files with file watching	Redis-backed configuration with pub/sub updates
Monitoring	Prometheus metrics with Grafana	Distributed tracing with Jaeger plus metrics
Testing	Embedded Redis with unit tests	Docker Compose with integration tests
Deployment	Single process with Redis connection	Kubernetes with Redis Operator

Recommended Module Structure

The following directory structure organizes the codebase into logical modules that align with the system's architectural components:

```
distributed-rate-limiter/
├── cmd/
│   ├── server/main.go              # HTTP API server entry point
│   └── cli/main.go                 # Command-line management tools
├── internal/
│   ├── algorithms/
│   │   ├── token_bucket.go         # Token bucket implementation
│   │   ├── sliding_window.go       # Sliding window implementations
│   │   └── algorithm_test.go       # Algorithm unit tests
│   ├── storage/
│   │   ├── redis.go                # Redis storage implementation
│   │   ├── fallback.go             # Local fallback storage
│   │   └── storage_test.go         # Storage integration tests
│   ├── config/
│   │   ├── rule_manager.go         # Rule loading and management
│   │   ├── watcher.go              # Configuration change monitoring
│   │   └── config_test.go          # Configuration tests
│   ├── ratelimit/
│   │   ├── distributed_limiter.go  # Main rate limiting logic
│   │   ├── multi_tier.go           # Multi-tier evaluation
│   │   └── ratelimit_test.go       # Core logic tests
│   ├── cluster/
│   │   ├── hash_ring.go            # Consistent hashing implementation
│   │   ├── health_checker.go       # Node health monitoring
│   │   └── cluster_test.go         # Clustering tests
│   └── api/
│       ├── handlers.go             # HTTP request handlers
│       ├── middleware.go           # Rate limiting middleware
│       └── api_test.go             # API integration tests
├── pkg/
│   ├── metrics/
│   │   ├── collector.go            # Metrics collection interfaces
│   │   └── prometheus.go           # Prometheus metrics implementation
│   └── logging/
│       ├── correlation.go          # Correlation context and logging
│       └── structured.go           # Structured logging utilities
├── test/
│   ├── fixtures/
│   │   ├── rules.yaml              # Test rate limiting rules
│   │   └── config.yaml             # Test configuration
│   ├── helpers/
│   │   ├── redis_helper.go         # Redis testing utilities
│   │   └── load_generator.go       # Load testing tools
│   └── integration/
│       ├── distributed_test.go     # Multi-instance integration tests
│       └── chaos_test.go           # Failure injection tests
├── configs/
│   ├── rules.yaml                  # Production rate limiting rules
│   └── server.yaml                 # Server configuration
├── scripts/
│   ├── lua/
│   │   ├── token_bucket.lua        # Token bucket Lua script
│   │   └── sliding_window.lua      # Sliding window Lua script
│   └── deploy/
│       ├── docker-compose.yaml     # Development environment
│       └── kubernetes/             # Kubernetes deployment manifests
└── docs/
    ├── api.md                      # API documentation
    ├── algorithms.md               # Algorithm documentation
    └── operations.md               # Operational procedures
```

This structure separates concerns clearly while maintaining logical grouping. The `internal/` directory contains private implementation details, while `pkg/` contains reusable packages that could be imported by other projects.

