Load Balancer: Design Document

Overview

This system implements an HTTP application load balancer that distributes incoming requests across multiple backend servers using various algorithms like round-robin, weighted distribution, and least connections. The key architectural challenge is maintaining high availability and even traffic distribution while handling backend failures gracefully through health checking and automatic failover.

> 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 (1-4) — this establishes the foundational understanding needed throughout the project

Mental Model: Restaurant Host

Think of a **load balancer** as a restaurant host who greets customers at the entrance and decides where to seat them. When diners arrive, they don't randomly wander around looking for empty tables — that would create chaos, with some sections overwhelmed while others sit empty. Instead, the host has a clear view of the entire restaurant, knows which tables are available, which servers are busy, and how to distribute customers evenly for the best dining experience.

The host uses different strategies depending on the situation. During quiet periods, they might simply rotate between sections in order — the **round-robin** approach. When one section gets backed up because a server called in sick, the host routes new customers elsewhere — this is **health-aware routing**. For VIP customers who prefer their regular server, the host remembers their preference and seats them consistently — this is **session affinity**. If some servers handle large parties better than others, the host might send more customers to those sections — this is **weighted distribution**.

Just like the restaurant host, a load balancer sits between clients (diners) and backend servers (restaurant sections). It receives incoming requests, chooses which backend should handle each one, and ensures no single server becomes overwhelmed while others remain idle. The load balancer must track which backends are healthy and responsive, maintain fairness in distribution, and gracefully handle situations where backends fail or become unavailable.

> The key insight is that load balancing is fundamentally about **coordination and optimization** — taking a stream of independent requests and making intelligent routing decisions to maximize overall system performance and reliability.

The Distribution Challenge

Building an effective load balancer requires solving several interconnected challenges that go beyond simply forwarding requests. Each challenge represents a fundamental problem in distributed systems that affects both performance and reliability.

Request Distribution Fairness represents the core algorithmic challenge. When requests arrive at unpredictable rates and backend servers have different processing capabilities, how do we ensure fair distribution? A naive approach might assign requests randomly, but this can lead to significant imbalances over short time periods. Some requests might be computationally expensive while others are simple database lookups, making simple request counting insufficient. The load balancer must implement algorithms that account for these variables while remaining efficient enough to route thousands of requests per second without becoming a bottleneck itself.

Backend Failure Detection and Recovery introduces the reliability challenge. Backend servers fail in various ways — they might crash completely, become unreachable due to network partitions, or degrade slowly under load while still accepting connections. The load balancer must distinguish between temporary hiccups and genuine failures, avoiding the mistake of removing healthy backends due to transient network issues. Equally important is the recovery process: when should a previously failed backend be given another chance, and how can the system avoid overwhelming a recovering server with a sudden flood of traffic?

State Management Under Concurrency presents the consistency challenge. Modern load balancers handle hundreds or thousands of concurrent requests, each potentially requiring updates to shared state like backend health status, connection counts, or round-robin counters. Race conditions can lead to uneven distribution, incorrect health states, or even crashes. The challenge is maintaining consistency without introducing locks that could serialize request processing and destroy throughput. Atomic operations, lock-free data structures, and careful state design become critical for high-performance operation.

Graceful Degradation During Failures defines the availability challenge. What happens when all backends in a pool become unhealthy? Should the load balancer return errors immediately, attempt to route to unhealthy backends anyway, or maintain some form of circuit breaker? The system must fail gracefully rather than catastrophically, providing meaningful error responses to clients while continuing to monitor for backend recovery. This requires sophisticated error handling that distinguishes between different failure modes and responds appropriately to each.

Challenge	Impact	Key Considerations	Common Failure Modes
Request Distribution	Performance, Fairness	Algorithm choice, Request characteristics, Backend capacity	Hot spots, Uneven distribution, Algorithm bias
Failure Detection	Availability, Reliability	Health check frequency, False positive rate, Recovery timing	Cascading failures, Premature removal, Thundering herd
State Management	Consistency, Throughput	Concurrency control, Data races, Lock contention	Race conditions, Stale state, Deadlocks
Graceful Degradation	User Experience, System Stability	Error responses, Fallback strategies, Circuit breaking	Error storms, Resource exhaustion, Total outage

The Observability Challenge emerges as systems scale. When request distribution becomes uneven or backends start failing, operators need visibility into the load balancer's decision-making process. Which algorithm is active? How are requests being distributed? What's the current health state of each backend? Are any backends consistently slower than others? Without proper logging and metrics, debugging load balancer issues becomes nearly impossible, especially under production load when problems manifest most acutely.

Load Balancing Landscape

The load balancing ecosystem offers numerous approaches, each with distinct trade-offs that influence architecture decisions. Understanding these options helps contextualize our design choices and explains why certain approaches are better suited for different scenarios.

Hardware vs Software Load Balancers represents the fundamental deployment choice. Hardware load balancers like F5 BigIP or Citrix ADX offer dedicated processing power, specialized network interfaces, and vendor support, but they're expensive, difficult to scale horizontally, and create vendor lock-in. They excel in environments requiring extremely high throughput with predictable workloads and established budgets. Software load balancers like HAProxy, NGINX, or our implementation run on commodity hardware, offer greater flexibility, and integrate easily with modern deployment pipelines. They sacrifice some raw performance for operational agility and cost effectiveness.

Layer 4 vs Layer 7 Load Balancing defines the protocol awareness level. Layer 4 (transport layer) load balancers make routing decisions based on IP addresses and port numbers, operating at the TCP/UDP connection level. They're fast because they don't need to parse application protocols, but they can't make content-aware routing decisions. Layer 7 (application layer) load balancers understand HTTP headers, URLs, cookies, and request content, enabling sophisticated routing rules based on user type, API version, or request characteristics. Our implementation operates at Layer 7, parsing HTTP requests to enable features like health checks via HTTP endpoints and potential future enhancements like path-based routing.

> **Decision: Layer 7 HTTP Load Balancer**
>
> - **Context**: We need to choose the protocol layer at which our load balancer operates, balancing simplicity with functionality
> - **Options Considered**: Layer 4 TCP proxy, Layer 7 HTTP proxy, Hybrid approach
> - **Decision**: Layer 7 HTTP load balancer
> - **Rationale**: HTTP awareness enables health checks via HTTP endpoints, better debugging through request logging, and extensibility for future features like path-based routing. The performance overhead is acceptable for learning purposes and typical application workloads
> - **Consequences**: More complex implementation due to HTTP parsing, but greater flexibility and observability

Approach	Performance	Features	Complexity	Use Cases
Layer 4	Very High	Basic routing, port-based	Low	High throughput, simple routing needs
Layer 7	High	Content-aware, rich routing	Medium	Application load balancing, microservices
Hybrid	Variable	Best of both	High	Complex applications with mixed requirements

Algorithm Sophistication varies dramatically across implementations. Simple algorithms like random selection or round-robin are easy to implement and understand but may not handle real-world traffic patterns well. Weighted algorithms account for server capacity differences. Least connections adapts to varying request processing times. Hash-based algorithms provide session affinity. Advanced algorithms like least response time or adaptive algorithms adjust to real-time performance metrics. Our implementation focuses on the most common algorithms that demonstrate key concepts while remaining implementable by intermediate developers.

Deployment Patterns influence architecture requirements. Reverse proxy deployment places the load balancer between clients and servers, typically in a DMZ or edge network. Direct server return (DSR) allows backends to respond directly to clients, reducing load balancer bandwidth requirements but complicating network configuration. Service mesh deployment integrates load balancing with service discovery and inter-service communication. Our design assumes reverse proxy deployment, the most common and straightforward pattern for application load balancing.

Existing Solutions Comparison helps position our implementation:

Solution	Strengths	Weaknesses	Best For
HAProxy	High performance, feature rich, proven	Complex configuration, steep learning curve	Production environments, high throughput
NGINX	Web server integration, good performance	Complex for pure load balancing, licensing costs	Web applications, content serving
Cloud Load Balancers	Managed service, auto-scaling, integration	Vendor lock-in, cost, limited customization	Cloud-native applications, managed operations
Envoy Proxy	Modern architecture, service mesh ready	Complex, resource intensive	Microservices, service mesh deployments
Our Implementation	Educational focus, clear code, customizable	Limited features, not production-hardened	Learning, prototyping, custom requirements

> The load balancing landscape reveals a fundamental tension between **simplicity and sophistication**. Simple solutions are easier to understand, debug, and maintain, but they may not handle complex real-world scenarios effectively. Our implementation deliberately chooses educational clarity over feature completeness, providing a solid foundation for understanding core concepts.

Configuration and Management approaches vary from static configuration files to dynamic service discovery integration. Static configuration is simple and predictable but requires manual updates when backends change. Dynamic configuration through APIs or service discovery systems enables automated scaling and deployment but introduces additional complexity and failure modes. Our design supports both approaches, starting with static configuration for simplicity while enabling API-based updates for operational flexibility.

The choice of implementation language significantly impacts the final system's characteristics. Go offers excellent HTTP handling, built-in concurrency, and deployment simplicity. Python provides rapid development and extensive libraries but may have performance limitations under high load. JavaScript (Node.js) enables full-stack teams to maintain the load balancer but has different concurrency characteristics. Our primary Go implementation balances performance, clarity, and learning accessibility while supporting other languages for different learning objectives.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
HTTP Server	Go `net/http`	Custom HTTP handling	Standard library provides robust foundation
HTTP Client	Go `net/http.Client` with connection pooling	Third-party HTTP client	Built-in pooling handles most needs efficiently
Configuration	JSON file with `encoding/json`	YAML with validation library	JSON is simple and well-supported
Concurrency	Go goroutines with `sync` package	Lock-free data structures	Standard approaches are sufficient for learning
Logging	Go `log` package	Structured logging (logrus, zap)	Simple logging aids debugging without complexity
Testing	Go `testing` with `net/http/httptest`	Third-party testing frameworks	Standard library provides excellent HTTP testing

Recommended Project Structure

Organize your load balancer project to separate concerns clearly and enable incremental development across milestones:

```
load-balancer/
├── cmd/
│   └── loadbalancer/
│       └── main.go                 ← Entry point, CLI parsing, startup
├── internal/
│   ├── proxy/
│   │   ├── proxy.go                ← HTTP reverse proxy (Milestone 1)
│   │   ├── proxy_test.go           ← Unit tests for proxy logic
│   │   └── headers.go              ← Header manipulation utilities
│   ├── backend/
│   │   ├── pool.go                 ← Backend pool management (Milestone 2)
│   │   ├── backend.go              ← Backend server model
│   │   └── health.go               ← Health checking (Milestone 3)
│   ├── algorithm/
│   │   ├── interface.go            ← Load balancing algorithm interface
│   │   ├── roundrobin.go           ← Round robin implementation (Milestone 2)
│   │   ├── weighted.go             ← Weighted round robin (Milestone 4)
│   │   ├── leastconn.go            ← Least connections (Milestone 4)
│   │   └── hash.go                 ← IP hash algorithm (Milestone 4)
│   ├── config/
│   │   ├── config.go               ← Configuration loading and validation
│   │   └── reload.go               ← Hot reload functionality
│   └── router/
│       ├── router.go               ← Main request router coordination
│       └── errors.go               ← Error response handling
├── configs/
│   ├── example.json                ← Example configuration
│   └── test.json                   ← Configuration for testing
├── test/
│   ├── integration/                ← End-to-end integration tests
│   └── backends/                   ← Test HTTP servers for development
├── docs/                           ← Documentation and design notes
├── go.mod
├── go.sum
└── README.md
```

Infrastructure Starter Code

Configuration Loading Infrastructure (complete implementation):

```go
// internal/config/config.go

package config


import (

    "encoding/json"

    "fmt"

    "os"

    "time"

)


type Config struct {

    Port            int               `json:"port"`

    Backends        []BackendConfig   `json:"backends"`

    Algorithm       string            `json:"algorithm"`

    HealthCheck     HealthCheckConfig `json:"health_check"`

}


type BackendConfig struct {

    URL    string `json:"url"`

    Weight int    `json:"weight"`

}


type HealthCheckConfig struct {

    Enabled         bool          `json:"enabled"`

    Interval        time.Duration `json:"interval"`

    Timeout         time.Duration `json:"timeout"`

    HealthyThreshold   int        `json:"healthy_threshold"`

    UnhealthyThreshold int        `json:"unhealthy_threshold"`

    Path            string        `json:"path"`

}


func LoadConfig(filename string) (*Config, error) {

    data, err := os.ReadFile(filename)

    if err != nil {

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

    }


    var config Config

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

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

    }


    if err := config.Validate(); err != nil {
```

```
        return nil, fmt.Errorf("invalid configuration: %w", err)
    }

    return &config, nil
}

func (c *Config) Validate() error {
    if c.Port <= 0 || c.Port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535")
    }
    if len(c.Backends) == 0 {
        return fmt.Errorf("at least one backend must be configured")
    }
    // Additional validation logic here
    return nil
}
```

HTTP Test Server Infrastructure (for development and testing):

GO

```go
// test/backends/server.go

package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"
)

type Response struct {
    Server    string    `json:"server"`
    Timestamp time.Time `json:"timestamp"`
    Path      string    `json:"path"`
}

func main() {
    port := 8081
    if len(os.Args) > 1 {
        if p, err := strconv.Atoi(os.Args[1]); err == nil {
            port = p
        }
    }

    serverName := fmt.Sprintf("backend-%d", port)

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        response := Response{
            Server:    serverName,
            Timestamp: time.Now(),
            Path:      r.URL.Path,
        }

        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(response)
        log.Printf("Handled request: %s %s", r.Method, r.URL.Path)
    })

    http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
```

```
        w.WriteHeader(http.StatusOK)
        fmt.Fprint(w, "OK")
    })

    log.Printf("Starting backend server %s on port %d", serverName, port)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d", port), nil))
}
```

Core Implementation Skeletons

Main Entry Point Structure:

```go
// cmd/loadbalancer/main.go
package main

import (
    "flag"
    "log"
    // TODO: Import your internal packages
)

func main() {
    configFile := flag.String("config", "configs/example.json", "Configuration file path")
    flag.Parse()

    // TODO 1: Load configuration using config.LoadConfig()
    // TODO 2: Create backend pool with loaded configuration
    // TODO 3: Initialize health checker if enabled in configuration
    // TODO 4: Create algorithm instance based on configuration
    // TODO 5: Initialize HTTP proxy with backend pool and algorithm
    // TODO 6: Start HTTP server on configured port
    // TODO 7: Set up graceful shutdown handling (SIGINT/SIGTERM)
    // TODO 8: Start background health checking goroutine

    log.Println("Load balancer started successfully")
    // TODO 9: Block waiting for shutdown signal
}
```

Algorithm Interface Definition:

```go
// internal/algorithm/interface.go

package algorithm


import "internal/backend"


type Algorithm interface {

    // SelectBackend chooses a backend server for the next request

    // Returns nil if no healthy backends are available

    SelectBackend(backends []*backend.Backend) *backend.Backend



    // Name returns the algorithm name for logging and configuration

    Name() string

}


// TODO: Implement this interface in roundrobin.go, weighted.go, etc.

// Each algorithm should handle the case where no healthy backends exist

// Thread safety is required - multiple goroutines will call SelectBackend concurrently
```

Language-Specific Development Hints

Go-Specific Implementation Tips:

- Use `sync/atomic` package for lock-free counters in round-robin algorithm: `atomic.AddUint64(&counter, 1)`
- HTTP connection pooling is automatic with `http.Client` - configure `MaxIdleConns` and `IdleConnTimeout`
- Use `context.Context` with timeouts for health checks: `ctx, cancel := context.WithTimeout(context.Background(), timeout)`
- Channel-based shutdown signaling: `make(chan os.Signal, 1)` with `signal.Notify()`
- Avoid holding locks during HTTP requests - they can block for seconds under load

Common Go Patterns for Load Balancers:

```go
// Atomic counter for round-robin
var counter uint64
next := atomic.AddUint64(&counter, 1)
index := int(next % uint64(len(backends)))

// HTTP client with connection pooling
client := &http.Client{
    Transport: &http.Transport{
        MaxIdleConns:    100,
        IdleConnTimeout: 90 * time.Second,
    },
    Timeout: 30 * time.Second,
}

// Non-blocking health check channel pattern
type HealthResult struct {
    Backend *Backend
    Healthy bool
}

results := make(chan HealthResult, len(backends))
for _, backend := range backends {
    go func(b *Backend) {
        // Perform health check
        results <- HealthResult{Backend: b, Healthy: isHealthy}
    }(backend)
}
```

Milestone Development Checkpoints

Milestone 1 Checkpoint - HTTP Proxy Foundation: After implementing basic reverse proxy functionality:

```bash
# Terminal 1: Start test backend
go run test/backends/server.go 8081

# Terminal 2: Start load balancer
go run cmd/loadbalancer/main.go -config configs/single-backend.json

# Terminal 3: Test proxying
curl -v http://localhost:8080/test
# Expected: Response from backend-8081 with timestamp and path
# Expected: X-Forwarded-For and X-Forwarded-Proto headers in backend logs
```

Verification Steps:

1. Check that request method, path, and headers are preserved
2. Verify response status code and body are returned correctly

3. Confirm connection errors return 502 Bad Gateway
4. Validate that logs show request details and response status

Milestone 2 Checkpoint - Round Robin Distribution:

```bash
# Start multiple backends
go run test/backends/server.go 8081 &
go run test/backends/server.go 8082 &
go run test/backends/server.go 8083 &

# Start load balancer with round-robin configuration
go run cmd/loadbalancer/main.go -config configs/round-robin.json

# Test distribution
for i in {1..9}; do curl http://localhost:8080/ | jq .server; done
# Expected output: backend-8081, backend-8082, backend-8083, backend-8081, ...
```

Signs of Issues:

- Uneven distribution suggests race conditions in counter increment
- Always hitting same backend indicates modulo calculation error
- Requests failing randomly suggests backend list synchronization issues

Goals and Non-Goals

> **Milestone(s):** All milestones (1-4) — this section establishes the scope and boundaries that guide implementation decisions throughout the entire project

Mental Model: Construction Blueprint

Think of this section as the construction blueprint and building permit for our load balancer project. Just as a construction project needs clear specifications about what will be built (a three-story house with specific rooms) and what won't be built (no swimming pool or garage), our load balancer needs explicit boundaries. The blueprint prevents scope creep during construction — when the contractor suggests adding a hot tub, you can point to the blueprint and say "that's not in scope." Similarly, when implementing the load balancer, clear goals prevent feature creep that could derail the learning objectives.

The building permit analogy extends further: permits specify safety requirements (the building must withstand certain wind loads), quality standards (electrical work must meet code), and timeline expectations (completion within 6 months). Our quality requirements serve the same function — they define performance targets, reliability expectations, and maintainability standards that guide implementation decisions without getting lost in endless optimization.

Functional Requirements

The functional requirements define the core capabilities that our load balancer must deliver to be considered complete and useful. These requirements map directly to the project milestones and represent the minimum viable functionality needed for a production-ready basic load balancer.

Core Request Distribution

The load balancer must accept incoming HTTP requests on a configurable port and distribute them across multiple backend servers. This fundamental capability forms the foundation of all other features and establishes the reverse proxy pattern that enables traffic distribution.

Requirement	Description	Acceptance Criteria	Milestone
HTTP Request Acceptance	Accept and parse incoming HTTP requests	All HTTP methods, headers, and body content preserved	1
Request Forwarding	Forward requests to backend servers	Method, path, headers, and body transmitted unchanged	1
Response Relay	Return backend responses to clients	Status code, headers, and response body preserved	1
Header Enrichment	Add proxy-specific headers	X-Forwarded-For and X-Forwarded-Proto headers added	1
Round Robin Distribution	Cycle through backends sequentially	Even distribution across healthy backends	2
Backend Pool Management	Manage list of available backends	Add, remove, and update backend configurations	2

The request forwarding capability requires preserving the complete HTTP request context while adapting it for backend communication. This includes handling edge cases like chunked transfer encoding, WebSocket upgrade requests, and large request bodies that exceed memory limits.

Decision: HTTP/1.1 Focus

- **Context**: Modern applications use various HTTP versions including HTTP/2 and HTTP/3
- **Options Considered**: HTTP/1.1 only, HTTP/2 support, Multi-version support
- **Decision**: Focus exclusively on HTTP/1.1
- **Rationale**: HTTP/1.1 provides sufficient complexity for learning reverse proxy concepts without the additional protocol complexity of multiplexing and stream management in HTTP/2
- **Consequences**: Simplifies implementation and testing while covering 90% of real-world load balancer use cases

Algorithm Flexibility

The load balancer must support multiple distribution algorithms to accommodate different application requirements and traffic patterns. Algorithm selection should be configurable at runtime without requiring service restart.

Algorithm	Purpose	Use Case	Implementation Complexity
`round_robin`	Sequential selection	Even load distribution	Low
`weighted_round_robin`	Proportional distribution	Heterogeneous backend capacity	Medium
`least_connections`	Connection-based selection	Long-lived connections	Medium
`ip_hash`	Consistent client routing	Session affinity requirements	Low
`random`	Random selection	Stateless distribution	Low

Each algorithm must implement the `Algorithm` interface with a `SelectBackend` method that accepts a slice of healthy backends and returns the chosen backend. The selection logic must be thread-safe and handle edge cases like empty backend lists and concurrent access to shared state.

The weighted round robin algorithm presents particular implementation challenges around smooth weight distribution. Rather than simple repetition (which creates bursts), it should use smooth weighted selection that distributes requests proportionally over time without clustering requests to high-weight backends.

Health Monitoring System

Active health checking enables the load balancer to detect backend failures and automatically remove unhealthy servers from rotation. This capability transforms a simple request forwarder into a reliable traffic distribution system that maintains service availability during partial backend failures.

Health Check Feature	Requirement	Configuration Parameter	Default Value
Periodic Probes	HTTP GET requests to health endpoint	`Interval time.Duration`	30 seconds
Failure Detection	Mark unhealthy after consecutive failures	`UnhealthyThreshold int`	3 failures
Recovery Detection	Restore after consecutive successes	`HealthyThreshold int`	2 successes
Probe Timeout	Maximum time to wait for response	`Timeout time.Duration`	5 seconds
Health Endpoint	Configurable probe path	`Path string`	"/health"

The health checking system must operate independently of request processing to avoid blocking normal traffic flow. Health state changes should be atomic and immediately reflected in backend selection algorithms. The system must handle edge cases like all backends becoming unhealthy (continue serving with best-effort routing to least-recently-failed backends) and rapid state oscillation (implement dampening to prevent flapping).

Decision: Active vs Passive Health Checking

- **Context**: Health can be monitored by sending dedicated probes (active) or inferring from request success rates (passive)
- **Options Considered**: Active only, Passive only, Hybrid approach
- **Decision**: Implement active health checking only
- **Rationale**: Active checking provides predictable failure detection timing and clear pass/fail criteria, making it easier to reason about system behavior during failures
- **Consequences**: Adds network overhead from health probes but ensures consistent failure detection regardless of traffic patterns

Quality Requirements

Quality requirements define the non-functional characteristics that make the load balancer suitable for production use. These requirements establish performance baselines, reliability expectations, and operational characteristics that guide implementation decisions throughout the project.

Performance Characteristics

The load balancer must handle realistic traffic loads without becoming a bottleneck between clients and backend services. Performance requirements focus on throughput, latency, and resource utilization under normal and peak load conditions.

Performance Metric	Target	Measurement Method	Rationale
Request Throughput	1000 requests/second	Load testing with concurrent clients	Handles typical web application traffic
Latency Overhead	<5ms median added latency	Comparison with direct backend access	Minimal impact on response times
Memory Usage	<100MB baseline	Process memory monitoring	Lightweight deployment footprint
Connection Pooling	Reuse backend connections	Monitor connection count vs request rate	Reduce connection establishment overhead
Algorithm Selection Time	<1ms for backend selection	Benchmark algorithm implementations	No bottleneck in request path

Performance testing should use realistic request patterns including varying request sizes, concurrent connections, and sustained load over extended periods. The load balancer should demonstrate consistent performance without memory leaks or degradation over time.

Connection pooling efficiency significantly impacts performance characteristics. The reverse proxy component must maintain persistent connections to backend servers and reuse them across multiple client requests. Connection pool sizing should balance resource usage with the ability to handle traffic bursts.

Reliability and Fault Tolerance

The load balancer must continue operating effectively during various failure scenarios including backend server outages, network partitions, and partial system degradation. Reliability requirements define how the system behaves under stress and failure conditions.

Reliability Requirement	Behavior	Recovery Expectation	Implementation Need
Backend Server Failure	Remove failed backends from rotation	Automatic restoration after recovery	Health checking integration
All Backends Unhealthy	Continue serving with degraded routing	Route to least-recently-failed backends	Graceful degradation logic
Network Partition	Timeout and retry failed requests	Client receives 502 Bad Gateway	Timeout configuration
Configuration Reload	Update settings without dropping connections	Zero-downtime configuration changes	Hot reload mechanism
High Load Handling	Maintain service under traffic spikes	Graceful performance degradation	Backpressure and circuit breaking

The system must implement graceful degradation strategies that maintain partial service availability rather than complete failure during adverse conditions. When all backends are unhealthy, the load balancer should attempt to route requests to the most recently healthy backends rather than rejecting all requests.

> **Decision: Fail-Fast vs Circuit Breaker**
>
> - **Context**: When backends fail, the load balancer can either immediately return errors (fail-fast) or temporarily stop trying failed backends (circuit breaker)
> - **Options Considered**: Immediate failure responses, Simple circuit breaker, Adaptive circuit breaker
> - **Decision**: Implement fail-fast with health check-based recovery
> - **Rationale**: Health checking provides the circuit breaker functionality with explicit recovery detection, avoiding the complexity of adaptive thresholds while ensuring failed backends are retried systematically
> - **Consequences**: Simpler implementation with predictable behavior, but may not handle transient failures as gracefully as adaptive circuit breakers

Maintainability and Observability

The load balancer must provide sufficient operational visibility and maintainable code structure to support ongoing development and production operations. These requirements ensure the system can be debugged, monitored, and extended effectively.

Maintainability Aspect	Requirement	Implementation Approach	Benefit
Request Logging	Log all proxied requests with timing	Structured logging with timestamp, method, path, backend, status	Request tracing and debugging
Health Check Logging	Log health state changes	State transition logging with backend and reason	Health debugging
Configuration Validation	Validate config on load with clear errors	Schema validation with specific error messages	Prevent misconfiguration
Error Categorization	Distinguish error types in responses	Different HTTP status codes for different failures	Client error handling
Code Organization	Separate concerns into focused modules	Interface-based design with clear boundaries	Easy testing and extension

Logging must be structured and configurable to support both development debugging and production monitoring. Request logs should include sufficient information to trace request flow without overwhelming log storage. Health check logs should clearly indicate why state changes occurred to support operational troubleshooting.

The code organization should follow Go package conventions with clear separation between HTTP handling, backend management, algorithm implementation, and health checking. Each component should have well-defined interfaces that enable independent testing and future extension.

Out of Scope

Explicitly defining features that are NOT included in this basic load balancer helps maintain focus on the core learning objectives and prevents scope creep during implementation. These exclusions represent functionality that would be added in a production load balancer but would complicate the learning experience.

SSL/TLS Termination

The load balancer will NOT handle SSL/TLS encryption or certificate management. All communication between clients and the load balancer, and between the load balancer and backends, will use plain HTTP.

Excluded SSL Feature	Rationale for Exclusion	Alternative Approach
Certificate Management	Complex PKI operations distract from load balancing logic	Use external TLS termination (nginx, cloud load balancer)
SSL Handshake Processing	Cryptographic operations add significant complexity	Deploy load balancer behind TLS-terminating reverse proxy
SNI (Server Name Indication)	Multiple certificate handling complicates configuration	Single domain deployments or external TLS handling
SSL Session Resumption	Performance optimization unrelated to load balancing	Focus on HTTP-level optimizations

This exclusion allows the implementation to focus on HTTP protocol handling, request routing, and backend management without the additional complexity of cryptographic operations and certificate lifecycle management.

Advanced Routing Rules

The load balancer will implement basic host-based routing only, without support for complex routing rules based on request content, headers, or path patterns.

Excluded Routing Feature	Complexity Added	Learning Value
Path-based routing (/api → backend1, /static → backend2)	Request parsing and rule evaluation	Low - focuses on configuration complexity
Header-based routing (User-Agent, Custom headers)	Header matching and rule precedence	Low - peripheral to core load balancing
Content-based routing (request body inspection)	Deep packet inspection	Low - performance anti-pattern
Geographic routing (client IP geolocation)	External data dependencies	Low - focuses on data integration
Rate limiting per client	Client state tracking	Medium - but orthogonal to load balancing

These routing features represent important production capabilities but would shift focus from core load balancing algorithms and health management to rule evaluation and configuration complexity.

Caching and Content Optimization

The load balancer will NOT implement any caching mechanisms or content optimization features, maintaining its role as a pure traffic distribution system.

Excluded Caching Feature	Implementation Complexity	Focus Distraction
Response Caching	Cache invalidation strategies, storage management	High - becomes a cache server project
Static File Serving	File system integration, MIME type handling	Medium - unrelated to load balancing
Compression (gzip, brotli)	Content encoding/decoding	Medium - focuses on content transformation
Response Buffering	Memory management for large responses	Low - useful but adds complexity

Caching functionality represents a separate system design challenge with its own complexity around cache invalidation, storage management, and consistency. Including caching would transform the project from a load balancer implementation into a caching proxy implementation.

Decision: Pure Load Balancer vs Feature-Rich Proxy

- **Context**: Production load balancers often include caching, SSL termination, and advanced routing to reduce infrastructure complexity
- **Options Considered**: Basic load balancer only, Load balancer with SSL, Full-featured proxy
- **Decision**: Implement basic load balancer only
- **Rationale**: Each additional feature adds significant complexity that obscures the core learning objectives of request distribution, health checking, and algorithm implementation
- **Consequences**: Results in a focused learning experience but produces a component that requires additional infrastructure (TLS termination, caching) for production use

Advanced Load Balancing Features

Several advanced load balancing capabilities are excluded to maintain implementation focus on fundamental concepts while still producing a useful and educational system.

Excluded Advanced Feature	Reason for Exclusion	Alternative Learning Path
Global Server Load Balancing (GSLB)	Multi-datacenter complexity	Separate distributed systems project
Service Discovery Integration	External system dependencies	Configuration-file based backend management
Real-time Metrics Dashboard	UI development distraction	Log-based monitoring and external tools
Connection Draining	Complex connection lifecycle management	Focus on health check-based removal
Sticky Sessions (beyond IP hash)	Session storage and synchronization	IP hash provides basic session affinity

These features represent important production capabilities but involve integration with external systems, complex state management, or user interface development that would significantly expand the project scope beyond its core learning objectives.

The excluded features provide natural extension points for learners who complete the basic implementation and want to explore additional challenges. Each exclusion represents a potential follow-up project that builds on the foundational load balancing concepts.

Implementation Guidance

The functional and quality requirements translate into specific technical decisions that guide implementation throughout the project milestones. This guidance provides concrete direction for key technology choices and architectural decisions.

Technology Recommendations

Component	Simple Option	Advanced Option	Recommendation
HTTP Server	`net/http` standard library	Gin or Echo framework	Standard library - better learning
HTTP Client	`net/http.Client` with connection pooling	Custom HTTP/2 client	Standard library with custom pool
Configuration	JSON file with `encoding/json`	YAML with validation library	JSON for simplicity
Logging	`log/slog` structured logging	Third-party logging framework	Standard library structured logging
Concurrency	`sync.Mutex` and `sync/atomic`	Third-party concurrent data structures	Standard library primitives
Testing	`testing` package with table tests	Behavior-driven testing framework	Standard library with custom helpers

Recommended Project Structure

The codebase should be organized to separate concerns clearly while maintaining simplicity for the learning environment:

```
load-balancer/
├── cmd/
│   └── load-balancer/
│       └── main.go                 ← Entry point and CLI handling
├── internal/
│   ├── config/
│   │   ├── config.go               ← Configuration loading and validation
│   │   └── config_test.go
│   ├── proxy/
│   │   ├── proxy.go                ← HTTP reverse proxy implementation
│   │   └── proxy_test.go
│   ├── backend/
│   │   ├── manager.go              ← Backend pool management
│   │   ├── health.go               ← Health checking system
│   │   └── backend_test.go
│   ├── algorithm/
│   │   ├── algorithm.go            ← Algorithm interface definition
│   │   ├── roundrobin.go           ← Round robin implementation
│   │   ├── weighted.go             ← Weighted round robin
│   │   ├── leastconn.go           ← Least connections
│   │   ├── iphash.go              ← IP hash algorithm
│   │   └── algorithm_test.go
│   └── router/
│       ├── router.go               ← Request routing coordination
│       └── router_test.go
├── configs/
│   └── example.json                ← Example configuration file
├── scripts/
│   └── test-backends.go            ← Simple HTTP servers for testing
└── go.mod
```

Infrastructure Starter Code

Configuration Management - Complete implementation for loading and validating configuration:

```go
// internal/config/config.go

package config

import (
    "encoding/json"
    "fmt"
    "os"
    "time"
    "net/url"
)

type Config struct {
    Port       int              `json:"port"`
    Backends   []BackendConfig  `json:"backends"`
    Algorithm  string           `json:"algorithm"`
    HealthCheck HealthCheckConfig `json:"health_check"`
}

type BackendConfig struct {
    URL    string `json:"url"`
    Weight int    `json:"weight"`
}

type HealthCheckConfig struct {
    Enabled            bool          `json:"enabled"`
    Interval           time.Duration `json:"interval"`
    Timeout            time.Duration `json:"timeout"`
    HealthyThreshold   int           `json:"healthy_threshold"`
    UnhealthyThreshold int           `json:"unhealthy_threshold"`
    Path               string        `json:"path"`
}

func LoadConfig(filename string) (*Config, error) {
    data, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("reading config file: %w", err)
    }

    var config Config
    if err := json.Unmarshal(data, &config); err != nil {
        return nil, fmt.Errorf("parsing config JSON: %w", err)
    }
```

```
    if err := config.Validate(); err != nil {
        return nil, fmt.Errorf("invalid configuration: %w", err)
    }

    return &config, nil
}

func (c *Config) Validate() error {
    if c.Port <= 0 || c.Port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535, got %d", c.Port)
    }

    if len(c.Backends) == 0 {
        return fmt.Errorf("at least one backend must be configured")
    }

    validAlgorithms := map[string]bool{
        "round_robin":         true,
        "weighted_round_robin": true,
        "least_connections":   true,
        "ip_hash":             true,
        "random":              true,
    }

    if !validAlgorithms[c.Algorithm] {
        return fmt.Errorf("unsupported algorithm: %s", c.Algorithm)
    }

    for i, backend := range c.Backends {
        if _, err := url.Parse(backend.URL); err != nil {
            return fmt.Errorf("invalid backend URL at index %d: %w", i, err)
        }
        if backend.Weight <= 0 {
            return fmt.Errorf("backend weight must be positive, got %d at index %d", backend.Weight, i)
        }
    }

    if c.HealthCheck.Enabled {
        if c.HealthCheck.Interval <= 0 {
            return fmt.Errorf("health check interval must be positive")
        }
```

```
        if c.HealthCheck.Timeout <= 0 {
            return fmt.Errorf("health check timeout must be positive")
        }
        if c.HealthCheck.HealthyThreshold <= 0 {
            return fmt.Errorf("healthy threshold must be positive")
        }
        if c.HealthCheck.UnhealthyThreshold <= 0 {
            return fmt.Errorf("unhealthy threshold must be positive")
        }
    }

    return nil
}
```

HTTP Request/Response Utilities - Helper functions for HTTP handling:

```go
// internal/proxy/http_utils.go

package proxy


import (

    "context"

    "io"

    "net/http"

    "time"

)


// RequestForwarder handles the details of forwarding HTTP requests

type RequestForwarder struct {

    client   *http.Client

    timeout  time.Duration

}


func NewRequestForwarder(timeout time.Duration) *RequestForwarder {

    return &RequestForwarder{

        client: &http.Client{

            Timeout: timeout,

            // Connection pooling is handled automatically by http.Client

        },

        timeout: timeout,

    }

}


// ForwardRequest forwards an HTTP request to the specified backend URL

func (rf *RequestForwarder) ForwardRequest(originalReq *http.Request, backendURL string) (*http.Response, error) {

    ctx, cancel := context.WithTimeout(context.Background(), rf.timeout)

    defer cancel()



    // TODO: Create new request with backend URL

    // TODO: Copy method, path, headers, and body from original request

    // TODO: Add X-Forwarded-For and X-Forwarded-Proto headers

    // TODO: Send request using rf.client.Do with context

    // TODO: Return response or error



    return nil, nil

}


// CopyHeaders copies headers from source to destination, excluding hop-by-hop headers

func CopyHeaders(dst, src http.Header) {

    hopByHopHeaders := map[string]bool{
```

```
        "Connection":          true,
        "Proxy-Connection":    true,
        "Keep-Alive":          true,
        "Proxy-Authenticate": true,
        "Proxy-Authorization": true,
        "Te":                  true,
        "Trailer":             true,
        "Transfer-Encoding":   true,
        "Upgrade":             true,
    }


    for key, values := range src {
        if !hopByHopHeaders[key] {
            for _, value := range values {
                dst.Add(key, value)
            }
        }
    }
}
```

Core Logic Skeleton Code

Algorithm Interface and Round Robin Implementation:

```go
// internal/algorithm/algorithm.go

package algorithm


import "github.com/yourorg/load-balancer/internal/backend"


// Algorithm defines the interface for backend selection algorithms

type Algorithm interface {

    // SelectBackend chooses a backend from the provided healthy backends

    // Returns nil if no suitable backend is available

    SelectBackend(backends []*backend.Backend) *backend.Backend



    // Name returns the algorithm name for logging and configuration

    Name() string

}


// internal/algorithm/roundrobin.go

package algorithm


import (

    "sync/atomic"

    "github.com/yourorg/load-balancer/internal/backend"

)


// RoundRobinAlgorithm implements sequential backend selection

type RoundRobinAlgorithm struct {

    counter uint64

}


func NewRoundRobinAlgorithm() *RoundRobinAlgorithm {

    return &RoundRobinAlgorithm{}

}


func (rr *RoundRobinAlgorithm) Name() string {

    return "round_robin"

}


func (rr *RoundRobinAlgorithm) SelectBackend(backends []*backend.Backend) *backend.Backend {

    // TODO 1: Check if backends slice is empty, return nil if so

    // TODO 2: Atomically increment the counter using atomic.AddUint64

    // TODO 3: Use modulo operation to get index within bounds of backends slice

    // TODO 4: Return the backend at the calculated index

    // Hint: Be careful with counter overflow - use modulo on the incremented value

    // Hint: atomic.AddUint64 returns the new value after increment
```

```
    return nil

}
```

Backend Management Core:

```go
// internal/backend/backend.go

package backend


import (

    "sync"

    "time"

    "net/url"

)


// Backend represents a backend server with health and connection tracking

type Backend struct {

    URL              *url.URL

    Weight           int

    Healthy          bool

    ActiveConnections int32

    LastHealthCheck  time.Time

    ConsecutiveFailures int

    mu               sync.RWMutex

}


// HealthResult represents the result of a health check

type HealthResult struct {

    Backend *Backend

    Healthy bool

    Error   error

    Duration time.Duration

}


// Manager handles the pool of backend servers

type Manager struct {

    backends []*Backend

    mu       sync.RWMutex

}


func NewManager() *Manager {

    return &Manager{

        backends: make([]*Backend, 0),

    }

}


func (m *Manager) AddBackend(url string, weight int) error {

    // TODO 1: Parse the URL using net/url.Parse

    // TODO 2: Create new Backend struct with parsed URL and weight
```

```
    // TODO 3: Initialize health status as true (assume healthy until proven otherwise)
    // TODO 4: Acquire write lock and add backend to slice
    // TODO 5: Return error if URL parsing fails

    return nil
}

func (m *Manager) GetHealthyBackends() []*Backend {
    // TODO 1: Acquire read lock for thread safety
    // TODO 2: Iterate through backends and collect healthy ones
    // TODO 3: Return slice of healthy backends
    // TODO 4: Handle case where no backends are healthy (return all for graceful degradation)

    return nil
}

func (m *Manager) UpdateHealth(backend *Backend, healthy bool) {
    // TODO 1: Acquire write lock on the backend
    // TODO 2: Update the Healthy field
    // TODO 3: Update LastHealthCheck timestamp
    // TODO 4: Reset or increment ConsecutiveFailures based on health status
    // TODO 5: Log health state changes for debugging

}
```

Language-Specific Hints

Go-Specific Implementation Tips:

- Use `sync/atomic` package for thread-safe counter operations in algorithms: `atomic.AddUint64(&counter, 1)`
- Use `sync.RWMutex` for backend pool access - read locks for getting backends, write locks for health updates
- Use `context.WithTimeout` for HTTP requests to backends with configurable timeouts
- Use `net/http.Client` with custom transport for connection pooling: `&http.Transport{MaxIdleConnsPerHost: 10}`
- Use `log/slog` for structured logging: `slog.Info("request forwarded", "backend", backend.URL, "status", response.StatusCode)`
- Use table-driven tests for algorithm testing: create slices of test cases with inputs and expected outputs
- Use `httptest.NewServer` for creating test backend servers during integration testing
- Handle graceful shutdown with `context.Context` and `sync.WaitGroup` for cleanup

Common Go Pitfalls to Avoid:

- Don't forget to check for nil pointers when backends slice might be empty
- Always close response bodies: `defer response.Body.Close()` after successful HTTP requests
- Use `io.ReadAll` carefully - it loads entire response into memory, consider streaming for large responses
- Don't share http.Client instances across goroutines without proper configuration
- Be careful with slice operations under concurrent access - always use locks
- Use `atomic` operations for simple counters rather than mutex locks for performance

Milestone Checkpoints

Milestone 1 Checkpoint - HTTP Proxy Foundation:

```bash
# Start your load balancer
go run cmd/load-balancer/main.go -config configs/single-backend.json

# In another terminal, start a test backend
go run scripts/test-backend.go -port 8081

# Test request forwarding
curl -v http://localhost:8080/test
# Expected: Response from backend on port 8081, with X-Forwarded-* headers

# Test with POST request and body
curl -X POST -d "test data" http://localhost:8080/api
# Expected: POST request forwarded with body intact

# Test backend failure
# Stop the test backend and make a request
curl http://localhost:8080/test
# Expected: 502 Bad Gateway response
```

Milestone 2 Checkpoint - Round Robin Distribution:

```bash
# Start multiple test backends
go run scripts/test-backend.go -port 8081 -name backend1 &
go run scripts/test-backend.go -port 8082 -name backend2 &
go run scripts/test-backend.go -port 8083 -name backend3 &

# Start load balancer with multiple backends
go run cmd/load-balancer/main.go -config configs/round-robin.json

# Test distribution
for i in {1..9}; do curl http://localhost:8080/; done
# Expected: Responses cycling through backend1, backend2, backend3

# Verify even distribution with more requests
for i in {1..30}; do curl -s http://localhost:8080/ | grep -o 'backend[0-9]'; done | sort | uniq -c
# Expected: Roughly 10 requests to each backend (within 1-2 requests)
```

Milestone 3 Checkpoint - Health Checks:

```bash
# Start backends and load balancer

# Stop one backend to trigger health check failure
kill <backend2-pid>

# Wait for health check interval to pass, then test
sleep 35  # Wait longer than health check interval
for i in {1..6}; do curl http://localhost:8080/; done
# Expected: Only backend1 and backend3 responses, no 502 errors

# Restart failed backend and verify recovery
go run scripts/test-backend.go -port 8082 -name backend2 &
sleep 35  # Wait for recovery detection
for i in {1..9}; do curl http://localhost:8080/; done
# Expected: All three backends receiving requests again
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All requests return 502	No healthy backends or backend URLs wrong	Check backend URLs in config, verify test backends are running	Fix URLs or start backends
Uneven distribution in round robin	Race condition in counter increment	Add logging to counter value, check for atomic operations	Use `sync/atomic` instead of regular increment
Health checks not working	Health check URL path incorrect	Check backend logs for health check requests	Verify health check path matches backend endpoint
Panic on backend selection	Empty backends slice not handled	Look for nil pointer dereference in algorithm	Add nil/empty checks before slice access
Memory leak during load testing	HTTP response bodies not closed	Monitor memory usage over time, check for `defer response.Body.Close()`	Add proper cleanup in request forwarding
Requests hanging indefinitely	No timeouts configured	Check if requests eventually complete	Add context timeouts to HTTP client

The debugging approach should start with logs (add structured logging throughout), then use Go's built-in tools (`go tool pprof` for memory/CPU profiling), and finally add targeted print statements in suspected areas. The key is to verify each milestone's functionality before moving to the next level of complexity.

High-Level Architecture

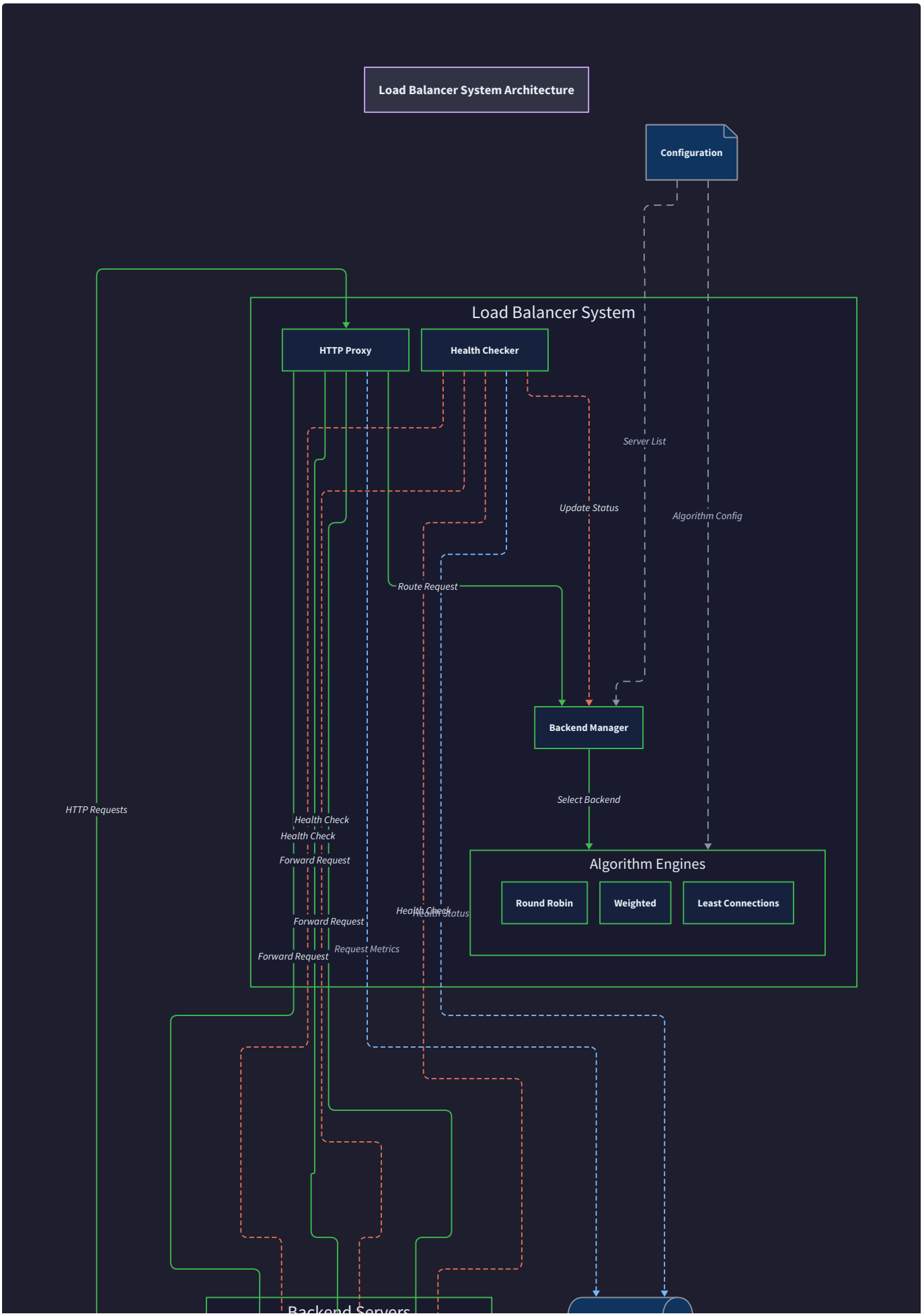
> **Milestone(s):** All milestones (1-4) — the architectural foundation established here supports HTTP proxy functionality (Milestone 1), request distribution (Milestone 2), health monitoring (Milestone 3), and algorithm extensibility (Milestone 4)

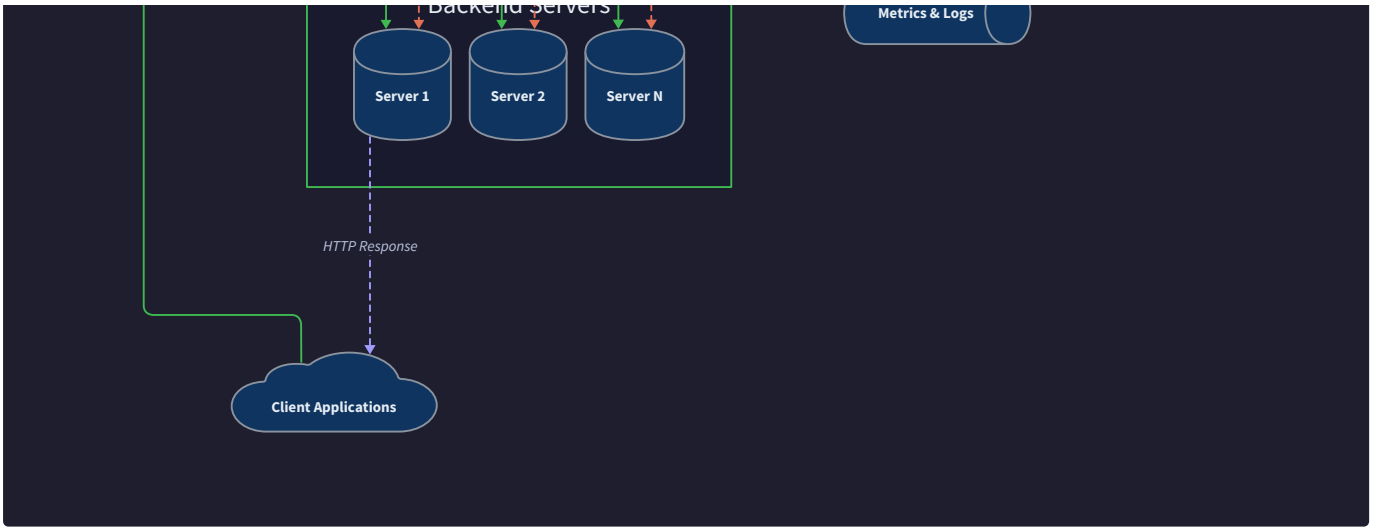
Mental Model: Airport Terminal System

Think of our load balancer as an airport terminal managing passenger flow to multiple boarding gates. The **arrivals hall** receives all incoming passengers (HTTP requests) and directs them to a **central information desk** (request router). The information desk consults the **gate assignment board** (backend manager) to determine which gate (backend server) should handle each passenger, using different strategies like sending passengers to the gate with the shortest line or rotating through gates in order.

Behind the scenes, **gate monitors** (health checkers) continuously verify that each gate is operational and ready to accept passengers. If a gate closes for maintenance, the monitors immediately update the assignment board so no passengers are sent there. The **traffic control algorithms** (load balancing algorithms) determine the specific logic for choosing gates - whether it's simple rotation, sending passengers to the least crowded gate, or ensuring frequent flyers always go to their preferred gate.

This mental model captures the key architectural insight: **separation of concerns between request handling, backend management, health monitoring, and selection algorithms**. Each component has a distinct responsibility but must coordinate seamlessly to provide a unified service.


Load Balancer System Architecture
Configuration
Load Balancer System
HTTP Proxy
Health Checker
Server List
Update Status
Algorithm Config
Route Request
Backend Manager
Select Backend
HTTP Requests
Health Check
Health Check
Forward Request
Algorithm Engines
Round Robin
Weighted
Least Connections
Health Check
Health Status
Forward Request
Request Metrics
Forward Request
Backend Servers



System Components

The load balancer architecture consists of four core components that work together to provide reliable request distribution across multiple backend servers. Each component has clearly defined responsibilities and interfaces, enabling modular development and testing while maintaining loose coupling between subsystems.

HTTP Reverse Proxy Component

The **HTTP Reverse Proxy** serves as the system's entry point, accepting incoming client connections and forwarding requests to selected backend servers. This component handles all the low-level HTTP protocol details, connection management, and request/response translation between clients and backends. It abstracts away the complexity of HTTP communication so other components can focus on their specific responsibilities.

The reverse proxy maintains connection pools to backend servers for efficiency, implements request and response buffering to handle variable network speeds, and adds standard proxy headers like `X-Forwarded-For` to preserve client information. When backend selection fails or no healthy backends are available, the proxy generates appropriate HTTP error responses to clients.

Responsibility	Description	Interface Method
Request Acceptance	Listen on configured port and accept HTTP connections	`ListenAndServe(port int) error`
Request Parsing	Parse HTTP method, path, headers, and body	`parseRequest(*http.Request) (*ProxyRequest, error)`
Backend Communication	Forward requests to selected backend servers	`forwardRequest(*ProxyRequest, *Backend) (*ProxyResponse, error)`
Response Relay	Return backend responses to original clients	`sendResponse(http.ResponseWriter, *ProxyResponse) error`
Connection Management	Handle keep-alive, timeouts, and connection pooling	`getConnection(*Backend) (*http.Client, error)`
Error Response Generation	Generate 502 Bad Gateway and other error responses	`sendErrorResponse(http.ResponseWriter, int, string) error`

Backend Pool Manager

The **Backend Pool Manager** maintains the authoritative registry of all backend servers and their current operational state. This component tracks which backends are available, their configuration parameters like weights, and their real-time health status. It provides a clean interface for other components to query available backends without exposing internal state management complexity.

The backend manager handles dynamic configuration updates, allowing backends to be added or removed without service interruption. It maintains thread-safe access to the backend pool since multiple goroutines will be selecting backends concurrently for different client requests. The manager also coordinates with the health checking system to update backend availability in real-time.

Responsibility	Description	Interface Method
Backend Registration	Add and remove backends from the active pool	`AddBackend(config BackendConfig) error`
State Management	Track health status, weights, and connection counts	`UpdateBackendState(*Backend, *HealthResult) error`
Backend Queries	Provide filtered views of healthy/available backends	`GetHealthyBackends() []*Backend`
Configuration Updates	Handle dynamic backend pool changes	`UpdatePool([]BackendConfig) error`
Concurrency Safety	Ensure thread-safe access to shared backend state	`lockBackends() *sync.RWMutex`
Statistics Tracking	Maintain request counts and performance metrics	`GetBackendStats(*Backend) *BackendStats`

Health Checking System

The **Health Checking System** runs as a background process that continuously monitors backend server availability through periodic HTTP health probes. This component implements the failure detection and recovery logic that automatically removes unhealthy backends from rotation and restores them when they recover. The health checker operates independently of request processing to provide consistent monitoring regardless of traffic patterns.

The health checking system maintains separate state for each backend's health status, including consecutive failure counts, last check timestamps, and response time metrics. It implements configurable thresholds for marking backends as unhealthy (consecutive failures) and healthy (consecutive successes), providing tunable sensitivity to transient issues versus persistent failures.

Responsibility	Description	Interface Method
Periodic Probing	Send HTTP health checks at configured intervals	`startHealthChecks(interval time.Duration) error`
Failure Detection	Count consecutive failures and mark backends unhealthy	`processHealthResult(*Backend, *HealthResult) error`
Recovery Detection	Detect recovered backends and restore to rotation	`checkRecovery(*Backend) error`
Health State Updates	Notify backend manager of health status changes	`updateHealthStatus(*Backend, bool) error`
Configuration Management	Handle health check settings and endpoint changes	`UpdateHealthConfig(*HealthCheckConfig) error`
Monitoring Integration	Expose health metrics for operational visibility	`GetHealthMetrics() *HealthMetrics`

Load Balancing Algorithm Engines

The **Algorithm Engines** implement the various strategies for selecting which backend server should handle each incoming request. Each algorithm is implemented as a separate component that conforms to a common interface, enabling runtime algorithm switching and easy addition of new selection strategies. The algorithms operate on the filtered list of healthy backends provided by the backend manager.

Different algorithms optimize for different goals: round robin provides even distribution, least connections minimizes server load, weighted algorithms allow capacity-based distribution, and hash-based algorithms provide session affinity. Each algorithm maintains its own internal state (like counters or connection tracking) while being thread-safe for concurrent access.

Algorithm Type	Purpose	State Maintained	Selection Strategy
Round Robin	Even request distribution	Current position counter	Sequential cycling through backends
Weighted Round Robin	Capacity-proportional distribution	Per-backend current/target weights	Smooth weighted selection algorithm
Least Connections	Minimize server load	Active connection count per backend	Select backend with minimum connections
IP Hash	Session affinity/stickiness	Hash ring or consistent mapping	Hash client IP to consistent backend
Random	Stateless even distribution	None (purely random)	Uniform random selection

Request Flow

Understanding how requests flow through the load balancer system is crucial for implementing correct coordination between components. The request processing pipeline involves multiple decision points where different failure scenarios can occur, requiring careful error handling and fallback logic at each stage.

Normal Request Processing Flow

The typical request flow begins when a client establishes an HTTP connection to the load balancer's listening port. The reverse proxy accepts the connection and parses the incoming HTTP request, extracting the method, path, headers, and body content. This parsing step validates that the request is well-formed and can be successfully forwarded to a backend server.

Once the request is parsed, the reverse proxy consults the backend pool manager to obtain the current list of healthy, available backends. If no backends are available, the proxy immediately returns a 503 Service Unavailable response to the client without attempting backend selection. This fail-fast approach prevents unnecessary processing and provides clear error feedback.

When healthy backends are available, the reverse proxy invokes the currently configured load balancing algorithm to select the specific backend that should handle this request. The algorithm receives the list of healthy backends and returns its selection based on the algorithm's internal logic and state. For example, round robin increments its counter and selects the next backend in sequence, while least connections examines the active connection count for each backend.

Processing Stage	Component	Input	Output	Error Conditions
Connection Accept	HTTP Reverse Proxy	Client TCP connection	Parsed HTTP request	Malformed HTTP, connection timeout
Backend Pool Query	Backend Pool Manager	Health status filter	List of healthy backends	No healthy backends available
Algorithm Selection	Load Balancing Algorithm	Healthy backend list	Selected backend	Algorithm state corruption
Request Forwarding	HTTP Reverse Proxy	Request + selected backend	Backend HTTP response	Backend connection failure, timeout
Response Relay	HTTP Reverse Proxy	Backend response	Client receives response	Client connection dropped

Backend Selection and Forwarding

After the algorithm selects a target backend, the reverse proxy establishes or reuses an HTTP connection to that backend server. The proxy modifies the original request by adding standard forwarding headers (`X-Forwarded-For` , `X-Forwarded-Proto` , `X-Forwarded-Host`) and potentially modifying the `Host` header to match the backend's expected hostname.

The request forwarding process involves streaming the request body to the backend while handling potential network issues like slow clients or backends. The reverse proxy implements appropriate timeouts to prevent connections from hanging indefinitely and monitors for connection failures that indicate the backend has become unavailable.

When the backend responds, the reverse proxy receives the HTTP response including status code, headers, and body. The response is then streamed back to the original client, preserving all headers and status information. Throughout this process, the proxy maintains connection state and updates any relevant metrics like active connection counts needed by the least connections algorithm.

Error Handling and Fallback Logic

Multiple failure scenarios can occur during request processing, each requiring specific error handling logic. **Connection failures** occur when the selected backend is unreachable due to network issues or the backend service being down. In this case, the reverse proxy returns a 502 Bad Gateway response and may trigger an immediate health check to update the backend's status.

Timeout failures happen when a backend accepts the connection but takes too long to respond. The reverse proxy enforces configurable timeouts for connection establishment, request sending, and response reading. When timeouts occur, the proxy closes the backend connection and returns a 504 Gateway Timeout response to the client.

Backend unavailability occurs when no healthy backends are available to handle requests. This situation requires careful handling because it represents a complete service outage from the client's perspective. The reverse proxy returns a 503 Service Unavailable response with appropriate retry-after headers to guide client behavior.

> **Critical Design Insight**: The request flow must be designed to fail fast and provide clear error signals rather than hanging indefinitely. Each component should have bounded processing time and clear error propagation to prevent cascading failures that could affect other concurrent requests.

The request flow also handles **partial failures** where some backends are available but others are not. The load balancing algorithm automatically excludes unhealthy backends from selection, allowing the system to continue operating with reduced capacity. This graceful degradation is essential for maintaining service availability during backend maintenance or failures.

Concurrency and Thread Safety

Multiple client requests are processed concurrently, requiring careful coordination between components to maintain consistency. The backend pool manager uses read-write locks to allow concurrent read access to the backend list while serializing updates from health checks or configuration changes. Each load balancing algorithm implements its own thread safety mechanisms appropriate for its internal state.

The reverse proxy creates separate goroutines for each client connection, enabling high concurrency without blocking. However, shared resources like connection pools and algorithm state require proper synchronization to prevent race conditions. The system uses atomic operations for simple counters and mutexes for more complex shared state.

Recommended Project Structure

Organizing the codebase into clear modules and packages is essential for managing the complexity of the load balancer system. The recommended structure separates concerns along component boundaries while providing clear interfaces between modules. This organization supports incremental development through the project milestones and makes testing individual components straightforward.

Top-Level Project Organization

The project follows Go's standard layout conventions with clearly separated packages for each major component. The `cmd` directory contains the main entry point and command-line interface, while `internal` packages contain the core load balancer logic that should not be imported by external projects. Configuration files and

documentation live at the project root for easy access.

```
load-balancer/
├── cmd/
│   └── loadbalancer/
│       ├── main.go                # Entry point and CLI setup
│       ├── server.go              # HTTP server lifecycle management
│       └── config.go              # Configuration loading and validation
├── internal/
│   ├── proxy/                     # HTTP reverse proxy implementation
│   ├── backend/                   # Backend pool management
│   ├── health/                    # Health checking system
│   ├── algorithm/                 # Load balancing algorithms
│   ├── config/                    # Configuration types and parsing
│   └── metrics/                   # Monitoring and statistics
├── pkg/                           # Public interfaces (if any)
├── configs/
│   ├── development.yaml           # Development configuration
│   ├── production.yaml            # Production configuration
│   └── example.yaml               # Example with all options
├── tests/
│   ├── integration/               # End-to-end tests
│   └── testdata/                  # Test fixtures and mock backends
├── docs/
│   ├── api.md                     # Configuration API documentation
│   └── deployment.md              # Deployment and operations guide
├── scripts/
│   ├── test-backend.go            # Simple test HTTP server
│   └── benchmark.go               # Load testing utilities
├── go.mod
├── go.sum
├── README.md
└── Makefile                       # Build and test automation
```

Core Component Package Design

Each major component is organized as a separate Go package with clear internal structure. The package design follows the principle of high cohesion within packages and loose coupling between packages, using well-defined interfaces for communication between components.

Proxy Package Structure (`internal/proxy/`): The proxy package contains all HTTP reverse proxy functionality including request parsing, backend communication, and response handling. It depends on the backend and algorithm packages through interfaces rather than concrete types.

File	Purpose	Key Types	Dependencies
`proxy.go`	Main proxy server and request handling	`ReverseProxy` , `ProxyRequest` , `ProxyResponse`	backend, algorithm interfaces
`forwarder.go`	HTTP request/response forwarding logic	`RequestForwarder` , `ResponseForwarder`	net/http, backend types
`connection.go`	Connection pooling and management	`ConnectionPool` , `BackendConnection`	net/http, sync primitives
`middleware.go`	Header manipulation and logging	`HeaderMiddleware` , `LoggingMiddleware`	logging, metrics interfaces
`proxy_test.go`	Unit tests for proxy functionality	Test fixtures and mocks	testing, httptest

Backend Package Structure (`internal/backend/`): The backend package manages the pool of backend servers including their configuration, health state, and runtime statistics. It provides thread-safe access to backend information for both request processing and health checking.

File	Purpose	Key Types	Dependencies
`manager.go`	Backend pool management and coordination	`BackendManager` , `Backend`	sync, config types
`backend.go`	Individual backend server representation	`Backend` , `BackendStats` , `BackendState`	net/url, atomic
`pool.go`	Backend collection and filtering operations	`BackendPool` , `PoolFilter`	sync, algorithm interfaces
`config.go`	Backend configuration and validation	`BackendConfig` , `PoolConfig`	config package
`manager_test.go`	Backend manager unit tests	Test backends and mock configs	testing, testify

Algorithm Package Structure (`internal/algorithm/`): The algorithm package implements all load balancing strategies as separate modules that conform to a common interface. This design enables easy addition of new algorithms and runtime algorithm switching.

File	Purpose	Key Types	Dependencies
`interface.go`	Common algorithm interface definition	`Algorithm`, `SelectionContext`	backend types
`roundrobin.go`	Round robin algorithm implementation	`RoundRobinAlgorithm`	atomic, backend types
`weighted.go`	Weighted round robin implementation	`WeightedAlgorithm`, `WeightTracker`	backend types, math
`leastconn.go`	Least connections algorithm	`LeastConnectionsAlgorithm`	backend types, atomic
`iphash.go`	IP hash algorithm for session affinity	`IPHashAlgorithm`, `ConsistentHash`	hash, backend types
`random.go`	Random selection algorithm	`RandomAlgorithm`	math/rand, backend types
`registry.go`	Algorithm registration and factory	`AlgorithmRegistry`, `AlgorithmFactory`	all algorithm implementations

Configuration and Startup Flow

The configuration system provides flexible deployment options while maintaining clear validation and error reporting. The main entry point coordinates component initialization in the correct order, ensuring dependencies are satisfied and resources are properly allocated.

Configuration Loading Process:

1. Parse command-line flags for config file path and override options
2. Load configuration file (YAML/JSON) and validate schema
3. Apply command-line overrides to configuration values
4. Validate complete configuration including backend reachability
5. Initialize logging and metrics systems based on configuration

Component Initialization Order:

1. **Metrics system** - Initialize first to capture initialization metrics
2. **Backend manager** - Create backend pool from configuration
3. **Health checker** - Start monitoring configured backends
4. **Algorithm registry** - Register all available algorithm implementations
5. **Request router** - Initialize with selected algorithm
6. **HTTP proxy** - Create proxy server with router dependency
7. **Signal handlers** - Setup graceful shutdown and configuration reload

Testing Strategy Integration

The project structure supports comprehensive testing at multiple levels, from unit tests of individual algorithms to integration tests of complete request flows. Each package contains its own unit tests, while integration tests exercise cross-component interactions.

Unit Testing Structure:

- Each package contains `*_test.go` files with comprehensive unit tests
- Test fixtures and mocks are co-located with the components they test
- Table-driven tests validate algorithm behavior across different scenarios
- Benchmark tests measure algorithm selection performance

Integration Testing Structure:

- `tests/integration/` contains end-to-end test scenarios
- Mock backend servers simulate various failure conditions
- Test configurations exercise different algorithm and health check settings
- Automated test scripts validate milestone acceptance criteria

Architecture Decision: Package Organization

- **Context**: Need to balance component separation with development velocity
- **Options Considered**:
  1. Monolithic package with all components
  2. Fine-grained packages for every type
  3. Component-based packages with clear interfaces
- **Decision**: Component-based packages (option 3)
- **Rationale**: Provides clear separation of concerns while avoiding excessive package proliferation that complicates imports and testing
- **Consequences**: Enables independent development of components, supports milestone-based implementation, requires interface discipline

Development Workflow Support

The project structure supports incremental development through the defined milestones, allowing learners to focus on one component at a time while maintaining a working system. Each milestone builds upon previous work without requiring major refactoring.

Milestone Development Path:

- **Milestone 1**: Implement proxy package with single backend forwarding
- **Milestone 2**: Add algorithm package with round robin implementation
- **Milestone 3**: Implement health package and integrate with backend manager
- **Milestone 4**: Add remaining algorithms and runtime switching capability

The structure also supports iterative refinement where learners can start with simple implementations and enhance them with additional features like connection pooling, advanced error handling, and monitoring integration.

Implementation Guidance

Technology Recommendations

The load balancer implementation uses Go's standard library extensively while incorporating specific third-party packages for configuration management and testing utilities. This approach minimizes external dependencies while leveraging robust, well-tested libraries for complex functionality.

Component	Simple Option	Advanced Option	Rationale
HTTP Server	`net/http` standard library	`fasthttp` or `fiber` framework	Standard library provides sufficient performance and better learning value
Configuration	`gopkg.in/yaml.v3` for YAML parsing	`viper` for multi-format config	YAML is simple and readable for load balancer configuration
Logging	`log/slog` standard structured logging	`logrus` or `zap` high-performance	Standard library slog provides structured logging with good performance
Testing	`testing` + `net/http/httptest`	`testify` + `ginkgo/gomega`	Standard library sufficient for unit tests, testify adds convenience
Metrics	Simple counters with atomic operations	`prometheus/client_golang`	Start simple, add Prometheus when monitoring is needed
JSON Parsing	`encoding/json` standard library	`json-iterator` for performance	Standard library performance adequate for configuration

Recommended File Structure

This structure supports milestone-based development where learners implement one component at a time while maintaining a working system throughout the development process.

