Reverse Proxy: Design Document

Overview

A high-performance reverse proxy that accepts client connections, forwards requests to backend servers, and returns responses, solving the key challenge of efficiently managing concurrent connections while providing load balancing, caching, and SSL termination. The system handles the complexities of HTTP protocol parsing, connection pooling, and asynchronous I/O to maximize throughput and minimize latency.

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

Context and Problem Statement

> **Milestone(s):** All milestones - this foundational understanding applies throughout the entire reverse proxy implementation journey.

Building a reverse proxy might seem straightforward at first glance - accept requests, forward them to backend servers, and return responses. However, beneath this simple description lies a complex system that must handle thousands of concurrent connections, parse intricate HTTP protocols, manage connection lifecycles, and maintain high performance under varying load conditions. Understanding why reverse proxies are essential and what makes them architecturally challenging provides the foundation for making informed design decisions throughout the implementation process.

The Reverse Proxy Problem

Think of a reverse proxy as a sophisticated receptionist in a large corporate building. When visitors arrive, the receptionist doesn't just blindly direct them to any available employee. Instead, she maintains detailed knowledge about which departments are currently busy, which employees are out sick, and which meeting rooms have the best equipment for specific types of meetings. She makes intelligent decisions about where to route each visitor based on their needs, current building capacity, and the availability of resources. Additionally, she remembers frequent visitors and can quickly access their preferred meeting locations from her files, avoiding the need to walk through the entire building directory each time.

This analogy captures the essence of what a **reverse proxy** accomplishes in network architecture. Unlike a forward proxy that sits between clients and the internet (acting on behalf of clients), a reverse proxy sits between clients and backend servers, acting on behalf of the servers. The proxy becomes the single point of contact for clients, while intelligently distributing their requests across multiple backend servers based on various criteria such as server health, current load, and request characteristics.

The fundamental challenge that reverse proxies solve is the **many-to-many connection problem**. Consider a web service that needs to handle 10,000 concurrent users, but each backend server can only handle 1,000 connections effectively. Without a reverse proxy, you would need to either over-provision individual servers (wasteful and expensive) or implement complex client-side load balancing logic (brittle and difficult to update). The reverse proxy elegantly solves this by maintaining a smaller number of persistent connections to backend servers while handling the full volume of client connections on the frontend.

However, implementing this solution introduces several architectural challenges that make reverse proxy development particularly complex:

Connection State Management Challenge: A reverse proxy must simultaneously manage two distinct connection contexts - the client-facing connection and the backend-facing connection. Each connection exists in its own lifecycle with different states

(connecting, reading request, forwarding, reading response, writing response, closing), and these states must be carefully synchronized. The proxy cannot simply pass bytes between connections because it needs to understand the HTTP protocol structure to make intelligent routing decisions and handle error scenarios gracefully.

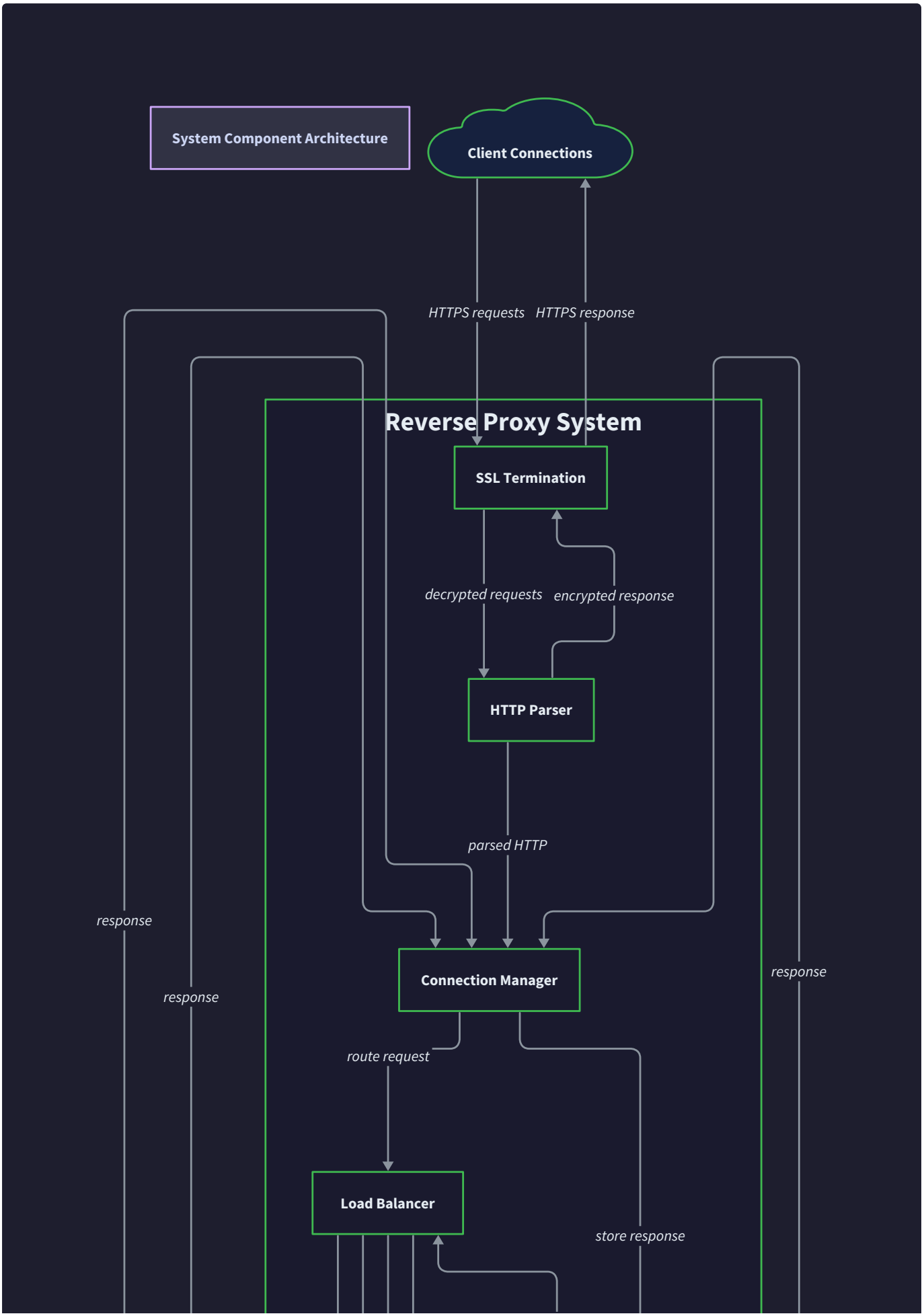
Protocol Parsing and Transformation Challenge: HTTP appears simple on the surface, but the protocol contains numerous edge cases and variations that must be handled correctly. The proxy must parse incoming requests completely enough to extract routing information (Host headers, URL paths), while simultaneously preserving the original request structure for forwarding. Additionally, the proxy must handle protocol version differences - a client might connect via HTTP/2 while the backend server only supports HTTP/1.1, requiring protocol translation.

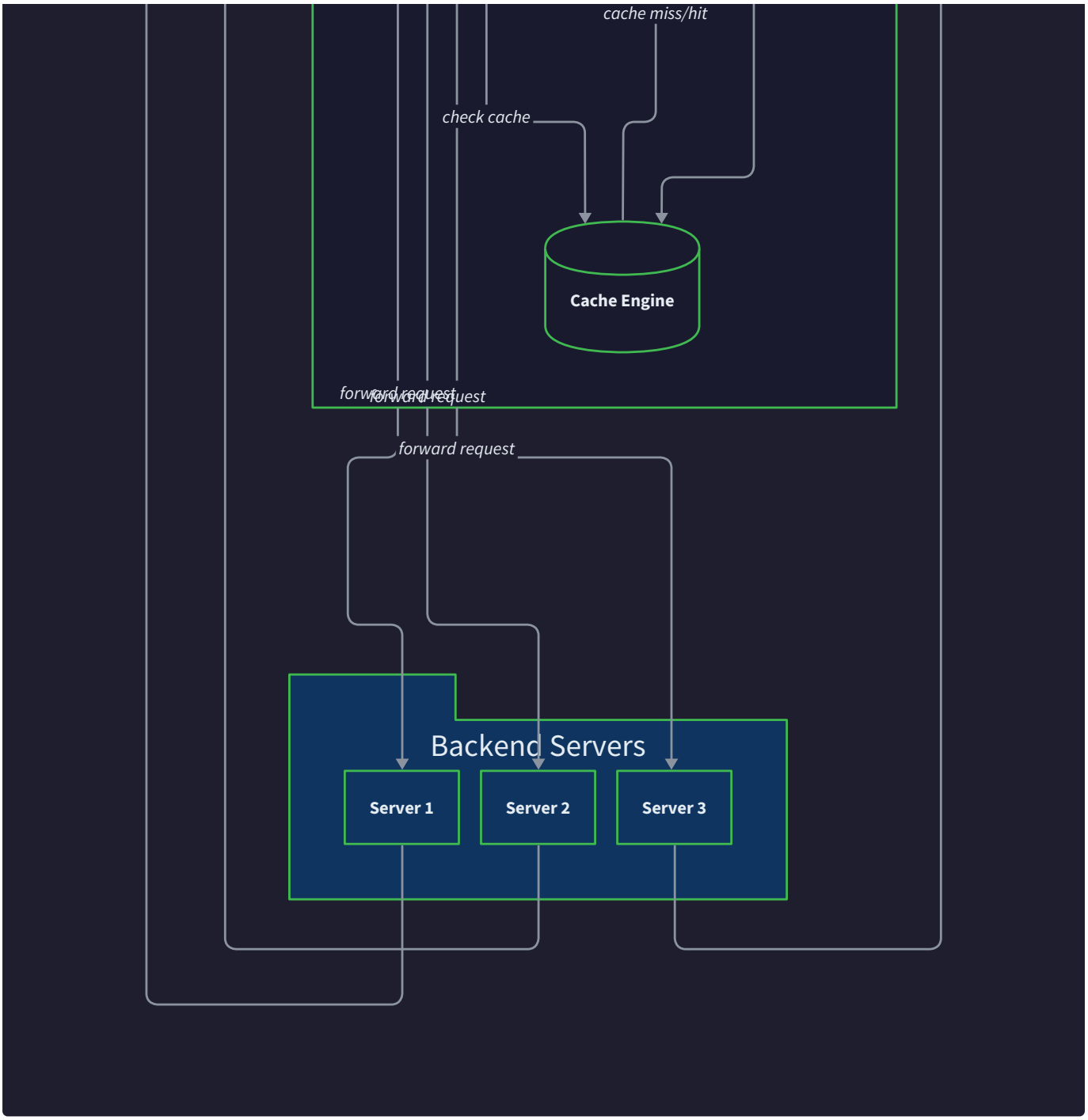
Asynchronous I/O Complexity: To achieve high performance, a reverse proxy cannot afford to block threads waiting for I/O operations. This necessitates an event-driven architecture using asynchronous I/O primitives like `epoll` on Linux or `kqueue` on BSD systems. However, asynchronous programming introduces complexity in error handling, connection lifecycle management, and state tracking. The proxy must handle partial reads and writes gracefully, manage timeouts across multiple concurrent operations, and ensure that connection state remains consistent even when I/O operations complete in unpredictable orders.

Memory Management and Resource Limits: Each active connection requires memory for buffers, connection state, and potentially cached data. A reverse proxy handling thousands of connections must carefully manage memory usage to avoid exhaustion. This includes implementing efficient buffer reuse strategies, setting appropriate limits on request sizes and connection counts, and ensuring that slow or malicious clients cannot consume unbounded resources through techniques like slowloris attacks.

Backend Health and Failure Handling: Unlike simple load balancers that can assume backend servers are always available, a production reverse proxy must handle backend failures gracefully. This includes detecting when backend servers become unavailable, removing them from the active pool, and implementing retry logic with appropriate backoff strategies. The proxy must also handle partial failures - situations where a backend server accepts a connection but fails during request processing.

The interplay between these challenges creates emergent complexity. For example, when a backend server fails during request processing, the proxy must simultaneously clean up the backend connection, maintain the client connection, generate an appropriate error response, update its load balancing state, and ensure that any cached data related to the failed request is invalidated. Managing these interdependent concerns requires careful architectural planning and robust error handling throughout the system.


System Component Architecture
Client Connections
HTTPS requests
HTTPS response
Reverse Proxy System
SSL Termination
decrypted requests
encrypted response
HTTP Parser
parsed HTTP
response
response
response
Connection Manager
route request
Load Balancer
store response



Existing Solutions Analysis

Understanding how existing reverse proxy implementations approach these challenges provides valuable context for design decisions. Each solution makes different trade-offs between performance, complexity, and feature richness, reflecting the diverse requirements of their target environments.

NGINX represents the dominant approach in production environments, built around an event-driven architecture using a master-worker process model. The master process handles configuration loading and worker process management, while worker processes handle actual request processing using asynchronous I/O. NGINX achieves exceptional performance by avoiding thread-per-connection models and instead using a small number of worker processes (typically one per CPU core) that handle thousands of connections each through event loops.

Aspect	NGINX Approach	Strengths	Weaknesses
Architecture	Master-worker with event loops	Excellent performance, proven scalability	Complex configuration, C codebase hard to modify
Memory Model	Shared memory pools	Low memory overhead per connection	Memory pool tuning requires expertise
Configuration	Declarative config files	Powerful and flexible	Complex syntax, requires restarts for changes
Extensibility	C modules	High performance modules	High barrier to entry for custom logic

NGINX's design philosophy prioritizes performance and memory efficiency above ease of development. Its configuration system, while powerful, requires deep understanding of HTTP semantics and NGINX-specific concepts. The C-based module system enables high-performance extensions but creates a significant barrier for developers who need custom business logic in their reverse proxy.

HAProxy takes a different approach, focusing specifically on load balancing with extensive health checking and traffic management features. Built around a single-threaded event loop model, HAProxy excels at connection management and provides sophisticated algorithms for backend server selection.

Aspect	HAProxy Approach	Strengths	Weaknesses
Architecture	Single-threaded event loop	Predictable performance, simpler debugging	Limited CPU scalability on multi-core systems
Load Balancing	Advanced algorithms	Comprehensive health checks, detailed statistics	Less general-purpose than NGINX
Configuration	Specialized DSL	Optimized for load balancing use cases	Learning curve for complex routing rules
Observability	Built-in statistics	Excellent monitoring capabilities	Limited extensibility for custom metrics

HAProxy's single-threaded model provides predictable performance characteristics and simplifies reasoning about connection state, but limits its ability to fully utilize multi-core systems for CPU-intensive operations like SSL termination.

Envoy represents a modern approach designed for cloud-native environments, built with observability, dynamic configuration, and service mesh integration as core requirements. Written in C++, Envoy uses a multi-threaded architecture with careful attention to lock-free data structures and thread-local storage.

Aspect	Envoy Approach	Strengths	Weaknesses
Architecture	Multi-threaded with thread-local workers	Good multi-core utilization	More complex than single-threaded designs
Configuration	gRPC-based dynamic config	Can update configuration without restarts	Requires additional infrastructure (control plane)
Extensibility	C++ filters and WASM	Flexible extension model	C++ expertise required for native extensions
Observability	Built-in metrics and tracing	Excellent for microservices environments	Higher resource overhead than simpler proxies

Envoy's dynamic configuration capabilities enable sophisticated deployment patterns but require additional infrastructure to manage the configuration distribution, making it more complex for simple use cases.

Cloud Load Balancers (like AWS ALB, Google Cloud Load Balancing) represent fully managed solutions that abstract away the implementation complexity entirely. These services handle scaling, health checking, and SSL certificate management automatically, but provide limited customization options.

Aspect	Cloud Load Balancers	Strengths	Weaknesses
Architecture	Fully managed	No operational overhead	Vendor lock-in, limited customization
Scalability	Automatic scaling	Handles traffic spikes transparently	Cost can be unpredictable
Features	Integrated with cloud services	SSL management, DDoS protection	Less control over routing logic
Configuration	Web UI or APIs	Easy to get started	Advanced configurations may hit limitations

The managed approach eliminates operational complexity but sacrifices the ability to implement custom business logic or optimize for specific use cases.

> **Key Insight**: The choice between these approaches depends heavily on the specific requirements and constraints of your environment. NGINX excels in high-performance scenarios with relatively static configurations. HAProxy provides superior load balancing algorithms and health checking for traditional server environments. Envoy offers the most flexibility for dynamic, cloud-native environments. Managed solutions work well when standardized features meet your requirements and operational simplicity is prioritized over customization.

For our educational implementation, we need to balance learning value with implementation complexity. Our design will draw inspiration from NGINX's event-driven architecture for its proven performance characteristics, while incorporating some of Envoy's modularity concepts to make the codebase more approachable for learning purposes.

> **Decision: Event-Driven Architecture with Modular Components**
>
> - **Context**: We need an architecture that demonstrates production-quality concepts while remaining understandable for educational purposes
> - **Options Considered**:
>   1. Thread-per-connection model (simple but poor performance)
>   2. Single-threaded event loop (simple but limited scalability)
>   3. Multi-threaded event-driven with worker pools (complex but realistic)
> - **Decision**: Multi-threaded event-driven architecture with clearly separated components
> - **Rationale**: This approach teaches modern high-performance patterns while keeping each component focused and testable. The modular design allows learners to understand each piece independently before seeing how they interact.
> - **Consequences**: Higher initial complexity but better representation of production systems. Each component can be developed and tested independently, making the learning process more manageable.

The architecture decision above shapes our entire implementation approach. Rather than building a monolithic proxy, we'll create distinct components (HTTP Parser, Connection Manager, Load Balancer, Cache Engine, SSL Termination) that communicate through well-defined interfaces. This mirrors how production systems achieve maintainability and testability while handling the inherent complexity of high-performance network programming.

Understanding these existing solutions and their trade-offs provides essential context for the design decisions we'll make throughout our implementation. Each component we build will face similar challenges to those solved by NGINX, HAProxy, and Envoy, but our solutions will prioritize learning clarity and implementation understandability while still demonstrating the fundamental concepts that make production reverse proxies successful.

Implementation Guidance

This implementation guidance bridges the gap between understanding reverse proxy concepts and building a working system. The recommendations here focus on practical technology choices and project organization that will support learning while building toward a production-quality architecture.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Parsing	Manual state machine with character-by-character parsing	HTTP parser library (http-parser, picohttpparser)
Async I/O	Basic `select()` or `poll()` with non-blocking sockets	Platform-specific `epoll` / `kqueue` with event libraries
SSL/TLS	OpenSSL with basic certificate loading	mbedTLS or BoringSSL with advanced features
Configuration	Simple key-value file parsing	JSON/YAML with schema validation
Logging	Printf-style logging to stdout/files	Structured logging with log levels and rotation
Memory Management	Standard malloc/free with careful tracking	Custom memory pools and arena allocators

For learning purposes, start with the simple options to understand the underlying concepts, then migrate to advanced options as your understanding deepens. The simple options expose more of the fundamental mechanisms, while advanced options provide production-ready performance and features.

B. Recommended Project Structure:

```
reverse-proxy/
├── src/
│   ├── main.c                  ← Entry point and main event loop
│   ├── config/
│   │   ├── config.h            ← Configuration data structures
│   │   ├── config.c            ← Configuration file parsing
│   │   └── config_test.c       ← Configuration parsing tests
│   ├── http/
│   │   ├── parser.h            ← HTTP parsing interface
│   │   ├── parser.c            ← HTTP request/response parsing
│   │   ├── message.h           ← HTTP message data structures
│   │   └── parser_test.c       ← HTTP parsing unit tests
│   ├── connection/
│   │   ├── manager.h           ← Connection management interface
│   │   ├── manager.c           ← Connection lifecycle and pooling
│   │   ├── pool.h              ← Connection pool data structures
│   │   └── connection_test.c   ← Connection management tests
│   ├── loadbalancer/
│   │   ├── balancer.h          ← Load balancing algorithms interface
│   │   ├── roundrobin.c        ← Round-robin implementation
│   │   ├── healthcheck.h       ← Health checking interface
│   │   └── balancer_test.c     ← Load balancing tests
│   ├── cache/
│   │   ├── cache.h             ← Cache engine interface
│   │   ├── lru.c               ← LRU cache implementation
│   │   ├── cache_control.h     ← HTTP cache control parsing
│   │   └── cache_test.c        ← Cache functionality tests
│   ├── ssl/
│   │   ├── termination.h       ← SSL termination interface
│   │   ├── termination.c       ← TLS context and SNI handling
│   │   └── ssl_test.c          ← SSL termination tests
│   └── util/
│       ├── buffer.h            ← Dynamic buffer management
│       ├── buffer.c            ← Buffer implementation
│       ├── logger.h            ← Logging utilities
│       ├── logger.c            ← Logging implementation
│       └── hashtable.h         ← Hash table for caching
├── config/
│   ├── proxy.conf              ← Example configuration file
│   └── ssl/
│       ├── server.crt          ← Example SSL certificate
│       └── server.key          ← Example private key
├── tests/
│   ├── integration/            ← End-to-end tests
│   └── fixtures/               ← Test data and mock servers
├── docs/
│   └── api.md                  ← Component interfaces documentation
├── Makefile                    ← Build configuration
└── README.md                   ← Setup and usage instructions
```

This structure separates concerns clearly, making it easier to understand and test individual components. Each component directory contains its interface header, implementation, and tests, following the principle that related code should live together.

C. Infrastructure Starter Code:

The following utilities handle cross-cutting concerns that every component needs, allowing you to focus on the core reverse proxy logic rather than reimplementing basic infrastructure.

Dynamic Buffer Management (`src/util/buffer.h` and `src/util/buffer.c`):

```c
#include <stddef.h>

#include <stdbool.h>


// Dynamic buffer for handling variable-length HTTP messages

typedef struct {

    char *data;          // Buffer memory

    size_t capacity;     // Total allocated size

    size_t length;       // Current data length

    size_t position;     // Current read/write position

} Buffer;


// Initialize a new buffer with initial capacity

Buffer* buffer_create(size_t initial_capacity);


// Ensure buffer has at least required_capacity bytes available

bool buffer_ensure_capacity(Buffer *buf, size_t required_capacity);


// Append data to buffer, growing if necessary

bool buffer_append(Buffer *buf, const char *data, size_t length);


// Read data from buffer starting at current position

size_t buffer_read(Buffer *buf, char *dest, size_t max_length);


// Reset buffer position to beginning for reading

void buffer_rewind(Buffer *buf);


// Clear buffer contents but preserve allocated memory

void buffer_clear(Buffer *buf);


// Free buffer and all associated memory

void buffer_destroy(Buffer *buf);
```

Logging Infrastructure (`src/util/logger.h` and `src/util/logger.c`):

```c
#include <stdio.h>

typedef enum {
    LOG_DEBUG = 0,
    LOG_INFO = 1,
    LOG_WARN = 2,
    LOG_ERROR = 3
} LogLevel;

// Initialize logging system with minimum level and output file
bool logger_init(LogLevel min_level, FILE *output);

// Log formatted message with specified level
void logger_log(LogLevel level, const char *component, const char *format, ...);

// Convenience macros for common logging patterns
#define LOG_DEBUG(component, ...) logger_log(LOG_DEBUG, component, __VA_ARGS__)
#define LOG_INFO(component, ...)  logger_log(LOG_INFO, component, __VA_ARGS__)
#define LOG_WARN(component, ...)  logger_log(LOG_WARN, component, __VA_ARGS__)
#define LOG_ERROR(component, ...) logger_log(LOG_ERROR, component, __VA_ARGS__)

// Shutdown logging system and close files
void logger_shutdown(void);
```

Hash Table for Caching (`src/util/hashtable.h`):

```c
#include <stddef.h>
#include <stdbool.h>

typedef struct HashTable HashTable;

// Create hash table with specified initial capacity
HashTable* hashtable_create(size_t initial_capacity);

// Insert key-value pair, returns false if out of memory
bool hashtable_put(HashTable *ht, const char *key, void *value);

// Retrieve value for key, returns NULL if not found
void* hashtable_get(HashTable *ht, const char *key);

// Remove entry for key, returns true if key existed
bool hashtable_remove(HashTable *ht, const char *key);

// Get current number of entries
size_t hashtable_size(HashTable *ht);

// Free hash table and all keys (values must be freed by caller)
void hashtable_destroy(HashTable *ht);
```

These utilities provide the foundational infrastructure that every reverse proxy component requires. The buffer management handles dynamic memory allocation for HTTP messages, the logging system enables debugging and operational monitoring, and the hash table supports caching and configuration lookups.

D. Core Logic Skeleton Code:

For the main event loop and component coordination, provide structure without implementation:

Main Event Loop (`src/main.c`):

```c
#include "config/config.h"

#include "connection/manager.h"

#include "util/logger.h"


int main(int argc, char *argv[]) {

    // TODO 1: Parse command line arguments for config file path

    // TODO 2: Load configuration from file using config_load()

    // TODO 3: Initialize logger with configured log level and output

    // TODO 4: Create connection manager with configured listen address

    // TODO 5: Initialize SSL contexts if SSL termination is enabled

    // TODO 6: Enter main event loop calling connection_manager_run()

    // TODO 7: Handle shutdown signals gracefully (SIGTERM, SIGINT)

    // TODO 8: Clean up resources and close all connections before exit

    // Hint: Use signal handlers to set a global shutdown flag

    return 0;

}
```

Configuration Loading (`src/config/config.c`):

```c
#include "config.h"


ProxyConfig* config_load(const char *config_file_path) {

    // TODO 1: Open configuration file and handle file not found errors

    // TODO 2: Parse listen address and port from config

    // TODO 3: Parse backend server list with weights and health check URLs

    // TODO 4: Parse SSL settings including certificate paths and cipher suites

    // TODO 5: Parse cache settings including max size and default TTL

    // TODO 6: Validate all configuration values for sanity

    // TODO 7: Return populated ProxyConfig struct or NULL on error

    // Hint: Start with simple key=value format, expand to JSON later

    return NULL;

}
```

E. Language-Specific Hints for C:

- **Socket Programming**: Use `socket(AF_INET, SOCK_STREAM, 0)` for TCP sockets, set `SO_REUSEADDR` to avoid "Address already in use" errors during development
- **Non-blocking I/O**: Set sockets to non-blocking mode with `fcntl(sockfd, F_SETFL, O_NONBLOCK)` and handle `EAGAIN` / `EWOULDBLOCK` return codes
- **Memory Management**: Always pair `malloc()` with `free()`, consider using `valgrind` to detect memory leaks during development
- **String Handling**: Use `strncpy()` instead of `strcpy()` and always null-terminate strings manually, HTTP headers are case-insensitive so implement case-insensitive comparison
- **Error Handling**: Check return values from all system calls, use `errno` and `strerror()` to provide meaningful error messages
- **Event-Driven I/O**: Start with `select()` for portability, then optimize with `epoll()` on Linux or `kqueue()` on BSD systems for better performance

F. Milestone Checkpoint:

After implementing the basic project structure and infrastructure components:

What to Run:

```bash
make clean && make

./reverse-proxy config/proxy.conf
```

Expected Output:

```
[INFO] Starting reverse proxy on 0.0.0.0:8080
[INFO] Loaded 3 backend servers from configuration
[INFO] SSL termination enabled with certificate: config/ssl/server.crt
[INFO] Cache enabled with 100MB maximum size
[INFO] Ready to accept connections
```

What to Verify Manually:

1. Start a simple HTTP server on port 8081 (can use Python's `python -m http.server 8081`)
2. Configure your proxy to forward to localhost:8081
3. Send a request to your proxy: `curl -v http://localhost:8080/`
4. Verify the proxy forwards the request and returns the backend's response
5. Check logs to see request processing information

Signs Something is Wrong:

- **"Address already in use" error**: Another process is using the port, or you forgot to set `SO_REUSEADDR`
- **Connection refused**: Check that backend servers are actually running and accessible
- **Segmentation fault**: Likely a memory management issue, run with `valgrind` to identify the problem
- **Hanging requests**: Probably an issue with non-blocking I/O handling or event loop logic

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Proxy accepts connections but never responds	Event loop not processing read events	Add debug logging to event handlers	Check `select()`/`epoll()` event masks
High CPU usage with no traffic	Busy loop in event handling	Profile with `top` or `htop`	Add proper blocking on empty event queues
Memory usage grows continuously	Memory leaks in connection handling	Run with `valgrind --leak-check=full`	Review buffer allocation and cleanup
Requests timeout randomly	Backend connection pool exhaustion	Log connection pool statistics	Implement connection limits and queuing
SSL handshake failures	Certificate or cipher suite issues	Check OpenSSL error messages	Verify certificate validity and supported ciphers

This implementation guidance provides a solid foundation for beginning the reverse proxy implementation. The modular structure and infrastructure components allow you to focus on learning the core concepts of each component without getting bogged down in repetitive utility code. As you implement each milestone, you'll build upon this foundation while gaining deep understanding of how production reverse proxies handle the complexities of high-performance network programming.

Goals and Non-Goals

> **Milestone(s):** All milestones - this section establishes the scope boundaries that guide implementation decisions throughout the entire project lifecycle.

Building a reverse proxy is like constructing a high-performance traffic control system for a busy metropolitan area. Just as a traffic control center must decide which roads to build, which intersections to prioritize, and which advanced features to include versus which to defer, our reverse proxy implementation requires clear boundaries about what functionality to include in our initial design versus what to leave for future iterations.

The challenge with reverse proxy development lies in the vast scope of potential features. Modern production reverse proxies like NGINX and HAProxy include hundreds of configuration directives, dozens of load balancing algorithms, sophisticated rate limiting, WebSocket proxying, HTTP/3 support, Lua scripting capabilities, and complex SSL certificate management. Attempting to implement everything would result in an overly complex system that never reaches completion. Instead, we must carefully choose a focused subset of functionality that provides a solid foundation while remaining achievable within reasonable development timeframes.

Think of this goals definition as drawing the blueprint boundaries for our traffic control system. We're deciding whether to build a basic intersection with traffic lights (core HTTP forwarding), add highway on-ramps (load balancing), include parking structures (caching), and install toll booths (SSL termination), while consciously deciding not to build subway systems (complex routing rules) or airport terminals (advanced features) in our initial version.

Core Functional Goals

Our reverse proxy implementation focuses on five fundamental capabilities that represent the essential building blocks of any production reverse proxy system. These goals directly align with our milestone structure and provide measurable success criteria for each development phase.

HTTP Request Forwarding and Response Relay

The primary goal involves implementing a robust HTTP proxy core that can accept incoming client connections, parse HTTP requests completely, forward those requests to configured backend servers, and relay the backend responses back to clients. This functionality forms the foundation upon which all other features build.

The system must handle both HTTP/1.1 and HTTP/2 protocol versions, supporting the complete request-response cycle including method extraction, header parsing, body handling, and proper connection management. This includes correctly implementing HTTP semantics for chunked transfer encoding, content-length validation, and connection keep-alive behavior.

Success criteria include the ability to proxy any valid HTTP request without modification to the request semantics, maintain request fidelity during forwarding, and properly handle various HTTP methods including GET, POST, PUT, DELETE, and HEAD requests. The proxy must preserve all client headers while adding appropriate forwarding headers like `X-Forwarded-For` and `Via`.

Load Balancing and Backend Distribution

The second core goal implements intelligent request distribution across multiple backend servers using proven load balancing algorithms. This capability transforms our simple proxy into a scalable traffic distribution system capable of handling high-volume applications.

The implementation must support multiple distribution strategies including round-robin for even distribution, least-connections for optimal resource utilization, weighted distribution for heterogeneous backend capacity, and IP hash for session affinity requirements. Each algorithm addresses different operational needs and scaling patterns.

Health checking integration ensures that only healthy backends receive traffic by implementing periodic health checks against configured endpoints, automatically removing failed backends from the active pool, and restoring them when health checks succeed. This provides automatic failover capabilities without manual intervention.

Connection Pooling and Resource Management

The third goal focuses on efficient resource utilization through persistent connection management. Rather than establishing new TCP connections for each request, the system maintains pools of reusable connections to backend servers, dramatically reducing connection establishment overhead and improving overall throughput.

Connection pooling involves maintaining separate pools for each backend server, implementing configurable pool size limits to prevent resource exhaustion, handling connection timeouts through idle connection eviction, and validating connection health before reuse. The pooling strategy must balance resource efficiency with connection freshness.