Core Interface Implementation

The following skeleton provides the foundation for implementing the main rate limiting interface:

```go
// Package ratelimit provides distributed rate limiting capabilities
package ratelimit

import (
    "context"
    "time"
)

// DistributedLimiter coordinates rate limiting across multiple application instances
type DistributedLimiter struct {
    storage       Storage
    ruleManager   *config.RuleManager
    localFallback Limiter
    algorithms    map[string]Algorithm
    keyComposer   *KeyComposer
    metrics       *metrics.Collector
}

// NewDistributedLimiter creates a new distributed rate limiter instance
func NewDistributedLimiter(storage Storage, ruleManager *config.RuleManager) *DistributedLimiter {
    // TODO 1: Initialize algorithm map with token bucket, sliding window implementations
    // TODO 2: Create key composer for generating Redis keys from rule patterns
    // TODO 3: Initialize local fallback limiter for degraded operation
    // TODO 4: Set up metrics collector for monitoring and debugging
    // TODO 5: Return configured DistributedLimiter instance
}

// Check performs rate limit evaluation and updates counters atomically
func (d *DistributedLimiter) Check(ctx context.Context, req RateLimitRequest) (*RateLimitResult, error) {
    // TODO 1: Extract correlation context for distributed tracing
    // TODO 2: Get matching rules based on request context (user, IP, API endpoint)
    // TODO 3: Evaluate all applicable tiers with short-circuit logic
    // TODO 4: For each tier, compose Redis key and select appropriate algorithm
    // TODO 5: Execute rate limit check using selected algorithm and storage backend
    // TODO 6: If any tier denies request, return denial with retry-after timing
    // TODO 7: If all tiers pass, return success with remaining quota information
    // TODO 8: Handle Redis failures by falling back to local rate limiting
    // TODO 9: Record metrics for monitoring and hot key detection
    // TODO 10: Log rate limiting decision with correlation context for debugging
```

```
}
```

Redis Storage Implementation Skeleton

The Redis storage backend requires careful implementation of atomic operations and connection management:

```go
// Package storage provides distributed storage backends for rate limiting
package storage

import (
    "context"
    "time"
    "github.com/go-redis/redis/v8"
)

// RedisStorage implements distributed rate limiting storage using Redis
type RedisStorage struct {
    client redis.UniversalClient
    config RedisConfig
    scripts map[string]*redis.Script
    circuitBreaker *CircuitBreaker
}

// NewRedisStorage creates Redis storage with connection pooling and health checking
func NewRedisStorage(config RedisConfig) (*RedisStorage, error) {
    // TODO 1: Create Redis universal client with cluster or sentinel support
    // TODO 2: Configure connection pool with appropriate size and timeouts
    // TODO 3: Load and register Lua scripts for atomic rate limit operations
    // TODO 4: Initialize circuit breaker for failure protection
    // TODO 5: Perform initial connectivity test and return configured storage
}

// CheckAndUpdate atomically checks rate limit and updates counters
func (r *RedisStorage) CheckAndUpdate(ctx context.Context, key string, limit int64, window time.Duration) (bool, int64,
time.Time, error) {
    // TODO 1: Check circuit breaker state before attempting Redis operation
    // TODO 2: Select appropriate Lua script based on algorithm requirements
    // TODO 3: Prepare script arguments including current timestamp and window
    // TODO 4: Execute Lua script with proper error handling and retries
    // TODO 5: Parse script result to determine allow/deny decision
    // TODO 6: Calculate remaining quota and reset time from Redis response
    // TODO 7: Update circuit breaker state based on operation success/failure
    // TODO 8: Return rate limiting decision with metadata for client response
}
```

Load Testing and Validation Tools

Comprehensive testing requires tools for validating distributed behavior under various conditions:

```go
// Package testhelpers provides utilities for testing distributed rate limiting
package testhelpers

import (
    "context"
    "time"
    "sync"
)

// LoadGenerator produces synthetic traffic for testing rate limiting behavior
type LoadGenerator struct {
    config    LoadGeneratorConfig
    limiter   ratelimit.DistributedLimiter
    results   *LoadTestResults
    ctx       context.Context
    cancelFn  context.CancelFunc
}

// Start begins load generation according to configured pattern
func (lg *LoadGenerator) Start() *LoadTestResults {
    // TODO 1: Initialize results tracking with thread-safe counters
    // TODO 2: Create context with cancellation for stopping load generation
    // TODO 3: Launch worker goroutines based on configured client count
    // TODO 4: Coordinate traffic pattern (constant, burst, ramp, spike, random)
    // TODO 5: Execute rate limit checks with latency measurement
    // TODO 6: Record results including success/failure counts and timing
    // TODO 7: Monitor for test completion or cancellation signal
    // TODO 8: Aggregate final results and return comprehensive report
}

// generateConstantLoad produces steady request rate for baseline testing
func (lg *LoadGenerator) generateConstantLoad() {
    // TODO 1: Calculate inter-request delay for desired requests per second
    // TODO 2: Create ticker for maintaining steady request rate
    // TODO 3: Execute requests in loop until context cancellation
    // TODO 4: Maintain request timing accuracy despite processing variations
    // TODO 5: Record each request result with timestamp for analysis
}
```