```
load-balancer/
├── cmd/
│   └── loadbalancer/
│       ├── main.go                # Entry point with signal handling and graceful shutdown
│       └── server.go              # HTTP server setup and lifecycle management
├── internal/
│   ├── config/
│   │   ├── config.go              # Configuration types and validation
│   │   ├── loader.go              # File loading and parsing logic
│   │   └── config_test.go         # Configuration validation tests
│   ├── proxy/
│   │   ├── proxy.go               # Main ReverseProxy type and request handling
│   │   ├── forwarder.go           # HTTP request/response forwarding
│   │   ├── headers.go             # X-Forwarded-* header manipulation
│   │   └── proxy_test.go          # Proxy functionality unit tests
│   ├── backend/
│   │   ├── manager.go             # BackendManager with thread-safe pool operations
│   │   ├── backend.go             # Backend type with health and stats tracking
│   │   ├── pool.go                # Backend filtering and selection helpers
│   │   └── manager_test.go        # Backend management unit tests
│   ├── algorithm/
│   │   ├── interface.go           # Algorithm interface definition
│   │   ├── roundrobin.go          # Round robin with atomic counter
│   │   ├── weighted.go            # Weighted round robin implementation
│   │   ├── leastconn.go           # Least connections algorithm
│   │   ├── iphash.go              # IP hash for session affinity
│   │   ├── random.go              # Random selection algorithm
│   │   ├── registry.go            # Algorithm registration and factory
│   │   └── algorithms_test.go     # Algorithm behavior unit tests
│   ├── health/
│   │   ├── checker.go             # Periodic health checking logic
│   │   ├── prober.go              # HTTP health probe implementation
│   │   ├── state.go               # Health state tracking and transitions
│   │   └── health_test.go         # Health checking unit tests
│   └── router/
│       ├── router.go              # Request routing coordination
│       ├── errors.go              # Error response generation
│       └── router_test.go         # Integration-style routing tests
├── configs/
│   ├── example.yaml               # Complete example configuration
│   ├── minimal.yaml               # Minimal working configuration
│   └── test.yaml                  # Test configuration for integration tests
├── tests/
│   ├── integration/
│   │   ├── basic_test.go          # End-to-end request flow tests
│   │   ├── health_test.go         # Health checking integration tests
│   │   └── failover_test.go       # Backend failure and recovery tests
│   └── testdata/
│       ├── backend_server.go      # Simple HTTP server for testing
│       └── test_configs/          # Various test configuration files
├── scripts/
│   ├── test-backend.go            # Standalone test HTTP server
│   ├── benchmark.sh               # Load testing script
│   └── health-server.go           # Backend with controllable health endpoint
├── go.mod
├── go.sum
├── Makefile                       # Build, test, and development commands
└── README.md
```

Infrastructure Starter Code

Configuration Loading Infrastructure (`internal/config/config.go`):

```go
package config

import (
    "fmt"
    "os"
    "time"
    "gopkg.in/yaml.v3"
)

// Config represents the complete load balancer configuration
type Config struct {
    Port      int               `yaml:"port" json:"port"`
    Backends  []BackendConfig   `yaml:"backends" json:"backends"`
    Algorithm string            `yaml:"algorithm" json:"algorithm"`
    HealthCheck HealthCheckConfig `yaml:"health_check" json:"health_check"`
}

// BackendConfig represents a single backend server configuration
type BackendConfig struct {
    URL    string `yaml:"url" json:"url"`
    Weight int    `yaml:"weight" json:"weight"`
}

// HealthCheckConfig contains health checking parameters
type HealthCheckConfig struct {
    Enabled            bool          `yaml:"enabled" json:"enabled"`
    Interval           time.Duration `yaml:"interval" json:"interval"`
    Timeout            time.Duration `yaml:"timeout" json:"timeout"`
    HealthyThreshold   int           `yaml:"healthy_threshold" json:"healthy_threshold"`
    UnhealthyThreshold int           `yaml:"unhealthy_threshold" json:"unhealthy_threshold"`
    Path               string        `yaml:"path" json:"path"`
}

// LoadConfig reads and validates configuration from the specified file
func LoadConfig(filename string) (*Config, error) {
    data, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

    var config Config
    if err := yaml.Unmarshal(data, &config); err != nil {
        return nil, fmt.Errorf("failed to parse config file: %w", err)
```

```
    }

    if err := config.Validate(); err != nil {
        return nil, fmt.Errorf("invalid configuration: %w", err)
    }

    return &config, nil
}

// Validate checks configuration values for correctness
func (c *Config) Validate() error {
    if c.Port <= 0 || c.Port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535, got %d", c.Port)
    }

    if len(c.Backends) == 0 {
        return fmt.Errorf("at least one backend must be configured")
    }

    validAlgorithms := map[string]bool{
        "round_robin": true, "weighted_round_robin": true,
        "least_connections": true, "ip_hash": true, "random": true,
    }
    if !validAlgorithms[c.Algorithm] {
        return fmt.Errorf("unsupported algorithm: %s", c.Algorithm)
    }

    // Validate each backend configuration
    for i, backend := range c.Backends {
        if backend.URL == "" {
            return fmt.Errorf("backend %d: URL cannot be empty", i)
        }
        if backend.Weight <= 0 {
            return fmt.Errorf("backend %d: weight must be positive, got %d", i, backend.Weight)
        }
    }

    // Validate health check configuration
    if c.HealthCheck.Enabled {
        if c.HealthCheck.Interval <= 0 {
            return fmt.Errorf("health check interval must be positive")
```

```
        }
        if c.HealthCheck.Timeout <= 0 {
            return fmt.Errorf("health check timeout must be positive")
        }
        if c.HealthCheck.HealthyThreshold <= 0 {
            return fmt.Errorf("healthy threshold must be positive")
        }
        if c.HealthCheck.UnhealthyThreshold <= 0 {
            return fmt.Errorf("unhealthy threshold must be positive")
        }
    }

    return nil
}
```

HTTP Test Backend Server (`tests/testdata/backend_server.go`):

```go
package main

import (
    "encoding/json"
    "flag"
    "fmt"
    "log"
    "net/http"
    "time"
)

type TestBackendServer struct {
    port   int
    name   string
    delay  time.Duration
    healthy bool
    requestCount int
}

func NewTestBackendServer(port int, name string) *TestBackendServer {
    return &TestBackendServer{
        port:   port,
        name:   name,
        delay:  0,
        healthy: true,
    }
}

func (s *TestBackendServer) Start() {
    mux := http.NewServeMux()

    // Main endpoint that returns backend info
    mux.HandleFunc("/", s.handleRequest)

    // Health check endpoint
    mux.HandleFunc("/health", s.handleHealth)

    // Control endpoints for testing
    mux.HandleFunc("/control/delay", s.handleSetDelay)
    mux.HandleFunc("/control/health", s.handleSetHealth)
    mux.HandleFunc("/control/stats", s.handleStats)
```

```
    addr := fmt.Sprintf(":%d", s.port)
    log.Printf("Test backend '%s' starting on %s", s.name, addr)

    server := &http.Server{
        Addr:    addr,
        Handler: mux,
    }

    log.Fatal(server.ListenAndServe())
}

func (s *TestBackendServer) handleRequest(w http.ResponseWriter, r *http.Request) {
    s.requestCount++

    // Simulate processing delay if configured
    if s.delay > 0 {
        time.Sleep(s.delay)
    }

    response := map[string]interface{}{
        "backend": s.name,
        "port":    s.port,
        "method":  r.Method,
        "path":    r.URL.Path,
        "headers": r.Header,
        "count":   s.requestCount,
        "timestamp": time.Now().Unix(),
    }

    w.Header().Set("Content-Type", "application/json")
    w.Header().Set("X-Backend-Name", s.name)
    json.NewEncoder(w).Encode(response)
}

func (s *TestBackendServer) handleHealth(w http.ResponseWriter, r *http.Request) {
    if !s.healthy {
        http.Error(w, "Backend unhealthy", http.StatusServiceUnavailable)
        return
    }

    w.WriteHeader(http.StatusOK)
```

```
    w.Write([]byte("OK"))
}

func (s *TestBackendServer) handleSetDelay(w http.ResponseWriter, r *http.Request) {
    delayStr := r.URL.Query().Get("ms")
    if delayStr == "" {
        s.delay = 0
    } else {
        if delayMs := parseInt(delayStr); delayMs > 0 {
            s.delay = time.Duration(delayMs) * time.Millisecond
        }
    }
    fmt.Fprintf(w, "Delay set to %v", s.delay)
}

func (s *TestBackendServer) handleSetHealth(w http.ResponseWriter, r *http.Request) {
    healthStr := r.URL.Query().Get("healthy")
    s.healthy = healthStr != "false"
    fmt.Fprintf(w, "Health set to %v", s.healthy)
}

func (s *TestBackendServer) handleStats(w http.ResponseWriter, r *http.Request) {
    stats := map[string]interface{}{
        "backend": s.name,
        "port":    s.port,
        "healthy": s.healthy,
        "delay":   s.delay.String(),
        "requests": s.requestCount,
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(stats)
}

func parseInt(s string) int {
    var result int
    fmt.Sscanf(s, "%d", &result)
    return result
}

func main() {
    port := flag.Int("port", 8081, "Port to listen on")
    name := flag.String("name", "backend", "Backend name")
```

```
    flag.Parse()


    server := NewTestBackendServer(*port, *name)

    server.Start()

}
```

Core Logic Skeleton Code

Backend Manager Core (`internal/backend/manager.go`):

```go
package backend

import (
    "net/url"
    "sync"
    "sync/atomic"
    "time"
)

// Backend represents a backend server with health and statistics tracking
type Backend struct {
    URL              *url.URL
    Weight           int
    healthy          int64  // atomic: 1 for healthy, 0 for unhealthy
    activeConnections int64 // atomic: current active connection count
    totalRequests    int64  // atomic: total requests processed

    // Health check state
    consecutiveFailures int
    consecutiveSuccesses int
    lastHealthCheck     time.Time

    mutex sync.RWMutex // protects health check state
}

// BackendManager manages the pool of backend servers
type BackendManager struct {
    backends []*Backend
    mutex    sync.RWMutex
}

// NewBackendManager creates a new backend manager
func NewBackendManager() *BackendManager {
    return &BackendManager{
        backends: make([]*Backend, 0),
    }
}

// AddBackend adds a new backend to the pool
func (bm *BackendManager) AddBackend(config BackendConfig) error {
    // TODO 1: Parse the backend URL and validate it's reachable
    // TODO 2: Create new Backend struct with initial healthy state
    // TODO 3: Acquire write lock and append to backends slice
```

```
    // TODO 4: Return nil on success or error if URL parsing fails
    // Hint: Use url.Parse() to validate the backend URL
    panic("implement me")
}

// GetHealthyBackends returns a slice of currently healthy backends
func (bm *BackendManager) GetHealthyBackends() []*Backend {
    // TODO 1: Acquire read lock to safely access backends slice
    // TODO 2: Iterate through all backends and check healthy status
    // TODO 3: Create result slice containing only healthy backends
    // TODO 4: Release lock and return filtered slice
    // Hint: Use atomic.LoadInt64() to read healthy status safely
    panic("implement me")
}

// UpdateBackendHealth updates a backend's health status
func (bm *BackendManager) UpdateBackendHealth(backend *Backend, healthy bool) error {
    // TODO 1: Acquire lock on the specific backend (not the manager)
    // TODO 2: Update consecutive success/failure counters
    // TODO 3: Set healthy atomic flag based on threshold logic
    // TODO 4: Update lastHealthCheck timestamp
    // Hint: Use atomic.StoreInt64() to update healthy flag safely
    panic("implement me")
}

// IsHealthy returns whether the backend is currently healthy
func (b *Backend) IsHealthy() bool {
    // TODO: Use atomic.LoadInt64() to read healthy flag safely
    panic("implement me")
}

// IncrementConnections increments the active connection count
func (b *Backend) IncrementConnections() {
    // TODO: Use atomic.AddInt64() to safely increment activeConnections
    panic("implement me")
}

// DecrementConnections decrements the active connection count
func (b *Backend) DecrementConnections() {
    // TODO: Use atomic.AddInt64() with negative value to safely decrement
    panic("implement me")
}
```

```go
// GetConnectionCount returns the current active connection count

func (b *Backend) GetConnectionCount() int64 {

    // TODO: Use atomic.LoadInt64() to read activeConnections safely

    panic("implement me")

}
```

Algorithm Interface and Round Robin (`internal/algorithm/interface.go` and `roundrobin.go`):

GO

```go
// internal/algorithm/interface.go

package algorithm


// Algorithm defines the interface for load balancing algorithms

type Algorithm interface {

    // SelectBackend chooses a backend from the healthy backends list

    SelectBackend(backends []*Backend) *Backend



    // Name returns the algorithm name for identification

    Name() string

}


// internal/algorithm/roundrobin.go

package algorithm


import (

    "sync/atomic"

)


// RoundRobinAlgorithm implements simple round-robin selection

type RoundRobinAlgorithm struct {

    counter uint64 // atomic counter for thread-safe access

}


// NewRoundRobinAlgorithm creates a new round-robin algorithm instance

func NewRoundRobinAlgorithm() *RoundRobinAlgorithm {

    return &RoundRobinAlgorithm{

        counter: 0,

    }

}


// SelectBackend implements round-robin backend selection

func (rr *RoundRobinAlgorithm) SelectBackend(backends []*Backend) *Backend {

    // TODO 1: Check if backends slice is empty and return nil if so

    // TODO 2: Atomically increment counter using atomic.AddUint64()

    // TODO 3: Use modulo operation to wrap counter within backends slice bounds

    // TODO 4: Return the backend at the calculated index

    // Hint: atomic.AddUint64(&rr.counter, 1) returns the new value

    // Hint: Use (counter - 1) % uint64(len(backends)) to get correct index

    panic("implement me")

}


// Name returns the algorithm name

func (rr *RoundRobinAlgorithm) Name() string {
```

```
    return "round_robin"
}
```

Language-Specific Implementation Hints

Go Concurrency Patterns:

- Use `sync.RWMutex` for backend pool access - allows multiple readers but exclusive writers
- Use `atomic` package operations for simple counters and flags to avoid mutex overhead
- Use `context.Context` for request timeout handling and cancellation propagation
- Use buffered channels for health check result communication between goroutines

HTTP Client Configuration:

```go
// Create HTTP client with appropriate timeouts for backend communication
client := &http.Client{
    Timeout: 30 * time.Second,
    Transport: &http.Transport{
        MaxIdleConns:        100,
        MaxIdleConnsPerHost: 10,
        IdleConnTimeout:     90 * time.Second,
        DisableKeepAlives:   false,
    },
}
```

Error Handling Best Practices:

- Wrap errors with context using `fmt.Errorf("operation failed: %w", err)`
- Define custom error types for different failure modes (backend unavailable, timeout, etc.)
- Use structured logging with `log/slog` to include request context in error logs
- Return appropriate HTTP status codes (502 for backend errors, 503 for no backends available)

Testing Utilities:

- Use `httptest.NewServer()` to create mock backends for unit tests
- Use `net/http/httptest.NewRecorder()` to capture HTTP responses in tests
- Use table-driven tests for algorithm validation with different backend configurations
- Use `t.Parallel()` for tests that don't share state to improve test performance

Milestone Checkpoints

Milestone 1 Checkpoint - HTTP Proxy Foundation:

```bash
# Start the load balancer with single backend
go run cmd/loadbalancer/main.go -config configs/minimal.yaml

# Test basic proxying (should return backend response)
curl -v http://localhost:8080/test

# Expected: Response from backend server with proxy headers added
# Look for: X-Forwarded-For header in backend logs
# Status: 200 OK with JSON response from backend
```

Milestone 2 Checkpoint - Round Robin Distribution:

```bash
# Start multiple test backends
go run tests/testdata/backend_server.go -port 8081 -name backend1 &
go run tests/testdata/backend_server.go -port 8082 -name backend2 &
go run tests/testdata/backend_server.go -port 8083 -name backend3 &

# Send multiple requests and verify distribution
for i in {1..9}; do
    curl -s http://localhost:8080/ | jq -r '.backend'
done

# Expected output: backend1, backend2, backend3, backend1, backend2, backend3...
# Verify: Each backend gets exactly 3 requests in round-robin order
```

Milestone 3 Checkpoint - Health Checks:

```bash
# Make one backend unhealthy
curl "http://localhost:8081/control/health?healthy=false"

# Wait for health check interval to detect failure
sleep 10

# Send requests - should only go to healthy backends
for i in {1..6}; do
    curl -s http://localhost:8080/ | jq -r '.backend'
done

# Expected: Only backend2 and backend3 responses
# Verify: No requests reach backend1 while it's unhealthy
```

Milestone 4 Checkpoint - Multiple Algorithms:

```bash
# Test least connections algorithm
# Update config to use "least_connections" algorithm and restart

# Create load on one backend
curl "http://localhost:8081/control/delay?ms=1000" # Add 1s delay
curl http://localhost:8080/ & # Start slow request

# Send more requests immediately
for i in {1..3}; do
    curl -s http://localhost:8080/ | jq -r '.backend'
done

# Expected: New requests avoid backend1 (has active connection)
# Verify: backend2 and backend3 get the new requests
```

Data Model

> **Milestone(s):** All milestones (1-4) — the data model established here supports HTTP proxy functionality (Milestone 1), request distribution across backends (Milestone 2), health checking implementation (Milestone 3), and multiple algorithm support (Milestone 4)

Mental Model: Medical Patient Chart

Think of the load balancer's data model like a comprehensive medical patient chart system. Each **backend server** is like a patient with a complete medical record — we track their identity (address, capabilities), current health status (vital signs), medical history (past health check results), and care instructions (weight for treatment priority). The **health state model** is like the vital signs monitoring system — it continuously tracks each patient's condition, records when they're healthy or sick, and maintains a timeline of their health changes. The **configuration schema** acts like the hospital's treatment protocols — it defines how often to check patients, what constitutes healthy vs unhealthy readings, which treatment algorithms to use, and how to prioritize care based on patient needs.

Just as a hospital must maintain accurate, up-to-date records for each patient while following standardized protocols for care delivery, our load balancer maintains detailed state for each backend while following configurable rules for traffic distribution and health management. The data structures must be precise, consistent, and accessible to all components that need to make routing and health decisions.

The Data Architecture Challenge

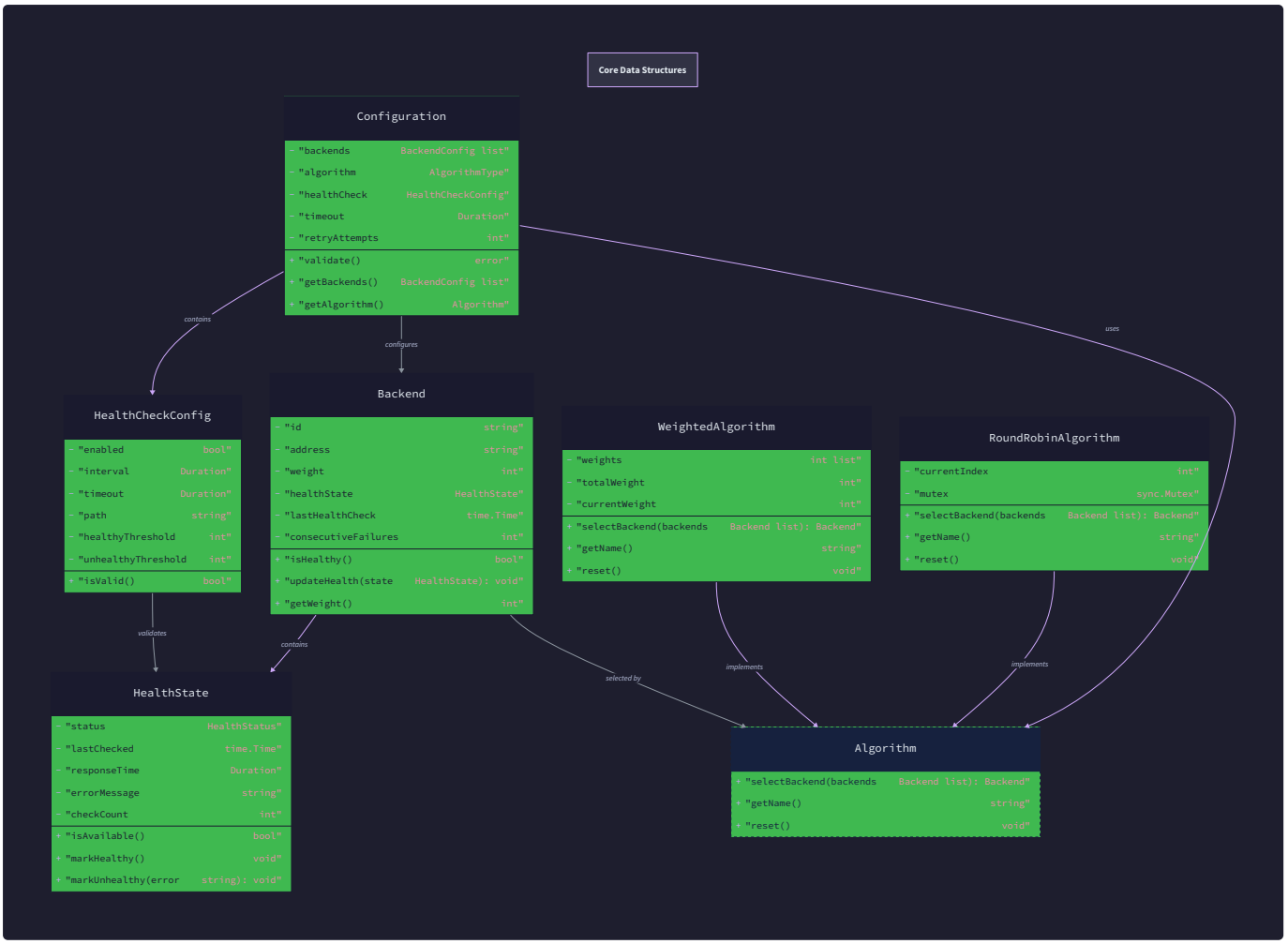
The load balancer's effectiveness hinges on maintaining accurate, consistent state about backend servers while supporting real-time decision making under high request volume. The primary data modeling challenges include:

State Consistency: Multiple goroutines simultaneously read backend health states for routing decisions while health checkers update the same state based on probe results. The data model must prevent race conditions while maintaining performance under concurrent access patterns.

Configuration Flexibility: The system must support runtime configuration changes without service interruption. This requires data structures that can be atomically swapped, validated incrementally, and merged with existing state while preserving ongoing connections and health check schedules.

Health State Accuracy: Health information becomes stale quickly in distributed systems. The data model must capture not just current health status, but health history, check timestamps, and transition patterns to enable sophisticated failure detection and recovery logic.

Algorithm Extensibility: Different load balancing algorithms require different metadata about backends. Round robin needs simple ordering, weighted algorithms need numeric weights, least connections needs active connection counts, and hash-based algorithms need consistent identification. The backend model must accommodate these varied requirements without algorithm-specific coupling.



The data model consists of three primary layers: the **Backend Server Model** that represents individual servers with all their associated state, the **Health State Model** that captures the current and historical health status of each backend, and the **Configuration Schema** that defines how the load balancer should operate. These models interact through well-defined interfaces that support both high-frequency read operations (request routing) and periodic write operations (health updates, configuration changes).

Backend Server Model

The backend server model represents an individual application server that can receive proxied requests from the load balancer. This model must capture all information necessary for routing decisions, health tracking, connection management, and algorithm-specific metadata.

Core Backend Structure

Field Name	Type	Description
ID	string	Unique identifier generated from URL, used for consistent hashing and logging
URL	*url.URL	Parsed backend server address including scheme, host, and port
Weight	int	Relative weight for weighted round robin algorithms (default: 1)
ActiveConnections	*int64	Current number of active HTTP connections (atomic access required)
HealthState	*HealthState	Current health status and check history (pointer for atomic updates)
CreatedAt	time.Time	Timestamp when backend was added to pool
LastUsed	*time.Time	Most recent request routing timestamp (atomic updates)
Metadata	map[string]interface{}	Algorithm-specific data storage for extensibility

The `Backend` structure serves as the central representation of a backend server throughout its lifecycle in the load balancer. The ID field provides a stable identifier that persists across configuration reloads and enables consistent hash calculations for session affinity algorithms. The URL field stores the parsed destination address, eliminating repeated parsing during request forwarding.

> **Design Insight**: Using atomic pointers for frequently updated fields like `ActiveConnections` and `LastUsed` prevents race conditions during concurrent request processing while maintaining read performance for routing decisions.

The weight field supports proportional traffic distribution in weighted algorithms. A weight of 0 effectively disables the backend without removing it from the pool, while higher weights increase the backend's share of traffic proportionally. The metadata map provides an extension point for algorithms that need custom state without modifying the core backend structure.

Connection Tracking Interface

Method Name	Parameters	Returns	Description
IncrementConnections	backend *Backend	int64	Atomically increments active connection count and returns new value
DecrementConnections	backend *Backend	int64	Atomically decrements active connection count and returns new value
GetActiveConnections	backend *Backend	int64	Reads current active connection count atomically
SetLastUsed	backend *Backend, timestamp time.Time	none	Atomically updates the last request timestamp
GetLastUsed	backend *Backend	time.Time	Reads the last used timestamp atomically

Connection tracking enables the least connections algorithm while providing observability into backend utilization. The atomic operations ensure accurate counts even under high concurrency. The increment and decrement methods return the new count to enable immediate algorithm decisions without additional reads.

> **Architecture Decision: Atomic Counters vs Mutex Protection**
>
> - **Context**: Active connection counts are updated on every request and need protection from race conditions
> - **Options Considered**:
>   1. Mutex-protected integer fields with read/write locking
>   2. Atomic int64 operations with pointer-based access
>   3. Channel-based counter service with dedicated goroutine
> - **Decision**: Atomic int64 operations with pointer-based access
> - **Rationale**: Atomic operations provide lock-free performance for simple counters, eliminating contention bottlenecks during high request volume. Mutex protection adds overhead for every request, while channels introduce unnecessary complexity and latency.
> - **Consequences**: Enables lock-free connection counting but limits counter operations to simple increment/decrement/read patterns. More complex counter logic would require mutex protection.

Backend State Management

Method Name	Parameters	Returns	Description
NewBackend	config BackendConfig	*Backend, error	Creates new backend from configuration with initial healthy state
Clone	backend *Backend	*Backend	Creates deep copy for safe concurrent access during updates
IsHealthy	backend *Backend	bool	Returns current health status thread-safely
String	backend *Backend	string	Returns human-readable representation for logging

The `NewBackend` constructor validates the configuration and initializes the backend with default values. It parses the URL, validates the scheme (http/https), and sets the initial health state to healthy with no check history. This optimistic initialization allows new backends to receive traffic immediately while health checking begins in the background.

The `Clone` method supports atomic backend pool updates during configuration reloads. Rather than modifying backends in-place, configuration changes create new backend instances and atomically swap the entire pool. This prevents partial state inconsistencies during updates.

> **Critical Implementation Detail**: The `IsHealthy` method must read the health state atomically to prevent seeing partial health state updates during health check transitions. Use atomic pointer loads to ensure consistent health state reading.

Health State Model

The health state model tracks the current and historical health status of backend servers. This model enables sophisticated failure detection, recovery logic, and health-based routing decisions while providing observability into backend reliability patterns.

Core Health State Structure

Field Name	Type	Description
Healthy	bool	Current health status (true = healthy, false = unhealthy)
ConsecutiveSuccesses	int	Number of consecutive successful health checks
ConsecutiveFailures	int	Number of consecutive failed health checks
LastCheckTime	time.Time	Timestamp of most recent health check attempt
LastSuccessTime	time.Time	Timestamp of most recent successful health check
LastFailureTime	time.Time	Timestamp of most recent failed health check
LastError	string	Error message from most recent failed health check
TotalChecks	int64	Lifetime total number of health checks performed
TotalSuccesses	int64	Lifetime total number of successful health checks
TotalFailures	int64	Lifetime total number of failed health checks
StateChanges	[]HealthStateChange	History of health state transitions (limited size)

The health state maintains both current status and historical patterns to enable intelligent failure detection. The consecutive counters drive threshold-based state transitions, while the timestamp fields provide timing information for health check scheduling and debugging.

> **Design Insight**: Separating consecutive counts from total counts allows threshold-based decisions (consecutive failures) while maintaining long-term reliability metrics for monitoring and capacity planning.

The error message field captures diagnostic information from failed health checks, enabling operators to distinguish between different failure types (connection refused, timeout, HTTP 5xx responses). The state changes history provides an audit trail of health transitions for debugging intermittent issues.

Health State Change Tracking

Field Name	Type	Description
Timestamp	time.Time	When the state change occurred
PreviousState	bool	Health status before the change
NewState	bool	Health status after the change
Reason	string	Cause of the state change (threshold reached, manual override)
CheckCount	int	Number of consecutive checks that triggered this change

State change tracking enables analysis of backend reliability patterns and health check effectiveness. Operators can identify backends that flap between healthy and unhealthy states, indicating network issues or marginal capacity problems.

The reason field distinguishes between automatic transitions (threshold reached) and manual interventions (administrative actions). This information helps operators understand whether state changes reflect actual backend issues or operational decisions.

Health State Transition Interface

Method Name	Parameters	Returns	Description
RecordSuccess	state *HealthState, timestamp time.Time	bool	Records successful check, returns true if state changed to healthy
RecordFailure	state *HealthState, timestamp time.Time, error string	bool	Records failed check, returns true if state changed to unhealthy
ShouldTransitionToHealthy	state *HealthState, threshold int	bool	Checks if consecutive successes meet healthy threshold
ShouldTransitionToUnhealthy	state *HealthState, threshold int	bool	Checks if consecutive failures meet unhealthy threshold
GetHealthSummary	state *HealthState	HealthSummary	Returns current health metrics for monitoring

The record methods update counters and timestamps while checking for state transitions based on configurable thresholds. They return boolean flags indicating whether the health status changed, enabling health checkers to log transitions and notify other components.

Architecture Decision: Threshold-Based State Transitions

- **Context**: Health checks can be noisy, with occasional false positives due to network hiccups or temporary backend overload
- **Options Considered**:
  1. Immediate state changes on single check results
  2. Threshold-based transitions requiring consecutive successes/failures
  3. Sliding window analysis of check history with percentage thresholds
- **Decision**: Threshold-based transitions with consecutive check counting
- **Rationale**: Consecutive thresholds provide simple, predictable behavior while filtering out transient issues. Operators can tune thresholds based on their reliability requirements without complex configuration. Sliding windows add complexity without significantly better filtering for most use cases.
- **Consequences**: Reduces false positive state changes but may delay detection of actual failures. Requires careful threshold tuning for different backend types and network conditions.

Health Metrics Aggregation

Field Name	Type	Description
SuccessRate	float64	Percentage of successful checks over lifetime
RecentSuccessRate	float64	Success rate over last 100 checks
AverageResponseTime	time.Duration	Mean health check response time
UptimePercentage	float64	Percentage of time backend has been healthy
StateChangeCount	int	Total number of health state transitions

Health metrics provide observability into backend reliability and health check effectiveness. The success rate calculations help identify backends with marginal reliability, while response time metrics can indicate performance degradation before hard failures occur.

The recent success rate uses a sliding window to provide more current reliability information than lifetime statistics. This helps identify backends that have recently become unstable even if their historical success rate remains high.

Configuration Schema

The configuration schema defines all operational parameters for the load balancer, including backend server definitions, algorithm selection, health check settings, and runtime behavior controls. The schema must support validation, hot reloading, and extensibility for new features.

Primary Configuration Structure

Field Name	Type	Description
Port	int	HTTP port for the load balancer to listen on (required)
Backends	[]BackendConfig	List of backend server configurations (minimum 1 required)
Algorithm	string	Load balancing algorithm name (default: "round_robin")
HealthCheck	HealthCheckConfig	Health checking configuration and settings
Proxy	ProxyConfig	HTTP proxy behavior and timeout settings
Logging	LoggingConfig	Logging level and output configuration
Metrics	MetricsConfig	Monitoring and metrics collection settings

The top-level configuration provides all settings needed to operate the load balancer. The port field defines where the load balancer accepts incoming requests, while the backends list specifies where requests should be forwarded. The algorithm field selects which load balancing strategy to use from the available implementations.

Design Insight: Grouping related settings into sub-configuration objects (HealthCheckConfig, ProxyConfig) improves maintainability and enables focused validation logic for each functional area.

Backend Configuration Structure

Field Name	Type	Description
URL	string	Backend server URL including scheme and port (required)
Weight	int	Relative weight for weighted algorithms (default: 1, minimum: 0)
MaxConnections	int	Maximum concurrent connections (0 = unlimited)
ConnectTimeout	time.Duration	TCP connection timeout (default: 5s)
ResponseTimeout	time.Duration	HTTP response timeout (default: 30s)
Enabled	bool	Whether backend accepts traffic (default: true)

Backend configuration defines the connection parameters and operational settings for each backend server. The URL must include the scheme (http/https) and may include a custom port. The weight field enables proportional traffic distribution in weighted algorithms.

The timeout settings allow per-backend tuning based on the backend's characteristics. Backends serving complex queries may need longer response timeouts, while simple APIs can use shorter timeouts for faster failure detection.

> **Architecture Decision: Per-Backend vs Global Timeouts**
>
> - **Context**: Different backend services may have different performance characteristics and timeout requirements
> - **Options Considered**:
>   1. Global timeout settings applied to all backends uniformly
>   2. Per-backend timeout configuration with global defaults
>   3. Automatic timeout adjustment based on response time history
> - **Decision**: Per-backend timeout configuration with global defaults
> - **Rationale**: Backend services often have different response time profiles (database queries vs static content), requiring different timeout values. Per-backend settings provide necessary flexibility while global defaults simplify configuration for homogeneous backends.
> - **Consequences**: Enables optimal timeout tuning for mixed backend environments but increases configuration complexity. Operators must understand each backend's performance characteristics.

Health Check Configuration Structure

Field Name	Type	Description
Enabled	bool	Whether to perform active health checking (default: true)
Interval	time.Duration	Time between health checks (default: 10s)
Timeout	time.Duration	Health check request timeout (default: 5s)
HealthyThreshold	int	Consecutive successes needed to mark healthy (default: 2)
UnhealthyThreshold	int	Consecutive failures needed to mark unhealthy (default: 3)
Path	string	HTTP path for health check requests (default: "/health")
ExpectedStatus	int	Expected HTTP status code for success (default: 200)
UserAgent	string	User-Agent header for health check requests

Health check configuration controls how the load balancer monitors backend server availability. The interval and timeout settings determine the frequency and responsiveness of health monitoring, while the threshold values control the sensitivity to transient issues.

The path and expected status fields allow health checks to target specific health endpoints rather than the general application path. Many applications provide dedicated health check endpoints that verify database connectivity, external dependencies, and other critical components.

Algorithm Configuration Options

Algorithm Name	Required Config	Optional Config	Description
round_robin	None	None	Simple sequential rotation through backends
weighted_round_robin	Backend weights	None	Proportional distribution based on backend weights
least_connections	None	Connection tracking	Routes to backend with fewest active connections
ip_hash	None	Hash function	Consistent client-to-backend mapping based on IP
random	None	Random seed	Random backend selection for each request

Algorithm configuration varies based on the selected algorithm's requirements. Simple algorithms like round robin need no additional configuration, while weighted algorithms require weight values for each backend. The configuration schema accommodates these differences through optional fields and algorithm-specific validation.

The IP hash algorithm supports different hash functions (consistent hashing, simple modulo) through optional configuration. This allows tuning the distribution characteristics and handling backend pool changes gracefully.

> **Critical Configuration Validation**: The configuration system must validate that weighted algorithms have weight values for all backends, and that threshold values are positive integers with healthy threshold ≤ unhealthy threshold to prevent configuration errors that could disable health checking.

Configuration Validation Rules

Validation Rule	Field(s)	Error Condition	Resolution
Port Range	Port	Port < 1 or Port > 65535	Must be valid TCP port number
Backend URL Format	Backends[].URL	Invalid URL or missing scheme	Must be valid HTTP/HTTPS URL
Algorithm Validity	Algorithm	Unknown algorithm name	Must match available algorithm implementations
Threshold Logic	HealthyThreshold, UnhealthyThreshold	Healthy > Unhealthy	Healthy threshold must be ≤ unhealthy threshold
Minimum Backends	Backends	Empty backends list	Must have at least one backend configured
Weight Range	Backends[].Weight	Weight < 0	Weight must be non-negative integer
Timeout Values	Various timeout fields	Timeout ≤ 0	All timeouts must be positive durations

Configuration validation prevents common misconfigurations that could cause runtime failures or unexpected behavior. The validation logic runs during configuration loading and before applying configuration updates, ensuring only valid configurations reach the running system.

The threshold logic validation prevents impossible health check configurations where backends could never transition to healthy state. URL validation ensures backend addresses can be parsed and used for HTTP requests.

Configuration Update Interface

Method Name	Parameters	Returns	Description
LoadConfig	filename string	*Config, error	Loads configuration from file with full validation
Validate	config *Config	error	Validates all configuration fields and dependencies
MergeConfig	current *Config, updates *Config	*Config, error	Merges partial updates with existing configuration
CompareConfigs	config1, config2 *Config	ConfigDiff	Identifies differences between configurations
ApplyUpdates	current *Config, diff ConfigDiff	*Config, error	Applies validated changes to create new configuration

The configuration interface supports both full configuration loading and incremental updates. The merge functionality enables partial configuration updates where only changed fields are specified, reducing the risk of accidentally reverting unrelated settings.

Configuration comparison identifies what changed between versions, enabling targeted updates that only affect modified components. For example, changing only health check settings should not restart the algorithm selector or rebuild backend connection pools.

Common Pitfalls

⚠️ Pitfall: Race Conditions in Health State Updates

Many implementations access health state fields directly without proper synchronization, leading to corrupted state during concurrent health checks and routing decisions. For example, reading the `Healthy` boolean while another goroutine updates consecutive failure counts can see inconsistent state where a backend appears healthy but has high failure counts.

Why it's wrong: Concurrent access to shared state without synchronization causes data races that can lead to incorrect routing decisions, such as sending traffic to unhealthy backends or incorrectly marking healthy backends as failed.

How to fix: Use atomic operations for simple fields or protect the entire health state with a read-write mutex. Better yet, make health state immutable and use atomic pointer swaps for updates, ensuring readers always see consistent state.

⚠️ Pitfall: Configuration Validation After Startup

Some implementations only validate configuration during initial loading, allowing invalid configurations to be applied during runtime updates. This can cause the load balancer to fail when applying configuration changes, potentially leaving it in an inconsistent state.

Why it's wrong: Runtime configuration updates without validation can introduce invalid settings that crash the load balancer or cause subtle behavioral issues that are difficult to diagnose.

How to fix: Always validate configuration changes before applying them. Create a staging configuration, validate all fields and dependencies, then atomically swap to the new configuration only if validation succeeds. Keep the previous valid configuration as a rollback option.

⚠️ Pitfall: Memory Leaks in Connection Tracking

Implementations often increment connection counters on request start but fail to decrement them on all exit paths (normal completion, client disconnect, backend failure). This causes connection counts to grow indefinitely, breaking the least connections algorithm.

Why it's wrong: Inaccurate connection counts cause traffic distribution problems and eventually integer overflow. The least connections algorithm becomes useless when counts don't reflect actual connections.

How to fix: Use defer statements or finally blocks to ensure connection counters are decremented on all exit paths. Implement connection count auditing that periodically verifies counts match actual connection states and resets counters if they diverge significantly.

⚠️ Pitfall: Health State History Memory Growth

Storing unlimited health check history in the `StateChanges` slice causes memory usage to grow without bound over time, especially for backends with frequent state changes due to intermittent issues.

Why it's wrong: Unbounded memory growth eventually causes the load balancer to run out of memory, particularly in environments with many backends or frequent health state transitions.

How to fix: Limit the state change history to a fixed number of recent entries (e.g., last 100 changes) using a circular buffer or by trimming old entries when adding new ones. Archive detailed history to external monitoring systems if needed for analysis.

⚠️ Pitfall: Backend ID Collisions

Generating backend IDs from URLs without proper normalization can create collisions when the same backend is specified with different URL formats (e.g., "http://api.example.com" vs "http://api.example.com/"). This causes backends to overwrite each other during configuration updates.

Why it's wrong: Backend collisions result in lost configuration, unpredictable behavior during updates, and difficulty tracking specific backends in logs and metrics.

How to fix: Normalize URLs before generating IDs by parsing the URL, standardizing the scheme/host/port format, and removing trailing slashes. Use the normalized form consistently for ID generation and backend comparison.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Configuration Format	JSON with encoding/json	YAML with gopkg.in/yaml.v3
Configuration Validation	Manual field checking	Schema validation with go-playground/validator
Concurrent State Access	sync.RWMutex protection	Atomic operations with sync/atomic
URL Parsing	net/url.Parse	Custom URL validation with regex
Time Handling	time.Time with time.Duration	Custom time types with validation
Configuration Hot Reload	File watching with fsnotify	Configuration API with HTTP endpoints

Recommended File Structure

```
internal/
  models/
    backend.go              ← Backend and HealthState definitions
    config.go               ← Configuration structures and validation
    health.go               ← Health checking data types
  config/
    loader.go               ← Configuration file loading and parsing
    validator.go            ← Configuration validation logic
    watcher.go              ← Hot reload file watching (optional)
```

Infrastructure Starter Code

Backend Model Implementation (backend.go):

```go
package models

import (
    "net/url"
    "sync/atomic"
    "time"
)

// Backend represents a backend server with health and connection tracking
type Backend struct {
    ID               string
    URL              *url.URL
    Weight           int
    activeConnections *int64
    healthState      atomic.Pointer[HealthState]
    CreatedAt        time.Time
    lastUsed         *int64 // Unix timestamp for atomic access
    Metadata         map[string]interface{}
}

// HealthState tracks backend health status and check history
type HealthState struct {
    Healthy              bool
    ConsecutiveSuccesses  int
    ConsecutiveFailures   int
    LastCheckTime        time.Time
    LastSuccessTime      time.Time
    LastFailureTime      time.Time
    LastError            string
    TotalChecks          int64
    TotalSuccesses       int64
    TotalFailures        int64
    StateChanges         []HealthStateChange
}

// HealthStateChange records a health state transition
type HealthStateChange struct {
    Timestamp     time.Time
    PreviousState bool
    NewState      bool
    Reason        string
    CheckCount    int
```

```go
}

// NewBackend creates a new backend from configuration
func NewBackend(config BackendConfig) (*Backend, error) {
    // TODO 1: Parse and validate the backend URL
    // TODO 2: Generate unique ID from normalized URL
    // TODO 3: Initialize health state as healthy with no history
    // TODO 4: Set up atomic counters for connections and last used time
    // TODO 5: Apply default weight if not specified
    return nil, nil
}

// Connection tracking methods
func (b *Backend) IncrementConnections() int64 {
    // TODO: Atomically increment connection counter and return new value
    return 0
}

func (b *Backend) DecrementConnections() int64 {
    // TODO: Atomically decrement connection counter and return new value
    return 0
}

func (b *Backend) GetActiveConnections() int64 {
    // TODO: Atomically read current connection count
    return 0
}

// Health state management
func (b *Backend) IsHealthy() bool {
    // TODO: Atomically load health state and return healthy status
    return false
}

func (b *Backend) UpdateHealthState(newState *HealthState) {
    // TODO: Atomically store new health state
}
```

Configuration Model Implementation (config.go):

```go
package models

import (
    "time"
)

// Config represents the complete load balancer configuration
type Config struct {
    Port        int                `json:"port" validate:"min=1,max=65535"`
    Backends    []BackendConfig    `json:"backends" validate:"min=1,dive"`
    Algorithm   string             `json:"algorithm" validate:"oneof=round_robin weighted_round_robin least_connections ip_hash random"`
    HealthCheck HealthCheckConfig  `json:"health_check"`
    Proxy       ProxyConfig        `json:"proxy"`
    Logging     LoggingConfig      `json:"logging"`
}

// BackendConfig defines a single backend server
type BackendConfig struct {
    URL             string        `json:"url" validate:"required,url"`
    Weight          int           `json:"weight" validate:"min=0"`
    MaxConnections  int           `json:"max_connections" validate:"min=0"`
    ConnectTimeout  time.Duration `json:"connect_timeout"`
    ResponseTimeout time.Duration `json:"response_timeout"`
    Enabled         bool          `json:"enabled"`
}

// HealthCheckConfig defines health checking behavior
type HealthCheckConfig struct {
    Enabled            bool          `json:"enabled"`
    Interval           time.Duration `json:"interval" validate:"min=1s"`
    Timeout            time.Duration `json:"timeout" validate:"min=100ms"`
    HealthyThreshold   int           `json:"healthy_threshold" validate:"min=1"`
    UnhealthyThreshold int           `json:"unhealthy_threshold" validate:"min=1"`
    Path               string        `json:"path"`
    ExpectedStatus     int           `json:"expected_status" validate:"min=100,max=599"`
    UserAgent          string        `json:"user_agent"`
}

// Additional config structures...
type ProxyConfig struct {
    ReadTimeout     time.Duration `json:"read_timeout"`
    WriteTimeout    time.Duration `json:"write_timeout"`
    IdleTimeout     time.Duration `json:"idle_timeout"`
```

```
    MaxHeaderBytes  int           `json:"max_header_bytes"`
}

type LoggingConfig struct {
    Level  string `json:"level" validate:"oneof=debug info warn error"`
    Format string `json:"format" validate:"oneof=json text"`
    Output string `json:"output"`
}

// Validate performs comprehensive configuration validation
func (c *Config) Validate() error {
    // TODO 1: Validate port range and availability
    // TODO 2: Validate each backend configuration
    // TODO 3: Check algorithm name against available implementations
    // TODO 4: Validate health check threshold relationships
    // TODO 5: Ensure weighted algorithms have weights for all backends
    // TODO 6: Validate timeout values are positive and reasonable
    return nil
}
```

Core Logic Skeleton Code

Health State Management:

```go
// RecordSuccess updates health state after successful check
func (hs *HealthState) RecordSuccess(timestamp time.Time) bool {
    // TODO 1: Update last check and last success timestamps
    // TODO 2: Increment consecutive successes counter and reset failures
    // TODO 3: Update total checks and total successes counters
    // TODO 4: Check if consecutive successes meet healthy threshold
    // TODO 5: If threshold met and currently unhealthy, transition to healthy
    // TODO 6: Record state change if transition occurred
    // TODO 7: Return true if health state changed, false otherwise
    return false
}

// RecordFailure updates health state after failed check
func (hs *HealthState) RecordFailure(timestamp time.Time, errorMsg string) bool {
    // TODO 1: Update last check and last failure timestamps
    // TODO 2: Store error message and increment consecutive failures
    // TODO 3: Reset consecutive successes counter to zero
    // TODO 4: Update total checks and total failures counters
    // TODO 5: Check if consecutive failures meet unhealthy threshold
    // TODO 6: If threshold met and currently healthy, transition to unhealthy
    // TODO 7: Record state change if transition occurred
    // TODO 8: Return true if health state changed, false otherwise
    return false
}
```

Configuration Loading and Validation:

```go
// LoadConfig reads and validates configuration from file
func LoadConfig(filename string) (*Config, error) {
    // TODO 1: Read configuration file contents
    // TODO 2: Parse JSON/YAML into Config struct
    // TODO 3: Apply default values for optional fields
    // TODO 4: Run comprehensive validation on all fields
    // TODO 5: Validate backend URLs and normalize them
    // TODO 6: Check algorithm compatibility with backend configurations
    // TODO 7: Return validated config or detailed error message
    return nil, nil
}

// validateBackendConfig checks individual backend configuration
func validateBackendConfig(config BackendConfig) error {
    // TODO 1: Parse and validate URL format and scheme
    // TODO 2: Check weight value for weighted algorithms
    // TODO 3: Validate timeout values are positive and reasonable
    // TODO 4: Ensure max connections limit is sensible
    // TODO 5: Return first validation error found or nil if valid
    return nil
}
```

Language-Specific Hints

Go-Specific Implementation Tips:

- Use `sync/atomic.Pointer` for lock-free health state updates in Go 1.19+, or `atomic.Value` for earlier versions
- Parse duration strings with `time.ParseDuration("10s")` for human-readable timeout configuration
- Use `net/url.Parse()` for URL validation and normalize with `url.String()` for consistent ID generation
- Implement `fmt.Stringer` interface on Backend and HealthState for readable logging output
- Use struct tags with validation libraries like `github.com/go-playground/validator/v10` for declarative validation
- Leverage `json:",omitempty"` tags to exclude empty fields from configuration output

Atomic Operations for Concurrent Access:

```go
// Reading atomic values
connections := atomic.LoadInt64(b.activeConnections)
lastUsed := time.Unix(atomic.LoadInt64(b.lastUsed), 0)

// Writing atomic values
atomic.StoreInt64(b.lastUsed, time.Now().Unix())
newCount := atomic.AddInt64(b.activeConnections, 1)
```

Milestone Checkpoint

Milestone 1 Checkpoint - Data Model Foundation: After implementing the data model structures, verify:

1. **Backend Creation**: Create backends from configuration and verify all fields are properly initialized

```bash
go test -run TestNewBackend ./internal/models/
```

2. **Atomic Operations**: Test concurrent access to connection counters and health state

```bash
go test -race -run TestConcurrentAccess ./internal/models/
```

3. **Configuration Validation**: Load valid and invalid configurations and verify validation catches errors

```bash
go test -run TestConfigValidation ./internal/config/
```

Expected Behavior:

- New backends should have healthy initial state and zero connections
- Concurrent connection increment/decrement should maintain accurate counts
- Invalid configurations should return specific validation errors
- Health state updates should be atomic and consistent

Signs Something is Wrong:

- Race condition warnings during `go test -race`
- Connection counters that drift or become negative
- Configuration validation that accepts invalid values
- Panics when accessing health state concurrently

HTTP Reverse Proxy Component

> **Milestone(s):** Milestone 1 (HTTP Proxy Foundation) — this component implements the core reverse proxy functionality that forwards HTTP requests between clients and backend servers

Mental Model: Relay Runner

Think of the HTTP reverse proxy as a **relay runner** in a communication chain between two parties who cannot speak directly to each other. Imagine you're at an international conference where a client speaks only English and a backend server speaks only French. The reverse proxy acts as a bilingual interpreter who:

1. **Receives the message** from the English-speaking client, understanding exactly what they want to say
2. **Translates and forwards** the message to the French-speaking backend server, preserving all the nuances and context
3. **Waits for the response** from the backend server, maintaining the connection and context
4. **Translates and relays** the response back to the original client, ensuring nothing is lost in translation

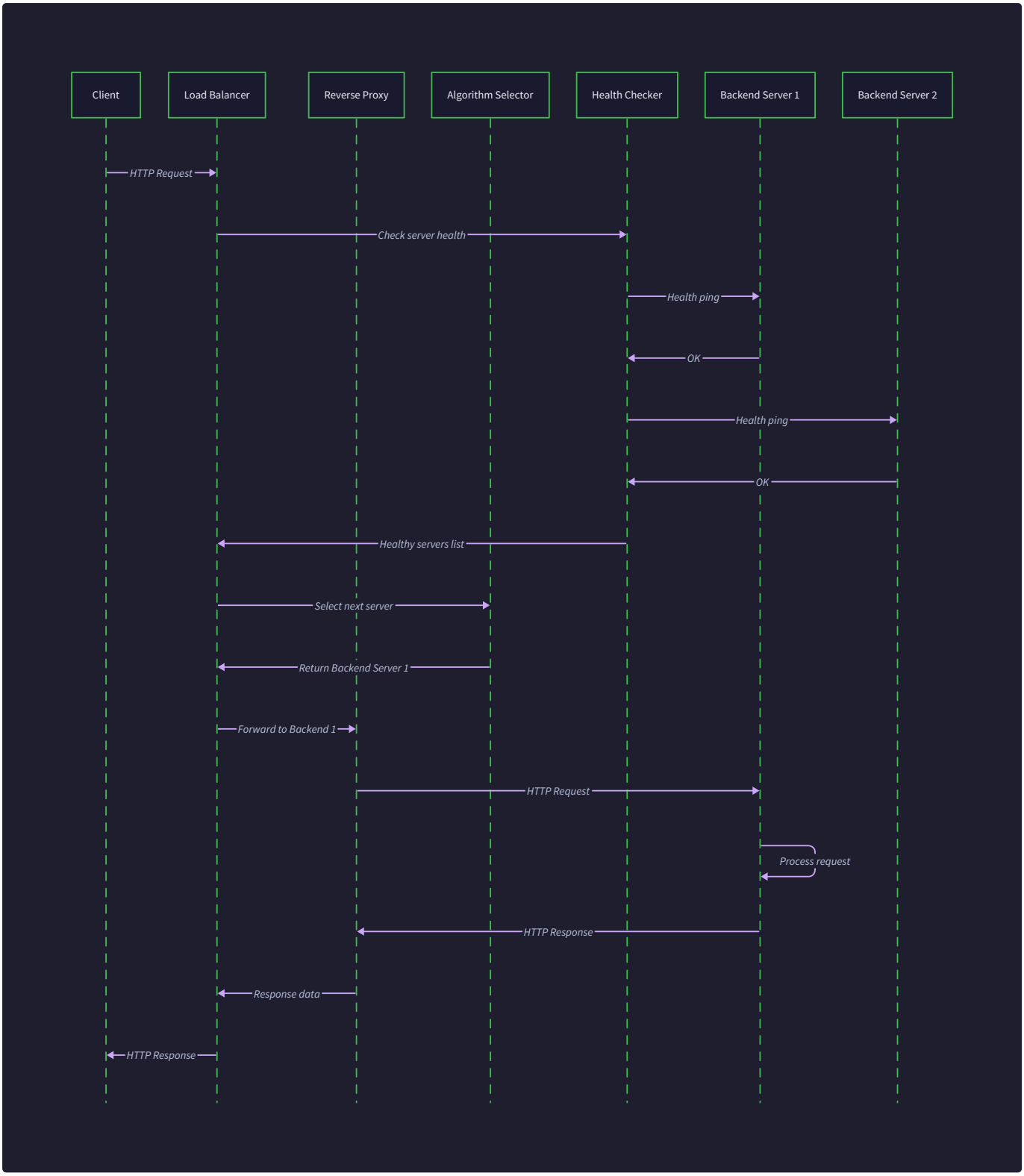
The key insight is that the reverse proxy is **transparent to both parties** — the client thinks they're talking directly to the backend server, and the backend server thinks the proxy is the original client. This transparency is crucial because it allows the load balancer to sit invisibly between clients and servers, distributing traffic without either side needing to be aware of the load balancing happening.

Unlike a forward proxy (which acts on behalf of clients), a reverse proxy acts on behalf of the servers. It's positioned at the server side of the network, accepting requests that clients think are going directly to the origin server, then deciding which actual backend server should handle each request.

The relay runner analogy helps us understand the three critical responsibilities: **message fidelity** (preserving all headers, body content, and HTTP semantics), **connection management** (maintaining efficient communication channels), and **error handling** (gracefully managing situations when the backend server is unavailable or unresponsive).

Request Forwarding Logic

The request forwarding logic forms the heart of the reverse proxy, responsible for accepting incoming HTTP requests and faithfully relaying them to selected backend servers. This process involves careful parsing, modification, and transmission of HTTP messages while preserving their semantic meaning and ensuring proper error handling.



The request forwarding process follows a structured sequence of steps that ensures reliable message transmission:

1. **Accept the incoming connection** from the client on the configured proxy port, establishing a TCP connection that will carry the HTTP request
2. **Parse the HTTP request** using Go's built-in HTTP server capabilities, extracting the method, path, headers, and body into a structured request object
3. **Select the target backend** by consulting the backend manager and load balancing algorithm to determine which server should handle this request
4. **Clone and modify the request** by creating a new HTTP request object with the backend server's URL while preserving the original method, path, and body
5. **Add proxy headers** including `X-Forwarded-For` (client's IP address), `X-Forwarded-Proto` (original protocol), and `X-Forwarded-Host` (original host header)
6. **Establish backend connection** using the HTTP client's connection pool, leveraging keep-alive connections when possible for efficiency
7. **Forward the request** by writing the modified request to the backend server, including all headers and streaming the body content
8. **Handle connection errors** by catching network failures, timeouts, and other transport-level issues that prevent request delivery

The request forwarding mechanism must handle several critical aspects of HTTP semantics correctly. **Method preservation** ensures that GET, POST, PUT, DELETE, and other HTTP methods are forwarded exactly as received, maintaining the intended operation semantics. **Header forwarding** requires careful attention to which headers should be passed through unchanged, which should be modified, and which should be stripped or added by the proxy.

Body handling presents particular challenges because HTTP request bodies can be large and may arrive as streaming data. The proxy must efficiently forward body content without loading entire payloads into memory, using Go's `io.Copy` or similar streaming mechanisms to transfer data directly from the client connection to the backend connection.

The forwarding logic maintains a **request context** throughout the process, allowing for proper timeout handling, cancellation propagation, and resource cleanup. This context carries deadlines from the original client request and enables the proxy to cancel backend requests if the client disconnects or times out.

Request Component	Forwarding Behavior	Modification Required	Error Handling
HTTP Method	Pass through unchanged	None	Invalid methods return 400 Bad Request
Request Path	Pass through unchanged	None	Malformed paths return 400 Bad Request
Query Parameters	Pass through unchanged	None	Invalid encoding returns 400 Bad Request
Host Header	Replace with backend host	Set to backend server address	Missing host returns 400 Bad Request
Content-Length	Pass through if present	None	Mismatched length returns 400 Bad Request
Transfer-Encoding	Pass through unchanged	None	Unsupported encodings return 501 Not Implemented
Authorization	Pass through unchanged	None	Invalid format passed to backend
User-Agent	Pass through unchanged	None	Missing user agent is acceptable
X-Forwarded-For	Add or append client IP	Append to existing or create new	Network errors logged but not fatal
X-Forwarded-Proto	Set to original protocol	Set to "http" or "https"	Default to "http" if uncertain
Connection	Strip and handle internally	Remove hop-by-hop header	Invalid values ignored

> **Key Design Insight**: The proxy must act as a **protocol translator** while maintaining semantic fidelity. Every HTTP request has multiple layers of meaning — the transport layer (TCP connection), the HTTP protocol layer (methods, headers, status codes), and the application layer (body content, authentication tokens). The proxy must preserve meaning at all layers while translating between different network endpoints.

Connection management during request forwarding requires careful attention to resource lifecycle. The proxy must track active requests to prevent resource leaks, implement appropriate timeouts to avoid hanging connections, and properly close connections when errors occur or requests complete.

Error scenarios during request forwarding fall into several categories that require different handling strategies. **Network connectivity errors** occur when the backend server is unreachable, the connection is refused, or DNS resolution fails — these should result in a 502 Bad Gateway response. **Timeout errors** happen when backend servers accept connections but don't respond within configured limits — these also generate 502 responses but may trigger health check updates. **Protocol errors** arise from malformed HTTP responses or unexpected connection closures — these require careful error response generation to avoid leaving clients hanging.

Response Processing

Response processing handles the return path of the HTTP proxy flow, receiving responses from backend servers and relaying them back to the original clients. This process requires careful attention to HTTP status codes, header handling, body streaming, and error recovery to ensure clients receive complete and correct responses.

The response processing workflow mirrors the request forwarding process but operates in reverse, with additional complexity around error response generation and connection cleanup:

1. **Receive the backend response** by reading the HTTP response from the established backend connection, parsing status code, headers, and preparing for body streaming
2. **Validate response integrity** by checking for proper HTTP format, valid status codes, and consistent header values that indicate a well-formed response
3. **Process response headers** by forwarding most headers unchanged while stripping hop-by-hop headers and adding any proxy-specific headers needed by clients
4. **Set response status** by copying the backend's HTTP status code to the client response, ensuring proper status line formatting and reason phrase handling
5. **Stream response body** by efficiently copying the backend response body to the client connection without buffering large payloads in memory
6. **Handle streaming errors** by detecting network failures during body transfer and implementing appropriate error recovery or connection termination
7. **Clean up connections** by properly closing backend connections when appropriate and returning them to connection pools for reuse when possible
8. **Log response completion** by recording the final status, bytes transferred, and timing information for monitoring and debugging purposes

Response header processing requires understanding which headers are safe to forward and which require special handling. **Content headers** like `Content-Type`, `Content-Length`, and `Content-Encoding` must be forwarded exactly as received to ensure proper client interpretation. **Caching headers** including `Cache-`

`Control`, `ETag`, and `Expires` should be passed through to allow client-side and intermediate proxy caching to function correctly.

Security-related headers such as `Set-Cookie`, `Authorization`, and custom authentication headers must be forwarded carefully to maintain application functionality while not exposing sensitive proxy internals. **CORS headers** including `Access-Control-Allow-Origin` and related directives should be passed through unchanged to preserve cross-origin request handling.

Response Header Category	Forwarding Policy	Modification Required	Special Considerations
Content Headers	Forward unchanged	None	Critical for proper client parsing
Caching Headers	Forward unchanged	None	Enables downstream caching
Security Headers	Forward unchanged	None	Required for authentication flows
CORS Headers	Forward unchanged	None	Needed for browser cross-origin requests
Server Headers	Forward unchanged	None	May reveal backend server information
Date Headers	Forward unchanged	None	Important for cache validation
Connection Headers	Strip from response	Remove hop-by-hop headers	Proxy manages connections independently
Proxy Headers	Add if configured	May add Via or X-Forwarded headers	Optional proxy identification

Response body streaming represents a critical performance and resource management concern. The proxy must handle response bodies of arbitrary size without consuming excessive memory, support various `Transfer-Encoding` formats including chunked encoding, and maintain streaming efficiency even for large payloads like file downloads or media content.

The streaming implementation uses Go's `io.Copy` function or similar mechanisms to create a direct pipe between the backend response body reader and the client response writer. This approach avoids loading response content into proxy memory while maintaining high throughput and low latency.

Error response generation becomes necessary when backend servers fail to respond properly or when network errors occur during response processing. The proxy must generate appropriate HTTP error responses that inform clients about the failure while providing sufficient information for debugging without exposing sensitive backend details.

Error Scenario	HTTP Status Code	Response Body	Client Action
Backend connection refused	502 Bad Gateway	"Backend server unavailable"	Retry or use alternative endpoint
Backend connection timeout	502 Bad Gateway	"Backend server timeout"	Retry with longer timeout
Backend returns invalid HTTP	502 Bad Gateway	"Invalid backend response"	Check backend server status
Response body streaming error	Connection closed	Partial response already sent	Client must handle incomplete response
Backend returns 5xx error	Forward status unchanged	Forward backend error response	Client handles backend application error
Proxy internal error	500 Internal Server Error	"Proxy processing error"	Contact proxy administrator

> **Critical Design Principle**: Response processing must maintain **streaming semantics** to support real-time applications, large file transfers, and long-polling connections. Buffering entire responses in proxy memory would create scalability bottlenecks and increase latency, violating the transparency principle of reverse proxying.

Connection lifecycle management during response processing involves coordinating between client and backend connections. When responses complete successfully, backend connections should be returned to the connection pool for reuse when using HTTP keep-alive. When errors occur, connections may need to be closed and discarded to prevent connection state corruption.

The response processing logic must also handle **partial response scenarios** where backend connections fail or close unexpectedly after the response has started streaming to the client. In these cases, the proxy cannot send a different HTTP status code (since headers have already been sent) and must log the error while closing the client connection.

Connection Pool Management

Connection pool management optimizes the reverse proxy's network efficiency by reusing HTTP connections to backend servers, reducing the overhead of establishing new TCP connections for each request. This component manages connection lifecycle, implements keep-alive functionality, and handles connection limits and timeouts to ensure reliable and performant backend communication.

The connection pool operates as a **shared resource manager** that coordinates connection allocation across concurrent requests. When a request needs to communicate with a backend server, it requests a connection from the pool rather than establishing a new one. After the request completes, the connection returns to the pool for reuse by subsequent requests to the same backend.

Go's built-in `http.Client` and `http.Transport` provide robust connection pooling capabilities that the reverse proxy leverages. The `http.Transport` automatically manages connection pools per backend server, implements HTTP/1.1 keep-alive semantics, and handles connection limits and timeouts according to configured parameters.

Connection pool configuration requires balancing several competing concerns: **connection reuse efficiency** (higher limits reduce connection establishment overhead), **resource consumption** (connection pools consume memory and file descriptors), and **backend server protection** (too many connections can overwhelm backend servers).

Configuration Parameter	Purpose	Default Value	Tuning Considerations
MaxIdleConns	Total idle connections across all backends	100	Higher values reduce connection establishment but consume memory
MaxIdleConnsPerHost	Idle connections per backend server	2	Should match expected concurrency per backend
MaxConnsPerHost	Total connections per backend server	Unlimited	Set to protect backend servers from connection exhaustion
IdleConnTimeout	How long to keep idle connections	90 seconds	Longer timeouts improve reuse but consume resources
TLSHandshakeTimeout	Timeout for TLS connection establishment	10 seconds	Shorter timeouts fail faster but may cause false failures
ResponseHeaderTimeout	Timeout for response headers	0 (no timeout)	Should be set to prevent hanging requests
ExpectContinueTimeout	Timeout for 100-continue responses	1 second	Affects POST/PUT request performance

The connection pool implements **connection lifecycle management** through several phases. **Connection establishment** occurs when no idle connections exist for a backend server, triggering TCP connection setup and any required TLS handshaking. **Connection allocation** assigns an available connection to an incoming request, removing it from the idle pool and marking it as active. **Connection return** happens when a request completes successfully with a reusable connection, returning it to the idle pool for future use. **Connection expiry** removes connections that exceed idle timeout limits or encounter errors that make them unsuitable for reuse.

Health-based connection management integrates with the load balancer's health checking system to prevent connection allocation to unhealthy backends. When a backend server is marked unhealthy, the connection pool should drain existing connections and refuse new connection allocations until the backend recovers.

The connection pool must handle **concurrent access patterns** safely, as multiple goroutines will simultaneously request connections for different client requests. Go's `http.Transport` handles this concurrency internally, but the proxy must coordinate between connection pool operations and backend health state changes.

Connection error handling requires distinguishing between transient network issues and persistent backend failures. **Connection refused errors** typically indicate backend server problems and should trigger health check updates. **Timeout errors** may indicate network congestion or backend overload and might warrant connection pool tuning. **Protocol errors** suggest backend server misconfigumentation and may require connection pool draining.

Connection Error Type	Pool Action	Health Check Impact	Recovery Strategy
Connection refused	Mark backend unhealthy	Trigger immediate health check	Wait for health recovery
Connection timeout	Close connection, retry once	No immediate action	May adjust timeout settings
TLS handshake failure	Close connection	Log security concern	Check certificate validity
Protocol error	Close connection	No immediate action	Log for backend investigation
Idle connection closed	Remove from pool	No action	Normal connection lifecycle
DNS resolution failure	No pool action	Mark backend unhealthy	Check DNS configuration

> **Performance Insight**: Connection pooling can reduce request latency by **50-80%** for workloads with frequent requests to the same backends, as it eliminates TCP handshake (typically 1 RTT) and TLS handshake (typically 2-3 RTTs) overhead. However, poorly configured pools can cause connection exhaustion or resource leaks.

Connection pool monitoring provides visibility into pool performance and helps identify configuration issues. Key metrics include **active connection count** (connections currently handling requests), **idle connection count** (connections available for reuse), **connection establishment rate** (new connections created per second), and **connection reuse rate** (percentage of requests using pooled connections).

The connection pool integrates with the proxy's **graceful shutdown** process by providing connection draining functionality. During shutdown, the pool stops accepting new connection requests, allows active requests to complete, and closes all idle connections in an orderly fashion.

Architecture Decisions

The HTTP reverse proxy component involves several critical architecture decisions that significantly impact performance, reliability, and maintainability. These decisions establish the foundation for how the proxy handles HTTP semantics, manages resources, and integrates with other load balancer components.

Decision: HTTP Client Implementation

- **Context**: The proxy needs an HTTP client to communicate with backend servers, with options ranging from low-level socket programming to high-level HTTP libraries
- **Options Considered**: Raw TCP sockets with manual HTTP parsing, Go's net/http package, third-party HTTP clients like fasthttp
- **Decision**: Use Go's standard `net/http` package with customized `http.Transport`
- **Rationale**: The standard library provides robust HTTP/1.1 implementation, built-in connection pooling, comprehensive error handling, and extensive testing. Custom transport configuration allows fine-tuning for proxy-specific needs without reimplementing HTTP protocol details
- **Consequences**: Leverages battle-tested HTTP implementation and reduces development complexity, but limits control over low-level connection behavior and may have higher memory overhead than specialized proxy libraries

HTTP Client Option	Pros	Cons	Performance	Complexity
Raw TCP + Manual HTTP	Maximum control, minimal overhead	High complexity, error-prone parsing	Highest	Very High
Go net/http	Standard library, robust, well-tested	Some overhead, less control	Good	Low
fasthttp Library	Optimized for high performance	Non-standard API, less ecosystem support	Highest	Medium
Custom HTTP Client	Tailored to proxy needs	Development and maintenance overhead	Variable	High

Decision: Header Processing Strategy

- **Context**: HTTP headers require careful handling to maintain protocol correctness while adding proxy functionality
- **Options Considered**: Forward all headers unchanged, selective header filtering, comprehensive header rewriting
- **Decision**: Implement selective header processing with explicit forwarding rules
- **Rationale**: Forwarding all headers unchanged preserves application functionality, while selective processing allows adding proxy-specific headers like X-Forwarded-For. Comprehensive rewriting introduces complexity and potential compatibility issues
- **Consequences**: Maintains application compatibility and simplifies debugging, but may forward headers that expose backend server information or cause client confusion

Decision: Request Body Handling

- **Context**: HTTP request bodies can be large and may arrive as streaming data, requiring efficient forwarding without excessive memory usage
- **Options Considered**: Buffer entire body in memory, stream body with fixed buffer, stream with adaptive buffering
- **Decision**: Use streaming with fixed buffer size and io.Copy for body forwarding
- **Rationale**: Streaming prevents memory exhaustion with large payloads, fixed buffers provide predictable resource usage, and io.Copy is optimized for efficient data transfer in Go
- **Consequences**: Supports arbitrarily large request bodies with bounded memory usage, but may not optimize buffer sizes for different payload characteristics

Body Handling Strategy	Memory Usage	Performance	Complexity	Large File Support
Buffer Entire Body	High (proportional to body size)	Good for small bodies	Low	Poor (memory exhaustion)
Fixed Buffer Streaming	Constant (buffer size)	Good overall	Low	Excellent
Adaptive Buffer Streaming	Variable (optimized per request)	Best	High	Excellent
Zero-Copy Streaming	Minimal	Best (when supported)	Very High	Excellent

Decision: Connection Timeout Strategy

- **Context**: Network operations can hang indefinitely without proper timeouts, but aggressive timeouts cause false failures
- **Options Considered**: Single global timeout, separate timeouts per operation type, dynamic timeout adjustment
- **Decision**: Implement separate configurable timeouts for connection establishment, request sending, and response receiving
- **Rationale**: Different operations have different latency characteristics and failure modes. Connection timeouts should be short (network reachable quickly), while response timeouts may need to accommodate backend processing time
- **Consequences**: Provides fine-grained control over failure detection speed, but requires careful tuning and increases configuration complexity

Decision: Error Response Generation

- **Context**: When backend servers fail or return errors, the proxy must decide what information to include in error responses to clients
- **Options Considered**: Forward all backend errors unchanged, generate generic proxy errors, provide detailed error information
- **Decision**: Forward backend application errors (4xx/5xx) unchanged, generate standard proxy errors for infrastructure failures
- **Rationale**: Application errors from backends contain information clients need for proper error handling. Infrastructure errors (network failures, timeouts) should be normalized to standard proxy error codes to avoid exposing backend topology
- **Consequences**: Preserves application error semantics while protecting infrastructure details, but may require careful classification of error types

Connection pooling architecture decisions directly impact both performance and resource usage patterns. The choice of pooling strategy affects how efficiently the proxy can handle concurrent requests and how gracefully it responds to backend server changes.

Request context propagation decisions determine how cancellation, timeouts, and trace information flow through the proxy. Proper context handling ensures that client disconnections cancel backend requests and that distributed tracing works correctly across the proxy boundary.

Logging and observability integration decisions establish what information the proxy records about requests and how that information integrates with monitoring systems. These decisions impact debugging capabilities and operational visibility.

Common Pitfalls

The HTTP reverse proxy implementation involves several areas where developers commonly encounter issues that can lead to incorrect behavior, performance problems, or security vulnerabilities. Understanding these pitfalls helps avoid frustrating debugging sessions and ensures robust proxy operation.

⚠️ Pitfall: Not Copying All Request Headers

Many developers forget to copy important headers when forwarding requests to backend servers, leading to broken application functionality. Common missed headers include `Authorization` (breaking authentication), `Content-Type` (causing backend parsing errors), and `Accept` (preventing content negotiation).

The problem occurs because developers focus on the obvious headers like `Host` and `User-Agent` while overlooking headers that specific applications depend on. This is particularly problematic with RESTful APIs that rely on `Accept` headers for response format selection or applications using custom authentication headers.

Fix: Use Go's header cloning functionality and implement explicit header forwarding lists. Copy all headers by default, then selectively remove hop-by-hop headers rather than selectively adding headers:

```go
// Wrong approach - easy to miss important headers
backendReq.Header.Set("Authorization", req.Header.Get("Authorization"))
backendReq.Header.Set("Content-Type", req.Header.Get("Content-Type"))
// ... what about Accept, Accept-Language, X-Custom-Auth, etc?