The system must gracefully handle pool exhaustion scenarios by either creating temporary connections or queuing requests, depending on configuration policies. Connection lifecycle management includes proper cleanup of idle connections and handling of backend connection failures.

HTTP Response Caching

The fourth goal implements intelligent response caching to reduce backend load and improve client response times. The caching system must respect HTTP caching semantics while providing significant performance improvements for cacheable content.

Cache implementation includes generating unique cache keys from request attributes, storing responses with appropriate TTL values based on `Cache-Control` headers, supporting cache invalidation through both time-based expiry and explicit purge operations, and properly handling conditional requests using ETags and `If-Modified-Since` headers.

The system must distinguish between cacheable and non-cacheable responses, respecting directives like `no-cache`, `no-store`, and `private`. Cache storage utilizes memory-based storage with LRU eviction policies when size limits are reached.

SSL Termination and Certificate Management

The fifth goal provides secure HTTPS connection handling through SSL/TLS termination at the proxy layer. This allows backend servers to operate with plain HTTP while presenting encrypted endpoints to clients, simplifying backend infrastructure while maintaining security.

SSL termination includes loading X.509 certificates and private keys from PEM files, supporting Server Name Indication (SNI) for multi-domain hosting, implementing strong cipher suite selection, enforcing minimum TLS version requirements, and providing automatic HTTP-to-HTTPS redirects for security.

Certificate management supports runtime certificate reloading without process restarts, proper certificate chain validation, and secure private key handling. The system must support multiple certificates for different domains while selecting the appropriate certificate based on SNI during TLS handshake.

Performance and Quality Goals

Beyond functional capabilities, our reverse proxy must meet specific performance and reliability standards that enable production deployment scenarios. These quality goals ensure the system can handle real-world traffic patterns and operational requirements.

Throughput and Latency Targets

The system must handle at least 10,000 concurrent connections with sub-millisecond proxy overhead added to backend response times. This performance target ensures the proxy doesn't become a bottleneck in high-traffic applications while maintaining responsive user experiences.

Latency measurements exclude backend processing time and network transit delays, focusing specifically on proxy overhead including request parsing, routing decisions, connection management, and response forwarding. The target assumes properly tuned operating system settings and adequate hardware resources.

Memory usage must remain bounded and predictable under load, with configurable limits for connection pools, cache storage, and request buffering. The system should avoid memory leaks and implement proper resource cleanup to support long-running operation.

Reliability and Error Handling

The proxy must provide robust error handling and graceful degradation capabilities, ensuring that individual backend failures don't affect overall system availability. Error handling includes proper timeout management, circuit breaker patterns for failing backends, and appropriate error responses to clients.

Logging and monitoring integration provides operational visibility through structured log output, metrics collection for key performance indicators, and debugging information for troubleshooting. The system must support configurable log levels and external monitoring system integration.

Recovery mechanisms handle various failure scenarios including backend connection failures, SSL certificate errors, configuration reload failures, and resource exhaustion conditions. Each failure mode requires specific detection and recovery strategies.

Explicit Non-Goals

Clearly defining what our reverse proxy will NOT implement is equally important as defining its goals. These non-goals establish scope boundaries that prevent feature creep and maintain implementation focus on core capabilities.

Advanced Routing and Traffic Management

Our implementation explicitly excludes complex routing features found in advanced reverse proxies and API gateways. We will not implement URL rewriting capabilities, complex path-based routing rules, regular expression matching for request routing, or header-based routing decisions beyond basic host header handling.

Rate limiting and traffic shaping features are excluded from the initial implementation. While these capabilities are valuable in production environments, they add significant complexity to the core proxy logic and can be implemented as future extensions without affecting the fundamental architecture.

Authentication and authorization features including OAuth integration, JWT validation, API key management, and access control lists are outside our scope. These features typically require integration with external identity systems and add substantial

complexity to the request processing pipeline.

Protocol Extensions and Advanced Features

WebSocket proxying support is explicitly excluded due to the different connection semantics and state management requirements. WebSocket connections require long-lived, bidirectional communication that differs significantly from standard HTTP request-response patterns.

HTTP/3 and QUIC protocol support are not included in the initial implementation. While these protocols represent the future of web communication, they require specialized libraries and significantly more complex implementation than HTTP/1.1 and HTTP/2.

Compression and content modification features including gzip compression, content transcoding, and response body modification are excluded. These features require streaming content processing and add complexity to the response handling pipeline.

Operational and Management Features

Administrative interfaces including REST APIs for configuration management, web-based dashboards, and runtime configuration modification are not included. The system will use file-based configuration with restart-required configuration changes.

Advanced monitoring features including Prometheus metrics export, distributed tracing integration, and detailed performance profiling are excluded from the core implementation. Basic logging provides operational visibility without requiring complex monitoring infrastructure.

Clustering and high availability features including configuration synchronization across multiple proxy instances, shared state management, and automatic failover are outside our scope. The system focuses on single-instance operation with external load balancing for high availability.

Security and Compliance Features

Advanced security features including request filtering, SQL injection detection, cross-site scripting protection, and other web application firewall capabilities are excluded. These features require deep content inspection and complex rule engines.

Certificate authority integration, automatic certificate provisioning through ACME protocol, and certificate rotation automation are not included. The system supports manual certificate management through file-based configuration.

Compliance features including detailed audit logging, request/response recording, and regulatory compliance reporting are outside our implementation scope.

Architecture Decision Records

The following decisions establish fundamental design principles that guide implementation choices throughout the development process.

> **Decision: Event-Driven Architecture with Asynchronous I/O**
>
> - **Context**: Reverse proxies must handle thousands of concurrent connections efficiently without blocking on slow operations like backend requests or SSL handshakes.
> - **Options Considered**: Thread-per-connection model, process-per-connection model, event-driven model with epoll/kqueue
> - **Decision**: Event-driven architecture using non-blocking I/O and event loops
> - **Rationale**: Thread-per-connection models consume excessive memory (8KB+ per thread stack) and suffer context switching overhead with thousands of connections. Event-driven models achieve higher concurrency with lower resource usage.
> - **Consequences**: Requires careful state management and non-blocking operation implementation, but enables handling 10,000+ concurrent connections efficiently.

Architecture Model	Memory per Connection	Context Switch Overhead	Max Connections	Chosen
Thread-per-connection	~8KB+ stack	High with many threads	~1,000	No
Process-per-connection	~1MB+ per process	Very high	~100	No
Event-driven	~1KB connection state	Minimal	10,000+	**Yes**

Decision: In-Memory Caching with LRU Eviction

- **Context**: Response caching requires balancing cache hit rates with memory usage and implementation complexity.
- **Options Considered**: In-memory hash table, Redis external cache, disk-based cache with memory index
- **Decision**: In-memory hash table with LRU eviction policy
- **Rationale**: In-memory storage provides microsecond access times without network round trips. LRU eviction provides good hit rates for web traffic patterns. External caches add network latency and operational complexity.
- **Consequences**: Cache size limited by available memory, but provides excellent performance and simple implementation.

Caching Strategy	Access Latency	Scalability	Operational Complexity	Chosen
In-memory hash table	Microseconds	Limited by RAM	Low	**Yes**
External Redis	Milliseconds	High	Medium	No
Disk with memory index	Milliseconds	Very high	High	No

Decision: File-Based Configuration with Restart-Required Reload

- **Context**: Configuration management requires balancing operational flexibility with implementation complexity.
- **Options Considered**: File-based with restart, file-based with hot reload, API-based runtime configuration
- **Decision**: File-based configuration requiring process restart for changes
- **Rationale**: File-based configuration integrates well with configuration management tools and version control. Hot reload requires complex state synchronization and error handling. API-based configuration requires additional security and persistence mechanisms.
- **Consequences**: Configuration changes require brief service interruption, but implementation remains simple and reliable.

Configuration Method	Change Latency	Implementation Complexity	Version Control	Chosen
File + restart	~1 second downtime	Low	Excellent	**Yes**
File + hot reload	Immediate	High	Good	No
API-based runtime	Immediate	Very high	Poor	No

Success Criteria and Validation

Each goal requires specific, measurable criteria that validate successful implementation and provide clear checkpoints for development progress.

Functional Validation Criteria

HTTP proxy functionality validation involves testing complete request-response cycles with various HTTP methods, header combinations, and body types. Success requires bit-for-bit response fidelity compared to direct backend communication, proper forwarding header addition, and correct error handling for invalid requests.

Load balancing validation tests distribution algorithms with multiple backend servers, verifying even distribution for round-robin, connection-aware routing for least-connections, and proper weight respect for weighted algorithms. Health checking validation requires automatic backend removal during simulated failures and restoration when health checks succeed.

Connection pooling validation measures connection reuse rates, pool size management under load, and proper connection cleanup during idle periods. Cache validation tests hit rates for repeated requests, proper TTL expiration, and correct cache-control header respect.

SSL termination validation requires successful TLS handshake completion, proper certificate selection based on SNI, and secure cipher suite negotiation. Backend communication validation ensures plain HTTP forwarding works correctly after SSL termination.

Performance Validation Benchmarks

Throughput testing uses tools like `ab` (Apache Bench) or `wrk` to generate sustained load and measure requests per second under various connection counts. Target performance includes maintaining 95th percentile response times under 1ms proxy overhead with 1,000 concurrent connections.

Memory usage testing validates bounded memory growth under sustained load, proper cleanup of idle resources, and absence of memory leaks during extended operation. Connection pool efficiency testing measures connection reuse rates and pool hit ratios.

Cache effectiveness testing measures hit rates for typical web traffic patterns and validates proper memory usage under various cache sizes. SSL performance testing ensures TLS handshake overhead remains acceptable under load.

Implementation Guidance

The goals and non-goals established in this section guide every architectural and implementation decision throughout the reverse proxy development process. Understanding these boundaries helps maintain focus on core capabilities while avoiding scope creep that could derail the project.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Parsing	Custom parser with state machine	libhttp-parser library
TLS Implementation	OpenSSL with basic configuration	mbedTLS with advanced features
Event Loop	select() system call	epoll (Linux) / kqueue (BSD)
Configuration Format	INI-style key-value pairs	YAML with validation schema
Logging Framework	fprintf to stderr/file	Structured JSON logging

Core Configuration Structure

The configuration system forms the foundation that enables all proxy functionality. A well-designed configuration structure supports the goals while maintaining simplicity and clarity.

```c
// Complete configuration structure supporting all goals
typedef struct {
    // Server configuration
    char listen_address[256];
    int listen_port;
    int max_connections;
    int worker_threads;

    // SSL configuration
    bool ssl_enabled;
    char ssl_cert_path[512];
    char ssl_key_path[512];
    char ssl_ca_path[512];
    int ssl_min_version;  // Minimum TLS version

    // Backend server configuration
    int backend_count;
    struct {
        char address[256];
        int port;
        int weight;
        int max_connections;
        int health_check_interval;
        char health_check_path[256];
    } backends[32];

    // Load balancing configuration
    enum {
        LB_ROUND_ROBIN,
        LB_LEAST_CONNECTIONS,
        LB_WEIGHTED_ROUND_ROBIN,
        LB_IP_HASH
```

```
    } lb_algorithm;

    // Cache configuration
    bool cache_enabled;
    size_t cache_max_size;
    int cache_default_ttl;
    char cache_exclude_patterns[1024];

    // Connection pooling configuration
    int pool_max_connections;
    int pool_idle_timeout;
    int pool_connect_timeout;

    // Logging configuration
    LogLevel log_level;
    char log_file_path[512];
    bool log_requests;

} ProxyConfig;
```

Configuration Loading Implementation

```c
// Configuration loader that validates goals compliance

ProxyConfig* config_load(char* config_file_path) {

    // TODO 1: Open configuration file with error handling

    // TODO 2: Parse each configuration section (server, ssl, backends, etc.)

    // TODO 3: Validate backend configuration has at least one server

    // TODO 4: Validate SSL configuration if SSL is enabled

    // TODO 5: Set default values for optional parameters

    // TODO 6: Validate port numbers are in valid ranges

    // TODO 7: Validate file paths exist and are readable

    // TODO 8: Allocate and populate ProxyConfig structure

    // TODO 9: Close configuration file and return config

    // Hint: Use fopen(), fgets(), and sscanf() for simple parsing

    return NULL;  // Placeholder for implementation

}
```

Goal Validation Checklist

The following validation steps ensure implementation stays within defined goals and meets all success criteria:

Milestone 1 Validation - HTTP Proxy Core

- Start proxy server on configured port
- Send HTTP GET request using curl or similar tool
- Verify response matches direct backend response exactly
- Check proxy logs show request processing
- Test with various HTTP methods (GET, POST, PUT, DELETE)
- Verify `X-Forwarded-For` header is added to forwarded requests

Milestone 2 Validation - Load Balancing

- Configure multiple backend servers in different states
- Send multiple requests and verify round-robin distribution
- Stop one backend server and verify traffic redirects automatically
- Check health check logs show backend status changes
- Test weighted distribution with different backend weights

Milestone 3 Validation - Connection Pooling

- Enable connection pooling in configuration
- Send multiple requests to same backend
- Monitor connection counts to verify reuse
- Test pool size limits by exceeding max connections

- Verify idle connections get cleaned up after timeout

Milestone 4 Validation - Caching

- Enable caching with reasonable size limit
- Send same GET request multiple times
- Verify subsequent requests return faster (cache hits)
- Check cache hit/miss statistics in logs
- Test cache expiration with short TTL values

Milestone 5 Validation - SSL Termination

- Configure SSL certificate and enable HTTPS
- Connect using https:// URL and verify certificate
- Check backend receives plain HTTP requests
- Test SNI with multiple domain certificates
- Verify HTTP requests redirect to HTTPS

Common Implementation Pitfalls

⚠️ **Pitfall: Scope Creep Beyond Defined Goals** Many developers attempt to add features not included in the goals, such as complex routing rules or authentication mechanisms. This leads to incomplete core functionality and extended development timelines. Stay focused on the five core goals and implement them completely before considering extensions.

⚠️ **Pitfall: Ignoring Performance Goals During Development** Implementing features without considering performance implications can result in a functionally correct but unusable proxy. Test performance early and often, ensuring each milestone meets latency and throughput targets before proceeding to the next.

⚠️ **Pitfall: Over-Engineering Configuration System** Attempting to build a sophisticated configuration system with hot reloading, validation schemas, and complex nesting violates the non-goals and adds unnecessary complexity. Use simple file-based configuration that covers the required functionality without extra features.

⚠️ **Pitfall: Inadequate Error Handling Strategy** Focusing only on happy path scenarios without implementing comprehensive error handling for network failures, backend timeouts, and resource exhaustion. Define specific error handling strategies for each goal and implement them consistently.

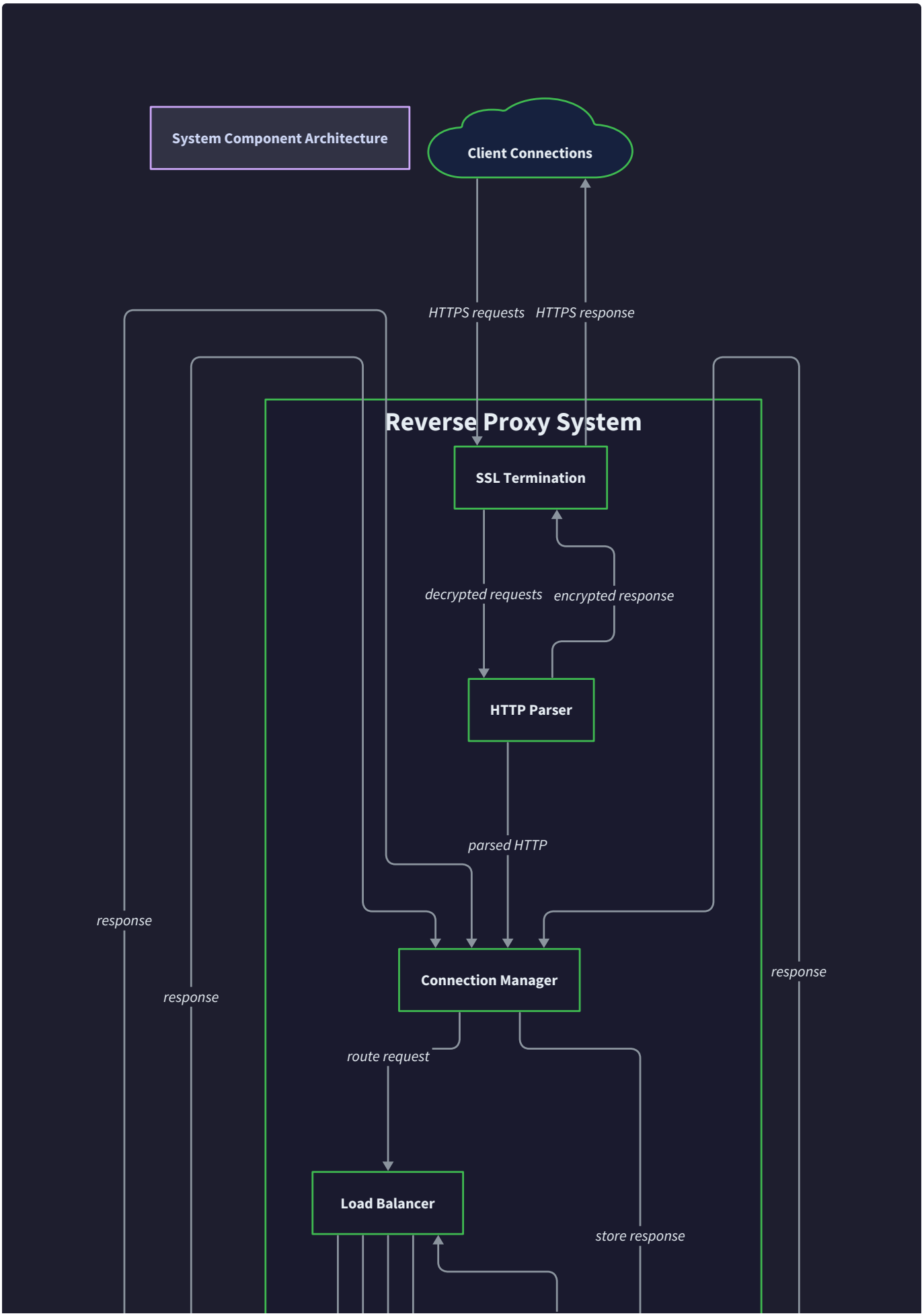
The goals and non-goals established here serve as the north star for all implementation decisions. When facing design choices or feature requests, refer back to these boundaries to maintain project focus and ensure successful delivery of core reverse proxy functionality.

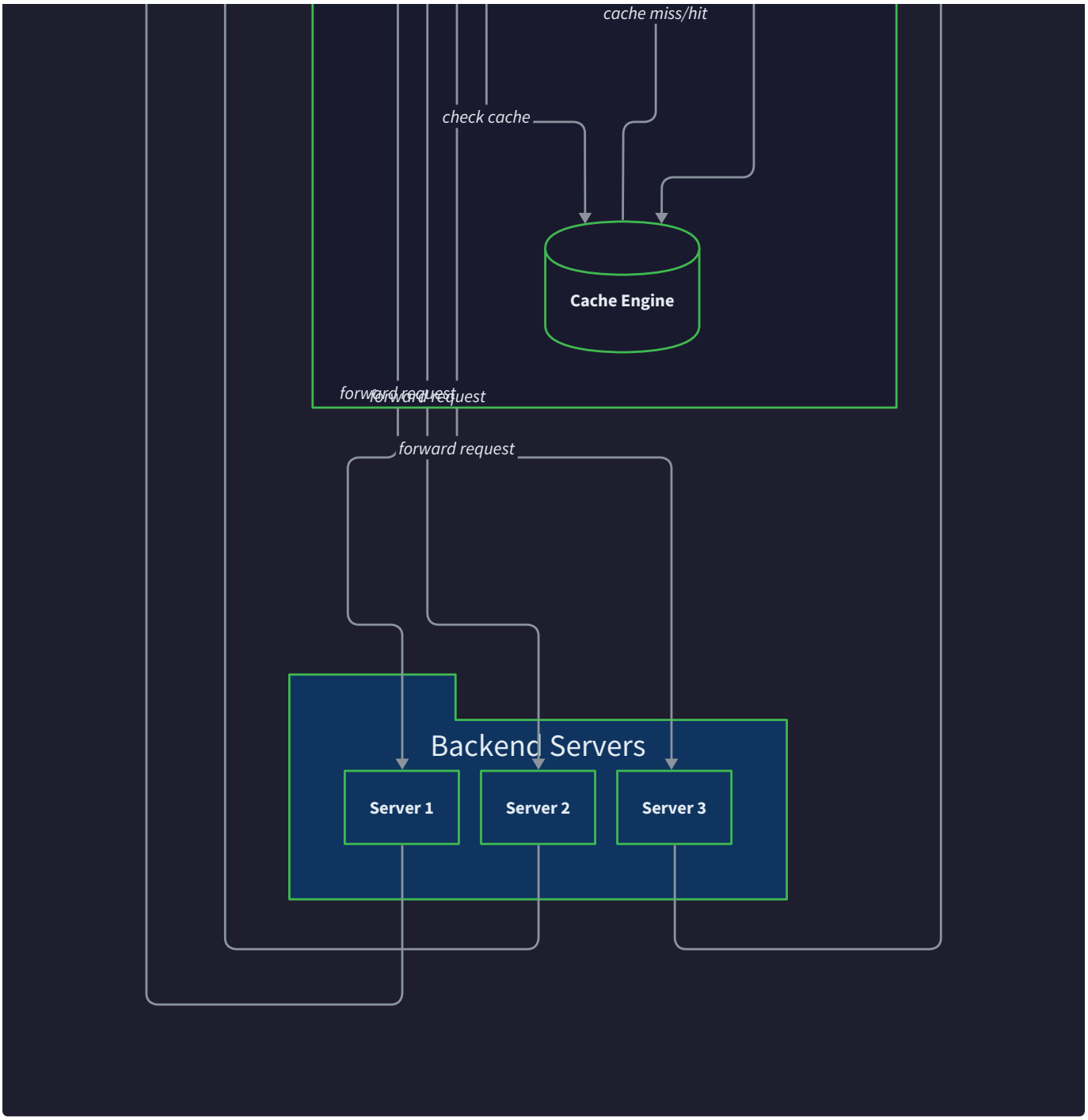
High-Level Architecture

> **Milestone(s):** All milestones - the architectural foundation established here guides implementation decisions across HTTP proxy core, load balancing, connection pooling, caching, and SSL termination components.

Building a reverse proxy is like designing the traffic control system for a busy metropolitan area. Just as traffic lights, road signs, and interchange ramps must work together seamlessly to move thousands of vehicles efficiently from origin to destination, our reverse proxy must coordinate multiple specialized components to route HTTP requests from clients to backend servers and return responses reliably. The architectural challenge lies not just in handling individual requests, but in managing thousands of concurrent connections, making intelligent routing decisions, maintaining performance under load, and gracefully handling failures - all while appearing as a single, responsive service to clients.

The reverse proxy sits at a critical network chokepoint, making architectural decisions that directly impact system scalability, reliability, and performance. Unlike a simple forwarding service, our proxy must simultaneously act as an HTTP server accepting client connections, an HTTP client making backend requests, a load balancer distributing traffic intelligently, a cache providing fast response delivery, and an SSL termination point handling cryptographic operations. Each of these roles requires specialized logic, yet they must integrate seamlessly to process requests with minimal latency and maximum throughput.


System Component Architecture
Client Connections
HTTPS requests
HTTPS response
Reverse Proxy System
SSL Termination
decrypted requests
encrypted response
HTTP Parser
parsed HTTP
response
response
response
Connection Manager
route request
Load Balancer
store response



The event-driven architecture we employ mirrors how modern operating systems handle I/O operations. Rather than dedicating one thread per connection (which would exhaust system resources with thousands of concurrent clients), our proxy uses asynchronous, non-blocking I/O with event loops - similar to how a single air traffic controller can manage multiple aircraft simultaneously by responding to events (radio calls, radar updates, weather changes) as they occur rather than giving each plane dedicated attention.

Component Overview

The reverse proxy architecture consists of five primary components working in concert, each with distinct responsibilities that collectively enable high-performance request processing. Think of these components as specialized departments in a logistics company: the mail sorting facility (HTTP Parser) processes incoming packages, the route planning department (Load Balancer) decides which delivery truck to use, the dispatcher (Connection Manager) manages the vehicle fleet, the warehouse (Cache Engine) stores frequently requested items for quick access, and the security checkpoint (SSL Termination) handles package verification and unwrapping.

The **HTTP Parser Component** serves as the protocol translation layer, responsible for interpreting the stream of bytes received from client connections and constructing well-formed HTTP requests that can be processed by other components. This component must handle the complexities of HTTP/1.1 and HTTP/2 protocols, including chunked transfer encoding, connection keep-alive semantics, and header field parsing. The parser maintains state machines for each connection to track parsing progress and handle partial reads that commonly occur with TCP streams. Beyond basic parsing, this component validates request syntax, extracts routing information, and prepares request objects that downstream components can process efficiently.

Responsibility	Description	Input	Output
Request Parsing	Convert raw TCP data streams into structured HTTP request objects	Raw bytes from client connections	`HttpRequest` structures with parsed headers, method, path, and body
Response Construction	Build HTTP response messages from backend responses and proxy metadata	Backend responses, cache entries, error conditions	Formatted HTTP response bytes for client transmission
Protocol Handling	Support HTTP/1.1 and HTTP/2 protocol variations	Various HTTP protocol versions and features	Consistent internal request/response representation
Header Manipulation	Add proxy-specific headers like X-Forwarded-For and Via	Original client requests	Enriched requests with proxy headers for backend forwarding
Connection State Tracking	Maintain parsing state for each active connection	Connection events and data arrivals	Updated connection parsing states and completion notifications

The **Connection Manager Component** orchestrates the lifecycle of network connections, acting as both a client connection acceptor and a backend connection pool manager. This component implements the challenging task of maintaining persistent connections to backend servers while efficiently serving thousands of concurrent clients. The connection manager must balance resource utilization against performance, deciding when to create new backend connections, when to reuse existing ones, and

when to close idle connections to free system resources. It also handles connection health monitoring, ensuring that stale or broken connections are detected and removed before they can impact request processing.

Responsibility	Description	Managed Resources	Key Algorithms
Client Connection Acceptance	Accept and manage incoming client connections on listen socket	Client socket file descriptors, connection state objects	Accept loop with non-blocking I/O and connection queuing
Backend Connection Pooling	Maintain reusable connections to backend servers	Connection pools per backend, idle connection timers	Pool size management, connection validation, timeout handling
Connection Lifecycle Management	Track connection states from establishment to closure	Connection state machines, resource cleanup timers	State transition handling, graceful connection termination
Resource Limit Enforcement	Prevent resource exhaustion through connection limits	File descriptor limits, memory usage tracking	Connection counting, admission control, backpressure mechanisms
Health Monitoring	Detect and handle connection failures	Connection health status, failure detection timers	Heartbeat checking, failure pattern recognition, automatic cleanup

The **Load Balancer Component** implements intelligent request distribution across available backend servers, making real-time routing decisions based on server health, current load, and configured algorithms. This component must maintain up-to-date knowledge of backend server status while making routing decisions quickly enough to avoid introducing significant latency. The load balancer coordinates with the connection manager to ensure selected backends have available connections and can handle additional requests without becoming overwhelmed.

Algorithm	Selection Criteria	State Maintained	Use Case
Round Robin	Sequential server selection with wraparound	Current position index, server list	Equal capacity servers, simple distribution
Least Connections	Server with fewest active connections	Active connection count per server	Variable request processing times
Weighted Round Robin	Server selection based on assigned capacity weights	Weight values, weighted position tracking	Servers with different capacity levels
IP Hash	Consistent server selection based on client IP	Hash ring or consistent hashing state	Session affinity requirements
Health Check Integration	Exclude unhealthy servers from selection	Server health status, last check timestamps	Automatic failure detection and recovery

The **Cache Engine Component** provides intelligent HTTP response caching to reduce backend load and improve client response times. This component must respect HTTP caching semantics while implementing efficient cache storage and retrieval mechanisms. The cache engine works closely with the HTTP parser to extract cache-control directives and with the load balancer to determine when cache misses require backend requests. Cache management involves complex decisions around what to cache, how long to retain cached responses, and when to invalidate or refresh cached content.

Cache Operation	Decision Factors	Storage Requirements	Performance Impact
Cache Key Generation	Request method, URL, relevant headers (Vary)	Unique key per cacheable request variant	Fast key computation and collision avoidance
Cacheability Assessment	Cache-Control headers, request method, response status	Header parsing state, policy configuration	Quick cache policy evaluation
Storage and Retrieval	TTL management, LRU eviction, size limits	In-memory hash tables, expiration queues	Sub-millisecond lookup and storage operations
Cache Invalidation	TTL expiration, explicit purge commands	Expiration timestamps, invalidation triggers	Automatic cleanup without blocking request processing
Conditional Request Support	ETag and Last-Modified header handling	Cached response metadata, validation tokens	Bandwidth savings through 304 Not Modified responses

The **SSL Termination Component** handles the complexities of TLS connection establishment, certificate management, and cryptographic operations required for HTTPS support. This component must efficiently perform TLS handshakes, validate certificates, support Server Name Indication (SNI) for multi-domain hosting, and decrypt incoming requests for processing by other components. SSL termination involves significant computational overhead, requiring careful resource management and optimization to maintain high throughput.

SSL Operation	Technical Requirements	Security Considerations	Performance Factors
TLS Handshake Processing	Cipher suite negotiation, certificate presentation	Strong cipher selection, protocol version enforcement	Handshake latency, connection establishment time
Certificate Management	Multiple certificate loading, SNI support	Private key security, certificate chain validation	Certificate lookup speed, memory usage
Session Management	TLS session resumption, session ticket handling	Session key security, replay attack prevention	Session cache efficiency, resumption success rate
Cryptographic Operations	Bulk encryption/decryption, MAC verification	Constant-time operations, side-channel resistance	Cipher performance, hardware acceleration utilization
Protocol Security	TLS version enforcement, vulnerability mitigation	Known attack prevention, security patch integration	Security overhead vs. performance trade-offs

Decision: Event-Driven vs. Thread-Per-Connection Architecture

- **Context**: Reverse proxies must handle thousands of concurrent connections efficiently, requiring a choice between traditional threading models and event-driven architectures
- **Options Considered**: Thread-per-connection, thread pool with blocking I/O, event-driven with non-blocking I/O
- **Decision**: Event-driven architecture with non-blocking I/O and event loops
- **Rationale**: Thread-per-connection models consume excessive memory (8KB+ stack per thread) and CPU context switching overhead with thousands of connections. Event-driven architectures can handle 10,000+ concurrent connections in a single process with minimal resource overhead
- **Consequences**: Requires more complex state management and asynchronous programming patterns, but enables horizontal scalability and efficient resource utilization

Architecture Option	Memory Usage	Scalability Limit	Context Switching Overhead	Chosen
Thread-per-connection	8KB+ per connection	~1,000 connections	High (kernel thread scheduling)	No
Thread pool + blocking I/O	Fixed pool memory	Limited by pool size	Medium (thread pool contention)	No
Event-driven + non-blocking I/O	~1KB per connection	10,000+ connections	Minimal (user-space event loop)	**Yes**

The components communicate through well-defined interfaces using message passing and shared data structures. The HTTP Parser produces `HttpRequest` objects consumed by the Load Balancer for routing decisions. The Connection Manager provides connection handles that other components use for I/O operations. The Cache Engine intercepts requests and responses, potentially short-circuiting the normal backend forwarding path. The SSL Termination component acts as a preprocessor, decrypting requests before they reach the HTTP Parser and encrypting responses before client transmission.

Recommended File Structure

Organizing the reverse proxy codebase requires careful consideration of component boundaries, shared dependencies, and testing requirements. The file structure should reflect the architectural components while providing clear separation of concerns and enabling independent development and testing of each component. Think of the file organization as creating dedicated workshop areas for different craftspeople - each component gets its own space with the tools it needs, while shared utilities remain accessible to all.