Debugging and Troubleshooting Utilities

Debugging distributed rate limiting requires specialized tools for analyzing Redis state and request flow:

```go
// Package debugging provides utilities for troubleshooting distributed rate limiting
package debugging

import (
    "context"
    "time"
)

// RedisDebugger provides Redis-specific debugging capabilities
type RedisDebugger struct {
    client redis.UniversalClient
    logger *CorrelationLogger
}

// DiagnoseKeyState provides comprehensive analysis of rate limiting key
func (rd *RedisDebugger) DiagnoseKeyState(ctx context.Context, key string) (*KeyDiagnostics, error) {
    // TODO 1: Check if key exists in Redis and retrieve TTL
    // TODO 2: Measure memory usage for the key using MEMORY USAGE command
    // TODO 3: Determine which Redis node(s) contain the key in cluster setup
    // TODO 4: Measure access latency by performing test operations
    // TODO 5: Analyze key access patterns from Redis slow log if available
    // TODO 6: Return comprehensive diagnostics for troubleshooting
}

// MeasureClusterHealth assesses overall Redis cluster health and performance
func (rd *RedisDebugger) MeasureClusterHealth(ctx context.Context) (*ClusterHealth, error) {
    // TODO 1: Enumerate all Redis nodes in cluster configuration
    // TODO 2: Check connectivity and responsiveness for each node
    // TODO 3: Measure latency and memory usage across all nodes
    // TODO 4: Analyze key distribution balance using CLUSTER NODES
    // TODO 5: Identify potential hotspots or performance issues
    // TODO 6: Return cluster health report with recommendations
}
```

Milestone Verification Checkpoints

After implementing each major component, verify functionality using these checkpoints:

Milestone 1 - Algorithm Implementation:

- Run `go test ./internal/algorithms/...` to verify algorithm correctness
- Execute load tests with different traffic patterns to validate burst handling

- Test boundary conditions at window transitions and capacity limits
- Verify token bucket allows configured burst size followed by steady rate
- Confirm sliding window algorithms respect configured request limits

Milestone 2 - Multi-Tier Evaluation:

- Test per-user, per-IP, and global rate limits simultaneously
- Verify short-circuit evaluation stops at first tier denial
- Validate rule precedence handling when multiple rules match
- Test burst allowance behavior across different tiers
- Confirm API endpoint rate limiting works correctly

Milestone 3 - Redis Integration:

- Start Redis instance and verify Lua script execution
- Test atomic check-and-update operations under concurrent load
- Simulate Redis failures and verify graceful degradation to local fallback
- Validate connection pooling reduces latency compared to per-request connections
- Test Redis cluster configuration with multiple nodes

Milestone 4 - Consistent Hashing:

- Add and remove Redis nodes while monitoring key redistribution
- Generate synthetic hot keys and verify detection and replication
- Test load balancing across nodes with different key distributions
- Simulate node failures and verify automatic failover behavior
- Validate virtual nodes improve distribution compared to simple hashing

Milestone 5 - Management API:

- Create, update, and delete rate limit rules through REST API
- Verify configuration changes propagate to all application instances
- Test real-time dashboard shows current usage and remaining quotas
- Validate rate limit headers in HTTP responses match RFC standards
- Ensure management API itself is protected by rate limiting

Each checkpoint should include specific commands to run, expected outputs, and signs that indicate proper functionality or potential issues requiring investigation.