// Better approach - copy all, then remove hop-by-hop
for name, values := range req.Header {
    if !isHopByHopHeader(name) {
        backendReq.Header[name] = values
    }
}
```

⚠️ Pitfall: Buffering Large Request Bodies

Developers often load entire request bodies into memory before forwarding them, causing memory exhaustion when clients upload large files or send streaming data. This happens because the obvious approach is to read the body, process it, then write it to the backend.

The issue manifests as the proxy consuming gigabytes of memory during large uploads, potentially causing out-of-memory crashes or degraded performance for other requests. It also increases latency because the proxy waits to receive the entire body before starting backend transmission.

Fix: Use streaming body forwarding with `io.Copy` or similar mechanisms that transfer data without intermediate buffering:

```go
// Wrong - buffers entire body in memory
bodyBytes, err := io.ReadAll(req.Body)
backendReq.Body = io.NopCloser(bytes.NewReader(bodyBytes))

// Right - streams body directly
backendReq.Body = req.Body
backendReq.ContentLength = req.ContentLength
```

⚠️ Pitfall: Not Handling Connection Errors Properly

Incorrect error handling when backend connections fail leads to confusing client error responses and poor debugging experience. Common mistakes include returning 500 Internal Server Error for all backend connection failures or not distinguishing between different types of network errors.

Backend connection failures should return 502 Bad Gateway to indicate that the proxy successfully received the request but couldn't reach the backend server. This helps clients and monitoring systems distinguish between proxy problems (5xx) and backend problems (502).

Fix: Classify connection errors correctly and return appropriate HTTP status codes:

```go
resp, err := client.Do(backendReq)
if err != nil {
    if isConnectionError(err) {
        http.Error(w, "Backend server unavailable", http.StatusBadGateway)
    } else if isTimeoutError(err) {
        http.Error(w, "Backend server timeout", http.StatusGatewayTimeout)
    } else {
        http.Error(w, "Proxy error", http.StatusInternalServerError)
    }
    return
}
```

⚠️ Pitfall: Leaking Connections on Errors

Failing to properly close connections when errors occur leads to connection leaks that eventually exhaust available file descriptors or connection pool resources. This happens when developers focus on the happy path but don't ensure cleanup in error cases.

Connection leaks are particularly problematic because they accumulate slowly and may not manifest until the proxy has been running under load for extended periods. The symptoms include increasing memory usage, declining performance, and eventually connection establishment failures.

Fix: Use defer statements and proper error handling to ensure connections are always cleaned up:

```go
resp, err := client.Do(backendReq)
if err != nil {
    // Handle error and return early
    return
}
defer resp.Body.Close() // Ensures connection cleanup

// Process response...
```

⚠️ Pitfall: Not Setting Appropriate Timeouts

Missing or incorrectly configured timeouts cause the proxy to hang indefinitely when backend servers become unresponsive, leading to resource exhaustion and poor user experience. Without timeouts, a single slow backend can cause request queuing and cascade failures.

The default timeout behavior in many HTTP clients is to wait indefinitely, which is inappropriate for a proxy that needs to provide reliable service even when backends are slow or unresponsive.

Fix: Configure comprehensive timeout settings on the HTTP transport:

```go
transport := &http.Transport{
    DialTimeout:           5 * time.Second,  // Connection establishment
    TLSHandshakeTimeout:   5 * time.Second,  // TLS handshake
    ResponseHeaderTimeout: 10 * time.Second, // Response headers
    IdleConnTimeout:       30 * time.Second, // Idle connection reuse
}
client := &http.Client{
    Transport: transport,
    Timeout:   30 * time.Second, // Total request timeout
}
```

⚠️ Pitfall: Modifying X-Forwarded Headers Incorrectly

Incorrectly handling `X-Forwarded-For`, `X-Forwarded-Proto`, and `X-Forwarded-Host` headers breaks client IP detection, protocol detection, and virtual host routing in backend applications. Common mistakes include overwriting existing values instead of appending, or using incorrect values.

This is especially problematic in proxy chains where multiple proxies add forwarding headers. Each proxy should append to existing `X-Forwarded-For` values rather than replacing them, preserving the complete chain of client and intermediate proxy IP addresses.

Fix: Properly append to existing forwarding headers:

```go
// Get client IP from connection
clientIP, _, _ := net.SplitHostPort(req.RemoteAddr)

// Append to existing X-Forwarded-For
existing := req.Header.Get("X-Forwarded-For")
if existing != "" {
    backendReq.Header.Set("X-Forwarded-For", existing+", "+clientIP)
} else {
    backendReq.Header.Set("X-Forwarded-For", clientIP)
}

// Set protocol and host appropriately
backendReq.Header.Set("X-Forwarded-Proto", "http") // or "https"
backendReq.Header.Set("X-Forwarded-Host", req.Host)
```

Implementation Guidance

This section provides concrete implementation guidance for building the HTTP reverse proxy component, focusing on the foundational reverse proxy functionality that forms the core of the load balancer system.

Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Server	Go net/http with http.ServeMux	Go net/http with gorilla/mux or chi router
HTTP Client	Go net/http.Client with custom transport	Custom HTTP client with connection pooling
Configuration	JSON files with encoding/json	YAML with gopkg.in/yaml.v3
Logging	Go log package	Structured logging with logrus or zap
Testing	Go testing package	testify for assertions and test suites

Recommended File Structure:

```
load-balancer/
├── cmd/
│   └── load-balancer/
│       └── main.go                ← Entry point and server setup
├── internal/
│   ├── proxy/
│   │   ├── proxy.go               ← ReverseProxy struct and core logic
│   │   ├── request.go             ← Request forwarding and header processing
│   │   ├── response.go            ← Response processing and streaming
│   │   ├── pool.go                ← Connection pool management
│   │   └── proxy_test.go          ← Proxy component tests
│   ├── config/
│   │   ├── config.go              ← Configuration structures and loading
│   │   └── config_test.go         ← Configuration validation tests
│   └── backend/
│       ├── backend.go             ← Backend representation (for next milestone)
│       └── manager.go             ← Backend pool management (for next milestone)
├── configs/
│   └── example.json               ← Example configuration file
└── go.mod                         ← Go module definition
```

Infrastructure Starter Code:

The following complete configuration management code handles loading and validating proxy settings:

```go
// internal/config/config.go

package config

import (
    "encoding/json"
    "fmt"
    "os"
    "time"
)

type Config struct {
    Port         int                `json:"port"`
    Backends     []BackendConfig    `json:"backends"`
    Algorithm    string             `json:"algorithm"`
    HealthCheck  HealthCheckConfig  `json:"health_check"`
    Proxy        ProxyConfig        `json:"proxy"`
    Logging      LoggingConfig      `json:"logging"`
}

type BackendConfig struct {
    URL             string        `json:"url"`
    Weight          int           `json:"weight"`
    MaxConnections  int           `json:"max_connections"`
    ConnectTimeout  time.Duration `json:"connect_timeout"`
    ResponseTimeout time.Duration `json:"response_timeout"`
    Enabled         bool          `json:"enabled"`
}

type ProxyConfig struct {
    ReadTimeout     time.Duration `json:"read_timeout"`
    WriteTimeout    time.Duration `json:"write_timeout"`
    IdleTimeout     time.Duration `json:"idle_timeout"`
    MaxHeaderBytes  int           `json:"max_header_bytes"`
}

type HealthCheckConfig struct {
    Enabled            bool          `json:"enabled"`
    Interval           time.Duration `json:"interval"`
    Timeout            time.Duration `json:"timeout"`
    HealthyThreshold   int           `json:"healthy_threshold"`
    UnhealthyThreshold int           `json:"unhealthy_threshold"`
    Path               string        `json:"path"`
    ExpectedStatus     int           `json:"expected_status"`
```

```
    UserAgent          string        `json:"user_agent"`
}

type LoggingConfig struct {
    Level  string `json:"level"`
    Format string `json:"format"`
    Output string `json:"output"`
}

func LoadConfig(filename string) (*Config, error) {
    data, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

    var config Config
    if err := json.Unmarshal(data, &config); err != nil {
        return nil, fmt.Errorf("failed to parse config: %w", err)
    }

    if err := config.Validate(); err != nil {
        return nil, fmt.Errorf("invalid configuration: %w", err)
    }

    return &config, nil
}

func (c *Config) Validate() error {
    if c.Port <= 0 || c.Port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535")
    }

    if len(c.Backends) == 0 {
        return fmt.Errorf("at least one backend must be configured")
    }

    for i, backend := range c.Backends {
        if backend.URL == "" {
            return fmt.Errorf("backend %d: URL is required", i)
        }
        if backend.Weight <= 0 {
            return fmt.Errorf("backend %d: weight must be positive", i)
        }
    }
```

```
    return nil
}
```

Core Logic Skeleton Code:

The following skeleton provides the structure for implementing the main reverse proxy functionality:

GO

```go
// internal/proxy/proxy.go

package proxy

import (
    "context"
    "fmt"
    "io"
    "log"
    "net/http"
    "net/url"
    "strings"
    "sync/atomic"
    "time"
)

// ReverseProxy handles HTTP request forwarding to backend servers
type ReverseProxy struct {
    backends    []string
    counter     int64
    client      *http.Client
    logger      *log.Logger
}

// NewReverseProxy creates a new reverse proxy with the given backend servers
func NewReverseProxy(backends []string, timeout time.Duration) *ReverseProxy {
    transport := &http.Transport{
        MaxIdleConns:        100,
        MaxIdleConnsPerHost: 10,
        IdleConnTimeout:     30 * time.Second,
        DisableKeepAlives:   false,
    }

    return &ReverseProxy{
        backends: backends,
        client: &http.Client{
            Transport: transport,
            Timeout:   timeout,
        },
        logger: log.New(os.Stdout, "[PROXY] ", log.LstdFlags),
    }
}
```

```go
// ServeHTTP handles incoming HTTP requests and forwards them to backends
func (p *ReverseProxy) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Select a backend server using round-robin algorithm
    //         - Use atomic.AddInt64 to increment counter safely
    //         - Use modulo to wrap around to first backend
    //         - Handle case where no backends are available

    // TODO 2: Parse the selected backend URL
    //         - Use url.Parse to convert backend string to *url.URL
    //         - Handle parsing errors by returning 502 Bad Gateway
    //         - Log any URL parsing failures for debugging

    // TODO 3: Create new request for backend server
    //         - Use http.NewRequestWithContext to preserve cancellation
    //         - Set the target URL to point to selected backend
    //         - Copy the original request method and body

    // TODO 4: Copy and modify request headers
    //         - Copy all headers from original request
    //         - Add X-Forwarded-For with client IP address
    //         - Add X-Forwarded-Proto with original protocol
    //         - Set Host header to backend server host

    // TODO 5: Forward request to backend server
    //         - Use p.client.Do() to send the request
    //         - Handle connection errors by returning 502 Bad Gateway
    //         - Handle timeout errors by returning 504 Gateway Timeout
    //         - Log all request forwarding attempts with timing

    // TODO 6: Copy response status and headers to client
    //         - Set response status code from backend response
    //         - Copy all response headers except hop-by-hop headers
    //         - Handle cases where backend returns invalid responses

    // TODO 7: Stream response body to client
    //         - Use io.Copy to stream body without buffering
    //         - Handle streaming errors gracefully
    //         - Ensure response body is always closed
    //         - Log response completion with status and bytes transferred
}
```

```go
// selectBackend implements round-robin backend selection
func (p *ReverseProxy) selectBackend() string {
    // TODO: Implement thread-safe round-robin selection
    //       - Use atomic operations to avoid race conditions
    //       - Handle empty backend list gracefully
    //       - Return empty string if no backends available
    return ""
}

// copyHeaders copies HTTP headers from source to destination
func copyHeaders(dst, src http.Header) {
    // TODO: Implement header copying logic
    //       - Copy all headers except hop-by-hop headers
    //       - Handle multi-value headers correctly
    //       - Skip Connection, Upgrade, Proxy-* headers
}

// addForwardingHeaders adds proxy-specific headers to the request
func addForwardingHeaders(req *http.Request, originalReq *http.Request) {
    // TODO: Add X-Forwarded-* headers
    //       - Extract client IP from RemoteAddr
    //       - Set X-Forwarded-For (append if already exists)
    //       - Set X-Forwarded-Proto based on original request
    //       - Set X-Forwarded-Host to original Host header
}

// isConnectionError determines if an error is a connection-related error
func isConnectionError(err error) bool {
    // TODO: Implement error type classification
    //       - Check for connection refused errors
    //       - Check for DNS resolution failures
    //       - Check for network unreachable errors
    //       - Return true for infrastructure errors, false for others
    return false
}
```

Language-Specific Hints:

- Use `http.NewRequestWithContext()` instead of `http.NewRequest()` to ensure proper cancellation propagation when clients disconnect
- The `RemoteAddr` field contains "IP:port" format, use `net.SplitHostPort()` to extract just the IP address for X-Forwarded-For
- Use `atomic.AddInt64()` for thread-safe counter increments in the round-robin algorithm
- Set `Transport.DisableKeepAlives = false` to enable connection reuse for better performance
- Use `defer response.Body.Close()` immediately after checking for request errors to ensure cleanup
- The `io.Copy()` function efficiently streams data without intermediate buffering

Milestone Checkpoint:

After implementing this component, verify functionality with these steps:

1. **Start the proxy server**: `go run cmd/load-balancer/main.go -config configs/example.json`
2. **Start test backend servers** on ports 8081, 8082: `python3 -m http.server 8081 &`
3. **Send test requests**: `curl -v http://localhost:8080/test`
4. **Expected behavior**:
   - Requests alternate between backend servers (round-robin)
   - Response headers include content from backend servers
   - X-Forwarded-For header added with client IP
   - Connection errors return 502 Bad Gateway status

Debugging Checklist:

Symptom	Likely Cause	Diagnostic Steps	Solution
All requests return 502	Backend servers not running	Check backend server status, verify URLs in config	Start backend servers, fix configuration
Requests hang indefinitely	Missing timeouts	Check HTTP client configuration	Add timeout settings to transport
Memory usage grows with large uploads	Body buffering	Profile memory usage during uploads	Implement streaming body forwarding
Some headers missing in backend	Incomplete header copying	Compare request headers vs forwarded headers	Fix header copying logic
Round-robin not working	Race condition in counter	Test with concurrent requests	Use atomic operations for counter

Backend Pool Manager

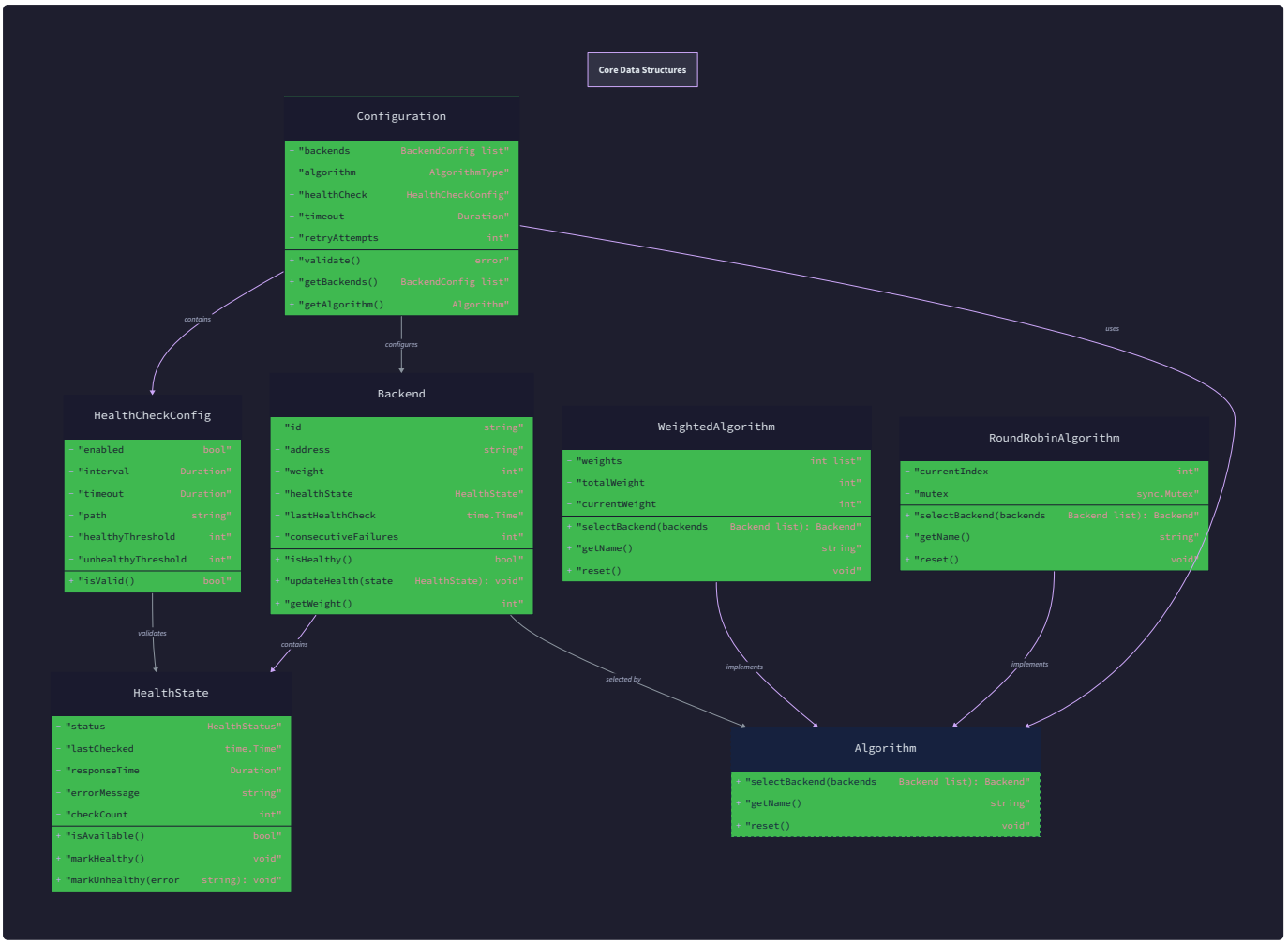
> **Milestone(s):** Milestone 2 (Round Robin Distribution), Milestone 3 (Health Checks), Milestone 4 (Additional Algorithms) — this component manages the collection of backend servers and provides the foundation for all load balancing algorithms and health monitoring

Mental Model: Team Roster

Think of the backend pool manager as a sports team's roster management system. Just as a coach maintains a roster of players with their positions, capabilities, and current availability status, the backend pool manager maintains a collection of backend servers with their URLs, weights, and health states.

When it's time to put a player in the game, the coach considers who's healthy, who's performing well, and what position needs to be filled. Similarly, when a request arrives, the backend pool manager considers which servers are healthy, which algorithm is active, and selects the most appropriate backend server. The coach tracks player statistics and injury status just as the pool manager tracks connection counts and health check results.

The roster can be updated mid-season as players are traded, injured, or recovered, and the backend pool can be updated during operation as servers are added, removed, or change health status. The key insight is that the roster manager provides a clean abstraction between the game strategy (load balancing algorithm) and the individual player management (backend server lifecycle).



The `BackendManager` serves as the central coordinator that bridges three critical concerns: maintaining the pool of available backend servers, tracking their health and connection states, and providing a clean interface for load balancing algorithms to select appropriate targets. This separation of concerns allows algorithms to focus purely on selection logic while the manager handles the operational complexities of backend lifecycle management.

Backend Registration

Backend registration encompasses the entire lifecycle of adding, updating, and removing backend servers from the active pool. The registration process must handle both initial configuration loading and runtime updates while maintaining service availability and ensuring data consistency across concurrent operations.

The registration process begins with parsing and validating backend configurations. Each backend requires a valid URL that the load balancer can reach, along with optional parameters like weight for proportional distribution and connection limits for capacity management. The validation phase checks URL format, ensures uniqueness within the pool, and verifies that network connectivity is possible before adding the backend to the active rotation.

Registration Operation	Input Parameters	Validation Checks	Side Effects
`AddBackend`	`BackendConfig`	URL format, uniqueness, connectivity	Creates `Backend` struct, initializes health state, adds to pool
`RemoveBackend`	Backend ID string	Backend exists, handle active connections	Updates pool, waits for connection drain
`UpdateBackend`	Backend ID, new config	Config validity, backward compatibility	Updates in-place, preserves health history
`LoadBackends`	Configuration file/API	Bulk validation, dependency checks	Replaces entire pool atomically

The `NewBackend` function transforms a `BackendConfig` into a fully initialized `Backend` struct with proper defaults and initial state. The backend receives a unique ID (typically a hash of the URL), atomic counters for connection tracking initialized to zero, and a fresh `HealthState` that assumes the backend is healthy until proven otherwise. The creation timestamp and metadata fields support monitoring and debugging scenarios.

Backend removal requires careful coordination with active connections and ongoing health checks. The removal process first marks the backend as unavailable for new requests, then waits for existing connections to complete or timeout. Health checkers must be notified to stop monitoring the removed backend, and any algorithm-specific state (like weighted round-robin counters) needs cleanup to prevent memory leaks or stale references.

Runtime updates to backend configuration present unique challenges because they must preserve operational history while applying new parameters. Weight changes for weighted algorithms require recalculating distribution ratios, while URL changes effectively create a new backend with fresh health state. The update

process uses copy-on-write semantics to ensure atomic transitions that don't disrupt ongoing requests.

> **Design Insight:** Backend registration uses optimistic concurrency control where operations assume success and handle conflicts during commit. This approach minimizes lock contention during high-traffic periods while ensuring consistency through atomic swap operations on the backend pool reference.

Architecture Decision: Backend Identification Strategy

> **Decision: Use URL-based hash IDs for backend identification**
>
> - **Context**: Need unique, stable identifiers for backends that survive configuration reloads and support efficient lookups
> - **Options Considered**:
>   - Sequential integers (simple but not stable across restarts)
>   - User-provided names (flexible but requires uniqueness validation)
>   - URL-based hashes (stable and deterministic)
> - **Decision**: Generate backend IDs using SHA-256 hash of the normalized URL
> - **Rationale**: URL hashes provide stable identifiers that survive restarts, enable idempotent configuration reloads, and eliminate the need for explicit ID management. Hash collisions are extremely unlikely with SHA-256.
> - **Consequences**: Backend IDs are deterministic and stable, but not human-readable. Debug scenarios require mapping IDs back to URLs, which the manager provides through lookup methods.

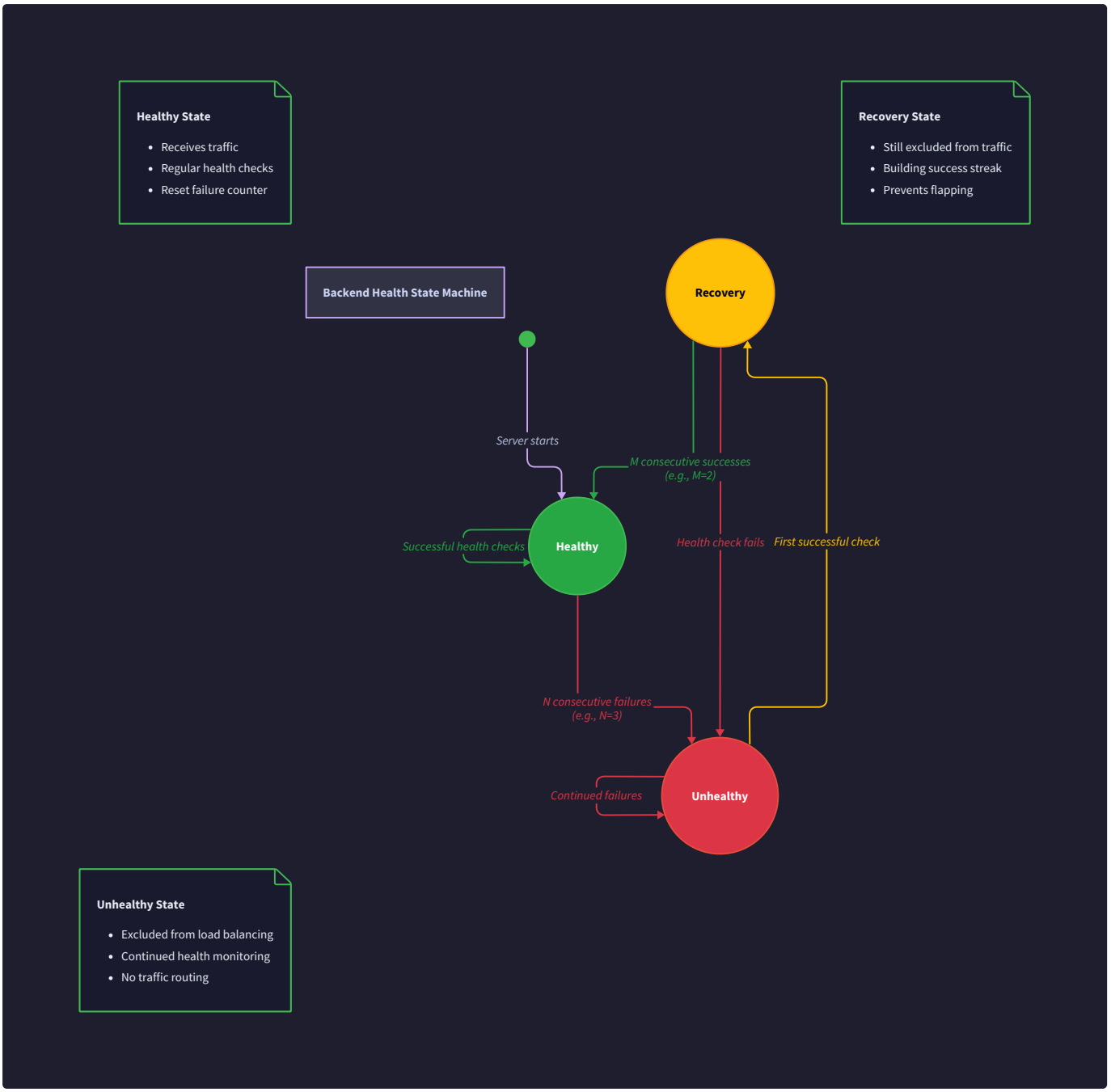
Option	Pros	Cons
Sequential integers	Simple, compact, human-readable	Not stable across restarts, requires state persistence
User-provided names	Human-readable, meaningful	Requires uniqueness validation, potential conflicts
URL-based hashes	Stable, deterministic, no coordination needed	Not human-readable, potential for collisions

Health State Tracking

Health state tracking maintains a comprehensive view of each backend's operational status through both active health checks and passive monitoring of request outcomes. The health state system implements a finite state machine with hysteresis to prevent rapid oscillations between healthy and unhealthy states that could destabilize request distribution.

The `HealthState` structure captures both current status and historical trends to support intelligent failover decisions. The consecutive success and failure counters implement the threshold-based state transitions that prevent single transient errors from removing healthy backends from rotation. The comprehensive timestamp tracking enables monitoring systems to understand health patterns and identify backends experiencing intermittent issues.

Health State Field	Type	Purpose	Update Trigger
`Healthy`	bool	Current availability status	State transition after threshold breach
`ConsecutiveSuccesses`	int	Consecutive successful checks	Each successful health check
`ConsecutiveFailures`	int	Consecutive failed checks	Each failed health check
`LastCheckTime`	time.Time	Most recent health check attempt	Every health check start
`LastSuccessTime`	time.Time	Most recent successful check	Successful health check completion
`LastFailureTime`	time.Time	Most recent failed check	Failed health check completion
`LastError`	string	Error message from last failure	Health check error occurrence
`TotalChecks`	int64	Lifetime health check count	Every health check attempt
`TotalSuccesses`	int64	Lifetime successful check count	Each successful health check
`TotalFailures`	int64	Lifetime failed check count	Each failed health check
`StateChanges`	[]HealthStateChange	History of state transitions	Healthy/unhealthy transitions



The state transition logic implements configurable thresholds to balance responsiveness with stability. When a healthy backend fails a health check, the consecutive failure counter increments while the consecutive success counter resets to zero. Only after reaching the `UnhealthyThreshold` does the backend transition to unhealthy state. The reverse process applies for recovery, requiring `HealthyThreshold` consecutive successes to restore the backend to healthy state.

Health state updates must be thread-safe because multiple goroutines may simultaneously process health check results, update connection counts, or read health status for request routing decisions. The `UpdateHealthState` method uses atomic operations for scalar fields and mutex protection for complex updates that modify multiple related fields.

The `RecordSuccess` and `RecordFailure` methods encapsulate the state transition logic and return boolean values indicating whether a state change occurred. This return value triggers notifications to other system components like metrics collectors or alerting systems that need to respond to backend availability changes.

Architecture Decision: Health State Persistence

Decision: Maintain health state only in memory with graceful startup behavior

- **Context**: Need to balance fast health state updates with persistence requirements and system complexity
- **Options Considered**:
  - In-memory only (fast but loses history on restart)
  - Database persistence (durable but adds latency)
  - File-based snapshots (balance of performance and persistence)
- **Decision**: Keep health state in memory with intelligent startup behavior that assumes all backends are healthy initially
- **Rationale**: Health checks provide rapid feedback on actual backend status, making historical state less critical. In-memory storage eliminates persistence-related latency and complexity. Assuming healthy at startup is safer than assuming unhealthy (which could cause unnecessary downtime).
- **Consequences**: Fast health state operations with no I/O overhead, but system restart temporarily loses health history. Health checks quickly rebuild accurate state within one check interval.

Option	Pros	Cons
In-memory only	Fast updates, no I/O overhead, simple	Loses history on restart
Database persistence	Durable, queryable history	Adds latency, complexity, dependencies
File-based snapshots	Balanced approach, good for debugging	Still adds I/O overhead, snapshot timing issues

Backend Selection Interface

The backend selection interface provides a clean abstraction that decouples load balancing algorithms from backend pool management. This abstraction allows algorithms to focus purely on selection logic while the manager handles filtering unhealthy backends, updating connection counts, and maintaining algorithm-agnostic operational concerns.

The core `Algorithm` interface defines the contract that all load balancing implementations must satisfy. The interface receives a slice of healthy backends and returns the selected backend, or nil if no suitable backend is available. This simple interface supports both stateless algorithms (like random selection) and stateful algorithms (like round-robin with counters) through the algorithm implementation's internal state management.

Interface Method	Parameters	Returns	Behavior Contract
`SelectBackend`	`[]*Backend` (healthy backends only)	`*Backend` or nil	Must handle empty slice, return nil for no selection
`Name`	none	string	Return human-readable algorithm name for logging/metrics

The `BackendManager` orchestrates the selection process by first filtering the backend pool to include only healthy backends, then delegating to the active algorithm for the actual selection decision. This two-stage process ensures that algorithms never receive unhealthy backends and don't need to duplicate health checking logic across every algorithm implementation.

The selection workflow follows a consistent pattern across all algorithm types:

1. The manager receives a request for backend selection from the HTTP proxy component
2. It queries the current backend pool and filters out any backends marked as unhealthy
3. If no healthy backends remain, it returns an error indicating service unavailability
4. The filtered healthy backend list is passed to the active algorithm's `SelectBackend` method
5. The algorithm applies its selection logic and returns the chosen backend
6. The manager increments the selected backend's active connection counter atomically
7. The selected backend is returned to the caller for request forwarding

The interface design supports runtime algorithm switching by allowing the manager to swap algorithm implementations without disrupting ongoing requests. Algorithm instances maintain their own internal state (like round-robin counters or least-connection tracking) independently of the backend pool, enabling clean separation of concerns.

Architecture Decision: Algorithm State Management

Decision: Algorithms maintain internal state with backend pool change notifications

- **Context**: Load balancing algorithms need to track state (counters, connection counts) while the backend pool changes dynamically
- **Options Considered**:
  - Stateless algorithms only (simple but limits algorithm types)
  - Manager-maintained algorithm state (centralized but tightly coupled)
  - Algorithm-internal state with notifications (flexible and decoupled)
- **Decision**: Algorithms maintain their own state and receive notifications when the backend pool changes
- **Rationale**: Different algorithms need different state (round-robin needs counters, least-connections needs connection tracking). Internal state keeps algorithms self-contained while notifications enable adaptation to pool changes.
- **Consequences**: Algorithms are more complex but also more flexible. Backend pool changes require notification broadcast, but this enables sophisticated algorithm behavior like weighted round-robin with smooth transitions.

Option	Pros	Cons
Stateless algorithms only	Simple, no synchronization needed	Limits available algorithms, worse distribution quality
Manager-maintained algorithm state	Centralized state management	Tight coupling, manager complexity grows with algorithms
Algorithm-internal state	Flexible, self-contained algorithms	More complex algorithms, need change notifications

The notification mechanism uses the observer pattern where algorithms can register for backend pool change events. When backends are added, removed, or change health status, the manager broadcasts notifications that allow algorithms to update their internal state accordingly. This approach maintains loose coupling while enabling stateful algorithm behavior.

Architecture Decisions

Several critical architecture decisions shape the backend pool manager's design and directly impact performance, maintainability, and operational characteristics. These decisions balance competing concerns like thread safety, performance, and operational flexibility.

Architecture Decision: Thread Safety Strategy

Decision: Use fine-grained locking with atomic operations for hot paths

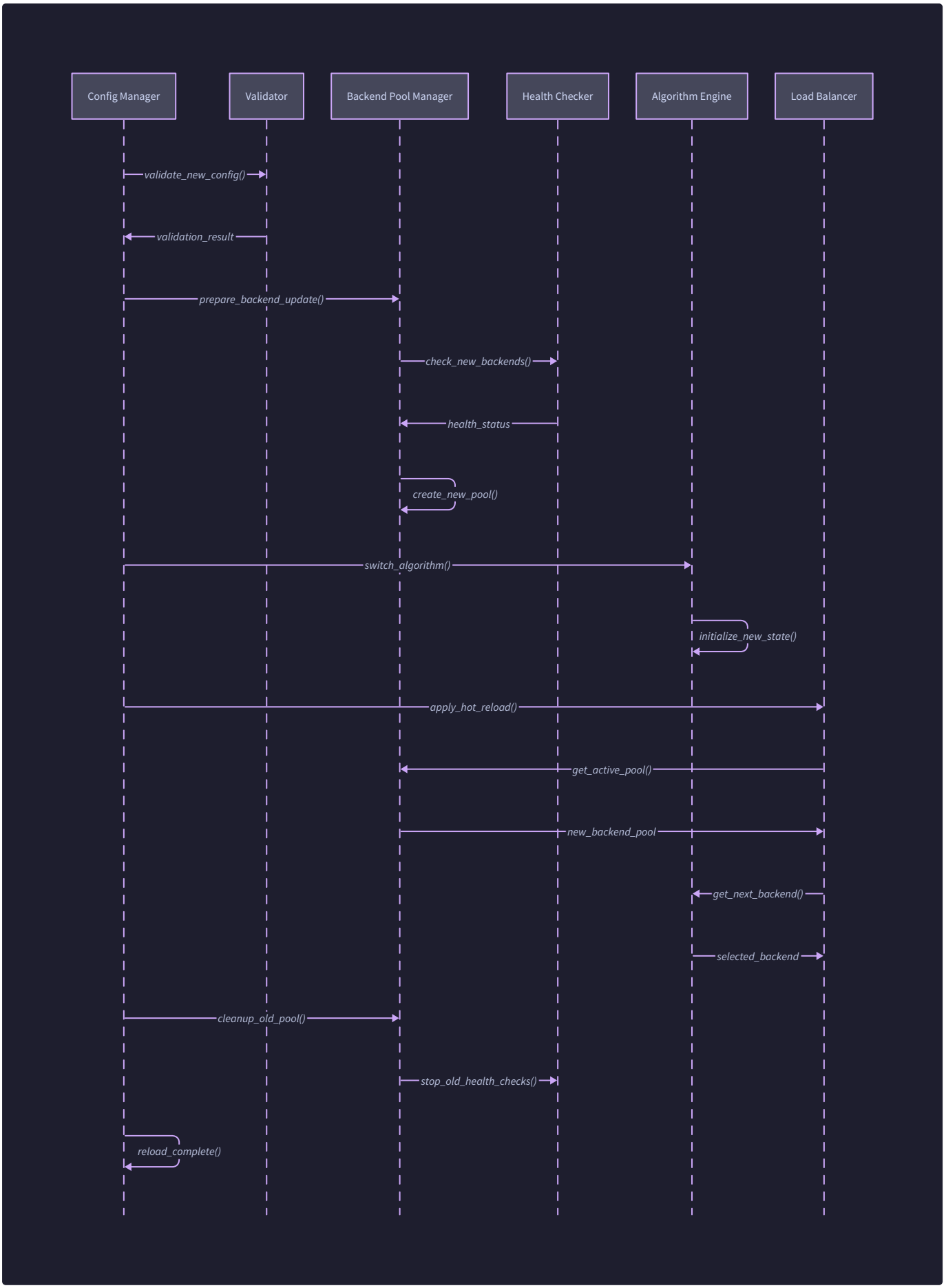
- **Context**: Backend manager faces concurrent access from multiple goroutines handling requests, health checks updating status, and configuration changes
- **Options Considered**:
  - Single global mutex (simple but serializes all access)
  - Read-write mutex for pool access (better than global mutex)
  - Fine-grained locking with atomics (complex but highest performance)
- **Decision**: Use atomic operations for connection counting, RWMutex for backend pool access, and separate mutexes for individual backend health state updates
- **Rationale**: Hot path operations (backend selection, connection counting) need minimal latency. Read-heavy workloads benefit from RWMutex allowing concurrent reads. Health state updates are infrequent and can use individual backend locks to avoid blocking pool access.
- **Consequences**: Higher performance under concurrent load but increased implementation complexity. Need careful lock ordering to prevent deadlocks during operations that affect multiple backends.

Option	Pros	Cons
Single global mutex	Simple, no deadlock risk	Serializes all access, poor performance
Read-write mutex for pool	Better read performance	Still serializes writes, health updates block reads
Fine-grained locking	Highest performance, targeted synchronization	Complex, potential deadlock scenarios

Architecture Decision: Configuration Update Strategy

Decision: Implement atomic backend pool replacement with graceful connection draining

- **Context**: Configuration updates must not drop existing connections while ensuring new requests use updated backend pool
- **Options Considered**:
  - Stop-and-restart (simple but causes downtime)
  - Incremental updates (complex coordination)
  - Atomic pool replacement (clean but requires connection tracking)
- **Decision**: Replace the entire backend pool atomically while tracking connections to removed backends
- **Rationale**: Atomic replacement eliminates complex state reconciliation and ensures consistent view of backend pool. Connection tracking enables graceful shutdown of removed backends without dropping active requests.
- **Consequences**: Configuration updates are clean and atomic, but require sophisticated connection tracking. Memory usage temporarily increases during transitions as both old and new backend pools exist simultaneously.


Config Manager
Validator
Backend Pool Manager
Health Checker
Algorithm Engine
Load Balancer
validate_new_config()
validation_result
prepare_backend_update()
check_new_backends()
health_status
create_new_pool()
switch_algorithm()
initialize_new_state()
apply_hot_reload()
get_active_pool()
new_backend_pool
get_next_backend()
selected_backend
cleanup_old_pool()
stop_old_health_checks()
reload_complete()

Option	Pros	Cons
Stop-and-restart	Simple, guaranteed consistency	Service downtime, lost connections
Incremental updates	No service interruption	Complex state reconciliation, potential inconsistencies
Atomic pool replacement	Clean semantics, no inconsistent state	Requires connection tracking, temporary memory overhead

Architecture Decision: Backend Pool Storage Structure

> **Decision: Use concurrent-safe map with slice snapshot for algorithm selection**
>
> - **Context**: Need efficient backend lookup by ID for health updates while providing algorithm-friendly iteration interface
> - **Options Considered**:
>   - Slice only (simple iteration but O(n) lookups)
>   - Map only (fast lookups but poor algorithm interface)
>   - Map with slice snapshots (best of both but requires synchronization)
> - **Decision**: Maintain backends in a concurrent map with periodic slice snapshots for algorithm selection
> - **Rationale**: Map enables O(1) backend lookup for health updates and configuration changes. Slice snapshots provide algorithms with stable, consistent views of the backend pool without holding locks during selection logic.
> - **Consequences**: Fast lookups and algorithm-friendly interface, but requires careful snapshot management and slightly increased memory usage for snapshot storage.

Option	Pros	Cons
Slice only	Simple, good for algorithms	O(n) lookups for health updates
Map only	Fast lookups, efficient updates	Poor interface for algorithm iteration
Map with slice snapshots	Fast lookups and good algorithm interface	More complex, requires snapshot synchronization

Common Pitfalls

Backend pool management involves several subtle concurrency and state management challenges that frequently cause issues for developers implementing load balancers. Understanding these pitfalls and their solutions prevents common bugs that can cause request failures or system instability.

⚠️ Pitfall: Race Conditions in Connection Counting

The most frequent error occurs when multiple goroutines simultaneously modify backend connection counters without proper synchronization. This typically happens when request handlers increment the connection count when selecting a backend, then later decrement it in a deferred function, but use non-atomic operations for the counter updates.

The symptom appears as connection counts that become negative or grow without bounds, leading to incorrect least-connections algorithm behavior. Backends may show impossibly high connection counts that never decrease, or negative counts that confuse the selection logic.

The fix requires using atomic operations for all connection counter modifications. The `IncrementConnections` and `DecrementConnections` methods must use `atomic.AddInt64` to ensure thread-safe updates. Additionally, ensure that every increment has a corresponding decrement, typically using deferred function calls that execute even when request processing encounters errors.

⚠️ Pitfall: Inconsistent Backend Pool Views During Updates

Configuration updates can create scenarios where different parts of the system see different versions of the backend pool simultaneously. This happens when the pool update process modifies the backend map incrementally while other goroutines are reading from it, leading to partially updated views.

Symptoms include algorithms selecting backends that no longer exist, health checkers monitoring removed backends, or new backends not appearing in selection rotation despite successful configuration updates. Error logs may show "backend not found" errors during request forwarding.

The solution uses atomic pointer swaps for the entire backend pool. Store the backend map in an atomic pointer value, create a complete new map during updates, then atomically swap the pointer. This ensures all readers see either the complete old state or the complete new state, never a mixed view.

⚠️ Pitfall: Health State Update Ordering

Health state updates from multiple concurrent health checks can arrive out of order, causing newer health check results to be overwritten by older results. This occurs when network delays cause health check responses to arrive in different order than they were sent, particularly under high load or network congestion.

The manifestation includes backends that appear to randomly flip between healthy and unhealthy states, health statistics that don't match actual check timing, and backends that remain unhealthy despite successful checks. The `LastCheckTime` field may show timestamps that move backward.

Prevention requires adding sequence numbers or timestamps to health check operations and ignoring updates that are older than the current health state timestamp. The `RecordSuccess` and `RecordFailure` methods should verify that the update timestamp is newer than `LastCheckTime` before applying changes.

⚠️ Pitfall: Empty Backend Pool Handling

Algorithms often fail to handle the case where no healthy backends are available, either returning nil pointers that cause panics or attempting to perform modulo operations with zero divisors. This scenario occurs during startup, configuration updates, or when all backends fail simultaneously.

Symptoms include application crashes with division by zero errors, null pointer dereferences during request forwarding, or infinite loops in selection algorithms that assume at least one backend exists.

The robust solution implements graceful degradation at multiple layers. The `GetHealthyBackends` method should document that it can return empty slices. Algorithm `SelectBackend` methods must check for empty input and return nil gracefully. The HTTP proxy component should handle nil backend selection by returning 503 Service Unavailable responses rather than crashing.

⚠️ Pitfall: Memory Leaks from Algorithm State

Stateful algorithms like weighted round-robin or least connections can accumulate state for backends that have been removed from the pool. This occurs when algorithms maintain internal maps or counters keyed by backend ID but don't clean up when backends are removed.

The memory leak grows over time as backends are added and removed during configuration updates, particularly in dynamic environments where backend pools change frequently. Memory usage increases monotonically, and algorithm state may reference deleted backends indefinitely.

The fix implements proper cleanup notifications where the backend manager sends removal events to all registered algorithms. Algorithms must implement cleanup logic that removes internal state for deleted backends. Using weak references or timeout-based cleanup can provide additional safety against leaked state.

⚠️ Pitfall: Health Check State Inconsistency During Algorithm Selection

Backend health state can change between the time an algorithm selects a backend and when the request is forwarded to it. This race condition occurs when health checks update backend state to unhealthy just after the algorithm selection but before request forwarding begins.

The result is requests sent to backends that are known to be unhealthy, causing unnecessary request failures and poor user experience. Error rates increase despite the health checking system functioning correctly.

The solution implements last-moment health validation in the HTTP proxy component immediately before forwarding requests. If the selected backend becomes unhealthy between selection and forwarding, the proxy can either retry with a new selection or return an appropriate error response. This adds a small latency cost but prevents forwarding to known-bad backends.

Implementation Guidance

The backend pool manager bridges abstract load balancing concepts with concrete implementation concerns like thread safety, state management, and configuration updates. This guidance provides working code foundations and specific implementation strategies for building a robust backend management system.

Technology Recommendations

Component	Simple Option	Advanced Option
Backend Storage	`sync.Map` for concurrent access	Custom concurrent data structure with RCU
Health State Updates	`sync.RWMutex` per backend	Lock-free atomic updates with CAS loops
Configuration Loading	Standard library `encoding/json`	`github.com/spf13/viper` for multiple formats
Algorithm Interface	Simple interface with sync.RWMutex	Plugin-based architecture with dynamic loading
Connection Counting	`sync/atomic` for int64 counters	Custom atomic counters with overflow protection

Recommended File Structure

```
internal/backend/
  manager.go                ← BackendManager implementation
  manager_test.go           ← comprehensive unit tests
  backend.go                ← Backend struct and methods
  health_state.go           ← HealthState management
  algorithm.go              ← Algorithm interface definition
  algorithms/
    round_robin.go          ← round robin implementation
    least_connections.go    ← least connections algorithm
    weighted.go             ← weighted round robin
    ip_hash.go              ← IP hash algorithm
    random.go               ← random selection
  config.go                 ← configuration structures
  config_test.go            ← configuration validation tests
```

Infrastructure Starter Code

Configuration Loading and Validation:

```go
package backend

import (
    "encoding/json"
    "fmt"
    "net/url"
    "os"
    "time"
)

// LoadConfig loads and validates configuration from file
func LoadConfig(filename string) (*Config, error) {
    data, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

    var config Config
    if err := json.Unmarshal(data, &config); err != nil {
        return nil, fmt.Errorf("failed to parse config JSON: %w", err)
    }

    if err := config.Validate(); err != nil {
        return nil, fmt.Errorf("config validation failed: %w", err)
    }

    return &config, nil
}

// Validate checks configuration for correctness and completeness
func (c *Config) Validate() error {
    if c.Port <= 0 || c.Port > 65535 {
        return fmt.Errorf("invalid port number: %d", c.Port)
    }

    if len(c.Backends) == 0 {
        return fmt.Errorf("at least one backend must be configured")
    }

    for i, backend := range c.Backends {
        if err := backend.Validate(); err != nil {
            return fmt.Errorf("backend %d validation failed: %w", i, err)
```

```
        }
    }

    validAlgorithms := map[string]bool{
        "round_robin": true, "least_connections": true,
        "weighted_round_robin": true, "ip_hash": true, "random": true,
    }
    if !validAlgorithms[c.Algorithm] {
        return fmt.Errorf("unknown algorithm: %s", c.Algorithm)
    }

    return nil
}

// Validate checks backend configuration
func (bc *BackendConfig) Validate() error {
    if bc.URL == "" {
        return fmt.Errorf("backend URL cannot be empty")
    }

    if _, err := url.Parse(bc.URL); err != nil {
        return fmt.Errorf("invalid backend URL: %w", err)
    }

    if bc.Weight <= 0 {
        bc.Weight = 1 // default weight
    }

    if bc.MaxConnections <= 0 {
        bc.MaxConnections = 100 // default limit
    }

    return nil
}
```

Backend Creation and Management:

```go
package backend


import (

    "crypto/sha256"

    "fmt"

    "net/url"

    "sync/atomic"

    "time"

)


// NewBackend creates a new backend from configuration

func NewBackend(config BackendConfig) (*Backend, error) {

    parsedURL, err := url.Parse(config.URL)

    if err != nil {

        return nil, fmt.Errorf("invalid backend URL: %w", err)

    }



    // Generate stable ID from URL

    hash := sha256.Sum256([]byte(config.URL))

    id := fmt.Sprintf("%x", hash[:8]) // use first 8 bytes of hash



    backend := &Backend{

        ID:               id,

        URL:              parsedURL,

        Weight:           config.Weight,

        ActiveConnections: new(int64), // atomic counter

        HealthState:      newHealthState(),

        CreatedAt:        time.Now(),

        LastUsed:         nil,

        Metadata:         make(map[string]interface{}),

    }



    return backend, nil

}


// newHealthState creates initial health state assuming backend is healthy

func newHealthState() *HealthState {

    return &HealthState{

        Healthy:              true, // assume healthy until proven otherwise

        ConsecutiveSuccesses:  0,

        ConsecutiveFailures:   0,

        LastCheckTime:        time.Time{},
```

```
        LastSuccessTime:      time.Time{},
        LastFailureTime:      time.Time{},
        LastError:            "",
        TotalChecks:          0,
        TotalSuccesses:       0,
        TotalFailures:        0,
        StateChanges:         make([]HealthStateChange, 0),
    }
}

// IncrementConnections atomically increments active connection count
func (b *Backend) IncrementConnections() int64 {
    return atomic.AddInt64(b.ActiveConnections, 1)
}

// DecrementConnections atomically decrements active connection count
func (b *Backend) DecrementConnections() int64 {
    newCount := atomic.AddInt64(b.ActiveConnections, -1)
    if newCount < 0 {
        // Safety: reset to zero if somehow went negative
        atomic.StoreInt64(b.ActiveConnections, 0)
        return 0
    }
    return newCount
}

// GetActiveConnections reads current connection count
func (b *Backend) GetActiveConnections() int64 {
    return atomic.LoadInt64(b.ActiveConnections)
}
```

Core Logic Skeleton Code

Backend Manager Implementation:

```go
package backend

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

type BackendManager struct {
    // TODO 1: Add mutex for protecting backend pool map
    // TODO 2: Add atomic pointer to current backend slice for algorithms
    // TODO 3: Add current algorithm instance
    // TODO 4: Add configuration reload channel
    // Hint: Use sync.RWMutex for pool map and atomic.Value for backend slice
}

// NewBackendManager creates a new backend pool manager
func NewBackendManager() *BackendManager {
    // TODO 1: Initialize the BackendManager struct with empty backend map
    // TODO 2: Set up atomic backend slice pointer
    // TODO 3: Create configuration reload channel
    // TODO 4: Start background goroutine for configuration updates
    // Hint: Use make(map[string]*Backend) and atomic.Value for slice
}

// AddBackend adds a new backend to the pool
func (bm *BackendManager) AddBackend(config BackendConfig) error {
    // TODO 1: Validate the backend configuration using config.Validate()
    // TODO 2: Create new Backend instance using NewBackend()
    // TODO 3: Acquire write lock on backend pool map
    // TODO 4: Check if backend ID already exists and return error if so
    // TODO 5: Add backend to pool map
    // TODO 6: Update atomic backend slice with new pool contents
    // TODO 7: Release write lock
    // Hint: Remember to update both the map and atomic slice atomically
}

// GetHealthyBackends returns slice of currently healthy backends
func (bm *BackendManager) GetHealthyBackends() []*Backend {
    // TODO 1: Load current backend slice from atomic pointer
    // TODO 2: Create new slice to hold only healthy backends
    // TODO 3: Iterate through all backends checking HealthState.Healthy
```

```
    // TODO 4: Add healthy backends to result slice
    // TODO 5: Return filtered slice (may be empty)
    // Hint: Use atomic.Value.Load() to get current backend slice
}

// UpdateBackendHealth updates health status for specified backend
func (bm *BackendManager) UpdateBackendHealth(backend *Backend, healthy bool) error {
    // TODO 1: Acquire read lock to find backend in pool
    // TODO 2: Verify backend still exists in pool
    // TODO 3: Call appropriate RecordSuccess or RecordFailure on backend
    // TODO 4: If health state changed, log the transition
    // TODO 5: Update atomic backend slice if needed
    // Hint: Check return value from Record* methods to detect state changes
}
```

Algorithm Interface Implementation:

```go
package backend

// Algorithm defines the interface for backend selection algorithms
type Algorithm interface {
    SelectBackend(backends []*Backend) *Backend
    Name() string
}

// SelectBackend chooses appropriate backend using active algorithm
func (bm *BackendManager) SelectBackend() (*Backend, error) {
    // TODO 1: Get healthy backends using GetHealthyBackends()
    // TODO 2: Check if healthy backend slice is empty and return error if so
    // TODO 3: Call current algorithm's SelectBackend method with healthy backends
    // TODO 4: If algorithm returns nil, return "no backend available" error
    // TODO 5: Increment selected backend's connection count
    // TODO 6: Update backend's LastUsed timestamp
    // TODO 7: Return selected backend
    // Hint: Use bm.currentAlgorithm.SelectBackend(healthyBackends)
}
```

Language-Specific Implementation Hints

Go-Specific Patterns:

- Use `sync.RWMutex` for backend pool access with frequent reads and infrequent writes
- Implement atomic backend slice updates using `atomic.Value` to avoid holding locks during algorithm selection
- Use `sync.Once` for lazy initialization of health checkers and algorithm instances
- Implement graceful shutdown using context cancellation and `sync.WaitGroup` for cleanup coordination

Thread Safety Guidelines:

- All connection count operations must use `sync/atomic` functions

- Backend health state updates need individual mutexes per backend to avoid blocking pool operations
- Use copy-on-write semantics for backend slice updates to provide consistent algorithm views
- Implement lock ordering (pool mutex before individual backend mutexes) to prevent deadlocks

Milestone Checkpoint

After implementing the backend pool manager, verify the following behavior:

Basic Functionality Test:

```
go test ./internal/backend/... -v
```
BASH

Manual Verification Steps:

1. Create a test configuration with 3 backend servers
2. Start the backend manager and verify all backends are loaded
3. Query `GetHealthyBackends()` and confirm all 3 backends are returned
4. Simulate a backend failure by updating health state
5. Verify `GetHealthyBackends()` now returns only 2 backends
6. Add a new backend via configuration reload
7. Confirm the new backend appears in the healthy backend list

Expected Behavior:

- Backend creation should generate stable, unique IDs for each URL
- Connection counting should remain accurate under concurrent access
- Health state updates should not block backend selection operations
- Configuration reloads should not drop existing connections

Signs of Problems:

- Race condition warnings from `go test -race`
- Connection counts that become negative or grow without bounds
- Backends that remain in healthy list despite being marked unhealthy
- Panics during concurrent access to backend pool or health state

This checkpoint ensures the backend pool manager provides a solid foundation for implementing load balancing algorithms and health checking in subsequent milestones.

Load Balancing Algorithms

> **Milestone(s):** Milestone 2 (Round Robin Distribution), Milestone 4 (Additional Algorithms) — this section implements the core algorithms that determine how requests are distributed across backend servers

Mental Model: Traffic Director

Imagine you're a traffic director at a busy intersection with multiple roads leading to the same destination. Each road represents a backend server, and every car represents an incoming HTTP request. Your job is to decide which road each car should take to reach their destination efficiently.

Just as a traffic director has different strategies for routing vehicles, a load balancer employs various algorithms to distribute requests. You might use **round-robin routing** like a simple traffic light that cycles through each road in turn, ensuring equal opportunity for all routes. For **weighted distribution**, imagine some roads are highways while others are small streets — you'd send more cars down the highways because they can handle more traffic. With **least-busy routing**, you'd count how many cars are currently on each road and direct new cars to the emptiest route. For **consistent routing**, you might assign specific types of vehicles (delivery trucks, emergency vehicles) to always use the same road to maintain familiarity and efficiency.

Each strategy has trade-offs. Round-robin is simple but doesn't account for road capacity differences. Least-busy routing is dynamic but requires constant monitoring. Consistent routing provides predictability but might create uneven distribution if certain vehicle types are more common. As a traffic director, you need to understand these trade-offs and choose the right strategy for your specific intersection's needs.

Round Robin Algorithm

The **round-robin algorithm** represents the simplest and most intuitive approach to load balancing. It operates like a circular queue, selecting each backend server in sequential order and cycling back to the first server after reaching the last one. This algorithm ensures that over time, each backend receives an approximately equal number of requests, regardless of their processing speed or current load.

The core challenge in implementing round-robin lies in maintaining thread safety when multiple goroutines simultaneously select backends. The algorithm maintains a shared counter that tracks the next backend to select, but this counter must be incremented atomically to prevent race conditions that could lead to uneven distribution or incorrect selections.

Algorithm Implementation Logic:

1. **Counter Management**: Maintain an atomic counter that tracks the next backend index to select from the healthy backend pool
2. **Atomic Increment**: Use atomic operations to increment the counter and retrieve the current value in a single, thread-safe operation
3. **Modulo Operation**: Apply modulo arithmetic to wrap the counter around to zero when it exceeds the number of available backends
4. **Health Filtering**: Only consider backends that are currently marked as healthy by the health checking system
5. **Empty Pool Handling**: Return an error or nil when no healthy backends are available for selection
6. **Index Validation**: Ensure the calculated index is within the bounds of the current healthy backend slice

The round-robin algorithm maintains fairness by treating all healthy backends equally, without considering their current load, processing capacity, or response times. This simplicity makes it highly predictable and easy to debug, but it may not be optimal when backend servers have different capabilities or when request processing times vary significantly.

Round Robin State	Field	Type	Description
Counter	`counter`	`*int64`	Atomic counter tracking next backend index to select
LastSelected	`lastSelected`	`*Backend`	Reference to the most recently selected backend for debugging
SelectionCount	`selectionCount`	`*int64`	Total number of backend selections performed since startup

Round Robin Methods	Parameters	Returns	Description
`SelectBackend`	`backends []*Backend`	`*Backend`	Selects next backend using round-robin algorithm
`Name`	none	`string`	Returns "round_robin" as algorithm identifier
`Reset`	none	`error`	Resets internal counter to zero for testing
`GetStats`	none	`map[string]int64`	Returns selection count and current counter value

> **Design Insight**: The round-robin algorithm's strength lies in its predictability and simplicity. Unlike more complex algorithms, its behavior is completely deterministic given the same sequence of healthy backends, making it ideal for testing and debugging load balancer behavior.

Atomic Counter Implementation Details:

The atomic counter implementation requires careful attention to integer overflow and wrap-around behavior. When the counter approaches the maximum value for `int64`, it should wrap around to zero gracefully without causing panics or negative indices. The modulo operation naturally handles this wrap-around, but the implementation must ensure that the modulo operand (number of backends) is never zero.

Thread Safety Considerations:

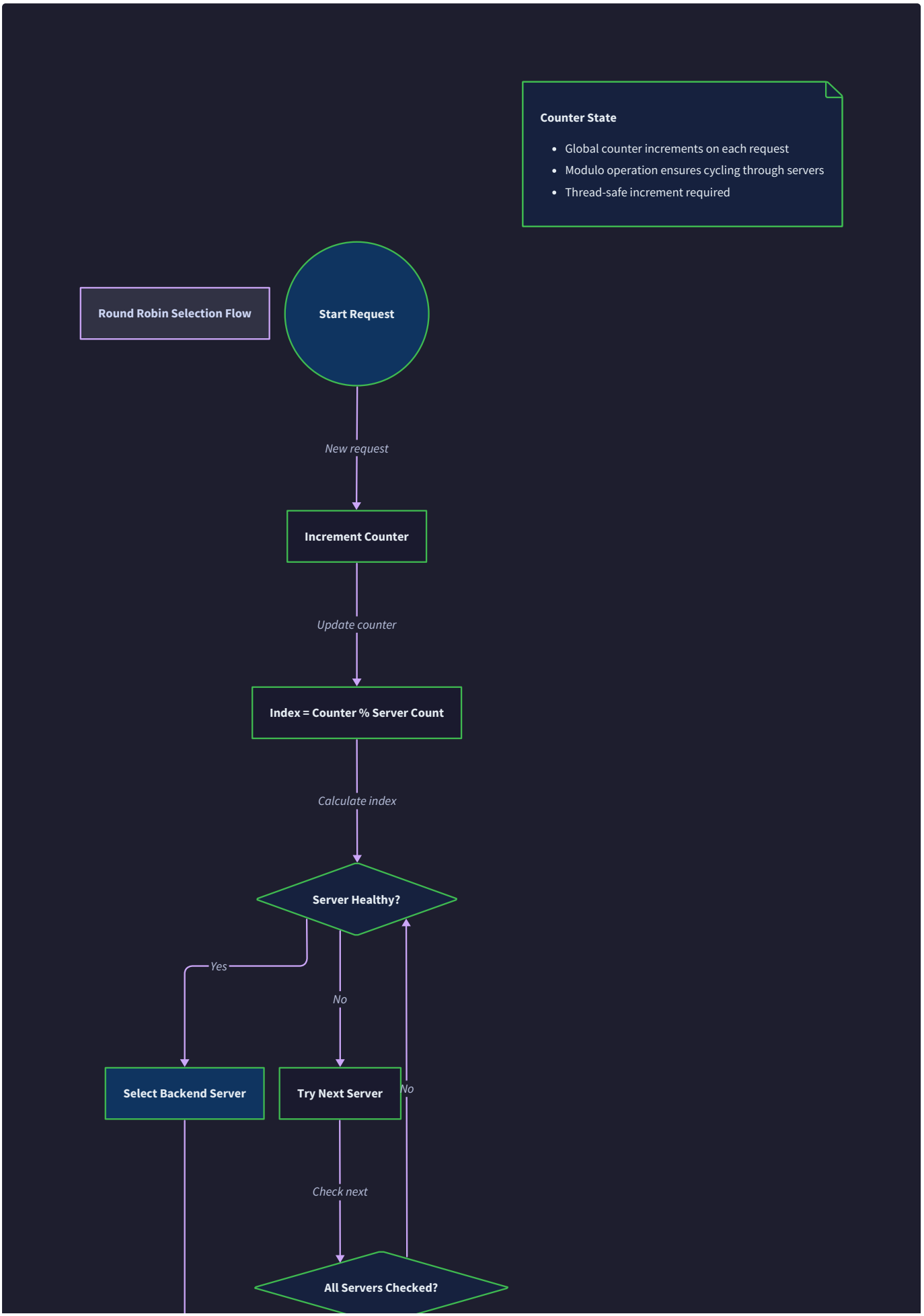
Multiple goroutines calling `SelectBackend` simultaneously must receive different backends when possible. The atomic increment operation ensures that each goroutine gets a unique counter value, preventing duplicate selections. However, the healthy backend slice itself may change between the atomic increment and the actual backend selection, requiring careful handling of race conditions where backends are added or removed during selection.

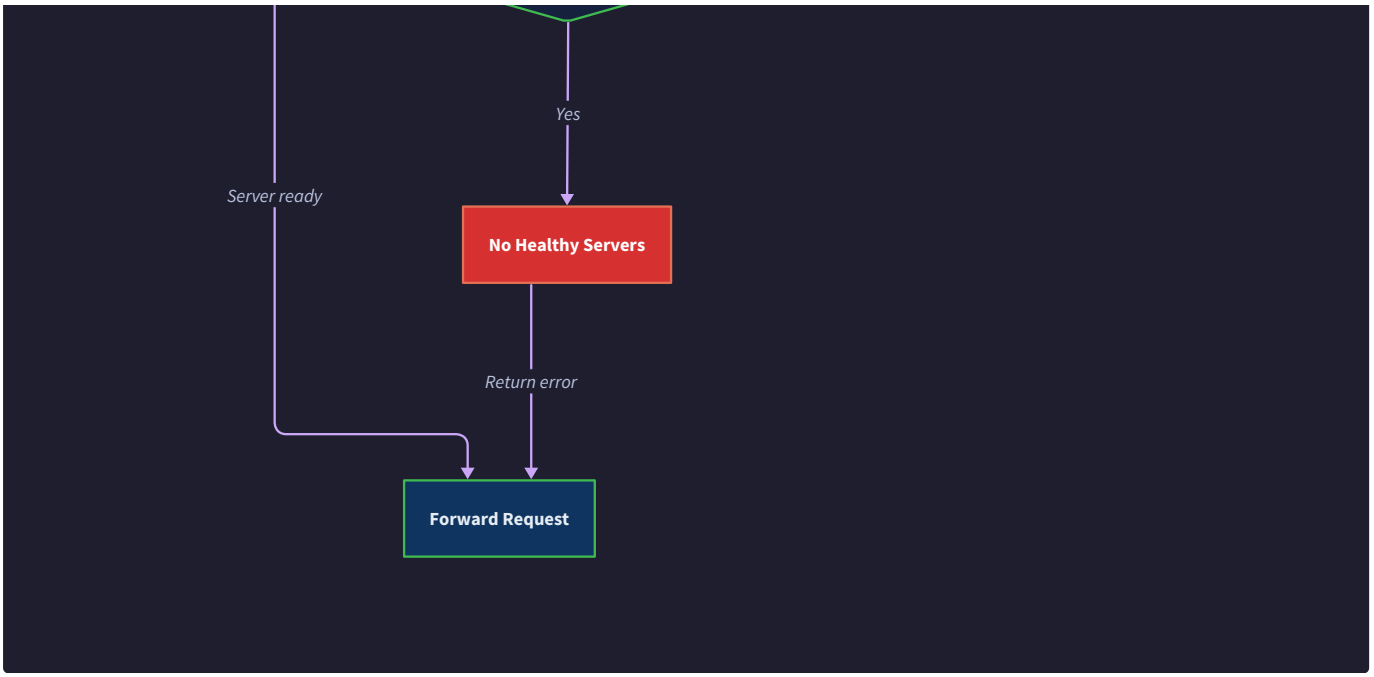
Example Round-Robin Selection Sequence:

Consider a load balancer with three healthy backends: Backend-A, Backend-B, and Backend-C. The selection sequence proceeds as follows:

1. Request 1: Counter=0, 0 % 3 = 0, selects Backend-A, increment counter to 1
2. Request 2: Counter=1, 1 % 3 = 1, selects Backend-B, increment counter to 2
3. Request 3: Counter=2, 2 % 3 = 2, selects Backend-C, increment counter to 3
4. Request 4: Counter=3, 3 % 3 = 0, selects Backend-A, increment counter to 4
5. Request 5: Counter=4, 4 % 3 = 1, selects Backend-B, increment counter to 5

This pattern continues indefinitely, ensuring each backend receives every third request in a predictable cycle.


Round Robin Selection Flow
Start Request
Counter State
Global counter increments on each request
Modulo operation ensures cycling through servers
Thread-safe increment required
New request
Increment Counter
Update counter
Index = Counter % Server Count
Calculate index
Server Healthy?
Yes
No
Select Backend Server
Try Next Server
No
Check next
All Servers Checked?



Weighted Round Robin Algorithm

The **weighted round-robin algorithm** extends basic round-robin by allowing administrators to assign different weights to backend servers, enabling proportional traffic distribution based on server capacity, performance, or cost considerations. A backend with weight 3 receives three times as many requests as a backend with weight 1 over a sufficient number of selections.

The challenge in weighted round-robin lies in achieving smooth distribution that avoids sending consecutive requests to high-weight backends. A naive implementation might select a weight-3 backend three times in a row before moving to the next server, creating bursty traffic patterns. The smooth weighted round-robin algorithm addresses this by gradually distributing weighted selections across time.

Smooth Weighted Round Robin Logic:

The algorithm maintains a current weight for each backend that starts at zero and increases by the backend's configured weight on each selection round. The backend with the highest current weight is selected, and its current weight is then decreased by the sum of all backend weights. This approach ensures that high-weight backends are selected more frequently but not consecutively.

1. **Weight Initialization**: Set each backend's current weight to zero at algorithm startup
2. **Weight Calculation**: In each selection round, increase each backend's current weight by its configured weight
3. **Maximum Selection**: Select the backend with the highest current weight value
4. **Weight Adjustment**: Decrease the selected backend's current weight by the total weight sum of all backends
5. **Health Filtering**: Only consider backends that are currently healthy for weight calculations
6. **Zero Weight Handling**: Treat backends with zero or negative weights as having weight 1 to prevent exclusion

Weighted Round Robin State	Field	Type	Description
BackendWeights	`backendWeights`	`map[string]int`	Configured weight for each backend by ID
CurrentWeights	`currentWeights`	`map[string]int`	Current weight values used for selection
TotalWeight	`totalWeight`	`int`	Sum of all configured backend weights
WeightMutex	`weightMutex`	`sync.RWMutex`	Protects weight maps from concurrent access

Weighted Round Robin Methods	Parameters	Returns	Description
`SelectBackend`	`backends []*Backend`	`*Backend`	Selects backend using smooth weighted algorithm
`UpdateWeights`	`weights map[string]int`	`error`	Updates backend weights with validation
`GetCurrentWeights`	none	`map[string]int`	Returns current weight values for debugging
`ResetWeights`	none	`error`	Resets all current weights to zero

Smooth Distribution Example:

Consider three backends with weights: A=5, B=1, C=1 (total weight = 7). The selection sequence demonstrates smooth distribution:

Round	A Current	B Current	C Current	Selected	A After	B After	C After
1	5	1	1	A (max)	-2 (5-7)	1	1
2	3 (-2+5)	2 (1+1)	2 (1+1)	A (max)	-4 (3-7)	2	2
3	1 (-4+5)	3 (2+1)	3 (2+1)	B or C	1	-4 or 3	3 or -4
4	6 (1+5)	-3 or 4	4 or -3	A (max)	-1 (6-7)	same	same

This pattern ensures Backend A receives approximately 5/7 of requests while Backends B and C each receive 1/7, but the selections are distributed smoothly rather than in bursts.

> **Decision: Smooth Weighted Round Robin vs Simple Weighted Round Robin**
>
> - **Context**: Need to distribute requests proportionally while avoiding traffic bursts to high-weight backends
> - **Options Considered**: Simple weighted (repeat selections), Smooth weighted (gradual distribution), Random weighted (probabilistic)
> - **Decision**: Implement smooth weighted round-robin algorithm
> - **Rationale**: Provides proportional distribution without creating request bursts, better for backend stability and performance
> - **Consequences**: More complex state management but significantly better traffic smoothness and backend resource utilization

Least Connections Algorithm

The **least connections algorithm** selects the backend server currently handling the fewest active connections, making it responsive to actual backend load rather than just request count. This algorithm excels when request processing times vary significantly or when backends have different processing capacities, as it naturally routes new requests to less busy servers.

The algorithm requires accurate tracking of active connection counts for each backend server. A connection becomes active when a request is forwarded to a backend and remains active until the response is completely returned to the client. This tracking must be thread-safe and atomic to prevent race conditions that could lead to incorrect connection counts.

Connection Tracking Implementation:

1. **Increment on Request**: Atomically increment the backend's active connection count when forwarding a request
2. **Decrement on Response**: Atomically decrement the count when the response is fully processed, regardless of success or failure
3. **Error Handling**: Ensure connection counts are decremented even when backend requests fail or time out
4. **Cleanup on Backend Removal**: Reset connection counts when backends are removed from the pool
5. **Recovery on Startup**: Initialize all connection counts to zero when the load balancer starts up
6. **Overflow Protection**: Handle potential integer overflow in connection counters gracefully

Least Connections State	Field	Type	Description
ConnectionCounts	`connectionCounts`	`map[string]*int64`	Active connection count for each backend by ID
CounterMutex	`counterMutex`	`sync.RWMutex`	Protects connection count map from concurrent access
MaxConnections	`maxConnections`	`map[string]int`	Maximum allowed connections per backend

Least Connections Methods	Parameters	Returns	Description
`SelectBackend`	`backends []*Backend`	`*Backend`	Selects backend with fewest active connections
`IncrementConnections`	`backendID string`	`int64`	Atomically increments connection count for backend
`DecrementConnections`	`backendID string`	`int64`	Atomically decrements connection count for backend
`GetConnectionCounts`	none	`map[string]int64`	Returns current connection counts for all backends

Selection Logic with Tie Breaking:

When multiple backends have the same lowest connection count, the algorithm needs a consistent tie-breaking strategy. Common approaches include selecting the first backend encountered, using round-robin among tied backends, or selecting randomly. The choice impacts load distribution when many backends are idle or when connection counts frequently tie.

Connection Count Synchronization:

The critical challenge lies in ensuring that connection count increments and decrements are properly paired and that counts accurately reflect reality. Network failures, timeout errors, and client disconnections can all cause situations where a connection count might become permanently incorrect if not handled properly.

Example Selection with Varying Loads:

Consider three backends with current active connections: A=5, B=2, C=8. A new request arrives:

1. **Evaluation**: Backend B has the fewest connections (2)
2. **Selection**: Choose Backend B for the new request
3. **Increment**: Backend B's count becomes 3 (atomically)
4. **Processing**: Request is forwarded to Backend B
5. **Completion**: When Backend B responds, decrement count back to 2

If Backend B fails to respond or times out, the decrement operation must still occur to prevent connection count drift.

Handling Maximum Connection Limits:

Advanced implementations can respect per-backend maximum connection limits, skipping backends that have reached their connection capacity even if they currently have the fewest connections. This prevents overloading backends and provides graceful degradation when backends approach their processing limits.

IP Hash Algorithm

The **IP hash algorithm** provides **session affinity** by consistently routing requests from the same client IP address to the same backend server. This algorithm is essential for applications that maintain server-side session state, shopping carts, or user authentication tokens that are not shared across backend servers.

The algorithm computes a hash value from the client's IP address and uses modulo arithmetic to map this hash to a specific backend server. The key requirement is consistency — the same IP address must always map to the same backend server as long as the backend pool remains unchanged.

Hash Function Selection and Properties:

1. **Deterministic**: The same IP address must always produce the same hash value
2. **Uniform Distribution**: Hash values should be evenly distributed across the output space to prevent clustering
3. **Fast Computation**: Hash calculation should be computationally efficient for high-throughput scenarios
4. **Collision Handling**: Different IP addresses may hash to the same value, which is acceptable with modulo mapping

IP Hash State	Field	Type	Description
HashSeed	`hashSeed`	`uint64`	Random seed for hash function to prevent prediction
BackendMapping	`backendMapping`	`map[string]string`	Cache of IP-to-backend mappings for debugging
ConsistencyMode	`consistencyMode`	`string`	Algorithm variant: "simple" or "consistent_hash"

IP Hash Methods	Parameters	Returns	Description
`SelectBackend`	`backends []*Backend, clientIP string`	`*Backend`	Selects backend based on client IP hash
`HashIP`	`ip string`	`uint64`	Computes hash value for given IP address
`GetIPMapping`	`ip string`	`string`	Returns backend ID that IP would map to
`ClearMappingCache`	none	`error`	Clears IP mapping cache for memory management

Simple Hash Implementation Logic:

1. **IP Address Parsing**: Extract the client IP from the request, handling both IPv4 and IPv6 addresses
2. **Hash Computation**: Apply a cryptographic hash function (SHA-256 or FNV) to the IP address bytes
3. **Modulo Mapping**: Use modulo arithmetic to map the hash value to a backend index
4. **Health Validation**: Ensure the selected backend is currently healthy, fallback to round-robin if unhealthy
5. **IPv6 Handling**: Normalize IPv6 addresses to handle different representation formats consistently

Consistent Hashing Alternative:

For environments where backends are frequently added or removed, simple modulo hashing causes significant remapping when the backend count changes. Consistent hashing algorithms minimize this disruption by maintaining most existing IP-to-backend mappings when backends change.

Example IP Hashing Sequence:

Client IP addresses map consistently to backends:

- IP 192.168.1.100: hash(IP) = 12345, 12345 % 3 = 0, maps to Backend-A
- IP 192.168.1.101: hash(IP) = 67890, 67890 % 3 = 1, maps to Backend-B
- IP 192.168.1.102: hash(IP) = 24680, 24680 % 3 = 2, maps to Backend-C
- IP 192.168.1.100: (later request) maps to Backend-A again (consistent)

Session Affinity Considerations:

Session affinity creates an inherent trade-off with load distribution. If certain IP addresses generate significantly more traffic than others, the load becomes uneven across backends. The algorithm may need fallback mechanisms when a client's designated backend becomes unhealthy, temporarily breaking session affinity to maintain service availability.

> **Design Insight**: IP hash algorithms must balance session consistency with load distribution fairness. In practice, many deployments use IP hashing only when session affinity is required and fall back to other algorithms when all backend servers are stateless.

Random Selection Algorithm

The **random selection algorithm** provides the simplest statistical approach to load balancing by selecting backends with equal probability using a cryptographically secure random number generator. While seemingly naive, random selection often achieves surprisingly good load distribution with minimal state management overhead.

Random selection shines in scenarios where request processing times are relatively uniform and backend capabilities are similar. Over a large number of requests, the law of large numbers ensures that each backend receives approximately equal traffic, but short-term distribution may be uneven compared to round-robin.

Random Selection Implementation:

1. **Random Number Generation**: Use a cryptographically secure random number generator seeded at startup
2. **Range Scaling**: Scale the random number to the range [0, number_of_healthy_backends)
3. **Index Selection**: Use the scaled random number as an index into the healthy backend slice
4. **Health Filtering**: Only include currently healthy backends in the selection pool
5. **Thread Safety**: Ensure the random number generator is safe for concurrent access across goroutines

Random Selection State	Field	Type	Description
RandomGenerator	`rng`	`*rand.Rand`	Seeded random number generator instance
GeneratorMutex	`rngMutex`	`sync.Mutex`	Protects random generator from concurrent access
SelectionHistory	`history`	`[]string`	Recent backend selections for debugging/testing

Random Selection Methods	Parameters	Returns	Description
`SelectBackend`	`backends []*Backend`	`*Backend`	Randomly selects a backend from healthy pool
`SetSeed`	`seed int64`	`error`	Sets random number generator seed for testing
`GetDistribution`	`window int`	`map[string]int`	Returns selection counts over recent window

Statistical Distribution Properties:

Random selection provides several valuable statistical properties. The variance in backend selection decreases as the number of requests increases, following the central limit theorem. For N backends and R requests, each backend receives approximately R/N requests with a standard deviation of sqrt(R * (1/N) * (1 - 1/N)).

Advantages of Random Selection:

- **Stateless**: No counters, weights, or connection tracking required
- **Simple**: Minimal code complexity and fewer edge cases to handle
- **Scalable**: Performance doesn't degrade as the number of backends increases
- **Resilient**: No state corruption possible during backend pool changes

Example Random Distribution:

Over 1000 requests to 3 backends, random selection might produce:

- Backend-A: 334 requests (33.4%)
- Backend-B: 329 requests (32.9%)
- Backend-C: 337 requests (33.7%)

While not perfectly even like round-robin, this distribution is acceptable for most applications and required no state synchronization.

Cryptographically Secure Randomness:

The random number generator should be seeded with a cryptographically secure source to prevent predictable selection patterns that could be exploited for denial-of-service attacks. However, the generator itself need not be cryptographically secure since load balancing decisions are not security-sensitive.

Architecture Decisions

Decision: Algorithm Interface Design

- **Context**: Need a common interface that supports multiple load balancing algorithms with different state requirements and parameters
- **Options Considered**: Single method interface, Strategy pattern with factory, Plugin-based architecture
- **Decision**: Implement Strategy pattern with common `Algorithm` interface containing `SelectBackend` and `Name` methods
- **Rationale**: Strategy pattern allows runtime algorithm switching while keeping implementation details encapsulated within each algorithm
- **Consequences**: Enables easy addition of new algorithms and runtime reconfiguration, but requires careful handling of algorithm-specific state

Algorithm Interface Options	Pros	Cons	Chosen?
Single Method Interface	Simple to implement and use	Cannot handle algorithm-specific configuration or state	No
Strategy Pattern	Clean separation, runtime switching, extensible	Slightly more complex state management	**Yes**
Plugin Architecture	Ultimate flexibility, external algorithms possible	Complex loading, versioning, and security concerns	No

Decision: Algorithm State Management

- **Context**: Algorithms need to maintain state (counters, weights, connection counts) that must be thread-safe and persistent across requests
- **Options Considered**: Global shared state, Per-algorithm state encapsulation, Stateless algorithms only
- **Decision**: Encapsulate state within each algorithm implementation using appropriate synchronization primitives
- **Rationale**: Encapsulation prevents state conflicts between algorithms and allows each to optimize its synchronization strategy
- **Consequences**: Each algorithm manages its own thread safety, but state is isolated and easier to reason about

Decision: Runtime Algorithm Switching

- **Context**: Administrators need to change load balancing algorithms without restarting the load balancer or dropping connections
- **Options Considered**: Restart required, Graceful transition, Immediate switch, Dual-algorithm mode
- **Decision**: Implement immediate switching with state preservation where possible
- **Rationale**: Immediate switching provides operational flexibility without service disruption, most algorithm state can be safely discarded
- **Consequences**: Enables rapid response to changing traffic patterns but may cause temporary distribution irregularities during transitions

Algorithm Selection Strategy:

The load balancer supports runtime selection of the active algorithm through configuration updates. When switching algorithms, the system preserves connection counts (for least connections) but resets algorithm-specific state like round-robin counters and weighted round-robin current weights.

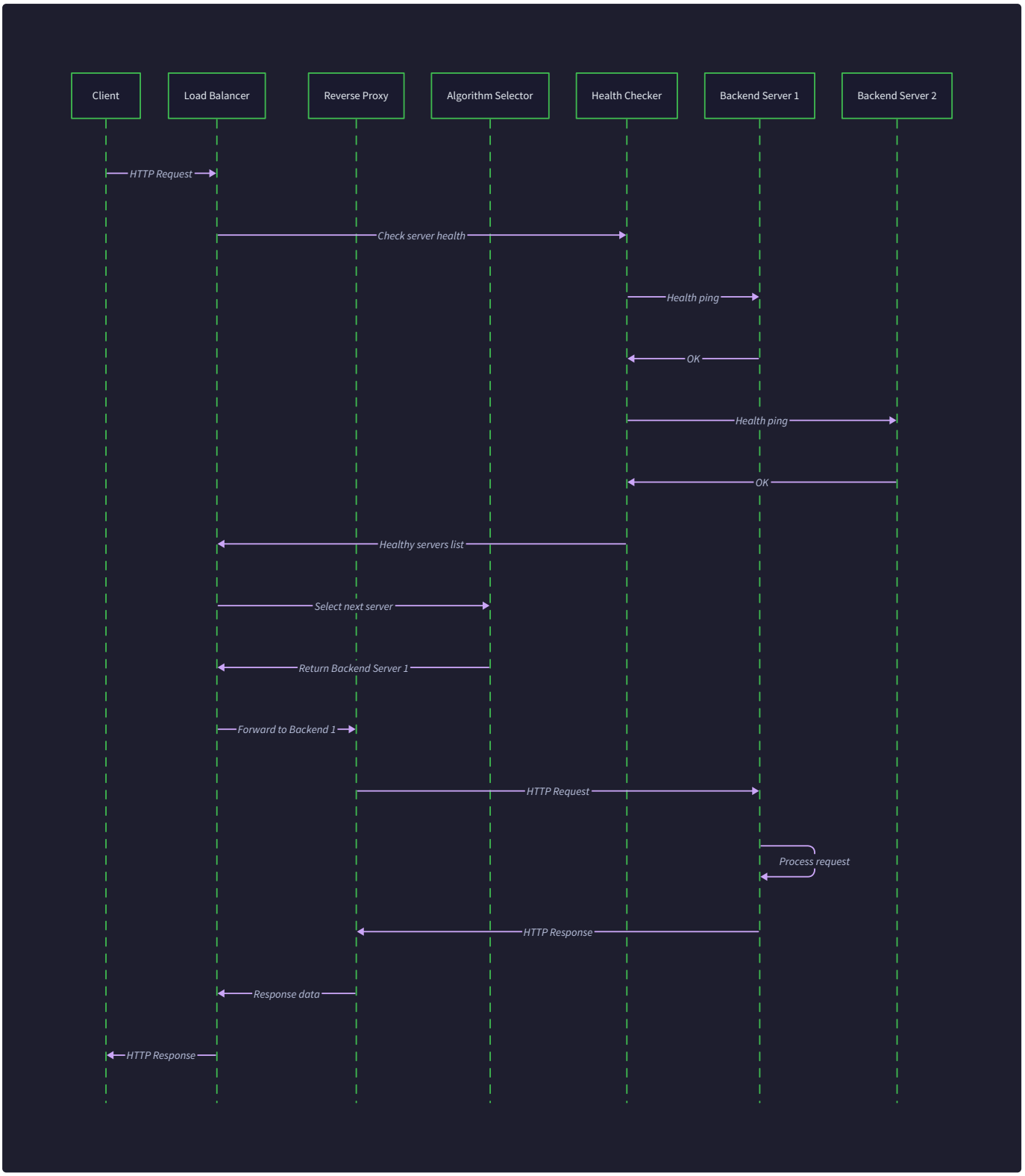
Algorithm Switching	Current State	New Algorithm	State Handling
Round Robin → Weighted	Counter reset	Weights initialized	Smooth transition
Least Connections → Round Robin	Connections preserved	Counter starts at 0	Distribution temporarily uneven
IP Hash → Random	Mappings cleared	No state needed	Immediate random selection
Any → Least Connections	State varies	Connection tracking starts	Gradual accuracy improvement

Thread Safety Architecture:

Each algorithm implementation manages its own thread safety using appropriate synchronization primitives. Atomic operations are preferred for simple counters, while mutexes protect complex data structures like weight maps and connection count tables.

Algorithm Performance Characteristics:

Algorithm	Time Complexity	Space Complexity	Thread Contention	Best Use Case
Round Robin	O(1)	O(1)	Low (atomic counter)	Uniform requests, equal backends
Weighted Round Robin	O(n) backends	O(n) state	Medium (mutex)	Different backend capacities
Least Connections	O(n) comparison	O(n) counters	High (frequent updates)	Variable request processing times
IP Hash	O(1) hash + O(1) select	O(1)	Low (stateless)	Session affinity required
Random	O(1)	O(1)	Low (minimal state)	Simple, stateless applications



Common Pitfalls

⚠️ Pitfall: Round Robin Counter Race Conditions

A common mistake is using non-atomic operations to increment the round-robin counter, leading to race conditions where multiple goroutines read the same counter value and select the same backend simultaneously. This breaks the even distribution guarantee.

Why it's wrong: Non-atomic counter increments create race conditions where two concurrent requests might get the same counter value, causing uneven distribution and potentially overloading specific backends.

How to fix: Use `atomic.AddInt64()` to both read and increment the counter in a single atomic operation:

```go
// Wrong: Race condition possible
nextIndex := rr.counter
rr.counter = (rr.counter + 1) % len(backends)

// Correct: Atomic operation
nextIndex := atomic.AddInt64(&rr.counter, 1) % int64(len(backends))
```

⚠️ Pitfall: Weighted Round Robin Integer Overflow

Weighted round-robin implementations often fail to handle integer overflow when current weights grow large over time. With high weights and many requests, current weight values can exceed integer limits and wrap to negative numbers.

Why it's wrong: Integer overflow causes current weights to become negative, breaking the weight comparison logic and potentially causing panics or incorrect selections.

How to fix: Implement weight normalization that resets all current weights to their relative positions when approaching overflow limits, or use larger integer types with overflow detection.

⚠️ Pitfall: Least Connections Count Drift

Failing to properly decrement connection counts when requests complete, fail, or time out leads to connection count drift where backends appear permanently busy even when idle.

Why it's wrong: Connection counts that drift upward prevent backends from receiving new requests, eventually making them appear overloaded when they're actually idle.

How to fix: Use defer statements to ensure connection counts are decremented regardless of how request processing completes:

```go
// Increment when starting request
atomic.AddInt64(backend.ActiveConnections, 1)

// Always decrement on completion
defer atomic.AddInt64(backend.ActiveConnections, -1)
```

⚠️ Pitfall: IP Hash Inconsistency After Backend Changes

Simple IP hash implementations using modulo arithmetic produce completely different mappings when backends are added or removed, breaking session affinity for most clients.

Why it's wrong: When backend count changes, modulo arithmetic causes most IP addresses to map to different backends, losing session state and breaking user experience.

How to fix: Implement consistent hashing or provide graceful degradation where IP hash falls back to other algorithms when backend pool changes are detected.

⚠️ Pitfall: Hash Function Predictability

Using simple hash functions or predictable seeds makes the IP hash algorithm vulnerable to deliberate traffic concentration attacks where attackers generate requests from IP addresses that all hash to the same backend.

Why it's wrong: Predictable hashing allows malicious clients to concentrate traffic on specific backends, creating denial-of-service conditions.

How to fix: Use cryptographically secure hash functions with random seeds that change on each load balancer restart, and consider rate limiting per-IP to prevent concentration attacks.

⚠️ Pitfall: Algorithm State Not Cleared on Pool Changes

When the backend pool changes (servers added/removed), algorithm state like round-robin counters and weight mappings may contain references to invalid backends or incorrect indices.

Why it's wrong: Stale algorithm state can cause index out-of-bounds errors, selections of non-existent backends, or skewed distribution patterns.

How to fix: Reset algorithm state when backend pool changes are detected, or implement algorithms that validate backend references before selection.

⚠️ Pitfall: No Healthy Backends Handling

Algorithms often fail to handle the case where no backends are currently healthy, leading to panics, infinite loops, or incorrect error responses.

Why it's wrong: When all backends are unhealthy, selection algorithms may panic on empty slices or return nil pointers that cause segmentation faults.

How to fix: Always check for empty or nil backend slices before selection and return appropriate errors or default responses when no backends are available.

Implementation Guidance

The load balancing algorithms form the decision-making core of the load balancer, determining how incoming requests are distributed across available backend servers. Each algorithm optimizes for different traffic patterns and operational requirements.

Technology Recommendations:

Component	Simple Option	Advanced Option
Algorithm Interface	Simple strategy pattern with method dispatch	Plugin architecture with dynamic loading
State Management	In-memory maps with mutexes	Lock-free concurrent data structures
Random Generation	math/rand with mutex protection	crypto/rand for cryptographic security
Hash Functions	FNV hash for IP hashing	SHA-256 for cryptographic applications
Atomic Operations	sync/atomic for counters	Custom lock-free algorithms
Configuration	Static algorithm selection	Hot-swappable algorithms with state migration

Recommended File Structure:

```
internal/algorithms/
  algorithm.go           ← Algorithm interface definition
  round_robin.go         ← Round robin implementation
  weighted_round_robin.go ← Weighted round robin implementation
  least_connections.go   ← Least connections implementation
  ip_hash.go             ← IP hash implementation
  random.go              ← Random selection implementation
  factory.go             ← Algorithm factory for creation
  manager.go             ← Algorithm manager for runtime switching
  algorithm_test.go      ← Comprehensive algorithm tests
```

Algorithm Interface Definition (Complete):

```go
package algorithms

import (
    "errors"
    "sync"
    "github.com/yourusername/loadbalancer/internal/types"
)

// Algorithm defines the interface for backend selection algorithms
type Algorithm interface {
    // SelectBackend chooses a backend from the healthy pool
    SelectBackend(backends []*types.Backend) *types.Backend

    // Name returns the algorithm identifier
    Name() string

    // Reset clears algorithm state for testing
    Reset() error

    // GetStats returns algorithm-specific statistics
    GetStats() map[string]interface{}
}

// Manager handles algorithm switching and lifecycle
type Manager struct {
    current    Algorithm
    algorithms map[string]Algorithm
    mutex      sync.RWMutex
}

// NewManager creates a new algorithm manager with all algorithms registered
func NewManager() *Manager {
    return &Manager{
        algorithms: map[string]Algorithm{
            "round_robin":          NewRoundRobin(),
            "weighted_round_robin": NewWeightedRoundRobin(),
            "least_connections":    NewLeastConnections(),
            "ip_hash":              NewIPHash(),
            "random":               NewRandom(),
        },
        current: NewRoundRobin(), // default algorithm
    }
```

```go
}

// SelectBackend delegates to the current algorithm
func (m *Manager) SelectBackend(backends []*types.Backend) *types.Backend {
    m.mutex.RLock()
    defer m.mutex.RUnlock()
    return m.current.SelectBackend(backends)
}

// SwitchAlgorithm changes the active algorithm at runtime
func (m *Manager) SwitchAlgorithm(name string) error {
    m.mutex.Lock()
    defer m.mutex.Unlock()

    algorithm, exists := m.algorithms[name]
    if !exists {
        return errors.New("unknown algorithm: " + name)
    }

    m.current = algorithm
    return nil
}
```

Round Robin Implementation (Complete):

```go
package algorithms

import (
    "sync/atomic"
    "github.com/yourusername/loadbalancer/internal/types"
)

// RoundRobin implements simple round-robin backend selection
type RoundRobin struct {
    counter int64
}

// NewRoundRobin creates a new round-robin algorithm instance
func NewRoundRobin() *RoundRobin {
    return &RoundRobin{counter: 0}
}

// SelectBackend selects the next backend in round-robin order
func (rr *RoundRobin) SelectBackend(backends []*types.Backend) *types.Backend {
    // TODO 1: Check if backends slice is empty or nil - return nil if so
    // TODO 2: Use atomic.AddInt64 to increment counter and get current value
    // TODO 3: Apply modulo operation to wrap counter within backend count
    // TODO 4: Ensure the modulo result is positive (handle negative wrap-around)
    // TODO 5: Return the backend at the calculated index
    // TODO 6: Handle potential race condition where backends slice changes between increment and access

    return nil // Remove this line when implementing
}

// Name returns the algorithm identifier
func (rr *RoundRobin) Name() string {
    return "round_robin"
}

// Reset clears the counter for testing
func (rr *RoundRobin) Reset() error {
    atomic.StoreInt64(&rr.counter, 0)
    return nil
}

// GetStats returns selection statistics
func (rr *RoundRobin) GetStats() map[string]interface{} {
    return map[string]interface{}{
        "selections": atomic.LoadInt64(&rr.counter),
```

```
        "algorithm":  "round_robin",
    }
}
```

Weighted Round Robin Implementation (Complete):

```go
package algorithms

import (
    "sync"
    "github.com/yourusername/loadbalancer/internal/types"
)

// WeightedRoundRobin implements smooth weighted round-robin selection
type WeightedRoundRobin struct {
    currentWeights map[string]int
    mutex          sync.RWMutex
}

// NewWeightedRoundRobin creates a new weighted round-robin algorithm
func NewWeightedRoundRobin() *WeightedRoundRobin {
    return &WeightedRoundRobin{
        currentWeights: make(map[string]int),
    }
}

// SelectBackend selects backend using smooth weighted round-robin
func (wrr *WeightedRoundRobin) SelectBackend(backends []*types.Backend) *types.Backend {
    // TODO 1: Acquire write lock to protect weight modifications
    // TODO 2: Initialize current weights for new backends not in map
    // TODO 3: Add each backend's configured weight to its current weight
    // TODO 4: Find the backend with the maximum current weight
    // TODO 5: Calculate total weight sum of all backends
    // TODO 6: Subtract total weight from selected backend's current weight
    // TODO 7: Release lock and return selected backend
    // TODO 8: Handle empty backends slice and zero-weight backends

    return nil // Remove this line when implementing
}

// UpdateWeights updates the configured weights for backends
func (wrr *WeightedRoundRobin) UpdateWeights(weights map[string]int) error {
    // TODO: Implement weight update with validation
    return nil
}
```

Algorithm Selection Guidelines:

Choose algorithms based on your specific requirements:

- **Round Robin**: Use when backends are identical and request processing times are uniform
- **Weighted Round Robin**: Use when backends have different capacities or you want to favor certain servers

- **Least Connections**: Use when request processing times vary significantly or during long-polling/websocket scenarios
- **IP Hash**: Use when you need session affinity and backend servers maintain client-specific state
- **Random**: Use for simple stateless applications where you want minimal overhead

Language-Specific Implementation Hints:

For Go implementations:

- Use `sync/atomic` for all counter operations to ensure thread safety
- Prefer `sync.RWMutex` over `sync.Mutex` for read-heavy operations like backend selection
- Use `crypto/rand` for cryptographically secure random number generation in production
- Consider using `sync.Pool` for frequently allocated temporary objects in high-throughput scenarios
- Handle integer overflow explicitly in weighted algorithms using checked arithmetic

Milestone Checkpoint (Milestone 2: Round Robin Distribution):

After implementing round-robin selection:

1. **Start the load balancer** with multiple backend servers configured
2. **Send 100 test requests** using a tool like curl or ab: `for i in {1..100}; do curl http://localhost:8080/test; done`
3. **Check backend logs** to verify each backend received approximately 33-34 requests (for 3 backends)
4. **Verify thread safety** by running concurrent requests: `ab -n 1000 -c 10 http://localhost:8080/test`
5. **Expected behavior**: Request distribution should be even (within ±5%) and no race conditions should occur

Signs something is wrong:

- **Uneven distribution**: Check for race conditions in counter increment
- **Panic on empty backends**: Add nil/empty slice checks
- **Same backend always selected**: Verify modulo operation and counter increment

Milestone Checkpoint (Milestone 4: Additional Algorithms):

After implementing all algorithms:

1. **Test algorithm switching**: Change configuration from round_robin to weighted_round_robin and reload
2. **Verify weighted distribution**: Configure backends with weights 3:2:1 and confirm proportional traffic
3. **Test session affinity**: Use IP hash and verify the same client IP always hits the same backend
4. **Test least connections**: Start backends with different response delays and verify faster backends get more requests
5. **Performance test**: Run `ab -n 10000 -c 100` to test under load

Expected results:

- **Algorithm switching** works without dropping connections
- **Weighted distribution** matches configured ratios within 10%
- **IP hash** provides 100% consistency for same IP
- **Least connections** adapts to backend performance differences

Health Checking System

> **Milestone(s):** Milestone 3 (Health Checks) — this section implements active health monitoring that enables the load balancer to detect backend failures and exclude unhealthy servers from request distribution

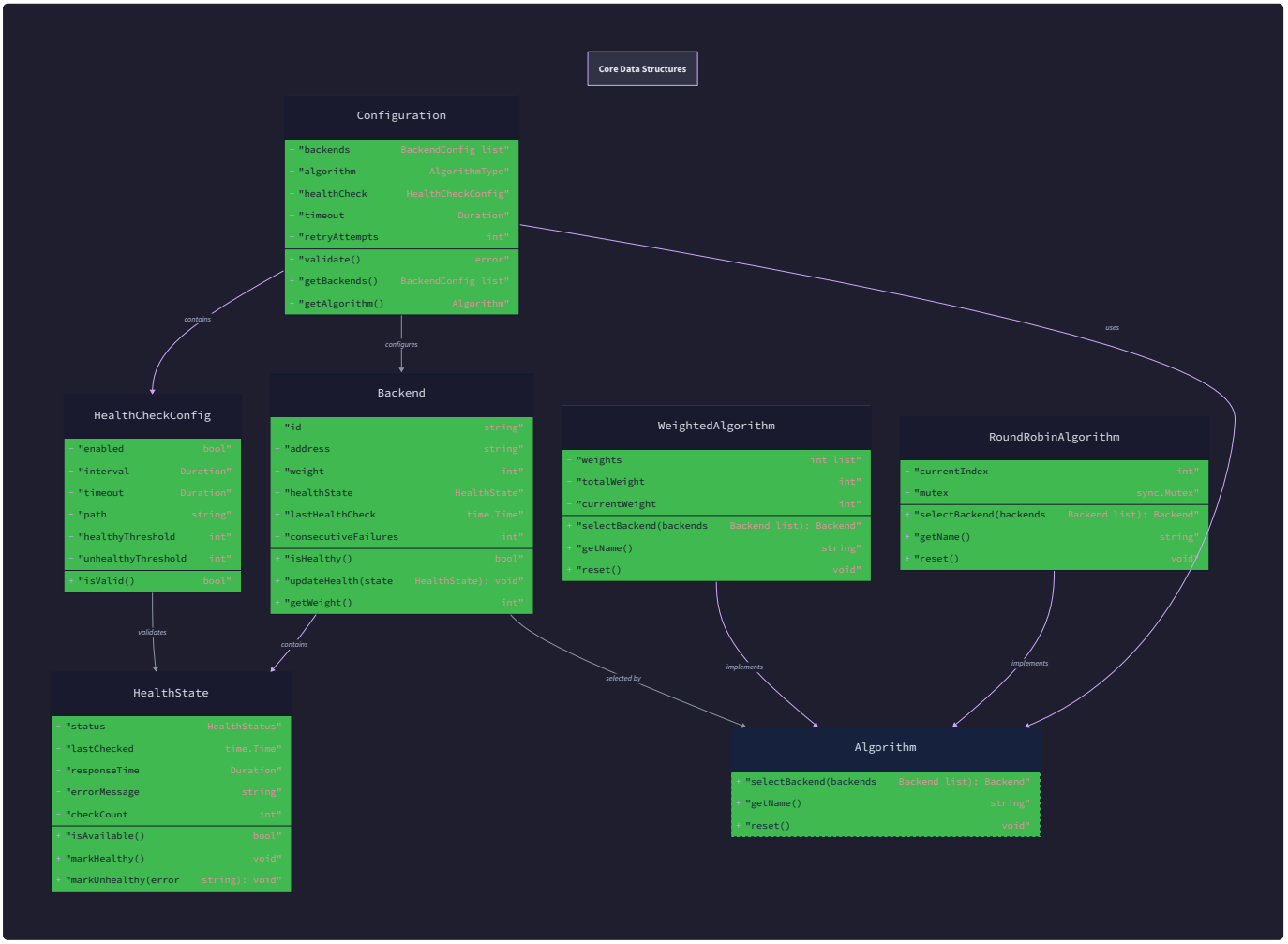
Mental Model: Medical Checkups

Think of the health checking system as a **medical monitoring program** for a professional sports team. Just as team doctors conduct regular checkups on athletes to ensure they're fit to play, our health checker continuously monitors backend servers to ensure they're ready to handle requests.

In this analogy, each backend server is like an athlete on the team roster. The health checker acts as the team doctor, performing scheduled examinations at regular intervals. Just as a doctor might check an athlete's pulse, reflexes, and stamina, our health checker sends HTTP requests to verify that backends can respond properly. When an athlete fails multiple checkups in a row, the doctor declares them unfit for play and removes them from the active lineup. Similarly, when a backend fails consecutive health checks, it gets marked as unhealthy and removed from the request distribution pool.

The recovery process mirrors an injured athlete's return to play. Just as an athlete must pass several consecutive medical clearances before rejoining the active roster, a previously unhealthy backend must succeed on multiple consecutive health checks before being restored to the available backend pool. This prevents flapping — rapidly switching between healthy and unhealthy states — which would disrupt the stability of request distribution.

The health checking system operates independently from request processing, much like how medical checkups happen during training sessions rather than during games. This separation ensures that health monitoring doesn't interfere with live request handling, and that failed health checks don't block ongoing traffic to healthy backends.



Periodic Health Probes

The **periodic health probes** form the foundation of active health monitoring, implementing scheduled HTTP requests to verify backend server availability. Unlike passive health checking that waits for request failures to detect problems, active probing discovers issues before they impact live traffic.

The health probe mechanism operates on a configurable interval, typically ranging from 5 to 60 seconds depending on the required detection speed and acceptable overhead. Each probe consists of an HTTP GET request sent to a designated health endpoint on the backend server, usually something like `/health` or `/status`. The probe includes specific headers to identify itself as a health check rather than regular application traffic.

Health Probe Component	Purpose	Implementation Details
Probe Scheduler	Manages timing of health checks	Uses `time.Ticker` with configurable interval
HTTP Client	Sends health check requests	Separate client with short timeout (2-5 seconds)
Response Evaluator	Determines probe success/failure	Checks status code, response time, optional body content
State Updater	Records probe results	Updates `HealthState` with atomic operations
Failure Classifier	Categorizes different failure types	Distinguishes connection errors, timeouts, and HTTP errors

The probe scheduler maintains independent timers for each backend server to prevent synchronized checking that could create load spikes. This **staggered scheduling** spreads health check traffic evenly over time, reducing the impact on backend servers and network infrastructure.

Decision: Dedicated Health Check HTTP Client

- **Context**: Health probes need different timeout and retry characteristics than request forwarding
- **Options Considered**:
  1. Reuse the main HTTP client from request forwarding
  2. Create a dedicated HTTP client for health checks
  3. Use raw TCP connections for simple connectivity testing
- **Decision**: Dedicated HTTP client with short timeouts
- **Rationale**: Health checks should fail fast (2-5 second timeout vs 30+ seconds for requests), need different retry logic, and shouldn't interfere with connection pooling for live traffic
- **Consequences**: Slightly more memory overhead but better isolation and more appropriate timeout behavior for health monitoring

Each health probe follows a structured sequence of operations designed to minimize false positives while detecting real failures quickly:

1. The probe scheduler triggers a health check for a specific backend server based on the configured interval
2. The health checker creates an HTTP GET request to the backend's health endpoint with appropriate headers
3. A dedicated HTTP client sends the request with a short timeout (typically 2-5 seconds)
4. The response evaluator examines the HTTP status code, comparing it against the expected healthy status (usually 200)
5. Optional response body validation checks for specific content patterns that indicate application readiness
6. The probe duration is measured and recorded for latency monitoring and timeout tuning
7. The result (success or failure with error details) is passed to the failure detection logic
8. The next probe is scheduled based on the configured interval, independent of the current result

Response evaluation goes beyond simple HTTP status code checking to provide more comprehensive health assessment. A backend might return HTTP 200 but include an error message in the response body indicating database connectivity issues or resource exhaustion. Advanced health probe configurations allow specifying expected response body patterns, response time thresholds, and custom headers that must be present.

Response Evaluation Criteria	Healthy Condition	Unhealthy Condition
HTTP Status Code	Matches expected status (default 200)	Any other status code
Response Time	Under configured threshold (default 5s)	Exceeds timeout threshold
Response Body	Contains expected pattern (if configured)	Missing pattern or contains error indicators
Connection	Successful TCP connection established	Connection refused, timeout, or network error
Content Type	Matches expected type (if configured)	Unexpected content type header

The health probe system handles various failure scenarios gracefully, categorizing them to enable appropriate response strategies:

Connection Failures: These include DNS resolution failures, connection refused errors, and network timeouts. Connection failures typically indicate infrastructure problems or complete backend unavailability and result in immediate failure classification.

HTTP Protocol Errors: These encompass invalid HTTP responses, malformed headers, and protocol violations. While less common than connection failures, protocol errors suggest backend application issues and also result in failure classification.

Application-Level Failures: These occur when the HTTP request succeeds but the application reports unhealthy status through error codes (500, 503) or response body content. Application-level failures often indicate partial outages or degraded performance.

> The health probe timeout should be significantly shorter than the request forwarding timeout. A backend that takes 10 seconds to respond to a health check will likely perform poorly for real requests, so it should be marked unhealthy even if it eventually responds.

Failure Detection Logic

The **failure detection logic** processes individual probe results to determine when a backend should be marked as unhealthy and removed from the active pool. Rather than making decisions based on single probe failures, the system uses **consecutive failure counting** to avoid false positives caused by temporary network glitches or brief backend hiccups.

The failure detection mechanism maintains a running count of consecutive failures for each backend server. When a health probe succeeds, the consecutive failure count resets to zero. When a probe fails, the count increments. Once the consecutive failure count reaches the configured **unhealthy threshold**, the backend transitions to the unhealthy state and gets excluded from request distribution.

Failure Detection State	Consecutive Failures	Action Taken	Next State Trigger
Healthy Monitoring	0 to (threshold - 1)	Continue normal checking	Failure count reaches unhealthy threshold
Transitioning to Unhealthy	Equals unhealthy threshold	Mark backend unhealthy, remove from pool	Additional failure (stays unhealthy) or success (start recovery)
Unhealthy Confirmed	Greater than unhealthy threshold	Keep backend out of pool	First successful probe (start recovery process)
Recovery Monitoring	Varies during recovery	Keep out of pool but continue checking	Consecutive successes reach healthy threshold

The consecutive failure approach provides several important benefits for production load balancer stability:

False Positive Reduction: Temporary network hiccups, brief CPU spikes, or momentary resource contention won't immediately remove a healthy backend from service. The system tolerates transient issues while still detecting sustained problems.

Configurable Sensitivity: Different environments require different failure detection speeds. A development environment might use a threshold of 2 consecutive failures for quick feedback, while a production system might use 5 consecutive failures to prioritize stability over immediate response.

Gradual Response: The system doesn't immediately panic when a single probe fails. Instead, it continues monitoring and only takes action when multiple consecutive probes confirm a persistent problem.

> **Decision: Consecutive Failure Counting vs. Sliding Window**
>
> - **Context**: Need to balance quick failure detection with false positive avoidance
> - **Options Considered**:
>   1. Single failure triggers unhealthy state (immediate response)
>   2. Consecutive failure counting (current approach)
>   3. Sliding window percentage (e.g., 80% failures in last 10 probes)
> - **Decision**: Consecutive failure counting with configurable threshold
> - **Rationale**: Simpler to understand and configure than sliding windows, more stable than single-failure triggering, and maps well to common operational expectations
> - **Consequences**: May be slower to detect intermittent failures compared to sliding windows, but provides predictable behavior and lower false positive rates

The failure detection logic tracks multiple types of information for each backend to support both operational decisions and debugging:

```
HealthState Structure Components:
- Healthy: Current overall health status (boolean)
- ConsecutiveSuccesses: Count of consecutive successful probes
- ConsecutiveFailures: Count of consecutive failed probes
- LastCheckTime: Timestamp of most recent probe attempt
- LastSuccessTime: Timestamp of most recent successful probe
- LastFailureTime: Timestamp of most recent failed probe
- LastError: Error message from most recent failure
- TotalChecks: Lifetime count of all probe attempts
- TotalSuccesses: Lifetime count of successful probes
- TotalFailures: Lifetime count of failed probes
- StateChanges: Historical log of healthy/unhealthy transitions
```

The state transition algorithm follows a clear decision tree for each probe result:

1. **Receive probe result** with timestamp, success/failure status, and error details (if failure)
2. **Update timing information** by recording the probe timestamp and updating last success/failure times
3. **Increment counters** for both total checks and either total successes or total failures
4. **Process success path** (if probe succeeded): Reset consecutive failure count to zero, increment consecutive success count, and check if recovery conditions are met
5. **Process failure path** (if probe failed): Reset consecutive success count to zero, increment consecutive failure count, and record error details
6. **Evaluate state transition**: Compare consecutive failure count against unhealthy threshold to determine if backend should be marked unhealthy
7. **Execute state change** (if threshold crossed): Update healthy status, log state change with reason, and notify backend manager of status change
8. **Update statistics** including failure rate calculations and availability metrics for monitoring

Error classification during failure detection helps operators understand the types of problems affecting their backends:

Error Category	Detection Pattern	Typical Causes	Operational Response
Connection Refused	TCP connection fails immediately	Backend process stopped, port not listening	Restart backend service, check process status
Timeout	No response within configured time limit	Backend overloaded, network congestion	Check backend CPU/memory, investigate network
DNS Failure	Cannot resolve backend hostname	DNS server issues, incorrect configuration	Verify DNS configuration, check name resolution
HTTP Error	Connection succeeds but HTTP status indicates failure	Application-level problems, database issues	Check backend application logs, verify dependencies
Protocol Error	Invalid HTTP response format	Backend returning malformed responses	Investigate backend application code, check for corruption

Recovery Detection Logic

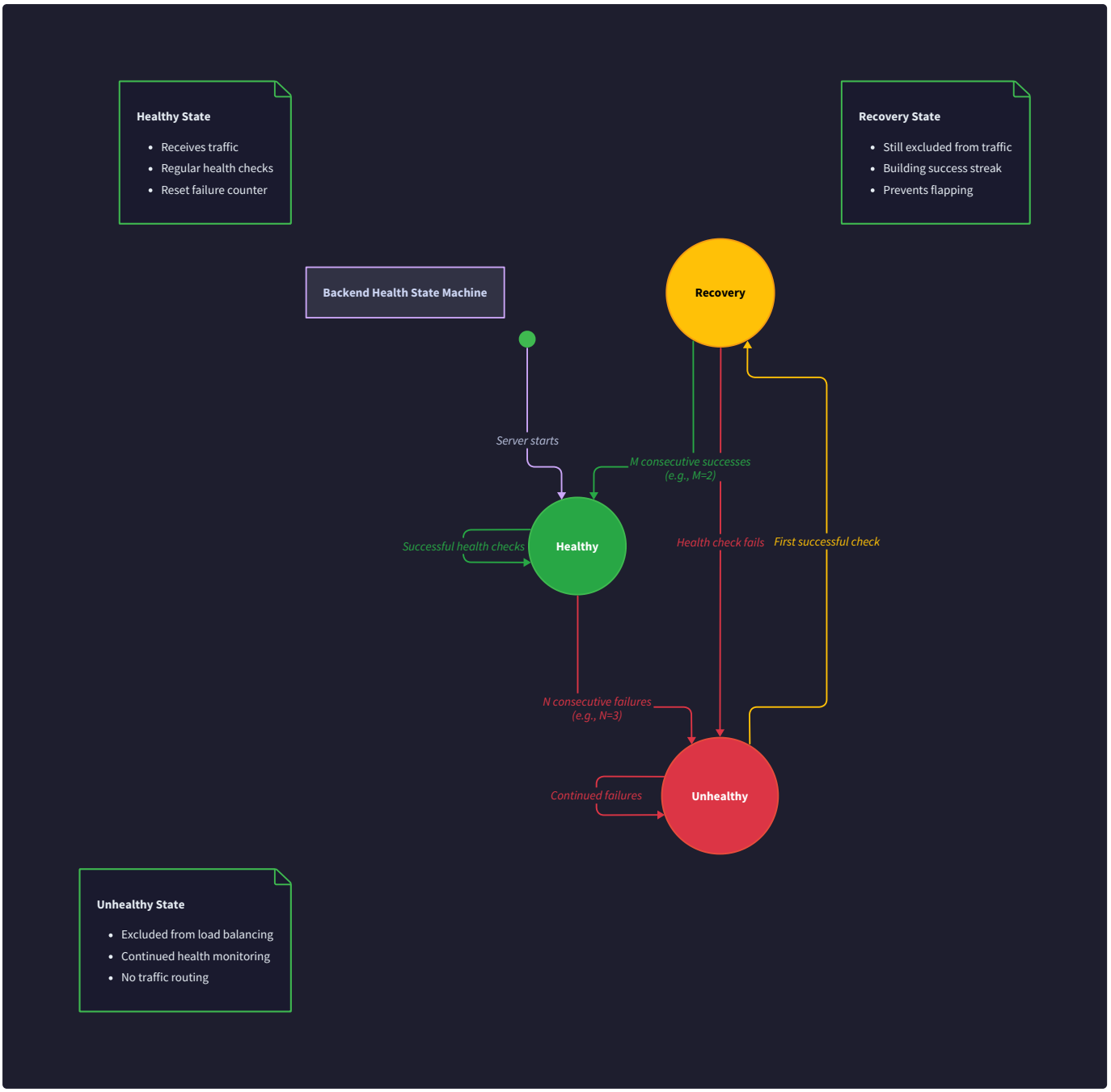
The **recovery detection logic** manages the process of bringing previously unhealthy backends back into the active pool once they demonstrate sustained recovery. This process mirrors the failure detection logic but operates in reverse, requiring multiple consecutive successful probes before restoring a backend to healthy status.

Recovery detection serves several critical functions in load balancer operation:

Preventing Flapping: Without requiring consecutive successes, a backend could rapidly oscillate between healthy and unhealthy states, causing request distribution instability and potentially overwhelming the recovering server with sudden traffic.

Verifying Sustained Recovery: A single successful probe might indicate temporary improvement rather than full recovery. Requiring multiple consecutive successes ensures the backend can handle sustained operation.

Gradual Traffic Restoration: The recovery process provides a natural ramp-up period where the backend proves its stability before receiving full traffic allocation.



The recovery process maintains its own configurable threshold, typically called the **healthy threshold**, which determines how many consecutive successful probes are required to restore a backend to healthy status. This threshold can be different from (and often smaller than) the unhealthy threshold to create asymmetric behavior.

Recovery Phase	Consecutive Successes	Backend Status	Traffic Handling
Initial Failure	0	Unhealthy	Excluded from all traffic
Recovery Attempt	1 to (healthy threshold - 1)	Still Unhealthy	Excluded from traffic, monitoring progress
Recovery Success	Equals healthy threshold	Transitioning to Healthy	About to rejoin active pool
Healthy Restored	Maintained above threshold	Healthy	Receiving normal traffic distribution

> **Decision: Asymmetric Thresholds for Failure vs Recovery**
>
> - **Context**: Need to balance quick recovery with stability during backend restoration
> - **Options Considered**:
>   1. Use same threshold for both failure detection and recovery (symmetric)
>   2. Use lower threshold for recovery than failure detection (favor quick recovery)
>   3. Use higher threshold for recovery than failure detection (favor stability)
> - **Decision**: Configurable thresholds with recommended asymmetric defaults (failure: 3, recovery: 2)
> - **Rationale**: Backends should be quickly restored once they show signs of recovery, but not so quickly that transient improvements cause flapping
> - **Consequences**: Slightly more complex configuration but better operational characteristics in most environments

The recovery detection algorithm processes successful probe results when a backend is currently in the unhealthy state:

1. **Verify current state** by confirming the backend is currently marked as unhealthy (recovery only applies to unhealthy backends)
2. **Process successful probe** by resetting the consecutive failure count and incrementing the consecutive success count
3. **Check recovery threshold** by comparing consecutive successes against the configured healthy threshold
4. **Execute recovery transition** (if threshold met) by marking backend as healthy, logging the state change, and notifying the backend manager
5. **Reset recovery counters** by clearing consecutive success count and preparing for normal healthy monitoring
6. **Log recovery event** with timestamp, number of probes required, and duration of unhealthy period for operational analysis

The system tracks detailed recovery statistics to support operational analysis and threshold tuning:

```
Recovery Tracking Information:
- RecoveryStartTime: When first successful probe occurred after unhealthy state
- RecoveryProbeCount: Number of successful probes during recovery process
- UnhealthyDuration: Total time spent in unhealthy state
- RecoveryDuration: Time from first success to healthy restoration
- RecoveryAttempts: Number of times recovery was attempted (may fail and restart)
- LastRecoveryTime: Timestamp when backend was last restored to healthy status
```

Recovery can be interrupted if a probe fails during the recovery process. When this happens, the consecutive success count resets to zero, and the backend remains unhealthy. This interrupted recovery gets logged as a separate event to help identify backends with intermittent problems.

Partial Recovery Scenarios occur when a backend begins recovering but then fails again before reaching the healthy threshold:

1. **Recovery Interruption**: A probe fails after some successful probes during recovery, resetting progress
2. **Recovery Restart**: The system begins counting consecutive successes again from zero
3. **Pattern Analysis**: Multiple interrupted recoveries might indicate underlying instability requiring investigation

The recovery logic also implements **gradual traffic restoration** strategies to prevent overwhelming a recently recovered backend:

Immediate Full Restoration: The backend immediately receives its full share of traffic distribution once marked healthy. This approach is simple but risks overwhelming a backend that's still stabilizing.

Warming Period: Some implementations include a warming period where a newly recovered backend receives reduced traffic initially, gradually increasing to full allocation. This isn't implemented in the basic version but represents a common production enhancement.

> During recovery detection, continue performing health checks at the same interval as for healthy backends. Don't increase the frequency just because a backend is recovering — this could overwhelm a backend that's struggling to stabilize.

Health Check Configuration

The **health check configuration** provides comprehensive control over all aspects of the health monitoring system, enabling operators to tune behavior for different environments, backend characteristics, and availability requirements. Proper configuration balances quick failure detection with system stability and resource efficiency.

The configuration system organizes health check parameters into logical groups that correspond to different aspects of the monitoring process:

Configuration Category	Purpose	Key Parameters
Probe Behavior	Controls how health checks are performed	Interval, timeout, HTTP method, endpoint path
Failure Detection	Determines when backends are marked unhealthy	Unhealthy threshold, failure classification rules
Recovery Detection	Controls when backends are restored to healthy status	Healthy threshold, recovery validation rules
Response Validation	Defines what constitutes a successful probe	Expected status codes, body patterns, headers
Resource Management	Manages health checking overhead	Concurrent check limits, client pool size

The **probe timing configuration** requires careful consideration of the trade-offs between detection speed and system overhead:

```
HealthCheckConfig Timing Parameters:
- Interval: Time between health checks (default: 30s, range: 5s-300s)
- Timeout: Maximum time to wait for probe response (default: 5s, range: 1s-30s)
- Jitter: Random variation in interval to prevent synchronization (default: 10%)
- InitialDelay: Wait time before starting health checks for new backends (default: 0s)
- MaxCheckDuration: Overall timeout including retries and processing (default: 2x timeout)
```

Decision: Health Check Interval Selection

- **Context**: Need to balance quick failure detection with resource overhead and backend load
- **Options Considered**:
  1. Fixed short interval (5-10 seconds) for fast detection
  2. Fixed long interval (60+ seconds) for low overhead
  3. Adaptive interval based on backend health history
  4. Configurable interval with reasonable defaults (current approach)
- **Decision**: Configurable interval with 30-second default and 5-300 second range
- **Rationale**: Different environments have different requirements; development needs fast feedback while production prioritizes stability
- **Consequences**: Requires operators to understand interval trade-offs, but provides flexibility for diverse environments

The **endpoint configuration** specifies how and where health checks are performed:

```
Health Check Endpoint Configuration:
- Path: HTTP path for health checks (default: "/health")
- Method: HTTP method to use (default: "GET", options: "GET", "HEAD", "POST")
- Headers: Custom headers to include in health check requests
- UserAgent: User-Agent header value (default: "LoadBalancer-HealthCheck/1.0")
- Host: Host header override (optional, uses backend hostname by default)
- Scheme: Protocol scheme override (default: "http", option: "https")
```

Response validation configuration determines what constitutes a successful health check beyond basic connectivity:

Validation Type	Configuration	Success Criteria	Use Cases
Status Code	ExpectedStatus (default: 200)	HTTP response matches expected code	Basic application responsiveness
Response Time	MaxResponseTime (optional)	Response received within time limit	Performance-based health assessment
Body Content	ExpectedBodyPattern (optional regex)	Response body matches pattern	Application-specific readiness checks
Header Presence	RequiredHeaders (list)	Specified headers present in response	Custom application health indicators
Content Type	ExpectedContentType (optional)	Response has expected content type	Ensure proper application response format

Threshold configuration provides fine-grained control over failure and recovery detection sensitivity:

```
Threshold Configuration Parameters:
- UnhealthyThreshold: Consecutive failures required to mark backend unhealthy (default: 3)
- HealthyThreshold: Consecutive successes required to restore backend to healthy (default: 2)
- MaxConsecutiveFailures: Cap on failure count for statistics (default: 100)
- FailureRateThreshold: Percentage failure rate over time window (optional)
- MinCheckCount: Minimum checks before applying failure rate threshold (default: 10)
```

The configuration system supports **environment-specific profiles** that provide appropriate defaults for different deployment scenarios:

Environment Profile	Interval	Timeout	Unhealthy Threshold	Healthy Threshold	Use Case
Development	10s	3s	2	1	Fast feedback, frequent code changes
Staging	15s	5s	3	2	Balance between speed and stability
Production	30s	5s	3	2	Stability over speed, avoid false positives
High-Frequency	5s	2s	5	3	Critical services requiring immediate detection
Low-Overhead	60s	10s	2	2	Resource-constrained environments

Dynamic configuration updates allow operators to modify health check behavior without restarting the load balancer:

1. **Configuration Validation**: New configuration is validated for parameter ranges, consistency, and required fields
2. **Gradual Application**: Changes are applied to health checkers as their current check cycles complete
3. **Backward Compatibility**: Running health checks complete with old parameters before adopting new ones
4. **Rollback Capability**: Invalid configurations are rejected and previous settings maintained
5. **Change Logging**: All configuration changes are logged with timestamps and operator identification

> When changing health check intervals, don't immediately apply the new interval to all backends. Instead, let current check cycles complete naturally to avoid creating synchronized load spikes or missing critical health transitions.

Architecture Decisions

The health checking system incorporates several key architecture decisions that significantly impact its behavior, performance, and operational characteristics. These decisions reflect trade-offs between competing requirements such as detection speed, resource efficiency, false positive rates, and implementation complexity.

> **Decision: Active vs Passive Health Checking**
>
> - **Context**: Need to detect backend failures before they impact live traffic
> - **Options Considered**:
>   1. Passive health checking (detect failures from live request errors)
>   2. Active health checking with periodic probes (current approach)
>   3. Hybrid approach combining both active and passive detection
> - **Decision**: Active health checking with periodic HTTP probes
> - **Rationale**: Active checking detects problems before they impact users, provides consistent monitoring regardless of traffic patterns, and enables predictable failure detection timing
> - **Consequences**: Higher resource overhead due to probe traffic, but better user experience and more reliable failure detection

The **concurrent vs sequential checking** decision fundamentally affects both performance characteristics and resource utilization patterns:

Approach	Pros	Cons	Resource Impact
Concurrent Checking	Faster overall check completion, independent timing per backend	Higher instantaneous resource usage, potential thundering herd	High CPU/network burst, more goroutines
Sequential Checking	Lower resource peaks, simpler resource management	Slower failure detection for later backends in sequence	Steady resource usage, fewer goroutines
Batched Checking	Balanced resource usage, tunable concurrency	More complex implementation, requires batch size tuning	Configurable resource usage patterns

> **Decision: Concurrent Health Checking with Semaphore Limiting**
>
> - **Context**: Need to balance fast failure detection with resource consumption limits
> - **Options Considered**:
>   1. Fully sequential checking (check backends one at a time)
>   2. Fully concurrent checking (check all backends simultaneously)
>   3. Concurrent with semaphore limiting (current approach)
> - **Decision**: Concurrent checking with configurable semaphore limit (default: 10 concurrent checks)
> - **Rationale**: Provides fast detection when needed while preventing resource exhaustion with large backend pools
> - **Consequences**: Requires semaphore management but provides tunable resource usage and good detection speed

The **health check isolation** decision determines how health checking interacts with the main request processing pipeline:

> **Decision: Separate HTTP Client for Health Checks**
>
> - **Context**: Health checks have different requirements than request forwarding (timeouts, connection pooling, retry behavior)
> - **Options Considered**:
>   1. Share HTTP client with request forwarding
>   2. Separate HTTP client dedicated to health checks (current approach)
>   3. Custom TCP-level health checking without HTTP
> - **Decision**: Dedicated HTTP client with health-check-specific configuration
> - **Rationale**: Health checks need short timeouts and different connection management, sharing could cause interference
> - **Consequences**: Slightly higher memory usage but better isolation and more appropriate timeout/retry behavior

State management architecture affects both performance and correctness under concurrent access:

The health checking system must handle concurrent access from multiple goroutines: the health checker updating states, the request router reading healthy backend lists, and administrative interfaces querying statistics. This requires careful synchronization to prevent data races while maintaining good performance.