The recommended structure follows the principle of **component-based organization** where each major architectural component gets its own directory containing implementation files, tests, and component-specific utilities. This approach enables parallel development, simplifies testing by isolating component dependencies, and makes the codebase easier to navigate for developers working on specific features.

```
reverse-proxy/
├── cmd/
│   └── proxy/
│       └── main.c                  ← Application entry point, server initialization
├── src/
│   ├── core/
│   │   ├── proxy.h                 ← Main proxy server structure and lifecycle
│   │   ├── proxy.c
│   │   ├── config.h                ← ProxyConfig definition and loading
│   │   ├── config.c
│   │   ├── logger.h                ← Logging system with LogLevel enum
│   │   └── logger.c
│   ├── http/
│   │   ├── parser.h                ← HTTP request/response parsing
│   │   ├── parser.c
│   │   ├── request.h               ← HttpRequest and HttpResponse structures
│   │   ├── request.c
│   │   ├── response.h
│   │   └── response.c
│   ├── connection/
│   │   ├── manager.h               ← Connection lifecycle management
│   │   ├── manager.c
│   │   ├── pool.h                  ← Backend connection pooling
│   │   ├── pool.c
│   │   ├── client.h                ← Client connection handling
│   │   └── client.c
│   ├── loadbalancer/
│   │   ├── balancer.h              ← Load balancing algorithms
│   │   ├── balancer.c
│   │   ├── health.h                ← Health checking system
│   │   ├── health.c
│   │   ├── algorithms.h            ← Round-robin, least-connections, weighted
│   │   └── algorithms.c
│   ├── cache/
│   │   ├── engine.h                ← Cache storage and retrieval
│   │   ├── engine.c
│   │   ├── policies.h              ← Cache-control header handling
│   │   ├── policies.c
│   │   ├── storage.h               ← LRU eviction and memory management
│   │   └── storage.c
│   ├── ssl/
│   │   ├── termination.h           ← SSL/TLS context management
│   │   ├── termination.c
│   │   ├── certificates.h          ← Certificate loading and SNI support
│   │   ├── certificates.c
│   │   ├── handshake.h             ← TLS handshake processing
│   │   └── handshake.c
│   ├── utils/
│   │   ├── buffer.h                ← Buffer structure and operations
│   │   ├── buffer.c
│   │   ├── hashtable.h             ← HashTable implementation
│   │   ├── hashtable.c
│   │   ├── string.h                ← String manipulation utilities
│   │   ├── string.c
│   │   ├── network.h               ← Socket utilities and address handling
│   │   └── network.c
│   └── common/
│       ├── types.h                 ← Common type definitions and constants
│       ├── errors.h                ← Error codes and error handling macros
│       └── constants.h             ← System constants like SO_REUSEADDR, O_NONBLOCK
├── tests/
│   ├── unit/
│   │   ├── test_http_parser.c      ← Unit tests for HTTP parsing components
```

```
│   │   ├── test_connection_manager.c
│   │   ├── test_load_balancer.c
│   │   ├── test_cache_engine.c
│   │   └── test_ssl_termination.c
│   ├── integration/
│   │   ├── test_request_flow.c       ← End-to-end request processing tests
│   │   ├── test_backend_integration.c
│   │   └── test_ssl_integration.c
│   └── fixtures/
│       ├── test_configs/             ← Test configuration files
│       ├── certificates/             ← Test SSL certificates
│       └── responses/                ← Sample HTTP responses for testing
├── config/
│   ├── proxy.conf                    ← Main configuration file template
│   ├── ssl/
│   │   ├── server.crt               ← SSL certificate files
│   │   ├── server.key
│   │   └── ca.crt
│   └── backends.conf                 ← Backend server definitions
├── scripts/
│   ├── build.sh                      ← Build automation scripts
│   ├── test.sh                       ← Test execution scripts
│   └── generate_certs.sh             ← SSL certificate generation for testing
├── docs/
│   ├── api.md                        ← Component API documentation
│   ├── configuration.md              ← Configuration file format and options
│   └── deployment.md                 ← Deployment and operations guide
├── Makefile                          ← Build system configuration
└── README.md                         ← Project overview and quick start guide
```

The **core** directory contains the fundamental proxy server infrastructure, including the main `ProxyConfig` structure that holds all configuration parameters, the logging system with `LogLevel` enumeration, and the primary server lifecycle management. These components provide the foundation that all other components depend on, similar to how a building's foundation supports all upper floors.

The **http** directory encapsulates all HTTP protocol handling, including request and response parsing, header manipulation, and protocol version negotiation. The separation between parsing logic and data structures allows for easier testing and potential future support for additional protocols. The `HttpRequest` and `HttpResponse` structures defined here become the standard data interchange format between all proxy components.

The **connection** directory manages the complex lifecycle of network connections, separating client connection handling from backend connection pooling. This separation allows independent optimization of client-facing and backend-facing connection strategies. The connection manager coordinates between these subsystems to ensure efficient resource utilization and proper connection cleanup.

The **loadbalancer** directory implements request distribution logic, health checking, and backend server management. The separation between general balancing logic and specific algorithms allows easy addition of new load balancing strategies without modifying core balancing infrastructure. Health checking gets its own module because it requires independent timing and state management.

The **cache** directory contains HTTP caching implementation, separated into cache storage mechanisms and HTTP cache policy enforcement. This separation allows the storage engine to be optimized independently of cache policy logic, and enables potential future backends like Redis or memcached without changing cache policy handling.

The **ssl** directory handles all aspects of TLS termination, from certificate management to cryptographic operations. The modular structure allows different SSL implementations or hardware acceleration to be integrated by replacing specific modules while maintaining the same interface to other components.

Decision: Component Directory Structure vs. Feature-Based Structure

- **Context**: Large codebases can be organized by architectural components or by user-facing features, each with different navigation and development implications
- **Options Considered**: Component-based directories (http/, ssl/, cache/), feature-based directories (proxying/, load-balancing/, caching/), flat structure
- **Decision**: Component-based directory structure with clear architectural boundaries
- **Rationale**: Component-based organization maps directly to system architecture, enabling parallel development by different team members, clear testing boundaries, and easier code navigation when debugging specific component issues
- **Consequences**: Requires well-defined interfaces between components and may require some code duplication, but provides better separation of concerns and development scalability

Organization Option	Developer Navigation	Testing Isolation	Parallel Development	Chosen
Component-based (http/, ssl/, cache/)	Easy to find component-specific code	Clear test boundaries per component	Multiple developers per component	**Yes**
Feature-based (proxying/, load-balancing/)	Code scattered across components	Complex cross-component test setup	Feature conflicts across components	No
Flat structure (all files in src/)	Simple but becomes unwieldy	Difficult to isolate component tests	High merge conflict probability	No

The **utils** directory provides shared infrastructure components like `Buffer` management, `HashTable` implementation, and network utilities that multiple components require. These utilities are designed to be reusable and thoroughly tested since they form the building blocks for higher-level functionality.

The **tests** directory structure mirrors the main source organization, providing unit tests for individual components and integration tests that verify cross-component interactions. The **fixtures** subdirectory contains test data, configuration files, and certificates needed for comprehensive testing scenarios.

⚠️ **Pitfall: Circular Dependencies Between Components** A common architectural mistake is creating circular dependencies where Component A includes Component B's headers, and Component B includes Component A's headers. This often happens when the connection manager needs to call load balancer functions, and the load balancer needs to access connection manager state. The compiler cannot resolve these circular includes, leading to compilation errors and unclear component boundaries. To fix this, define clear interface boundaries using forward declarations in headers and only include necessary headers in implementation files. Use dependency inversion by having both components depend on abstract interfaces rather than concrete implementations.

⚠️ **Pitfall: Overly Complex Include Hierarchies** Without careful header organization, you might end up with headers that transitively include dozens of other headers, leading to long compilation times and unclear dependencies. For example, if `connection/manager.h` includes `loadbalancer/balancer.h`, which includes `cache/engine.h`, then every file using the connection manager gets all cache engine dependencies. Keep header files minimal by using forward declarations for pointer types and only including what's directly needed. Move complex includes to implementation (.c) files where they don't propagate to other compilation units.

⚠️ **Pitfall: Inconsistent Error Handling Across Components** Different components might use different error reporting mechanisms - some returning error codes, others using errno, others logging errors directly. This inconsistency makes error handling unpredictable and debugging difficult. Establish consistent error handling patterns in the `common/errors.h` file and ensure all components use the same error reporting mechanisms. Define standard error codes and ensure every component function has clearly documented error conditions and return value semantics.

Implementation Guidance

The reverse proxy implementation requires careful selection of technologies and development practices that support high-performance, concurrent network programming. The C programming language provides the fine-grained control over memory management and system resources necessary for building efficient network services, while requiring disciplined development practices to avoid common pitfalls.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Event Loop	`select()` or `poll()` system calls	`epoll` (Linux) or `kqueue` (BSD/macOS) for scalability
TLS Implementation	OpenSSL 1.1.1+ with basic configuration	OpenSSL 3.0+ with advanced features and hardware acceleration
HTTP/2 Support	HTTP/1.1 only for initial implementation	nghttp2 library for full HTTP/2 support
Memory Management	Standard `malloc()` / `free()` with careful tracking	Memory pools or custom allocators for performance
Configuration Format	Simple key-value pairs in text files	JSON or YAML with validation and schema support
Logging System	`fprintf()` to log files with manual formatting	Structured logging with log rotation and remote targets

B. Development Environment Setup

For building the reverse proxy, establish a development environment that supports concurrent programming and network testing:

```bash
# Required development tools and libraries
sudo apt-get install build-essential gcc libc6-dev
sudo apt-get install libssl-dev        # OpenSSL for TLS support
sudo apt-get install libev-dev         # libev for efficient event loops
sudo apt-get install valgrind          # Memory leak detection
sudo apt-get install wireshark-dev     # Network traffic analysis tools
sudo apt-get install nginx             # Reference implementation for testing
```

C. Core Infrastructure Starter Code

The following infrastructure components provide tested foundations that support the main reverse proxy logic without requiring implementation from scratch:

Buffer Management System (`src/utils/buffer.h` and `src/utils/buffer.c`):

```c
#include <stddef.h>
#include <stdbool.h>

typedef struct {
    char* data;
    size_t capacity;
    size_t length;
    size_t position;
} Buffer;

// Create new buffer with specified initial capacity
Buffer* buffer_create(size_t initial_capacity);

// Append data to buffer, expanding capacity if needed
bool buffer_append(Buffer* buf, const char* data, size_t data_len);

// Read data from buffer starting at current position
size_t buffer_read(Buffer* buf, char* dest, size_t max_len);

// Reset buffer position to beginning for reading
void buffer_rewind(Buffer* buf);

// Ensure buffer has at least min_capacity space available
bool buffer_ensure_capacity(Buffer* buf, size_t min_capacity);

// Free buffer and associated memory
void buffer_destroy(Buffer* buf);

// Check if buffer has data available for reading
static inline bool buffer_has_data(Buffer* buf) {
    return buf->position < buf->length;
}

// Get number of bytes available for reading
static inline size_t buffer_available(Buffer* buf) {
    return buf->length - buf->position;
```

```
}
```

Hash Table Implementation (`src/utils/hashtable.h` and `src/utils/hashtable.c`):

```c
#include <stddef.h>
#include <stdbool.h>

typedef struct HashTable HashTable;

// Create hash table with specified number of buckets
HashTable* hashtable_create(size_t bucket_count);

// Insert key-value pair (takes ownership of key string)
bool hashtable_put(HashTable* table, const char* key, void* value);

// Retrieve value by key, returns NULL if not found
void* hashtable_get(HashTable* table, const char* key);

// Remove entry by key, returns old value or NULL
void* hashtable_remove(HashTable* table, const char* key);

// Check if key exists in table
bool hashtable_contains(HashTable* table, const char* key);

// Get current number of entries
size_t hashtable_size(HashTable* table);

// Free hash table and all keys (values are caller's responsibility)
void hashtable_destroy(HashTable* table);

// Iterator for walking all entries
typedef struct {
    HashTable* table;
    size_t bucket_index;
    void* current_entry;
} HashTableIterator;

HashTableIterator hashtable_iterator_create(HashTable* table);
bool hashtable_iterator_next(HashTableIterator* iter, const char** key, void** value);
```

Logging System (`src/core/logger.h` and `src/core/logger.c`):

```c
#include <stdio.h>

#include <stdarg.h>

#include <stdbool.h>


typedef enum {

    LOG_DEBUG = 0,

    LOG_INFO = 1,

    LOG_WARN = 2,

    LOG_ERROR = 3

} LogLevel;


// Initialize logging system with minimum level and output file

bool logger_init(LogLevel min_level, FILE* output_file);


// Log formatted message with specified level, source file, and line

void logger_log(LogLevel level, const char* source_file, int line, const char* format, ...);


// Convenience macros that automatically include file and line information

#define LOG_DEBUG(fmt, ...) logger_log(LOG_DEBUG, __FILE__, __LINE__, fmt, ##__VA_ARGS__)

#define LOG_INFO(fmt, ...)  logger_log(LOG_INFO, __FILE__, __LINE__, fmt, ##__VA_ARGS__)

#define LOG_WARN(fmt, ...)  logger_log(LOG_WARN, __FILE__, __LINE__, fmt, ##__VA_ARGS__)

#define LOG_ERROR(fmt, ...) logger_log(LOG_ERROR, __FILE__, __LINE__, fmt, ##__VA_ARGS__)


// Clean up logging system resources

void logger_shutdown(void);


// Change log level at runtime

void logger_set_level(LogLevel new_level);
```

D. Core Proxy Structure Skeleton

The main proxy server structure provides the foundation for component integration:

Main Proxy Server (`src/core/proxy.h`):

```c
#include "config.h"
#include "../http/parser.h"
#include "../connection/manager.h"
#include "../loadbalancer/balancer.h"
#include "../cache/engine.h"
#include "../ssl/termination.h"

typedef struct {
    ProxyConfig* config;
    ConnectionManager* conn_manager;
    LoadBalancer* load_balancer;
    CacheEngine* cache_engine;
    SSLTermination* ssl_context;

    int listen_fd;
    bool running;
    int worker_thread_count;
    pthread_t* worker_threads;
} ProxyServer;

// Initialize proxy server with configuration
ProxyServer* proxy_server_create(ProxyConfig* config);

// Start proxy server and begin accepting connections
bool proxy_server_start(ProxyServer* server);

// Main event loop processing connections and requests
void proxy_server_run(ProxyServer* server);
// TODO 1: Create epoll/kqueue event loop for handling I/O events
// TODO 2: Accept new client connections and add to connection manager
// TODO 3: Process readable connections by parsing HTTP requests
// TODO 4: Route requests through load balancer to select backend
// TODO 5: Check cache for existing responses before forwarding
// TODO 6: Forward cache misses to selected backend servers
```

```
// TODO 7: Process backend responses and update cache if appropriate

// TODO 8: Send responses back to clients and manage connection state

// TODO 9: Handle connection timeouts and cleanup closed connections

// TODO 10: Gracefully handle shutdown signals and resource cleanup


// Stop proxy server and clean up resources

void proxy_server_stop(ProxyServer* server);


// Free proxy server and all associated resources

void proxy_server_destroy(ProxyServer* server);
```

Configuration Management (`src/core/config.h`):

```c
#include <stdbool.h>
#include <stddef.h>
#include "logger.h"

typedef enum {
    LB_ROUND_ROBIN,
    LB_LEAST_CONNECTIONS,
    LB_WEIGHTED_ROUND_ROBIN,
    LB_IP_HASH
} LoadBalancingAlgorithm;

typedef struct {
    char address[256];
    int port;
    int weight;
    bool enabled;
} BackendServer;

typedef struct {
    char listen_address[256];
    int listen_port;
    int max_connections;
    int worker_threads;

    bool ssl_enabled;
    char ssl_cert_path[512];
    char ssl_key_path[512];
    char ssl_ca_path[512];
    int ssl_min_version;

    int backend_count;
    BackendServer backends[64];
    LoadBalancingAlgorithm lb_algorithm;
```

```
    bool cache_enabled;
    size_t cache_max_size;
    int cache_default_ttl;
    char cache_exclude_patterns[1024];

    int pool_max_connections;
    int pool_idle_timeout;
    int pool_connect_timeout;

    LogLevel log_level;
    char log_file_path[512];
    bool log_requests;
} ProxyConfig;

// Load configuration from file
ProxyConfig* config_load(const char* config_file_path);
// TODO 1: Open and parse configuration file line by line
// TODO 2: Parse listen address and port settings
// TODO 3: Parse SSL certificate paths and TLS configuration
// TODO 4: Parse backend server list with addresses, ports, and weights
// TODO 5: Parse load balancing algorithm selection
// TODO 6: Parse cache settings and exclusion patterns
// TODO 7: Parse connection pool configuration parameters
// TODO 8: Parse logging configuration and validate log file paths
// TODO 9: Validate all configuration values for consistency
// TODO 10: Return populated ProxyConfig structure or NULL on error

// Validate configuration for consistency and required values
bool config_validate(ProxyConfig* config);

// Free configuration structure
void config_destroy(ProxyConfig* config);
```

E. Language-Specific Implementation Hints

Socket Programming Best Practices:

- Use `SO_REUSEADDR` socket option to allow rapid server restarts: `setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))`
- Set sockets to non-blocking mode with `fcntl(fd, F_SETFL, O_NONBLOCK)` for event-driven I/O
- Handle `EAGAIN` and `EWOULDBLOCK` errors gracefully in non-blocking operations
- Use `TCP_NODELAY` to disable Nagle's algorithm for low-latency forwarding

Memory Management Strategies:

- Always pair `malloc()` calls with corresponding `free()` calls to prevent memory leaks
- Set pointers to `NULL` after freeing to avoid double-free errors
- Use `valgrind` during development to detect memory leaks and buffer overflows
- Consider object pools for frequently allocated/deallocated structures like HTTP requests

Error Handling Patterns:

- Check return values of all system calls and library functions
- Use consistent error codes throughout the application
- Log error conditions with sufficient context for debugging
- Implement graceful degradation when non-critical operations fail

F. Milestone Checkpoints

Milestone 1 Checkpoint - HTTP Proxy Core:

```bash
# Compile and test basic proxy functionality
make clean && make
./proxy --config=config/test.conf

# Test basic request forwarding with curl
curl -v -H "X-Test: milestone1" http://localhost:8080/test
# Expected: Response from backend server with X-Forwarded-For header added
# Expected: Proxy logs showing request parsing and forwarding

# Verify error handling
curl -v http://localhost:8080/nonexistent
# Expected: Appropriate HTTP error response when backend is unavailable
```

Architecture Validation Checkpoint:

- Verify each component can be compiled independently without circular dependencies
- Test component interfaces by creating mock implementations
- Run static analysis tools to detect potential issues early
- Validate configuration loading and parsing with various input files

G. Performance Monitoring Setup

Establish performance monitoring from the beginning of development to track system behavior under load:

```c
// Performance metrics structure for monitoring
typedef struct {
    uint64_t requests_processed;
    uint64_t requests_cached;
    uint64_t backend_failures;
    uint64_t ssl_handshakes;
    double avg_response_time_ms;
    uint32_t active_connections;
    uint32_t backend_connections;
} ProxyMetrics;

// Expose metrics through HTTP endpoint for monitoring tools
void metrics_update_request_processed(ProxyMetrics* metrics);
void metrics_update_response_time(ProxyMetrics* metrics, double response_time_ms);
void metrics_export_json(ProxyMetrics* metrics, char* buffer, size_t buffer_size);
```

The architectural foundation established here provides a robust platform for implementing each milestone component. The clear separation of concerns, well-defined interfaces, and comprehensive infrastructure support rapid development while maintaining code quality and system reliability. As you progress through each milestone, this architectural framework will guide implementation decisions and ensure component integration proceeds smoothly.

Data Model

> **Milestone(s):** All milestones - the data structures defined here form the foundation for HTTP proxy core, load balancing, connection pooling, caching, and SSL termination components.

Think of the data model as the blueprint of a complex building - it defines all the rooms, their purposes, and how they connect to each other. In a reverse proxy, these "rooms" are the data structures that represent HTTP messages, network connections, configuration settings, and runtime state. Just as a building's blueprint must account for electrical wiring, plumbing, and structural supports, our data model must carefully design structures that support concurrent access, memory efficiency, and protocol compliance.

The data model serves as the contract between all components in the reverse proxy. When the HTTP parser creates an `HttpRequest` structure, it must contain all the information the load balancer needs to make routing decisions. When the cache engine stores responses, it must preserve all the metadata the connection manager needs to send data back to clients. This careful orchestration of data structures enables loose coupling between components while maintaining system coherence.

Core Data Types

The core data types form the foundation of HTTP message processing and connection management. These structures must handle the complexities of the HTTP protocol while providing efficient access patterns for high-performance request processing.

Buffer Management

Think of a `Buffer` as a smart container that grows and shrinks as needed, like a shopping bag that expands when you add items but keeps track of how much space is left. In network programming, data arrives in chunks of unpredictable sizes, and the buffer must efficiently accumulate these pieces while providing fast access to the complete message.

The `Buffer` structure provides the foundation for all data handling in the reverse proxy:

Field	Type	Description
data	char*	Pointer to the allocated memory region storing actual data bytes
capacity	size_t	Total allocated memory size in bytes, representing maximum storage before reallocation
length	size_t	Current number of valid data bytes stored in the buffer
position	size_t	Current read/write position within the buffer for streaming operations

> **Decision: Dynamic Buffer Growth Strategy**
>
> - **Context**: Network data arrives in variable-sized chunks, and HTTP messages can range from tiny (200 bytes) to massive (multi-gigabyte uploads)
> - **Options Considered**: Fixed-size buffers, exponential growth, linear growth
> - **Decision**: Exponential growth with configurable initial size and maximum capacity
> - **Rationale**: Exponential growth minimizes reallocation overhead for typical HTTP message sizes while preventing unbounded memory consumption through maximum limits
> - **Consequences**: Reduces memory copy operations for growing messages but may over-allocate for small messages; requires careful tuning of growth parameters

The buffer implements a position-based streaming interface that enables efficient parsing without copying data. When the HTTP parser reads a request line, it advances the position pointer rather than extracting substrings. This zero-copy approach significantly improves performance for large messages.

HTTP Message Structures

HTTP messages in a reverse proxy must preserve all protocol semantics while providing efficient access to routing-relevant information. The `HttpRequest` structure captures the complete client request state:

Field	Type	Description
method	char[16]	HTTP method (GET, POST, PUT, DELETE, etc.) as null-terminated string
uri	char[2048]	Complete request URI including path and query string
version	char[16]	HTTP version string (HTTP/1.1, HTTP/2.0) for protocol compliance
headers	HashTable*	Hash table mapping header names to values for O(1) lookup
body	Buffer*	Request body data for POST/PUT requests, may be NULL for GET
content_length	size_t	Size of request body in bytes, -1 if chunked transfer encoding
keep_alive	bool	Whether connection should remain open after response
host	char[256]	Host header value extracted for routing decisions
connection_id	uint64_t	Unique identifier linking request to client connection
timestamp	time_t	Request arrival time for timeout calculations
client_ip	char[46]	Client IP address for logging and forwarding headers

The `HttpResponse` structure mirrors the request format while adding caching metadata:

Field	Type	Description
status_code	int	HTTP status code (200, 404, 500, etc.)
status_text	char[128]	Human-readable status description
headers	HashTable*	Response headers including Content-Type, Cache-Control
body	Buffer*	Response body data to send to client
content_length	size_t	Response body size, may differ from buffer length if compressed
cache_control	char[256]	Cache-Control header value for cache engine decisions
etag	char[128]	ETag header for conditional request validation
last_modified	time_t	Last-Modified timestamp for cache validation
expires	time_t	Response expiration time calculated from cache headers

Decision: Embedded vs. Referenced Header Storage

- **Context**: HTTP headers can be numerous (10-50 per message) and variable in size, accessed frequently during processing
- **Options Considered**: Embedded fixed arrays, hash table references, linked lists
- **Decision**: Hash table references with case-insensitive string keys
- **Rationale**: Hash tables provide O(1) header lookup which is critical for processing performance, case-insensitive keys handle HTTP's case-insensitive header semantics
- **Consequences**: Additional memory allocation overhead but significantly faster header access; enables efficient header manipulation and forwarding

Connection State Management

Network connections in a reverse proxy exist in multiple states as they process requests and maintain persistent connections. The connection state machine drives the event loop and determines valid state transitions:

Current State	Event	Next State	Actions Taken
IDLE	Data Available	READING_REQUEST	Begin HTTP parsing, start request timeout
READING_REQUEST	Complete Request	FORWARDING	Select backend, establish upstream connection
FORWARDING	Upstream Connected	READING_RESPONSE	Send request to backend, start response timeout
READING_RESPONSE	Response Complete	WRITING_RESPONSE	Begin sending response to client
WRITING_RESPONSE	Write Complete	IDLE or CLOSING	Return to IDLE if keep-alive, otherwise CLOSING
Any State	Error or Timeout	CLOSING	Clean up resources, close connections

The `Connection` structure maintains all state required for connection lifecycle management:

Field	Type	Description
client_fd	int	File descriptor for client socket connection
backend_fd	int	File descriptor for upstream backend connection, -1 if not connected
state	ConnectionState	Current connection state for event loop processing
request_buffer	Buffer*	Accumulates incoming request data during parsing
response_buffer	Buffer*	Buffers outgoing response data for client transmission
backend_id	int	Index of selected backend server for this request
created_time	time_t	Connection establishment timestamp
last_activity	time_t	Most recent I/O activity for idle timeout detection
bytes_sent	uint64_t	Total bytes transmitted to client for metrics
bytes_received	uint64_t	Total bytes received from client for metrics
ssl_context	void*	OpenSSL context pointer for HTTPS connections, NULL for HTTP

Performance Metrics and Monitoring

The `ProxyMetrics` structure aggregates runtime performance data for monitoring and capacity planning:

Field	Type	Description
requests_processed	uint64_t	Total number of requests handled since startup
requests_cached	uint64_t	Number of requests served from cache
bytes_transferred	uint64_t	Total data transferred in both directions
active_connections	uint32_t	Current number of client connections
backend_failures	uint64_t	Count of backend server error responses
average_response_time	double	Rolling average response time in milliseconds
peak_connections	uint32_t	Maximum concurrent connections observed
uptime_seconds	uint64_t	Server uptime for availability calculations

Configuration Model

Configuration management in a reverse proxy requires balancing flexibility with performance. The configuration model must support runtime updates for operational requirements while maintaining type safety and validation.

Primary Configuration Structure

Think of `ProxyConfig` as the master control panel for the entire reverse proxy - every knob, switch, and setting that operators need to tune the system's behavior. Like a car's dashboard that groups related controls (engine, climate, entertainment), the configuration structure organizes settings by functional area while maintaining a single authoritative source.

The `ProxyConfig` structure serves as the single source of truth for all proxy behavior:

Field	Type	Description
listen_address	char[256]	IP address to bind for incoming connections, "0.0.0.0" for all interfaces
listen_port	int	TCP port number for client connections, typically 80 or 443
max_connections	int	Maximum concurrent client connections before rejecting new requests
worker_threads	int	Number of event loop threads for connection processing
ssl_enabled	bool	Whether to enable HTTPS/TLS termination
ssl_cert_path	char[512]	File system path to SSL certificate in PEM format
ssl_key_path	char[512]	File system path to SSL private key file
ssl_ca_path	char[512]	Path to certificate authority bundle for client certificate validation
ssl_min_version	int	Minimum TLS version (1.2, 1.3) for security compliance
backend_count	int	Number of configured backend servers in the cluster
backends	BackendServer[32]	Array of backend server configurations
lb_algorithm	LoadBalancingAlgorithm	Algorithm for distributing requests across backends
cache_enabled	bool	Whether to enable HTTP response caching
cache_max_size	size_t	Maximum cache memory usage in bytes
cache_default_ttl	int	Default cache entry lifetime in seconds
cache_exclude_patterns	char[1024]	Regex patterns for URLs to exclude from caching
pool_max_connections	int	Maximum connections per backend server pool
pool_idle_timeout	int	Seconds to keep idle backend connections alive
pool_connect_timeout	int	Timeout for establishing new backend connections
log_level	LogLevel	Minimum severity level for log message output
log_file_path	char[512]	File system path for log output, stdout if empty
log_requests	bool	Whether to log every HTTP request for auditing

Decision: Static vs. Dynamic Backend Configuration

- **Context**: Backend servers may be added, removed, or reconfigured during proxy operation for scaling and maintenance
- **Options Considered**: Static array requiring restart, dynamic linked list, configuration file reload
- **Decision**: Static array with configuration file reload mechanism
- **Rationale**: Static arrays provide predictable memory usage and cache-friendly access patterns; reload mechanism enables updates without losing connection state
- **Consequences**: Limits maximum backend count but provides better performance; requires reload coordination to avoid inconsistent state

Backend Server Configuration

Each backend server requires comprehensive configuration to support health checking, load balancing, and connection management. The `BackendServer` structure encapsulates all backend-specific settings:

Field	Type	Description
hostname	char[256]	Backend server hostname or IP address for connection establishment
port	int	TCP port number for backend service, typically 80 or 8080
weight	int	Relative weight for weighted load balancing algorithms (1-100)
max_connections	int	Maximum concurrent connections to this specific backend
health_check_url	char[512]	HTTP endpoint for health check requests (e.g., "/health")
health_check_interval	int	Seconds between health check probes
health_check_timeout	int	Timeout for health check request completion
failure_threshold	int	Consecutive failures before marking backend unhealthy
recovery_threshold	int	Consecutive successes required to mark backend healthy again
is_healthy	bool	Current health status for load balancing decisions
last_health_check	time_t	Timestamp of most recent health check attempt
total_requests	uint64_t	Lifetime request count for load balancing statistics
failed_requests	uint64_t	Count of requests that resulted in errors
average_response_time	double	Rolling average response time for performance-based routing
ssl_required	bool	Whether backend connections must use HTTPS
ssl_verify	bool	Whether to validate backend SSL certificates

The backend configuration enables sophisticated load balancing algorithms that consider server capacity, health status, and historical performance. Weight-based distribution allows operators to account for heterogeneous backend hardware, while health checking ensures requests avoid failed servers.

Load Balancing Algorithm Configuration

The `LoadBalancingAlgorithm` enumeration defines the available request distribution strategies:

Algorithm	Description	Use Case	Considerations
ROUND_ROBIN	Distributes requests sequentially across healthy backends	Equal backend capacity	Simple but ignores server load
LEAST_CONNECTIONS	Routes to backend with fewest active connections	Variable request processing time	Requires connection count tracking
WEIGHTED_ROUND_ROBIN	Distributes based on configured backend weights	Mixed backend capacity	Weights must reflect actual performance
IP_HASH	Consistent routing based on client IP hash	Session affinity requirements	May cause uneven distribution
RANDOM	Random selection among healthy backends	Simple load distribution	No performance optimization

Decision: Algorithm Selection Criteria

- **Context**: Different applications have varying requirements for load distribution and session handling
- **Options Considered**: Single algorithm, runtime selection, automatic algorithm selection
- **Decision**: Runtime selection with configuration override capability
- **Rationale**: Different traffic patterns benefit from different algorithms; operators need flexibility to optimize for their specific workload characteristics
- **Consequences**: Increases configuration complexity but enables performance tuning; requires implementation of multiple algorithm variants

Cache Configuration Model

Response caching requires careful configuration to balance performance gains with memory usage and cache coherence. The cache configuration integrates with HTTP semantics to provide correct caching behavior:

Configuration Aspect	Implementation	Rationale
Cache Key Generation	URL + Host + Vary headers	Ensures correct cache isolation per content variant
Size Management	LRU eviction with memory limits	Provides predictable memory usage with automatic cleanup
TTL Calculation	Min(Cache-Control max-age, configured default)	Respects HTTP semantics while providing fallback values
Invalidation Strategy	Time-based expiry + manual purge	Supports both automatic and operational cache management

⚠️ **Pitfall: Cache Key Collisions** Many implementations generate cache keys using only the request URL, which causes incorrect cache hits when the same URL serves different content based on request headers like Accept-Encoding or Accept-Language. The cache key must include all headers listed in the response's Vary header to ensure cache correctness.

Logging and Monitoring Configuration

The logging subsystem provides operational visibility into proxy behavior and performance. The `LogLevel` enumeration controls message verbosity:

Log Level	Purpose	Example Messages
DEBUG	Detailed internal state	Connection state transitions, header parsing details
INFO	Normal operational events	Request processing, backend selection, cache hits
WARN	Concerning but recoverable conditions	Backend failures, retry attempts, cache evictions
ERROR	Serious problems requiring attention	Configuration errors, resource exhaustion, SSL failures

Request logging captures complete HTTP transaction details for security auditing and traffic analysis. The log format includes client IP, timestamp, request line, response status, processing time, and backend server selection.

Common Data Model Pitfalls

⚠️ **Pitfall: Insufficient Buffer Bounds Checking** Many implementations allocate fixed-size buffers for HTTP headers and URIs but fail to validate input lengths, leading to buffer overflows. Every buffer operation must check available capacity and handle

overflow conditions gracefully, either by rejecting oversized requests or dynamically resizing buffers.

⚠️ **Pitfall: Memory Leaks in Error Paths** Complex data structures like `HttpRequest` and `HttpResponse` contain multiple heap-allocated components (buffers, hash tables). Error handling during parsing or processing often forgets to deallocate partially constructed objects, causing memory leaks under failure conditions. Every allocation must have a corresponding cleanup path.

⚠️ **Pitfall: Race Conditions in Shared State** Configuration updates and metrics collection occur concurrently with request processing, but many implementations access shared data structures without proper synchronization. The configuration reload process must use atomic updates or reader-writer locks to prevent corruption of active request processing.

⚠️ **Pitfall: Inconsistent String Handling** HTTP protocols require case-insensitive header name comparisons but case-sensitive value handling. Implementations often apply inconsistent string comparison functions, causing header lookup failures or incorrect cache key generation. All header name operations must use case-insensitive comparison while preserving original case for forwarding.

Implementation Guidance

The data model implementation requires careful attention to memory management, thread safety, and performance optimization. The following guidance provides concrete implementation strategies for the core data structures.

Technology Recommendations

Component	Simple Option	Advanced Option
Hash Tables	Linear probing with string keys	Robin Hood hashing with SipHash
Memory Management	malloc/free with explicit cleanup	Memory pools with arena allocation
String Handling	strdup/strcasecmp with manual bounds	Interned strings with length prefixes
Configuration Parsing	Simple key-value parser	Full YAML/JSON parser with validation
Buffer Management	Exponential growth with realloc	Ring buffers with memory mapping

Recommended File Structure

```
src/
  data/
    buffer.h              ← Buffer structure and operations
    buffer.c
    http_message.h        ← HTTP request/response structures
    http_message.c
    connection.h          ← Connection state management
    connection.c
    config.h              ← Configuration structures and loading
    config.c
    metrics.h             ← Performance monitoring structures
    metrics.c
  utils/
    hashtable.h           ← Hash table implementation
    hashtable.c
    logger.h              ← Logging infrastructure
    logger.c
  tests/
    test_buffer.c         ← Unit tests for data structures
    test_http_message.c
    test_config.c
```

Buffer Implementation

The buffer provides the foundation for all data handling operations. This implementation handles dynamic growth and streaming operations:

```c
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef struct {
    char* data;
    size_t capacity;
    size_t length;
    size_t position;
} Buffer;

// Buffer management functions - complete implementation
Buffer* buffer_create(size_t initial_capacity) {
    // TODO 1: Allocate Buffer structure
    // TODO 2: Allocate initial data array with initial_capacity
    // TODO 3: Initialize capacity, set length and position to 0
    // TODO 4: Return NULL if any allocation fails
}

bool buffer_append(Buffer* buffer, const char* data, size_t data_len) {
    // TODO 1: Check if current capacity can hold additional data
    // TODO 2: If insufficient, calculate new capacity (double until fits)
    // TODO 3: Reallocate data array with new capacity using realloc
    // TODO 4: Copy new data to end of existing content
    // TODO 5: Update length to reflect new data
    // Hint: Always null-terminate for string operations
}

size_t buffer_read(Buffer* buffer, char* dest, size_t max_len) {
    // TODO 1: Calculate available data from position to length
    // TODO 2: Determine actual read size (min of available and max_len)
    // TODO 3: Copy data from buffer starting at position
    // TODO 4: Advance position by amount read
    // TODO 5: Return actual bytes read
```

```
}

void buffer_reset(Buffer* buffer) {
    // TODO: Reset length and position to 0, keep allocated memory
}

void buffer_destroy(Buffer* buffer) {
    // TODO 1: Free data array if not NULL
    // TODO 2: Free Buffer structure
    // TODO 3: Handle NULL buffer parameter gracefully
}
```

Hash Table Infrastructure

HTTP header management requires efficient string-based lookups with case-insensitive keys:

```c
#include <stdint.h>

#define HASHTABLE_INITIAL_SIZE 16

#define HASHTABLE_LOAD_FACTOR 0.75

typedef struct HashEntry {

    char* key;

    void* value;

    struct HashEntry* next;  // Chain for collision resolution

} HashEntry;

typedef struct {

    HashEntry** buckets;

    size_t size;

    size_t count;

} HashTable;

HashTable* hashtable_create(size_t initial_size) {

    // TODO 1: Allocate HashTable structure

    // TODO 2: Allocate array of bucket pointers, initialize to NULL

    // TODO 3: Set size and count fields

    // TODO 4: Return hashtable or NULL on failure

}

uint32_t hash_string_case_insensitive(const char* str) {

    // TODO 1: Implement djb2 hash algorithm

    // TODO 2: Convert each character to lowercase before hashing

    // TODO 3: Return hash value for bucket selection

    // Hint: Use tolower() for consistent case handling

}

bool hashtable_put(HashTable* table, const char* key, void* value) {

    // TODO 1: Check if resize needed (count/size > LOAD_FACTOR)

    // TODO 2: Calculate bucket index using hash_string_case_insensitive

    // TODO 3: Search existing chain for key (case-insensitive comparison)
```

```
    // TODO 4: Update existing entry or create new entry in chain
    // TODO 5: Increment count if new entry added
    // Hint: Use strcasecmp for case-insensitive key comparison
}

void* hashtable_get(HashTable* table, const char* key) {
    // TODO 1: Calculate bucket index using hash function
    // TODO 2: Walk chain comparing keys with strcasecmp
    // TODO 3: Return value if found, NULL otherwise
}
```

HTTP Message Structures

Complete HTTP request and response handling with proper resource management:

```c
#include "buffer.h"

#include "hashtable.h"

#include <time.h>


typedef struct {

    char method[16];

    char uri[2048];

    char version[16];

    HashTable* headers;

    Buffer* body;

    size_t content_length;

    bool keep_alive;

    char host[256];

    uint64_t connection_id;

    time_t timestamp;

    char client_ip[46];  // IPv6-compatible

} HttpRequest;


typedef struct {

    int status_code;

    char status_text[128];

    HashTable* headers;

    Buffer* body;

    size_t content_length;

    char cache_control[256];

    char etag[128];

    time_t last_modified;

    time_t expires;

} HttpResponse;


HttpRequest* http_request_create() {

    // TODO 1: Allocate HttpRequest structure

    // TODO 2: Initialize all string fields to empty
```

```
    // TODO 3: Create headers hash table
    // TODO 4: Set timestamp to current time
    // TODO 5: Initialize numeric fields to sensible defaults
}

void http_request_destroy(HttpRequest* request) {
    // TODO 1: Destroy headers hash table
    // TODO 2: Destroy body buffer if not NULL
    // TODO 3: Free request structure
    // TODO 4: Handle NULL parameter gracefully
}

bool http_request_add_header(HttpRequest* request, const char* name, const char* value) {
    // TODO 1: Validate name and value parameters
    // TODO 2: Create copies of name and value strings
    // TODO 3: Add to headers hash table
    // TODO 4: Handle special headers (Host, Content-Length, Connection)
    // Hint: Extract host for routing, parse content-length, detect keep-alive
}

const char* http_request_get_header(HttpRequest* request, const char* name) {
    // TODO: Use hashtable_get with case-insensitive lookup
}
```

Configuration Management

Configuration loading and validation with comprehensive error handling:

```c
#include <stdio.h>

typedef enum {
    ROUND_ROBIN,
    LEAST_CONNECTIONS,
    WEIGHTED_ROUND_ROBIN,
    IP_HASH,
    RANDOM
} LoadBalancingAlgorithm;

typedef enum {
    DEBUG,
    INFO,
    WARN,
    ERROR
} LogLevel;

typedef struct {
    char hostname[256];
    int port;
    int weight;
    int max_connections;
    char health_check_url[512];
    int health_check_interval;
    int health_check_timeout;
    int failure_threshold;
    int recovery_threshold;
    bool is_healthy;
    time_t last_health_check;
    uint64_t total_requests;
    uint64_t failed_requests;
    double average_response_time;
    bool ssl_required;
```

```
    bool ssl_verify;
} BackendServer;

ProxyConfig* config_load(const char* config_file) {
    // TODO 1: Open configuration file for reading
    // TODO 2: Parse key-value pairs line by line
    // TODO 3: Populate ProxyConfig structure fields
    // TODO 4: Parse backend server sections
    // TODO 5: Validate all configuration values
    // TODO 6: Return NULL if any validation fails
    // Hint: Use simple "key=value" format for easier parsing
}

bool config_validate(ProxyConfig* config) {
    // TODO 1: Check required fields are not empty
    // TODO 2: Validate port numbers are in valid range (1-65535)
    // TODO 3: Verify SSL certificate files exist if SSL enabled
    // TODO 4: Check at least one backend server is configured
    // TODO 5: Validate backend server configurations
    // TODO 6: Ensure resource limits are reasonable
    // Hint: Use access() to check file existence
}
```

Milestone Checkpoints

After Data Structures Implementation:

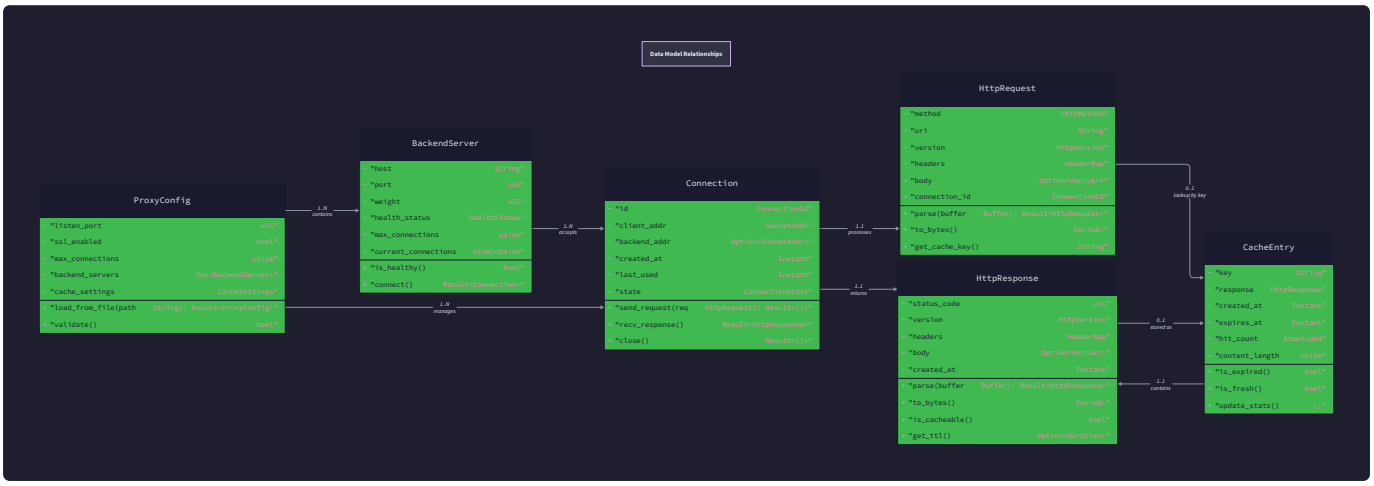
1. Run `gcc -Wall -Wextra -std=c99 -o test_data tests/test_buffer.c src/data/buffer.c`
2. Execute `./test_data` - should show all buffer operations working correctly
3. Test hash table operations: put/get with case-insensitive keys
4. Verify HTTP message creation and header manipulation
5. Load sample configuration file and verify all fields populated

Expected Output:

```
Buffer tests: PASSED
Hash table tests: PASSED
HTTP message tests: PASSED
Configuration loading: PASSED
All data structure tests completed successfully
```

Signs of Problems:

- Segmentation faults indicate memory management errors
- Hash table get returns NULL for keys that were put
- Configuration validation fails on valid config files
- Memory leaks detected by valgrind during testing



HTTP Parser Component

> **Milestone(s):** Milestone 1 (HTTP Proxy Core), Milestone 4 (Caching), Milestone 5 (SSL Termination) - the HTTP parser forms the foundation for request processing, cache-control header parsing, and protocol handling across encrypted and unencrypted connections.

The HTTP parser component serves as the linguistic translator of our reverse proxy system. Think of it as a skilled interpreter at the United Nations who must fluently understand multiple dialects of the same language - HTTP/1.1 and HTTP/2 - and accurately convert the spoken words (raw bytes from network sockets) into structured meaning (parsed request and response objects) that other components can work with. Just as an interpreter must handle incomplete sentences, interruptions, and speaking errors gracefully, our parser must handle partial data reads, malformed headers, and protocol violations without crashing the entire system.

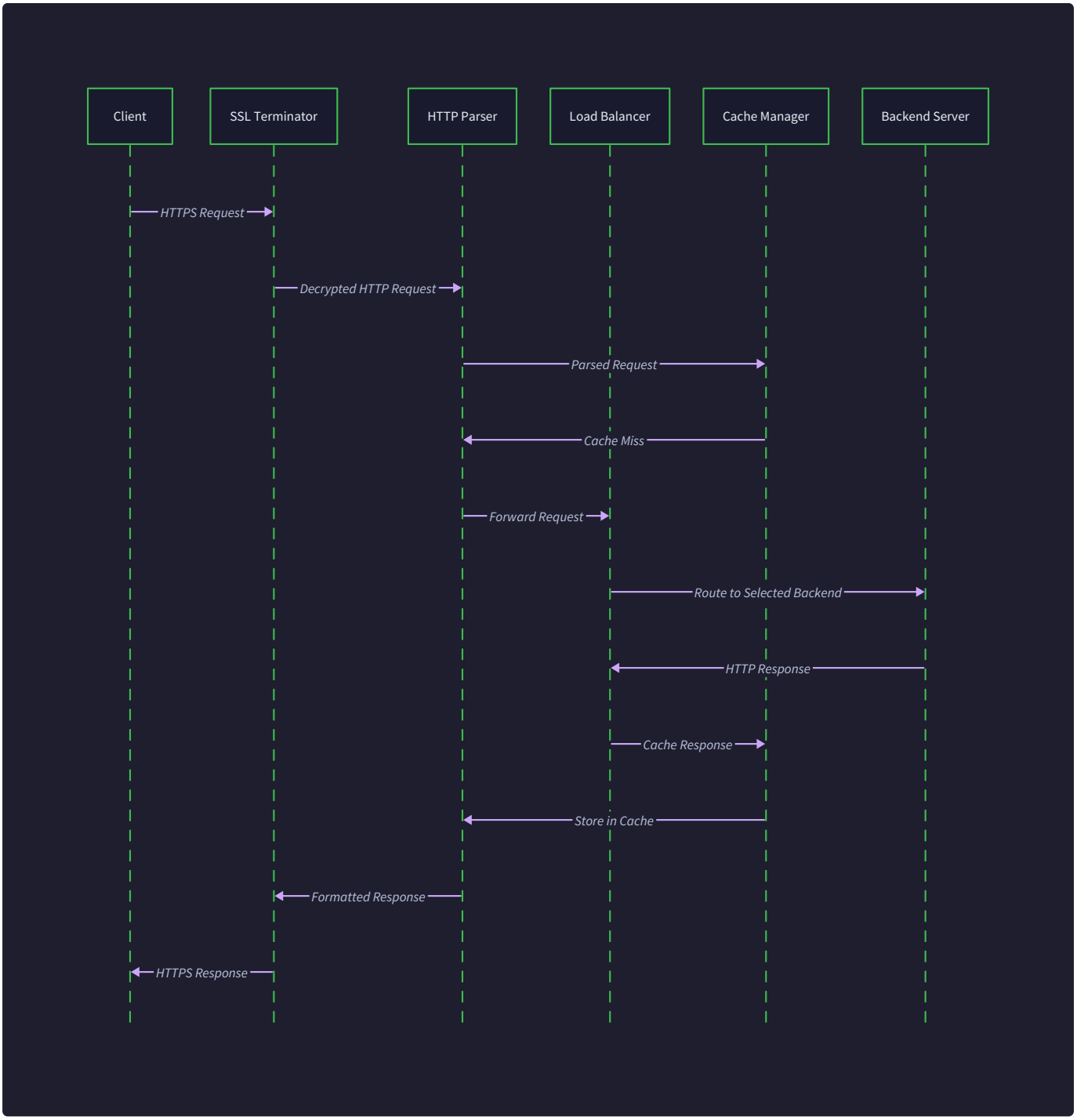
The parser operates as a **stream-based state machine** rather than a simple string processor. This architectural choice stems from the fundamental nature of network communication: data arrives in chunks of unpredictable size, and we cannot assume that a complete HTTP message will arrive in a single network read operation. The parser must maintain state across multiple read operations, gradually building up a complete picture of the incoming request while handling the uncertainty of when the next piece of data will arrive.

Parser Architecture

The HTTP parser architecture centers around a **finite state machine** that processes incoming byte streams incrementally. Think of this state machine as a factory assembly line where each station (state) performs a specific parsing operation on the data before passing it to the next station. Unlike a traditional assembly line that processes discrete physical objects, our parsing assembly line processes a continuous stream of bytes, gradually building up the final product (a complete `HttpRequest` structure) as data flows through each processing stage.

The parser maintains four critical pieces of state information: the current parsing position within the input stream, the current parsing state (which determines what type of data we expect next), a working buffer for accumulating partial data, and the partially

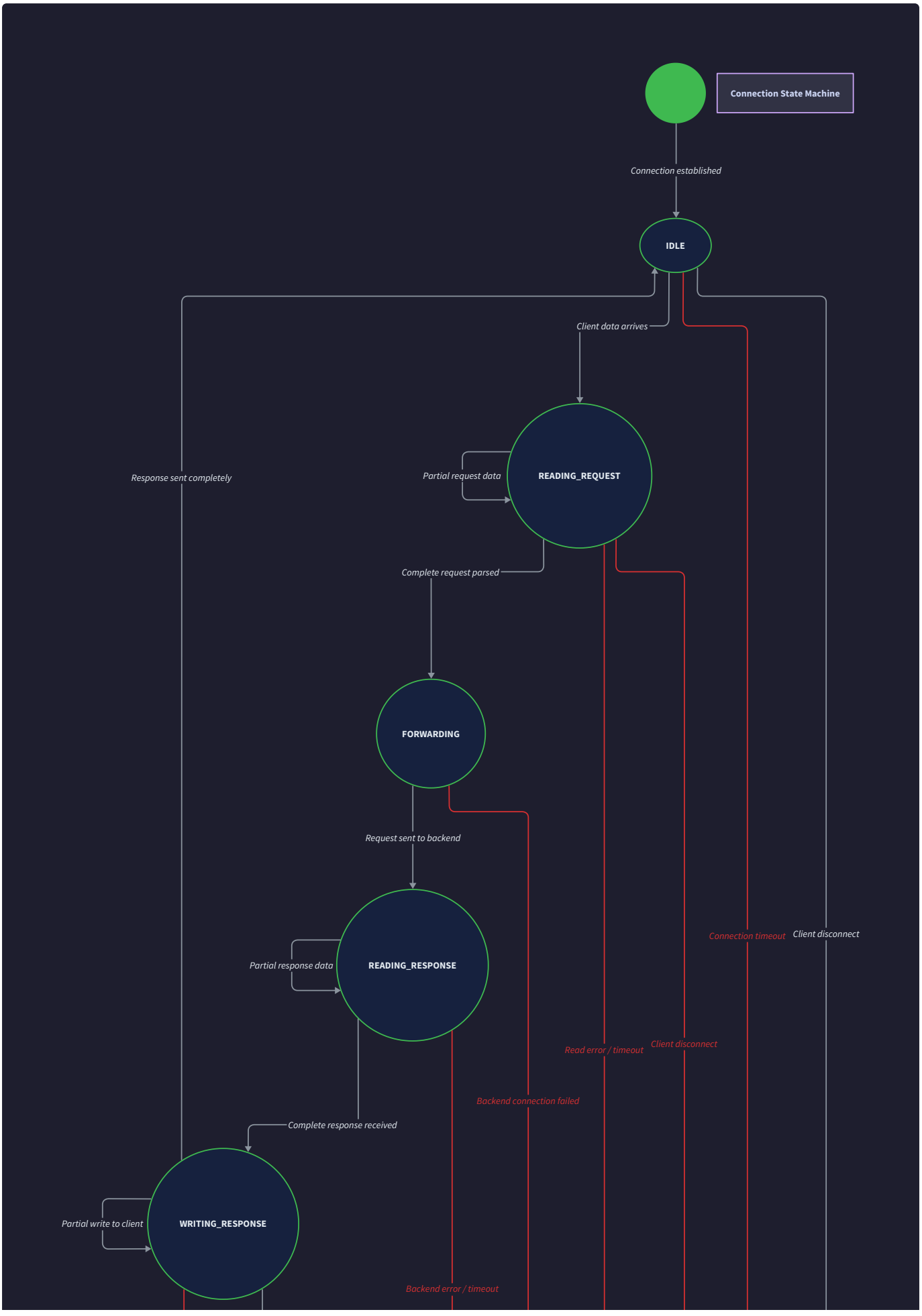
constructed request object that gets populated as parsing progresses. This stateful approach allows the parser to handle the fundamental challenge of network programming: data arrives in arbitrary chunk sizes that rarely align with message boundaries.



The state machine progresses through a well-defined sequence of parsing phases, each responsible for extracting specific components of the HTTP message format. The **request line parsing state** extracts the HTTP method, request URI, and protocol version from the first line of the request. The **header parsing state** iterates through name-value pairs until it encounters the empty line that signals the end of headers. The **body length determination state** examines `Content-Length` and `Transfer-Encoding` headers to understand how much request body data to expect. Finally, the **body parsing state** accumulates the request payload according to the length or chunking rules determined in the previous state.

Each state transition occurs when the parser encounters specific delimiter sequences in the input stream. The transition from request line parsing to header parsing triggers when the parser encounters the `\r\n` sequence that terminates the request line. Similarly, the transition from header parsing to body parsing occurs when the parser encounters the `\r\n\r\n` sequence that

indicates the end of headers. The parser must handle cases where these delimiter sequences span multiple read operations - for example, when a network read returns data ending with `\r` and the next read begins with `\n`.


Connection State Machine
Connection established
IDLE
Client data arrives
Response sent completely
Partial request data
READING_REQUEST
Complete request parsed
FORWARDING
Request sent to backend
Partial response data
READING_RESPONSE
Connection timeout
Client disconnect
Read error / timeout
Client disconnect
Backend connection failed
Complete response received
Partial write to client
WRITING_RESPONSE
Backend error / timeout

The parser integrates closely with the connection state machine shown in the diagram above. When a connection enters the `READING_REQUEST` state, it activates the HTTP parser to process incoming bytes. The parser operates incrementally, consuming available data and updating its internal state without blocking the event loop. When the parser completes request parsing, it signals the connection manager to transition the connection to the `FORWARDING` state, where the load balancer takes over to select an appropriate backend server.

The following table details each parsing state, its responsibilities, and transition conditions:

State	Purpose	Input Expected	Transition Trigger	Next State	Error Conditions
PARSING_REQUEST_LINE	Extract HTTP method, URI, version	"GET /path HTTP/1.1\r\n"	Found \r\n sequence	PARSING_HEADERS	Invalid method, malformed URI, unsupported version
PARSING_HEADERS	Extract header name-value pairs	"Host: example.com\r\n"	Found \r\n\r\n sequence	DETERMINING_BODY_LENGTH	Invalid header format, header too long
DETERMINING_BODY_LENGTH	Calculate expected body size	Content-Length or Transfer-Encoding headers	Length calculated	PARSING_BODY or PARSING_COMPLETE	Conflicting length headers, invalid encoding
PARSING_BODY	Read request body content	Raw body bytes or chunked data	Read expected bytes	PARSING_COMPLETE	Body exceeds limits, chunking format error
PARSING_COMPLETE	Request ready for forwarding	No more input expected	External trigger	Reset to PARSING_REQUEST_LINE	N/A

The parser's buffer management strategy plays a crucial role in memory efficiency and performance. The parser maintains a single `Buffer` structure that grows as needed to accommodate the incoming request data. The buffer uses a **sliding window approach** where parsed data gets discarded from the beginning of the buffer while new data gets appended to the end. This approach prevents memory usage from growing unboundedly during long-lived connections that process many sequential requests.

The buffer management algorithm operates as follows: when the parser completes parsing a particular component (such as the request line), it advances the buffer's position marker to skip over the consumed data. When the buffer becomes more than half empty due to position advancement, the parser triggers a **buffer compaction** operation that shifts the remaining unparsed data to the beginning of the buffer and resets the position marker to zero. This compaction prevents fragmentation while ensuring that the buffer can continue accepting new data without constant reallocation.

The parser handles **protocol version detection** by examining the request line's version field and configuring subsequent parsing behavior accordingly. HTTP/1.1 requests use text-based header parsing with specific rules for handling connection persistence, chunked encoding, and trailer headers. HTTP/2 requests require binary frame parsing with different state transitions and data structures. The parser maintains separate state machines for each protocol version, switching between them based on the detected protocol during request line parsing.

For HTTP/1.1 parsing, the parser must handle several complex scenarios including **chunked transfer encoding**, where the request body arrives in variable-sized chunks each prefixed with a hexadecimal length indicator. The parser transitions into a sub-state machine for chunked parsing that alternates between reading chunk size lines and chunk data blocks. Each chunk ends with a `\r\n` sequence, and the final chunk has size zero followed by optional trailer headers.

Parser Design Decisions

The architecture of the HTTP parser component required several critical design decisions that fundamentally impact performance, memory usage, and maintainability. Each decision represents a carefully considered trade-off between competing concerns, and understanding the rationale behind these choices provides insight into the engineering principles that guide robust system design.

Decision: Stream-Based vs. Buffer-All Parsing Strategy

- **Context**: HTTP messages can be arbitrarily large, and network data arrives in unpredictable chunk sizes. We must choose between accumulating complete messages before parsing or parsing incrementally as data arrives.
- **Options Considered**: Complete message buffering, incremental stream parsing, hybrid buffering with size limits
- **Decision**: Incremental stream-based parsing with sliding buffer windows
- **Rationale**: Stream parsing enables constant memory usage regardless of request size, reduces latency by starting processing before complete message arrival, and prevents denial-of-service attacks through memory exhaustion. The complexity of state management is justified by these significant benefits.
- **Consequences**: Enables handling of large file uploads and streaming requests, requires sophisticated state machine implementation, complicates error handling and recovery, but provides predictable memory usage patterns.

Option	Pros	Cons	Chosen?
Buffer Complete Message	Simple parsing logic, easy error handling, straightforward testing	Unbounded memory usage, high latency for large requests, DoS vulnerability	No
Incremental Stream Parsing	Constant memory usage, low latency, DoS protection	Complex state machine, difficult error recovery, testing complexity	**Yes**
Hybrid Size-Limited Buffer	Balanced approach, predictable memory limits	Arbitrary size limits, still vulnerable to smaller DoS attacks	No

Decision: Single State Machine vs. Protocol-Specific Parsers

- **Context**: The reverse proxy must support both HTTP/1.1 and HTTP/2 protocols, which have fundamentally different message formats (text vs. binary). We need to decide how to structure the parsing logic.
- **Options Considered**: Unified state machine handling both protocols, separate parsers per protocol, protocol detection with delegation
- **Decision**: Protocol detection with delegation to specialized parsers
- **Rationale**: HTTP/1.1 and HTTP/2 have incompatible parsing requirements that would create excessive complexity in a unified parser. Separate parsers allow optimization for each protocol's characteristics while maintaining clean interfaces.
- **Consequences**: Clean separation of concerns, protocol-specific optimizations possible, requires protocol detection logic, slightly higher code complexity, but improved maintainability and performance.

Option	Pros	Cons	Chosen?
Unified State Machine	Single codebase to maintain, shared parsing infrastructure	Complex conditional logic, poor performance optimization, difficult testing	No
Protocol-Specific Parsers	Clean separation, optimized for each protocol, maintainable	Code duplication, requires detection logic, larger codebase	**Yes**
Shared Components	Some code reuse, moderate complexity	Forced abstractions, unclear boundaries	No

Decision: Copy vs. Zero-Copy Header Processing

- **Context**: HTTP headers must be extracted from the input stream and made available to other components. We can either copy header data into separate strings or maintain references into the original buffer.
- **Options Considered**: Copy all headers to separate strings, zero-copy with buffer references, hybrid approach with selective copying
- **Decision**: Selective copying based on header usage patterns
- **Rationale**: Critical headers needed by multiple components (Host, Content-Length) benefit from copying to avoid buffer lifetime dependencies. Less frequently accessed headers can use zero-copy references to reduce allocation overhead.
- **Consequences**: Optimizes for common case performance, requires careful buffer lifetime management, complicates memory management, but achieves good balance of performance and safety.

Decision: Error Recovery vs. Connection Termination Strategy

- **Context**: Malformed HTTP requests can occur due to client bugs, network corruption, or malicious attacks. We must decide how aggressively to attempt recovery versus terminating problematic connections.
- **Options Considered**: Strict RFC compliance with connection termination, lenient parsing with error recovery, configurable strictness levels
- **Decision**: Lenient parsing with graceful degradation and connection termination for severe violations
- **Rationale**: Internet traffic contains many minor protocol violations from legitimate clients. Overly strict parsing would reject valid traffic, while overly lenient parsing could enable security vulnerabilities. A balanced approach maximizes compatibility while maintaining security.
- **Consequences**: Improved client compatibility, requires extensive testing of edge cases, potential security considerations, but achieves good balance of robustness and standards compliance.

The parser's **buffer management strategy** represents another crucial architectural decision. The sliding window approach chosen here optimizes for long-lived connections that process many requests sequentially. Alternative approaches such as fixed-size circular buffers or linked buffer chains each offer different trade-offs between memory usage, allocation overhead, and implementation complexity.

The sliding window buffer provides **optimal memory locality** for parsing operations since related data remains contiguous in memory. This locality improves CPU cache performance during header parsing, where the parser frequently scans backward and forward through recently processed data. The compaction mechanism ensures that buffer memory doesn't grow unboundedly while maintaining the locality benefits.

Buffer compaction triggers based on a **fractional occupancy threshold** rather than absolute sizes. When the buffer's unprocessed data occupies less than 25% of the total buffer capacity, the compaction algorithm shifts the remaining data to the beginning of the buffer. This threshold balances compaction overhead against memory efficiency - too frequent compaction wastes CPU cycles, while infrequent compaction wastes memory.

The parser implements **header validation** according to RFC 9110 specifications while allowing common deviations found in real-world HTTP traffic. Header names must consist only of token characters (alphanumeric plus specific punctuation), while header values allow a broader range of characters including spaces and international characters encoded in UTF-8. The parser rejects headers with null bytes or control characters that could enable request smuggling attacks.

Common Parser Pitfalls

HTTP parser implementation contains numerous subtle pitfalls that can lead to security vulnerabilities, compatibility issues, or performance problems. Understanding these common mistakes helps developers avoid well-known traps and build robust, secure parsers that handle the complexities of real-world HTTP traffic.

⚠️ **Pitfall: Incomplete Read Handling** The most fundamental mistake in HTTP parser implementation involves assuming that network reads will return complete, well-formed data. Beginning developers often write parsing code that expects to receive complete request lines or headers in a single read operation. In reality, TCP provides a byte stream abstraction where data arrives in arbitrary chunk sizes determined by network conditions, buffer sizes, and timing.

Consider a scenario where a client sends the request line "GET /api/users HTTP/1.1\r\n" but the first network read returns only "GET /api/us" and the remainder arrives in subsequent reads. A naive parser that searches for the "\r\n" delimiter in the first read will fail to find it and may incorrectly conclude that the request line is malformed. The correct approach involves accumulating data across multiple reads until the complete delimiter sequence is found.