```
State Management Approach:
- HealthState embedded in Backend struct with atomic fields for counters
- RWMutex protecting state transitions and complex updates
- Atomic operations for frequently accessed counters (consecutive failures/successes)
- Copy-on-read for expensive operations like generating healthy backend lists
- Channel-based state change notifications to avoid polling
```

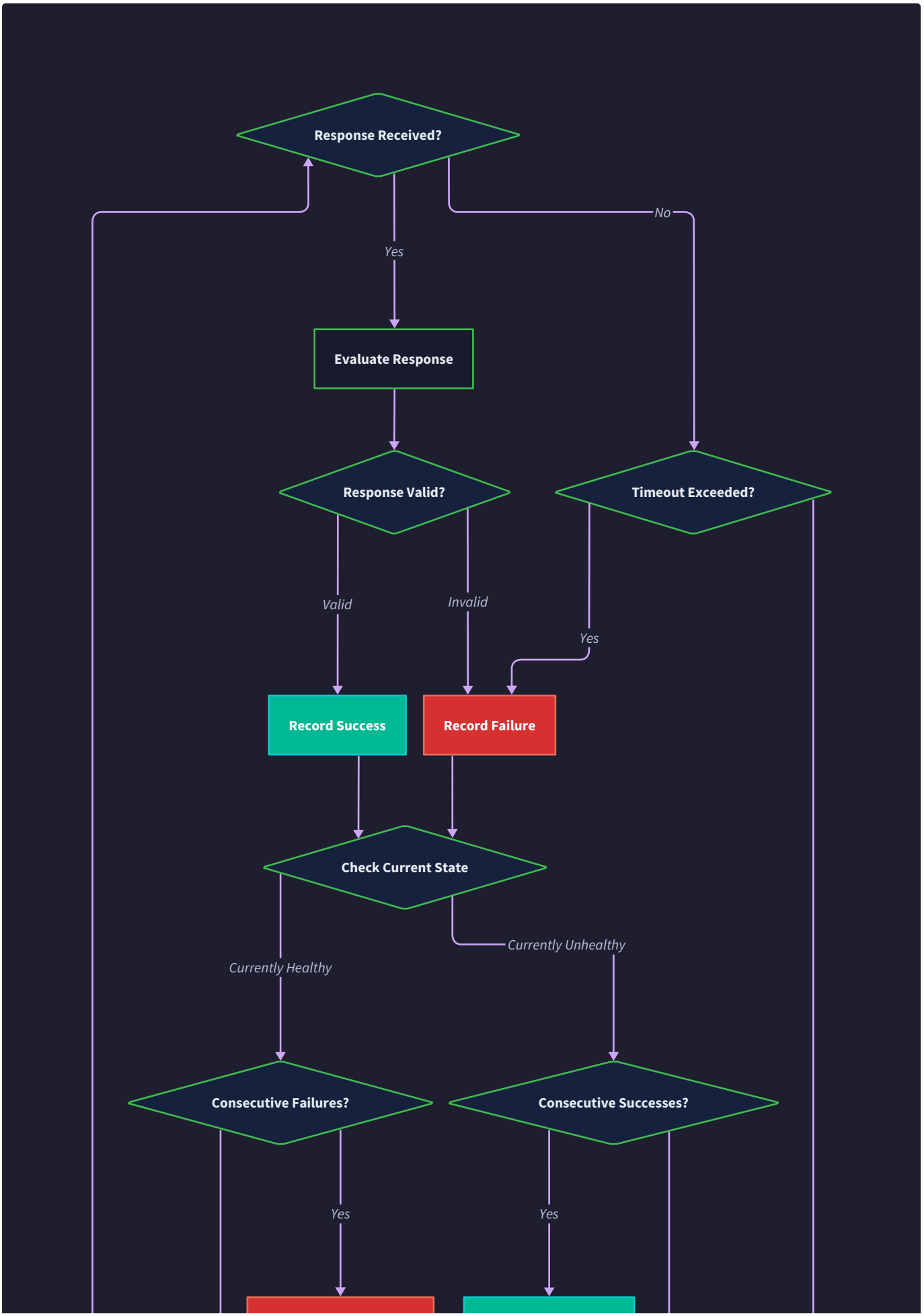
> **Decision: Embedded vs Separate Health State Storage**
>
> - **Context**: Need to associate health state with backend instances while supporting efficient updates
> - **Options Considered**:
>   1. Separate health state map with backend ID keys
>   2. Embed HealthState directly in Backend struct (current approach)
>   3. Health state stored in external database/cache
> - **Decision**: Embed HealthState as pointer field in Backend struct
> - **Rationale**: Ensures state stays associated with backend, simplifies memory management, and enables atomic pointer swapping for state updates
> - **Consequences**: Slightly larger Backend structs but better data locality and simpler state management

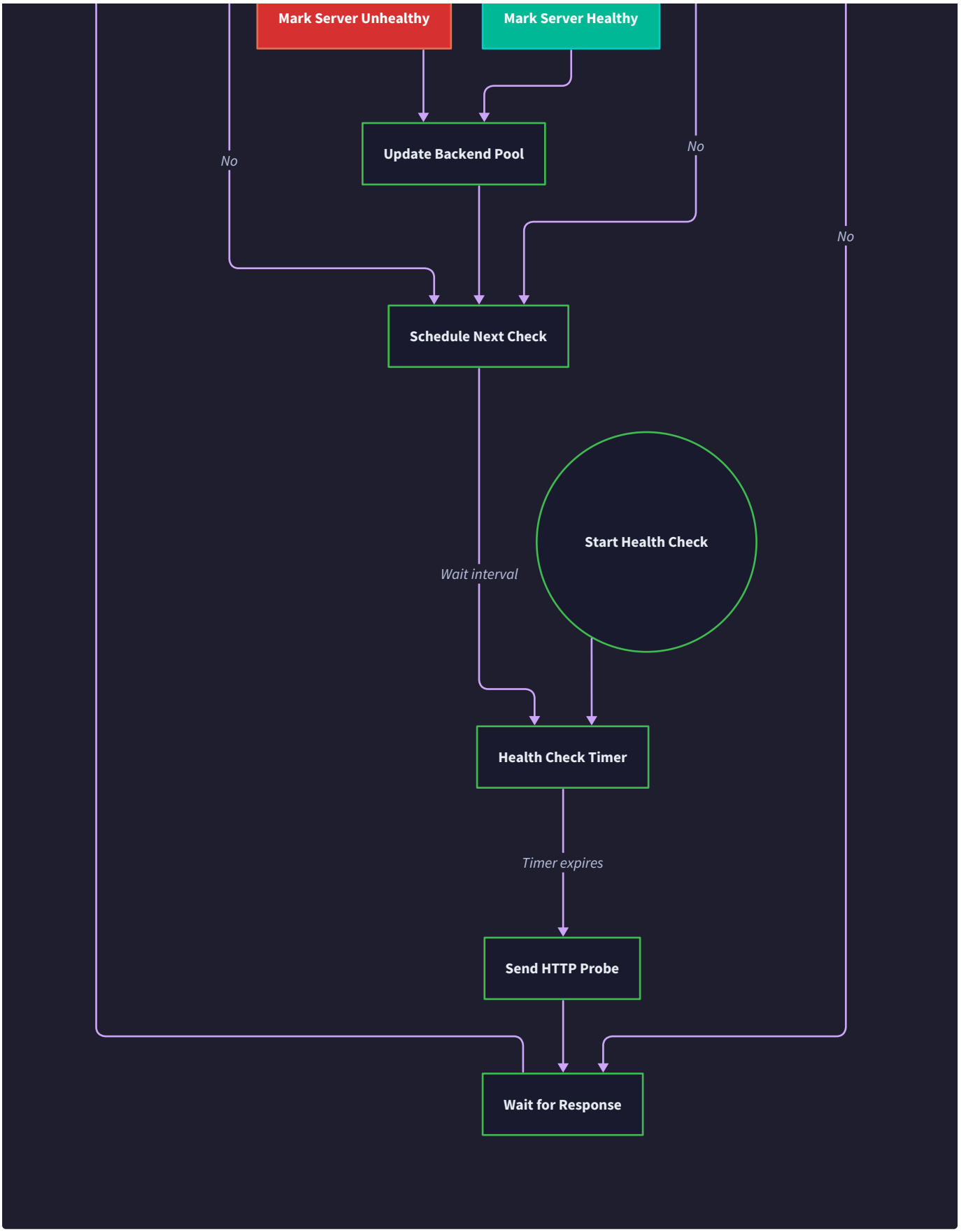
The **error handling and logging architecture** determines how health check problems are diagnosed and monitored:

Error Handling Aspect	Design Choice	Rationale
Error Categorization	Structured error types (ConnectionError, TimeoutError, HTTPError)	Enables appropriate responses and better debugging
Error Persistence	Store last error message and timestamp in HealthState	Provides debugging information without expensive logging
Logging Strategy	Log state transitions and periodic summaries, not individual checks	Reduces log noise while preserving important events
Metrics Integration	Expose health check metrics via standard interfaces	Enables monitoring and alerting on health check performance
Circuit Breaking	Temporary health check suspension after repeated infrastructure failures	Prevents wasting resources on systemically unreachable backends

Recovery coordination affects how quickly backends rejoin the active pool and whether this causes stability issues:

> **Decision: Independent Recovery vs Coordinated Restoration**
>
> - **Context**: Multiple backends might recover simultaneously, potentially overwhelming the system
> - **Options Considered**:
>   1. Independent recovery (each backend restored as soon as it meets threshold)
>   2. Coordinated recovery (limit number of backends restored simultaneously)
>   3. Gradual traffic restoration (restored backends receive reduced traffic initially)
> - **Decision**: Independent recovery with monitoring for mass recovery events
> - **Rationale**: Simpler implementation, faster restoration of capacity, with logging to detect and address mass recovery scenarios
> - **Consequences**: Potential for traffic spikes when multiple backends recover simultaneously, but maintains simplicity and fast recovery


Response Received?
No
Yes
Evaluate Response
Response Valid?
Timeout Exceeded?
Valid
Invalid
Yes
Record Success
Record Failure
Check Current State
Currently Unhealthy
Currently Healthy
Consecutive Failures?
Consecutive Successes?
Yes
Yes



Common Pitfalls

The health checking system presents several common implementation and configuration pitfalls that can significantly impact load balancer reliability and performance. Understanding these pitfalls helps avoid subtle bugs and operational issues that often emerge only under specific conditions.

⚠️ Pitfall: Health Check Overwhelming Backends

A frequently encountered problem occurs when health check traffic itself causes backend performance degradation or failure. This happens when health check intervals are too aggressive, health check endpoints are too expensive, or the health checking system doesn't account for backend capacity limits.

Why This Occurs: Health checks run continuously regardless of backend load, and poorly configured intervals can generate significant additional traffic. For example, checking 100 backends every 5 seconds with a 30-second timeout creates sustained load that might overwhelm already-stressed backends.

How to Detect: Monitor backend server logs for high health check request volume, observe correlation between health check frequency and backend response times, or notice that backends fail health checks during high traffic periods but recover during low traffic periods.

Prevention and Fix: Configure reasonable check intervals (30+ seconds for production), implement lightweight health check endpoints that don't perform expensive operations, use jitter to distribute check timing, and monitor health check traffic as a percentage of total backend load.

⚠️ **Pitfall: Race Conditions in Health State Updates**

Concurrent access to health state information can create race conditions where multiple goroutines read and modify state simultaneously, leading to inconsistent state transitions or lost updates.

Why This Occurs: Health check updates, request routing decisions, and administrative queries all access backend health state concurrently. Without proper synchronization, the health checker might mark a backend unhealthy while the request router is simultaneously reading it as healthy.

How to Detect: Inconsistent health state reports between different system components, backends that seem to flicker between healthy and unhealthy states, or occasional panics related to concurrent map access.

Prevention and Fix: Use appropriate synchronization primitives (RWMutex for state transitions, atomic operations for counters), implement copy-on-read for expensive operations, and ensure state changes are atomic from the perspective of other system components.

⚠️ Pitfall: Thundering Herd During Recovery

When multiple backends recover simultaneously after a shared outage (network partition, dependency failure), they can overwhelm the system with sudden traffic spikes, potentially causing a cascade failure.

Why This Occurs: All backends fail health checks simultaneously during an infrastructure outage, then all begin recovery at roughly the same time when the outage resolves. Without coordination, they all rejoin the active pool simultaneously, receiving full traffic allocation immediately.

How to Detect: Sudden traffic spikes coinciding with backend recovery events, performance degradation immediately after outage resolution, or repeated cycles of recovery and failure.

Prevention and Fix: Implement jitter in recovery timing, consider gradual traffic restoration for recently recovered backends, monitor for mass recovery events and implement rate limiting when detected, or use longer healthy thresholds to spread recovery timing.

⚠️ Pitfall: Health Check Endpoint Doesn't Reflect Application Health

Health check endpoints that only verify basic connectivity (returning 200 OK) without checking application dependencies can create false positives where backends appear healthy but can't actually serve requests successfully.

Why This Occurs: Simple health checks might only verify that the HTTP server is running without checking database connectivity, external service availability, or resource availability (memory, disk space).

How to Detect: Backends pass health checks but return errors for actual requests, user-facing errors that don't correlate with health check failures, or performance issues that aren't reflected in health check results.

Prevention and Fix: Design health check endpoints to verify critical dependencies, implement multiple levels of health checking (shallow for basic connectivity, deep for application readiness), and consider separate endpoints for different health aspects.

⚠️ Pitfall: Inappropriate Threshold Configuration

Incorrectly configured failure and recovery thresholds can cause either excessive sensitivity (frequent false positives) or insufficient sensitivity (slow failure detection), both of which degrade system reliability.

Why This Occurs: Thresholds are often set based on intuition rather than measurement, different environments require different sensitivity levels, and threshold requirements change as system scale and reliability requirements evolve.

How to Detect: Frequent backend state flapping indicating overly sensitive thresholds, delayed failure detection causing user impact indicating insufficient sensitivity, or correlation between network conditions and health state changes.

Prevention and Fix: Start with conservative thresholds and tune based on observed behavior, implement different threshold profiles for different environments, monitor threshold effectiveness through failure detection time and false positive rates, and document threshold rationale for future tuning.

⚠️ Pitfall: Memory Leaks in Health State History

Accumulating health check history, state changes, and error messages without proper cleanup can cause memory usage to grow unbounded over time, particularly in systems with many backends or long uptimes.

Why This Occurs: Health state structures accumulate historical data for debugging and analysis, but without cleanup policies this data grows indefinitely. State change logs, error message histories, and statistical counters can consume significant memory.

How to Detect: Gradually increasing memory usage over time, memory usage that correlates with system uptime rather than current load, or large amounts of memory attributed to health checking structures.

Prevention and Fix: Implement retention policies for historical data (keep last N state changes, limit error message storage), use ring buffers for historical data instead of unlimited slices, periodically clean up old statistical data, and monitor memory usage of health checking components.

⚠️ Pitfall: Health Check Timeout Longer Than Request Timeout

Configuring health check timeouts that are longer than request forwarding timeouts creates a situation where backends might appear healthy to the health checker but still fail user requests due to slowness.

Why This Occurs: Health check and request forwarding timeouts are often configured independently, and it's counterintuitive that health checks should have shorter timeouts than actual requests.

How to Detect: Backends that pass health checks but frequently time out on user requests, user-reported slow response times that don't correlate with health check failures.

Prevention and Fix: Set health check timeouts to be 50-80% of request timeouts, consider separate performance-based health checking that measures response times, and monitor correlation between health check performance and request success rates.

Implementation Guidance

The health checking system requires careful implementation to handle concurrent operations, manage resource usage, and provide reliable failure detection. This guidance provides concrete implementation patterns for building a robust health monitoring system.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Client	`net/http` with custom `Transport`	`fasthttp` for high-performance checking
Timing	`time.Ticker` for periodic scheduling	`time.AfterFunc` for more precise timing control
Concurrency	`sync.WaitGroup` for check coordination	Worker pool with buffered channels
State Storage	In-memory maps with `sync.RWMutex`	Redis for distributed health state
Configuration	JSON/YAML file loading	etcd/Consul for dynamic configuration
Logging	`log/slog` structured logging	`logrus` or `zap` for high-performance logging

Recommended File Structure

```
internal/health/
  health_checker.go          ← main health checking logic
  health_checker_test.go     ← comprehensive test suite
  probe.go                   ← HTTP probe implementation
  probe_test.go              ← probe unit tests
  state.go                   ← health state management
  state_test.go              ← state transition tests
  config.go                  ← health check configuration
  config_test.go             ← configuration validation tests
  metrics.go                 ← health check metrics collection

cmd/health-check-test/
  main.go                    ← standalone health check testing tool
```

Infrastructure Starter Code

Health Check Configuration (complete implementation):

```go
// config.go

package health


import (

    "encoding/json"

    "fmt"

    "os"

    "time"

)


// HealthCheckConfig defines all health checking parameters

type HealthCheckConfig struct {

    Enabled             bool          `json:"enabled"`

    Interval            time.Duration `json:"interval"`

    Timeout             time.Duration `json:"timeout"`

    HealthyThreshold    int           `json:"healthy_threshold"`

    UnhealthyThreshold  int           `json:"unhealthy_threshold"`

    Path                string        `json:"path"`

    ExpectedStatus      int           `json:"expected_status"`

    UserAgent           string        `json:"user_agent"`

    MaxConcurrentChecks int           `json:"max_concurrent_checks"`

    Jitter              bool          `json:"jitter"`

}


// DefaultHealthCheckConfig returns sensible defaults

func DefaultHealthCheckConfig() HealthCheckConfig {

    return HealthCheckConfig{

        Enabled:             true,

        Interval:            30 * time.Second,

        Timeout:             5 * time.Second,

        HealthyThreshold:    2,

        UnhealthyThreshold:  3,

        Path:                "/health",

        ExpectedStatus:      200,

        UserAgent:           "LoadBalancer-HealthCheck/1.0",

        MaxConcurrentChecks: 10,

        Jitter:              true,

    }

}


// Validate checks configuration parameters for consistency and valid ranges

func (hcc HealthCheckConfig) Validate() error {
```

```
    if hcc.Interval < time.Second {
        return fmt.Errorf("interval must be at least 1 second, got %v", hcc.Interval)
    }
    if hcc.Timeout >= hcc.Interval {
        return fmt.Errorf("timeout (%v) must be less than interval (%v)", hcc.Timeout, hcc.Interval)
    }
    if hcc.HealthyThreshold < 1 {
        return fmt.Errorf("healthy_threshold must be at least 1, got %d", hcc.HealthyThreshold)
    }
    if hcc.UnhealthyThreshold < 1 {
        return fmt.Errorf("unhealthy_threshold must be at least 1, got %d", hcc.UnhealthyThreshold)
    }
    if hcc.ExpectedStatus < 100 || hcc.ExpectedStatus > 599 {
        return fmt.Errorf("expected_status must be valid HTTP status code (100-599), got %d", hcc.ExpectedStatus)
    }
    return nil
}

// LoadHealthCheckConfig loads configuration from JSON file
func LoadHealthCheckConfig(filename string) (HealthCheckConfig, error) {
    config := DefaultHealthCheckConfig()

    data, err := os.ReadFile(filename)
    if err != nil {
        return config, fmt.Errorf("reading config file: %w", err)
    }

    if err := json.Unmarshal(data, &config); err != nil {
        return config, fmt.Errorf("parsing config JSON: %w", err)
    }

    if err := config.Validate(); err != nil {
        return config, fmt.Errorf("validating config: %w", err)
    }

    return config, nil
}
```

HTTP Probe Implementation (complete implementation):

```go
// probe.go

package health


import (

    "context"

    "fmt"

    "net"

    "net/http"

    "net/url"

    "strings"

    "time"

)


// ProbeResult contains the outcome of a health check probe

type ProbeResult struct {

    Backend     *Backend

    Healthy     bool

    StatusCode  int

    Duration    time.Duration

    Error       error

    Timestamp   time.Time

}


// HTTPProber performs HTTP-based health checks

type HTTPProber struct {

    client *http.Client

    config HealthCheckConfig

}


// NewHTTPProber creates a prober with appropriate HTTP client configuration

func NewHTTPProber(config HealthCheckConfig) *HTTPProber {

    transport := &http.Transport{

        DialContext: (&net.Dialer{

            Timeout:   config.Timeout / 2, // TCP connection timeout

            KeepAlive: 30 * time.Second,

        }).DialContext,

        TLSHandshakeTimeout:   config.Timeout / 2,

        ResponseHeaderTimeout: config.Timeout,

        IdleConnTimeout:       90 * time.Second,

        MaxIdleConnsPerHost:   2, // Limit connections for health checks

    }
```

```
    client := &http.Client{
        Transport: transport,
        Timeout:   config.Timeout,
        CheckRedirect: func(req *http.Request, via []*http.Request) error {
            return http.ErrUseLastResponse // Don't follow redirects
        },
    }

    return &HTTPProber{
        client: client,
        config: config,
    }
}

// ProbeBackend performs a single health check probe against a backend
func (p *HTTPProber) ProbeBackend(ctx context.Context, backend *Backend) ProbeResult {
    startTime := time.Now()
    result := ProbeResult{
        Backend:   backend,
        Timestamp: startTime,
    }

    // Build health check URL
    healthURL := &url.URL{
        Scheme: backend.URL.Scheme,
        Host:   backend.URL.Host,
        Path:   p.config.Path,
    }

    // Create HTTP request
    req, err := http.NewRequestWithContext(ctx, "GET", healthURL.String(), nil)
    if err != nil {
        result.Error = fmt.Errorf("creating request: %w", err)
        return result
    }

    // Set health check headers
    req.Header.Set("User-Agent", p.config.UserAgent)
    req.Header.Set("Connection", "close") // Don't reuse connections

    // Perform the request
```

```
    resp, err := p.client.Do(req)
    result.Duration = time.Since(startTime)

    if err != nil {
        result.Error = p.categorizeError(err)
        return result
    }

    defer resp.Body.Close()
    result.StatusCode = resp.StatusCode

    // Evaluate response
    if resp.StatusCode == p.config.ExpectedStatus {
        result.Healthy = true
    } else {
        result.Error = fmt.Errorf("unexpected status code: got %d, expected %d",
            resp.StatusCode, p.config.ExpectedStatus)
    }

    return result
}

// categorizeError classifies errors for better debugging and handling
func (p *HTTPProber) categorizeError(err error) error {
    if err == nil {
        return nil
    }

    errStr := err.Error()
    switch {
    case strings.Contains(errStr, "connection refused"):
        return fmt.Errorf("connection refused: backend not accepting connections")
    case strings.Contains(errStr, "timeout"):
        return fmt.Errorf("timeout: backend not responding within %v", p.config.Timeout)
    case strings.Contains(errStr, "no such host"):
        return fmt.Errorf("DNS error: cannot resolve backend hostname")
    case strings.Contains(errStr, "network unreachable"):
        return fmt.Errorf("network error: backend unreachable")
    default:
        return fmt.Errorf("probe error: %w", err)
    }
```

```
}
```

Core Logic Skeleton Code

Main Health Checker (core implementation to complete):

```go
// health_checker.go

package health


import (

    "context"

    "math/rand"

    "sync"

    "time"

)


// HealthChecker manages health monitoring for all backends

type HealthChecker struct {

    config      HealthCheckConfig

    prober      *HTTPProber

    backends    map[string]*Backend

    backendsMux sync.RWMutex

    stopChan    chan struct{}

    semaphore   chan struct{} // Limit concurrent checks

    wg          sync.WaitGroup

}


// NewHealthChecker creates a new health checker instance

func NewHealthChecker(config HealthCheckConfig) *HealthChecker {

    return &HealthChecker{

        config:    config,

        prober:    NewHTTPProber(config),

        backends:  make(map[string]*Backend),

        stopChan:  make(chan struct{}),

        semaphore: make(chan struct{}, config.MaxConcurrentChecks),

    }

}


// AddBackend registers a backend for health checking

func (hc *HealthChecker) AddBackend(backend *Backend) error {

    // TODO 1: Acquire write lock on backendsMux

    // TODO 2: Add backend to backends map using backend.ID as key

    // TODO 3: Release lock

    // TODO 4: If health checking is enabled and running, start monitoring this backend

    // TODO 5: Log backend addition for debugging

    return nil

}


// RemoveBackend stops health checking for a backend
```

```go
func (hc *HealthChecker) RemoveBackend(backendID string) error {
    // TODO 1: Acquire write lock on backendsMux
    // TODO 2: Remove backend from backends map
    // TODO 3: Release lock
    // TODO 4: Log backend removal
    // Hint: Existing health check goroutines will naturally stop when they next try to check this backend
    return nil
}

// Start begins health checking for all registered backends
func (hc *HealthChecker) Start(ctx context.Context) error {
    // TODO 1: Check if health checking is enabled in config, return early if disabled
    // TODO 2: Start a goroutine for each backend that performs periodic health checks
    // TODO 3: Each goroutine should use a ticker with interval + jitter for timing
    // TODO 4: Handle context cancellation and stopChan to gracefully shut down
    // TODO 5: Use semaphore to limit concurrent checks to config.MaxConcurrentChecks
    // TODO 6: Log health checker startup
    // Hint: Use hc.wg.Add(1) before starting each goroutine, defer hc.wg.Done() in goroutine
    return nil
}

// Stop gracefully shuts down health checking
func (hc *HealthChecker) Stop() {
    // TODO 1: Close stopChan to signal all health check goroutines to stop
    // TODO 2: Wait for all goroutines to complete using hc.wg.Wait()
    // TODO 3: Log health checker shutdown
}

// performHealthCheck executes a health check for a specific backend
func (hc *HealthChecker) performHealthCheck(ctx context.Context, backend *Backend) {
    // TODO 1: Acquire semaphore slot (send to hc.semaphore channel)
    // TODO 2: Defer releasing semaphore slot (receive from hc.semaphore channel)
    // TODO 3: Use hc.prober.ProbeBackend to perform the actual health check
    // TODO 4: Call hc.updateBackendHealth with the probe result
    // TODO 5: Log health check result (success/failure with error details)
    // Hint: Use select with ctx.Done() to handle cancellation during semaphore acquisition
}

// updateBackendHealth processes a probe result and updates backend health state
func (hc *HealthChecker) updateBackendHealth(result ProbeResult) {
    // TODO 1: Get current health state from result.Backend
    // TODO 2: If result.Healthy is true, call RecordSuccess on health state
```

```
    // TODO 3: If result.Healthy is false, call RecordFailure with error details
    // TODO 4: Check if health state changed (healthy -> unhealthy or vice versa)
    // TODO 5: If state changed, log the transition with reason and timing
    // TODO 6: Update any metrics or monitoring counters
    // Hint: Use atomic operations or appropriate locking when updating health state
}

// GetHealthyBackends returns a list of all currently healthy backends
func (hc *HealthChecker) GetHealthyBackends() []*Backend {
    // TODO 1: Acquire read lock on backendsMux
    // TODO 2: Iterate through all backends in backends map
    // TODO 3: Check each backend's health state using IsHealthy()
    // TODO 4: Add healthy backends to result slice
    // TODO 5: Release lock and return result slice
    // Hint: Create a copy of the slice to avoid holding the lock during caller's use
    return nil
}

// calculateCheckInterval returns the interval with optional jitter
func (hc *HealthChecker) calculateCheckInterval() time.Duration {
    interval := hc.config.Interval
    if hc.config.Jitter {
        // TODO 1: Add random jitter of ±10% to the base interval
        // TODO 2: Use rand.Float64() to generate random factor between 0.9 and 1.1
        // TODO 3: Apply jitter to avoid synchronized checking across all backends
        // Hint: jitter := time.Duration(float64(interval) * (0.9 + rand.Float64()*0.2))
    }
    return interval
}
```

Health State Management (core implementation to complete):

```go
// state.go

package health


import (

    "sync"

    "time"

)


// HealthState tracks the health status and history for a backend

type HealthState struct {

    Healthy              bool                 `json:"healthy"`

    ConsecutiveSuccesses  int                 `json:"consecutive_successes"`

    ConsecutiveFailures   int                 `json:"consecutive_failures"`

    LastCheckTime         time.Time           `json:"last_check_time"`

    LastSuccessTime       time.Time           `json:"last_success_time"`

    LastFailureTime       time.Time           `json:"last_failure_time"`

    LastError             string              `json:"last_error"`

    TotalChecks          int64               `json:"total_checks"`

    TotalSsuccesses       int64               `json:"total_successes"`

    TotalFailures        int64               `json:"total_failures"`

    StateChanges         []HealthStateChange  `json:"state_changes"`

    mutex                sync.RWMutex         `json:"-"`

}


// HealthStateChange records a transition between healthy and unhealthy states

type HealthStateChange struct {

    Timestamp     time.Time `json:"timestamp"`

    PreviousState bool      `json:"previous_state"`

    NewState      bool      `json:"new_state"`

    Reason        string    `json:"reason"`

    CheckCount    int       `json:"check_count"`

}


// NewHealthState creates a new health state starting in healthy condition

func NewHealthState() *HealthState {

    return &HealthState{

        Healthy:       true, // Start optimistically

        StateChanges:  make([]HealthStateChange, 0, 10),

    }

}


// RecordSuccess processes a successful health check result

func (hs *HealthState) RecordSuccess(timestamp time.Time) bool {
```

```
    // TODO 1: Acquire write lock on hs.mutex
    // TODO 2: Update LastCheckTime and LastSuccessTime to timestamp
    // TODO 3: Increment TotalChecks and TotalSuccesses counters
    // TODO 4: Reset ConsecutiveFailures to 0
    // TODO 5: Increment ConsecutiveSuccesses
    // TODO 6: Check if we need to transition from unhealthy to healthy
    // TODO 7: If transition needed, update Healthy status and record state change
    // TODO 8: Release lock and return whether state changed
    // Hint: State change happens when currently unhealthy and consecutive successes >= healthy threshold
    return false
}

// RecordFailure processes a failed health check result
func (hs *HealthState) RecordFailure(timestamp time.Time, errorMsg string) bool {
    // TODO 1: Acquire write lock on hs.mutex
    // TODO 2: Update LastCheckTime, LastFailureTime, and LastError
    // TODO 3: Increment TotalChecks and TotalFailures counters
    // TODO 4: Reset ConsecutiveSuccesses to 0
    // TODO 5: Increment ConsecutiveFailures
    // TODO 6: Check if we need to transition from healthy to unhealthy
    // TODO 7: If transition needed, update Healthy status and record state change
    // TODO 8: Release lock and return whether state changed
    // Hint: State change happens when currently healthy and consecutive failures >= unhealthy threshold
    return false
}

// IsHealthy returns the current health status thread-safely
func (hs *HealthState) IsHealthy() bool {
    // TODO 1: Acquire read lock on hs.mutex
    // TODO 2: Read and return hs.Healthy value
    // TODO 3: Release lock
    return true
}

// GetStats returns current statistics in a thread-safe manner
func (hs *HealthState) GetStats() map[string]interface{} {
    // TODO 1: Acquire read lock on hs.mutex
    // TODO 2: Create map with current statistics (healthy status, counters, timing)
    // TODO 3: Calculate derived metrics like success rate, availability percentage
    // TODO 4: Include recent state changes (last 5-10 changes)
    // TODO 5: Release lock and return statistics map
    // Hint: Calculate success rate as TotalSuccesses / TotalChecks if TotalChecks > 0
```

```
    return nil
}

// recordStateChange adds a state transition to the history
func (hs *HealthState) recordStateChange(previousState, newState bool, reason string) {
    change := HealthStateChange{
        Timestamp:     time.Now(),
        PreviousState: previousState,
        NewState:      newState,
        Reason:        reason,
        CheckCount:    int(hs.TotalChecks),
    }

    // TODO 1: Add change to StateChanges slice
    // TODO 2: Limit StateChanges to last 20 entries to prevent memory growth
    // TODO 3: If slice is at capacity, remove oldest entry before adding new one
    // Hint: Use append and slice operations to maintain fixed-size ring buffer behavior
}
```

Milestone Checkpoint

After implementing the health checking system, verify functionality with these checkpoints:

Basic Health Checking Test:

```bash
# Start load balancer with health checking enabled
go run cmd/loadbalancer/main.go --config config/development.json

# In another terminal, check health status endpoint
curl http://localhost:8080/admin/backends

# Expected output: JSON showing all backends with health states
# Verify that healthy backends show "healthy": true
```

Health Check Failure Simulation:

```bash
# Stop one backend server (if using Docker)
docker stop backend-server-1

# Wait for unhealthy threshold * interval seconds
# Check backend status again
curl http://localhost:8080/admin/backends

# Expected: Backend-1 should show "healthy": false
# Expected: Consecutive failures count should equal unhealthy threshold
```

Recovery Testing:

```bash
# Restart the stopped backend
docker start backend-server-1

# Wait for healthy threshold * interval seconds
# Check status again
curl http://localhost:8080/admin/backends

# Expected: Backend-1 should return to "healthy": true
# Expected: State change log should show unhealthy -> healthy transition
```

Signs of Correct Implementation:

- Backends transition to unhealthy after configured consecutive failures
- Healthy backends remain available for request distribution
- State transitions are logged with timestamps and reasons
- Recovery requires configured consecutive successes
- Health check requests appear in backend server logs

Common Issues and Debugging:

- **Health checks not running**: Verify `Enabled: true` in configuration and check for startup errors
- **All backends marked unhealthy**: Check network connectivity and health endpoint paths
- **Frequent state flapping**: Increase failure/success thresholds or check health endpoint reliability
- **Memory usage growth**: Verify state change history is limited and old entries are cleaned up

Request Router Integration

> **Milestone(s):** All milestones (1-4) — the request router serves as the central coordinator that integrates HTTP proxying (Milestone 1), backend selection algorithms (Milestone 2), health checking (Milestone 3), and advanced routing features (Milestone 4) into a unified request processing pipeline

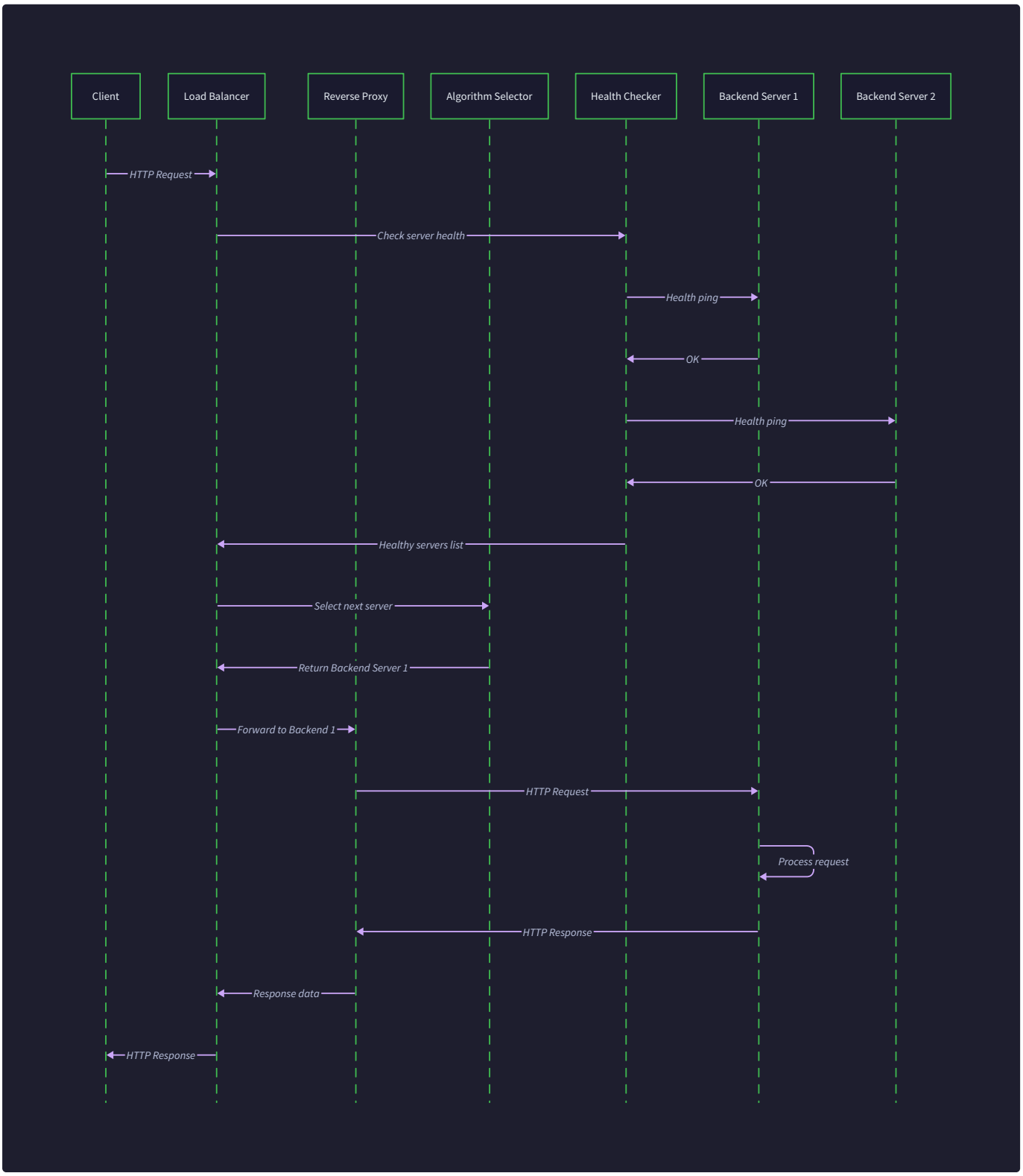
Mental Model: Airport Control Tower

Think of the request router as an airport control tower coordinating the complex dance of aircraft arrivals and departures. Just as an air traffic controller receives incoming flight requests, checks runway availability, weather conditions, and aircraft capabilities before directing each plane to the appropriate runway, the request router receives HTTP requests, checks backend health status, applies load balancing algorithms, and directs each request to the most suitable backend server.

The control tower doesn't fly the planes itself — it coordinates between multiple specialized systems. Weather monitoring provides current conditions (health checking system), runway management tracks capacity and availability (backend pool manager), and flight planning algorithms determine optimal routing (load balancing algorithms). The controller integrates information from all these systems to make real-time routing decisions while handling edge cases like emergency landings (error responses) and runway closures (backend failures).

This coordination role is critical because individual components work well in isolation, but the real complexity emerges in their integration. The request router must handle timing issues (what happens when a backend becomes unhealthy between selection and request forwarding?), resource coordination (ensuring connection limits are respected), and failure scenarios (graceful degradation when multiple systems experience problems simultaneously).

Request Processing Pipeline



The request router orchestrates a sophisticated multi-stage pipeline that transforms incoming HTTP requests into successful responses through careful coordination of specialized components. Understanding this pipeline is essential because each stage introduces potential failure points and requires different error handling strategies.

The **request reception stage** begins when the HTTP server accepts an incoming connection and parses the request headers, body, and metadata. The request router immediately creates a request context containing the original request, client connection information, and timing metadata. This context travels with the request through every subsequent stage, ensuring that timeouts, logging, and cleanup operations can be performed correctly even if intermediate stages fail.

During the **backend selection stage**, the request router queries the `BackendManager` for the current list of healthy backends, then invokes the active load balancing algorithm to choose the most appropriate target. This stage requires careful error handling because the backend pool can change between the health

check and the selection call. The router must handle cases where no healthy backends are available, where the selected backend becomes unhealthy after selection but before connection, and where algorithm state becomes inconsistent due to concurrent updates.

The **request enhancement stage** prepares the original request for forwarding by adding necessary proxy headers, adjusting the request URL to point to the selected backend, and applying any request transformations required by the proxy configuration. The router adds standard headers like `X-Forwarded-For`, `X-Forwarded-Proto`, and `X-Forwarded-Host`, while preserving all original headers that should be forwarded to the backend. Connection-specific headers like `Connection` and `Proxy-Connection` must be handled specially to prevent forwarding connection management directives meant for the proxy itself.

The **connection management stage** establishes or reuses an HTTP connection to the selected backend server. The request router works with the reverse proxy component's connection pool to minimize connection overhead while respecting per-backend connection limits configured in the `BackendConfig`. This stage tracks active connections for algorithms like least connections and enforces timeout policies to prevent resource exhaustion. Connection establishment failures trigger immediate fallback to error response generation without attempting to forward the request.

During the **request forwarding stage**, the enhanced request is transmitted to the backend server with appropriate timeout monitoring. The request router must handle various failure scenarios: connection timeouts during request transmission, backend servers that accept connections but never respond, and backends that respond with partial data before failing. The router maintains request context throughout this process to enable proper cleanup and logging regardless of how the forwarding attempt concludes.

The **response processing stage** receives the backend response and prepares it for transmission back to the original client. The router removes backend-specific headers that shouldn't be exposed to clients, adds any required proxy response headers, and begins streaming the response body back to the client. Response processing must handle cases where backends return invalid HTTP responses, where the client connection is lost during response streaming, and where backends close connections unexpectedly during response transmission.

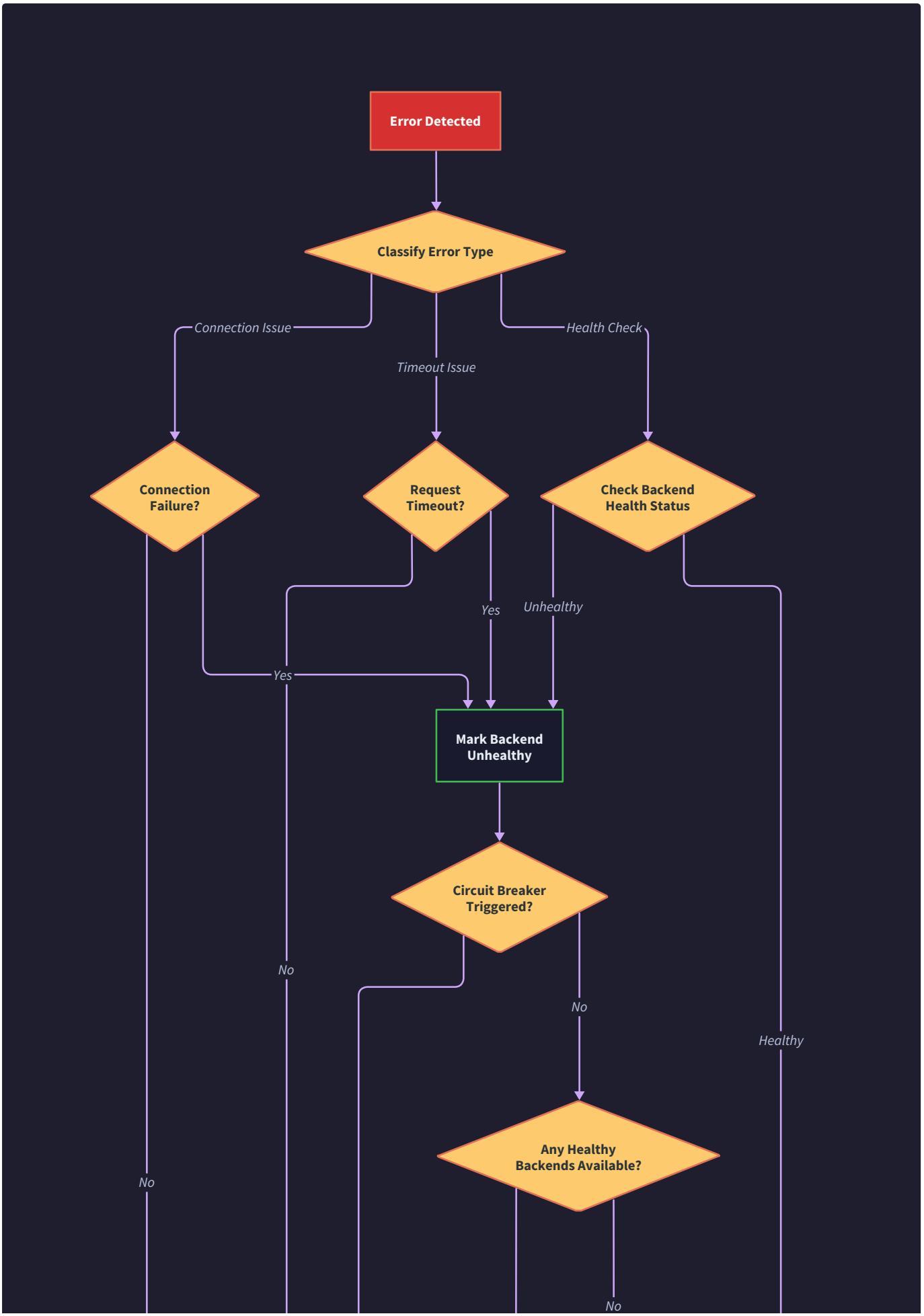
Finally, the **cleanup and logging stage** ensures that all resources are properly released, connection counts are accurately decremented, and comprehensive request logs are generated for monitoring and debugging purposes. This stage executes regardless of whether the request succeeded or failed, making it critical for preventing resource leaks and maintaining accurate system state.

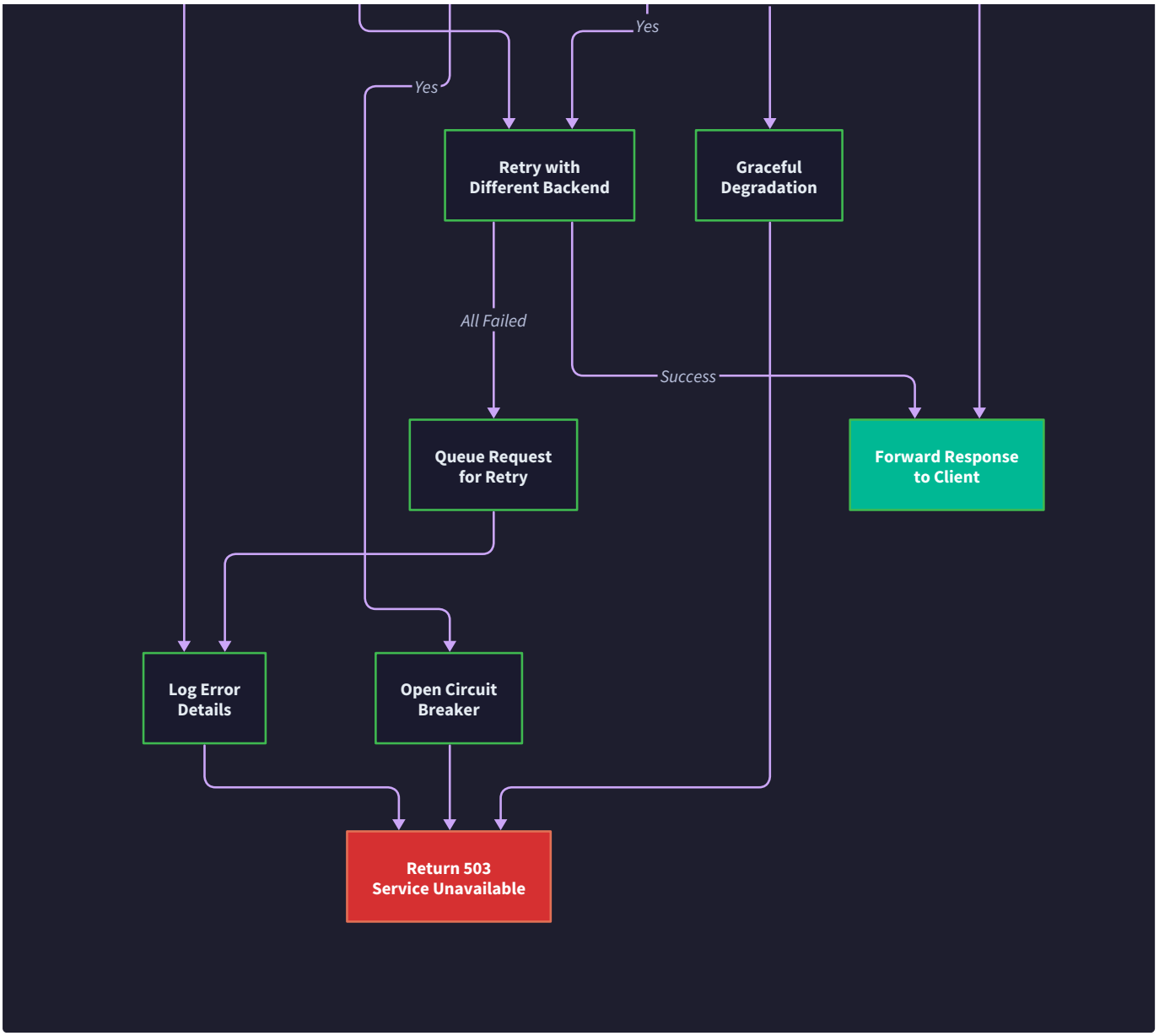
The pipeline stages are designed with **explicit handoff points** where control and responsibility transfer between components. Each handoff includes validation that prerequisites are met and that the receiving component has the information needed to proceed or handle failures gracefully. This design ensures that failures in one stage don't cascade unpredictably to other stages.

Pipeline Stage	Input	Output	Failure Modes	Recovery Strategy
Request Reception	HTTP connection	Parsed request + context	Malformed HTTP, oversized request	Return 400 Bad Request immediately
Backend Selection	Request context + healthy backends	Selected backend + algorithm state	No healthy backends available	Return 503 Service Unavailable
Request Enhancement	Original request + selected backend	Enhanced request ready for forwarding	Header processing errors	Log warning, continue with original headers
Connection Management	Enhanced request + backend config	Active connection to backend	Connection timeout, connection limit exceeded	Return 502 Bad Gateway or 503 Service Unavailable
Request Forwarding	Enhanced request + active connection	Response from backend	Backend timeout, connection lost	Return 502 Bad Gateway, decrement connection count
Response Processing	Backend response + client connection	Client response stream	Invalid response headers, client disconnect	Log error, close connections cleanly
Cleanup and Logging	Request context + final status	Resource cleanup + log entries	Logging system failure	Continue cleanup, fallback to stderr logging

> **Critical Design Insight**: The request router maintains request context throughout the entire pipeline to ensure that timeouts, resource cleanup, and error logging work correctly even when individual stages fail unexpectedly. This context includes the original request timestamp, selected backend information, connection references, and accumulated timing data.

Error Response Handling


Error Detected
Classify Error Type
Connection Issue
Timeout Issue
Health Check
Connection Failure?
Request Timeout?
Check Backend Health Status
Yes
Unhealthy
Yes
Mark Backend Unhealthy
Circuit Breaker Triggered?
No
No
Healthy
Any Healthy Backends Available?
No
No



Error response handling represents one of the most complex aspects of request router integration because errors can originate from multiple sources simultaneously, and the appropriate response depends on the timing and nature of each failure. The request router must generate HTTP responses that accurately reflect the underlying problem while providing useful information for debugging without exposing internal system details.

The **error classification system** categorizes failures into distinct types that require different response strategies. **Client errors** (4xx responses) occur when the incoming request is malformed, oversized, or otherwise invalid — these are detected during request reception and should be handled immediately without attempting backend forwarding. **Backend unavailability errors** (503 Service Unavailable) occur when no healthy backends are available or when all backends are at connection capacity — these indicate temporary system overload. **Backend communication errors** (502 Bad Gateway) occur when backends are selected successfully but fail during connection establishment or request processing — these indicate problems with specific backend servers. **Timeout errors** can manifest as either 504 Gateway Timeout responses when backends are slow to respond, or as connection timeouts that appear as 502 errors when backends never establish connections.

The **error detection pipeline** monitors for failures at each stage of request processing and applies different detection strategies based on the failure type. Connection errors are detected through standard network error handling, examining error types returned by the HTTP client to distinguish between connection refused (backend down), connection timeout (backend overloaded), and DNS resolution failures (configuration problems). Response validation errors are detected by examining HTTP status codes, header formats, and response streaming behavior from backends. Resource exhaustion errors are detected through connection pool monitoring, memory usage tracking, and active request count monitoring.

Error response generation follows HTTP specifications while providing actionable information for debugging. The request router maintains a structured error response format that includes appropriate HTTP status codes, descriptive error messages that don't expose internal architecture details, and correlation IDs that enable request tracing across log files. Error responses include standard headers like `Content-Type`, `Content-Length`, and `Connection` management headers, while avoiding headers that might confuse HTTP clients or proxy chains.

Error Category	HTTP Status	Response Headers	Body Content	When to Use
Invalid Request	400 Bad Request	`Content-Type: text/plain`	"Bad Request: Invalid HTTP request format"	Malformed HTTP, oversized headers, invalid methods
Backend Unavailable	503 Service Unavailable	`Retry-After: 30`	"Service Unavailable: No backend servers available"	Zero healthy backends, all at capacity
Backend Connection Failed	502 Bad Gateway	`Content-Type: text/plain`	"Bad Gateway: Unable to connect to backend server"	Connection refused, DNS failure, connection timeout
Backend Response Timeout	504 Gateway Timeout	`Content-Type: text/plain`	"Gateway Timeout: Backend server response timeout"	Backend accepts request but never responds
Backend Invalid Response	502 Bad Gateway	`Content-Type: text/plain`	"Bad Gateway: Invalid response from backend server"	Malformed HTTP response, connection closed during headers
Internal Router Error	500 Internal Server Error	`Content-Type: text/plain`	"Internal Server Error: Request processing failed"	Algorithm failures, configuration errors, resource exhaustion

The **error correlation system** generates unique request IDs that are included in error responses and logged with sufficient detail to enable effective debugging. Each error log entry includes the request ID, client IP address, requested URL, selected backend (if any), error category, detailed error message, and timing information for each pipeline stage. This correlation data enables operators to trace problematic requests through the entire system and identify patterns in error occurrence.

Graceful degradation strategies determine how the request router behaves when multiple error conditions occur simultaneously. When the health checking system fails, the router can continue operating with the last known healthy backend list, gradually marking backends as unhealthy based on connection failures observed during request forwarding. When the backend selection algorithm fails due to corrupted state, the router can fall back to simple round-robin selection to maintain service availability. When the logging system fails, the router continues processing requests while writing error information to standard error streams.

> **Decision: Error Response Timing**
>
> - **Context**: Error responses must be generated quickly to prevent client timeouts while including sufficient diagnostic information for debugging
> - **Options Considered**: Immediate error responses with minimal detail, delayed responses with comprehensive diagnostic data, or adaptive timing based on error type
> - **Decision**: Generate immediate error responses with essential information while logging comprehensive diagnostic data asynchronously
> - **Rationale**: Client applications typically have short timeout expectations (5-30 seconds), making response speed critical for user experience, while comprehensive diagnostic data is needed for operational debugging
> - **Consequences**: Enables fast client error handling while maintaining debugging capabilities, but requires careful log correlation to match error responses with diagnostic details

Architecture Decisions

The request router integration design requires several critical architectural decisions that fundamentally impact system behavior, performance, and maintainability. These decisions involve trade-offs between consistency, performance, and operational complexity.

Decision: Request Context Management

- **Context**: Request contexts must carry information through the processing pipeline while enabling cleanup and timeout handling across component boundaries
- **Options Considered**: Thread-local storage with global request state, immutable context objects passed between functions, or request-scoped dependency injection containers
- **Decision**: Use immutable context objects with explicit passing between pipeline stages
- **Rationale**: Immutable contexts prevent accidental state modification that could affect concurrent requests, explicit passing makes data flow clear for debugging, and context objects can carry cancellation signals for timeout handling
- **Consequences**: Enables clean timeout handling and resource cleanup, makes request flow explicit for debugging, but requires disciplined context passing throughout the codebase

Context Management Option	Pros	Cons	Chosen?
Thread-local storage	Simple access, no parameter passing	Hidden dependencies, difficult testing	No
Immutable context objects	Explicit data flow, clean cancellation	Requires consistent parameter passing	**Yes**
Dependency injection	Flexible component wiring	Complex configuration, runtime overhead	No

Decision: Error Response Format Standardization

- **Context**: Error responses must provide consistent format for client applications while including sufficient information for debugging without exposing internal architecture
- **Options Considered**: Standard HTTP status codes only, JSON error responses with structured data, or plain text responses with correlation IDs
- **Decision**: Plain text error responses with HTTP status codes and correlation IDs in headers
- **Rationale**: Plain text responses are universally compatible with HTTP clients, correlation IDs in headers enable request tracing without requiring clients to parse response bodies, and avoiding JSON prevents content-type negotiation complexity
- **Consequences**: Maximizes client compatibility and simplifies error handling, enables effective debugging through correlation IDs, but provides less structured error information than JSON responses would

Decision: Backend Selection Timing

- **Context**: Backend selection must occur at an optimal point in the request pipeline to balance load distribution accuracy with failure recovery capabilities
- **Options Considered**: Select backend during request reception, select backend immediately before connection establishment, or pre-select backends and cache selections
- **Decision**: Select backend immediately before connection establishment with fallback selection on connection failure
- **Rationale**: Late selection incorporates the most recent health check information and connection count data, enables fallback to different backends on connection failure, and avoids holding backend selections during request parsing overhead
- **Consequences**: Maximizes selection accuracy and enables connection-level fallback, but requires handling backend selection failures during request processing and increases latency for backend selection overhead

Backend Selection Timing	Pros	Cons	Chosen?
During request reception	Simple pipeline, early failure detection	Stale health data, no connection-level fallback	No
Before connection establishment	Fresh health data, connection fallback possible	Selection failure during processing	**Yes**
Pre-selected with caching	Minimal per-request overhead	Stale selections, complex cache invalidation	No

Decision: Connection Pool Integration

- **Context**: Connection pooling must be integrated with backend selection algorithms while respecting per-backend connection limits and providing accurate connection counting
- **Options Considered**: Global connection pool shared across backends, per-backend connection pools with separate management, or algorithm-managed connection pools with custom lifecycle
- **Decision**: Per-backend connection pools with centralized connection count tracking
- **Rationale**: Per-backend pools enable independent connection management and timeout policies, centralized counting provides accurate data for least-connections algorithm, and separate pools prevent connection starvation between backends with different performance characteristics
- **Consequences**: Enables accurate connection-based load balancing and independent backend tuning, but requires coordination between connection pool management and backend health state updates

Decision: Request Retry Logic

- **Context**: Failed requests could potentially be retried with different backends to improve success rates, but retry logic introduces complexity and potential for request amplification
- **Options Considered**: No retry logic with single-attempt forwarding, automatic retry with exponential backoff, or configurable retry with idempotency detection
- **Decision**: No automatic retry logic in the request router, delegating retry decisions to client applications
- **Rationale**: HTTP requests may have side effects that make retries unsafe without idempotency guarantees, automatic retries can amplify load during backend failures, and client applications have better context for determining retry appropriateness
- **Consequences**: Simplifies request router logic and prevents unsafe request amplification, but requires client applications to implement retry logic and handle transient failures

Retry Logic Option	Pros	Cons	Chosen?
No retry logic	Simple, prevents unsafe retries	Clients must handle failures	**Yes**
Automatic retry with backoff	Improved success rates	Unsafe for non-idempotent requests	No
Configurable retry with idempotency detection	Safe retries, configurable behavior	Complex implementation, difficult configuration	No

Common Pitfalls

Request router integration involves coordination between multiple concurrent systems, making it particularly susceptible to subtle bugs that only manifest under specific timing conditions or failure scenarios. Understanding these common pitfalls helps prevent difficult-to-reproduce issues in production environments.

⚠️ Pitfall: Request Context Loss During Error Handling

The most frequent integration mistake is losing request context information when errors occur, making it impossible to perform proper cleanup or generate meaningful log entries. This typically happens when error handling code paths bypass the normal context passing mechanisms or when panic recovery doesn't preserve context data.

For example, when a backend selection algorithm throws an exception due to corrupted internal state, the error handling code might generate a 500 error response without access to the original request context. This results in connection leaks (because connection counts can't be decremented without knowing which backend was selected), incomplete log entries (because request timing and client information is lost), and difficult debugging (because error correlation becomes impossible).

The fix requires implementing comprehensive error handling that preserves request context throughout all failure scenarios. Error handling code must receive the same context object as success paths, and panic recovery mechanisms must capture context from the goroutine or thread stack. All error response generation must include request correlation IDs from the preserved context.

⚠️ Pitfall: Race Conditions Between Health Updates and Request Routing

Backend health states can change between the time a backend is selected by the load balancing algorithm and when the actual connection attempt is made. This creates a race condition where the router attempts to connect to a backend that has just been marked unhealthy by the health checking system.

The symptom is intermittent connection failures that seem to contradict the load balancer's health checking logic — clients receive 502 Bad Gateway responses even when health checks show all backends as healthy. This happens because health state updates and request routing operate on different timelines and use different data synchronization mechanisms.

The fix requires implementing connection-time health validation that double-checks backend health immediately before connection establishment. If the selected backend has become unhealthy since selection, the request router should attempt fallback selection to choose a different healthy backend. The health checking system must also use appropriate synchronization primitives to ensure that health state updates are visible to concurrent request processing threads.

⚠️ Pitfall: Improper Error Response Headers

Generating error responses with incorrect or missing HTTP headers can cause problems for client applications, proxy chains, and debugging tools. Common header mistakes include missing `Content-Length` headers for error response bodies, incorrect `Content-Type` headers, missing `Connection` management headers, and accidentally forwarding backend-specific headers in error responses.

For example, when a backend connection fails, the request router might generate a 502 error response but forget to set appropriate `Content-Type: text/plain` headers. Client applications that expect JSON responses might then attempt to parse the plain text error message as JSON, leading to client-side parsing errors that obscure the original backend connection problem.

The fix requires implementing standardized error response generation that always includes appropriate headers for each error type. Error response generation should use template-based approaches that ensure header consistency and should never forward headers from failed backend responses. Error responses should include correlation headers that enable request tracing without exposing internal architecture details.

⚠️ Pitfall: Resource Leaks During Connection Failures

When backend connections fail after successful establishment but before request completion, the request router must carefully manage resource cleanup to prevent connection leaks and inaccurate connection counting. Failed connection cleanup often introduces subtle resource leaks that accumulate over time and eventually

cause connection pool exhaustion.

The symptom is gradually increasing connection counts that never decrease properly, eventually leading to "too many open files" errors or connection pool exhaustion. This often occurs during periods of backend instability when many connections are being established and torn down rapidly.

The fix requires implementing comprehensive resource cleanup using defer statements or try-finally blocks that execute regardless of how the request processing concludes. Connection counting must be decremented in all code paths that increment it, including error paths and panic recovery handlers. Connection cleanup must also handle partially established connections that fail during HTTP handshaking.

⚠️ Pitfall: Blocking Operations in Request Processing Pipeline

Request processing pipelines that include blocking operations without appropriate timeout handling can cause cascading failures where slow or unresponsive backends affect the router's ability to handle other requests. This is particularly problematic when backend selection algorithms or health state updates perform blocking operations during request processing.

For example, if the backend selection algorithm needs to acquire a lock on internal state and that lock is held by a goroutine performing a slow health check operation, all incoming requests will block waiting for backend selection to complete. This turns a problem with one component into a system-wide availability issue.

The fix requires implementing non-blocking patterns for all operations in the request processing pipeline, using timeouts for any operations that might block, and designing component interfaces to avoid holding locks during external operations. Backend selection should use lock-free algorithms where possible, and health state updates should use optimistic concurrency control rather than long-held exclusive locks.

⚠️ Pitfall: Inconsistent Logging During Failure Scenarios

Request routing failures often result in incomplete or inconsistent log entries that make debugging difficult, especially when multiple error conditions occur simultaneously. This happens when different components use different logging formats, when error handling code paths don't include proper logging, or when log correlation data is lost during failure scenarios.

The symptom is log files that contain error messages without sufficient context to understand what went wrong, making it difficult to identify patterns in failures or correlate related events. Production debugging becomes particularly challenging when the information needed to understand a failure is spread across multiple log entries with different formats.

The fix requires implementing structured logging with consistent correlation IDs throughout the request processing pipeline. All error handling code paths must generate log entries with the same format and correlation data as success paths. Log entries should include sufficient context information to understand the failure without requiring correlation with other log entries, while still providing correlation IDs to enable cross-component tracing when needed.

Implementation Guidance

The request router integration serves as the central coordinator that must handle complex interactions between multiple concurrent systems while maintaining high performance and reliability. This section provides concrete implementation guidance for building a robust request router that can handle the complexities of production traffic.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
HTTP Server	`net/http` with default ServeMux	`net/http` with custom handler + middleware chain
Request Context	`context.Context` with values	Custom context struct with typed fields
Error Handling	Standard `error` interface + logging	Structured errors with error codes + correlation IDs
Connection Pooling	`http.Client` with default transport	Custom `http.RoundTripper` with per-backend pools
Timeout Management	Request-level timeouts	Per-stage timeouts with context cancellation
Logging	Standard `log` package	Structured logging with `logrus` or `zap`

B. Recommended File Structure:

```
internal/router/
  router.go                 ← main RequestRouter implementation
  context.go                ← request context management
  pipeline.go               ← request processing pipeline stages
  errors.go                 ← error handling and response generation
  middleware.go             ← HTTP middleware for logging/metrics
  router_test.go           ← integration tests
  testdata/
    requests.http           ← test HTTP requests
    responses.json          ← expected response formats
```

C. Infrastructure Starter Code:

```go
// Package router integrates HTTP proxying, backend selection, and health checking
// into a unified request processing pipeline.
package router

import (
    "context"
    "fmt"
    "net/http"
    "sync/atomic"
    "time"

    "github.com/yourproject/internal/backend"
    "github.com/yourproject/internal/health"
    "github.com/yourproject/internal/proxy"
)

// RequestContext carries request information through the processing pipeline
type RequestContext struct {
    RequestID     string
    StartTime     time.Time
    ClientIP      string
    OriginalURL   string
    SelectedBackend *backend.Backend
    Stage         string
    Metrics       map[string]interface{}
}

// NewRequestContext creates a new request context with correlation ID
func NewRequestContext(r *http.Request) *RequestContext {
    return &RequestContext{
        RequestID:   generateRequestID(),
        StartTime:   time.Now(),
        ClientIP:    extractClientIP(r),
        OriginalURL: r.URL.String(),
        Stage:       "reception",
        Metrics:     make(map[string]interface{}),
    }
}

// ErrorResponse represents a standardized error response
type ErrorResponse struct {
    Status      int
```

```
    Message     string
    RequestID   string
    Headers     map[string]string
}

// WriteErrorResponse writes a standardized error response to the client
func WriteErrorResponse(w http.ResponseWriter, err ErrorResponse) {
    // Set standard error response headers
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    w.Header().Set("X-Request-ID", err.RequestID)
    w.Header().Set("Connection", "close")

    // Add any additional headers
    for key, value := range err.Headers {
        w.Header().Set(key, value)
    }

    w.WriteHeader(err.Status)
    fmt.Fprintf(w, "%s\n", err.Message)
}

// PipelineStage represents a stage in the request processing pipeline
type PipelineStage func(*RequestContext, http.ResponseWriter, *http.Request) error

// RequestRouter coordinates between HTTP proxy, backend selection, and health checking
type RequestRouter struct {
    proxy          *proxy.ReverseProxy
    backendManager *backend.Manager
    healthChecker  *health.HealthChecker

    // Metrics and monitoring
    totalRequests   int64
    successRequests int64
    errorRequests   int64
}