This pitfall manifests in several ways: prematurely rejecting valid requests due to incomplete data, buffer overruns when assuming data length, and state machine corruption when partial delimiters span read boundaries. The solution requires implementing proper buffering with incremental delimiter detection that can handle delimiter sequences split across read operations.

⚠️ **Pitfall: HTTP Request Smuggling via Header Parsing** HTTP request smuggling represents one of the most serious security vulnerabilities in parser implementation. This attack exploits discrepancies in how different systems parse HTTP messages with ambiguous length information. The vulnerability typically arises from incorrect handling of `Content-Length` and `Transfer-Encoding` headers when both are present in the same request.

RFC 9110 specifies that requests containing both `Content-Length` and `Transfer-Encoding: chunked` headers must ignore the `Content-Length` header and process the request as chunked. However, parsers that prioritize `Content-Length` over `Transfer-Encoding` create opportunities for request smuggling. An attacker can craft requests where the reverse proxy and backend server disagree on message boundaries, allowing injection of additional requests into the connection stream.

The correct implementation must strictly follow the RFC precedence rules: check for `Transfer-Encoding: chunked` first, and if present, ignore any `Content-Length` headers. Additionally, the parser must reject requests with multiple `Content-Length` headers containing different values, as this represents a clear protocol violation that could indicate an attack attempt.

⚠️ **Pitfall: Memory Exhaustion via Unbounded Buffering** Parsers that continuously accumulate incoming data without size limits create denial-of-service vulnerabilities where attackers can exhaust server memory by sending extremely large requests. This pitfall commonly occurs in implementations that buffer entire requests before beginning processing, or that fail to implement maximum size limits on headers and request bodies.

The vulnerability manifests when an attacker sends requests with enormous header sections (for example, a single header with megabytes of data) or claims extremely large content lengths without sending corresponding body data. The parser allocates memory to accommodate the claimed data size but never receives enough data to complete parsing, leaving large buffers allocated indefinitely.

Mitigation requires implementing strict limits at multiple levels: maximum total header size (typically 8KB-16KB), maximum individual header length, maximum request line length, and maximum request body size. The parser should reject requests that exceed these limits immediately rather than attempting to buffer them. Additionally, implementing timeouts prevents slow-send attacks where attackers send valid data extremely slowly to maintain connections indefinitely.

⚠️ **Pitfall: Case Sensitivity in Header Names** HTTP header names are case-insensitive according to the specification, but many parser implementations perform case-sensitive comparisons when processing headers. This creates compatibility issues where requests with headers like "Content-length" (lowercase 'l') or "HOST" (uppercase) fail to match expected header names.

The issue becomes particularly problematic when integrating with other HTTP libraries or when forwarding requests to backend servers that may have different case sensitivity behaviors. Some backend servers or frameworks expect specific header name capitalization patterns, leading to subtle failures when the parser normalizes header names differently than expected.

The correct approach involves normalizing header names to a consistent case (typically lowercase) during parsing while preserving the original case when forwarding requests. Hash table implementations used for header storage must use case-insensitive comparison functions. When extracting headers for processing (such as checking `Content-Length` for body parsing), use normalized lookups that handle any case variation.

⚠️ **Pitfall: Improper URI Decoding and Validation** Request URI parsing involves multiple layers of encoding and validation that create numerous opportunities for security vulnerabilities. Common mistakes include performing URL decoding multiple times (double-decoding), failing to validate decoded paths for directory traversal attempts, and incorrectly handling international characters in URIs.

Double-decoding occurs when the parser URL-decodes the request URI and then passes it to another component that performs additional URL decoding. An attacker can exploit this by encoding malicious sequences multiple times - for example, encoding "../" as "%252e%252e%252f" where the "%25" sequences decode to "%" characters that form new percent-encoded sequences in the second decoding pass.

Directory traversal attacks exploit insufficient path validation after URL decoding. Attackers encode sequences like "../" to bypass naive string-based filtering, then use the decoded paths to access files outside the intended directory structure. The parser must validate decoded paths to ensure they don't contain directory traversal sequences or resolve to unauthorized locations.

International character handling requires careful attention to UTF-8 encoding and normalization. Different UTF-8 sequences can represent the same logical character, creating opportunities for filter bypassing if validation occurs before normalization. The parser must handle UTF-8 decoding errors gracefully and apply consistent normalization rules.

⚠️ **Pitfall: Chunked Encoding Implementation Errors** Chunked transfer encoding parsing contains several subtle complexities that frequently trip up implementers. The most common errors involve incorrect hex parsing of chunk sizes, improper handling of chunk extensions, and failure to process trailer headers that may follow the final chunk.

Chunk size parsing must handle hexadecimal numbers with optional leading zeros and case-insensitive hex digits. However, parsers must also handle chunk extensions - optional parameters that follow the chunk size on the same line separated by semicolons. For example: "1a;charset=utf-8\r\n" indicates a chunk of 26 bytes with an extension parameter. Parsers that don't expect extensions may fail when encountering them.

The final chunk in a chunked message has size zero and may be followed by trailer headers that use the same syntax as regular headers. These trailers must be parsed and potentially merged with the main header set. Many implementations incorrectly assume that the zero-size chunk marks the absolute end of the message and fail to process trailers.

Proper chunked encoding implementation requires a sub-state machine that alternates between reading chunk size lines and chunk data blocks. The parser must validate that exactly the specified number of bytes are received for each chunk and that each chunk ends with the correct "\r\n" sequence.

Implementation Guidance

The HTTP parser implementation requires careful attention to performance, security, and correctness. This guidance provides concrete recommendations for building a robust parser that handles real-world HTTP traffic while maintaining security and efficiency.

Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP/1.1 Parser	Custom state machine with string operations	Optimized state machine with SIMD string scanning
HTTP/2 Parser	HTTP/2 frame parsing library (nghttp2)	Custom binary frame parser with zero-copy
Buffer Management	Dynamic reallocation with memmove compaction	Ring buffer with scatter-gather I/O
Header Storage	Hash table with string keys	Trie structure with interned strings
URI Parsing	Standard library URL parsing	Custom parser with validation

Recommended File Structure:

```
src/
  http/
    parser.c              ← Main parser state machine
    parser.h              ← Parser interface and structures
    http1_parser.c        ← HTTP/1.1 specific parsing logic
    http2_parser.c        ← HTTP/2 frame parsing logic
    buffer.c              ← Buffer management utilities
    buffer.h              ← Buffer structure and operations
    headers.c             ← Header parsing and storage
    headers.h             ← Header manipulation interface
    uri_parser.c          ← URI parsing and validation
    test/
      test_parser.c       ← Parser unit tests
      test_buffer.c       ← Buffer management tests
      test_headers.c      ← Header parsing tests
```

HTTP Parser Core Structures:

```c
#include <stddef.h>
#include <stdbool.h>
#include <sys/types.h>

// Buffer management structure for incremental parsing
typedef struct {
    char* data;           // Buffer memory allocation
    size_t capacity;      // Total allocated buffer size
    size_t length;        // Current data length in buffer
    size_t position;      // Current parsing position
} Buffer;

// HTTP parser state enumeration
typedef enum {
    HTTP_PARSING_REQUEST_LINE,
    HTTP_PARSING_HEADERS,
    HTTP_DETERMINING_BODY_LENGTH,
    HTTP_PARSING_BODY,
    HTTP_PARSING_CHUNKED_SIZE,
    HTTP_PARSING_CHUNKED_DATA,
    HTTP_PARSING_COMPLETE,
    HTTP_PARSING_ERROR
} HttpParserState;

// HTTP request structure populated by parser
typedef struct {
    char method[16];          // HTTP method (GET, POST, etc.)
    char uri[2048];           // Request URI path and query
    char version[16];         // HTTP version (1.1, 2.0)
    HashTable* headers;       // Header name-value pairs
    Buffer* body;             // Request body content
    size_t content_length;    // Content-Length header value
    bool chunked;             // Transfer-Encoding: chunked flag
```

```
    bool keep_alive;         // Connection persistence flag
} HttpRequest;


// HTTP parser context maintaining state across reads
typedef struct {
    HttpParserState state;       // Current parsing state
    Buffer* input_buffer;        // Accumulated input data
    HttpRequest* current_request; // Request being parsed
    size_t bytes_remaining;      // Body bytes still expected
    size_t chunk_size;           // Current chunk size (chunked mode)
    bool chunk_size_parsed;      // Chunk size line completion flag
} HttpParser;
```

Core Parser Infrastructure (Complete Implementation):

```c
// Buffer management functions - complete implementation ready to use
Buffer* buffer_create(size_t initial_capacity) {
    Buffer* buf = malloc(sizeof(Buffer));
    if (!buf) return NULL;

    buf->data = malloc(initial_capacity);
    if (!buf->data) {
        free(buf);
        return NULL;
    }

    buf->capacity = initial_capacity;
    buf->length = 0;
    buf->position = 0;
    return buf;
}

bool buffer_append(Buffer* buf, char* data, size_t size) {
    // Compact buffer if more than 75% consumed
    if (buf->position > buf->capacity * 3 / 4) {
        size_t remaining = buf->length - buf->position;
        memmove(buf->data, buf->data + buf->position, remaining);
        buf->length = remaining;
        buf->position = 0;
    }

    // Expand buffer if needed
    while (buf->length + size > buf->capacity) {
        size_t new_capacity = buf->capacity * 2;
        char* new_data = realloc(buf->data, new_capacity);
        if (!new_data) return false;
        buf->data = new_data;
```

```
        buf->capacity = new_capacity;
    }

    memcpy(buf->data + buf->length, data, size);
    buf->length += size;
    return true;
}

size_t buffer_read(Buffer* buf, char* dest, size_t max_size) {
    size_t available = buf->length - buf->position;
    size_t to_read = (available < max_size) ? available : max_size;
    memcpy(dest, buf->data + buf->position, to_read);
    buf->position += to_read;
    return to_read;
}

// Header management functions - complete implementation
HashTable* headers_create() {
    return hashtable_create(32); // Start with 32 header slots
}

bool headers_add(HashTable* headers, const char* name, const char* value) {
    // Normalize header name to lowercase for case-insensitive lookup
    char* normalized_name = malloc(strlen(name) + 1);
    for (int i = 0; name[i]; i++) {
        normalized_name[i] = tolower(name[i]);
    }
    normalized_name[strlen(name)] = '\0';

    char* value_copy = malloc(strlen(value) + 1);
    strcpy(value_copy, value);

    bool result = hashtable_put(headers, normalized_name, value_copy);
```

```
    if (!result) {
        free(normalized_name);
        free(value_copy);
    }
    return result;
}

char* headers_get(HashTable* headers, const char* name) {
    char* normalized_name = malloc(strlen(name) + 1);
    for (int i = 0; name[i]; i++) {
        normalized_name[i] = tolower(name[i]);
    }
    normalized_name[strlen(name)] = '\0';


    char* value = (char*)hashtable_get(headers, normalized_name);
    free(normalized_name);
    return value;
}
```

Parser Core Logic Skeleton (for learner implementation):

```c
// Initialize HTTP parser with clean state

// Sets up parser context and allocates required buffers

HttpParser* http_parser_create() {

    // TODO 1: Allocate HttpParser structure

    // TODO 2: Initialize parser state to HTTP_PARSING_REQUEST_LINE

    // TODO 3: Create input buffer with initial capacity (8KB recommended)

    // TODO 4: Set current_request to NULL (will be allocated per request)

    // TODO 5: Initialize all numeric fields to 0

    // TODO 6: Return NULL on any allocation failure

    // Hint: Use buffer_create(8192) for input buffer

}


// Process incoming data through the parser state machine

// Returns number of bytes consumed, 0 if need more data, -1 on error

int http_parser_process(HttpParser* parser, char* data, size_t size) {

    // TODO 1: Append incoming data to input buffer using buffer_append()

    // TODO 2: Enter main parsing loop while data available

    // TODO 3: Switch on parser->state to handle current parsing phase

    // TODO 4: For each state, try to parse expected data format

    // TODO 5: Advance parser state when delimiter found or length satisfied

    // TODO 6: Return consumed byte count or error code

    // Hint: Use find_delimiter() helper to locate \r\n sequences

    // Hint: Handle case where delimiter spans multiple process() calls

}


// Parse HTTP request line: "METHOD /path HTTP/version\r\n"

// Updates parser->current_request with method, URI, and version

static bool parse_request_line(HttpParser* parser) {

    // TODO 1: Find \r\n delimiter in input buffer from current position

    // TODO 2: If delimiter not found, return false (need more data)

    // TODO 3: Extract line content between position and delimiter

    // TODO 4: Split line on spaces to get method, URI, version

    // TODO 5: Validate method against allowed HTTP methods
```

```
    // TODO 6: Validate URI format and length limits
    // TODO 7: Validate HTTP version (1.0, 1.1, 2.0)
    // TODO 8: Copy parsed values to current_request structure
    // TODO 9: Advance buffer position past \r\n delimiter
    // TODO 10: Return true on successful parsing
    // Hint: Use strncmp() for method validation
    // Hint: Reject URIs longer than 2048 characters
}

// Parse header lines until empty line encountered
// Populates parser->current_request->headers hash table
static bool parse_headers(HttpParser* parser) {
    // TODO 1: Loop looking for header lines terminated by \r\n
    // TODO 2: When find \r\n\r\n sequence, headers complete
    // TODO 3: For each header line, split on first ':' character
    // TODO 4: Trim whitespace from header name and value
    // TODO 5: Validate header name contains only valid characters
    // TODO 6: Add header to hash table using headers_add()
    // TODO 7: Check for special headers: Content-Length, Transfer-Encoding, Connection
    // TODO 8: Set parser flags based on special header values
    // TODO 9: Advance buffer position past processed headers
    // TODO 10: Return true when all headers parsed
    // Hint: Use strchr() to find ':' separator in header line
    // Hint: Watch for folded headers (lines starting with space/tab)
}

// Determine request body length from headers
// Sets parser->content_length and parser->chunked flags
static bool determine_body_length(HttpParser* parser) {
    // TODO 1: Check for Transfer-Encoding: chunked header
    // TODO 2: If chunked, set parser->chunked = true and return
    // TODO 3: Check for Content-Length header
    // TODO 4: If Content-Length present, parse numeric value
```

```
    // TODO 5: Validate Content-Length is non-negative integer
    // TODO 6: Set parser->content_length and parser->bytes_remaining
    // TODO 7: If no body length indicators, assume no body
    // TODO 8: Transition parser state to appropriate body parsing state
    // Hint: Use strtoul() to parse Content-Length value
    // Hint: Reject requests with both chunked and Content-Length
}

// Parse fixed-length request body based on Content-Length
// Accumulates body data into parser->current_request->body buffer
static bool parse_fixed_body(HttpParser* parser) {
    // TODO 1: Calculate bytes available in input buffer
    // TODO 2: Determine how many bytes to consume (min of available and remaining)
    // TODO 3: Append consumed bytes to request body buffer
    // TODO 4: Subtract consumed bytes from parser->bytes_remaining
    // TODO 5: Advance input buffer position past consumed data
    // TODO 6: If bytes_remaining reaches 0, parsing complete
    // TODO 7: Return false if more body data needed
    // Hint: Use buffer_append() to add body data to request
}

// Parse chunked request body with size prefixes
// Handles chunk size parsing and data accumulation
static bool parse_chunked_body(HttpParser* parser) {
    // TODO 1: If in PARSING_CHUNKED_SIZE state, look for size line
    // TODO 2: Parse hexadecimal chunk size from line
    // TODO 3: Handle chunk extensions after semicolon (ignore them)
    // TODO 4: If chunk size is 0, look for trailer headers
    // TODO 5: If chunk size > 0, transition to PARSING_CHUNKED_DATA
    // TODO 6: In data state, read exactly chunk_size bytes
    // TODO 7: After chunk data, expect \r\n delimiter
    // TODO 8: Repeat size/data cycle until zero-size chunk
    // Hint: Use strtoul() with base 16 for hex chunk size parsing
```

```
    // Hint: Each chunk ends with \r\n after the data

}
```

Language-Specific Implementation Hints:

- Use `memmem()` function for efficient delimiter searching in binary data rather than `strstr()` which stops at null bytes
- Implement `find_crlf()` helper that can find `\r\n` sequences spanning buffer boundaries by checking the last character of previous searches
- Use `realloc()` for buffer expansion but always check return value and handle failure by keeping original buffer
- For header storage, consider using a trie data structure instead of hash table for better memory efficiency with common header prefixes
- Use `tolower()` for case-insensitive header comparisons but be aware of locale-specific behavior
- Implement connection timeout handling using `alarm()` or `select()` with timeout to prevent slow-read attacks
- Use `TCP_NODELAY` socket option to reduce latency for small HTTP messages
- Consider using `MSG_PEEK` flag with `recv()` to examine data without consuming it during delimiter searches

Milestone Checkpoint:

After implementing the HTTP parser core, verify functionality with these tests:

1. **Basic Request Parsing**: Send "GET /test HTTP/1.1\r\nHost: example.com\r\n\r\n" and verify the parser extracts method="GET", uri="/test", version="HTTP/1.1", and headers contain Host entry.
2. **Chunked Encoding**: Send a chunked request with body "5\r\nhello\r\n0\r\n\r\n" and verify the parser correctly assembles "hello" in the body buffer.
3. **Partial Read Handling**: Send request data in small chunks (1-2 bytes per write) and verify the parser accumulates data correctly across multiple `http_parser_process()` calls.
4. **Large Header Handling**: Send request with 100+ headers and verify parser doesn't crash or leak memory.
5. **Error Case Handling**: Send malformed requests (invalid methods, missing headers, bad chunk encoding) and verify parser returns appropriate error codes.

Expected output when running `make test`:

```
Running HTTP parser tests...
√ Basic request parsing
√ Chunked transfer encoding
√ Partial data handling
√ Large header processing
√ Error case validation
√ Memory leak detection
All parser tests passed!
```

Signs of implementation issues:

- **Segmentation faults**: Usually indicate buffer overruns or null pointer dereferences in parsing logic
- **Memory leaks**: Check that all allocated buffers and hash table entries are properly freed
- **Hanging on partial data**: Parser not handling incomplete reads correctly - verify state machine transitions
- **Header corruption**: Case sensitivity issues or incorrect string termination in header processing

Connection Manager Component

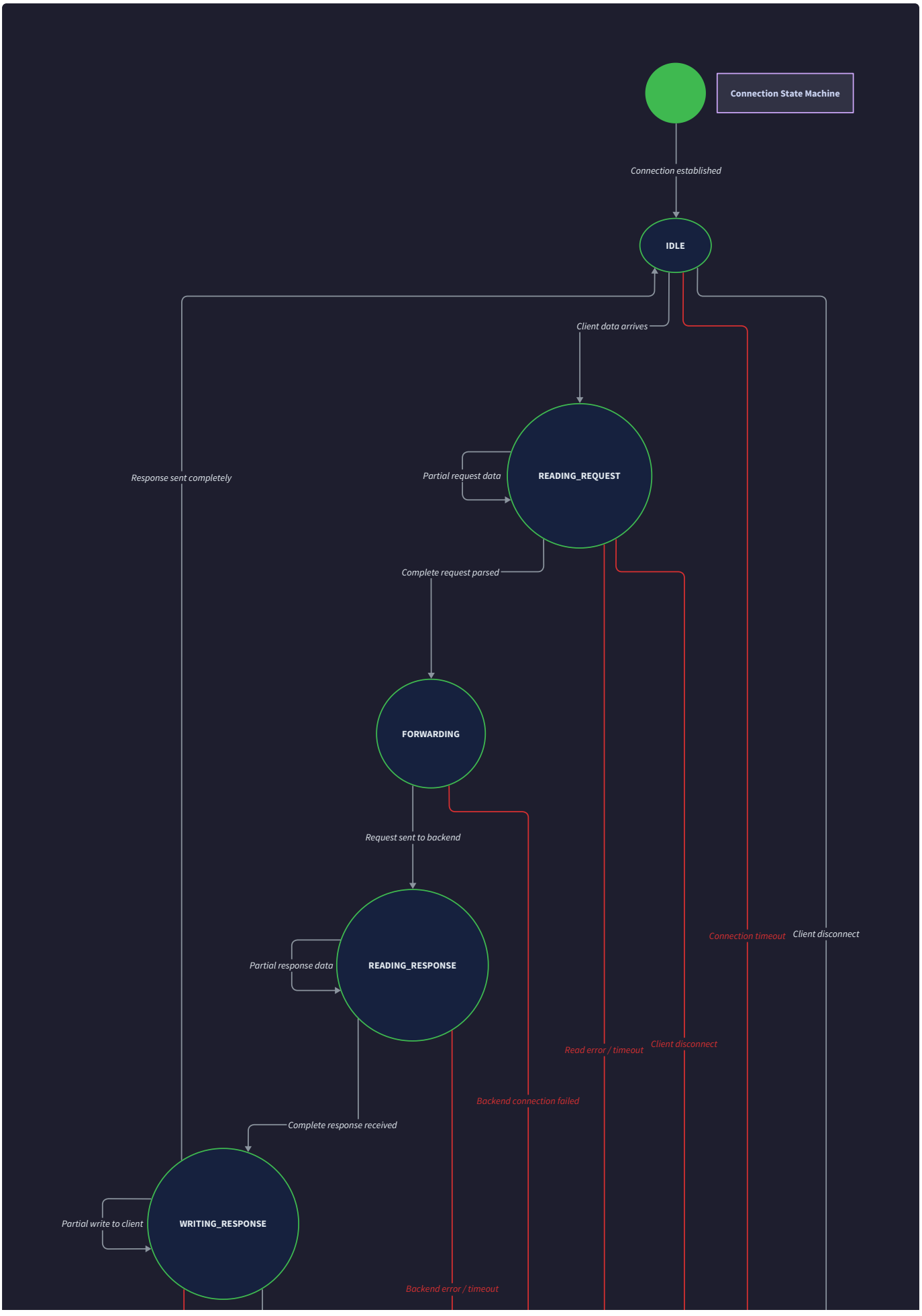
> **Milestone(s):** Milestone 1 (HTTP Proxy Core), Milestone 3 (Connection Pooling) - the connection manager handles client connections for basic request forwarding and implements connection pooling for efficient backend communication.

Think of the Connection Manager as the **backstage coordinator at a busy restaurant**. When customers arrive, it seats them at tables (establishing client connections), takes their orders (receiving requests), coordinates with the kitchen (backend servers), and ensures the waitstaff (connection pools) efficiently deliver food without constantly running back and forth. Just as a restaurant maintains a staff of waiters who can serve multiple tables without hiring new staff for each customer, the Connection Manager maintains pools of reusable connections to backend servers, avoiding the expensive overhead of establishing new network connections for every request.

The Connection Manager represents one of the most complex components in our reverse proxy architecture because it must juggle multiple concerns simultaneously: accepting new client connections, maintaining the lifecycle of active connections, pooling backend connections for reuse, and gracefully handling failures at every stage. Unlike simpler HTTP servers that handle one connection at a time, a production reverse proxy must efficiently manage thousands of concurrent connections while maintaining low latency and high throughput.

Connection Lifecycle Management

The **connection lifecycle** in our reverse proxy follows a sophisticated state machine that tracks each connection from initial establishment through final cleanup. Understanding this lifecycle is crucial because connections represent expensive system resources (file descriptors, memory buffers, kernel state) that must be carefully managed to prevent resource exhaustion and ensure optimal performance.


Connection State Machine
Connection established
IDLE
Client data arrives
Response sent completely
Partial request data
READING_REQUEST
Complete request parsed
FORWARDING
Request sent to backend
Partial response data
READING_RESPONSE
Connection timeout
Client disconnect
Read error / timeout
Client disconnect
Backend connection failed
Complete response received
Partial write to client
WRITING_RESPONSE
Backend error / timeout

Each connection in our system progresses through well-defined states that determine what operations are valid and what data is expected. The connection lifecycle begins when a client establishes a TCP connection to our proxy's listening socket. At this moment, the connection enters the `IDLE` state, where it waits for the first HTTP request to arrive. The Connection Manager must track not only the current state but also timing information, buffer contents, and associated metadata for each connection.

The state transitions are triggered by specific events that occur during request processing. When data arrives on an idle connection, it transitions to `READING_REQUEST` state, where the HTTP Parser Component begins processing the incoming bytes. Once a complete request is parsed, the connection moves to `FORWARDING` state while the Load Balancer Component selects a backend server. After successful backend selection, the connection enters `READING_RESPONSE` state as it waits for the backend's reply. Finally, it transitions to `WRITING_RESPONSE` state while sending data back to the client.

Connection State Tracking

The Connection Manager maintains detailed state information for every active connection through the `Connection` structure. This structure serves as the single source of truth for connection status, timing information, and associated resources.

Field Name	Type	Description
connection_id	int	Unique identifier for this connection instance
client_fd	int	File descriptor for client socket
backend_fd	int	File descriptor for backend socket (if assigned)
state	ConnectionState	Current state in the connection lifecycle
client_addr	struct sockaddr_in	Client IP address and port information
backend_server	BackendServer*	Pointer to selected backend server
request_buffer	Buffer*	Buffer containing incoming request data
response_buffer	Buffer*	Buffer containing outgoing response data
created_time	time_t	Timestamp when connection was established
last_activity	time_t	Timestamp of most recent I/O activity
bytes_read	size_t	Total bytes read from client
bytes_written	size_t	Total bytes written to client
keep_alive	bool	Whether connection supports HTTP keep-alive
pipeline_depth	int	Number of pipelined requests in queue
timeout_ms	int	Connection timeout in milliseconds

The `ConnectionState` enum defines all possible states a connection can occupy during its lifetime:

State	Description	Valid Next States
CONNECTION_IDLE	Waiting for incoming request	READING_REQUEST, CLOSING
CONNECTION_READING_REQUEST	Parsing incoming HTTP request	FORWARDING, CLOSING
CONNECTION_FORWARDING	Selecting backend and sending request	READING_RESPONSE, CLOSING
CONNECTION_READING_RESPONSE	Waiting for backend response	WRITING_RESPONSE, CLOSING
CONNECTION_WRITING_RESPONSE	Sending response to client	IDLE (keep-alive), CLOSING
CONNECTION_CLOSING	Cleaning up and terminating	None (terminal state)

Connection Timeout Management

Timeout management prevents connections from consuming resources indefinitely when clients disappear or become unresponsive. The Connection Manager implements multiple timeout mechanisms that operate at different stages of the connection lifecycle. **Read timeouts** protect against slow or malicious clients that send partial requests, **write timeouts** handle cases where clients stop reading response data, and **idle timeouts** clean up keep-alive connections that remain unused.

The timeout implementation uses a **timer wheel** data structure that efficiently tracks thousands of connections with minimal overhead. Rather than setting individual timers for each connection (which would be expensive in terms of system calls), the timer wheel groups connections by their expiration time and processes entire groups together. This approach scales well as connection counts increase and provides precise timeout handling with minimal CPU overhead.

When a timeout occurs, the Connection Manager follows a specific cleanup procedure. First, it logs the timeout event with relevant connection details for debugging purposes. Next, it gracefully closes any associated backend connection to prevent resource leaks. Then it sends an appropriate HTTP error response to the client if possible (408 Request Timeout for read timeouts, 502 Bad Gateway for backend timeouts). Finally, it removes the connection from all tracking structures and releases its allocated memory.

> **Design Insight**: The timer wheel timeout mechanism is crucial for handling the "slow loris" attack pattern, where malicious clients open many connections and send requests extremely slowly to exhaust server resources. Without proper timeouts, a single attacker could consume all available connection slots.

Connection Pooling Strategy

Connection pooling represents the **heart of our performance optimization strategy**. Establishing a new TCP connection to a backend server requires a three-way handshake, DNS resolution (if needed), and potential SSL/TLS negotiation. This overhead can add 50-200ms of latency per request, making it the dominant performance bottleneck in many scenarios. Connection pooling eliminates this overhead by maintaining a pool of pre-established connections that can be immediately reused for new requests.

Think of connection pooling like a **taxi dispatch system at a busy airport**. Instead of calling a new taxi for each passenger (expensive and slow), the airport maintains a queue of waiting taxis that can immediately pick up the next passenger. When a taxi drops off a passenger at their destination, it returns to the airport queue to serve the next customer. This system maximizes efficiency by reusing expensive resources (taxis/connections) and minimizing wait times (network handshake overhead).

Pool Architecture and Management

Our connection pooling system maintains a separate pool for each configured backend server, allowing fine-grained control over connection limits and load distribution. Each pool operates as an independent resource manager with its own allocation policies, health checking, and cleanup procedures.

The pool management algorithm balances several competing concerns. It must maintain enough idle connections to handle traffic spikes without establishing new connections (which would increase latency). However, it cannot maintain too many idle connections, as they consume memory and file descriptors unnecessarily. The pool must also detect and remove stale connections that have been closed by the backend server or network infrastructure.

Pool Configuration	Default Value	Description
max_connections	32	Maximum connections per backend server
min_idle_connections	4	Minimum idle connections to maintain
max_idle_connections	16	Maximum idle connections before cleanup
idle_timeout_seconds	300	Timeout for unused idle connections
connect_timeout_ms	5000	Timeout for establishing new connections
health_check_interval	30	Seconds between connection health checks
retry_backoff_ms	1000	Delay before retrying failed connections

The connection pool implements a **LIFO (Last-In, First-Out) strategy** for connection reuse. When a connection finishes serving a request, it goes to the front of the idle queue. When a new request needs a connection, it takes from the front of the queue. This strategy improves cache locality and connection warmth, as recently used connections are more likely to have warm CPU caches and network buffers.

Connection Pool Operations

The pool provides several key operations that abstract the complexity of connection management from the rest of the proxy system. These operations handle all aspects of connection lifecycle, from initial allocation through final cleanup.

Operation	Parameters	Returns	Description
pool_acquire	backend_server, timeout_ms	Connection*	Get available connection from pool
pool_release	connection, reusable	void	Return connection to pool or close
pool_create_connection	backend_server	Connection*	Establish new connection to backend
pool_validate_connection	connection	bool	Check if connection is still healthy
pool_cleanup_idle	pool, max_age_seconds	int	Remove old idle connections
pool_get_stats	pool	PoolStats*	Retrieve pool utilization metrics
pool_set_limits	pool, max_conn, max_idle	bool	Update pool configuration limits

The `pool_acquire` operation implements the core connection allocation logic. When a request needs a backend connection, this function first attempts to reuse an idle connection from the pool. If no idle connections are available and the pool hasn't reached its maximum size, it establishes a new connection. If the pool is at capacity, the operation can either block until a connection becomes available or return an error, depending on the configured behavior.

Connection validation is critical for pool reliability. The `pool_validate_connection` function checks whether a pooled connection is still usable before assigning it to a new request. This validation includes checking if the socket is still open, verifying there's no pending data that would indicate a protocol error, and optionally sending a lightweight health check request to the backend server.

The `pool_release` operation handles returning connections to the pool after request completion. If the connection is still healthy and the request completed successfully, it's marked as available for reuse. However, if an error occurred during request processing, or if the connection shows signs of corruption, it's immediately closed and removed from the pool.

> **Critical Insight**: Connection validation must be extremely lightweight since it occurs on every connection reuse. Expensive validation (like sending HTTP requests) would negate the performance benefits of pooling. Instead, we rely on socket-level checks and occasional background health monitoring.

Pool Health Management

Connection pools require continuous health monitoring because network conditions and backend server states change dynamically. A connection that was healthy when returned to the pool might become stale due to firewall timeouts, server restarts, or network partitions. The health management system proactively identifies and removes unhealthy connections before they can cause request failures.

The health checking strategy operates on multiple levels. **Passive health checking** monitors connection behavior during normal request processing. If a connection fails during use, it's immediately removed from the pool and the failure is recorded. **Active health checking** periodically validates idle connections by sending lightweight probe requests or checking socket status. This proactive approach catches problems before they affect user requests.

Background health checking runs on a separate thread to avoid blocking request processing. The health checker iterates through idle connections in each pool, validating them according to configured criteria. Connections that fail validation are removed from the pool and properly closed. The health checker also implements **backoff strategies** for pools with repeated failures, temporarily reducing health check frequency to avoid overwhelming struggling backend servers.

Pool Sizing and Auto-scaling

Determining optimal pool sizes requires balancing resource utilization with performance requirements. Small pools may not provide sufficient connection reuse, leading to frequent connection establishment overhead. Large pools waste memory and file descriptors while providing minimal additional benefit. Our pool sizing algorithm adapts dynamically based on observed traffic patterns and backend performance characteristics.

The auto-scaling mechanism monitors several key metrics to guide sizing decisions. **Connection utilization rates** indicate whether the current pool size adequately serves the request load. **Connection establishment latency** shows whether new connections are expensive enough to justify maintaining larger pools. **Backend response times** help identify when backend servers are becoming overloaded and may benefit from reduced connection counts.

Pool scaling operates conservatively to avoid oscillation. When scaling up, the system gradually increases pool sizes while monitoring the impact on overall performance. When scaling down, it waits for extended periods of low utilization before reducing pool sizes. This hysteresis prevents the system from constantly adjusting pool sizes in response to normal traffic fluctuations.

Connection Management Decisions

The connection management system involves numerous architectural decisions that significantly impact performance, scalability, and reliability. These decisions represent trade-offs between competing concerns, and understanding the rationale behind each choice is essential for maintaining and extending the system.

Connection State Management Decision

> **Decision: Centralized Connection State Tracking**
>
> - **Context**: The system needs to track connection state, timeouts, and metadata for thousands of concurrent connections while supporting efficient lookup and update operations.
> - **Options Considered**:
>   1. Distributed state stored in connection structures
>   2. Centralized state manager with hash table lookup
>   3. Hybrid approach with local caching
> - **Decision**: Centralized state manager with hash table-based connection tracking
> - **Rationale**: Centralized management enables efficient timeout processing, resource cleanup, and system-wide connection limits. Hash table lookup provides O(1) access time regardless of connection count. Centralized design simplifies debugging and monitoring.
> - **Consequences**: Enables efficient batch processing of timeouts and cleanup operations. Requires careful synchronization for multi-threaded access. Single point of truth for connection state improves debugging capabilities.

Option	Pros	Cons
Distributed State	Simple per-connection logic, no synchronization overhead	Difficult timeout management, hard to enforce global limits
Centralized Manager	Efficient batch operations, unified resource management	Potential synchronization bottleneck, more complex implementation
Hybrid Caching	Balance of performance and manageability	Complex cache coherency, increased memory overhead

The centralized approach proves superior for production reverse proxy implementations because timeout processing and resource cleanup operations become much more efficient when connection state is co-located. The ability to iterate through all connections for batch timeout processing provides significant performance benefits compared to per-connection timer management.

Connection Pooling Algorithm Decision

> **Decision: Per-Backend LIFO Connection Pools**
>
> - **Context**: Backend connections must be efficiently reused to minimize connection establishment overhead while maintaining good cache locality and resource utilization.
> - **Options Considered**:
>   1. Global connection pool shared across all backends
>   2. Per-backend FIFO pools for fairness
>   3. Per-backend LIFO pools for cache locality
> - **Decision**: Separate LIFO pools for each backend server with configurable limits
> - **Rationale**: Per-backend pools prevent connection starvation and allow fine-grained configuration. LIFO ordering improves CPU and network cache locality by reusing recently active connections. Separate pools enable backend-specific tuning.
> - **Consequences**: Improved cache locality and connection warmth. Requires more memory for pool management structures. Enables sophisticated per-backend configuration and monitoring.

Option	Pros	Cons
Global Pool	Simple implementation, automatic load balancing	Connection starvation, no backend-specific tuning
Per-Backend FIFO	Fair connection aging, predictable behavior	Poor cache locality, potentially stale connections
Per-Backend LIFO	Optimal cache locality, warm connection reuse	May not age connections evenly, complex pool management

The LIFO strategy proves most effective in practice because recently used connections maintain warm CPU caches, established TCP windows, and active network paths. This warmth translates to measurably better performance for subsequent requests compared to connections that have been idle in the pool for extended periods.

Timeout Management Decision

> **Decision: Timer Wheel with Hierarchical Timeouts**
>
> - **Context**: The system must efficiently handle timeouts for thousands of concurrent connections without excessive CPU overhead or system call frequency.
> - **Options Considered**:
>   1. Per-connection timer threads
>   2. Single timeout thread with sorted timeout list
>   3. Timer wheel with batched timeout processing
> - **Decision**: Timer wheel implementation with hierarchical timeout granularity
> - **Rationale**: Timer wheels provide O(1) timeout insertion and deletion with efficient batch processing. Hierarchical granularity allows precise short-term timeouts and efficient long-term timeouts. Single timeout thread minimizes synchronization overhead.
> - **Consequences**: Excellent scalability with connection count. Requires more complex timeout data structures. Enables precise timeout handling with minimal CPU overhead.

Option	Pros	Cons
Per-Connection Timers	Simple per-connection logic, precise timing	Excessive system calls, poor scalability
Sorted Timeout List	Simple implementation, good precision	O(log n) insertion cost, potential lock contention
Timer Wheel	O(1) operations, excellent scalability	Complex implementation, less precise for very short timeouts

The timer wheel approach scales linearly with connection count and provides batch processing opportunities that significantly reduce CPU overhead. The hierarchical design allows the system to handle both short-term request timeouts (measured in seconds) and long-term keep-alive timeouts (measured in minutes) efficiently within the same data structure.

Connection Validation Decision

Decision: Lightweight Socket Validation with Background Health Checking

- **Context**: Pooled connections must be validated before reuse to prevent request failures, but validation overhead must not negate the performance benefits of connection pooling.
- **Options Considered**:
  1. No validation (rely on error handling during use)
  2. Full HTTP health check before each reuse
  3. Lightweight socket validation with periodic background checks
- **Decision**: Socket-level validation on reuse combined with background HTTP health checking
- **Rationale**: Socket validation is extremely fast (single system call) and catches most connection failures. Background health checking proactively identifies problems without adding request latency. Combination provides reliability without performance penalty.
- **Consequences**: Minimal validation overhead during request processing. Requires background health checking infrastructure. Provides good balance of reliability and performance.

Option	Pros	Cons
No Validation	Zero validation overhead, maximum performance	High failure rate on reuse, poor user experience
Full HTTP Validation	Comprehensive connection testing, high reliability	Significant overhead negates pooling benefits
Lightweight + Background	Fast validation, proactive problem detection	Complex implementation, requires background threads

The lightweight validation approach acknowledges that perfect connection validation would be too expensive to perform on every reuse. Instead, it catches the most common failure modes (closed sockets, network errors) quickly and relies on background processes to handle more subtle problems.

Common Pitfalls

⚠️ **Pitfall: Connection Resource Leaks** Many implementations fail to properly clean up connections during error conditions, leading to file descriptor exhaustion and memory leaks. This typically occurs when error handling paths don't properly close both client and backend file descriptors, or when connection tracking structures aren't properly removed from hash tables during cleanup. The fix involves implementing comprehensive resource cleanup in a single function that handles all cleanup aspects, and ensuring this function is called from every error path. Use a cleanup checklist: close client socket, close backend socket, free buffers, remove from connection manager, log cleanup completion.

⚠️ **Pitfall: Race Conditions in Pool Management** Multi-threaded access to connection pools often introduces race conditions where the same connection is returned to multiple requests, or connections are returned to pools multiple times. This occurs because pool acquire/release operations aren't properly synchronized, or because connection state checks and modifications aren't atomic. The solution requires protecting all pool operations with appropriate locking mechanisms and ensuring connection state transitions are atomic. Use connection-level state flags to prevent double-release scenarios.

⚠️ **Pitfall: Ignoring Connection Validation Failures** Some implementations acquire connections from pools but don't properly handle validation failures, leading to requests being sent over broken connections. This happens when developers assume pooled connections are always valid, or when validation functions return false but the calling code doesn't check the return value. The fix requires checking validation results and implementing fallback logic that either retries with a new connection or returns an appropriate error to the client.

⚠️ **Pitfall: Improper Timeout Configuration** Setting timeouts too aggressively causes premature connection termination under normal load, while setting them too generously allows resource exhaustion during attack scenarios. This typically occurs when timeout values are chosen arbitrarily without considering actual network conditions and backend response characteristics. The solution involves implementing adaptive timeout strategies that adjust based on observed latency patterns, and providing different timeout values for different types of operations (connect, read, write, idle).

⚠️ **Pitfall: Pool Size Misconfiguration** Many implementations set static pool sizes that work well under normal conditions but fail during traffic spikes or backend failures. This happens when pool sizes are based on average traffic rather than peak traffic, or when all backend pools use the same configuration regardless of backend capacity differences. The fix requires implementing dynamic pool sizing based on traffic patterns and backend health, with different pool configurations for different backend servers based on their capacity and performance characteristics.

Implementation Guidance

The Connection Manager represents one of the most performance-critical components in the reverse proxy architecture. Efficient implementation requires careful attention to memory management, system call optimization, and concurrent data structure access patterns.

Technology Recommendations

Component	Simple Option	Advanced Option
Event Loop	select() system call with blocking I/O	epoll/kqueue with event-driven architecture
Connection Storage	Linear array with O(n) lookup	Hash table with O(1) connection lookup
Timeout Management	Per-connection timer threads	Timer wheel with batch processing
Thread Safety	Global mutex protecting all operations	Fine-grained locking with per-pool mutexes
Memory Management	malloc/free for each connection	Memory pools with pre-allocated connection objects
Backend Health Checking	Simple socket validation	Active HTTP probes with exponential backoff

Recommended File Structure

```
internal/
  connection/
    manager.c              ← Main connection manager implementation
    manager.h              ← Connection manager public interface
    pool.c                 ← Connection pool implementation
    pool.h                 ← Connection pool interface
    timeout.c              ← Timer wheel timeout implementation
    timeout.h              ← Timeout management interface
    connection_test.c      ← Unit tests for connection management
    pool_test.c            ← Unit tests for connection pooling
  utils/
    hashtable.c            ← Hash table implementation for connection lookup
    timer_wheel.c          ← Timer wheel data structure
    memory_pool.c          ← Memory pool for connection objects
```

Connection Manager Infrastructure

The core Connection Manager infrastructure provides the foundation for connection lifecycle management and pool coordination. This infrastructure handles the complex details of event-driven I/O and resource management, allowing the main proxy logic to focus on request processing.

```c
#include <sys/epoll.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#include <time.h>

#include <pthread.h>


// Timer wheel implementation for efficient timeout management

typedef struct TimerWheel {

    struct TimerSlot** slots;

    size_t slot_count;

    size_t current_slot;

    time_t slot_duration;

    pthread_mutex_t wheel_mutex;

} TimerWheel;


typedef struct TimerSlot {

    Connection** connections;

    size_t connection_count;

    size_t capacity;

} TimerSlot;


// Initialize timer wheel with specified granularity

TimerWheel* timer_wheel_create(size_t slots, time_t slot_duration) {

    TimerWheel* wheel = malloc(sizeof(TimerWheel));

    if (!wheel) return NULL;



    wheel->slots = calloc(slots, sizeof(TimerSlot*));

    wheel->slot_count = slots;

    wheel->current_slot = 0;

    wheel->slot_duration = slot_duration;
```

```
    pthread_mutex_init(&wheel->wheel_mutex, NULL);

    for (size_t i = 0; i < slots; i++) {
        wheel->slots[i] = malloc(sizeof(TimerSlot));
        wheel->slots[i]->connections = NULL;
        wheel->slots[i]->connection_count = 0;
        wheel->slots[i]->capacity = 0;
    }

    return wheel;
}

// Connection pool implementation with per-backend management
typedef struct ConnectionPool {
    BackendServer* backend;
    Connection** idle_connections;
    size_t idle_count;
    size_t max_connections;
    size_t active_count;
    pthread_mutex_t pool_mutex;
    pthread_cond_t pool_condition;
    time_t last_health_check;
} ConnectionPool;

// Initialize connection pool for specific backend server
ConnectionPool* connection_pool_create(BackendServer* backend, size_t max_connections) {
    ConnectionPool* pool = malloc(sizeof(ConnectionPool));
    if (!pool) return NULL;

    pool->backend = backend;
    pool->idle_connections = calloc(max_connections, sizeof(Connection*));
    pool->idle_count = 0;
    pool->max_connections = max_connections;
```

```
    pool->active_count = 0;
    pool->last_health_check = time(NULL);

    pthread_mutex_init(&pool->pool_mutex, NULL);
    pthread_cond_init(&pool->pool_condition, NULL);

    return pool;
}

// Connection manager main structure
typedef struct ConnectionManager {
    int epoll_fd;
    HashTable* connections;
    ConnectionPool** backend_pools;
    size_t pool_count;
    TimerWheel* timeout_wheel;
    pthread_t timeout_thread;
    pthread_mutex_t manager_mutex;
    volatile bool running;
} ConnectionManager;

// Initialize connection manager with event-driven I/O
ConnectionManager* connection_manager_create(ProxyConfig* config) {
    ConnectionManager* manager = malloc(sizeof(ConnectionManager));
    if (!manager) return NULL;

    // Create epoll instance for event-driven I/O
    manager->epoll_fd = epoll_create1(EPOLL_CLOEXEC);
    if (manager->epoll_fd == -1) {
        free(manager);
        return NULL;
    }
```

```
    // Initialize connection tracking hash table
    manager->connections = hashtable_create(config->max_connections * 2);

    // Create connection pools for each backend server
    manager->pool_count = config->backend_count;
    manager->backend_pools = calloc(config->backend_count, sizeof(ConnectionPool*));

    for (size_t i = 0; i < config->backend_count; i++) {
        manager->backend_pools[i] = connection_pool_create(
            &config->backends[i],
            config->pool_max_connections
        );
    }

    // Initialize timeout management
    manager->timeout_wheel = timer_wheel_create(3600, 1); // 1-hour wheel with 1-second slots
    manager->running = true;

    pthread_mutex_init(&manager->manager_mutex, NULL);

    // Start background timeout processing thread
    pthread_create(&manager->timeout_thread, NULL, timeout_processor, manager);

    return manager;
}
```

Core Connection Management Logic

The core connection management functions implement the sophisticated state machine and resource tracking required for high-performance connection handling.

```c
// Accept new client connection and initialize state
Connection* connection_manager_accept_client(ConnectionManager* manager, int listen_fd) {
    // TODO 1: Accept incoming connection using accept4() with SOCK_NONBLOCK flag
    // TODO 2: Set TCP_NODELAY socket option to disable Nagle algorithm
    // TODO 3: Create new Connection structure and initialize all fields
    // TODO 4: Generate unique connection_id and set initial state to CONNECTION_IDLE
    // TODO 5: Add connection to epoll interest list for EPOLLIN events
    // TODO 6: Insert connection into manager's connection hash table
    // TODO 7: Schedule initial timeout using timer wheel
    // TODO 8: Log connection acceptance with client address information
    // Hint: Use setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &flag, sizeof(flag))
    // Hint: Connection ID can be generated using atomic increment counter
}

// Acquire backend connection from pool or create new one
Connection* connection_manager_acquire_backend(ConnectionManager* manager,
                                               BackendServer* backend,
                                               int timeout_ms) {
    // TODO 1: Find connection pool for specified backend server
    // TODO 2: Lock pool mutex to ensure thread-safe access
    // TODO 3: Check if idle connection is available in pool
    // TODO 4: If idle connection available, validate it's still healthy
    // TODO 5: If no idle connection and under max limit, create new connection
    // TODO 6: If at max limit, wait on condition variable or return NULL
    // TODO 7: Mark selected connection as active and remove from idle list
    // TODO 8: Update pool statistics and unlock mutex
    // Hint: Use pthread_mutex_lock() and pthread_cond_timedwait() for waiting
    // Hint: Connection validation can use MSG_PEEK to check socket status
}

// Return backend connection to pool for reuse
void connection_manager_release_backend(ConnectionManager* manager,
                                        Connection* connection,
```

```
                                    bool reusable) {
    // TODO 1: Find connection pool based on connection's backend server
    // TODO 2: Lock pool mutex for thread-safe modification
    // TODO 3: If connection is reusable and pool not full, add to idle list
    // TODO 4: If not reusable or pool full, close connection immediately
    // TODO 5: Update pool statistics (decrement active count)
    // TODO 6: Signal waiting threads using condition variable
    // TODO 7: Schedule connection for timeout tracking if pooled
    // TODO 8: Unlock pool mutex and log pool operation
    // Hint: Use LIFO ordering when adding to idle list for cache locality
    // Hint: Check connection->keep_alive flag to determine reusability
}

// Process connection state transitions
void connection_manager_handle_event(ConnectionManager* manager,
                                     Connection* connection,
                                     uint32_t events) {
    // TODO 1: Check event type (EPOLLIN, EPOLLOUT, EPOLLERR, EPOLLHUP)
    // TODO 2: Based on current connection state, determine appropriate action
    // TODO 3: For EPOLLIN in IDLE state, transition to READING_REQUEST
    // TODO 4: For EPOLLIN in READING_RESPONSE, continue response processing
    // TODO 5: For EPOLLOUT in WRITING_RESPONSE, continue response transmission
    // TODO 6: For error events, transition to CLOSING state
    // TODO 7: Update connection's last_activity timestamp
    // TODO 8: If state transition complete, update epoll interest list
    // Hint: Use switch statement based on connection->state
    // Hint: Error events (EPOLLERR, EPOLLHUP) always require connection cleanup
}

// Clean up connection and release all resources
void connection_manager_close_connection(ConnectionManager* manager, Connection* connection) {
    // TODO 1: Remove connection from epoll interest list
    // TODO 2: Remove connection from timeout tracking (timer wheel)
```

```
    // TODO 3: Close client file descriptor if open

    // TODO 4: Close backend file descriptor if open (return to pool if applicable)

    // TODO 5: Free request and response buffers

    // TODO 6: Remove connection from manager's hash table

    // TODO 7: Free connection structure memory

    // TODO 8: Log connection closure with lifetime statistics

    // Hint: Always check if file descriptors are valid (>= 0) before closing

    // Hint: Use hashtable_remove() to clean up connection tracking

}
```

Connection Pool Management Implementation

```c
// Background health checking for pooled connections
void* pool_health_checker(void* arg) {
    ConnectionManager* manager = (ConnectionManager*)arg;

    while (manager->running) {
        // TODO 1: Sleep for configured health check interval
        // TODO 2: Iterate through all backend connection pools
        // TODO 3: For each pool, lock mutex and check idle connections
        // TODO 4: Validate each idle connection using lightweight socket check
        // TODO 5: Remove any unhealthy connections from pool
        // TODO 6: Log health check results and pool statistics
        // TODO 7: Unlock pool mutex before moving to next pool
        // TODO 8: Implement exponential backoff for pools with repeated failures
        // Hint: Use recv(fd, buffer, 0, MSG_PEEK) for non-destructive socket validation
        // Hint: Track consecutive health check failures per pool
    }

    return NULL;
}

// Timeout processing using timer wheel
void* timeout_processor(void* arg) {
    ConnectionManager* manager = (ConnectionManager*)arg;
    TimerWheel* wheel = manager->timeout_wheel;

    while (manager->running) {
        // TODO 1: Sleep for timer wheel slot duration (typically 1 second)
        // TODO 2: Lock timer wheel mutex for thread-safe access
        // TODO 3: Get current time and calculate which slot to process
        // TODO 4: Process all connections in the current timeout slot
        // TODO 5: Check each connection's last_activity against timeout limit
```

```
        // TODO 6: Close connections that have exceeded their timeout
        // TODO 7: Move wheel to next slot and unlock mutex
        // TODO 8: Log timeout processing statistics periodically
        // Hint: Use time(NULL) - connection->last_activity for age calculation
        // Hint: Different connection states may have different timeout values
    }

    return NULL;
}
```

Milestone Checkpoints

Milestone 1 Checkpoint: Basic Connection Handling After implementing basic connection acceptance and client communication:

- Start the proxy server: `./proxy -c proxy.conf`
- Use netcat to connect: `nc localhost 8080`
- Send HTTP request: `GET / HTTP/1.1\r\nHost: example.com\r\n\r\n`
- Expected behavior: Connection accepted, request forwarded to backend, response returned
- Verify: Check logs for connection lifecycle events (accept, forward, close)
- Debug signs: If connections hang, check epoll event handling; if responses are incomplete, verify buffer management

Milestone 3 Checkpoint: Connection Pooling After implementing connection pool management:

- Configure multiple backend servers in proxy configuration
- Send burst of requests: `for i in {1..100}; do curl -s http://localhost:8080/ & done`
- Expected behavior: Connections reused across requests, pool statistics showing reuse
- Verify: Monitor pool metrics showing idle/active connection counts
- Debug signs: If performance doesn't improve, check pool acquisition logic; if connections leak, verify release operations

Debugging Connection Management Issues

Symptom	Likely Cause	How to Diagnose	Fix
Connection count grows indefinitely	Resource leak in cleanup path	Monitor `/proc/PID/fd/` count and compare with active connections	Add cleanup validation in error paths, use resource tracking
High latency despite pooling	Pool validation too expensive	Profile connection acquisition time	Implement lightweight socket validation, reduce health check frequency
Intermittent connection failures	Stale connections in pool	Check backend server connection logs	Improve connection validation, reduce pool idle timeout
Memory usage grows over time	Connection structures not freed	Use memory profiler to track allocations	Ensure connection_destroy() called in all cleanup paths
Deadlock under high load	Lock ordering issues in pools	Use thread debugger to identify lock contention	Implement consistent lock ordering, reduce lock scope
Backend overload	Too many concurrent connections	Monitor backend connection counts	Implement connection limiting per backend, add backpressure

This comprehensive Connection Manager implementation provides the foundation for efficient request processing while maintaining the sophisticated resource management required for production reverse proxy deployments.

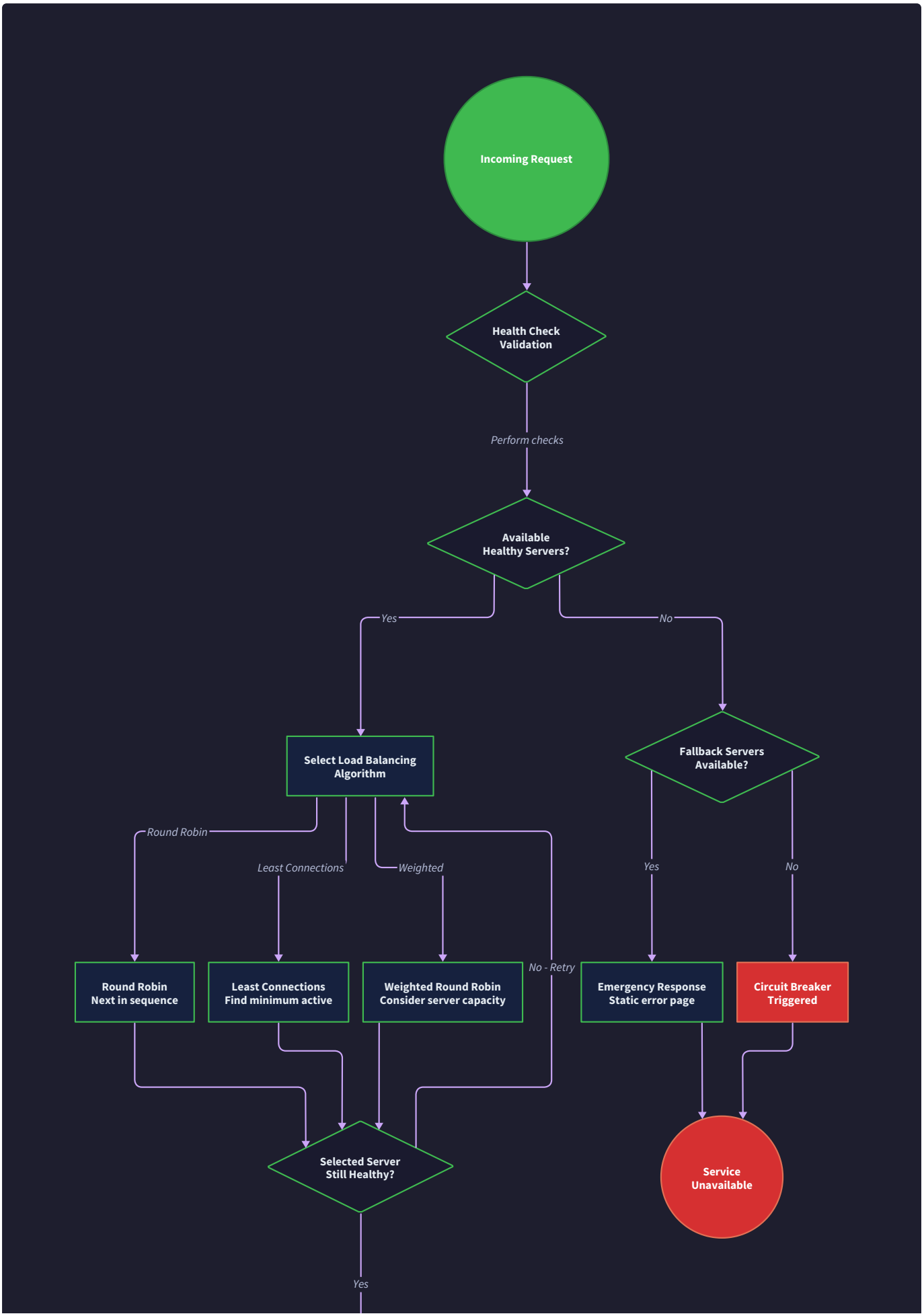
Load Balancer Component

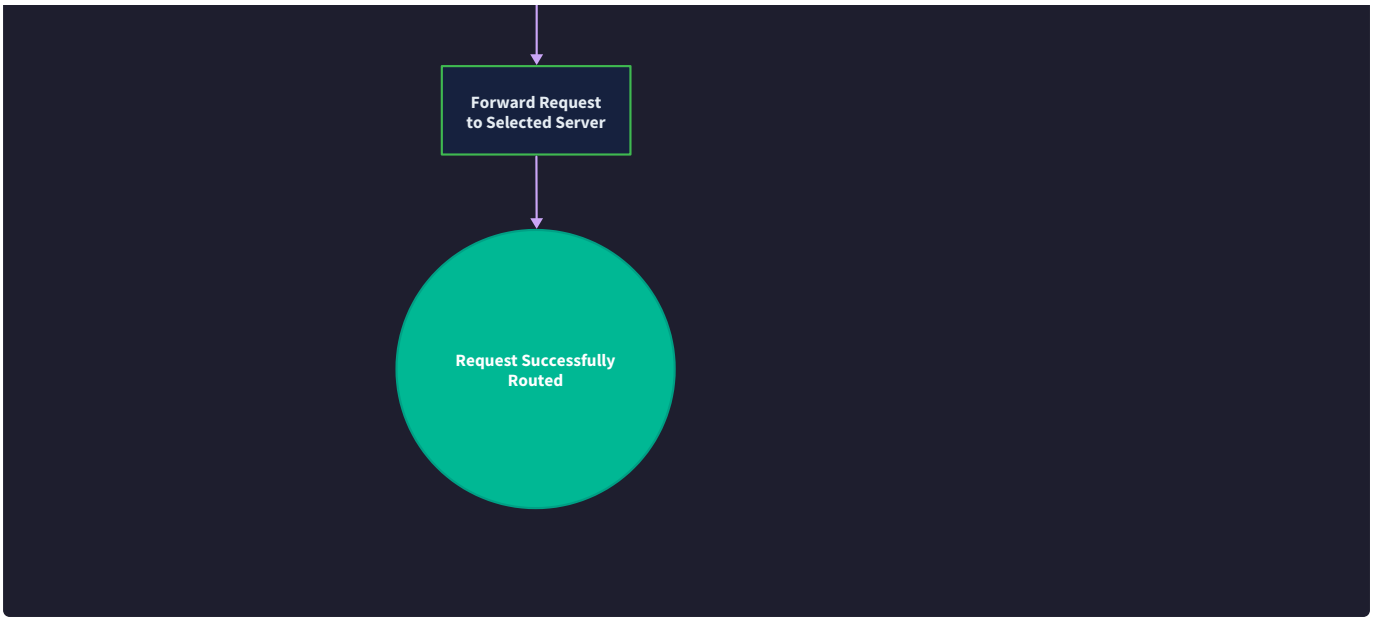
> **Milestone(s):** Milestone 2 (Load Balancing) - implements request distribution across backend servers with health checking and dynamic backend management.

Think of a load balancer as the dispatcher at a busy restaurant during peak hours. When customers arrive, the dispatcher doesn't just randomly assign them to tables - they consider which servers have the lightest workload, which tables are currently available, and which servers might be on break due to illness. The dispatcher keeps track of each server's current capacity and health status, ensuring that no single server gets overwhelmed while others sit idle. Similarly, our load balancer component acts as this intelligent dispatcher, examining each incoming request and making smart decisions about which backend server should handle it based on current load, server health, and configured distribution policies.

The **load balancer component** sits at the heart of our reverse proxy's request distribution logic, responsible for selecting the most appropriate backend server for each incoming client request. This component must balance multiple competing objectives: distributing load evenly across available servers, respecting server capacity differences, avoiding unhealthy servers, and maintaining session affinity where required. Unlike a simple random selection, the load balancer maintains sophisticated state about each backend server, continuously monitoring their health and performance characteristics to make informed routing decisions.

The challenge lies in implementing this intelligence efficiently - the load balancer operates in the critical path of every request, so any slowdown here directly impacts client response times. The component must make routing decisions in microseconds while maintaining accurate load statistics, performing health checks, and handling dynamic backend changes without dropping requests. This requires careful algorithm selection, efficient data structures, and robust error handling to ensure that backend failures don't cascade into client-visible outages.


Incoming Request
Health Check Validation
Perform checks
Available Healthy Servers?
Yes
No
Select Load Balancing Algorithm
Fallback Servers Available?
Round Robin
Least Connections
Weighted
Yes
No
Round Robin Next in sequence
Least Connections Find minimum active
Weighted Round Robin Consider server capacity
No - Retry
Emergency Response Static error page
Circuit Breaker Triggered
Selected Server Still Healthy?
Service Unavailable
Yes



Load Balancing Algorithms

The foundation of effective load balancing lies in choosing the right algorithm for your specific traffic patterns and backend characteristics. Each algorithm makes different assumptions about server capabilities, request costs, and client behavior, leading to vastly different performance outcomes under varying conditions.

Round-robin algorithm represents the simplest approach to load distribution, cycling through available backend servers in a fixed order regardless of their current load or capacity. Think of it like dealing cards from a deck - each backend gets the next request in sequence, wrapping back to the first server after reaching the last one. While this approach ensures perfectly even distribution over time, it assumes all servers have identical capacity and all requests require similar processing resources.

The round-robin implementation maintains a simple counter that tracks the index of the last selected server within the available backend list. For each new request, the algorithm increments this counter modulo the number of healthy backends, ensuring automatic wraparound when reaching the end of the list. This stateless approach makes round-robin extremely fast and memory-efficient, requiring only a single integer of state per upstream group.

However, round-robin's simplicity becomes a liability when servers have different capabilities or when request processing times vary significantly. A powerful server with twice the CPU cores of its peers will receive the same number of requests as the weaker machines, leading to suboptimal resource utilization. Similarly, if some requests trigger expensive database queries while others serve cached content, the server unlucky enough to receive multiple expensive requests in sequence may become overloaded while others remain idle.

Least-connections algorithm addresses round-robin's load blindness by tracking the number of active connections to each backend server and preferentially routing new requests to the server with the fewest current connections. This approach assumes that connection count serves as a reasonable proxy for server load - servers handling more concurrent requests are likely under higher load than those with fewer active connections.

The algorithm maintains a connection counter for each backend server, incrementing the counter when establishing a new connection and decrementing it when connections close or requests complete. For each incoming request, the load balancer scans all healthy backends to find the server with the minimum connection count. In case of ties, the algorithm can break them using secondary criteria such as server ID for deterministic behavior or random selection for additional load spreading.

Least-connections works particularly well for applications with long-lived connections or requests with highly variable processing times. Web applications that mix quick static file requests with expensive API calls benefit significantly from this approach, as servers can naturally shed load by completing requests quickly rather than being forced to accept new work regardless of their current burden.

The algorithm does introduce additional complexity and state management overhead compared to round-robin. Each connection establishment and termination requires updating the connection counters in a thread-safe manner, and the backend selection process requires examining all available servers rather than simple arithmetic. However, this overhead is typically insignificant compared to the performance benefits of better load distribution.

Weighted round-robin algorithm extends basic round-robin by allowing administrators to assign different weights to backend servers based on their relative capacity or performance characteristics. A server with weight 3 will receive three times as many requests as a server with weight 1, allowing the load balancer to account for hardware differences, geographic proximity, or other capacity factors.