// NewRequestRouter creates a new request router with integrated components
func NewRequestRouter(proxy *proxy.ReverseProxy, backends *backend.Manager, health *health.HealthChecker) *RequestRouter {
    return &RequestRouter{
        proxy:          proxy,
        backendManager: backends,
        healthChecker:  health,
```

```
    }

}
```

D. Core Logic Skeleton Code:

```go
// ServeHTTP handles incoming HTTP requests through the integrated processing pipeline
func (rr *RequestRouter) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Create request context with correlation ID and timing information
    // TODO 2: Execute request processing pipeline with error handling
    // TODO 3: Update request metrics and generate log entries
    // Hint: Use defer for cleanup operations that must execute regardless of success/failure
}

// processRequest executes the main request processing pipeline
func (rr *RequestRouter) processRequest(ctx *RequestContext, w http.ResponseWriter, r *http.Request) error {
    // TODO 1: Validate incoming request format and size limits
    // TODO 2: Select backend using current load balancing algorithm
    // TODO 3: Enhance request with proxy headers and backend target URL
    // TODO 4: Forward request to selected backend with timeout handling
    // TODO 5: Stream response back to client with error detection
    // TODO 6: Update connection counts and algorithm state
    // Hint: Each stage should update ctx.Stage for debugging and logging
}

// selectBackend chooses an appropriate backend server for the request
func (rr *RequestRouter) selectBackend(ctx *RequestContext) (*backend.Backend, error) {
    // TODO 1: Get list of currently healthy backends from backend manager
    // TODO 2: Apply load balancing algorithm to select specific backend
    // TODO 3: Validate selected backend is still healthy (race condition check)
    // TODO 4: Update context with selected backend information
    // TODO 5: Handle case where no healthy backends are available
    // Hint: Return specific error types for different failure scenarios
}

// enhanceRequest adds proxy headers and modifies request for backend forwarding
func (rr *RequestRouter) enhanceRequest(ctx *RequestContext, r *http.Request, backend *backend.Backend) *http.Request {
    // TODO 1: Clone original request to avoid modifying client request
    // TODO 2: Update request URL to point to selected backend
    // TODO 3: Add standard proxy headers (X-Forwarded-For, X-Forwarded-Proto, etc.)
    // TODO 4: Add correlation headers for request tracing
    // TODO 5: Remove connection-specific headers that shouldn't be forwarded
    // Hint: Use http.Request.Clone() and modify the clone, not the original
}

// handleRequestError generates appropriate error responses based on failure type and stage
func (rr *RequestRouter) handleRequestError(ctx *RequestContext, w http.ResponseWriter, err error) {
    // TODO 1: Classify error type (client error, backend unavailable, connection failed, etc.)
```

```
    // TODO 2: Generate appropriate HTTP status code and error message
    // TODO 3: Include correlation ID and timing information in response
    // TODO 4: Log error with full context for debugging
    // TODO 5: Update error metrics and notify monitoring systems
    // Hint: Different error types in different pipeline stages require different HTTP status codes
}

// updateMetrics records request metrics for monitoring and observability
func (rr *RequestRouter) updateMetrics(ctx *RequestContext, success bool, statusCode int) {
    // TODO 1: Increment total request counter atomically
    // TODO 2: Update success/error counters based on outcome
    // TODO 3: Record request latency and backend selection timing
    // TODO 4: Update per-backend request counts and latency metrics
    // TODO 5: Generate structured log entry with all context information
    // Hint: Use atomic operations for counters accessed from multiple goroutines
}
```

E. Language-Specific Hints:

- Use `context.WithTimeout()` to enforce request-level timeouts that propagate through the entire pipeline
- Implement `defer` statements for resource cleanup that must execute in all code paths
- Use `atomic.AddInt64()` for connection counting and metrics that need thread-safe updates
- Clone HTTP requests with `r.Clone(ctx)` to avoid modifying the original request during enhancement
- Use `http.Error()` for simple error responses, but implement custom error response formatting for consistency
- Handle `context.Canceled` and `context.DeadlineExceeded` explicitly to distinguish client disconnects from timeouts
- Use buffered channels for component communication to prevent blocking in the request processing pipeline

F. Milestone Checkpoint:

After implementing the request router integration:

1. **Start the load balancer server**: `go run cmd/server/main.go`
2. **Test basic request routing**: `curl -v http://localhost:8080/test` should return a response from a backend server
3. **Verify error handling**: Stop all backend servers, then `curl -v http://localhost:8080/test` should return `503 Service Unavailable`
4. **Check request correlation**: Look for `X-Request-ID` headers in responses and matching correlation IDs in log files
5. **Validate pipeline stages**: Log entries should show progression through pipeline stages (reception → selection → forwarding → response)

Expected Behavior:

- Successful requests return backend responses with added proxy headers
- Backend failures result in appropriate 5xx error responses with correlation IDs
- Request metrics increment properly for both successful and failed requests
- Log entries contain sufficient context for debugging without being overly verbose
- Connection counts reflect actual backend connections and decrease after request completion

Signs of Problems:

- Missing correlation IDs in error responses indicates context loss during error handling
- Connection counts that never decrease suggests resource leaks in cleanup code
- Log entries without request context suggests error handling bypassing normal logging
- Responses missing proxy headers indicates request enhancement stage failures
- High memory usage over time suggests resource leaks in connection management

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Requests hang indefinitely	Missing timeout handling in pipeline stages	Check for blocking operations without context timeouts	Add context.WithTimeout to all external calls
Error responses missing correlation IDs	Context loss during error handling paths	Trace error handling code for context parameter passing	Ensure all error handlers receive and use request context
Connection count constantly increasing	Resource leaks in connection cleanup	Monitor connection increment/decrement in logs	Add defer statements for connection cleanup in all code paths
502 errors for healthy backends	Race condition between health checks and routing	Check timing of health updates vs. request processing	Add connection-time health validation before forwarding
Inconsistent load distribution	Algorithm state corruption during errors	Examine algorithm state after error scenarios	Implement proper error recovery in algorithm state management
Memory usage growing over time	Request contexts not being garbage collected	Profile memory usage and examine context lifecycle	Ensure request contexts don't hold references to large objects

Configuration and Runtime Control

> **Milestone(s):** All milestones (1-4) — configuration management is essential throughout the project, supporting backend registration (Milestone 2), health check parameters (Milestone 3), and algorithm selection (Milestone 4) with hot reload capabilities

Mental Model: Theater Program Changes

Think of running a load balancer like managing a live theater production. During a performance, the theater director occasionally needs to make changes to the program — perhaps adding an understated actor when someone calls in sick, adjusting the lighting cues, or changing the intermission timing. These changes must happen seamlessly without stopping the show or disrupting the audience experience.

In a theater, the stage manager receives updated instructions from the director, validates that the changes make sense (the understudy knows the role, the new lighting cue won't blind the actors), and then coordinates with the crew to implement the changes during appropriate moments — perhaps between scenes or during natural breaks. The audience never sees the coordination happening behind the scenes.

Similarly, a load balancer must accept configuration updates while continuing to serve live traffic. When operators need to add new backend servers, adjust health check intervals, or switch load balancing algorithms, these changes should take effect without dropping existing connections or causing service interruptions. The configuration system acts like the stage manager, validating changes and coordinating their implementation across all components.

Just as the theater maintains a master program that all crew members reference, the load balancer maintains a central configuration that all components — the HTTP proxy, backend manager, health checker, and algorithm engines — use as their source of truth. When this configuration changes, each component must gracefully adopt the new settings while preserving its current operational state.

Configuration Loading

The configuration loading system provides the foundation for how the load balancer discovers its operational parameters and validates their correctness before applying them. This system must handle both initial startup configuration and runtime updates, ensuring that invalid configurations never compromise system stability.

File-Based Configuration Structure

The primary configuration mechanism uses structured files that operators can edit and deploy through standard configuration management tools. The configuration file contains all parameters needed to operate the load balancer, organized into logical sections that correspond to system components.

Configuration Section	Purpose	Key Parameters
Port	HTTP listener configuration	Port number, bind address
Backends	Backend server definitions	URLs, weights, connection limits, timeouts
Algorithm	Load balancing strategy	Algorithm name, algorithm-specific parameters
HealthCheck	Health monitoring settings	Intervals, thresholds, probe endpoints
Proxy	HTTP proxy behavior	Timeouts, header limits, connection pooling
Logging	Observability configuration	Log levels, output formats, destinations

The `Config` structure represents the complete system configuration with strongly typed fields that enable compile-time validation and IDE assistance. Each backend server gets its own `BackendConfig` entry with comprehensive connection and timeout parameters that allow fine-tuning for different backend characteristics.

Configuration Validation Logic

Before any configuration takes effect, the validation system performs comprehensive checks to ensure operational safety and logical consistency. This validation happens in multiple phases, from basic syntax checking to complex inter-dependency validation.

The validation process follows this sequence:

1. **Syntax Validation**: Parse the configuration file format and verify all required fields are present with correct data types
2. **Range Validation**: Check that numeric values fall within acceptable ranges (positive timeouts, valid port numbers)
3. **URL Validation**: Verify that backend URLs are well-formed and use supported protocols (HTTP/HTTPS)
4. **Dependency Validation**: Ensure referenced components exist (algorithm names match implemented algorithms)
5. **Consistency Validation**: Check that related settings make logical sense together (health check timeout less than interval)
6. **Resource Validation**: Verify that the configuration is achievable with available system resources

Validation Type	Check Performed	Example Failure	Recovery Action
Syntax	JSON/YAML parsing	Malformed JSON	Reject configuration, keep current
Range	Numeric bounds	Negative timeout	Reject configuration, log specific error
URL Format	Backend URL parsing	Invalid hostname	Reject configuration, identify bad URL
Algorithm	Known algorithm name	Typo in algorithm	Reject configuration, suggest alternatives
Consistency	Cross-field validation	Timeout > interval	Reject configuration, explain conflict
Resource	System capability	Too many backends	Reject configuration, suggest limits

Error Handling and Fallback Behavior

When configuration loading encounters errors, the system must maintain operational stability while providing clear feedback about what went wrong. The error handling strategy prioritizes service availability over configuration updates.

> **Critical Design Principle**: Configuration errors never bring down a running load balancer. Invalid updates are rejected while the current configuration remains active.

The error handling system categorizes failures by severity and recoverability:

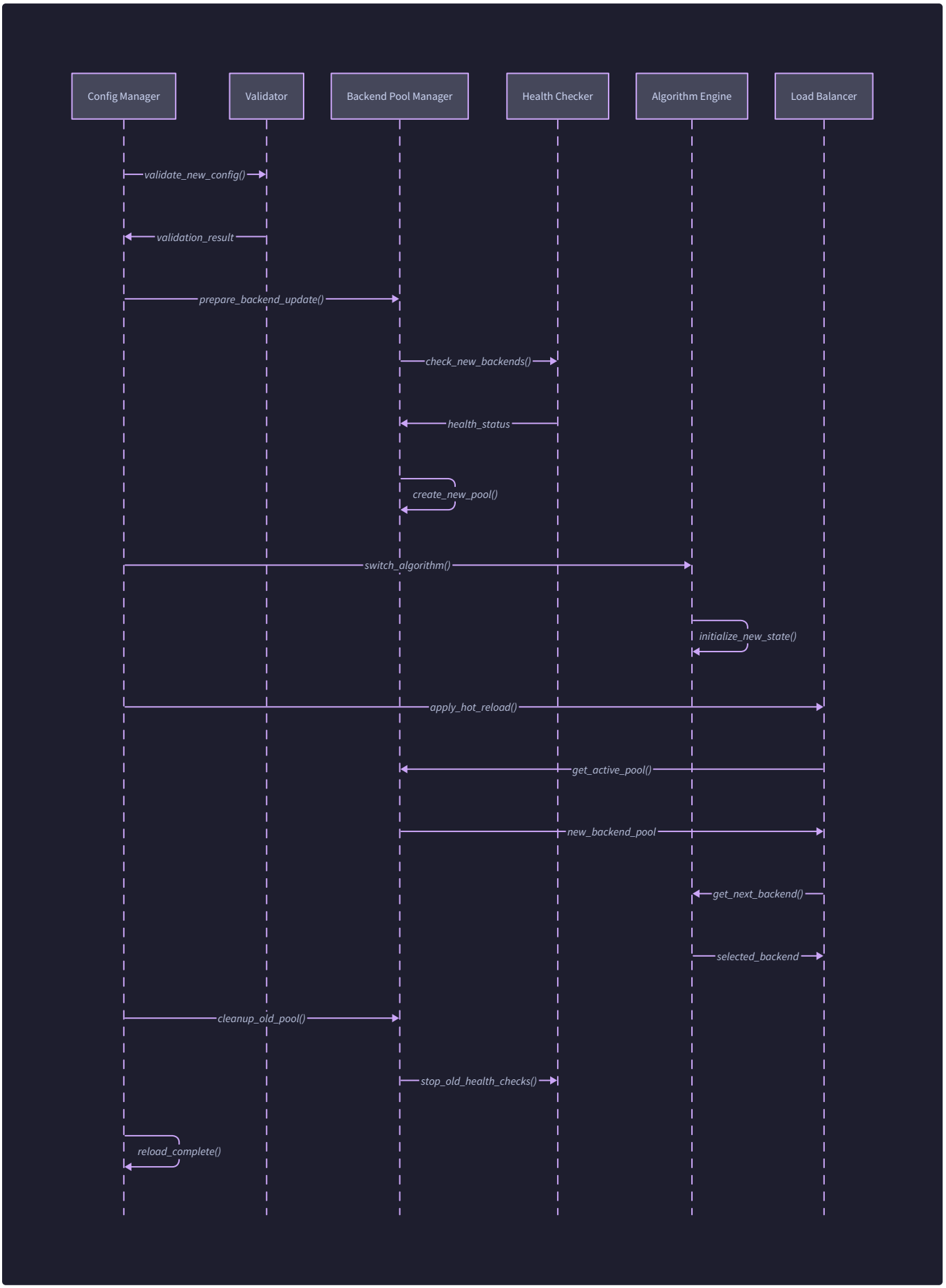
- **Fatal Errors**: Problems that prevent any operation (missing configuration file at startup)
- **Validation Errors**: Invalid configuration content that can be corrected (bad syntax, invalid values)
- **Warning Conditions**: Suboptimal but workable configuration (single backend server, very short timeouts)

For each error category, the system provides detailed diagnostic information including the specific configuration element that failed, the nature of the problem, and suggested corrections. This feedback helps operators quickly identify and fix configuration issues.

> **Decision: Configuration Validation Strategy**
>
> - **Context**: Configuration errors can cause service outages or unpredictable behavior, but overly strict validation might prevent legitimate configurations
> - **Options Considered**:
>   1. Strict validation that rejects any questionable configuration
>   2. Permissive validation that allows dubious but syntactically correct configuration
>   3. Layered validation with errors vs warnings
> - **Decision**: Implement layered validation with clear error/warning distinction
> - **Rationale**: This approach prevents dangerous configurations while allowing operators flexibility for unusual but valid setups. Warning messages help operators understand potential issues without blocking deployments.
> - **Consequences**: More complex validation logic but better operator experience and safer deployments

Hot Reload Mechanism

The hot reload mechanism enables configuration updates without service interruption, allowing operators to modify load balancer behavior while maintaining existing connections and preserving service availability. This capability is essential for production environments where downtime is unacceptable.



Configuration Change Detection

The hot reload system monitors for configuration changes through multiple mechanisms that accommodate different operational workflows. File-based monitoring watches for changes to configuration files on disk, while API-based updates provide programmatic configuration management integration.

The change detection system uses file system notifications (inotify on Linux, fsevents on macOS) to avoid polling overhead while ensuring rapid response to configuration updates. When file system notifications are unavailable, the system falls back to periodic file modification time checking with configurable intervals.

Detection Method	Mechanism	Latency	Use Case
File System Events	inotify/fsevents	< 100ms	Development, manual updates
API Endpoint	HTTP POST	< 50ms	Automated deployment, orchestration
Periodic Polling	Stat() calls	1-30s	Fallback, network filesystems
Signal Handling	SIGHUP	< 10ms	Traditional Unix administration

Graceful Configuration Transitions

When a configuration update is detected and validated, the system must coordinate the transition across all components while preserving operational state. This coordination happens through a carefully orchestrated sequence that minimizes disruption.

The transition process follows these phases:

1. **Preparation Phase**: Load and validate the new configuration without affecting current operations
2. **Coordination Phase**: Notify all components of the pending configuration change and allow them to prepare
3. **Transition Phase**: Switch components to the new configuration in dependency order
4. **Validation Phase**: Verify that all components successfully adopted the new configuration
5. **Cleanup Phase**: Release resources associated with the old configuration

Each component implements a configuration transition interface that supports atomic updates with rollback capability. If any component fails to adopt the new configuration, the entire update is rolled back to maintain consistency.

Backend Pool Updates

Backend pool changes represent the most common configuration updates and require special handling to manage existing connections gracefully. When backend servers are added, removed, or modified, the system must update its internal state while respecting active connections.

Adding new backend servers involves creating new `Backend` instances, initializing their health state, and registering them with the health checker before making them available for request routing. This process ensures that new backends are verified as healthy before receiving traffic.

Removing backend servers requires a more complex process to handle existing connections gracefully:

1. **Mark for Removal**: Flag the backend as no longer accepting new requests
2. **Drain Connections**: Allow existing requests to complete naturally
3. **Health Check Cleanup**: Remove the backend from active health monitoring
4. **Resource Cleanup**: Close connection pools and release backend-specific resources

Backend configuration changes (weight adjustments, timeout modifications) can often be applied immediately since they don't affect connection state. However, changes that affect connection behavior (timeouts, limits) may require connection pool recreation.

Update Type	Immediate Effect	Graceful Actions	Rollback Requirements
Add Backend	Health check start	Register with algorithms	Remove from pool
Remove Backend	Mark unavailable	Drain connections	Re-add to pool
Weight Change	Update algorithms	None	Revert weight value
Timeout Change	Update proxy config	Recreate connection pools	Revert timeouts

Algorithm Switching

Changing load balancing algorithms during operation requires careful state management since different algorithms maintain different internal state structures. The algorithm switching process must preserve request distribution fairness while transitioning to the new selection strategy.

The switching process coordinates with the request router to ensure that in-flight requests complete using the current algorithm while new requests use the updated algorithm. This prevents inconsistent behavior during the transition window.

Some algorithm transitions require state migration or reset:

- **Round Robin to Weighted Round Robin**: Preserve current position in rotation
- **Least Connections to IP Hash**: Reset connection tracking, build hash state
- **Any Algorithm to Random**: No state preservation needed

Decision: Algorithm State Preservation vs Reset

- **Context**: When switching algorithms, existing state might be incompatible with the new algorithm's requirements
- **Options Considered**:
  1. Always preserve state and attempt migration
  2. Always reset state on algorithm changes
  3. Preserve when possible, reset when necessary
- **Decision**: Preserve compatible state, reset incompatible state with clear logging
- **Rationale**: This provides the best balance of consistency and performance. Compatible transitions maintain fairness, while incompatible transitions are handled safely with clear operator feedback.
- **Consequences**: More complex transition logic but better operational experience and fewer traffic distribution surprises

Health Check Parameter Updates

Health check configuration changes affect the monitoring behavior for all backend servers and must be applied consistently across the entire backend pool. These updates can significantly impact system behavior, so they require careful coordination.

Interval changes take effect at the next scheduled check cycle to avoid disrupting the current check timing. Threshold changes (healthy/unhealthy counts) are applied immediately but may trigger state reevaluation for backends near threshold boundaries.

The health checker must handle parameter updates atomically to prevent inconsistent checking behavior:

1. **Pause Scheduling**: Stop scheduling new health checks temporarily
2. **Update Configuration**: Apply new intervals, timeouts, and thresholds
3. **Reevaluate State**: Check if current backend states remain valid under new thresholds
4. **Resume Scheduling**: Restart periodic checking with new parameters

Architecture Decisions

The configuration and runtime control system involves several critical architecture decisions that affect both operational flexibility and implementation complexity. Each decision represents a trade-off between different operational requirements.

Decision: Configuration Format Choice

- **Context**: The configuration format affects both human readability and machine processing, with implications for validation, tooling, and operator experience
- **Options Considered**:
  1. JSON for simplicity and universal parsing support
  2. YAML for human readability and comment support
  3. TOML for simplicity with better human readability than JSON
- **Decision**: Support both JSON and YAML with automatic format detection
- **Rationale**: JSON provides universal tooling support and parsing reliability, while YAML offers better operator experience with comments and readability. Supporting both accommodates different operational preferences and integration requirements.
- **Consequences**: More complex parsing logic but better flexibility for different deployment scenarios and operator preferences

Format	Pros	Cons	Best Use Case
JSON	Universal support, strict syntax	No comments, less readable	Automated generation, APIs
YAML	Human readable, supports comments	More complex parsing, indentation sensitive	Manual editing, documentation
TOML	Simple syntax, good readability	Less widespread support	Small configurations

Decision: Hot Reload vs Restart Requirement

- **Context**: Configuration updates can be handled either through service restart (simpler) or hot reload (more complex but zero downtime)
- **Options Considered**:
  1. Require full service restart for all configuration changes
  2. Support hot reload for safe changes, restart for others
  3. Support hot reload for all configuration changes
- **Decision**: Implement hot reload for all changes with atomic rollback on failure
- **Rationale**: Production load balancers cannot afford downtime for routine configuration updates. Complete hot reload support enables operational flexibility and reduces deployment risk.
- **Consequences**: Significantly more complex implementation but essential for production operation and continuous deployment workflows

Decision: Configuration Validation Strictness

- **Context**: Configuration validation can be strict (reject questionable configs) or permissive (allow operator flexibility)
- **Options Considered**:
  1. Strict validation that rejects any potentially problematic configuration
  2. Permissive validation that allows any syntactically correct configuration
  3. Layered validation with mandatory checks and optional warnings
- **Decision**: Implement layered validation with clear error vs warning classification
- **Rationale**: This approach prevents dangerous configurations while preserving operator flexibility for legitimate edge cases. Warning messages provide guidance without blocking valid but unusual configurations.
- **Consequences**: More nuanced validation logic but better operator experience and fewer false positives that block legitimate deployments

Configuration Storage and Persistence

The configuration storage strategy determines how the system maintains authoritative configuration state and handles persistence across restarts. This decision affects both operational complexity and reliability requirements.

The system maintains configuration state in memory for runtime performance while persisting validated configurations to disk for restart recovery. This dual storage approach provides fast access for request processing while ensuring configuration durability.

Storage Location	Purpose	Update Frequency	Backup Requirements
Memory	Runtime access	Every request	Not applicable
Local File	Persistence	On validation	Regular filesystem backup
External Store	Centralized management	On deploy	Distributed backup
Version Control	Change tracking	On commit	Repository backup

Configuration Distribution in Multi-Instance Deployments

When multiple load balancer instances operate together, configuration distribution ensures consistency across all instances while accommodating network partitions and deployment timing differences.

The configuration distribution strategy must handle several challenges:

- **Consistency**: All instances should eventually converge to the same configuration
- **Availability**: Instance startup should not depend on external configuration services
- **Versioning**: Configuration updates should be applied atomically across instances
- **Rollback**: Failed deployments should be recoverable across all instances

Decision: Configuration Distribution Strategy

- **Context**: Multi-instance deployments need consistent configuration across all load balancer instances
- **Options Considered**:
  1. Centralized configuration service with polling
  2. File-based distribution with shared storage
  3. Each instance manages its own configuration independently
- **Decision**: File-based configuration with shared storage and local caching
- **Rationale**: This provides good consistency with operational simplicity. Shared storage handles distribution while local caching ensures availability during storage outages. Standard deployment tools can manage file distribution effectively.
- **Consequences**: Dependency on shared storage but simpler architecture than distributed configuration services

Common Pitfalls

Configuration and runtime control systems are prone to several categories of errors that can cause service disruptions or unpredictable behavior. Understanding these pitfalls helps avoid common implementation mistakes.

⚠️ Pitfall: Partial Configuration Application

A common mistake is applying configuration updates partially when some components succeed in adopting new settings while others fail. This creates an inconsistent system state where different components operate with different configuration assumptions.

Why it's wrong: Partial configuration application can cause request routing failures, connection mismatches, and health check inconsistencies. For example, if the backend pool updates successfully but the health checker fails to adopt new thresholds, backends might be incorrectly marked as unhealthy.

How to fix: Implement atomic configuration updates with all-or-nothing semantics. If any component fails to adopt the new configuration, roll back all changes and maintain the previous consistent state. Use two-phase configuration updates: validate all changes first, then apply all changes together.

⚠️ Pitfall: Configuration Validation Race Conditions

When multiple configuration updates arrive simultaneously, validation logic can interfere with itself, leading to corrupted validation state or inconsistent rejection decisions.

Why it's wrong: Race conditions in validation can cause the system to accept invalid configurations or reject valid ones. This creates unpredictable behavior that's difficult to debug and can compromise system reliability.

How to fix: Serialize configuration updates using a mutex or queue to ensure that only one configuration change is processed at a time. Validate against a consistent snapshot of current state rather than live state that might be changing during validation.

⚠️ Pitfall: Memory Leaks During Configuration Updates

Each configuration update creates new objects (backend instances, algorithm state, connection pools) but may fail to clean up old objects properly, leading to memory leaks over time.

Why it's wrong: Memory leaks eventually cause the load balancer to exhaust available memory and crash. Even small leaks become significant problems over time in production systems with frequent configuration updates.

How to fix: Implement explicit lifecycle management for all configuration-related objects. Use reference counting or ownership tracking to ensure that old configurations are fully cleaned up when new configurations are applied. Include memory usage monitoring to detect leaks early.

⚠️ Pitfall: Invalid Default Value Assumptions

Configuration loading often assumes that missing configuration values should use "reasonable" defaults, but these defaults might not be appropriate for all deployment scenarios.

Why it's wrong: Inappropriate defaults can cause performance problems (timeouts too short), security issues (permissive settings), or operational difficulties (verbose logging). Operators might not realize that critical settings are using defaults rather than explicit values.

How to fix: Make critical configuration explicit rather than defaulted. Provide clear documentation about default values and their implications. Consider requiring explicit values for settings that significantly affect behavior rather than silently applying defaults.

⚠️ Pitfall: Configuration File Format Lock-in

Choosing a single configuration file format without considering future requirements can create operational difficulties when integration needs change.

Why it's wrong: Different operational tools and deployment pipelines have preferences for different configuration formats. Forcing operators to use an incompatible format creates unnecessary friction and may prevent integration with existing workflows.

How to fix: Support multiple configuration formats or choose formats with broad tooling support. Implement format detection so operators can use their preferred format without explicit format specification. Document the trade-offs between different formats clearly.

⚠️ Pitfall: Health Check Configuration Inconsistencies

When health check parameters are updated, existing backend health states might become invalid under the new configuration, but the system continues using stale state information.

Why it's wrong: Stale health state can cause backends to remain in incorrect states (healthy when they should be unhealthy or vice versa) until the next natural state transition. This defeats the purpose of configuration updates and can impact traffic distribution.

How to fix: Reevaluate all backend health states when health check configuration changes. Reset consecutive success/failure counters appropriately when thresholds change. Document which configuration changes trigger health state reevaluation.

Implementation Guidance

The configuration and runtime control system requires careful implementation to balance flexibility, reliability, and performance. This section provides practical guidance for building a robust configuration management system.

Technology Recommendations

Component	Simple Option	Advanced Option
File Format	JSON with `encoding/json`	YAML with `gopkg.in/yaml.v3`
File Watching	Periodic stat() polling	`fsnotify` library for events
Validation	Manual field checking	JSON Schema with `gojsonschema`
Atomic Updates	Mutex-protected single config	Copy-on-write with atomic pointer
Hot Reload API	HTTP endpoint with `net/http`	gRPC with `google.golang.org/grpc`

Recommended File Structure

```
project-root/
  cmd/loadbalancer/
    main.go                   ← entry point with config loading
  internal/config/
    config.go                 ← configuration types and validation
    loader.go                 ← file loading and parsing logic
    watcher.go                ← file change detection
    validator.go              ← validation logic and rules
    hotreload.go              ← hot reload coordination
    config_test.go            ← comprehensive config tests
  configs/
    loadbalancer.json         ← example JSON configuration
    loadbalancer.yaml         ← example YAML configuration
    schema.json               ← JSON schema for validation
  internal/components/
    coordinator.go            ← coordinates config updates across components
```

Configuration Infrastructure Starter Code

Complete Configuration Types (use as-is):

```go
package config

import (
    "encoding/json"
    "fmt"
    "net/url"
    "time"
)

// Config represents the complete load balancer configuration
type Config struct {
    Port        int                `json:"port" yaml:"port"`
    Backends    []BackendConfig    `json:"backends" yaml:"backends"`
    Algorithm   string             `json:"algorithm" yaml:"algorithm"`
    HealthCheck HealthCheckConfig  `json:"healthcheck" yaml:"healthcheck"`
    Proxy       ProxyConfig        `json:"proxy" yaml:"proxy"`
    Logging     LoggingConfig      `json:"logging" yaml:"logging"`
}

// BackendConfig defines a single backend server configuration
type BackendConfig struct {
    URL             string        `json:"url" yaml:"url"`
    Weight          int           `json:"weight" yaml:"weight"`
    MaxConnections  int           `json:"maxConnections" yaml:"maxConnections"`
    ConnectTimeout  time.Duration `json:"connectTimeout" yaml:"connectTimeout"`
    ResponseTimeout time.Duration `json:"responseTimeout" yaml:"responseTimeout"`
    Enabled         bool          `json:"enabled" yaml:"enabled"`
}

// HealthCheckConfig defines health checking parameters
type HealthCheckConfig struct {
    Enabled            bool          `json:"enabled" yaml:"enabled"`
    Interval           time.Duration `json:"interval" yaml:"interval"`
    Timeout            time.Duration `json:"timeout" yaml:"timeout"`
    HealthyThreshold   int           `json:"healthyThreshold" yaml:"healthyThreshold"`
    UnhealthyThreshold int           `json:"unhealthyThreshold" yaml:"unhealthyThreshold"`
    Path               string        `json:"path" yaml:"path"`
    ExpectedStatus     int           `json:"expectedStatus" yaml:"expectedStatus"`
    UserAgent          string        `json:"userAgent" yaml:"userAgent"`
}

// ProxyConfig defines HTTP proxy behavior settings
type ProxyConfig struct {
```

```
    ReadTimeout    time.Duration `json:"readTimeout" yaml:"readTimeout"`
    WriteTimeout   time.Duration `json:"writeTimeout" yaml:"writeTimeout"`
    IdleTimeout    time.Duration `json:"idleTimeout" yaml:"idleTimeout"`
    MaxHeaderBytes int           `json:"maxHeaderBytes" yaml:"maxHeaderBytes"`
}

// LoggingConfig defines logging behavior
type LoggingConfig struct {
    Level  string `json:"level" yaml:"level"`
    Format string `json:"format" yaml:"format"`
    Output string `json:"output" yaml:"output"`
}

// ApplyDefaults fills in default values for missing configuration
func (c *Config) ApplyDefaults() {
    if c.Port == 0 {
        c.Port = 8080
    }
    if c.Algorithm == "" {
        c.Algorithm = "round_robin"
    }
    c.HealthCheck.ApplyDefaults()
    c.Proxy.ApplyDefaults()
    c.Logging.ApplyDefaults()

    for i := range c.Backends {
        c.Backends[i].ApplyDefaults()
    }
}

// ApplyDefaults sets default values for backend configuration
func (b *BackendConfig) ApplyDefaults() {
    if b.Weight <= 0 {
        b.Weight = 1
    }
    if b.MaxConnections <= 0 {
        b.MaxConnections = 100
    }
    if b.ConnectTimeout == 0 {
        b.ConnectTimeout = 5 * time.Second
    }
    if b.ResponseTimeout == 0 {
```

```
        b.ResponseTimeout = 30 * time.Second
    }
    if !b.hasExplicitEnabled {
        b.Enabled = true
    }
}
```

Configuration File Watcher (complete implementation):

```go
package config

import (
    "context"
    "log"
    "os"
    "path/filepath"
    "time"

    "github.com/fsnotify/fsnotify"
)

// FileWatcher monitors configuration files for changes
type FileWatcher struct {
    filename     string
    watcher      *fsnotify.Watcher
    changeNotify chan string
    errorNotify  chan error
}

// NewFileWatcher creates a new file watcher for the specified configuration file
func NewFileWatcher(filename string) (*FileWatcher, error) {
    watcher, err := fsnotify.NewWatcher()
    if err != nil {
        return nil, fmt.Errorf("failed to create file watcher: %w", err)
    }

    // Watch the directory containing the file (some editors replace files)
    dir := filepath.Dir(filename)
    if err := watcher.Add(dir); err != nil {
        watcher.Close()
        return nil, fmt.Errorf("failed to watch directory %s: %w", dir, err)
    }

    return &FileWatcher{
        filename:     filename,
        watcher:      watcher,
        changeNotify: make(chan string, 1),
        errorNotify:  make(chan error, 1),
    }, nil
}
```

```
// Watch starts monitoring for file changes and returns channels for notifications
func (fw *FileWatcher) Watch(ctx context.Context) (<-chan string, <-chan error) {
    go fw.watchLoop(ctx)
    return fw.changeNotify, fw.errorNotify
}

// watchLoop handles file system events and filters for relevant changes
func (fw *FileWatcher) watchLoop(ctx context.Context) {
    defer fw.watcher.Close()
    defer close(fw.changeNotify)
    defer close(fw.errorNotify)

    for {
        select {
        case event, ok := <-fw.watcher.Events:
            if !ok {
                return
            }

            // Filter for events affecting our target file
            if event.Name == fw.filename && (event.Op&fsnotify.Write == fsnotify.Write ||
                                    event.Op&fsnotify.Create == fsnotify.Create) {
                // Debounce rapid successive writes
                fw.debounceNotify()
            }

        case err, ok := <-fw.watcher.Errors:
            if !ok {
                return
            }

            select {
            case fw.errorNotify <- err:
            case <-ctx.Done():
                return
            }

        case <-ctx.Done():
            return
        }
    }
```

```
}


// debounceNotify prevents rapid successive notifications for the same file
func (fw *FileWatcher) debounceNotify() {
    time.Sleep(100 * time.Millisecond) // Wait for write to complete


    select {
    case fw.changeNotify <- fw.filename:
    default:
        // Channel full, change already pending
    }
}
```

Core Configuration Logic Skeleton

Configuration Loader (implement the TODOs):

```go
// LoadConfig loads and validates configuration from the specified file
func LoadConfig(filename string) (*Config, error) {
    // TODO 1: Read the configuration file from disk
    // Hint: Use os.ReadFile(filename) to read the entire file

    // TODO 2: Detect file format based on extension (.json, .yaml, .yml)
    // Hint: Use filepath.Ext(filename) to get file extension

    // TODO 3: Parse the configuration using appropriate parser
    // Hint: Use json.Unmarshal for JSON, yaml.Unmarshal for YAML

    // TODO 4: Apply default values for missing configuration
    // Hint: Call config.ApplyDefaults() to fill in defaults

    // TODO 5: Validate the complete configuration
    // Hint: Call config.Validate() and return any validation errors

    // TODO 6: Return the validated configuration
    return nil, fmt.Errorf("not implemented")
}

// Validate performs comprehensive validation of configuration values
func (c *Config) Validate() error {
    // TODO 1: Validate port number is in valid range (1-65535)

    // TODO 2: Validate that at least one backend is configured

    // TODO 3: Validate algorithm name is supported
    // Hint: Check against known algorithms: round_robin, weighted_round_robin, etc.

    // TODO 4: Validate each backend configuration
    // Hint: Call ValidateBackend for each backend in c.Backends

    // TODO 5: Validate health check configuration consistency
    // Hint: Ensure timeout < interval, thresholds > 0, etc.

    // TODO 6: Validate proxy configuration values
    // Hint: Ensure timeouts are positive, header size limits reasonable

    return nil
}
```

```go
// ValidateBackend validates a single backend server configuration
func (bc *BackendConfig) ValidateBackend() error {
    // TODO 1: Parse and validate backend URL
    // Hint: Use url.Parse(bc.URL) and check for http/https scheme

    // TODO 2: Validate weight is positive

    // TODO 3: Validate connection limits are positive

    // TODO 4: Validate timeouts are positive and reasonable
    // Hint: ConnectTimeout should be less than ResponseTimeout

    return nil
}
```

Hot Reload Coordinator (implement the TODOs):

GO

```go
// HotReloadManager coordinates configuration updates across all components
type HotReloadManager struct {
    currentConfig  *Config
    configMutex    sync.RWMutex
    updateChannel  chan *Config
    components     []ConfigurableComponent
}

// ConfigurableComponent interface for components that can be reconfigured
type ConfigurableComponent interface {
    UpdateConfiguration(*Config) error
    RollbackConfiguration(*Config) error
}

// ApplyConfiguration attempts to update all components with new configuration
func (hrm *HotReloadManager) ApplyConfiguration(newConfig *Config) error {
    // TODO 1: Validate the new configuration before applying
    // Hint: Call newConfig.Validate() and reject if invalid

    // TODO 2: Acquire write lock to prevent concurrent updates
    // Hint: Use hrm.configMutex.Lock() and defer hrm.configMutex.Unlock()

    // TODO 3: Store current configuration for potential rollback
    // Hint: Keep a reference to hrm.currentConfig before changing it

    // TODO 4: Attempt to update each component with new configuration
    // Hint: Iterate through hrm.components and call UpdateConfiguration

    // TODO 5: If any component fails, rollback all successful updates
    // Hint: Call RollbackConfiguration on all previously updated components

    // TODO 6: If all components succeed, commit the new configuration
    // Hint: Set hrm.currentConfig = newConfig

    return fmt.Errorf("not implemented")
}

// RegisterComponent adds a component to receive configuration updates
func (hrm *HotReloadManager) RegisterComponent(component ConfigurableComponent) {
    // TODO: Add component to hrm.components slice with thread safety
}
```

Language-Specific Implementation Hints

Go-Specific Configuration Tips:

- Use `encoding/json` for JSON parsing and `gopkg.in/yaml.v3` for YAML support
- Implement custom `UnmarshalJSON` methods for duration fields to support human-readable formats like "30s", "5m"
- Use `sync/atomic` for configuration pointer swapping to enable lock-free reads during updates
- Use `context.Context` for cancellable file watching and configuration update operations
- Leverage Go's zero values for optional configuration fields and document the behavior clearly

Validation Implementation Patterns:

```go
// Example custom duration unmarshaling for human-readable timeouts
func (d *Duration) UnmarshalJSON(b []byte) error {
    var v interface{}
    if err := json.Unmarshal(b, &v); err != nil {
        return err
    }

    switch value := v.(type) {
    case float64:
        *d = Duration(time.Duration(value) * time.Second)
    case string:
        duration, err := time.ParseDuration(value)
        if err != nil {
            return err
        }
        *d = Duration(duration)
    default:
        return fmt.Errorf("invalid duration format")
    }

    return nil
}
```

Milestone Checkpoints

Configuration Loading Validation:

1. Create a test configuration file with valid settings
2. Run `go test ./internal/config/...` to verify loading works correctly
3. Create an invalid configuration and verify proper error handling
4. Expected behavior: Valid configs load successfully, invalid configs are rejected with clear error messages

Hot Reload Testing:

1. Start the load balancer with initial configuration
2. Modify the configuration file while the service is running
3. Send a SIGHUP signal or trigger file watching
4. Verify that configuration changes take effect without dropping connections
5. Expected behavior: Service continues operating with updated settings, no connection drops

Configuration Validation Testing:

1. Test various invalid configurations (negative timeouts, invalid URLs, etc.)
2. Verify that specific validation errors are returned for each case
3. Test edge cases like empty backend lists, algorithm typos
4. Expected behavior: Each invalid configuration is rejected with a specific, actionable error message

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Configuration changes ignored	File watcher not working	Check file system event logs	Verify file permissions, use polling fallback
Service crashes on config update	Invalid configuration applied	Check validation logic	Implement proper validation before applying
Inconsistent behavior after update	Partial configuration application	Check component update logs	Implement atomic updates with rollback
Memory usage increasing	Configuration objects not cleaned up	Monitor memory usage during updates	Implement proper cleanup in configuration transitions
Hot reload not working	File watcher events not triggering	Test file modification detection	Check file editor behavior, some create new files instead of modifying

Error Handling and Edge Cases

> **Milestone(s):** All milestones (1-4) — comprehensive error handling is essential throughout the project, from basic HTTP proxy errors (Milestone 1) through request distribution failures (Milestone 2), health check edge cases (Milestone 3), and algorithm-specific error scenarios (Milestone 4)

Mental Model: Emergency Response System

Think of the load balancer's error handling like a hospital's emergency response system. Just as a hospital has protocols for different types of emergencies — cardiac arrest requires immediate CPR, while a broken bone needs stabilization and transport — the load balancer must recognize different failure patterns and respond appropriately. The triage nurse (error classifier) quickly assesses each situation, the emergency protocols (graceful degradation) keep critical systems running even when some equipment fails, and the circuit breaker acts like an automatic fire suppression system that stops spreading damage before it can cascade throughout the facility.

The key insight is that not all failures are created equal. A single backend timing out is like a patient with a minor injury — handle it quietly and move on. But when multiple backends start failing simultaneously, it's like a mass casualty event requiring coordinated response to prevent the entire system from collapse.

Failure Mode Analysis

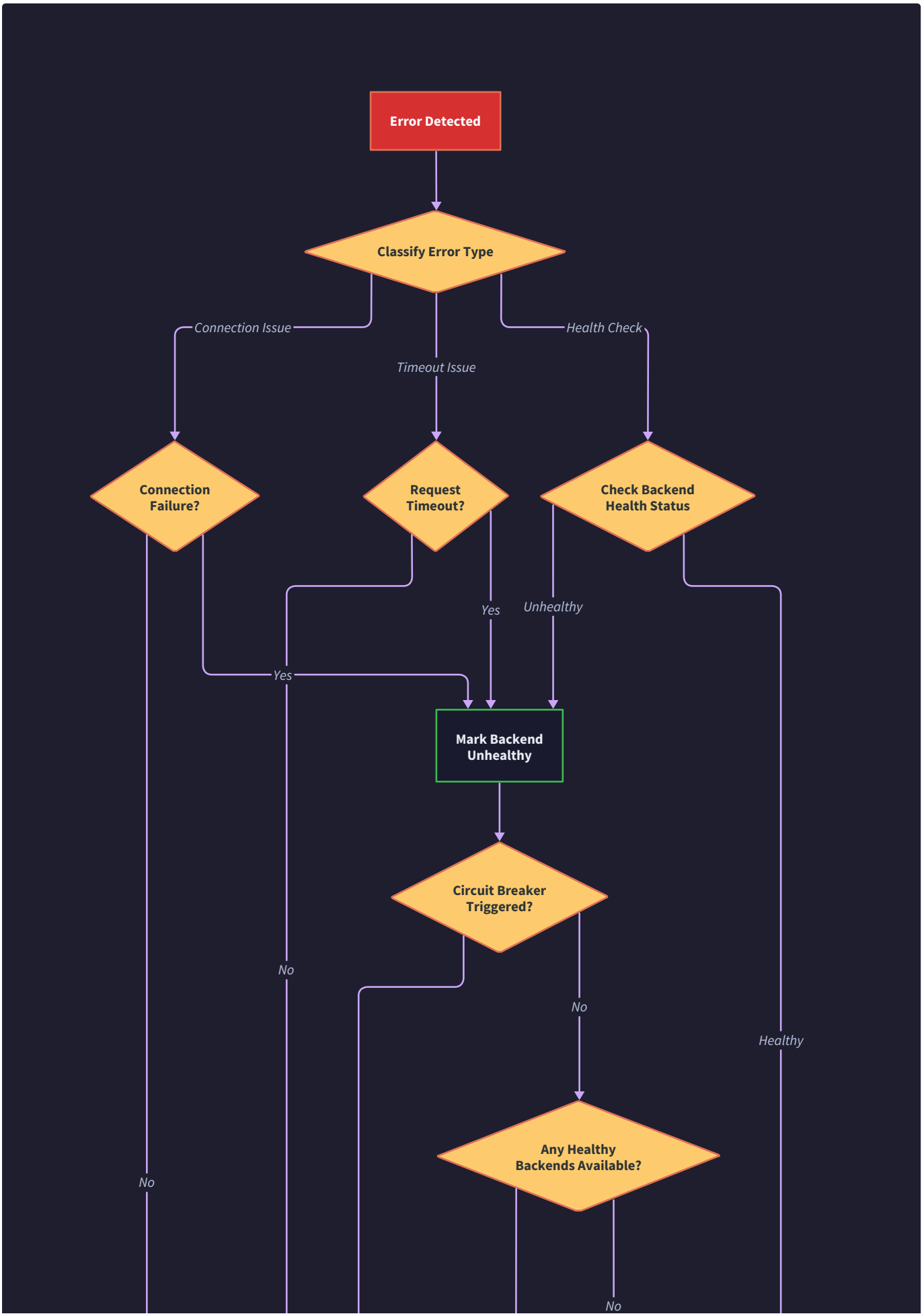
The load balancer operates in a complex environment where failures can occur at multiple levels and cascade in unpredictable ways. Understanding the complete spectrum of possible failures allows the system to respond appropriately rather than treating all errors as equivalent catastrophes.

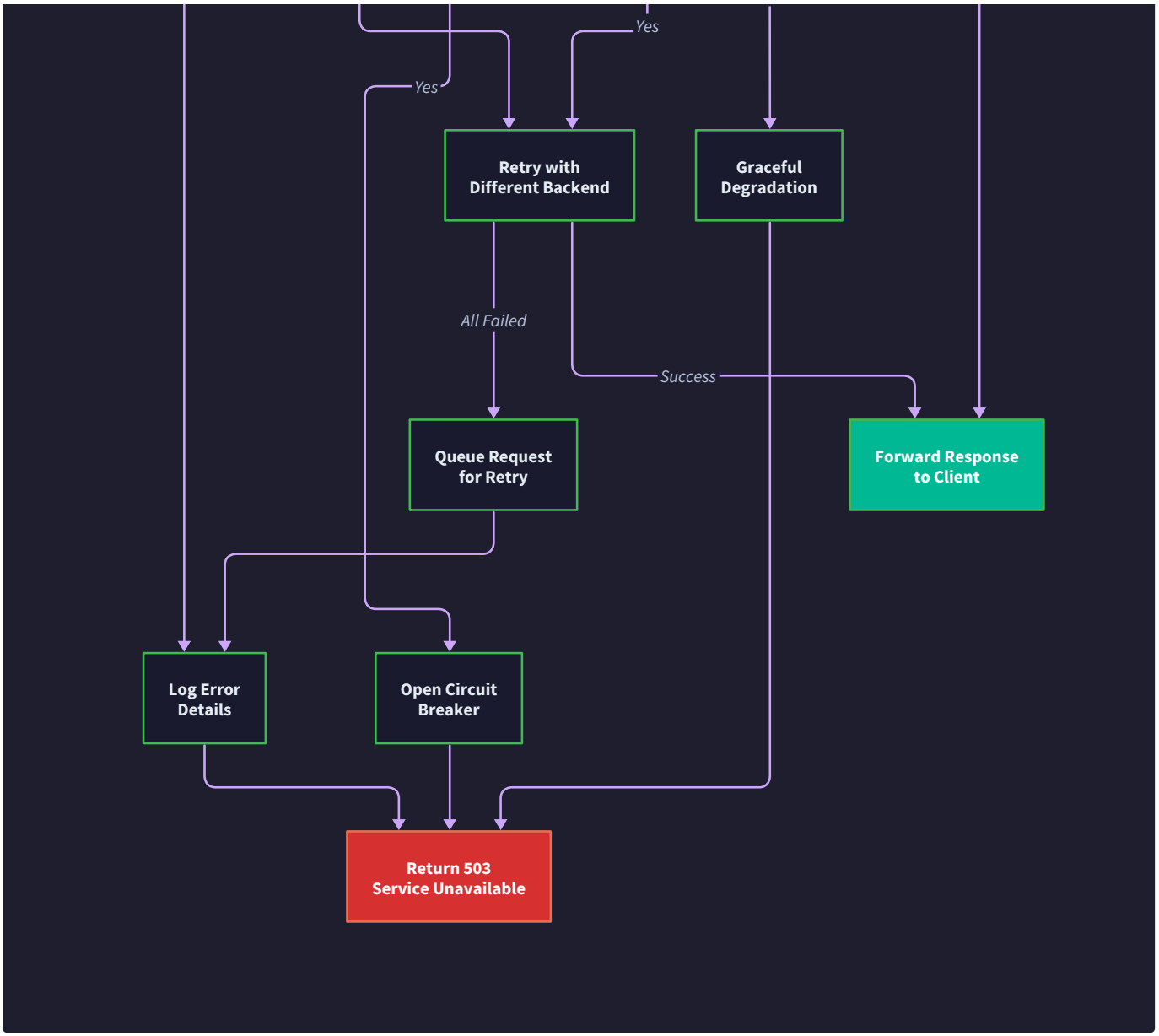
Network-Level Failures represent the most common category of issues the load balancer encounters. Connection failures occur when the load balancer cannot establish TCP connections to backend servers, typically manifesting as connection refused errors when backends are down or connection timeout errors when backends are overloaded but still accepting connections slowly. DNS resolution failures can occur when backend hostnames cannot be resolved, either due to DNS server issues or misconfigured backend addresses. Network partitions represent more severe scenarios where network connectivity exists but is unreliable, leading to intermittent failures that can be particularly challenging to diagnose.

Backend Server Failures encompass issues originating from the application servers themselves. Complete backend unavailability occurs when servers crash, are shut down for maintenance, or become completely unresponsive. Partial backend degradation happens when servers are running but experiencing high load, returning errors for some requests while successfully processing others. Application-level errors manifest as HTTP 5xx responses from backends that are otherwise healthy, indicating internal application issues rather than infrastructure problems. Resource exhaustion on backends can cause them to reject new connections or respond with errors even when the load balancer's health checks pass.

Load Balancer Internal Failures can critically impact the entire system's operation. Algorithm selection failures occur when the chosen load balancing algorithm cannot select a backend, typically because no healthy backends are available or algorithm state has become corrupted. Health checker failures can leave the system with stale health information, potentially routing traffic to failed backends or excluding healthy ones from rotation. Configuration errors can cause the load balancer to operate with invalid backend lists, incorrect timeouts, or malformed routing rules.

Resource Exhaustion Scenarios affect the load balancer's ability to handle incoming requests. Connection pool exhaustion occurs when all available connections to backends are in use, preventing new requests from being processed. Memory exhaustion can cause the load balancer itself to become unstable or crash. File descriptor exhaustion prevents new connections from being established either from clients or to backends.


Error Detected
Classify Error Type
Connection Issue
Timeout Issue
Health Check
Connection Failure?
Request Timeout?
Check Backend Health Status
Yes
Unhealthy
Yes
Mark Backend Unhealthy
Circuit Breaker Triggered?
No
No
Healthy
Any Healthy Backends Available?
No
No



The following table catalogs the primary failure modes, their detection mechanisms, and immediate impact on system operation:

Failure Mode	Detection Method	System Impact	Time to Detect	Recovery Complexity
Backend Connection Refused	TCP connection error	Request fails immediately	< 1 second	Low - automatic retry
Backend Connection Timeout	Connect timeout expires	Request delayed then fails	5-30 seconds	Medium - may indicate overload
Backend Response Timeout	Read timeout expires	Request fails after delay	30-120 seconds	Medium - backend may be struggling
DNS Resolution Failure	Hostname lookup error	All requests to backend fail	1-5 seconds	High - infrastructure issue
Backend Returns 5xx Error	HTTP status code	Request fails with error	< 1 second	Low - application level
All Backends Unhealthy	Health check failures	Service completely unavailable	30-180 seconds	High - requires manual intervention
Health Checker Failure	Health check process crash	Stale health information	Varies	High - monitoring blind spot
Algorithm State Corruption	Selection returns null/error	Request routing fails	Immediate	Medium - algorithm reset needed
Load Balancer Memory Exhaustion	Memory monitoring alerts	Performance degradation/crash	Minutes	High - requires restart
Configuration File Corruption	Config parsing errors	Cannot reload configuration	Immediate	Medium - fix config and reload

Cascade Failure Scenarios represent the most dangerous category where initial failures trigger additional failures throughout the system. The classic scenario begins when one backend fails, increasing load on remaining backends, which may cause additional backends to fail under the increased pressure, eventually leading to complete service unavailability. Health check storms can occur when many backends fail simultaneously and then recover, causing a thundering herd of

traffic that immediately overwhelms the recovering backends. Client retry storms happen when the load balancer starts returning errors, causing upstream clients to retry requests, multiplying the load on an already stressed system.

Timing-Related Failures introduce subtle bugs that can be difficult to reproduce and diagnose. Race conditions in algorithm state updates can cause inconsistent backend selection or connection counting. Health check timing issues can cause backends to flap between healthy and unhealthy states. Configuration update races can leave the system in inconsistent states where different components are operating with different configuration versions.

> The most critical insight for failure analysis is that the load balancer sits at a convergence point where failures from many different systems and layers can manifest. A single client request might fail due to DNS issues, network problems, backend overload, application bugs, or load balancer resource exhaustion. The error handling system must be sophisticated enough to distinguish between these different failure classes and respond appropriately to each.

Graceful Degradation Strategies

Graceful degradation ensures the load balancer continues providing service even when operating under adverse conditions. The fundamental principle is to maintain the highest possible level of service availability by sacrificing non-essential functionality when resources become constrained or components fail.

Adaptive Backend Selection represents the first line of defense against backend failures. When the health checking system identifies failed backends, the load balancer immediately excludes them from the selection pool, ensuring that client requests are only routed to backends capable of handling them. However, this simple approach can create problems if implemented naively. When backend capacity is reduced, the load balancer must account for the increased load on remaining backends and potentially implement backpressure mechanisms to prevent cascade failures.

The system implements a tiered backend classification approach rather than the binary healthy/unhealthy model. Backends are classified as: Primary (fully healthy, normal response times), Secondary (healthy but experiencing elevated response times or error rates), Degraded (functional but unreliable, use only if no better options), and Failed (completely unavailable). This classification allows the algorithm to prefer better backends while still utilizing degraded ones when necessary.

Request Admission Control provides protection against overwhelming the system during high load or reduced backend capacity. Rather than accepting all incoming requests and allowing them to fail at backend selection time, the load balancer can implement admission control that rejects requests early when system capacity is constrained. This approach is counterintuitive but essential — by rejecting some requests immediately with proper error responses, the system can successfully process more requests than if it attempted to handle all incoming traffic.

The admission control mechanism monitors several key metrics: current backend capacity (number of healthy backends multiplied by their individual capacity), request queue depth, average response time, and error rate trends. When these metrics indicate the system is approaching overload, admission control begins rejecting a percentage of incoming requests, starting with the least critical traffic if request prioritization is configured.

Degraded Mode Operations allow the load balancer to continue functioning even when supporting systems fail. When the health checking system fails, the load balancer can operate in static mode using the last known good health states, gradually reducing confidence in this information over time. If configuration hot reloading fails, the system continues operating with the current configuration while logging errors and attempting periodic reload retries.

During degraded operations, the system reduces functionality rather than failing completely. Complex algorithms like least connections may fall back to simpler round-robin selection if connection tracking becomes unreliable. Detailed request logging might be disabled to reduce I/O load. Non-essential monitoring endpoints may return simplified responses to reduce processing overhead.

Circuit Breaker Integration prevents the load balancer from repeatedly attempting operations that are likely to fail, reducing resource waste and improving response times. Each backend has an associated circuit breaker that monitors error rates and response times. When error thresholds are exceeded, the circuit breaker opens, immediately failing requests to that backend without attempting connection. This prevents the load balancer from wasting time and resources on backends that are known to be failing.

The circuit breaker implements three states: Closed (normal operation), Open (failing fast), and Half-Open (testing recovery). In the closed state, requests proceed normally while the circuit breaker monitors error rates. When failure thresholds are exceeded, the circuit opens, immediately returning errors for all requests to that backend. After a configured timeout period, the circuit enters half-open state, allowing a limited number of test requests through. If these succeed, the circuit closes and normal operation resumes. If they fail, the circuit reopens for another timeout period.

Partial Service Availability ensures that even when some functionality is unavailable, the core request routing capability continues operating. If the health checking system fails completely, the load balancer can continue routing requests using the last known backend health states, with appropriate warnings in responses indicating degraded operation. If specific load balancing algorithms fail due to state corruption, the system can fall back to simpler algorithms that require less state management.

The following table outlines the degradation strategies for different failure scenarios:

Failure Scenario	Degradation Strategy	Functionality Retained	Functionality Lost	Recovery Trigger
Single Backend Failed	Exclude from rotation	Full service with reduced capacity	None - transparent to clients	Backend health recovery
Multiple Backends Failed	Admission control + reduced capacity	Core routing with selective rejection	Full request handling guarantee	Backend capacity restoration
Health Checker Failed	Static health state + gradual degradation	Request routing with stale data	Automatic failure detection	Health checker restart
Primary Algorithm Failed	Fallback to simple round-robin	Basic load distribution	Advanced algorithm features	Algorithm state recovery
Configuration System Failed	Continue with last known config	Stable operation	Configuration updates	Config system restoration
Partial Network Connectivity	Route to reachable backends only	Service to accessible regions	Global load distribution	Network connectivity restoration

> The key insight for graceful degradation is that availability is more valuable than perfection. A load balancer that serves 80% of requests successfully is infinitely better than one that fails completely while trying to maintain 100% functionality. The art lies in determining which 80% to serve and how to fail gracefully for the remaining 20%.

Progressive Degradation Levels provide a structured approach to reducing functionality as conditions worsen. Level 0 represents normal operation with all features enabled. Level 1 disables non-essential features like detailed metrics collection and verbose logging. Level 2 simplifies algorithms and reduces health check frequency. Level 3 implements admission control and disables advanced routing features. Level 4 represents minimal functionality with only basic round-robin distribution and static backend lists.

Each degradation level has clear entry and exit criteria based on system health metrics. The system automatically transitions between levels based on conditions, but also provides manual override capabilities for operational control during maintenance or emergency situations.

Circuit Breaker Pattern

The circuit breaker pattern provides a critical protection mechanism that prevents the load balancer from repeatedly attempting operations that are likely to fail, while also providing a pathway for automatic recovery when conditions improve. Unlike simple timeout mechanisms that wait for each request to fail individually, circuit breakers learn from failure patterns and proactively prevent resource waste.

Circuit Breaker State Management implements a state machine with three distinct states, each optimized for different failure scenarios. The Closed state represents normal operation where requests flow through to backends normally, but the circuit breaker continuously monitors error rates, response times, and other health indicators. Statistical tracking includes rolling windows of success/failure counts, recent response time percentiles, and consecutive failure streaks.

When failure thresholds are exceeded in the Closed state, the circuit immediately transitions to Open state. In the Open state, all requests to the affected backend fail immediately with circuit breaker errors, preventing resource waste on operations that are likely to fail. This provides immediate feedback to clients while protecting backend servers from additional load that might prevent their recovery. The circuit breaker maintains the Open state for a configured timeout period, allowing the backend time to recover without additional request pressure.

After the timeout period expires, the circuit enters Half-Open state, which serves as a careful testing phase for backend recovery. In Half-Open state, the circuit breaker allows a limited number of test requests through to the backend while continuing to fail-fast for all other requests. If the test requests succeed, indicating the backend has recovered, the circuit transitions back to Closed state and normal operation resumes. If the test requests fail, the circuit immediately returns to Open state for another timeout period.

Per-Backend Circuit Breaker Configuration recognizes that different backends may have different failure characteristics and recovery patterns. High-capacity backends might tolerate higher error rates before circuit breaking, while smaller backends might need more aggressive protection. Database backends might have longer recovery times requiring extended Open state timeouts, while stateless web services might recover quickly.

The circuit breaker configuration includes several key parameters: failure rate threshold (percentage of requests that must fail to open the circuit), minimum request count (circuits don't open until sufficient data is available), timeout duration (how long to stay in Open state), half-open request count (number of test requests in recovery phase), and monitoring window duration (time period for calculating failure rates).

```
Circuit Breaker State Transitions:
Closed → Open: failure_rate > threshold AND request_count >= minimum
Open → Half-Open: timeout_duration elapsed
Half-Open → Closed: test_requests_successful >= required_successes
Half-Open → Open: any_test_request_failed
```

Integration with Health Checking creates a comprehensive backend protection system where circuit breakers provide immediate failure response while health checks provide longer-term backend monitoring. Circuit breakers react to request-level failures within seconds, while health checks monitor backend availability over minutes. The two systems complement each other — circuit breakers prevent immediate damage from failed requests, while health checks provide the authoritative backend health state for load balancing decisions.

When a circuit breaker opens for a backend, this information influences but doesn't override health check status. A backend might be healthy according to health checks but have an open circuit breaker due to high error rates under load. Conversely, a backend marked unhealthy by health checks will have its circuit breaker opened to prevent any test traffic until health recovery is confirmed.

Circuit Breaker Metrics and Observability provide essential visibility into failure patterns and system protection effectiveness. Key metrics include circuit state transitions over time, failure rates leading to circuit opening, recovery success rates, and request volume reduction during circuit open periods. These metrics help operators understand system behavior and tune circuit breaker parameters for optimal protection.

The circuit breaker maintains detailed logs of state transitions including timestamps, triggering conditions, and relevant metrics at transition time. This information proves invaluable for post-incident analysis and system tuning. Circuit breaker dashboards should display current state for all backends, recent state transition history, and trending metrics that might indicate approaching threshold breaches.

Advanced Circuit Breaker Patterns extend the basic three-state model to handle more complex failure scenarios. Adaptive thresholds adjust failure rate requirements based on overall system load and backend capacity. During high load periods, circuits might tolerate higher error rates before opening, recognizing that some failures are inevitable under stress.

Cascading circuit breakers coordinate between related backends to prevent failure spread. When a primary backend's circuit opens, related backup backends might automatically reduce their failure thresholds, recognizing that they're likely to experience increased load and potential secondary failures.

The following table describes circuit breaker behavior for different failure patterns:

Failure Pattern	Circuit Breaker Response	Protection Provided	Recovery Strategy	Monitoring Focus
Intermittent Errors	Remains closed until threshold	None - errors passed through	Automatic - no action needed	Error rate trending
Steady High Error Rate	Opens after threshold breach	Immediate fail-fast	Half-open testing after timeout	Error rate stability
Complete Backend Failure	Opens immediately on 100% errors	Full protection from dead backend	Coordinated with health checks	Backend recovery signals
Slow Response Times	Opens based on timeout thresholds	Prevents request queuing	Response time improvement	Latency percentiles
Resource Exhaustion	Opens on connection/resource errors	Prevents resource waste	Backend capacity restoration	Resource utilization metrics
Cascade Failure	Coordinated opening across backends	System-wide protection	Staged recovery testing	Cross-backend correlation

> The circuit breaker pattern embodies the principle of failing fast and recovering gracefully. By quickly detecting failure patterns and stopping wasteful operations, circuit breakers not only protect the load balancer's resources but often help backends recover more quickly by reducing the load pressure that contributed to their failure.

Circuit Breaker Implementation Considerations address the practical challenges of implementing effective circuit breaking in a high-throughput load balancer. Thread safety becomes critical when multiple concurrent requests might simultaneously detect failure conditions and attempt to change circuit state. The implementation uses atomic operations for state changes and careful synchronization around shared data structures.

Memory management for circuit breaker statistics requires attention to prevent memory leaks from accumulating failure data. Rolling windows for error rate calculation must efficiently add new data points while discarding old ones. The implementation uses circular buffers and periodic cleanup to maintain bounded memory usage even during extended operation periods.

Integration with load balancing algorithms requires careful coordination to ensure circuit breaker state changes are immediately reflected in backend selection decisions. When a circuit opens, the backend must be immediately excluded from algorithm consideration. When circuits close, backends should be gradually reintroduced rather than immediately receiving full traffic allocation.

Common Pitfalls

⚠️ **Pitfall: Treating All Errors Equally** Many load balancer implementations handle all errors with the same generic "502 Bad Gateway" response, losing valuable diagnostic information. A DNS resolution failure indicates a configuration problem requiring immediate attention, while a backend timeout might indicate normal load fluctuation. The error handling system must classify errors appropriately and generate responses that help operators and clients understand the underlying issue. Implement specific error classification logic that examines error types, sources, and contexts to provide meaningful error responses.

⚠️ **Pitfall: Circuit Breaker Thundering Herd** When multiple circuit breakers transition from Open to Half-Open simultaneously, the resulting test traffic can immediately overwhelm recovering backends, causing them to fail again and preventing successful recovery. This occurs when multiple backends fail and recover simultaneously, or when circuit breaker timeouts are configured with identical values. Implement jittered timeout values and coordinated recovery strategies that stagger test traffic across time and backends.

⚠️ **Pitfall: Health Check Masking Circuit Breaker State** Health checks that only test basic connectivity may report backends as healthy even when they're experiencing high error rates under load, leading to circuit breaker conflicts where backends appear healthy but circuits remain open. Design health checks to include load testing elements that simulate real request patterns, and ensure circuit breaker state influences backend availability decisions even when health checks pass.

⚠️ **Pitfall: Insufficient Error Context Propagation** Error responses that don't include correlation IDs, backend identifiers, or failure timestamps make debugging nearly impossible in distributed environments. When a request fails, operators need to understand which backend failed, what type of failure occurred, and how to correlate the failure with logs and monitoring systems. Implement comprehensive error context that includes request correlation IDs, backend selection details, failure classifications, and timestamps.

⚠️ **Pitfall: Cascading Timeout Configuration** When load balancer timeouts are longer than client timeouts, clients give up and retry while the load balancer continues processing the original request, leading to resource waste and duplicate processing. Conversely, when load balancer timeouts are too short compared to backend processing times, requests that would eventually succeed are prematurely failed. Carefully configure timeout hierarchies where each layer has appropriate timeout values that account for downstream processing times and expected client behavior.

⚠️ **Pitfall: Memory Leaks in Error Tracking** Error tracking systems that accumulate failure statistics without bounds can consume unbounded memory over time, especially in high-traffic environments with frequent failures. Circuit breaker statistics, error rate calculations, and failure logs must implement proper cleanup and bounded data structures. Use rolling windows with fixed sizes, periodic cleanup of old error data, and memory usage monitoring to prevent error tracking from becoming a source of failures.

Implementation Guidance

The error handling implementation requires careful integration with all load balancer components while maintaining clear separation of concerns and comprehensive failure coverage.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Error Classification	Custom error type hierarchy	Structured error codes with metadata
Circuit Breaker	Simple state machine with timers	Library with advanced patterns (sony/gobreaker)
Metrics Collection	Basic counters and gauges	Full observability with Prometheus client
Error Logging	Standard log levels with context	Structured logging with correlation IDs
Configuration	Static configuration files	Dynamic configuration with validation

B. Recommended File/Module Structure:

```
project-root/
  internal/errors/
    classifier.go           ← error classification and handling
    circuit_breaker.go      ← circuit breaker implementation
    degradation.go          ← graceful degradation logic
    responses.go            ← standardized error responses
  internal/metrics/
    collector.go            ← metrics collection for error tracking
  internal/router/
    error_handler.go        ← integration with request router
  cmd/server/
    main.go                 ← error handling initialization
```

C. Infrastructure Starter Code:

```go
// Package errors provides comprehensive error handling for the load balancer
package errors

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "sync/atomic"
    "time"
)

// ErrorType represents different categories of errors for appropriate handling
type ErrorType string

const (
    ErrorTypeConnection     ErrorType = "connection"
    ErrorTypeTimeout        ErrorType = "timeout"
    ErrorTypeDNS            ErrorType = "dns"
    ErrorTypeBackend        ErrorType = "backend"
    ErrorTypeInternal       ErrorType = "internal"
    ErrorTypeCapacity       ErrorType = "capacity"
)

// ErrorClassifier determines error types and appropriate responses
type ErrorClassifier struct {
    mu sync.RWMutex
    classificationRules map[string]ErrorType
}

func NewErrorClassifier() *ErrorClassifier {
    return &ErrorClassifier{
        classificationRules: map[string]ErrorType{
            "connection refused": ErrorTypeConnection,
            "timeout":            ErrorTypeTimeout,
            "no such host":       ErrorTypeDNS,
            "context deadline":   ErrorTypeTimeout,
        },
    }
}

// ClassifyError determines the appropriate error type and response
```

```go
func (ec *ErrorClassifier) ClassifyError(err error) ErrorType {
    if err == nil {
        return ErrorTypeInternal
    }

    errStr := err.Error()
    ec.mu.RLock()
    defer ec.mu.RUnlock()

    for pattern, errorType := range ec.classificationRules {
        if contains(errStr, pattern) {
            return errorType
        }
    }
    return ErrorTypeInternal
}

// CircuitState represents the current state of a circuit breaker
type CircuitState int32

const (
    CircuitClosed CircuitState = iota
    CircuitOpen
    CircuitHalfOpen
)

// CircuitBreaker implements the circuit breaker pattern for backend protection
type CircuitBreaker struct {
    mu               sync.RWMutex
    state            int32 // CircuitState stored as atomic int32
    failureCount     int64
    requestCount     int64
    lastFailureTime  time.Time
    lastStateChange  time.Time

    // Configuration
    failureThreshold int64
    timeout          time.Duration
    maxRequests      int64 // Half-open state request limit
}

func NewCircuitBreaker(failureThreshold int64, timeout time.Duration) *CircuitBreaker {
```

```
    return &CircuitBreaker{
        failureThreshold: failureThreshold,
        timeout:         timeout,
        maxRequests:     10,
        lastStateChange: time.Now(),
    }
}

// Allow determines if a request should be allowed through the circuit
func (cb *CircuitBreaker) Allow() bool {
    state := CircuitState(atomic.LoadInt32(&cb.state))
    now := time.Now()

    switch state {
    case CircuitClosed:
        return true
    case CircuitOpen:
        if now.Sub(cb.lastStateChange) > cb.timeout {
            cb.setState(CircuitHalfOpen)
            return true
        }
        return false
    case CircuitHalfOpen:
        return atomic.LoadInt64(&cb.requestCount) < cb.maxRequests
    }
    return false
}

// RecordSuccess records a successful request through the circuit
func (cb *CircuitBreaker) RecordSuccess() {
    atomic.AddInt64(&cb.requestCount, 1)

    state := CircuitState(atomic.LoadInt32(&cb.state))
    if state == CircuitHalfOpen {
        cb.setState(CircuitClosed)
        atomic.StoreInt64(&cb.failureCount, 0)
        atomic.StoreInt64(&cb.requestCount, 0)
    }
}

// RecordFailure records a failed request through the circuit
func (cb *CircuitBreaker) RecordFailure() {
```

```
    atomic.AddInt64(&cb.requestCount, 1)
    failures := atomic.AddInt64(&cb.failureCount, 1)

    cb.mu.Lock()
    cb.lastFailureTime = time.Now()
    cb.mu.Unlock()

    state := CircuitState(atomic.LoadInt32(&cb.state))
    if state == CircuitHalfOpen {
        cb.setState(CircuitOpen)
    } else if failures >= cb.failureThreshold {
        cb.setState(CircuitOpen)
    }
}

func (cb *CircuitBreaker) setState(newState CircuitState) {
    atomic.StoreInt32(&cb.state, int32(newState))
    cb.mu.Lock()
    cb.lastStateChange = time.Now()
    cb.mu.Unlock()
}

// Helper function for string matching
func contains(s, substr string) bool {
    return len(s) >= len(substr) &&
           (s == substr ||
            (len(s) > len(substr) &&
             (s[:len(substr)] == substr ||
              s[len(s)-len(substr):] == substr ||
              indexString(s, substr) >= 0)))
}

func indexString(s, substr string) int {
    for i := 0; i <= len(s)-len(substr); i++ {
        if s[i:i+len(substr)] == substr {
            return i
        }
    }
    return -1
}
```

D. Core Logic Skeleton Code:

```go
// ErrorHandler provides comprehensive error handling for the load balancer
type ErrorHandler struct {
    classifier    *ErrorClassifier
    circuitBreakers map[string]*CircuitBreaker
    degradationLevel int32
    metrics         *MetricsCollector
    mu              sync.RWMutex
}

// HandleError processes errors and determines appropriate response actions
func (eh *ErrorHandler) HandleError(ctx context.Context, err error, backend *Backend) *ErrorResponse {
    // TODO 1: Use ErrorClassifier to determine the error type
    // TODO 2: Check if this error should trigger circuit breaker state change
    // TODO 3: Determine if graceful degradation level should be adjusted
    // TODO 4: Generate appropriate ErrorResponse with proper status code
    // TODO 5: Update metrics for error tracking and monitoring
    // TODO 6: Log error with correlation ID and backend information
    // Hint: Different error types require different HTTP status codes
    panic("TODO: Implement comprehensive error handling logic")
}

// CheckCircuitBreaker verifies if requests to a backend should be allowed
func (eh *ErrorHandler) CheckCircuitBreaker(backendID string) bool {
    // TODO 1: Get or create circuit breaker for the backend
    // TODO 2: Call Allow() method on the circuit breaker
    // TODO 3: Update metrics for circuit breaker state
    // TODO 4: Return whether request should proceed
    // Hint: Circuit breakers are per-backend, create them lazily
    panic("TODO: Implement circuit breaker checking")
}

// RecordResult updates circuit breaker state based on request outcome
func (eh *ErrorHandler) RecordResult(backendID string, success bool, duration time.Duration) {
    // TODO 1: Get circuit breaker for the backend
    // TODO 2: Call RecordSuccess or RecordFailure based on outcome
    // TODO 3: Update response time metrics for the backend
    // TODO 4: Check if degradation level should be adjusted based on overall health
    // Hint: Success/failure affects both circuit breaker and degradation decisions
    panic("TODO: Implement result recording logic")
}

// DetermineGracefulDegradation calculates appropriate system degradation level
```

```
func (eh *ErrorHandler) DetermineGracefulDegradation(healthyBackends, totalBackends int) int {
    // TODO 1: Calculate percentage of healthy backends
    // TODO 2: Determine appropriate degradation level (0-4) based on capacity
    // TODO 3: Consider recent error rates and circuit breaker states
    // TODO 4: Update degradation level atomically if change is needed
    // TODO 5: Log degradation level changes with reasoning
    // Hint: Level 0 = normal, Level 4 = minimal functionality
    panic("TODO: Implement graceful degradation level calculation")
}

// GenerateErrorResponse creates appropriate HTTP error response for clients
func (eh *ErrorHandler) GenerateErrorResponse(errorType ErrorType, backendID string, requestID string) *ErrorResponse {
    // TODO 1: Map error type to appropriate HTTP status code
    // TODO 2: Create descriptive error message without exposing internal details
    // TODO 3: Include correlation ID for debugging
    // TODO 4: Add appropriate headers (Retry-After for capacity errors)
    // TODO 5: Ensure response format is consistent and parseable
    // Hint: Don't expose internal backend details in client-facing errors
    panic("TODO: Implement error response generation")
}
```

E. Language-Specific Hints:

- Use `sync/atomic` for thread-safe counter operations in circuit breakers and metrics
- Implement proper cleanup with `context.Context` cancellation for long-running error tracking
- Use `time.Time` consistently for all timestamp operations and duration calculations
- Leverage Go's error interface for clean error type classification and handling
- Use `sync.RWMutex` for read-heavy data structures like circuit breaker configuration
- Implement proper JSON marshaling for error responses with `encoding/json` struct tags
- Use buffered channels for async error logging to prevent blocking request processing

F. Milestone Checkpoint:

After implementing comprehensive error handling, verify the system correctly handles various failure scenarios:

Testing Connection Failures:

1. Start the load balancer with multiple backends
2. Stop one backend server and send requests
3. Verify requests are routed to healthy backends only
4. Check that 502 responses are returned when no backends available
5. Confirm circuit breaker opens after consecutive failures

Testing Graceful Degradation:

1. Configure load balancer with 4 backends
2. Stop backends one by one while sending steady traffic
3. Verify degradation levels increase as capacity decreases
4. Confirm admission control activates at higher degradation levels
5. Check that system maintains core functionality even with minimal capacity

Testing Circuit Breaker Recovery:

1. Cause backend to fail and verify circuit opens
2. Fix backend and wait for circuit timeout

3. Verify circuit enters half-open state and allows test requests
4. Confirm circuit closes after successful test requests
5. Check that traffic gradually returns to recovered backend

Expected log output should include error classifications, circuit breaker state changes, degradation level transitions, and correlation IDs for request tracking.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
All requests return 502	No healthy backends available	Check backend health status and network connectivity	Fix backend issues or verify health check configuration
Circuit breaker never opens	Failure threshold too high or errors not recorded	Check failure rate calculation and threshold values	Adjust threshold or fix error recording logic
Backends never recover	Circuit timeout too short or recovery logic broken	Monitor circuit state transitions and test request success	Increase timeout or fix half-open state handling
Memory usage grows continuously	Error tracking data not cleaned up	Monitor error data structures and cleanup routines	Implement proper cleanup with bounded data structures
Inconsistent error responses	Race conditions in error handling	Check concurrent access patterns and synchronization	Add proper locking around shared error state
Degradation level incorrect	Metrics calculation wrong or thresholds misconfigured	Verify healthy backend counts and degradation logic	Fix metrics or adjust degradation thresholds

Testing Strategy

> **Milestone(s):** All milestones (1-4) — testing strategy supports validation of HTTP proxy functionality (Milestone 1), request distribution algorithms (Milestone 2), health checking systems (Milestone 3), and multiple load balancing algorithms (Milestone 4)

Mental Model: Quality Assurance Assembly Line

Think of testing a load balancer like a multi-stage quality assurance assembly line in a manufacturing plant. Just as a car manufacturer tests individual components (engine, brakes, transmission) in isolation before integrating them into a complete vehicle for road testing, we must validate load balancer components independently before testing the entire system end-to-end.

The **unit testing** stage resembles testing individual car parts on specialized benches — we verify each algorithm, health checker, and proxy component works correctly in controlled conditions with predictable inputs. The **integration testing** stage is like road testing the complete vehicle — we run the assembled load balancer against real HTTP servers, inject failures, and verify it handles complex real-world scenarios. The **milestone validation** checkpoints act like quality gates between assembly line stages — each component must pass specific criteria before moving to the next integration level.

This layered approach ensures that when something breaks during complex integration testing, we can quickly isolate whether the problem lies in individual component logic or in the interaction between components. Just as automotive quality assurance catches defects early when they're cheapest to fix, our testing strategy identifies load balancer issues at the appropriate level of granularity.

The Testing Philosophy

A comprehensive testing strategy for a load balancer must address the unique challenges of distributed systems testing. Unlike testing a simple web application, load balancer testing involves multiple moving parts: backend servers that can fail, network connections that can timeout, concurrent requests that can create race conditions, and health checks that must detect failures accurately without overwhelming backends.

The testing pyramid for load balancers has three distinct layers, each serving different purposes. **Unit tests** verify individual algorithms and components work correctly in isolation with mocked dependencies. **Integration tests** validate that components work together correctly and handle real network failures, server responses, and timing issues. **Milestone validation checkpoints** provide structured verification that each development phase produces working functionality before moving to more complex features.

The key insight is that load balancer testing must simulate failure conditions extensively, since the primary value of load balancing is handling backend failures gracefully. A load balancer that works perfectly when all backends are healthy but fails catastrophically when backends go down is worse than no load balancer at all. Therefore, our testing strategy emphasizes failure injection, concurrent request handling, and edge case validation.

> **Design Principle**: Test failure scenarios as thoroughly as success scenarios. A load balancer's value comes from handling failures gracefully, so failure conditions must be first-class citizens in the test suite.

Testing Challenges and Approaches

Load balancer testing presents several unique challenges that require specialized approaches. **Race condition detection** is critical because multiple concurrent requests can expose subtle bugs in counter incrementation, connection tracking, and backend selection. **Timing-sensitive behavior** like health check intervals and circuit breaker timeouts requires careful test design to avoid flaky tests that fail intermittently. **Network simulation** must replicate real-world conditions like connection timeouts, DNS failures, and partial response reads.

Our testing approach addresses these challenges through deterministic mocking for unit tests, controlled integration environments with fault injection capabilities, and structured milestone validation that builds confidence incrementally. The testing strategy emphasizes **reproducible failure scenarios** where we can reliably trigger specific error conditions, measure the system's response, and verify recovery behavior.

Unit Testing Approach

Unit testing for load balancer components focuses on testing individual algorithms, health checkers, and request forwarders in isolation with predictable, controlled inputs. The goal is to verify that each component's core logic works correctly without dependencies on network I/O, timing, or external services.

Algorithm Testing Strategy

Load balancing algorithm testing requires validating both correctness and concurrency safety. Each algorithm must demonstrate correct selection behavior under normal conditions and maintain consistency under concurrent access patterns that simulate real load balancer usage.

Round Robin Algorithm Testing focuses on verifying sequential selection with proper wraparound behavior. The test suite creates a pool of mock backends and verifies that the `RoundRobin` algorithm selects each backend exactly once before cycling back to the first. Concurrency testing spawns multiple goroutines making simultaneous selection calls and verifies that no backend is skipped and the distribution remains even within acceptable variance.

Test Scenario	Input	Expected Output	Verification Method
Sequential Selection	3 backends, 6 requests	Backends selected in order: 0,1,2,0,1,2	Assert sequence matches expected pattern
Empty Backend Pool	Empty slice	Error or nil return	Verify graceful handling without panic
Single Backend	1 backend, 5 requests	Same backend selected 5 times	Assert all selections return same backend
Concurrent Access	1000 concurrent requests	Even distribution ±5%	Statistical analysis of selection counts
Counter Overflow	Max int64 counter value	Proper wraparound behavior	Verify counter resets without errors

Weighted Round Robin Algorithm Testing validates proportional distribution according to backend weights. The algorithm must ensure that over a sufficient number of selections, each backend receives requests proportional to its weight while avoiding burst selections of high-weight backends (smooth weighted round robin behavior).

Test Scenario	Backend Weights	Request Count	Expected Distribution	Tolerance
Equal Weights	[5,5,5]	300 requests	[100,100,100]	±5%
Unequal Weights	[1,2,3]	600 requests	[100,200,300]	±5%
Zero Weight	[1,0,2]	300 requests	[100,0,200]	±5%
Large Weight Difference	[1,100]	1010 requests	[10,1000]	±2%
Weight Changes	[1,1] → [1,3]	Before/after distribution	Smooth transition	Manual verification

Least Connections Algorithm Testing verifies selection of backends with minimum active connections. The test suite manually sets connection counts for mock backends and verifies that the algorithm consistently selects the backend with the fewest active connections, with proper tie-breaking behavior when multiple backends have equal connection counts.

IP Hash Algorithm Testing validates consistent client-to-backend mapping. The same client IP must always map to the same backend (session affinity), and the hash distribution should be reasonably uniform across different IP addresses. Testing includes edge cases like IPv4 vs IPv6 addresses and hash collision handling.

Health Checker Component Testing

Health checker testing focuses on the `HealthChecker` component's ability to track backend health state accurately and transition between healthy and unhealthy states according to configured thresholds. Since health checking involves timing and network I/O, unit tests use mock HTTP clients and controllable time sources.

Health State Transitions testing verifies the state machine behavior defined in the `HealthState` structure. Tests manually trigger probe successes and failures to verify that backends transition from healthy to unhealthy after exceeding the `UnhealthyThreshold` consecutive failures, and back to healthy after exceeding the `HealthyThreshold` consecutive successes.

Current State	Event	Expected New State	Additional Verifications
Healthy	Single failure	Healthy	`ConsecutiveFailures` = 1
Healthy	3 consecutive failures	Unhealthy	`LastFailureTime` updated
Unhealthy	Single success	Unhealthy	`ConsecutiveSuccesses` = 1
Unhealthy	2 consecutive successes	Healthy	`LastSuccessTime` updated
Healthy	Success after failure	Healthy	Counters reset properly

Probe Result Processing testing validates the `HTTPProber` component's ability to interpret HTTP responses correctly. Tests use a mock HTTP server that returns controlled status codes, timeouts, and connection errors to verify that the prober classifies responses correctly as healthy or unhealthy according to the configured `ExpectedStatus`.

Concurrent Health Checking testing verifies thread safety when multiple health check operations run simultaneously. Tests spawn multiple goroutines calling health check methods concurrently and verify that health state updates are atomic and no race conditions corrupt the backend health tracking data structures.

Backend Pool Manager Testing

The `BackendManager` component requires testing of backend registration, health state tracking, and thread-safe access to the backend pool. Unit tests focus on the core logic without network dependencies by using mock backends and health checkers.

Backend Registration testing verifies the `AddBackend` and `RemoveBackend` methods work correctly. Tests add backends with various configurations, verify they appear in the pool with correct initial state, remove backends, and verify they no longer appear in selection results.

Health State Integration testing validates that the backend manager correctly integrates with health checking results. Tests simulate health state changes and verify that `GetHealthyBackends` returns only currently healthy backends, and that unhealthy backends are excluded from algorithm selection.

Connection Tracking testing verifies the `IncrementConnections` and `DecrementConnections` methods maintain accurate connection counts per backend. Tests simulate concurrent connection establishment and closure to verify that connection counting is thread-safe and accurate.

Test Operation Sequence	Expected Connection Count	Verification Method
Increment, Increment, Decrement	1	Assert `GetActiveConnections()` returns 1
100 concurrent increments	100	Verify no race condition corruption
Decrement below zero	0	Verify count doesn't go negative
Reset backend state	0	Verify counter resets properly

Request Forwarding Testing

The `ReverseProxy` component's HTTP request forwarding logic requires testing of header manipulation, request body handling, and error response generation. Unit tests use mock HTTP servers and clients to verify request forwarding behavior without real network dependencies.

Header Forwarding testing verifies that HTTP headers are copied correctly between client requests and backend requests. Tests create requests with various header combinations and verify that all headers are forwarded, forbidden headers are filtered out, and forwarding headers like `X-Forwarded-For` are added correctly.

Request Body Handling testing validates that request bodies of various sizes and content types are forwarded correctly without corruption. Tests use requests with JSON payloads, binary data, and streaming bodies to verify that the proxy correctly streams request data to backends without buffering limitations.

Error Response Generation testing verifies that various backend failure modes result in appropriate HTTP error responses to clients. Tests simulate connection failures, timeouts, and invalid responses to verify that the proxy returns correct status codes (502 Bad Gateway, 504 Gateway Timeout) with appropriate error messages.

Common Pitfalls in Unit Testing

⚠️ **Pitfall: Testing with Real Network I/O** Many developers write unit tests that make actual HTTP requests to localhost servers or external services. This makes tests slow, flaky, and dependent on network conditions. Instead, use mock HTTP servers that return controlled responses, or mock the HTTP client interface entirely to return predetermined responses and errors.

⚠️ **Pitfall: Ignoring Race Conditions** Load balancer algorithms involve shared state (counters, connection counts, health states) that can be corrupted by concurrent access. Tests must specifically verify thread safety by spawning multiple goroutines and checking that results remain consistent. A test that passes in single-threaded execution may fail under concurrent load.

⚠️ **Pitfall: Insufficient Error Scenario Coverage** Developers often test only the happy path where all backends are healthy and responsive. Load balancers must handle various failure modes gracefully, so unit tests must cover scenarios like empty backend pools, all backends unhealthy, network timeouts, and invalid responses.

⚠️ **Pitfall: Non-Deterministic Test Assertions** Tests that rely on exact timing, random number generation, or hash functions may produce different results on different runs or platforms. Use controlled time sources, seeded random number generators, and statistical assertions (within tolerance ranges) rather than exact equality checks.

Integration Testing

Integration testing validates that load balancer components work together correctly in realistic environments with actual HTTP servers, network connections, and failure injection. Unlike unit tests that use mocks, integration tests exercise the complete request processing pipeline from client request receipt through backend selection, health checking, and response forwarding.

End-to-End Request Processing

Integration tests must verify the complete request lifecycle through the load balancer system. This involves starting a real load balancer instance, configuring it with multiple backend servers, sending HTTP requests from test clients, and verifying that requests are distributed correctly and responses are returned properly.

Multi-Backend Distribution Testing creates multiple HTTP test servers (typically 3-5 backends) that return unique responses identifying which backend served the request. Test clients send a series of requests through the load balancer and verify that requests are distributed according to the configured algorithm (round-robin, weighted, etc.) and that all responses are received correctly.

The test environment setup involves starting backend servers on different ports, configuring the load balancer with the backend addresses, waiting for health checks to mark all backends healthy, then executing the request sequence. Each backend server logs incoming requests so that distribution patterns can be verified independently of response analysis.

Test Scenario	Backend Count	Request Count	Algorithm	Verification Method
Round Robin Distribution	3 backends	30 requests	`round_robin`	Each backend gets 10 requests ±1
Weighted Distribution	3 backends (weights 1:2:3)	60 requests	`weighted_round_robin`	Distribution 10:20:30 ±5%
Least Connections	3 backends	Variable load	`least_connections`	Requests flow to least loaded
IP Hash Consistency	3 backends	Same client IP	`ip_hash`	All requests to same backend
Algorithm Switching	3 backends	Before/after switch	Runtime change	Distribution changes correctly

Header and Body Preservation testing verifies that HTTP requests are forwarded to backends without corruption. Test clients send requests with various header combinations, different content types, and request bodies of varying sizes. Backend servers echo back the received headers and bodies so that the test can verify no information is lost or modified during forwarding.

Critical headers to test include `Content-Type`, `Authorization`, custom application headers, and headers containing special characters or Unicode content. Request bodies to test include JSON payloads, form data, binary content, and large bodies that exceed typical buffer sizes to verify streaming behavior.

Failure Injection and Recovery

Integration testing must validate that the load balancer handles various failure modes gracefully and recovers properly when backends return to healthy state. This requires controlled failure injection that simulates real-world backend problems.

Backend Server Failures are simulated by stopping backend servers during active request processing. Tests verify that in-flight requests to the failed backend receive appropriate error responses (502 Bad Gateway), new requests are redirected to healthy backends, and the failed backend is marked unhealthy by the health checking system.

The test sequence involves establishing steady request traffic across all backends, stopping one backend server, verifying that requests continue to be served by remaining backends with minimal error rate, then restarting the backend and verifying that it's restored to the rotation after successful health checks.

Network Partition Simulation tests scenarios where backends are reachable but respond slowly or partially. This is implemented using proxy servers that introduce controlled delays, drop connections at specific points, or return partial responses. The load balancer must handle these conditions with appropriate timeouts and error responses.

Failure Mode	Simulation Method	Expected Behavior	Recovery Verification
Backend crash	Kill backend process	502 errors, traffic redirected	Health checks restore backend
Slow response	Add response delay	Timeouts, failover to healthy backends	Fast backends handle load
Connection drops	Close connections mid-response	Connection errors, retry logic	New connections succeed
Invalid responses	Return malformed HTTP	Parse errors, backend marked unhealthy	Valid responses restore health
DNS failure	Invalid backend hostname	Connection failures, backend excluded	DNS resolution restores backend

Health Check Validation testing verifies that the health checking system accurately detects backend problems and recovers backends appropriately. Tests configure backends to return different health check responses (success, failure, timeout) and verify that the health checker transitions backend states correctly according to the configured thresholds.

The test setup includes mock backends that can be controlled to return specific health check responses on command. Tests verify that backends are marked unhealthy after the configured number of consecutive failures, excluded from request routing while unhealthy, and restored after consecutive successful health checks.

Concurrent Load Testing

Load balancer integration testing must verify correct behavior under concurrent request load that simulates realistic production conditions. This involves spawning multiple client goroutines that send requests simultaneously while monitoring for race conditions, uneven distribution, and performance degradation.

High Concurrency Distribution testing spawns hundreds of concurrent client goroutines that each send multiple requests through the load balancer. The test verifies that request distribution remains statistically correct under high concurrency and that no backends are overwhelmed while others remain idle.

The test monitors backend request counts in real-time and calculates distribution variance across backends. For round-robin algorithms, the variance should remain low even under high concurrent load. For weighted algorithms, the distribution should match configured weights within acceptable tolerance.

Race Condition Detection focuses on scenarios that can expose subtle concurrency bugs in counter incrementation, backend selection, and health state updates. Tests specifically design request patterns that maximize the likelihood of race conditions, such as synchronized request bursts and rapid backend state changes.

Concurrency Test	Concurrent Clients	Requests per Client	Duration	Success Criteria
Distribution Accuracy	100 goroutines	10 requests each	30 seconds	±5% distribution variance
Race Condition Detection	500 goroutines	20 requests each	60 seconds	No panics, consistent counters
Mixed Algorithm Load	200 goroutines	Continuous requests	2 minutes	Algorithm switching works
Health Check Integration	100 goroutines	Continuous requests	5 minutes	Health changes don't break distribution
Connection Tracking	300 goroutines	Keep-alive requests	90 seconds	Connection counts accurate

Backend Failure During Load combines high concurrent request load with backend failures to verify that the load balancer maintains service availability and doesn't cascade failures. Tests establish high request rates across all backends, then systematically fail backends while monitoring error rates and response times.

Configuration and Runtime Changes

Integration testing must validate that configuration changes can be applied during operation without dropping existing connections or causing service interruption. This includes adding and removing backends, changing algorithms, and updating health check settings.

Hot Configuration Reload testing modifies the load balancer configuration file while the system is actively processing requests. Tests verify that configuration changes take effect without interrupting ongoing requests and that new requests use the updated configuration immediately.

The test procedure involves establishing steady request traffic, modifying configuration parameters (backend list, algorithm, health check intervals), triggering configuration reload, and verifying that behavior changes appropriately without connection drops or request failures.

Backend Pool Updates testing adds and removes backends from the pool during active request processing. Tests verify that new backends are integrated smoothly after health checks confirm they're healthy, and removed backends don't receive new requests but complete existing requests properly.

Performance and Resource Usage

Integration testing includes performance validation to ensure the load balancer doesn't introduce excessive latency or resource consumption. Performance tests measure request latency, throughput, memory usage, and connection overhead compared to direct backend access.

Latency Impact Measurement compares response times for requests sent directly to backends versus requests processed through the load balancer. The load balancer should add minimal latency overhead (typically <5ms) for request processing and forwarding.

Throughput Testing measures the maximum request rate the load balancer can sustain while maintaining correct distribution and error handling. Tests gradually increase request rate until error rates exceed acceptable thresholds or response times degrade significantly.

Memory Usage Monitoring tracks memory consumption during extended operation to verify that the load balancer doesn't have memory leaks in connection pooling, health checking, or request processing. Tests run for extended periods (hours) while monitoring memory usage patterns.

Common Pitfalls in Integration Testing

⚠️ **Pitfall: Insufficient Test Environment Isolation** Integration tests often fail when run concurrently or on shared systems because they compete for ports, file descriptors, or other system resources. Each test should use unique port ranges, temporary directories, and isolated configuration to avoid conflicts with other tests or system services.

⚠️ **Pitfall: Race Conditions in Test Setup** Tests that start multiple servers and immediately begin sending requests may fail because servers haven't finished initializing or health checks haven't completed. Include proper synchronization to wait for all components to be ready before starting test operations.

⚠️ **Pitfall: Unrealistic Failure Scenarios** Tests that simulate failures by immediately killing processes or cutting network connections may not reflect realistic failure modes where backends degrade gradually or become partially responsive. Include tests for gradual degradation, intermittent failures, and partial response scenarios.

⚠️ **Pitfall: Inadequate Cleanup** Integration tests that don't properly clean up started servers, open connections, or temporary files can cause resource exhaustion and test failures. Implement proper test teardown that ensures all resources are released even if tests fail or panic.

Milestone Validation Checkpoints

Milestone validation checkpoints provide structured verification that each development phase produces working functionality before proceeding to more complex features. Each checkpoint includes specific behavioral criteria, testing commands, and debugging guidance to ensure learners can validate their progress systematically.

Milestone 1: HTTP Proxy Foundation Validation

The HTTP Proxy Foundation milestone validation ensures that basic reverse proxy functionality works correctly for single-backend forwarding before implementing load balancing algorithms.

Functional Verification Steps:

1. **Start Backend Server**: Launch a simple HTTP server on port 8081 that returns a JSON response containing the request path and timestamp: `{"path": "/test", "backend": "server1", "timestamp": "2024-01-01T12:00:00Z"}`.
2. **Configure Load Balancer**: Create configuration file specifying the single backend server and start the load balancer on port 8080 with logging enabled for all proxied requests.
3. **Basic Request Forwarding**: Send GET request to `http://localhost:8080/hello` and verify that the response contains the expected JSON from the backend with path "/hello".
4. **Header Preservation**: Send request with custom headers (`X-Test-Header: value123`) and verify that the backend receives these headers correctly by examining backend logs.
5. **Request Body Forwarding**: Send POST request with JSON body to the proxy and verify that the backend receives the complete request body without corruption.
6. **Error Response Handling**: Stop the backend server and send a request to the proxy. Verify that the client receives a 502 Bad Gateway response with appropriate error message.

Test Case	Request	Expected Response	Verification Method
GET forwarding	`GET /api/status`	Backend response with path "/api/status"	Response body contains correct path
Header forwarding	Custom headers present	Backend logs show received headers	Check backend request logs
POST body forwarding	JSON payload	Backend receives complete payload	Backend echoes received body
Backend failure	Backend stopped	502 Bad Gateway	HTTP status code verification
Request logging	Any request	Log entry with timestamp, method, path	Check load balancer logs

Command Line Validation:

```bash
# Start backend server (port 8081)
go run backend-server.go -port=8081

# Start load balancer (port 8080, single backend)
go run main.go -config=milestone1-config.json

# Test basic forwarding
curl -v http://localhost:8080/test
# Expected: 200 response with backend JSON

# Test header forwarding
curl -H "X-Custom: test123" http://localhost:8080/headers
# Expected: Backend receives X-Custom header

# Test POST body
curl -X POST -d '{"test": "data"}' -H "Content-Type: application/json" http://localhost:8080/api/data
# Expected: Backend receives JSON body correctly

# Test error handling (stop backend first)
curl http://localhost:8080/test
# Expected: 502 Bad Gateway
```

Success Criteria Checklist:

- ☐ All HTTP methods (GET, POST, PUT, DELETE) are forwarded correctly
- ☐ Request headers including custom headers are preserved
- ☐ Request bodies of various sizes are forwarded without corruption
- ☐ Response headers and status codes are returned to client
- ☐ Connection failures result in 502 responses, not crashes
- ☐ All requests are logged with required information (timestamp, method, path, backend, response status)
- ☐ Keep-alive connections are handled properly
- ☐ No memory leaks during extended operation

Milestone 2: Round Robin Distribution Validation

The Round Robin Distribution milestone validation ensures that requests are distributed evenly across multiple backends using the round-robin algorithm.

Multi-Backend Setup:

1. **Start Multiple Backends**: Launch three backend servers on ports 8081, 8082, 8083, each returning responses that identify which backend served the request.
2. **Configure Round Robin**: Update load balancer configuration with all three backends and set algorithm to `round_robin`.
3. **Distribution Verification**: Send 30 requests and verify that each backend receives exactly 10 requests with no skipped backends.
4. **Concurrent Request Testing**: Use multiple concurrent clients to send requests simultaneously and verify that distribution remains even under concurrent load.
5. **Backend Configuration Changes**: Add a fourth backend and verify that round-robin distribution adapts to include the new backend without disrupting existing request handling.

Distribution Testing Commands:

```bash
# Start three backend servers
go run backend-server.go -port=8081 -id=backend1 &
go run backend-server.go -port=8082 -id=backend2 &
go run backend-server.go -port=8083 -id=backend3 &

# Start load balancer with round-robin configuration
go run main.go -config=milestone2-config.json

# Test sequential distribution
for i in {1..30}; do
    curl -s http://localhost:8080/test | jq .backend_id
done
# Expected: Cycling pattern backend1, backend2, backend3, backend1, ...

# Test concurrent distribution
seq 1 100 | xargs -n1 -P10 -I{} curl -s http://localhost:8080/test | jq .backend_id | sort | uniq -c
# Expected: Approximately 33-34 requests per backend

# Test backend addition
# Add fourth backend to config and reload
curl http://localhost:8080/reload-config
# Send more requests and verify 4-backend distribution
```

Success Criteria Checklist:

- ☐ Sequential requests cycle through backends in order
- ☐ Distribution variance is less than 5% over 1000 requests
- ☐ Concurrent requests maintain even distribution
- ☐ Counter increment is thread-safe (no race conditions)
- ☐ Configuration changes update backend list correctly
- ☐ Empty backend pool handled gracefully
- ☐ Single backend configuration works correctly

Milestone 3: Health Checks Validation

The Health Checks milestone validation ensures that backend health monitoring works correctly and unhealthy backends are excluded from request routing.

Health Check Testing Procedure:

1. **Configure Health Checking**: Set up health checks with 10-second interval, 2-failure threshold, and `/health` endpoint.
2. **Verify Healthy State**: Start all backends with healthy `/health` endpoints returning 200 OK and verify all backends receive requests in round-robin order.
3. **Simulate Backend Failure**: Make one backend return 500 errors on health check endpoint and verify it's excluded from request routing after configured failure threshold.
4. **Test Recovery**: Restore the backend to return healthy responses and verify it's restored to rotation after successful health checks.
5. **Health Check Endpoint Testing**: Verify health checks don't interfere with normal request processing and use appropriate timeouts.

Health Status Monitoring:

```bash
# Configure health checks (10s interval, /health endpoint)

# Start load balancer with health checking enabled

go run main.go -config=milestone3-config.json


# Monitor health check logs

tail -f loadbalancer.log | grep "health_check"


# Check health status endpoint

curl http://localhost:8080/admin/health

# Expected: All backends marked healthy initially


# Simulate backend failure (make backend return 500 on /health)

curl -X POST http://localhost:8081/admin/set-health-status -d '{"healthy": false}'


# Wait for health check interval and verify backend excluded

# Monitor request distribution - should skip failed backend


# Restore backend health

curl -X POST http://localhost:8081/admin/set-health-status -d '{"healthy": true}'


# Verify backend restored to rotation
```

Success Criteria Checklist:

- ☐ Health checks run at configured intervals
- ☐ Backends marked unhealthy after consecutive failures exceed threshold
- ☐ Unhealthy backends excluded from request routing
- ☐ Backends restored after consecutive successful health checks
- ☐ Health check timeouts handled properly
- ☐ Health checking doesn't overwhelm backends
- ☐ No healthy backends scenario handled gracefully
- ☐ Health state changes logged appropriately

Milestone 4: Additional Algorithms Validation

The Additional Algorithms milestone validation ensures that multiple load balancing algorithms work correctly and can be switched at runtime.

Algorithm Testing Matrix:

Each algorithm requires specific validation scenarios that demonstrate its unique characteristics and verify correct implementation of the selection logic.

Algorithm	Test Scenario	Validation Method	Expected Behavior
`weighted_round_robin`	Backends with weights 1:2:3	600 requests, count distribution	100:200:300 requests ±5%
`least_connections`	Variable backend load	Monitor connection counts	Requests flow to least loaded backend
`ip_hash`	Same client IP	Multiple requests from same source	All requests to same backend
`random`	Large request sample	Statistical distribution analysis	Approximately even distribution

Weighted Round Robin Testing:

```bash
# Configure backends with different weights
# Backend1: weight=1, Backend2: weight=2, Backend3: weight=3
go run main.go -config=weighted-config.json

# Send 600 requests and analyze distribution
seq 1 600 | xargs -n1 -P1 -I{} curl -s http://localhost:8080/test | \
    jq -r .backend_id | sort | uniq -c
# Expected output:
# 100 backend1
# 200 backend2
# 300 backend3
```

Least Connections Testing:

```bash
# Start load balancer with least_connections algorithm
go run main.go -config=least-connections-config.json

# Create uneven load by sending long-running requests to some backends
curl http://localhost:8080/slow-endpoint &  # Creates connection to one backend
curl http://localhost:8080/slow-endpoint &  # Creates connection to another backend

# Send new request - should go to backend with fewest connections
curl http://localhost:8080/test
# Verify request goes to backend with lowest connection count
```

IP Hash Consistency Testing:

```bash
# Configure ip_hash algorithm
go run main.go -config=ip-hash-config.json

# Send multiple requests from same source IP
for i in {1..20}; do
    curl -s http://localhost:8080/test | jq .backend_id
done
# Expected: All requests return same backend_id

# Test different source IPs (using X-Forwarded-For simulation)
curl -H "X-Forwarded-For: 192.168.1.100" http://localhost:8080/test
curl -H "X-Forwarded-For: 192.168.1.101" http://localhost:8080/test
# Expected: Different IPs may map to different backends, but consistent per IP
```

Runtime Algorithm Switching:

```bash
# Start with round_robin algorithm
go run main.go -config=initial-config.json

# Send requests and observe round-robin distribution
seq 1 15 | xargs -n1 -I{} curl -s http://localhost:8080/test | jq .backend_id
# Expected: 1,2,3,1,2,3,1,2,3,1,2,3,1,2,3

# Switch to weighted_round_robin algorithm
curl -X POST http://localhost:8080/admin/set-algorithm -d '{"algorithm": "weighted_round_robin"}'

# Send requests and observe weighted distribution
seq 1 60 | xargs -n1 -I{} curl -s http://localhost:8080/test | \
    jq -r .backend_id | sort | uniq -c
# Expected: Distribution according to configured weights
```

Success Criteria Checklist:

- ☐ Weighted round robin distributes proportionally to weights
- ☐ Least connections tracks active connections accurately
- ☐ IP hash provides consistent client-to-backend mapping
- ☐ Random selection distributes approximately evenly
- ☐ Algorithm switching works without service interruption
- ☐ All algorithms handle concurrent requests correctly
- ☐ Edge cases (zero weights, equal connections) handled properly
- ☐ Algorithm-specific statistics available via admin interface

Debugging Common Validation Issues

Request Distribution Problems:

- **Symptom**: Uneven distribution in round-robin
- **Likely Cause**: Race condition in counter increment or modulo operation with changing backend count
- **Diagnosis**: Add detailed logging of counter values and selected backend indices
- **Fix**: Use atomic operations for counter increment and ensure modulo calculation uses consistent backend count

Health Check Issues:

- **Symptom**: Backends not marked unhealthy despite health check failures
- **Likely Cause**: Health check threshold logic incorrect or consecutive failure counting reset improperly
- **Diagnosis**: Log all health check results and state transitions with timestamps
- **Fix**: Verify threshold comparison logic and ensure failure counters only reset on successful checks

Algorithm Selection Errors:

- **Symptom**: Wrong algorithm behavior after runtime switching
- **Likely Cause**: Algorithm state not reset properly or interface methods not implemented correctly
- **Diagnosis**: Verify algorithm interface implementation and state initialization
- **Fix**: Implement proper `Reset()` method for each algorithm and call during algorithm switching

Connection Tracking Problems:

- **Symptom**: Least connections algorithm selects wrong backends
- **Likely Cause**: Connection increment/decrement not called at correct times or race conditions in connection counting
- **Diagnosis**: Add logging for all connection count changes with request correlation
- **Fix**: Ensure connection counting occurs in proper request lifecycle phases and use atomic operations

Implementation Guidance

The testing strategy implementation requires setting up comprehensive test infrastructure that supports both unit testing with mocks and integration testing with real servers.

A. Technology Recommendations

Testing Component	Simple Option	Advanced Option
Unit Test Framework	Go standard testing package	Testify with assertions and mocks
Mock HTTP Server	httptest.NewServer	Ginkgo/Gomega for BDD testing
Integration Testing	Custom test servers	Docker containers for isolation
Load Testing	Sequential curl commands	Apache Bench (ab) or custom Go clients
Failure Injection	Manual server control	Chaos engineering tools
Test Data Management	Hardcoded test data	Property-based testing with go-fuzz

B. Recommended Test Structure

```
project-root/
  cmd/
    loadbalancer/main.go
    test-backend/main.go        ← Simple backend server for testing
  internal/
    proxy/
      proxy.go
      proxy_test.go             ← Unit tests with mocked HTTP client
    algorithms/
      roundrobin.go
      roundrobin_test.go        ← Algorithm unit tests
      weighted.go
      weighted_test.go
    health/
      checker.go
      checker_test.go           ← Health check unit tests with mock HTTP
    manager/
      backend.go
      backend_test.go           ← Backend pool management tests
  test/
    integration/
      basic_proxy_test.go       ← Milestone 1 integration tests
      distribution_test.go      ← Milestone 2 integration tests
      health_check_test.go      ← Milestone 3 integration tests
      algorithms_test.go        ← Milestone 4 integration tests
    fixtures/
      config-milestone1.json    ← Test configurations
      config-milestone2.json
    helpers/
      test_servers.go           ← Common test infrastructure
      assertions.go             ← Custom test assertions
```

C. Test Infrastructure Starter Code

Test Backend Server (complete, ready-to-use):

GO

```go
// cmd/test-backend/main.go

package main

import (
    "encoding/json"
    "flag"
    "fmt"
    "log"
    "net/http"
    "strconv"
    "sync"
    "time"
)

type TestBackend struct {
    ID          string `json:"backend_id"`
    Port        int    `json:"port"`
    RequestCount int64  `json:"request_count"`
    HealthStatus bool   `json:"healthy"`
    SlowResponse bool   `json:"slow_response"`
    mutex       sync.RWMutex
}

func (tb *TestBackend) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    tb.mutex.Lock()
    tb.RequestCount++
    count := tb.RequestCount
    tb.mutex.Unlock()

    if tb.SlowResponse && r.URL.Path == "/slow-endpoint" {
        time.Sleep(5 * time.Second)
    }

    response := map[string]interface{}{
        "backend_id":    tb.ID,
        "port":          tb.Port,
        "path":          r.URL.Path,
        "method":        r.Method,
        "request_count": count,
        "timestamp":     time.Now().Format(time.RFC3339),
        "headers":       r.Header,
    }
```

```
    if r.Method == "POST" && r.ContentLength > 0 {
        // Echo request body for testing body forwarding
        body := make(map[string]interface{})
        json.NewDecoder(r.Body).Decode(&body)
        response["received_body"] = body
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}

func (tb *TestBackend) handleHealth(w http.ResponseWriter, r *http.Request) {
    tb.mutex.RLock()
    healthy := tb.HealthStatus
    tb.mutex.RUnlock()

    if healthy {
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(map[string]string{"status": "healthy"})
    } else {
        w.WriteHeader(http.StatusInternalServerError)
        json.NewEncoder(w).Encode(map[string]string{"status": "unhealthy"})
    }
}

func (tb *TestBackend) handleSetHealth(w http.ResponseWriter, r *http.Request) {
    var req struct {
        Healthy bool `json:"healthy"`
    }
    json.NewDecoder(r.Body).Decode(&req)

    tb.mutex.Lock()
    tb.HealthStatus = req.Healthy
    tb.mutex.Unlock()

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(map[string]bool{"updated": true})
}

func main() {
    port := flag.Int("port", 8081, "Backend server port")
    id := flag.String("id", "backend1", "Backend server ID")
    flag.Parse()
```

```
    backend := &TestBackend{
        ID:           *id,
        Port:         *port,
        HealthStatus: true,
    }

    http.Handle("/", backend)
    http.HandleFunc("/health", backend.handleHealth)
    http.HandleFunc("/admin/set-health-status", backend.handleSetHealth)

    log.Printf("Starting test backend %s on port %d", *id, *port)
    log.Fatal(http.ListenAndServe(":"+strconv.Itoa(*port), nil))
}
```

Integration Test Helpers (complete infrastructure):

```go
// test/helpers/test_servers.go

package helpers

import (
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "testing"
    "time"
)

type TestServerPool struct {
    servers []*TestServer
    ports   []int
    t       *testing.T
}

type TestServer struct {
    ID       string
    Port     int
    Server   *http.Server
    Requests []RequestInfo
    mutex    sync.RWMutex
}

type RequestInfo struct {
    Method    string
    Path      string
    Headers   http.Header
    Timestamp time.Time
}

func NewTestServerPool(t *testing.T, count int, startPort int) *TestServerPool {
    pool := &TestServerPool{
        servers: make([]*TestServer, count),
        ports:   make([]int, count),
        t:       t,
    }

    for i := 0; i < count; i++ {
        port := startPort + i
        pool.servers[i] = &TestServer{
```

```
            ID:   fmt.Sprintf("backend%d", i+1),
            Port: port,
        }
        pool.ports[i] = port
    }

    return pool
}

func (tsp *TestServerPool) Start() error {
    for _, server := range tsp.servers {
        if err := server.Start(); err != nil {
            tsp.Stop() // Clean up any started servers
            return err
        }
    }

    // Wait for all servers to be ready
    time.Sleep(100 * time.Millisecond)
    return nil
}

func (tsp *TestServerPool) Stop() {
    for _, server := range tsp.servers {
        server.Stop()
    }
}

func (ts *TestServer) Start() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", ts.handleRequest)
    mux.HandleFunc("/health", ts.handleHealth)

    ts.Server = &http.Server{
        Addr:    fmt.Sprintf(":%d", ts.Port),
        Handler: mux,
    }

    go func() {
        if err := ts.Server.ListenAndServe(); err != http.ErrServerClosed {
            fmt.Printf("Server error: %v\n", err)
        }
    }()
```

```
    return nil
}

// TODO: Implement request tracking in handleRequest
// TODO: Implement health check endpoint in handleHealth
// TODO: Add method to get request distribution statistics
// TODO: Add method to simulate server failures
```

D. Core Test Skeleton Code

Round Robin Algorithm Test (signatures with TODOs):

```go
// internal/algorithms/roundrobin_test.go
package algorithms

import (
    "sync"
    "testing"
)

func TestRoundRobinSequentialSelection(t *testing.T) {
    // TODO 1: Create RoundRobin algorithm instance
    // TODO 2: Create slice of 3 mock Backend structs
    // TODO 3: Call SelectBackend() 9 times in sequence
    // TODO 4: Verify selection pattern is [0,1,2,0,1,2,0,1,2]
    // TODO 5: Verify counter wraps around properly after reaching end
}

func TestRoundRobinConcurrentAccess(t *testing.T) {
    // TODO 1: Create RoundRobin instance with 3 backends
    // TODO 2: Spawn 100 goroutines, each selecting 10 backends
    // TODO 3: Collect all selections in thread-safe slice
    // TODO 4: Count selections per backend after all goroutines complete
    // TODO 5: Verify distribution is within 5% of expected (333 selections each)
    // TODO 6: Verify no backend was skipped (count > 0 for all backends)
}

func TestRoundRobinEmptyBackendPool(t *testing.T) {
    // TODO 1: Create RoundRobin instance
    // TODO 2: Call SelectBackend with empty backend slice
    // TODO 3: Verify returns nil without panicking
    // TODO 4: Verify counter remains at initial state
}
```

Integration Test Skeleton (milestone validation):

```go
// test/integration/basic_proxy_test.go

package integration

import "testing"

func TestMilestone1BasicProxyForwarding(t *testing.T) {
    // TODO 1: Start single test backend server on port 8081
    // TODO 2: Create load balancer config with single backend
    // TODO 3: Start load balancer on port 8080
    // TODO 4: Send GET request to proxy and verify backend response received
    // TODO 5: Send POST request with JSON body and verify body forwarded correctly
    // TODO 6: Stop backend and verify proxy returns 502 Bad Gateway
    // TODO 7: Clean up all servers and verify no resource leaks
}

func TestMilestone2RoundRobinDistribution(t *testing.T) {
    // TODO 1: Start 3 backend servers on ports 8081-8083
    // TODO 2: Configure load balancer with round_robin algorithm
    // TODO 3: Send 30 sequential requests and collect backend responses
    // TODO 4: Verify each backend received exactly 10 requests
    // TODO 5: Test concurrent requests maintain distribution within tolerance
    // TODO 6: Add fourth backend and verify distribution adapts correctly
}

func TestMilestone3HealthCheckIntegration(t *testing.T) {
    // TODO 1: Start 3 backends with health check endpoints
    // TODO 2: Configure health checking with 5-second interval, 2-failure threshold
    // TODO 3: Verify all backends initially marked healthy and receive requests
    // TODO 4: Make one backend return 500 on health checks
    // TODO 5: Wait for health check failures and verify backend excluded from rotation
    // TODO 6: Restore backend health and verify it returns to rotation
    // TODO 7: Test scenario with all backends unhealthy
}
```

E. Language-Specific Testing Hints

Go Testing Best Practices:

- Use `t.Parallel()` for tests that can run concurrently to speed up test execution
- Use `testing.Short()` to skip long-running tests during development: `if testing.Short() { t.Skip("skipping integration test") }`
- Use `httptest.NewServer()` for creating mock HTTP servers in unit tests
- Use `sync/atomic` package for thread-safe counter operations in concurrent tests
- Use `time.After()` for timeout handling in integration tests to prevent hanging

Test Data Management:

- Store test configurations in `test/fixtures/` directory with descriptive names
- Use `json.Marshal()` to generate test configuration files programmatically
- Use `os.TempDir()` for creating temporary files and directories during tests

- Clean up temporary resources in `defer` statements or test cleanup functions

Concurrent Testing:

- Use `sync.WaitGroup` to coordinate goroutines in concurrent tests
- Use buffered channels to collect results from multiple goroutines safely
- Set `GOMAXPROCS` environment variable to test with different CPU core counts
- Use `go test -race` flag to detect race conditions during test execution

F. Milestone Checkpoints

Milestone 1 Checkpoint:

```bash
# Run unit tests for proxy component
go test ./internal/proxy/... -v

# Run integration tests for basic forwarding
go test ./test/integration/ -run TestMilestone1 -v

# Expected output: All tests pass, no connection errors
# Manual verification: curl commands work as documented above
```

Milestone 2 Checkpoint:

```bash
# Test algorithm implementations
go test ./internal/algorithms/... -v -race

# Test distribution integration
go test ./test/integration/ -run TestMilestone2 -v

# Expected: Even distribution within tolerance, no race conditions detected
```

Milestone 3 Checkpoint:

```bash
# Test health checking unit logic
go test ./internal/health/... -v

# Test health integration with backend failures
go test ./test/integration/ -run TestMilestone3 -v -timeout=60s

# Expected: Backends properly excluded and restored based on health status
```

Milestone 4 Checkpoint:

```bash
# Test all algorithm implementations
go test ./internal/algorithms/... -v -race

# Test algorithm switching integration
go test ./test/integration/ -run TestMilestone4 -v

# Expected: All algorithms work correctly, runtime switching successful
```

Debugging Guide

> **Milestone(s):** All milestones (1-4) — debugging skills are essential throughout the project, from basic HTTP proxy connection issues (Milestone 1) through request distribution problems (Milestone 2), health check failures (Milestone 3), and algorithm selection issues (Milestone 4)

Mental Model: Medical Diagnosis

Think of debugging a load balancer like a medical diagnosis process. Just as a doctor systematically examines symptoms, runs tests, and narrows down the root cause before prescribing treatment, debugging a load balancer requires methodical observation of symptoms, targeted diagnostic tests, and systematic elimination of potential causes. The load balancer exhibits "symptoms" like failed requests or uneven distribution, and we use "diagnostic tools" like logs, metrics, and network traces to identify the underlying "disease" — whether it's a configuration error, network issue, or algorithmic bug.

The debugging process follows the same systematic approach as medical diagnosis: first gather symptoms from multiple sources (logs, metrics, user reports), then form hypotheses about potential causes, design targeted tests to verify or eliminate each hypothesis, and finally apply the appropriate fix. Just as doctors maintain differential diagnosis lists, effective load balancer debugging requires maintaining a mental catalog of common failure modes and their characteristic symptoms.

Connection and Networking Issues

Network connectivity problems represent the most fundamental category of load balancer failures. When the basic HTTP reverse proxy functionality fails to establish connections or properly forward requests, the entire system becomes non-functional regardless of how sophisticated the load balancing algorithms might be.

Connection failures manifest in several distinct patterns, each requiring different diagnostic approaches. **Connection refused errors** typically indicate that the target backend server is not listening on the expected port, while **timeout errors** suggest that packets are being sent but no response is received within the configured time limit. **DNS resolution failures** prevent the load balancer from even attempting to connect to backends, and **TLS handshake failures** occur when HTTPS backends have certificate or protocol configuration issues.

The diagnostic process for connection issues follows a systematic bottom-up approach, starting with basic network connectivity and progressing through increasingly sophisticated layers. This layered approach ensures that we don't waste time debugging application-layer issues when the root cause lies in basic network configuration.

Symptom	Likely Root Cause	Primary Diagnostic Steps	Secondary Investigation
"Connection refused" errors	Backend not running or wrong port	Check backend service status, verify port binding	Examine firewall rules, network routing
Timeout on all requests	Network routing issue or backend overload	Test direct backend connectivity, check network path	Monitor backend resource usage, examine proxy timeout settings
DNS resolution failures	Incorrect backend hostnames or DNS server issues	Test DNS resolution manually, verify hostname configuration	Check DNS server connectivity, examine domain configuration
TLS handshake failures	Certificate mismatch or protocol version incompatibility	Examine certificate validity and hostname matching	Check supported TLS versions, cipher suite compatibility
Intermittent connection failures	Backend intermittently unavailable or network instability	Monitor backend availability over time, check network stability	Examine backend startup/shutdown patterns, investigate network infrastructure
Port binding errors	Load balancer port already in use or permission issues	Check port availability, verify user permissions	Examine other processes using the port, check system limits

Connection Pool Management Issues

Connection pooling problems create particularly subtle debugging challenges because they often manifest as performance degradation rather than outright failures. The `ReverseProxy` component maintains connection pools to backend servers for efficiency, but misconfigured pooling can cause resource exhaustion, connection leaks, or unnecessary connection churn.

Connection leaks occur when HTTP connections to backends are opened but never properly closed, gradually exhausting the available connection pool. This typically manifests as gradually increasing response times followed by eventual connection timeouts as the pool becomes exhausted. The diagnostic approach involves monitoring the active connection count over time and examining connection lifecycle management in the proxy code.

Port Binding and Permission Problems

Load balancer startup failures often stem from port binding issues that prevent the HTTP server from accepting incoming connections. These problems require examining both the network configuration and system-level permissions that control port access.

> **Key Insight**: Port binding failures during load balancer startup are often the first indication of configuration conflicts or permission issues. A systematic approach to port diagnostics can quickly identify whether the problem lies in port availability, user permissions, or network interface configuration.

Error Type	Diagnostic Command	Expected Output	Troubleshooting Steps
Port already in use	`netstat -tlnp | grep :8080`	Shows process using port 8080	Identify conflicting process, change port, or terminate conflict
Permission denied	`sudo netstat -tlnp | grep :80`	Requires root to bind privileged ports	Run as root, use port >1024, or configure capabilities
Interface binding failure	`ip addr show`	Lists available network interfaces	Verify interface exists, check interface configuration
IPv6/IPv4 configuration	`ss -tlnp | grep :8080`	Shows listening addresses and protocols	Configure correct IP version, verify dual-stack support

Network Routing and Connectivity

Backend connectivity issues often stem from network routing problems that prevent the load balancer from reaching backend servers. These issues require systematic testing of network paths and routing configuration.

The diagnostic process for routing issues involves testing connectivity at progressively higher network layers. We start with basic IP connectivity using ping, progress to port-level connectivity using telnet or nc, and finally test HTTP-level connectivity using curl or similar tools. This layered approach helps isolate whether problems exist at the network, transport, or application layer.

Test Level	Command Example	Success Indication	Failure Diagnosis
IP connectivity	`ping backend1.example.com`	Packets received with normal latency	Check routing tables, firewall rules, DNS resolution
Port connectivity	`telnet backend1.example.com 8080`	Successfully connected	Examine service binding, port configuration, local firewall
HTTP connectivity	`curl -I http://backend1.example.com:8080/health`	HTTP status code returned	Check HTTP service configuration, request handling, application firewall
TLS connectivity	`openssl s_client -connect backend1.example.com:8443`	TLS handshake completes	Examine certificate configuration, TLS version compatibility

Load Distribution Problems

Load distribution issues represent algorithmic and concurrency problems in the backend selection logic. Unlike connection issues which prevent any request processing, distribution problems allow some requests to succeed while creating uneven traffic patterns or algorithmic failures that compromise the load balancer's effectiveness.

Distribution problems typically manifest as statistical anomalies in request routing patterns. The `RoundRobin` algorithm should distribute requests evenly across healthy backends, but race conditions in the counter increment or modulo operation can create clustering or skipped backends. The `WeightedRoundRobin` algorithm should respect configured weight ratios over time, but implementation bugs can cause disproportionate distribution or integer overflow issues.

Round Robin Counter Race Conditions

The most common distribution problem involves race conditions in the round robin counter increment operation. When multiple goroutines simultaneously increment the `counter` field in the `RoundRobin` struct without proper synchronization, the atomic increment operation can be lost or duplicated, leading to uneven distribution patterns.

Race condition symptoms typically appear under concurrent load rather than during sequential testing. Single-threaded tests might show perfect distribution while concurrent load tests reveal significant skew. The diagnostic approach involves examining distribution statistics over large request volumes and monitoring counter increment behavior under concurrent access.

Distribution Pattern	Root Cause	Diagnostic Approach	Verification Method
Some backends never receive requests	Counter increment race condition causing skipped values	Monitor counter increments vs backend selection, check for gaps	Load test with request tracking, verify each backend receives requests
Clustering of requests to single backend	Modulo operation using stale counter value	Examine counter read/increment timing, check atomic operation usage	Track sequential backend selections, look for repeated selections
Uneven distribution over time	Integer overflow in counter causing negative modulo results	Monitor counter value growth, check for overflow conditions	Test with large request volumes, monitor counter wraparound behavior
Random distribution instead of round robin	Multiple counter increments per request	Trace counter increments per request processing, verify single increment	Add counter increment logging, ensure one increment per selection

Algorithm Selection and State Management

Algorithm switching problems occur when the load balancer attempts to change from one selection algorithm to another during runtime. The `RequestRouter` coordinates between different algorithm implementations, but state inconsistencies can cause selection failures or incorrect algorithm behavior.

Algorithm state corruption often occurs during transitions when the previous algorithm's state remains cached or when the new algorithm initializes with incorrect state. The `WeightedRoundRobin` algorithm maintains complex internal state in the `currentWeights` map that must be properly reset when switching algorithms or updating backend configurations.

Weight Configuration and Calculation Errors

Weighted distribution algorithms face additional complexity from weight calculation and normalization logic. The `WeightedRoundRobin` implementation must handle weight configuration changes, zero weights, and weight ratio calculations without introducing mathematical errors or division-by-zero conditions.

> **Key Insight**: Weight-based algorithms are particularly susceptible to configuration errors and edge cases. A backend configured with zero weight should be excluded from selection, while negative weights typically indicate configuration errors that should be detected during validation rather than causing runtime failures.

Weight Configuration Issue	Symptom	Root Cause Analysis	Resolution Strategy
Backend with zero weight receives requests	Selection algorithm ignores weight values	Check weight application in selection logic	Implement weight validation, exclude zero-weight backends
Integer overflow in weight calculations	Sudden distribution changes or selection failures	Monitor weight sum calculations, check for overflow	Use appropriate integer types, implement overflow detection
Negative weights cause selection failures	Algorithm exceptions or undefined behavior	Examine weight validation during configuration loading	Add configuration validation, reject negative weights
Weight changes not reflected in distribution	Algorithm uses cached weight values	Check weight update propagation to selection logic	Implement proper weight update notification

Backend Pool Synchronization Issues

Backend pool management creates complex synchronization challenges when backends are added, removed, or have their health status updated while requests are being processed. The `BackendManager` must coordinate between health checking updates and algorithm selection requests without introducing race conditions or inconsistent state.

Synchronization problems typically manifest as selection of unavailable backends or failure to utilize newly added healthy backends. The diagnostic approach involves examining the timing relationship between backend state changes and selection algorithm state updates.

Health Check Debugging

Health checking systems introduce temporal complexity into load balancer debugging because health state changes occur asynchronously relative to request processing. The `HealthChecker` component runs periodic probes against backend servers and updates health state based on consecutive success or failure counts, creating multiple timing-dependent failure modes.

Health check false positives occur when healthy backends are marked as unhealthy due to transient network issues, overly aggressive failure thresholds, or health check implementation problems. False negatives occur when unhealthy backends remain marked as healthy, continuing to receive traffic and generate client-visible failures.

Health Check False Positives and Negatives

False positive health check failures create availability problems by unnecessarily removing healthy backends from the rotation. This typically occurs when health check timeouts are too aggressive for the network environment, when health check intervals create excessive load on backend servers, or when health endpoints don't accurately reflect backend health.

The diagnostic process for false positives involves comparing health check results with actual backend availability and examining the correlation between health check timing and backend performance. Manual health endpoint testing often reveals discrepancies between health check logic and actual service availability.

False Positive Pattern	Root Cause	Diagnostic Steps	Corrective Actions
Healthy backends marked unhealthy during peak load	Health check timeout too aggressive	Compare health check timing with backend response times under load	Increase health check timeout, implement adaptive timeouts
Intermittent false failures on stable backends	Network instability affecting health checks	Monitor network path to backends, compare with direct connectivity tests	Increase consecutive failure threshold, implement jittered retry logic
All backends fail health checks simultaneously	Health check implementation error or network partition	Examine health check implementation, test health endpoints manually	Fix health check logic, implement network connectivity verification
New backends never pass initial health checks	Health endpoint configuration mismatch	Verify health endpoint paths and expected responses	Correct health endpoint configuration, implement endpoint discovery

Health State Transition Problems

Backend health state management involves complex transition logic between healthy, unhealthy, and recovery states based on consecutive success and failure counts. The `HealthState` structure tracks multiple counters and timestamps that must be updated atomically to prevent inconsistent state transitions.

State transition bugs often manifest as backends that become "stuck" in unhealthy states despite successful health checks, or backends that rapidly alternate between healthy and unhealthy states due to inadequate hysteresis in the transition logic. This "flapping" behavior can cause traffic disruption and excessive logging.

Health Check Endpoint Configuration Issues

Health check configuration problems stem from mismatches between the health check implementation and backend server health endpoint design. The `HealthCheckConfig` specifies the expected HTTP path, status code, and response characteristics, but backends may implement health endpoints with different semantics.

Configuration debugging involves verifying that health check requests match the exact format expected by backend health endpoints. This includes HTTP method, request headers, expected response codes, and response body validation logic.

Configuration Mismatch	Symptom	Verification Method	Solution
Wrong health endpoint path	All health checks return 404	Manual curl test of health endpoint	Correct health check path configuration
Incorrect expected status code	Health checks fail despite healthy backend	Compare expected vs actual response codes	Update expected status code in configuration
Missing required headers	Backend rejects health check requests	Examine backend access logs for health check requests	Add required headers to health check configuration
Response timeout too short	Health checks timeout on slow but healthy backends	Monitor health check response times vs timeout setting	Increase health check timeout or optimize backend health endpoint

Health Check Timing and Frequency Issues

Health check timing configuration requires balancing between rapid failure detection and avoiding excessive load on backend servers. The `DefaultInterval` of 30 seconds provides a reasonable starting point, but production environments may require different timing based on failure tolerance and backend capacity.

Overly frequent health checks can overwhelm backend servers, particularly during startup or high load conditions. This creates a feedback loop where health checks contribute to backend overload, causing legitimate health check failures that remove backends from rotation. The diagnostic approach involves monitoring backend resource usage during health check intervals and correlating health check timing with backend performance metrics.

Debugging Tools and Techniques

Effective load balancer debugging requires a systematic toolkit of logging strategies, monitoring approaches, and diagnostic techniques tailored to distributed system challenges. Unlike debugging single-process applications, load balancer debugging must account for network timing, concurrent request processing, and interactions between multiple backend servers.

The foundation of load balancer debugging lies in comprehensive logging that provides correlation between client requests, backend selection decisions, and response outcomes. The `RequestContext` structure carries correlation information throughout the request processing pipeline, enabling end-to-end request tracing even when requests span multiple components and goroutines.

Structured Logging and Request Correlation

Request correlation represents the most critical debugging capability for load balancer systems. Each incoming request receives a unique request ID that gets logged at every processing stage, enabling reconstruction of the complete request flow from initial receipt through backend selection, health checking, and response forwarding.

The logging strategy must balance between providing sufficient detail for debugging while avoiding performance impact under high request volumes. Structured logging formats like JSON enable programmatic log analysis while maintaining human readability for interactive debugging sessions.

Log Level	Information Captured	Example Entry	Use Ca
DEBUG	Detailed algorithm selection, health check probes, connection pool operations	`{"level":"debug","time":"2023-10-15T10:30:45Z","request_id":"req-123","event":"backend_selected","algorithm":"round_robin","backend":"backend-2","counter":42}`	Algorith debugg perform analysi
INFO	Request processing, backend health changes, configuration updates	`{"level":"info","time":"2023-10-15T10:30:45Z","request_id":"req-123","event":"request_completed","method":"GET","path":"/api/users","backend":"backend-2","status":200,"duration_ms":45}`	Operati monitor request tracking
WARN	Health check failures, backend unavailability, configuration issues	`{"level":"warn","time":"2023-10-15T10:30:45Z","event":"backend_unhealthy","backend":"backend-1","consecutive_failures":3,"last_error":"connection_timeout"}`	Early warning system capacit plannin
ERROR	Request failures, connection errors, system failures	`{"level":"error","time":"2023-10-15T10:30:45Z","request_id":"req-123","event":"request_failed","error":"no_healthy_backends","total_backends":3,"healthy_backends":0}`	Inciden respons failure analysi

Metrics Collection and Analysis

Metrics collection provides quantitative analysis capabilities that complement qualitative log analysis. The load balancer should expose metrics for request distribution patterns, backend health statistics, response time distributions, and error rates categorized by type and backend.

Request distribution metrics enable detection of algorithm implementation bugs by comparing actual distribution patterns against expected theoretical distributions. The round robin algorithm should show nearly equal request counts across all healthy backends over time, while weighted algorithms should show proportional distribution matching configured weights.

Performance Profiling and Resource Monitoring

Load balancer performance profiling requires monitoring both internal component performance and overall system resource usage. Go's built-in pprof package provides CPU and memory profiling capabilities that help identify bottlenecks in request processing, algorithm selection, or health checking logic.

Memory profiling becomes particularly important for connection pool management and health state tracking, where memory leaks can gradually degrade system performance. The diagnostic approach involves capturing memory profiles during different operational phases and identifying objects that persist longer than expected.

Resource Metric	Monitoring Method	Normal Range	Investigation Threshold
CPU utilization	System monitoring, pprof CPU profiles	<50% during normal load	>80% sustained
Memory usage	Process memory monitoring, pprof heap profiles	Stable after warmup	Continuous growth indicating leaks
Goroutine count	Runtime metrics, pprof goroutine profiles	Stable with request load	Unbounded growth
File descriptor usage	System limits monitoring	<50% of system limits	>80% of available descriptors
Network connection count	Netstat analysis, connection pool metrics	Matches expected backend connections	Excessive connection accumulation

Network-Level Debugging Tools

Network-level debugging tools provide visibility into the actual HTTP traffic between the load balancer and backend servers. Tools like tcpdump, Wireshark, and curl enable examination of raw HTTP requests and responses, helping identify protocol-level issues that might not be apparent from application logs.

Network debugging becomes essential when diagnosing subtle HTTP protocol issues like header forwarding problems, connection keep-alive behavior, or TLS handshake failures. The systematic approach involves capturing network traffic during problem reproduction and analyzing the packet-level interaction between

components.

Automated Testing and Validation

Automated testing tools enable systematic validation of load balancer behavior under various failure scenarios. The `TestServerPool` provides a foundation for creating controlled testing environments where backend failures, network issues, and load patterns can be simulated reproducibly.

Testing automation should cover both normal operation validation and failure mode testing. Normal operation tests verify that request distribution matches algorithm expectations, that healthy backends receive appropriate traffic, and that response forwarding preserves HTTP semantics. Failure mode tests verify that backend failures are detected appropriately, that traffic fails over correctly, and that recovery behavior works as expected.

Implementation Guidance

The debugging infrastructure for a load balancer requires careful instrumentation throughout the codebase combined with external tooling for monitoring and analysis. This implementation guidance provides complete debugging utilities and systematic troubleshooting procedures.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	Standard library `log/slog` with JSON formatting	`logrus` or `zap` with structured fields and sampling
Metrics Collection	Prometheus client library with HTTP exposition	Comprehensive APM solution like DataDog or New Relic
Health Monitoring	Simple HTTP endpoint returning JSON status	Full health dashboard with historical trending
Request Tracing	UUID-based correlation IDs in headers	Distributed tracing with OpenTelemetry or Jaeger
Network Analysis	Command-line tools (curl, netstat, tcpdump)	Wireshark GUI for detailed packet analysis
Load Testing	Custom test harness with concurrent requests	Professional tools like Apache Bench or wrk

Debugging Infrastructure Code