The implementation uses a more sophisticated counter system that tracks both the current server index and the number of requests sent to that server within its weight allocation. When a server receives its full allocation of requests (equal to its weight), the algorithm advances to the next server and resets the request counter. This ensures that over any complete cycle, each server receives requests proportional to its configured weight.

Consider a backend pool with three servers having weights 3, 2, and 1 respectively. The distribution pattern over six requests would be: Server A (weight 3) receives requests 1, 2, 3; Server B (weight 2) receives requests 4, 5; Server C (weight 1) receives request 6. This pattern repeats for subsequent request batches, maintaining the 3:2:1 ratio over time while preserving the round-robin property of cycling through all servers.

Weighted round-robin proves invaluable in heterogeneous environments where backend servers have significantly different capabilities. Cloud deployments often mix instance types with different CPU, memory, and network characteristics, making it critical to account for these differences in load distribution. The algorithm also supports gradual traffic shifting scenarios, such as blue-green deployments where new server versions initially receive low weights that gradually increase as confidence grows.

The following table compares the characteristics of each load balancing algorithm:

Algorithm	Complexity	Memory Usage	Load Awareness	Hardware Heterogeneity	Session Affinity
Round-Robin	O(1)	Minimal	None	Poor	Difficult
Least-Connections	O(n)	Moderate	Good	Good	Natural
Weighted Round-Robin	O(1)	Minimal	None	Excellent	Moderate

Decision: Primary Load Balancing Algorithm Selection

- **Context**: The reverse proxy must efficiently distribute requests across backend servers while supporting different server capacities and providing good performance under various traffic patterns.
- **Options Considered**:
  1. Round-robin only (simplest implementation)
  2. Least-connections only (best load awareness)
  3. Multiple algorithms with runtime selection (maximum flexibility)
- **Decision**: Implement all three algorithms with runtime configuration selection
- **Rationale**: Different applications have vastly different characteristics - some benefit from round-robin's simplicity and predictability, others need least-connections' load awareness, and heterogeneous environments require weighted distribution. Supporting multiple algorithms allows the reverse proxy to adapt to diverse deployment scenarios.
- **Consequences**: Increases implementation complexity and testing surface area, but provides maximum deployment flexibility and better performance across diverse workloads.

Health Checking System

Health checking forms the critical foundation that prevents the load balancer from routing requests to failed or degraded backend servers. Without robust health checking, even the most sophisticated load balancing algorithm becomes counterproductive, as it may consistently route requests to servers that cannot fulfill them, leading to cascading failures and poor user experience.

Think of health checking as the reverse proxy's immune system - it continuously monitors the health of each backend server, detecting problems before they impact client requests, and automatically removing unhealthy servers from the active pool while attempting to rehabilitate them. This system must balance responsiveness (detecting failures quickly) with stability (avoiding false positives that unnecessarily remove healthy servers).

Active health checking proactively monitors backend server health by periodically sending synthetic health check requests and evaluating the responses. This approach provides the most reliable health assessment because it exercises the same code paths that real client requests would use, detecting application-level failures that might not be visible through simple network connectivity tests.

The health checker maintains a separate connection pool for health check requests, isolated from the main request processing to ensure that health checks continue even when the backend server is under heavy load or experiencing connection pool exhaustion. Each backend server receives health check requests at configurable intervals, typically ranging from 5 to 30 seconds depending on the desired failure detection speed and the overhead tolerance.

Health check requests should target a lightweight endpoint specifically designed for health monitoring rather than expensive application routes. Many applications provide dedicated health check endpoints (such as `/health` or `/ping`) that perform essential system checks without the overhead of full request processing. These endpoints typically verify database connectivity, cache availability, and other critical dependencies while returning simple success indicators.

The health checker evaluates multiple criteria when determining server health status. HTTP response status codes provide the primary indicator - responses in the 200-299 range typically indicate healthy servers, while 5xx errors suggest server problems. However, the system also monitors response times, as servers that respond correctly but very slowly may indicate resource exhaustion or performance degradation that warrants load reduction.

Failure threshold logic prevents transient network issues or temporary server hiccups from unnecessarily removing healthy servers from the active pool. Rather than marking servers unhealthy after a single failed health check, the system implements configurable failure thresholds that require multiple consecutive failures before changing server status.

A typical failure threshold configuration might require three consecutive health check failures before marking a server unhealthy, and five consecutive successes before returning it to service. This asymmetric threshold design reflects the principle that removing servers from service should be conservative (to avoid unnecessary capacity reduction) while returning servers to service should be even more conservative (to avoid repeatedly adding and removing flapping servers).

The health checking system maintains detailed state for each backend server, tracking not only the current health status but also the history of recent health check results, failure counts, and timing information. This rich state enables sophisticated health assessment logic that can detect patterns such as intermittent failures, gradual performance degradation, or partial service degradation.

Health Check Parameter	Typical Range	Purpose	Impact of Too Low	Impact of Too High
Check Interval	5-30 seconds	Detection speed vs overhead	Higher server load	Slower failure detection
Timeout	1-10 seconds	Unresponsive server detection	False positives	Slow detection of hanging servers
Failure Threshold	2-5 failures	Stability vs responsiveness	Flapping servers	Slow unhealthy server removal
Success Threshold	3-7 successes	Recovery confidence	Premature server return	Slow healthy server recovery

Graceful degradation ensures that health check failures don't immediately impact in-flight requests to the affected backend server. When a server transitions from healthy to unhealthy status, the load balancer stops routing new requests to that server but allows existing connections and requests to complete naturally. This approach prevents abrupt connection termination that could cause client errors while still protecting against sending additional load to problematic servers.

The system implements a grace period during which unhealthy servers remain in a "draining" state, continuing to handle existing connections while being excluded from new request routing. The duration of this grace period balances client experience (allowing time for requests to complete) with failure isolation (limiting exposure to a potentially failing server).

For servers that fail health checks but maintain existing connections, the connection manager monitors these connections more aggressively, applying shorter timeouts and more aggressive error detection to prevent hanging connections from consuming resources indefinitely.

> **Decision: Health Check Strategy**
>
> - **Context**: Backend servers can fail in various ways (network issues, application crashes, performance degradation), and the load balancer must detect these failures quickly while avoiding false positives that unnecessarily reduce capacity.
> - **Options Considered**:
>   1. Active HTTP health checks only
>   2. Passive failure detection based on request failures
>   3. Combined active and passive health checking
> - **Decision**: Implement active HTTP health checks with passive failure detection as a backup
> - **Rationale**: Active health checks provide proactive failure detection and can catch issues before they impact client requests. Passive detection serves as a safety net for failures that might not be caught by periodic health checks, such as servers that pass health checks but fail real requests due to race conditions or resource exhaustion.
> - **Consequences**: Requires additional implementation complexity and generates constant background traffic to backend servers, but provides the most reliable failure detection and fastest recovery times.

Load Balancing Decisions

The architecture of the load balancing component requires several critical design decisions that fundamentally impact performance, reliability, and operational characteristics. These decisions interact with each other in complex ways, making it essential to understand the trade-offs and select options that align with the overall system goals.

Backend selection synchronization addresses the challenge of making load balancing decisions safely in a multi-threaded environment where multiple worker threads may simultaneously attempt to select backend servers for different client requests. The naive approach of protecting the entire backend selection process with a single mutex would create a severe bottleneck, as every request would need to acquire the same lock before proceeding.

The selected approach uses fine-grained locking with separate mutexes for different aspects of the backend selection process. The backend server list itself is protected by a read-write lock that allows multiple threads to read the list simultaneously while ensuring exclusive access for updates when servers are added, removed, or change health status. Load balancing algorithm state (such as connection counters for least-connections or round-robin positions) uses separate synchronization primitives appropriate to each algorithm's needs.

Round-robin algorithm state uses atomic integer operations for the server index counter, eliminating lock contention entirely for the common case of backend selection. Least-connections algorithm requires more complex synchronization because it needs to both read connection counts across all servers and update the selected server's count, which is handled through a combination of atomic operations for individual counters and brief critical sections for the selection logic.

> **Decision: Backend Selection Synchronization Strategy**
>
> - **Context**: Multiple worker threads need to select backend servers simultaneously, and the backend selection process is in the critical path for request processing performance.
> - **Options Considered**:
>   1. Single global mutex protecting all backend selection
>   2. Per-algorithm synchronization strategies with minimal locking
>   3. Lock-free algorithms using atomic operations and hazard pointers
> - **Decision**: Implement per-algorithm synchronization with atomic operations where possible and fine-grained locking where necessary
> - **Rationale**: Single global mutex would create unacceptable performance bottlenecks under high concurrency. Lock-free algorithms provide the best performance but significantly increase implementation complexity and debugging difficulty. Per-algorithm synchronization strikes the right balance of performance and maintainability.
> - **Consequences**: Requires careful implementation of each algorithm's synchronization needs and thorough testing for race conditions, but provides good performance scaling with multiple worker threads.

Health check integration determines how health status changes propagate to the load balancing algorithm and how quickly the system responds to backend server state transitions. The health checking system operates on a different timeline than request processing - health checks occur every few seconds while request routing happens thousands of times per second.

The integration design uses event-driven health status updates where health check results trigger immediate updates to the load balancing algorithm's view of available servers. When a server transitions from healthy to unhealthy, the health checker immediately removes it from all load balancing algorithms' active server lists. Conversely, when an unhealthy server recovers, it gets added back to the active lists with appropriate state initialization (such as resetting connection counters for least-connections).

This event-driven approach ensures that load balancing decisions always reflect the most current health information without requiring the request processing path to perform health status checks. The alternative of having each request check server health status would introduce unacceptable latency and complexity to the critical path.

Server weight management for weighted round-robin algorithm requires careful consideration of how weight changes propagate and when they take effect. Changing server weights while the algorithm is actively processing requests could disrupt the distribution pattern or create inconsistent behavior.

The implementation uses a two-phase weight update process where new weights are staged in a separate configuration structure and then atomically activated during the next algorithm cycle completion. This approach ensures that weight changes take effect cleanly without disrupting ongoing request distribution patterns or creating race conditions between configuration updates and request processing.

The system also supports dynamic weight adjustment based on server performance metrics, allowing weights to automatically decrease for servers showing signs of stress (such as increasing response times) and increase for servers with excess capacity.

This adaptive weighting provides a bridge between the simplicity of weighted round-robin and the load awareness of least-connections.

Synchronization Approach	Pros	Cons	Best Use Case
Global Mutex	Simple implementation, Easy reasoning	Severe performance bottleneck	Low-concurrency deployments
Per-Algorithm Locks	Balanced complexity/performance	Still some lock contention	Most production deployments
Lock-Free Atomic	Maximum performance	High implementation complexity	Ultra-high performance requirements

Error handling and fallback logic defines how the load balancer behaves when all backend servers are unhealthy or when the selected backend server fails after being chosen but before the request is forwarded. These edge cases require careful handling to provide graceful degradation rather than complete service failure.

When all backend servers in a pool are marked unhealthy, the load balancer implements a "best effort" fallback mode where it continues routing requests to the least-recently-failed servers while continuing aggressive health checking. This approach recognizes that health check failures might represent false positives due to network issues or temporary overload, and that attempting to serve requests is better than refusing all traffic.

The system maintains a "failure recency" ranking for unhealthy servers, preferring servers that failed more recently (and thus might have a higher chance of recovery) over servers that have been failing for extended periods. This heuristic helps identify servers that might be experiencing transient issues versus those with more serious problems.

For requests that fail after backend selection (such as connection establishment failures or immediate HTTP errors), the load balancer implements limited retry logic with different backend selection. A request that fails to connect to the initially selected backend server will be retried against up to two additional servers before returning an error to the client. This retry logic uses an exponential backoff delay to avoid overwhelming failing servers while still providing reasonable client response times.

Common Pitfalls

⚠️ Pitfall: Ignoring Connection State During Server Health Transitions

Many implementations make the mistake of immediately updating load balancing algorithm state when servers transition between healthy and unhealthy status, without considering the impact on existing connections to those servers. When a server becomes unhealthy, naive implementations might immediately mark all its connections as bad and close them, causing unnecessary client errors for requests that could have completed successfully.

The correct approach requires coordinating between the health checking system and the connection manager to handle server state transitions gracefully. When a server becomes unhealthy, new requests should stop being routed to it, but existing connections should be allowed to complete normally unless they individually show signs of failure. This requires the load balancer to distinguish between "don't send new requests here" and "this server is completely unusable."

⚠️ Pitfall: Race Conditions in Connection Count Updates

The least-connections algorithm requires careful synchronization when updating connection counts, particularly during rapid connection establishment and termination. A common mistake is updating connection counts at the wrong time in the connection lifecycle, leading to inaccurate counts that cause poor load distribution or even integer underflow when decrementing counts for connections that weren't properly registered.

Connection counts must be updated atomically and at precisely defined points in the connection lifecycle: increment when a connection is successfully established and assigned to a backend server, decrement when the connection is closed or returned to

the connection pool. The count update must be paired with the actual connection state change to prevent race conditions where the count and reality diverge.

⚠️ Pitfall: Not Handling Backend List Changes During Request Processing

Backend server lists can change dynamically as servers are added, removed, or change health status. Implementations often fail to properly handle these changes when they occur during active request processing, leading to array bounds errors, null pointer dereferences, or routing requests to servers that no longer exist.

The solution requires implementing proper read-copy-update semantics for the backend server list, where request processing threads work with stable snapshots of the server list while background threads can safely update the master list. This prevents request processing from being impacted by concurrent backend list modifications while ensuring that changes eventually become visible to new requests.

Implementation Guidance

The load balancer component builds upon the connection manager and HTTP parser to provide intelligent request distribution across backend servers. This component operates entirely within the request processing critical path, so implementation choices directly impact overall proxy performance.

Technology Recommendations:

Component	Simple Option	Advanced Option
Algorithm Selection	Single round-robin implementation	Pluggable algorithm interface with runtime selection
Health Checking	Synchronous health checks in main loop	Dedicated health check thread with async updates
Backend Configuration	Static configuration file	Dynamic backend management with configuration reload
Metrics Collection	Simple counters	Detailed per-backend performance metrics

Recommended File Structure:

```
internal/loadbalancer/
  loadbalancer.go          ← main LoadBalancer struct and public interface
  algorithms.go            ← load balancing algorithm implementations
  healthcheck.go           ← health checking system and backend monitoring
  backend.go               ← BackendServer management and configuration
  metrics.go               ← load balancing metrics and statistics
  loadbalancer_test.go     ← unit tests for load balancing logic
  algorithms_test.go       ← algorithm-specific test cases
  healthcheck_test.go      ← health checking test scenarios
```

Load Balancer Infrastructure Code:

```c
// LoadBalancer manages request distribution across backend servers
typedef struct {
    BackendServer** backends;
    size_t backend_count;
    LoadBalancingAlgorithm algorithm;

    // Algorithm-specific state
    size_t rr_current_index;
    uint32_t* connection_counts;
    uint32_t* weights;
    uint32_t* current_weights;

    // Health checking
    pthread_t health_check_thread;
    bool* backend_healthy;
    time_t* last_health_check;
    time_t* last_failure_time;
    uint32_t* failure_count;

    // Synchronization
    pthread_rwlock_t backends_lock;
    pthread_mutex_t state_mutex;

    // Configuration
    int health_check_interval;
    int health_check_timeout;
    int failure_threshold;
    int success_threshold;

    bool running;
} LoadBalancer;
```

```c
// Backend server configuration and runtime state
typedef struct {
    char host[256];
    int port;
    int weight;
    bool enabled;

    // Health check configuration
    char health_check_path[512];
    int health_check_port;

    // Runtime metrics
    uint32_t active_connections;
    uint64_t total_requests;
    uint64_t failed_requests;
    double avg_response_time;
    time_t last_success_time;
    time_t last_failure_time;
} BackendServer;

// Load balancing algorithm enumeration
typedef enum {
    LB_ROUND_ROBIN,
    LB_LEAST_CONNECTIONS,
    LB_WEIGHTED_ROUND_ROBIN,
    LB_IP_HASH
} LoadBalancingAlgorithm;

LoadBalancer* loadbalancer_create(BackendServer** backends, size_t count, LoadBalancingAlgorithm algorithm);
void loadbalancer_destroy(LoadBalancer* lb);
bool loadbalancer_start(LoadBalancer* lb);
void loadbalancer_stop(LoadBalancer* lb);
BackendServer* loadbalancer_select_backend(LoadBalancer* lb, HttpRequest* request);
```

```
void loadbalancer_update_connection_count(LoadBalancer* lb, BackendServer* backend, int delta);

bool loadbalancer_is_backend_healthy(LoadBalancer* lb, BackendServer* backend);
```

Core Load Balancing Algorithm Skeletons:

```c
// SelectBackendRoundRobin selects the next backend using round-robin algorithm.
// This algorithm cycles through healthy backends in order, providing even distribution.
BackendServer* select_backend_round_robin(LoadBalancer* lb) {
    // TODO 1: Acquire read lock on backends list to ensure stable access
    // TODO 2: Count number of healthy backends in current list
    // TODO 3: If no healthy backends available, return NULL
    // TODO 4: Use atomic increment on rr_current_index to get next position
    // TODO 5: Modulo operation to wrap around when reaching end of healthy backends
    // TODO 6: Scan from current position to find next healthy backend
    // TODO 7: Handle wraparound case if no healthy backends found after current position
    // TODO 8: Release read lock and return selected backend
    // Hint: Use __atomic_fetch_add for thread-safe index increment
}

// SelectBackendLeastConnections selects backend with minimum active connections.
// This algorithm examines all healthy backends to find the least loaded server.
BackendServer* select_backend_least_connections(LoadBalancer* lb) {
    // TODO 1: Acquire read lock on backends list for stable access
    // TODO 2: Initialize min_connections to UINT32_MAX and selected_backend to NULL
    // TODO 3: Iterate through all backends in the list
    // TODO 4: Skip backends that are marked unhealthy
    // TODO 5: Read current connection count using atomic operation
    // TODO 6: Compare with current minimum, update if lower
    // TODO 7: Handle tie-breaking by preferring backends with lower index
    // TODO 8: Increment selected backend's connection count atomically
    // TODO 9: Release read lock and return selected backend
    // Hint: Use __atomic_load for reading connection counts safely
}

// SelectBackendWeightedRoundRobin selects backend based on configured weights.
// This algorithm ensures backends receive requests proportional to their weights.
BackendServer* select_backend_weighted_round_robin(LoadBalancer* lb) {
    // TODO 1: Acquire read lock on backends list
```

```
    // TODO 2: Find backend with highest current_weight among healthy backends

    // TODO 3: Select backend with maximum current_weight value

    // TODO 4: Decrease selected backend's current_weight by sum of all backend weights

    // TODO 5: Increase all healthy backends' current_weight by their configured weight

    // TODO 6: Handle edge case where all backends have zero weight

    // TODO 7: Release read lock and return selected backend

    // Hint: This implements the "smooth weighted round-robin" algorithm

}
```

Health Checking System Implementation:

```c
// HealthCheckBackend performs a single health check against specified backend.
// Returns true if backend responds successfully, false otherwise.
bool health_check_backend(LoadBalancer* lb, BackendServer* backend) {
    // TODO 1: Create TCP socket with non-blocking flag
    // TODO 2: Set socket timeout using SO_RCVTIMEO and SO_SNDTIMEO
    // TODO 3: Connect to backend's health check port (or main port if not specified)
    // TODO 4: Send HTTP GET request to health check path
    // TODO 5: Read HTTP response with timeout handling
    // TODO 6: Parse response status code from response line
    // TODO 7: Consider 200-299 status codes as healthy, others as unhealthy
    // TODO 8: Close socket and return health status
    // Hint: Use poll() or select() for timeout handling during connect and read
}

// HealthCheckLoop runs continuously in background thread checking all backends.
// This function implements the main health checking logic with configurable intervals.
void* health_check_loop(void* arg) {
    LoadBalancer* lb = (LoadBalancer*)arg;

    // TODO 1: Loop while lb->running flag is true
    // TODO 2: Sleep for health_check_interval seconds (use nanosleep for precision)
    // TODO 3: Iterate through all configured backends
    // TODO 4: Skip disabled backends in configuration
    // TODO 5: Perform health check using health_check_backend function
    // TODO 6: Update failure/success counts based on health check result
    // TODO 7: Apply failure_threshold and success_threshold logic
    // TODO 8: Update backend_healthy array when status changes
    // TODO 9: Log health status changes for operational visibility
    // TODO 10: Update last_health_check timestamp
    // Hint: Use compare-and-swap for atomic backend status updates
}

// UpdateBackendHealth changes backend health status with proper synchronization.
```

```
// This function handles the transition between healthy and unhealthy states.
void update_backend_health(LoadBalancer* lb, size_t backend_index, bool healthy) {
    // TODO 1: Acquire write lock on backends to ensure exclusive access
    // TODO 2: Check if health status is actually changing
    // TODO 3: Update backend_healthy array with new status
    // TODO 4: Reset failure/success counters when status changes
    // TODO 5: Update last_failure_time or last_success_time as appropriate
    // TODO 6: Log status change with backend details for monitoring
    // TODO 7: Release write lock
    // Hint: Only log when status actually changes to avoid log spam
}
```

Milestone Checkpoint:

After implementing the load balancer component, verify correct operation by:

1. **Algorithm Testing**: Start the proxy with different load balancing algorithms and send multiple requests. Observe backend selection patterns:
   - Round-robin should cycle through backends evenly
   - Least-connections should prefer backends with fewer active connections
   - Weighted round-robin should respect configured backend weights
2. **Health Check Verification**: Stop one backend server and observe health check behavior:
   - Requests should stop routing to the failed backend within one health check interval
   - Backend should return to service after restarting and passing success threshold
   - Check logs for health status change notifications
3. **Concurrent Access Testing**: Use load testing tools to send many concurrent requests:
   - No race condition errors or crashes should occur
   - Backend selection should remain fair under high load
   - Health checking should continue operating during heavy traffic
4. **Configuration Changes**: Test dynamic backend configuration updates:
   - Adding new backends should start receiving requests immediately
   - Removing backends should drain existing connections gracefully
   - Weight changes should affect distribution in next algorithm cycle

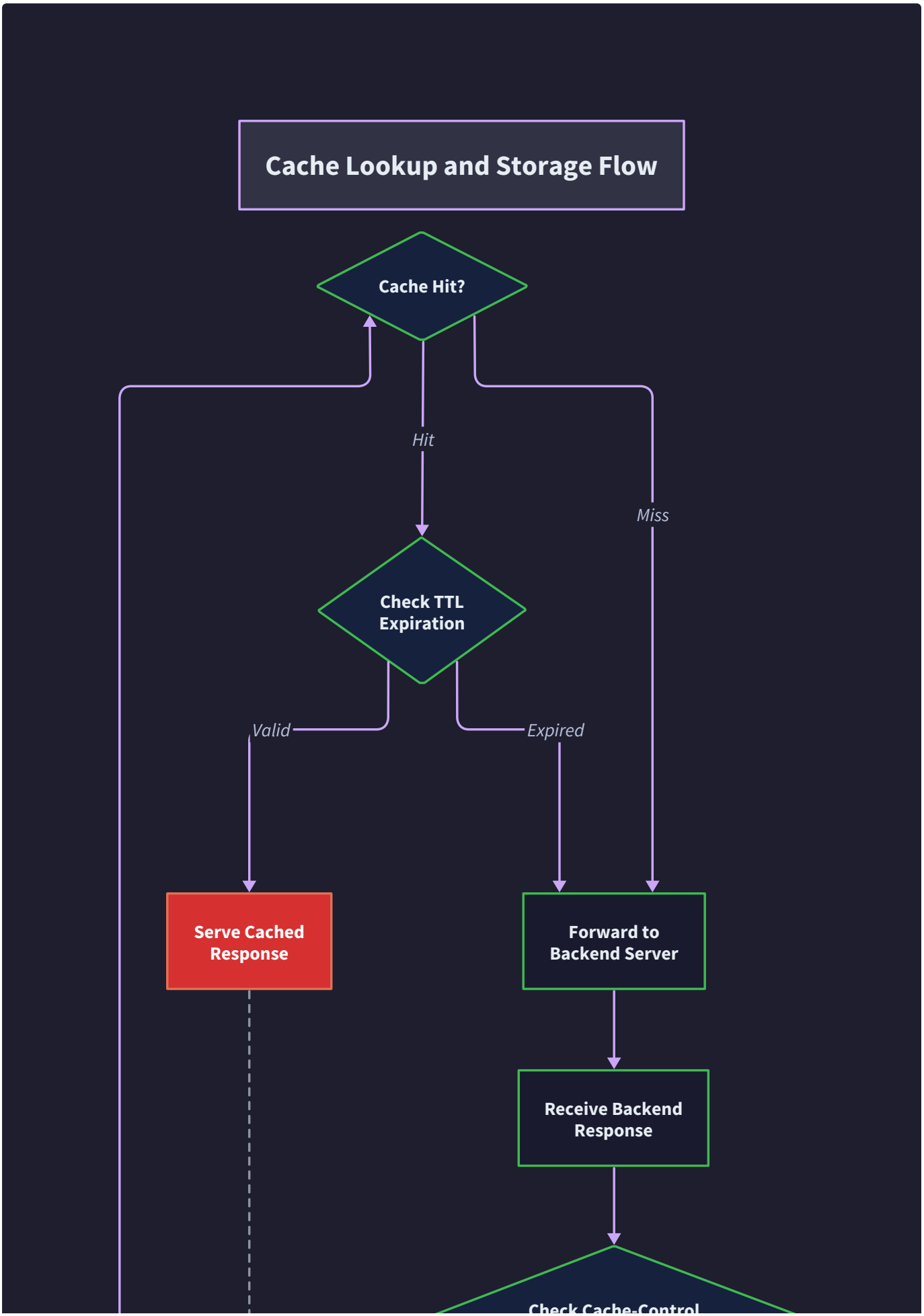
Expected behavior: The load balancer should distribute requests according to the configured algorithm, automatically remove failed backends from service, and handle high concurrency without race conditions or performance degradation.

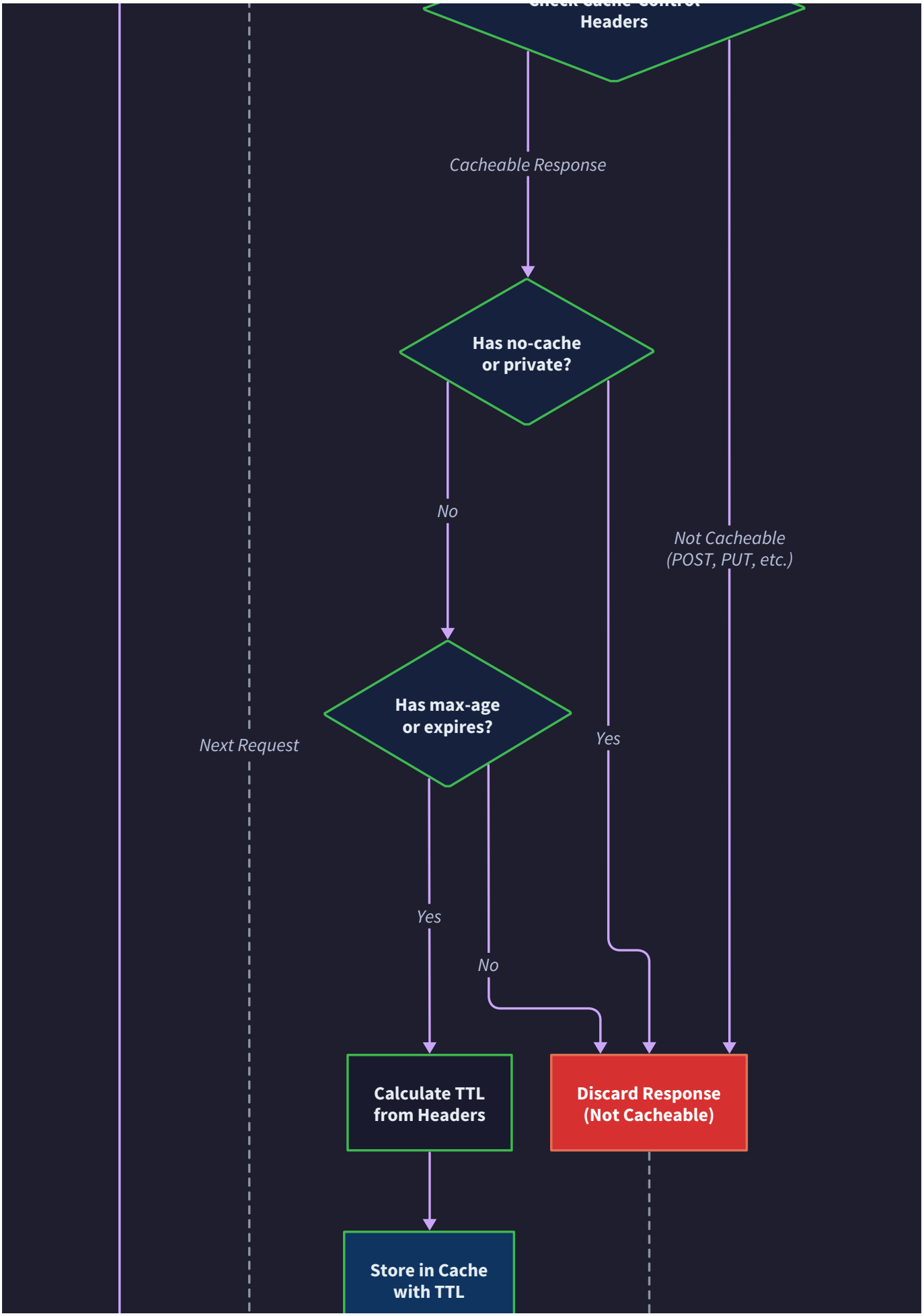
Cache Engine Component

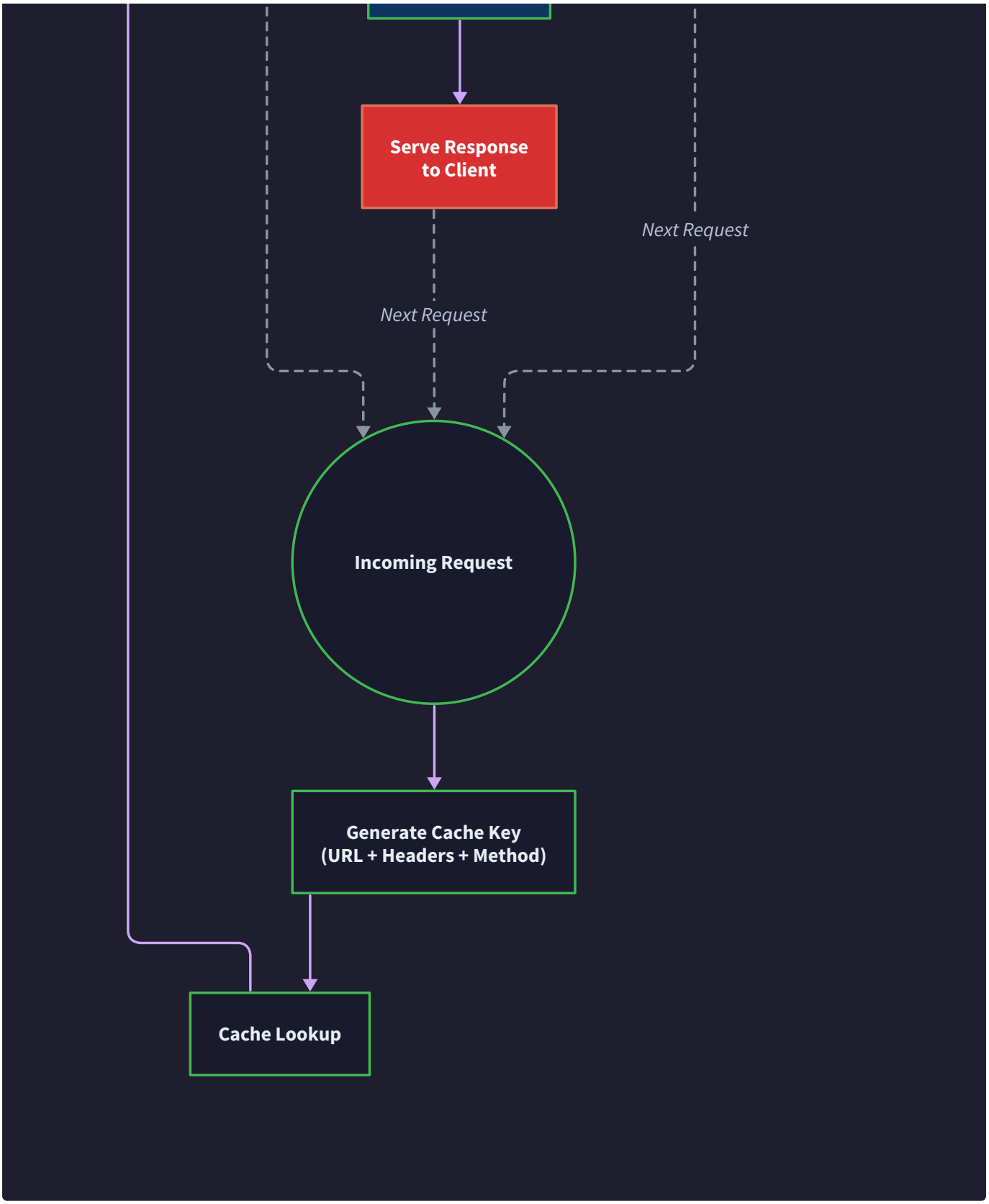
> **Milestone(s):** Milestone 4 (Caching) - implements HTTP response caching with proper cache-control header handling, TTL management, and cache invalidation strategies.