```go
// internal/debug/logger.go

package debug


import (

    "context"

    "encoding/json"

    "log/slog"

    "os"

    "time"

)


// DebugLogger provides structured logging with request correlation

type DebugLogger struct {

    logger *slog.Logger

    level  slog.Level

}


// NewDebugLogger creates a configured logger instance

func NewDebugLogger(level string) *DebugLogger {

    var logLevel slog.Level

    switch level {

    case "debug":

        logLevel = slog.LevelDebug

    case "info":

        logLevel = slog.LevelInfo

    case "warn":

        logLevel = slog.LevelWarn

    case "error":

        logLevel = slog.LevelError

    default:

        logLevel = slog.LevelInfo

    }


    handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{

        Level: logLevel,

    })



    return &DebugLogger{

        logger: slog.New(handler),

        level:  logLevel,

    }
```

```go
}

// LogRequest records request processing events with correlation
func (d *DebugLogger) LogRequest(ctx context.Context, event string, attrs map[string]interface{}) {
    requestID := ctx.Value("request_id")
    if requestID == nil {
        requestID = "unknown"
    }

    slogAttrs := []slog.Attr{
        slog.String("event", event),
        slog.String("request_id", requestID.(string)),
        slog.Time("timestamp", time.Now()),
    }

    for key, value := range attrs {
        slogAttrs = append(slogAttrs, slog.Any(key, value))
    }

    d.logger.LogAttrs(ctx, slog.LevelInfo, "Request event", slogAttrs...)
}

// LogBackendHealth records backend health state changes
func (d *DebugLogger) LogBackendHealth(backend string, healthy bool, reason string) {
    d.logger.Info("Backend health change",
        slog.String("backend", backend),
        slog.Bool("healthy", healthy),
        slog.String("reason", reason),
        slog.Time("timestamp", time.Now()),
    )
}

// LogAlgorithmSelection records load balancing algorithm decisions
func (d *DebugLogger) LogAlgorithmSelection(algorithm string, backend string, counter int64) {
    d.logger.Debug("Backend selected",
        slog.String("algorithm", algorithm),
        slog.String("backend", backend),
        slog.Int64("counter", counter),
        slog.Time("timestamp", time.Now()),
    )
}
```

```go
// internal/debug/metrics.go

package debug


import (

    "sync/atomic"

    "time"

)


// MetricsCollector tracks load balancer performance metrics

type MetricsCollector struct {

    TotalRequests        int64

    SuccessfulRequests   int64

    FailedRequests       int64

    BackendSelections    map[string]int64

    ResponseTimes        []time.Duration

    ConnectionPoolSize   int64

    ActiveConnections    int64

}


// NewMetricsCollector creates a metrics collection instance

func NewMetricsCollector() *MetricsCollector {

    return &MetricsCollector{

        BackendSelections: make(map[string]int64),

        ResponseTimes:     make([]time.Duration, 0, 1000),

    }

}


// RecordRequest increments request counters based on success

func (m *MetricsCollector) RecordRequest(success bool, duration time.Duration) {

    atomic.AddInt64(&m.TotalRequests, 1)

    if success {

        atomic.AddInt64(&m.SuccessfulRequests, 1)

    } else {

        atomic.AddInt64(&m.FailedRequests, 1)

    }



    // Store response time (simplified - production would use histogram)

    if len(m.ResponseTimes) < cap(m.ResponseTimes) {

        m.ResponseTimes = append(m.ResponseTimes, duration)

    }

}


// RecordBackendSelection tracks which backends are being selected
```

```go
func (m *MetricsCollector) RecordBackendSelection(backend string) {
    // Note: In production, use atomic operations or mutex for map access
    m.BackendSelections[backend]++
}

// GetDistributionStats returns request distribution statistics
func (m *MetricsCollector) GetDistributionStats() map[string]interface{} {
    total := atomic.LoadInt64(&m.TotalRequests)
    successful := atomic.LoadInt64(&m.SuccessfulRequests)
    failed := atomic.LoadInt64(&m.FailedRequests)

    stats := map[string]interface{}{
        "total_requests":      total,
        "successful_requests": successful,
        "failed_requests":     failed,
        "success_rate":        float64(successful) / float64(total),
        "backend_selections":  m.BackendSelections,
    }

    if len(m.ResponseTimes) > 0 {
        var sum time.Duration
        for _, duration := range m.ResponseTimes {
            sum += duration
        }
        stats["average_response_time"] = sum / time.Duration(len(m.ResponseTimes))
    }

    return stats
}
```

Diagnostic Test Harness

```go
// internal/debug/diagnostic.go

package debug


import (

    "context"

    "fmt"

    "net/http"

    "sync"

    "time"

)


// DiagnosticTest performs systematic load balancer testing

type DiagnosticTest struct {

    LoadBalancerURL string

    BackendURLs     []string

    Logger          *DebugLogger

    Metrics         *MetricsCollector

}


// NewDiagnosticTest creates a diagnostic test instance

func NewDiagnosticTest(lbURL string, backends []string) *DiagnosticTest {

    return &DiagnosticTest{

        LoadBalancerURL: lbURL,

        BackendURLs:     backends,

        Logger:          NewDebugLogger("debug"),

        Metrics:         NewMetricsCollector(),

    }

}


// TestConnectivity verifies basic connectivity to load balancer and backends

func (d *DiagnosticTest) TestConnectivity(ctx context.Context) error {

    // TODO 1: Test direct connectivity to load balancer port

    // TODO 2: Test direct connectivity to each backend server

    // TODO 3: Verify that load balancer can reach all backends

    // TODO 4: Record connectivity results and timing

    // Hint: Use http.Client with short timeout for connectivity tests

    return fmt.Errorf("not implemented")

}


// TestRequestDistribution validates load balancing algorithm behavior

func (d *DiagnosticTest) TestRequestDistribution(ctx context.Context, requestCount int) error {

    // TODO 1: Send requestCount requests to load balancer
```

```
    // TODO 2: Track which backend responds to each request (using response headers or content)
    // TODO 3: Calculate distribution statistics across backends
    // TODO 4: Verify that distribution matches expected algorithm behavior
    // TODO 5: Report any distribution skew or algorithm failures
    // Hint: Add backend identification headers to responses for tracking
    return fmt.Errorf("not implemented")
}

// TestHealthCheckBehavior validates health checking and failover
func (d *DiagnosticTest) TestHealthCheckBehavior(ctx context.Context) error {
    // TODO 1: Send requests to verify initial healthy state
    // TODO 2: Simulate backend failure (stop backend or block health endpoint)
    // TODO 3: Monitor health check detection timing and traffic failover
    // TODO 4: Restore backend and verify recovery detection
    // TODO 5: Validate that no requests were lost during failover
    // Hint: Monitor health check logs and request success rates during failure
    return fmt.Errorf("not implemented")
}

// RunConcurrentLoad generates concurrent load to expose race conditions
func (d *DiagnosticTest) RunConcurrentLoad(ctx context.Context, goroutines int, duration time.Duration) error {
    var wg sync.WaitGroup
    results := make(chan error, goroutines)

    for i := 0; i < goroutines; i++ {
        wg.Add(1)
        go func(workerID int) {
            defer wg.Done()
            // TODO 1: Generate requests continuously for specified duration
            // TODO 2: Record response times and success/failure rates
            // TODO 3: Track backend selection patterns per worker
            // TODO 4: Report any errors or unexpected behavior
            // Hint: Use time.NewTicker for consistent request timing
            results <- fmt.Errorf("worker %d not implemented", workerID)
        }(i)
    }

    wg.Wait()
    close(results)

    for err := range results {
```

```
        if err != nil {
            return err
        }
    }

    return nil
}
```

Systematic Debugging Procedures

Connection Issue Debugging Checklist:

1. **Port Availability Check**: `netstat -tlnp | grep :8080` to verify load balancer port binding
2. **Backend Connectivity Test**: `curl -I http://backend1:8080/health` for each configured backend
3. **Network Route Verification**: `traceroute backend1` to identify routing issues
4. **DNS Resolution Test**: `nslookup backend1.example.com` to verify hostname resolution
5. **Firewall Rule Check**: `iptables -L` or equivalent to examine filtering rules
6. **Connection Pool Monitoring**: Track active connections with `netstat -tn | grep :8080 | wc -l`

Distribution Issue Debugging Steps:

1. **Algorithm State Inspection**: Add debug logging to track counter values and backend selections
2. **Statistical Distribution Analysis**: Generate 1000+ requests and verify distribution patterns
3. **Race Condition Detection**: Run concurrent load tests while monitoring selection patterns
4. **Configuration Validation**: Verify algorithm configuration matches expected behavior
5. **Backend Health Correlation**: Ensure distribution only includes healthy backends

Health Check Debugging Process:

1. **Manual Health Endpoint Testing**: `curl -v http://backend1:8080/health` to verify endpoint behavior
2. **Health Check Timing Analysis**: Monitor health check intervals and response times
3. **State Transition Logging**: Track consecutive success/failure counts and state changes
4. **False Positive Investigation**: Compare health check results with actual backend availability
5. **Recovery Behavior Validation**: Verify that recovered backends receive traffic appropriately

Milestone Validation Checkpoints

Milestone 1 Validation (HTTP Proxy Foundation):

```bash
# Start load balancer with single backend
./load-balancer --config single-backend.yaml &

# Test basic request forwarding
curl -H "X-Test: milestone1" http://localhost:8080/api/test

# Expected: Request forwarded to backend, response returned with X-Forwarded-For header
# Check logs for request correlation ID and successful forwarding
```

Milestone 2 Validation (Round Robin Distribution):

```bash
# Start load balancer with 3 backends
./load-balancer --config three-backends.yaml &

# Generate 30 requests to test distribution
for i in {1..30}; do curl -s http://localhost:8080/api/test | grep backend_id; done

# Expected: Each backend receives exactly 10 requests (±1 for timing)
# Check distribution metrics endpoint: curl http://localhost:8080/debug/metrics
```

Milestone 3 Validation (Health Checks):

```bash
# Start system with health checking enabled
./load-balancer --config health-enabled.yaml &

# Stop one backend to trigger health check failure
docker stop backend2

# Monitor health check detection (should take 2-3 check intervals)
curl http://localhost:8080/debug/health

# Generate requests - should only reach healthy backends
for i in {1..10}; do curl http://localhost:8080/api/test; done

# Restart backend and verify recovery
docker start backend2
```

Future Extensions

> **Milestone(s):** All milestones (1-4) — this section explores potential enhancements beyond the core load balancer implementation, demonstrating how the architectural decisions made throughout the project enable natural evolution and growth of the system's capabilities.

Mental Model: Growing Software Ecosystem

Think of extending the load balancer like expanding a thriving city. The basic load balancer is like the city's foundational infrastructure — roads, utilities, and core services. Future extensions are like adding specialized districts: a financial district (SSL termination), a smart traffic management system (advanced routing), and a comprehensive monitoring network (observability features). Each extension builds on the existing foundation while adding new capabilities that serve different needs.

Just as a well-planned city leaves room for growth with proper zoning and infrastructure corridors, the load balancer's modular architecture creates natural extension points. The component boundaries, interface abstractions, and data models established in earlier sections provide the "zoning laws" that guide where and how new features can be integrated without disrupting existing functionality.

The key insight is that extensions should feel like natural evolution rather than awkward retrofitting. When the core architecture is designed with extensibility in mind, new features integrate smoothly and leverage existing components rather than working around them.

SSL/TLS Termination: Adding HTTPS Support and Certificate Management

The **SSL/TLS termination** extension transforms the load balancer from a simple HTTP proxy into a comprehensive HTTPS gateway that handles encryption, certificate management, and secure client communications. This extension addresses the critical requirement for secure web traffic while offloading encryption overhead from backend servers.

In modern web architectures, SSL/TLS termination at the load balancer provides significant operational advantages. It centralizes certificate management, reduces computational load on backend servers, and enables the load balancer to inspect encrypted traffic for routing decisions. The load balancer becomes the security boundary, handling all client-facing encryption while communicating with backends over internal networks using HTTP.

The SSL termination capability integrates naturally with the existing `ReverseProxy` component by adding a TLS layer before HTTP processing. The current request flow of client → load balancer → backend becomes client → TLS termination → load balancer → backend, with the load balancer now responsible for the

cryptographic handshake and session management.

Certificate Management Architecture

The certificate management system requires several new components that work together to provide automated certificate lifecycle management:

Component	Responsibility	Integration Point
CertificateStore	Manages certificate storage, loading, and rotation	Integrates with Config loading system
TLSListener	Handles TLS handshakes and client connections	Replaces HTTP listener in ReverseProxy
CertificateValidator	Validates certificate chains and expiration	Used by HealthChecker for certificate monitoring
AutoRenewal	Automatic certificate renewal via ACME protocol	Runs as background service alongside HealthChecker
SNIRouter	Routes requests based on Server Name Indication	Integrates with request routing pipeline

The certificate store manages multiple certificates simultaneously, enabling the load balancer to serve multiple domains from a single instance. This is particularly important in microservices environments where a single load balancer might front dozens of different services, each with their own domain and certificate requirements.

```go
type TLSConfig struct {
    Enabled           bool                   `json:"enabled"`
    CertificatesPath  string                 `json:"certificates_path"`
    PrivateKeyPath    string                 `json:"private_key_path"`
    CipherSuites      []uint16               `json:"cipher_suites"`
    MinTLSVersion     uint16                 `json:"min_tls_version"`
    MaxTLSVersion     uint16                 `json:"max_tls_version"`
    ClientAuth        tls.ClientAuthType     `json:"client_auth"`
    Certificates      []CertificateConfig    `json:"certificates"`
    ACME              ACMEConfig             `json:"acme"`
    OCSP              OCSPConfig             `json:"ocsp"`
}

type CertificateConfig struct {
    Domain           string    `json:"domain"`
    CertificatePath  string    `json:"certificate_path"`
    PrivateKeyPath   string    `json:"private_key_path"`
    ExpirationTime   time.Time `json:"expiration_time"`
    AutoRenew        bool      `json:"auto_renew"`
    Backends         []string  `json:"backends"`
}
```

ACME Protocol Integration

Automatic Certificate Management Environment (ACME) integration enables the load balancer to automatically obtain and renew certificates from Certificate Authorities like Let's Encrypt. This removes the operational burden of manual certificate management while ensuring certificates remain current.

The ACME integration requires careful coordination with the existing health checking system. The load balancer must temporarily serve ACME challenge responses while maintaining normal traffic flow. This creates an interesting architectural challenge where the request router must distinguish between ACME challenges and normal application traffic.

> **Design Insight: ACME Challenge Routing**
>
> ACME HTTP-01 challenges require the load balancer to serve specific content at `/.well-known/acme-challenge/` paths. This creates a conflict with normal request forwarding, as these requests must be handled locally rather than forwarded to backends. The solution is to add ACME challenge detection to the request routing pipeline, creating a short-circuit path for challenge requests.

Architecture Decision: Certificate Storage Strategy

> **Decision: File-Based Certificate Storage with In-Memory Caching**
>
> - **Context**: Certificates need persistent storage for restart survival and fast access during TLS handshakes
> - **Options Considered**:
>   - Database storage with encrypted certificate data
>   - File-based storage with periodic reload
>   - In-memory only with external certificate injection
> - **Decision**: File-based storage with automatic reload detection and in-memory caching
> - **Rationale**: File-based storage provides simplicity and compatibility with standard certificate management tools, while in-memory caching ensures TLS handshake performance isn't degraded by disk I/O
> - **Consequences**: Enables standard certificate management workflows while maintaining performance, but requires file system security and backup considerations

TLS Performance Considerations

TLS termination significantly impacts load balancer performance due to the computational overhead of cryptographic operations. Modern approaches use hardware acceleration where available and optimize connection reuse to amortize handshake costs across multiple requests.

The integration with the existing connection pooling system becomes more complex with TLS termination. While backend connections can remain HTTP and benefit from existing pooling, client-facing TLS connections require different management strategies. TLS session resumption and connection reuse become critical for maintaining acceptable performance levels.

Client Certificate Authentication

Advanced TLS configurations support mutual authentication where clients present certificates for verification. This creates additional integration points with the request routing system, as certificate-based identity can influence backend selection and routing decisions.

The client certificate validation integrates with the error handling system established in earlier sections. Certificate validation failures generate appropriate error responses while maintaining security by avoiding information disclosure about the specific failure reasons.

Advanced Routing Rules: Path-Based, Header-Based, and Content-Aware Distribution

The **advanced routing rules** extension transforms the load balancer from simple request distribution into intelligent application-aware traffic management. This extension enables sophisticated routing decisions based on request content, enabling use cases like API versioning, canary deployments, and multi-tenant architectures.

Current load balancing algorithms focus purely on backend selection without considering request characteristics. Advanced routing adds a decision layer that examines request properties before applying load balancing algorithms, creating a two-stage process: first route to the appropriate backend pool, then apply load balancing within that pool.

Routing Rule Architecture

The routing rule system introduces several new concepts that extend the existing data model:

Concept	Description	Integration Point
RoutingRule	Defines conditions and actions for request routing	Added to Config structure
RequestMatcher	Evaluates whether requests match rule conditions	Integrated with RequestRouter
BackendPool	Groups backends for specific routing rules	Extends existing Backend grouping
RoutingAction	Defines what happens when rule matches	Applied before algorithm selection
RulePriority	Determines evaluation order for multiple rules	Managed by request processing pipeline

The routing rule evaluation happens early in the request processing pipeline, immediately after request parsing but before backend selection. This positioning allows routing decisions to influence which backend pool is considered for load balancing, while still benefiting from health checking and algorithm selection within the chosen pool.

```go
type RoutingConfig struct {
    Rules            []RoutingRule     `json:"rules"`
    DefaultPool      string            `json:"default_pool"`
    FailureHandling  string            `json:"failure_handling"`
    RuleEvaluation   RuleEvalConfig    `json:"rule_evaluation"`
}

type RoutingRule struct {
    ID               string            `json:"id"`
    Priority         int               `json:"priority"`
    Conditions       []RuleCondition   `json:"conditions"`
    Actions          []RuleAction      `json:"actions"`
    Enabled          bool              `json:"enabled"`
    Description      string            `json:"description"`
    CreatedAt        time.Time         `json:"created_at"`
    Statistics       RuleStats         `json:"statistics"`
}

type RuleCondition struct {
    Type             string            `json:"type"`
    Field            string            `json:"field"`
    Operator         string            `json:"operator"`
    Value            string            `json:"value"`
    CaseSensitive    bool              `json:"case_sensitive"`
    Negate           bool              `json:"negate"`
}
```

Path-Based Routing Implementation

Path-based routing enables different URL paths to route to different backend pools, supporting microservices architectures where different services handle different API endpoints. This routing strategy is essential for API gateways and service mesh implementations.

The path matching system supports various matching strategies:

Matching Strategy	Example	Use Case
Exact Match	`/api/users`	Specific endpoint routing
Prefix Match	`/api/v1/*`	API version routing
Regex Match	`/users/\d+/profile`	Pattern-based routing
Wildcard Match	`/static/**`	Static content routing

Path-based routing integrates naturally with the existing request processing pipeline. The URL path is available immediately after request parsing, enabling early routing decisions without requiring request body inspection or complex header analysis.

Header-Based Routing Implementation

Header-based routing enables routing decisions based on HTTP headers, supporting use cases like API versioning through Accept headers, tenant routing through custom headers, and device-specific routing through User-Agent headers.

Header-based routing is particularly powerful for implementing canary deployments and A/B testing. Requests can be routed to experimental backend pools based on specific header values, enabling controlled rollouts and feature flagging at the load balancer level.

The header matching system requires careful consideration of header normalization and case sensitivity. HTTP headers are case-insensitive by specification, but different clients and frameworks may use different casing conventions. The routing rule system normalizes headers for consistent matching while preserving original header values for forwarding to backends.

Content-Aware Distribution

Content-aware distribution represents the most sophisticated routing capability, enabling decisions based on request body content. This capability supports use cases like routing GraphQL queries to specialized backends, directing large file uploads to high-capacity storage services, and implementing content-based sharding.

Content-aware routing requires careful integration with the existing request forwarding system. Reading request bodies for routing decisions means the body must be buffered and made available for both routing evaluation and subsequent forwarding to backends. This creates memory and performance implications that must be carefully managed.

> **Architecture Decision: Request Body Buffering Strategy**
>
> - **Context**: Content-aware routing requires access to request bodies, but HTTP proxying typically streams bodies to avoid memory consumption
> - **Options Considered**:
>   - Always buffer bodies in memory for routing evaluation
>   - Stream bodies while extracting routing-relevant content
>   - Conditional buffering based on content-type and size
> - **Decision**: Conditional buffering with configurable size limits and content-type filtering
> - **Rationale**: Provides content-aware routing capabilities while limiting memory impact and maintaining streaming performance for large uploads
> - **Consequences**: Enables sophisticated routing at the cost of increased memory usage and latency for content-inspected requests

Routing Rule Evaluation Engine

The routing rule evaluation engine processes rules in priority order, applying the first matching rule's action. This requires careful consideration of rule ordering and conflict resolution to ensure predictable behavior.

The evaluation engine integrates with the existing request context system, adding routing decisions to the request correlation data for observability and debugging. Failed routing evaluations are logged with sufficient detail to enable troubleshooting of complex rule interactions.

Backend Pool Management

Advanced routing requires extending the backend management system to support multiple named pools rather than a single backend list. Each pool can have its own load balancing algorithm, health check configuration, and operational parameters.

The backend pool concept extends naturally from the existing `BackendManager`, creating a hierarchical structure where the manager contains multiple pools, each containing multiple backends. This structure maintains backward compatibility while enabling advanced routing scenarios.

Monitoring and Metrics: Adding Detailed Metrics, Monitoring Endpoints, and Observability Features

The **monitoring and metrics** extension transforms the load balancer from a black box into a fully observable system that provides deep insights into traffic patterns, performance characteristics, and operational health. This extension addresses the critical operational requirement for visibility into system behavior and performance.

Observability becomes increasingly important as load balancers handle more traffic and support more complex routing logic. Operators need visibility into request distribution, backend health patterns, algorithm performance, and error trends to effectively manage and optimize the system. The monitoring extension provides this visibility through metrics collection, monitoring endpoints, and integration with external observability systems.

Metrics Architecture

The metrics system introduces several new components that integrate with existing load balancer functionality:

Component	Responsibility	Integration Point
MetricsCollector	Collects and aggregates metrics from all components	Integrated throughout request processing pipeline
MetricsRegistry	Manages metric definitions and storage	Singleton service used by all components
MetricsExporter	Exports metrics in standard formats (Prometheus, JSON)	HTTP endpoint handler for metrics scraping
HistogramManager	Manages latency and duration histograms	Integrated with request timing
AlertingEngine	Evaluates metric thresholds and generates alerts	Background service monitoring metrics

The metrics collection system requires minimal performance impact while providing comprehensive visibility. This creates a design challenge where metrics collection must be efficient enough to run on every request without significantly impacting load balancer performance.

```go
type MetricsConfig struct {
    Enabled          bool                   `json:"enabled"`
    CollectionInterval time.Duration        `json:"collection_interval"`
    RetentionPeriod  time.Duration          `json:"retention_period"`
    Exporters        []ExporterConfig       `json:"exporters"`
    Histograms       HistogramConfig        `json:"histograms"`
    Alerting         AlertingConfig         `json:"alerting"`
    CustomMetrics    []CustomMetricConfig   `json:"custom_metrics"`
}

type LoadBalancerMetrics struct {
    TotalRequests        int64                  `json:"total_requests"`
    SuccessfulRequests   int64                  `json:"successful_requests"`
    FailedRequests       int64                  `json:"failed_requests"`
    RequestRate          float64                `json:"request_rate"`
    ErrorRate            float64                `json:"error_rate"`
    ResponseTime         ResponseTimeMetrics    `json:"response_time"`
    BackendMetrics       map[string]BackendMetrics `json:"backend_metrics"`
    AlgorithmMetrics     AlgorithmMetrics       `json:"algorithm_metrics"`
    HealthCheckMetrics   HealthCheckMetrics     `json:"health_check_metrics"`
    ConnectionMetrics    ConnectionMetrics      `json:"connection_metrics"`
}
```

Request-Level Metrics Collection

Request-level metrics provide the foundation for understanding load balancer behavior and performance. These metrics capture every aspect of request processing, from initial receipt through backend selection to response delivery.

The request metrics collection integrates with the existing `RequestContext` system established in earlier sections. Each request context carries a metrics collection object that accumulates timing, status, and routing information throughout the request lifecycle.

Metric Category	Examples	Collection Point
Request Volume	Requests per second, total requests	Request start/completion
Response Times	P50, P95, P99 latency	Request start/end timing
Status Codes	2xx, 4xx, 5xx response counts	Response processing
Backend Selection	Algorithm choice, backend distribution	Algorithm selection
Error Rates	Connection failures, timeouts, health check failures	Error handling points

Backend Performance Metrics

Backend performance metrics provide visibility into individual backend server behavior and health patterns. These metrics enable operators to identify problematic backends, understand capacity utilization, and optimize backend pool configuration.

Backend metrics collection extends the existing `Backend` data structure with metrics collection capabilities. Each backend maintains its own metrics collector that aggregates performance data over configurable time windows.

The backend metrics integrate with the health checking system to provide correlation between health check results and actual request performance. This correlation enables more sophisticated health assessment that considers both probe success and actual request performance.

Load Balancing Algorithm Metrics

Algorithm-specific metrics provide insights into how different load balancing strategies perform under various traffic patterns. These metrics enable operators to choose appropriate algorithms and tune algorithm parameters for optimal performance.

Each algorithm implementation extends the base `Algorithm` interface with metrics collection capabilities. The metrics collected vary by algorithm type but generally include selection patterns, decision timing, and effectiveness measures.

Algorithm	Key Metrics	Purpose
RoundRobin	Selection distribution, counter resets	Verify even distribution
WeightedRoundRobin	Weight utilization, smooth distribution	Optimize weight settings
LeastConnections	Connection count accuracy, selection efficiency	Validate connection tracking
IPHash	Hash distribution, consistency maintenance	Assess hash function quality
Random	Distribution uniformity, selection performance	Evaluate randomness quality

Health Check Metrics

Health check metrics provide detailed visibility into backend health patterns, failure detection accuracy, and recovery behavior. These metrics are essential for tuning health check parameters and understanding system reliability characteristics.

The health check metrics extend the existing `HealthState` data structure with additional statistical information. Historical health data enables trend analysis and predictive health assessment.

Performance Impact Considerations

Metrics collection must balance comprehensiveness with performance impact. The metrics system uses several strategies to minimize overhead:

- **Atomic counters**: Use atomic operations for high-frequency metrics to avoid locking overhead
- **Batched updates**: Aggregate metrics in memory and flush periodically rather than updating external systems synchronously
- **Sampling**: For very high-traffic scenarios, sample a percentage of requests rather than collecting metrics on every request
- **Lock-free data structures**: Use lock-free ring buffers and other concurrent data structures for metrics collection

> **Architecture Decision: Metrics Collection Strategy**
>
> - **Context**: Comprehensive metrics are essential for operations, but collection overhead can impact load balancer performance
> - **Options Considered**:
>   - Synchronous metrics with external system updates on every request
>   - Asynchronous metrics with in-memory aggregation and periodic export
>   - Sampling-based metrics collection with configurable sample rates
> - **Decision**: Asynchronous collection with in-memory aggregation and multiple export formats
> - **Rationale**: Provides comprehensive metrics while maintaining performance, with flexibility in how metrics are consumed by external systems
> - **Consequences**: Enables rich observability without performance penalties, but requires careful memory management for metrics storage

Prometheus Integration

Prometheus integration enables the load balancer to export metrics in the industry-standard Prometheus format, enabling integration with existing monitoring infrastructure and alerting systems.

The Prometheus integration requires exposing a `/metrics` endpoint that serves metrics in Prometheus text format. This endpoint integrates with the existing HTTP server infrastructure while providing access to all collected metrics.

Custom Metrics and Extensibility

The metrics system provides extensibility for application-specific metrics that operators may want to collect. Custom metrics can be defined through configuration and collected at various points in the request processing pipeline.

Custom metrics support enables advanced use cases like business metrics collection (e.g., API usage by customer tier) and application-specific performance metrics (e.g., cache hit rates for cached responses).

Alerting Integration

The alerting engine evaluates metric thresholds and generates alerts when conditions are met. Alerting rules can be defined for any collected metric, enabling operators to create comprehensive monitoring coverage.

The alerting system integrates with external notification systems through webhooks and standard protocols. Alert conditions can include simple threshold checks, trend analysis, and complex multi-metric conditions.

Dashboard and Visualization Support

The metrics export system provides data in formats suitable for popular dashboard systems like Grafana. Pre-built dashboard templates enable operators to quickly establish comprehensive load balancer monitoring.

The dashboard integration includes both real-time metrics for operational monitoring and historical data for capacity planning and performance analysis.

Integration Points with Existing Architecture

All three extension areas integrate naturally with the existing load balancer architecture through well-defined extension points established in the original design.

Configuration Extension Points

The `Config` structure established in earlier sections uses Go's struct embedding to enable extension:

```go
type ExtendedConfig struct {
    Config                  // Embed existing configuration
    TLS        *TLSConfig     `json:"tls,omitempty"`
    Routing    *RoutingConfig `json:"routing,omitempty"`
    Metrics    *MetricsConfig `json:"metrics,omitempty"`
}
```

This approach maintains backward compatibility while enabling new functionality through configuration.

Component Integration Patterns

Each extension integrates with existing components through established interfaces and patterns:

- **Middleware Pattern**: SSL termination and routing rules act as middleware layers in the request processing pipeline
- **Observer Pattern**: Metrics collection observes events throughout the system without modifying core logic
- **Strategy Pattern**: Advanced routing rules extend the existing algorithm selection strategy
- **Plugin Architecture**: Extensions can be enabled/disabled through configuration without code changes

Data Flow Extensions

The extensions modify the request processing pipeline by adding new stages while preserving the core flow:

1. **Original Flow**: Client → HTTP Proxy → Backend Selection → Backend Request → Response
2. **Extended Flow**: Client → TLS Termination → Routing Rules → Metrics Collection → HTTP Proxy → Backend Selection → Metrics Collection → Backend Request → Response → Metrics Collection

Each extension adds processing stages without disrupting the fundamental request flow, maintaining system reliability while adding new capabilities.

Implementation Guidance

The future extensions build naturally on the foundation established in the core load balancer implementation. Each extension represents a significant enhancement that can be tackled as an independent project phase.

Technology Recommendations

Extension	Simple Option	Advanced Option
SSL/TLS Termination	Go's crypto/tls with file-based certificates	ACME integration with automatic renewal
Advanced Routing	JSON-based routing rules with simple matching	Rule engine with complex expressions
Monitoring & Metrics	Prometheus client library with HTTP endpoint	Full observability stack with tracing

Development Sequence Recommendation

The extensions should be implemented in dependency order to maximize reuse and minimize complexity:

1. **Start with Monitoring**: Establishes observability foundation needed for testing other extensions
2. **Add SSL Termination**: Provides security foundation and demonstrates middleware pattern
3. **Implement Advanced Routing**: Builds on monitoring for rule effectiveness measurement

Metrics Collection Starter Code

```go
// MetricsCollector provides thread-safe metrics collection for load balancer components
type MetricsCollector struct {
    totalRequests    int64
    successRequests  int64
    errorRequests    int64
    responseTimes    []time.Duration
    backendMetrics   map[string]*BackendMetrics
    mu               sync.RWMutex
    startTime        time.Time
}

// NewMetricsCollector creates a new metrics collector instance
func NewMetricsCollector() *MetricsCollector {
    return &MetricsCollector{
        backendMetrics: make(map[string]*BackendMetrics),
        startTime:      time.Now(),
        responseTimes:  make([]time.Duration, 0, 10000), // Pre-allocate for performance
    }
}

// RecordRequest increments request counters and records response time
func (m *MetricsCollector) RecordRequest(success bool, duration time.Duration, backendID string) {
    // TODO 1: Atomically increment totalRequests counter
    // TODO 2: If success is true, atomically increment successRequests, otherwise increment errorRequests
    // TODO 3: Acquire write lock and append duration to responseTimes slice
    // TODO 4: Update backend-specific metrics for the selected backend
    // TODO 5: If responseTimes slice exceeds capacity, remove oldest entries to prevent memory growth
    // Hint: Use atomic.AddInt64 for counters to avoid locking overhead
}

// GetCurrentStats returns snapshot of current metrics
func (m *MetricsCollector) GetCurrentStats() LoadBalancerMetrics {
    // TODO 1: Create LoadBalancerMetrics struct to return
    // TODO 2: Read atomic counters for request totals
    // TODO 3: Acquire read lock and calculate response time percentiles from responseTimes slice
    // TODO 4: Calculate request rate as total requests divided by uptime
    // TODO 5: Calculate error rate as failed requests divided by total requests
    // TODO 6: Copy backend metrics map (need deep copy to avoid race conditions)
    // TODO 7: Return populated metrics struct
}
```

SSL Termination Integration Points

```go
// TLSProxy wraps ReverseProxy with SSL termination capabilities
type TLSProxy struct {
    proxy           *ReverseProxy
    tlsConfig       *tls.Config
    certificateStore *CertificateStore
    server          *http.Server
    metrics         *MetricsCollector
}

// NewTLSProxy creates SSL-terminating proxy that wraps existing ReverseProxy
func NewTLSProxy(proxy *ReverseProxy, config *TLSConfig) (*TLSProxy, error) {
    // TODO 1: Create TLS configuration from provided config
    // TODO 2: Initialize certificate store and load certificates
    // TODO 3: Set up TLS config with certificate callback for SNI support
    // TODO 4: Create HTTP server with TLS config
    // TODO 5: Configure server to use existing ReverseProxy as handler
    // TODO 6: Return configured TLSProxy instance
}

// ServeHTTPS starts HTTPS server with SSL termination
func (t *TLSProxy) ServeHTTPS(addr string) error {
    // TODO 1: Configure server address
    // TODO 2: Start listening on TLS port
    // TODO 3: Handle graceful shutdown signals
    // TODO 4: Start certificate renewal background process if ACME enabled
    // TODO 5: Return any startup errors
}
```

Advanced Routing Rule Engine

```go
// RoutingEngine evaluates routing rules and selects appropriate backend pools
type RoutingEngine struct {
    rules           []RoutingRule
    defaultPool     string
    matchers        map[string]RequestMatcher
    metrics         *MetricsCollector
    mu              sync.RWMutex
}

// EvaluateRouting determines backend pool for request based on routing rules
func (r *RoutingEngine) EvaluateRouting(req *http.Request) (string, error) {
    // TODO 1: Acquire read lock to safely access rules slice
    // TODO 2: Sort rules by priority (highest first) if not already sorted
    // TODO 3: For each rule, evaluate all conditions using appropriate matchers
    // TODO 4: If all conditions match, apply rule action and return backend pool
    // TODO 5: If no rules match, return default pool
    // TODO 6: Record metrics for rule evaluation (hits, misses, evaluation time)
    // TODO 7: Handle rule evaluation errors gracefully with fallback to default pool
}

// AddRoutingRule adds new routing rule with validation
func (r *RoutingEngine) AddRoutingRule(rule RoutingRule) error {
    // TODO 1: Validate rule syntax and conditions
    // TODO 2: Check for conflicting rules with same priority
    // TODO 3: Acquire write lock for rule modification
    // TODO 4: Insert rule in priority-sorted position
    // TODO 5: Update metrics about total rules and rule changes
}
```

Milestone Checkpoints for Extensions

After implementing SSL termination:

- Verify HTTPS connections work: `curl -k https://localhost:8443/health`
- Check certificate loading: Look for certificate validation messages in logs
- Test SNI support: Configure multiple certificates for different domains
- Validate TLS metrics: Check that SSL handshake metrics are collected

After implementing advanced routing:

- Test path-based routing: Configure rules for different URL paths and verify backend selection
- Test header-based routing: Send requests with different headers and verify routing decisions
- Validate rule priority: Configure overlapping rules and verify highest priority rule wins
- Check routing metrics: Verify rule hit counts and evaluation timing are recorded

After implementing monitoring system:

- Access metrics endpoint: `curl http://localhost:8080/metrics` shows Prometheus format
- Verify metric collection: Generate load and confirm metrics update correctly
- Test alerting rules: Configure threshold and verify alerts trigger appropriately

- Validate dashboard integration: Import metrics into Grafana and verify visualization

Common Extension Pitfalls

⚠️ **Pitfall: SSL Performance Degradation** Adding SSL termination can significantly impact performance due to cryptographic overhead. Monitor request latency and throughput after enabling SSL. Use hardware acceleration when available and optimize TLS configuration for performance. Consider connection reuse and session resumption to amortize handshake costs.

⚠️ **Pitfall: Routing Rule Conflicts** Complex routing rules can interact in unexpected ways, causing requests to route to wrong backends. Implement rule validation that checks for conflicts and ambiguous conditions. Provide clear error messages when rules conflict. Test routing decisions thoroughly with comprehensive request scenarios.

⚠️ **Pitfall: Metrics Collection Memory Leaks** Unbounded metrics collection can cause memory leaks, especially for response time histograms and per-backend metrics. Implement metric retention policies with automatic cleanup of old data. Use sampling for high-frequency metrics. Monitor metrics collector memory usage and implement circuit breakers if needed.

⚠️ **Pitfall: Extension Configuration Complexity** Multiple extensions can create configuration files that are difficult to understand and maintain. Design configuration schemas that are hierarchical and well-documented. Provide configuration validation that gives clear error messages. Consider configuration templates and examples for common use cases.

⚠️ **Pitfall: Backward Compatibility Breaks** Extensions should not break existing configurations or APIs. Use configuration versioning and provide migration tools when necessary. Test extensively with existing configurations to ensure compatibility. Document any breaking changes clearly and provide upgrade paths.

The future extensions demonstrate the value of thoughtful architectural design in the original load balancer implementation. By establishing clear component boundaries, extensible interfaces, and modular data models, the core system provides natural extension points that enable sophisticated functionality without architectural disruption.

Each extension represents a significant enhancement that addresses real operational requirements in production load balancer deployments. The SSL termination extension enables secure communications, the advanced routing extension supports complex application architectures, and the monitoring extension provides essential operational visibility.

Glossary

> **Milestone(s):** All milestones (1-4) — this glossary defines technical terms, acronyms, and domain-specific vocabulary used throughout the load balancer design and implementation

Mental Model: Technical Dictionary

Think of this glossary as a technical dictionary for a specialized field. Just as medical professionals have precise terminology like "myocardial infarction" instead of "heart attack," load balancing has specific vocabulary that carries exact meaning. Each term represents a concept that has been refined through years of distributed systems practice, and using the correct terminology helps communicate complex ideas precisely between engineers.

The terminology in load balancing draws from multiple domains — networking (Layer 4 vs Layer 7), distributed systems (graceful degradation, circuit breakers), and algorithms (round-robin, consistent hashing). Understanding these terms deeply helps engineers discuss trade-offs, debug issues, and design systems that behave predictably under various conditions.

Core Load Balancing Terms

Term	Definition	Context
load balancer	A system that distributes incoming requests across multiple backend servers to prevent any single server from becoming overwhelmed. Acts as a traffic director that makes intelligent routing decisions based on configured algorithms and backend health status.	Central component of the entire system
reverse proxy	A server that sits between clients and backend servers, forwarding client requests to backends and returning backend responses to clients. Unlike a forward proxy that acts on behalf of clients, a reverse proxy acts on behalf of servers.	Core functionality implemented in Milestone 1
backend server	An application server that sits behind the load balancer and processes the actual business logic for client requests. Backend servers are the ultimate destination for forwarded requests.	Target destinations for all load-balanced traffic
backend pool	A collection of backend servers that are available to receive forwarded requests. The pool is dynamically managed based on configuration changes and health check results.	Managed by the Backend Pool Manager component
upstream server	Alternative term for backend server, commonly used in nginx and other proxy contexts. Refers to servers that are "upstream" in the request processing flow from the load balancer's perspective.	Synonym for backend server
session affinity	A load balancing strategy that routes requests from the same client to the same backend server consistently. Also known as "sticky sessions," this is essential for applications that store session state locally.	Implemented through IP hash algorithm
graceful degradation	The ability of a system to maintain partial functionality when some components fail, rather than failing completely. In load balancing, this means continuing to serve requests with fewer backends when some become unhealthy.	Key principle in error handling and system resilience

HTTP and Networking Terms

Term	Definition	Context
Layer 4 load balancing	Load balancing that operates at the transport layer (TCP/UDP) and makes routing decisions based only on IP addresses and ports. Cannot inspect HTTP headers or content.	Alternative approach not used in this project
Layer 7 load balancing	Load balancing that operates at the application layer (HTTP) and can make routing decisions based on HTTP headers, URLs, cookies, and request content.	The approach implemented in this project
connection pooling	Reusing established HTTP connections for multiple requests instead of creating new connections for each request. Improves performance by avoiding the overhead of TCP handshakes and SSL negotiations.	Optimization technique in HTTP proxy implementation
header forwarding	The process of copying HTTP headers from the original client request to the backend request, and from the backend response to the client response. Critical for preserving client context.	Essential part of proper reverse proxying
X-Forwarded-For	HTTP header that contains the IP addresses of clients and intermediate proxies in a request chain. Allows backend servers to identify the original client IP address.	Added by the request enhancement process
X-Forwarded-Proto	HTTP header that indicates the original protocol (HTTP or HTTPS) used by the client to connect to the load balancer. Important for applications that need to generate correct URLs.	Added by the request enhancement process
request forwarding	The process of receiving an HTTP request from a client and sending a corresponding request to a backend server, preserving the essential request properties.	Core functionality of the reverse proxy
response streaming	Transferring response data from backend to client without buffering the entire response in memory. Essential for handling large responses efficiently.	Performance optimization in response handling

Load Balancing Algorithm Terms

Term	Definition	Context
load balancing algorithm	A strategy or method for selecting which backend server should receive each incoming request. Different algorithms optimize for different goals like even distribution or session consistency.	Core component with multiple implementations
round-robin	A load balancing algorithm that cycles through backend servers in sequential order, giving each server an equal number of requests over time. Simple and effective for homogeneous backends.	Primary algorithm implemented in Milestone 2
weighted distribution	A load balancing approach where backends receive traffic proportional to assigned weights. Higher-capacity servers receive proportionally more requests.	Advanced algorithm variation
least connections	An algorithm that routes each request to the backend server currently handling the fewest active connections. Useful when request processing times vary significantly.	Implemented in Milestone 4
smooth weighted round robin	A refined weighted algorithm that avoids burst selections of high-weight backends by gradually adjusting selection probabilities. Provides more even distribution than naive weighted approaches.	Advanced weighted algorithm implementation
connection tracking	Monitoring the number of active HTTP connections or requests being processed by each backend server. Required for implementing least connections algorithm effectively.	Supporting mechanism for connection-based algorithms
hash consistency	The property that the same input (like client IP address) consistently maps to the same backend server, even when the backend pool changes. Essential for session affinity.	Key property of IP hash algorithm
atomic operations	Thread-safe operations that complete entirely or not at all, without interference from other concurrent operations. Critical for maintaining consistent counters in multi-threaded environments.	Required for thread-safe algorithm implementations
algorithm switching	The ability to change the active load balancing algorithm at runtime without restarting the service or dropping existing connections.	Runtime configuration capability

Health Checking Terms

Term	Definition	Context
health check	A periodic verification mechanism that determines whether a backend server is available and capable of processing requests. Can be active (proactive probing) or passive (based on request failures).	Core monitoring mechanism implemented in Milestone 3
active health checking	A proactive monitoring approach where the load balancer periodically sends health check requests to backend servers, independent of client traffic.	Primary health checking approach
passive health checking	A monitoring approach that infers backend health from the success or failure of actual client requests, without sending dedicated health check probes.	Alternative approach not implemented
consecutive failure counting	Tracking the number of sequential health check failures for each backend to determine when it should be marked unhealthy. Prevents marking backends unhealthy due to transient failures.	State transition logic in health checking
recovery detection	The process of monitoring unhealthy backends for signs of recovery and restoring them to the active rotation when they become healthy again.	Critical for automatic system recovery
probe	A single health check request sent to a backend server to verify its availability and responsiveness. Typically a simple HTTP GET request to a dedicated health endpoint.	Basic unit of health checking
failure detection	The overall process of identifying when backend servers become unavailable or unresponsive, including both health check failures and request-time failures.	System reliability mechanism
thundering herd	A problematic scenario where many clients or processes simultaneously attempt to access a resource that has just become available, potentially overwhelming it.	Common pitfall in health check recovery
health endpoint	A dedicated URL path on backend servers that responds to health check requests with status information. Typically lightweight and doesn't perform complex business logic.	Backend server responsibility
state flapping	Rapid alternation between healthy and unhealthy states, often due to a backend server that is marginally functional or experiencing intermittent issues.	Problem mitigated by consecutive check thresholds

Request Processing Terms

Term	Definition	Context
request router	The central coordinator component that integrates HTTP proxy functionality, backend selection, and health checking to route requests effectively through the complete processing pipeline.	Central orchestration component
request processing pipeline	A multi-stage flow that handles requests from initial reception through backend selection, request forwarding, response processing, and final client delivery.	Overall request handling architecture
request context	Correlation data and metadata carried through the processing pipeline for a single request, including request ID, timing information, and selected backend details.	Request tracking and debugging aid
error response handling	The process of generating appropriate HTTP error responses for various failure scenarios, including backend failures, timeouts, and configuration issues.	Error management strategy
backend selection	The process of choosing an appropriate backend server for an incoming request using the configured load balancing algorithm and considering backend health status.	Algorithm application point
request enhancement	The process of adding proxy-specific headers (like X-Forwarded-For) and modifying the original client request to prepare it for forwarding to a backend server.	Request preparation step
connection management	Establishing, maintaining, and properly closing HTTP connections to backend servers, including connection pooling and timeout handling for efficiency and reliability.	Resource management concern
resource cleanup	Proper deallocation of resources like HTTP connections, goroutines, and memory buffers when requests complete or fail, preventing resource leaks.	Memory and connection management
error correlation	The practice of tracking requests through system logs using correlation IDs, making it possible to trace a single request's path through all system components.	Debugging and monitoring aid

Configuration and Runtime Terms

Term	Definition	Context
hot reload	The capability to update system configuration without interrupting service operation or dropping existing connections. Changes take effect immediately without restart.	Runtime configuration management
configuration validation	The process of checking configuration correctness, completeness, and internal consistency before applying changes to the running system.	Safety mechanism for configuration changes
atomic updates	Configuration changes that either succeed completely or fail completely, without leaving the system in an inconsistent intermediate state.	Reliability pattern for configuration management
configuration distribution	The process of spreading configuration updates across multiple system components and ensuring all components receive consistent configuration information.	Multi-component coordination
file watcher	A mechanism that monitors configuration files for changes and triggers reload processes when modifications are detected.	Automated configuration update trigger
rollback capability	The ability to revert to a previous working configuration when a configuration update causes problems or fails validation.	Safety mechanism for configuration management
two-phase updates	A configuration update process that first validates changes across all components, then applies them only if validation succeeds everywhere.	Consistency mechanism for complex updates

Error Handling and Reliability Terms

Term	Definition	Context
error classification	The process of categorizing errors by type, source, and appropriate handling strategy. Different error types require different response approaches.	Systematic error management
circuit breaker	A design pattern that prevents repeated failed operations by "opening" when failure rates exceed thresholds, failing fast instead of wasting resources on doomed operations.	Failure prevention pattern
failure mode analysis	A systematic catalog and study of all possible ways a system can fail, including their causes, detection methods, and recovery strategies.	Comprehensive reliability engineering
cascade failure	A failure scenario where initial failures trigger additional failures in dependent components, potentially leading to total system failure.	System reliability risk
admission control	The practice of rejecting requests early when the system is operating at or near capacity, rather than accepting requests that cannot be processed successfully.	Load management strategy
fail-fast	A design principle of immediately returning errors when problems are detected, rather than wasting time and resources attempting operations that are likely to fail.	Efficiency and reliability pattern
half-open state	An intermediate circuit breaker state where limited requests are allowed through to test whether a previously failing service has recovered.	Circuit breaker recovery mechanism

Testing and Development Terms

Term	Definition	Context
unit testing	Testing individual components in isolation using mocks and stubs for dependencies, focusing on the correctness of component logic without external dependencies.	Component validation approach
integration testing	End-to-end testing with real HTTP servers and network connections, validating that components work correctly together in realistic scenarios.	System validation approach
milestone validation	Structured verification process that confirms each development phase meets its acceptance criteria before proceeding to the next phase.	Development process checkpoint
failure injection	Controlled simulation of backend failures, network issues, and other error conditions to test system resilience and error handling capabilities.	Reliability testing technique
concurrent load testing	Testing system behavior under multiple simultaneous requests to identify race conditions, resource contention, and scalability limitations.	Performance and correctness testing
health check validation	Verifying that the health checking system accurately detects backend failures and recoveries, and properly excludes unhealthy backends from rotation.	Health system testing
distribution testing	Validating that load balancing algorithms distribute requests appropriately across available backends according to their specifications.	Algorithm correctness testing
algorithm testing	Verifying that each load balancing algorithm implementation behaves correctly under various conditions including edge cases and concurrent access.	Algorithm implementation validation
race condition detection	Identifying thread safety issues that occur when multiple concurrent operations access shared state without proper synchronization.	Concurrency correctness testing
test server pool	A collection of mock backend servers used in integration testing to simulate real backend behavior without requiring actual application servers.	Testing infrastructure
request tracking	Monitoring and logging request distribution patterns during testing to verify that algorithms behave as expected over time.	Testing measurement technique

Debugging and Operations Terms

Term	Definition	Context
request correlation	The practice of tracking individual requests through system logs using unique identifiers, enabling engineers to trace request paths during debugging.	Debugging and monitoring technique
connection leaks	A resource management problem where HTTP connections are opened but never properly closed, eventually exhausting available connection resources.	Common operational problem
false positives	Health check results that incorrectly identify healthy backends as unhealthy, potentially causing unnecessary service degradation.	Health checking accuracy issue
false negatives	Health check results that incorrectly identify unhealthy backends as healthy, allowing failed requests to be forwarded to non-functional backends.	Health checking accuracy issue

Future Extension Terms

Term	Definition	Context
SSL/TLS termination	The process of handling HTTPS encryption and decryption at the load balancer, allowing backend servers to communicate over unencrypted HTTP internally.	Security and performance enhancement
certificate management	Automated handling of SSL certificate lifecycle including acquisition, renewal, and distribution across load balancer instances.	SSL/TLS infrastructure requirement
ACME protocol	Automated Certificate Management Environment - a standard protocol for automatically obtaining and renewing SSL certificates from certificate authorities.	Automated certificate management
path-based routing	Request routing decisions based on URL paths, allowing different application services to be reached through different URL patterns.	Advanced routing capability
header-based routing	Request routing decisions based on HTTP headers like User-Agent, Accept-Language, or custom headers, enabling sophisticated traffic segmentation.	Advanced routing capability
content-aware distribution	Load balancing decisions based on request body content or other application-specific data, enabling routing based on business logic.	Advanced algorithmic routing
monitoring and metrics	Comprehensive system observability including request rates, error rates, response times, and backend health statistics.	Operational visibility enhancement
Prometheus integration	Support for exposing metrics in Prometheus format, enabling integration with industry-standard monitoring and alerting infrastructure.	Standardized metrics format
observability	The overall capability to understand system behavior through metrics, logs, traces, and other telemetry data.	Operational management philosophy

Algorithm and Data Structure Constants

Term	Definition	Context
round_robin	Algorithm identifier constant for sequential backend selection algorithm that cycles through servers in order	Algorithm configuration value
least_connections	Algorithm identifier constant for connection-based selection algorithm that chooses backends with fewest active connections	Algorithm configuration value
weighted_round_robin	Algorithm identifier constant for proportional distribution algorithm based on backend weights	Algorithm configuration value
ip_hash	Algorithm identifier constant for consistent client-to-backend mapping based on client IP address hashing	Algorithm configuration value
random	Algorithm identifier constant for random backend selection algorithm	Algorithm configuration value

Health Check Configuration Constants

Term	Definition	Context
DefaultInterval	Standard time period (30 seconds) between consecutive health check probes for each backend server	Health check timing configuration
DefaultTimeout	Standard maximum time (5 seconds) to wait for health check responses before considering them failed	Health check reliability configuration
DefaultHealthyThreshold	Standard number (2) of consecutive successful health checks required to mark an unhealthy backend as healthy	Health check state transition configuration
DefaultUnhealthyThreshold	Standard number (3) of consecutive failed health checks required to mark a healthy backend as unhealthy	Health check state transition configuration
DefaultPath	Standard URL path (/health) used for backend health check requests	Health check endpoint configuration
DefaultExpectedStatus	Standard HTTP status code (200 OK) expected from healthy backend responses	Health check success criteria

Error Classification Constants

Term	Definition	Context
ErrorTypeConnection	Error classification for network connectivity problems, TCP connection failures, and socket-level issues	Connection failure category
ErrorTypeTimeout	Error classification for operations that exceed configured time limits, including connect timeouts and read timeouts	Timing failure category
ErrorTypeDNS	Error classification for domain name resolution failures and DNS-related connectivity issues	Name resolution failure category
ErrorTypeBackend	Error classification for HTTP-level errors returned by backend servers, including 4xx and 5xx status codes	Backend server failure category
ErrorTypeInternal	Error classification for load balancer internal errors, configuration problems, and algorithmic failures	Load balancer internal failure category
ErrorTypeCapacity	Error classification for resource exhaustion issues including connection limits and memory constraints	Resource limitation failure category

Circuit Breaker State Constants

Term	Definition	Context
CircuitClosed	Normal operation state where requests are allowed through to backends and failures are tracked	Circuit breaker normal state
CircuitOpen	Failing fast state where requests are immediately rejected without attempting to contact backends	Circuit breaker protection state
CircuitHalfOpen	Testing recovery state where limited requests are allowed through to determine if backends have recovered	Circuit breaker recovery state

Architectural Pattern Terms

Term	Definition	Context
middleware pattern	An architectural approach where request processing is organized as a series of composable layers, each handling specific concerns	Request processing architecture
extension points	Architectural locations specifically designed for adding new functionality without modifying existing code	Extensibility design
backward compatibility	The principle that system changes should not break existing functionality or require changes to client code	API and configuration stability
configuration versioning	Managing the evolution of configuration schema over time while supporting older configuration formats	Configuration management strategy
performance impact	The measurable effect of system changes on throughput, latency, and resource utilization	Performance engineering consideration
rule evaluation	The process of applying configured routing rules to incoming requests to determine appropriate backend pools	Advanced routing logic
metrics retention	Managing the storage and lifecycle of historical performance and health metrics data	Operational data management
alerting integration	Automated notification systems that trigger based on metric thresholds and system health conditions	Operational monitoring capability

Implementation and Development Terms

This section covers terms specific to the practical implementation aspects of building a load balancer, including development practices, testing approaches, and operational considerations that bridge the gap between design and working software.

Term	Definition	Context
configuration hot reload	The specific implementation capability to update load balancer settings without dropping existing client connections or interrupting request processing	Runtime configuration management implementation
error response validation	Testing practice that verifies the system generates appropriate HTTP status codes, headers, and response bodies for different failure scenarios	Testing methodology for error handling
performance testing	Systematic measurement of system throughput, latency, and resource utilization under various load conditions to verify scalability requirements	System validation approach
configuration versioning	Implementation strategy for managing configuration schema evolution while maintaining compatibility with existing configurations	Configuration management implementation
extension points	Specific code locations and interfaces designed to allow adding new functionality without modifying core system components	Software architecture principle
backward compatibility	Implementation practice ensuring new software versions continue to work with existing configurations and client integrations	API stability principle

> **Key Design Insight**: The terminology in load balancing reflects the evolution of distributed systems from simple request forwarding to sophisticated traffic management systems. Terms like "graceful degradation" and "circuit breaker" represent hard-won lessons from operating systems at scale, where partial functionality is often more valuable than complete failure.

Acronyms and Abbreviations

Acronym	Full Form	Definition	Context
HTTP	HyperText Transfer Protocol	Application-layer protocol used for communication between clients and servers on the web	Primary protocol handled by the load balancer
HTTPS	HTTP Secure	HTTP over TLS/SSL, providing encrypted communication	Secure variant that may require SSL termination
TCP	Transmission Control Protocol	Reliable, connection-oriented transport layer protocol	Underlying transport for HTTP connections
UDP	User Datagram Protocol	Connectionless transport layer protocol	Alternative transport not used in HTTP load balancing
TLS	Transport Layer Security	Cryptographic protocol for secure communication	Security layer for HTTPS traffic
SSL	Secure Sockets Layer	Predecessor to TLS, term often used interchangeably	Legacy term for TLS in many contexts
DNS	Domain Name System	System for translating domain names to IP addresses	Required for backend server name resolution
IP	Internet Protocol	Network layer protocol for routing packets	Addressing used in IP hash algorithms
URL	Uniform Resource Locator	Web address format specifying protocol, host, and path	Format used for backend server specifications
API	Application Programming Interface	Set of protocols and tools for building software applications	Interface for programmatic configuration
JSON	JavaScript Object Notation	Lightweight data interchange format	Common format for configuration and APIs
CLI	Command Line Interface	Text-based interface for interacting with programs	Common interface for load balancer administration
WAF	Web Application Firewall	Security system that filters HTTP traffic	Related security component not implemented
CDN	Content Delivery Network	Distributed system for delivering web content	Related but separate infrastructure concern
SLA	Service Level Agreement	Contract specifying expected service performance	Business requirement that influences design
SLO	Service Level Objective	Specific measurable goals for service performance	Technical targets for availability and performance
RTO	Recovery Time Objective	Maximum acceptable time to restore service after failure	Disaster recovery requirement
RPO	Recovery Point Objective	Maximum acceptable data loss in terms of time	Data consistency requirement

Protocol and Standards References

Term	Definition	Context
RFC 7230	HTTP/1.1 Message Syntax and Routing specification	Defines HTTP message format and proxy behavior
RFC 7231	HTTP/1.1 Semantics and Content specification	Defines HTTP methods, status codes, and headers
RFC 7232	HTTP/1.1 Conditional Requests specification	Defines caching and conditional request behavior
RFC 7234	HTTP/1.1 Caching specification	Defines HTTP caching behavior relevant to proxies
RFC 7540	HTTP/2 specification	Next-generation HTTP protocol
RFC 8446	TLS 1.3 specification	Current TLS standard for encrypted connections

> **Implementation Note**: Understanding these standards is crucial for implementing HTTP proxy functionality correctly. The load balancer must handle HTTP message parsing, header manipulation, and connection management according to these specifications to ensure compatibility with clients and backend servers.

Common Pitfalls and Anti-Patterns

This section identifies terminology around common mistakes and anti-patterns that developers encounter when building load balancers. Understanding these terms helps prevent common implementation errors.

Term	Definition	Context
request buffering	Anti-pattern of loading entire request bodies into memory before forwarding, causing memory exhaustion with large requests	Should use streaming instead
response buffering	Anti-pattern of loading entire response bodies into memory before forwarding, causing memory and latency issues	Should use streaming instead
sticky connection pools	Problem where HTTP connections become associated with specific backends permanently, reducing load distribution effectiveness	Connection pool management issue
health check storms	Problem where health check frequency overwhelms backend servers, especially during failure scenarios	Health check configuration issue
split-brain scenarios	Situation where different load balancer instances have inconsistent views of backend health, leading to inconsistent routing decisions	Distributed system coordination problem
configuration drift	Problem where load balancer instances have different configurations due to incomplete updates or failures during configuration distribution	Configuration management problem
timeout cascades	Failure mode where timeout configurations create chains of failures, often making problems worse rather than better	Timeout configuration anti-pattern
retry storms	Problem where automatic retries overwhelm already-struggling backend servers, making recovery more difficult	Retry logic anti-pattern

Performance and Scalability Terms

Term	Definition	Context
connection multiplexing	Technique of using a single connection to handle multiple concurrent requests, improving efficiency	HTTP/2 feature not implemented in basic version
connection keep-alive	HTTP feature allowing multiple requests to reuse the same TCP connection, reducing connection establishment overhead	HTTP/1.1 efficiency feature
request pipelining	HTTP feature allowing multiple requests to be sent without waiting for responses, improving throughput	HTTP/1.1 efficiency feature with limited adoption
load shedding	Technique of deliberately dropping requests when system capacity is exceeded, maintaining service for remaining requests	Capacity management strategy
backpressure	Mechanism for controlling request flow when downstream components cannot keep up with request rate	Flow control technique
connection limiting	Practice of restricting the number of concurrent connections per backend to prevent overwhelming servers	Resource protection mechanism
request queuing	Technique of buffering incoming requests when all backends are busy, up to configured limits	Load management strategy
horizontal scaling	Increasing system capacity by adding more backend servers rather than upgrading existing servers	Scalability approach enabled by load balancing
vertical scaling	Increasing system capacity by upgrading individual servers with more CPU, memory, or other resources	Alternative scalability approach

Security-Related Terms

Term	Definition	Context
DDoS protection	Defenses against Distributed Denial of Service attacks that attempt to overwhelm systems with traffic	Security consideration for load balancers
rate limiting	Technique of restricting request frequency from individual clients to prevent abuse and ensure fair resource usage	Traffic shaping and security feature
request sanitization	Process of validating and cleaning request headers and content to prevent injection attacks	Security preprocessing step
backend isolation	Security practice of preventing direct client access to backend servers, routing all traffic through the load balancer	Network security architecture
header injection	Security vulnerability where malicious clients can inject harmful headers through improper header handling	Security risk in header forwarding
request smuggling	Attack technique exploiting differences in HTTP parsing between load balancer and backend servers	Security risk in HTTP proxy implementation
SSL stripping	Attack where encrypted HTTPS connections are downgraded to unencrypted HTTP	Security risk mitigated by proper HTTPS handling

---

Implementation Guidance

The terminology defined in this glossary forms the foundation for precise communication about load balancer design and implementation. When building your load balancer, use these exact terms consistently throughout code, comments, documentation, and discussions.

Terminology Usage Guidelines

Context	Guideline	Example
Code Comments	Use precise technical terms from the glossary	`// Implement round-robin algorithm with atomic counter for thread safety`
Configuration Keys	Use standardized terms for consistency	`"algorithm": "round_robin"` instead of `"strategy": "rotating"`
Log Messages	Use consistent terminology for operational clarity	`"backend_health_changed"` with `"healthy": false`
Error Messages	Use classification terms for systematic error handling	`ErrorTypeConnection` for network-related failures
API Responses	Use standard terminology in JSON responses	`{"backend_pool": [...], "health_state": "healthy"}`
Test Names	Use descriptive terms that map to requirements	`TestRoundRobinAlgorithmThreadSafety`

Common Terminology Mistakes

⚠️ **Pitfall: Inconsistent Terminology** Using different terms for the same concept throughout the codebase creates confusion and makes debugging more difficult. For example, mixing "backend server," "upstream," and "target server" to refer to the same thing makes logs and documentation harder to follow.

How to avoid: Establish a project glossary early and use consistent terminology throughout. Use linters or code review processes to catch terminology inconsistencies.

⚠️ **Pitfall: Overloading Generic Terms** Using generic terms like "server," "error," or "request" without qualification can be ambiguous in complex systems. This makes troubleshooting and system understanding more difficult.

How to avoid: Use qualified terms like "backend server," "ErrorTypeConnection," and "health check request" to provide precise meaning.

⚠️ **Pitfall: Inventing Custom Terminology** Creating project-specific terms for well-established concepts makes it harder for new team members to understand the system and reduces the value of external documentation and resources.

How to avoid: Use industry-standard terminology from this glossary. When you need to extend concepts, build on established terms rather than creating entirely new ones.

Terminology Integration Checklist

Use this checklist when implementing each milestone to ensure consistent terminology usage:

Milestone 1 - HTTP Proxy Foundation:

- [ ] Use "reverse proxy" in code and documentation
- [ ] Implement "header forwarding" with X-Forwarded headers
- [ ] Use "request forwarding" and "response streaming" terminology
- [ ] Handle "ErrorTypeConnection" for backend failures

Milestone 2 - Round Robin Distribution:

- [ ] Use "round_robin" as algorithm identifier constant
- [ ] Implement "backend pool" management
- [ ] Use "atomic operations" for thread-safe counters
- [ ] Use "backend selection" terminology in method names

Milestone 3 - Health Checks:

- [ ] Use "active health checking" terminology
- [ ] Implement "consecutive failure counting"
- [ ] Use "probe" for individual health check requests
- [ ] Handle "recovery detection" logic

Milestone 4 - Additional Algorithms:

- [ ] Use standard algorithm constants: "least_connections," "weighted_round_robin," "ip_hash," "random"
- [ ] Implement "connection tracking" for least connections
- [ ] Use "hash consistency" for IP hash algorithm
- [ ] Support "algorithm switching" at runtime

Code Documentation Standards

When documenting your implementation, use this glossary as the authoritative reference for technical terms. This ensures that your documentation remains consistent with industry standards and facilitates knowledge transfer between team members.

Function Documentation Template:

```go
// SelectBackend chooses a backend server from the healthy backend pool
// using the configured load balancing algorithm. Implements round-robin
// selection with atomic counter operations for thread safety.
//
// Returns nil if no healthy backends are available in the pool.
func (r *RoundRobin) SelectBackend(backends []*Backend) *Backend {
    // Implementation using terminology from glossary
}
```

Configuration Documentation: Use glossary terms in configuration schemas and examples to maintain consistency between code and configuration.

Error Message Documentation: Reference error classification terms when documenting error handling approaches, making it easier to create systematic error handling strategies.

By maintaining consistent terminology throughout your implementation, you create a more maintainable and understandable system that aligns with industry standards and best practices.