Think of the cache engine as a smart librarian for your reverse proxy. Just as a librarian maintains an organized collection of books, knows which books are popular and should stay on the easy-access shelves, and periodically removes outdated materials, the cache engine stores frequently requested responses, understands HTTP caching rules, and automatically removes stale content. The librarian doesn't store every book ever requested (space is limited), doesn't lend out damaged books (respects cache-control directives), and knows when a book's information is too old to be useful (TTL expiration). This analogy captures the three core responsibilities: intelligent storage decisions, rule-based access control, and proactive cleanup.

The cache engine sits between the load balancer and backend servers, intercepting requests before they reach the backend and serving cached responses when possible. This positioning allows the cache to dramatically reduce backend load by serving repeated requests from memory, while respecting HTTP semantics ensures clients receive correct, up-to-date responses. The cache engine must balance performance gains against memory usage and cache coherence requirements.


Cache Lookup and Storage Flow
Cache Hit?
Hit
Miss
Check TTL Expiration
Valid
Expired
Serve Cached Response
Forward to Backend Server
Receive Backend Response
Check Cache-Control


Headers
Cacheable Response
Has no-cache or private?
No
Not Cacheable (POST, PUT, etc.)
Has max-age or expires?
Next Request
Yes
Yes
No
Calculate TTL from Headers
Discard Response (Not Cacheable)
Store in Cache with TTL



Caching Strategy

The caching strategy encompasses three fundamental decisions: what to cache, how to organize cached content, and when to remove it. These decisions directly impact both performance gains and memory efficiency.

Cache Key Generation

Cache keys uniquely identify cacheable responses and determine whether two requests can share the same cached response. Think of cache keys as precise addresses in a massive warehouse - they must be specific enough to avoid delivering the wrong item, but consistent enough that identical requests always generate the same address.

The cache key generation process follows a deterministic algorithm that combines multiple request attributes:

1. **Base Key Construction**: Start with the HTTP method and normalized URI (lowercased, sorted query parameters)
2. **Header Variation Handling**: Examine the response `Vary` header to determine which request headers affect the response
3. **Header Value Incorporation**: For each header listed in `Vary`, append its value to the cache key
4. **Authorization Consideration**: Never cache responses for requests containing authorization headers unless explicitly allowed
5. **Normalization**: Apply consistent encoding and ordering to ensure identical requests produce identical keys

Key Component	Source	Example	Purpose
HTTP Method	Request line	`GET`	Distinguish method semantics
Normalized URI	Request line	`/api/users?sort=name&limit=10`	Core resource identifier
Vary Headers	Response Vary + Request	`accept-encoding:gzip`	Content negotiation
Host Header	Request headers	`api.example.com`	Multi-tenant support
Query Parameters	URI query string	`limit=10&sort=name`	Parameterized requests

Storage Architecture

The cache engine uses a two-level storage architecture combining hash table lookup with LRU eviction tracking. This design provides O(1) average-case lookup performance while maintaining efficient memory management.

The primary storage structure is a hash table mapping cache keys to cache entries. Each cache entry contains the complete HTTP response (status, headers, body) plus metadata for cache management. A separate doubly-linked list tracks access recency for LRU eviction decisions.

Storage Component	Purpose	Data Structure	Access Pattern
Primary Index	Fast key lookup	Hash table	O(1) average lookup
Eviction Tracker	LRU ordering	Doubly-linked list	O(1) move-to-head
Size Monitor	Memory limits	Running total	O(1) size tracking
TTL Index	Expiration cleanup	Min-heap by expiry	O(log n) expiration

> **Design Insight**: The two-level architecture separates concerns between access speed (hash table) and replacement policy (LRU list). This separation allows independent optimization of each concern and supports future replacement policy changes without affecting the lookup mechanism.

Eviction Policies

The cache engine implements multiple eviction triggers to maintain memory limits and data freshness. Eviction operates as a background process triggered by memory pressure or expiration events.

LRU (Least Recently Used) Eviction removes the oldest accessed entries when memory limits are reached. Each cache access moves the corresponding entry to the head of the LRU list, maintaining precise access ordering. When eviction is needed, entries are removed from the tail until sufficient space is available.

TTL (Time-To-Live) Expiration removes entries that exceed their maximum age, regardless of access patterns. Each cache entry stores an absolute expiration time calculated from the response `Cache-Control` directives. A background thread periodically scans for expired entries and removes them proactively.

Size-Based Eviction prevents individual large responses from consuming excessive memory. Responses exceeding a configurable size threshold are never cached, and existing entries may be evicted if they prevent caching newer, smaller responses.

Decision: Hybrid Eviction Strategy

- **Context**: Different response patterns require different eviction strategies - some content becomes stale (TTL), others are rarely accessed (LRU), and some consume too much memory (size-based)
- **Options Considered**:
  1. TTL-only eviction with fixed expiration times
  2. Pure LRU eviction based solely on access patterns
  3. Hybrid approach combining TTL, LRU, and size constraints
- **Decision**: Implement hybrid eviction with TTL, LRU, and size-based triggers
- **Rationale**: HTTP caching semantics require TTL respect for correctness, LRU provides performance optimization, and size limits prevent memory exhaustion from large responses
- **Consequences**: More complex implementation but correctly handles diverse caching scenarios while providing memory safety guarantees

Eviction Type	Trigger Condition	Selection Criteria	Performance Impact
TTL Expiration	Entry exceeds max-age	Absolute expiry time	O(1) per entry
LRU Pressure	Memory limit exceeded	Least recently accessed	O(1) per eviction
Size Rejection	Response too large	Individual response size	O(1) size check
Proactive Cleanup	Periodic background	Batch expired entries	O(n) scan overhead

HTTP Cache-Control Handling

HTTP cache-control handling ensures the cache engine respects standard HTTP semantics while maximizing caching opportunities. Think of cache-control directives as traffic signals for cached content - they provide clear rules about when it's safe to serve cached responses, when fresh validation is required, and when caching is prohibited entirely.

Cache-Control Directive Processing

The cache engine parses and evaluates cache-control directives from both request and response headers. Response directives control cachability and expiration, while request directives influence cache lookups and validation requirements.

Response Cache-Control Evaluation:

1. **Parse Response Headers**: Extract and parse the `Cache-Control` header into individual directives
2. **Evaluate Cachability**: Check for `no-store` (never cache), `no-cache` (cache but validate), and `private` (cache only in private caches)
3. **Calculate Expiration**: Use `max-age` directive or fall back to `Expires` header for TTL calculation
4. **Store Validation Info**: Preserve `ETag` and `Last-Modified` headers for conditional requests
5. **Apply Storage Decision**: Cache the response only if cachable according to evaluated directives

Directive	Effect on Caching	TTL Calculation	Validation Required
`no-store`	Never cache	N/A	N/A
`no-cache`	Cache with validation	From max-age/expires	Always
`private`	Skip (reverse proxy)	N/A	N/A
`public`	Cache allowed	From max-age/expires	When stale
`max-age=N`	Cache for N seconds	N seconds	When stale
`must-revalidate`	Cache with strict validation	From max-age/expires	When stale

Conditional Request Handling

Conditional requests allow the cache engine to validate stale entries with backend servers, potentially avoiding full response transfers. The cache engine generates conditional requests using stored validation information and processes 304 Not Modified responses appropriately.

Conditional Request Generation Process:

1. **Check Stale Entry**: Identify cached entries that exceed their max-age but have validation headers
2. **Generate If-None-Match**: Use stored `ETag` value to create `If-None-Match` header
3. **Generate If-Modified-Since**: Use stored `Last-Modified` value to create `If-Modified-Since` header
4. **Forward Conditional Request**: Send conditional request to backend server with original URI and validation headers
5. **Process Validation Response**: Handle 304 Not Modified (refresh cache entry) or 200 OK (replace cache entry)

> **Design Insight**: Conditional requests transform cache misses into cache refreshes, significantly reducing bandwidth usage and backend load. A 304 Not Modified response allows the cache to serve the existing response body while updating expiration information, providing the performance benefits of caching with the correctness guarantees of validation.

Vary Header Processing

The `Vary` response header indicates which request headers affect the response content, requiring the cache engine to incorporate these headers into cache key generation. Proper `Vary` handling prevents serving inappropriate cached responses to clients with different capabilities or preferences.

Vary Processing Algorithm:

1. **Parse Vary Header**: Extract list of header names from the response `Vary` header
2. **Extract Request Values**: For each header name in `Vary`, extract the corresponding value from the original request
3. **Normalize Values**: Apply consistent case normalization and whitespace handling to header values
4. **Incorporate into Key**: Append normalized header values to the cache key in deterministic order
5. **Store Vary Information**: Preserve the complete `Vary` list with the cached response for future key generation

Vary Header	Request Impact	Key Component	Example
`Accept-Encoding`	Content compression	`accept-encoding:gzip`	Different compressions cached separately
`Accept-Language`	Content localization	`accept-language:en-US`	Different languages cached separately
`User-Agent`	Browser-specific content	`user-agent:Mozilla/...`	Browser-specific responses
`Accept`	Content type negotiation	`accept:application/json`	JSON vs XML responses

Decision: Full Vary Support with Key Segmentation

- **Context**: The `Vary` header allows responses to depend on arbitrary request headers, creating complex cache key requirements
- **Options Considered**:
  1. Ignore `Vary` headers and cache only the first response variant
  2. Support limited `Vary` headers (only common ones like `Accept-Encoding`)
  3. Full `Vary` support with dynamic key generation
- **Decision**: Implement full `Vary` support with request header incorporation into cache keys
- **Rationale**: Correct HTTP semantics require full `Vary` support to avoid serving incorrect cached responses to clients with different capabilities
- **Consequences**: More complex cache key generation and potentially reduced cache hit rates, but guaranteed correctness for content negotiation scenarios

Caching Design Decisions

The cache engine architecture reflects several critical design decisions that balance performance, correctness, and operational complexity. Each decision involves trade-offs between competing requirements.

Decision: In-Memory Cache with LRU Eviction

- **Context**: The cache engine needs fast access to cached responses while managing memory limits and avoiding storage complexity
- **Options Considered**:
  1. Pure in-memory hash table with no eviction
  2. Disk-backed cache with persistent storage
  3. In-memory cache with LRU eviction
  4. Tiered cache with memory and disk levels
- **Decision**: In-memory cache with LRU eviction and configurable size limits
- **Rationale**: In-memory storage provides microsecond access times essential for reverse proxy performance, LRU eviction prevents memory exhaustion, and avoiding disk eliminates I/O bottlenecks
- **Consequences**: Cache is lost on restart and limited by available memory, but provides maximum performance for actively cached content

Option	Performance	Persistence	Complexity	Memory Usage
Pure Memory	Excellent	None	Low	Unbounded
Disk-Backed	Poor	Full	High	Bounded
Memory + LRU	Excellent	None	Medium	Bounded
Tiered	Good	Partial	Very High	Tiered

Decision: Thread-Safe Cache with Read-Write Locks

- **Context**: Multiple worker threads need concurrent access to cache data structures for both reads (cache lookups) and writes (cache updates and evictions)
- **Options Considered**:
  1. Single-threaded cache with message passing
  2. Coarse-grained mutex protecting entire cache
  3. Fine-grained locking with per-entry locks
  4. Read-write locks with optimistic concurrency
- **Decision**: Read-write locks with separate locks for cache operations and LRU management
- **Rationale**: Cache lookups vastly outnumber updates, making read-write locks optimal for this read-heavy workload with occasional writes
- **Consequences**: Excellent read concurrency but potential write contention during eviction storms; requires careful lock ordering to prevent deadlocks

Decision: Proactive TTL Cleanup with Background Thread

- **Context**: Expired cache entries consume memory and may be served incorrectly if not removed promptly
- **Options Considered**:
  1. Lazy expiration checking only on access
  2. Periodic full cache scanning for expired entries
  3. Priority queue with next-expiry tracking
  4. Background thread with batched cleanup
- **Decision**: Background thread that periodically scans and removes expired entries in batches
- **Rationale**: Proactive cleanup prevents memory leaks from expired content, batched processing amortizes scanning costs, and background execution avoids blocking request processing
- **Consequences**: Constant low-level CPU overhead for cleanup scanning, but guaranteed memory reclamation and elimination of stale content serving

Cache Engine Data Structures

The cache engine maintains several interconnected data structures to support fast lookup, efficient eviction, and proper HTTP semantics compliance.

Structure	Type	Purpose	Key Operations
`cache_table`	`HashTable*`	Primary cache lookup	get, put, remove
`lru_list`	`LRUList*`	Access ordering	move_to_head, remove_tail
`ttl_heap`	`TTLHeap*`	Expiration tracking	peek_expired, remove_expired
`cache_mutex`	`pthread_rwlock_t`	Concurrency control	read_lock, write_lock
`size_current`	`size_t`	Memory usage tracking	atomic_add, atomic_sub
`size_limit`	`size_t`	Memory limit	configuration value

CacheEngine Structure Definition:

Field	Type	Description
`cache_table`	`HashTable*`	Primary storage mapping cache keys to entries
`lru_head`	`CacheEntry*`	Head of doubly-linked LRU list
`lru_tail`	`CacheEntry*`	Tail of doubly-linked LRU list
`ttl_heap`	`TTLHeap*`	Min-heap ordered by expiration time
`cache_rwlock`	`pthread_rwlock_t`	Reader-writer lock for cache operations
`lru_mutex`	`pthread_mutex_t`	Mutex protecting LRU list manipulation
`size_current`	`size_t`	Current total cache size in bytes
`size_limit`	`size_t`	Maximum allowed cache size
`default_ttl`	`time_t`	Default TTL for responses without cache headers
`cleanup_thread`	`pthread_t`	Background thread for TTL cleanup
`cleanup_interval`	`int`	Seconds between cleanup cycles
`hit_count`	`uint64_t`	Cache hit counter for metrics
`miss_count`	`uint64_t`	Cache miss counter for metrics
`eviction_count`	`uint64_t`	Eviction counter for metrics
`running`	`bool`	Flag controlling background thread execution

CacheEntry Structure Definition:

Field	Type	Description
`cache_key`	`char[512]`	Unique identifier for cached response
`response_status`	`int`	HTTP status code of cached response
`response_headers`	`HashTable*`	Complete response header collection
`response_body`	`Buffer*`	Response body content
`content_length`	`size_t`	Size of response body in bytes
`created_time`	`time_t`	When entry was first cached
`last_access`	`time_t`	Most recent access timestamp
`expires_time`	`time_t`	Absolute expiration time
`etag`	`char[256]`	ETag header value for validation
`last_modified`	`char[128]`	Last-Modified header for validation
`vary_headers`	`char[512]`	Comma-separated list of Vary header names
`cache_control`	`CacheControl*`	Parsed cache-control directives
`lru_prev`	`CacheEntry*`	Previous entry in LRU list
`lru_next`	`CacheEntry*`	Next entry in LRU list
`ttl_heap_index`	`size_t`	Position in TTL expiration heap
`entry_size`	`size_t`	Total memory consumed by this entry
`hit_count`	`uint32_t`	Number of times entry was served

Cache Engine Interface Methods:

Method	Parameters	Returns	Description
`cache_engine_create`	`size_t max_size, int default_ttl`	`CacheEngine*`	Initialize cache with size and TTL limits
`cache_engine_lookup`	`CacheEngine*, HttpRequest*`	`CacheEntry*`	Find cached response for request
`cache_engine_store`	`CacheEngine*, char*, HttpResponse*`	`bool`	Store response in cache if cacheable
`cache_engine_invalidate`	`CacheEngine*, char*`	`bool`	Remove specific entry from cache
`cache_engine_clear`	`CacheEngine*`	`void`	Remove all cached entries
`cache_engine_stats`	`CacheEngine*`	`CacheStats*`	Retrieve hit/miss/eviction statistics
`cache_generate_key`	`HttpRequest*`	`char*`	Generate cache key for request
`cache_is_cacheable`	`HttpResponse*`	`bool`	Determine if response can be cached
`cache_is_fresh`	`CacheEntry*, time_t`	`bool`	Check if cached entry is still fresh
`cache_create_conditional`	`HttpRequest*, CacheEntry*`	`HttpRequest*`	Create conditional request for validation
`cache_update_from_304`	`CacheEntry*, HttpResponse*`	`void`	Update entry from 304 Not Modified

Common Pitfalls

⚠️ Pitfall: Caching Non-Cacheable Responses

Beginning developers often cache every response to maximize hit rates, ignoring HTTP semantics that prohibit caching certain responses. Caching responses with `Cache-Control: no-store`, responses to authenticated requests, or responses with `Set-Cookie` headers can cause security vulnerabilities and incorrect behavior.

This occurs because the caching logic focuses on performance rather than correctness. The cache engine must evaluate cache-control directives, authentication headers, and response characteristics before making storage decisions.

To avoid this pitfall, implement a comprehensive `cache_is_cacheable` function that checks all HTTP caching restrictions: verify no `no-store` directive, ensure no authorization headers in the request unless `public` is specified, check that the response status code is cacheable (200, 203, 300, 301, 410, etc.), and confirm no `Set-Cookie` headers unless specifically allowed.

⚠️ Pitfall: Ignoring Vary Headers in Cache Keys

Many implementations generate cache keys using only the request URI, ignoring the response `Vary` header that indicates which request headers affect the response. This causes the cache to serve incorrect responses to clients with different capabilities or preferences.

For example, a server might return gzipped content for clients supporting compression and uncompressed content for others, using `Vary: Accept-Encoding`. Without incorporating the `Accept-Encoding` header into the cache key, a client that doesn't support compression might receive a cached gzipped response, causing display errors.

The solution requires parsing the `Vary` header from cached responses and incorporating the corresponding request header values into cache key generation. Store the complete `Vary` header list with each cache entry and use it to generate keys for

subsequent requests.

⚠️ Pitfall: Serving Stale Content Beyond max-age

Cache implementations sometimes serve expired content when backend servers are unavailable, violating HTTP semantics unless the response included `Cache-Control: stale-while-revalidate` or similar directives.

This happens when error handling logic prioritizes availability over correctness, serving any cached content during backend failures. While this might seem helpful, it can serve dangerously stale content (hours or days old) that no longer represents accurate information.

Implement proper TTL checking that never serves content beyond its `max-age` unless the response explicitly allows stale serving. Use conditional requests to validate stale content when possible, and return appropriate 502/503 errors when fresh content cannot be obtained.

⚠️ Pitfall: Memory Leaks from Unbounded Cache Growth

Without proper eviction mechanisms, cache implementations can consume unlimited memory, eventually causing out-of-memory crashes in high-traffic scenarios.

This occurs when developers implement cache storage without size limits or eviction policies, assuming that cache hit rates will naturally limit memory usage. In reality, diverse request patterns can create large numbers of unique cache entries that collectively exceed available memory.

Implement comprehensive size tracking that includes both response body size and metadata overhead. Use LRU eviction triggered by configurable memory limits, and consider implementing maximum entry size limits to prevent individual large responses from dominating cache space.

⚠️ Pitfall: Race Conditions in Concurrent Cache Access

Multi-threaded environments require careful synchronization of cache data structures, but many implementations use insufficient locking that leads to corruption, crashes, or incorrect behavior.

The most common issue is using a single mutex for all cache operations, creating unnecessary contention between read operations (cache lookups) that could proceed concurrently. Another issue is inconsistent locking between cache lookup and LRU list updates, potentially corrupting the access ordering.

Use read-write locks that allow concurrent reads while serializing writes. Implement separate locks for cache table operations and LRU list manipulation, but ensure consistent lock ordering to prevent deadlocks. Consider atomic operations for simple counter updates like hit/miss statistics.

Implementation Guidance

The cache engine implementation requires careful attention to memory management, concurrency control, and HTTP semantics compliance. This guidance provides working infrastructure code and detailed implementation skeletons for the core caching logic.

Technology Recommendations:

Component	Simple Option	Advanced Option
Hash Table	Simple chaining with linked lists	Robin Hood hashing with open addressing
Memory Management	malloc/free with manual tracking	Memory pools with fixed-size allocations
Threading	pthread with read-write locks	Lock-free data structures with hazard pointers
Time Handling	time() system calls	High-resolution timers with cached current time
HTTP Parsing	String manipulation with strstr	Dedicated HTTP header parsing library

Recommended File Structure:

```
src/
  cache/
    cache_engine.h          ← Main cache engine interface
    cache_engine.c          ← Core cache logic implementation
    cache_entry.h           ← Cache entry data structures
    cache_entry.c           ← Entry lifecycle management
    cache_key.h             ← Cache key generation
    cache_key.c             ← Key generation algorithms
    cache_control.h         ← HTTP cache-control parsing
    cache_control.c         ← Cache directive evaluation
    lru_list.h              ← LRU eviction list
    lru_list.c              ← LRU list operations
    ttl_heap.h              ← TTL expiration heap
    ttl_heap.c              ← Heap-based expiration tracking
    cache_stats.h           ← Cache metrics and statistics
    cache_stats.c           ← Statistics collection
  tests/
    test_cache_engine.c     ← Cache engine unit tests
    test_cache_key.c        ← Key generation tests
    test_cache_control.c    ← HTTP directive tests
```

Infrastructure Starter Code:

C

```c
// cache_control.h - Complete HTTP Cache-Control parsing
#ifndef CACHE_CONTROL_H
#define CACHE_CONTROL_H

#include <stdbool.h>
#include <time.h>

typedef struct {
    bool no_store;
    bool no_cache;
    bool must_revalidate;
    bool private;
    bool public;
    int max_age;           // -1 if not specified
    int s_maxage;          // -1 if not specified
    bool has_etag;
    bool has_last_modified;
} CacheControl;

// Parse Cache-Control header into structured directives
CacheControl* cache_control_parse(const char* header_value);

// Check if response is cacheable based on cache-control directives
bool cache_control_is_cacheable(const CacheControl* cc, bool has_authorization);

// Calculate expiration time from cache-control and response headers
time_t cache_control_calculate_expiry(const CacheControl* cc,
                                      const char* expires_header,
                                      time_t response_time);

// Check if cached entry is fresh at given time
bool cache_control_is_fresh(const CacheControl* cc, time_t cached_time,
                            time_t current_time);

void cache_control_destroy(CacheControl* cc);
```

```c
#endif

// cache_control.c - Implementation
#include "cache_control.h"
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

CacheControl* cache_control_parse(const char* header_value) {
    if (!header_value) return NULL;

    CacheControl* cc = calloc(1, sizeof(CacheControl));
    cc->max_age = -1;
    cc->s_maxage = -1;

    char* header_copy = strdup(header_value);
    char* token = strtok(header_copy, ",");

    while (token) {
        // Skip whitespace
        while (isspace(*token)) token++;

        if (strncmp(token, "no-store", 8) == 0) {
            cc->no_store = true;
        } else if (strncmp(token, "no-cache", 8) == 0) {
            cc->no_cache = true;
        } else if (strncmp(token, "must-revalidate", 15) == 0) {
            cc->must_revalidate = true;
        } else if (strncmp(token, "private", 7) == 0) {
            cc->private = true;
        } else if (strncmp(token, "public", 6) == 0) {
            cc->public = true;
        } else if (strncmp(token, "max-age=", 8) == 0) {
```

```
            cc->max_age = atoi(token + 8);
        } else if (strncmp(token, "s-maxage=", 9) == 0) {
            cc->s_maxage = atoi(token + 9);
        }

        token = strtok(NULL, ",");
    }

    free(header_copy);
    return cc;
}

bool cache_control_is_cacheable(const CacheControl* cc, bool has_authorization) {
    // Never cache if no-store directive present
    if (cc->no_store) return false;

    // Don't cache authorized requests unless explicitly public
    if (has_authorization && !cc->public) return false;

    // Don't cache private responses in shared cache (reverse proxy)
    if (cc->private) return false;

    return true;
}

time_t cache_control_calculate_expiry(const CacheControl* cc,
                                      const char* expires_header,
                                      time_t response_time) {
    // Use s-maxage for shared caches if present
    if (cc->s_maxage >= 0) {
        return response_time + cc->s_maxage;
    }
```

```
    // Use max-age if present
    if (cc->max_age >= 0) {
        return response_time + cc->max_age;
    }

    // Fall back to Expires header parsing
    if (expires_header) {
        // Parse HTTP date format - simplified for example
        struct tm tm = {0};
        if (strptime(expires_header, "%a, %d %b %Y %H:%M:%S GMT", &tm)) {
            return mktime(&tm);
        }
    }

    // No expiration information
    return 0;
}

bool cache_control_is_fresh(const CacheControl* cc, time_t cached_time,
                            time_t current_time) {
    int max_age = cc->s_maxage >= 0 ? cc->s_maxage : cc->max_age;
    if (max_age < 0) return false; // No expiration specified

    return (current_time - cached_time) <= max_age;
}

void cache_control_destroy(CacheControl* cc) {
    if (cc) free(cc);
}
```

```c
// lru_list.h - Complete LRU list implementation
#ifndef LRU_LIST_H
#define LRU_LIST_H

#include "cache_entry.h"
#include <pthread.h>

typedef struct LRUList {
    CacheEntry* head;
    CacheEntry* tail;
    size_t count;
    pthread_mutex_t lru_mutex;
} LRUList;

LRUList* lru_list_create(void);
void lru_list_move_to_head(LRUList* list, CacheEntry* entry);
void lru_list_add_to_head(LRUList* list, CacheEntry* entry);
CacheEntry* lru_list_remove_tail(LRUList* list);
void lru_list_remove_entry(LRUList* list, CacheEntry* entry);
void lru_list_destroy(LRUList* list);

#endif

// lru_list.c
#include "lru_list.h"
#include <stdlib.h>

LRUList* lru_list_create(void) {
    LRUList* list = calloc(1, sizeof(LRUList));
    pthread_mutex_init(&list->lru_mutex, NULL);
    return list;
}

void lru_list_move_to_head(LRUList* list, CacheEntry* entry) {
    pthread_mutex_lock(&list->lru_mutex);
```

```
    // Remove from current position
    if (entry->lru_prev) entry->lru_prev->lru_next = entry->lru_next;
    if (entry->lru_next) entry->lru_next->lru_prev = entry->lru_prev;
    if (list->tail == entry) list->tail = entry->lru_prev;

    // Add to head
    entry->lru_prev = NULL;
    entry->lru_next = list->head;
    if (list->head) list->head->lru_prev = entry;
    list->head = entry;
    if (!list->tail) list->tail = entry;

    pthread_mutex_unlock(&list->lru_mutex);
}

void lru_list_add_to_head(LRUList* list, CacheEntry* entry) {
    pthread_mutex_lock(&list->lru_mutex);

    entry->lru_prev = NULL;
    entry->lru_next = list->head;
    if (list->head) list->head->lru_prev = entry;
    list->head = entry;
    if (!list->tail) list->tail = entry;
    list->count++;

    pthread_mutex_unlock(&list->lru_mutex);
}

CacheEntry* lru_list_remove_tail(LRUList* list) {
    pthread_mutex_lock(&list->lru_mutex);

    CacheEntry* entry = list->tail;
```

```
    if (entry) {
        list->tail = entry->lru_prev;
        if (list->tail) list->tail->lru_next = NULL;
        else list->head = NULL;

        entry->lru_prev = entry->lru_next = NULL;
        list->count--;
    }

    pthread_mutex_unlock(&list->lru_mutex);
    return entry;
}

void lru_list_remove_entry(LRUList* list, CacheEntry* entry) {
    pthread_mutex_lock(&list->lru_mutex);

    if (entry->lru_prev) entry->lru_prev->lru_next = entry->lru_next;
    if (entry->lru_next) entry->lru_next->lru_prev = entry->lru_prev;
    if (list->head == entry) list->head = entry->lru_next;
    if (list->tail == entry) list->tail = entry->lru_prev;

    entry->lru_prev = entry->lru_next = NULL;
    list->count--;

    pthread_mutex_unlock(&list->lru_mutex);
}

void lru_list_destroy(LRUList* list) {
    if (list) {
        pthread_mutex_destroy(&list->lru_mutex);
        free(list);
    }
```

```
}
```

Core Cache Engine Implementation Skeleton:

```c
// cache_engine.c - Core cache logic with detailed TODOs

#include "cache_engine.h"

#include "cache_control.h"

#include "lru_list.h"

#include <stdlib.h>

#include <string.h>

#include <time.h>


CacheEngine* cache_engine_create(size_t max_size, int default_ttl) {

    // TODO 1: Allocate and initialize CacheEngine structure

    // TODO 2: Create hash table with reasonable initial size (e.g., 1024 buckets)

    // TODO 3: Initialize LRU list for eviction tracking

    // TODO 4: Initialize read-write lock for concurrent access

    // TODO 5: Set size limits and TTL defaults from parameters

    // TODO 6: Initialize statistics counters (hit_count, miss_count, etc.)

    // TODO 7: Start background cleanup thread for TTL expiration

    // Hint: Use hashtable_create(1024) for initial table size

    // Hint: pthread_rwlock_init for read-write lock initialization

}


CacheEntry* cache_engine_lookup(CacheEngine* engine, HttpRequest* request) {

    // TODO 1: Generate cache key from request (method, URI, headers)

    // TODO 2: Acquire read lock on cache for thread-safe lookup

    // TODO 3: Search hash table using generated cache key

    // TODO 4: If entry found, check if it's still fresh (TTL validation)

    // TODO 5: If fresh, update access time and move to LRU head

    // TODO 6: If stale, consider conditional request creation

    // TODO 7: Increment hit or miss counters appropriately

    // TODO 8: Release read lock and return entry (or NULL for miss)

    // Hint: Use cache_generate_key(request) for key generation

    // Hint: Use cache_control_is_fresh() for TTL checking

}
```

```
bool cache_engine_store(CacheEngine* engine, char* key, HttpResponse* response) {
    // TODO 1: Parse response Cache-Control headers for cachability
    // TODO 2: Check if response is cacheable (no no-store, appropriate status)
    // TODO 3: Calculate response size including headers and body
    // TODO 4: Check if response size exceeds individual entry limit
    // TODO 5: Acquire write lock for cache modification
    // TODO 6: Check if storing would exceed total cache size limit
    // TODO 7: Evict LRU entries until sufficient space available
    // TODO 8: Create new CacheEntry with response data and metadata
    // TODO 9: Insert entry into hash table and add to LRU head
    // TODO 10: Update current cache size and storage statistics
    // TODO 11: Release write lock and return success status
    // Hint: Use cache_control_parse() for Cache-Control evaluation
    // Hint: Use lru_list_remove_tail() for eviction when space needed
}

char* cache_generate_key(HttpRequest* request) {
    // TODO 1: Start with HTTP method and normalized URI
    // TODO 2: Check for cached response Vary header (if doing validation)
    // TODO 3: For each header in Vary, append header name and value
    // TODO 4: Include Host header for virtual hosting support
    // TODO 5: Ensure consistent ordering and normalization
    // TODO 6: Allocate and return null-terminated key string
    // Hint: Use fixed-size buffer (512 bytes) for key construction
    // Hint: Normalize header names to lowercase for consistency
}

bool cache_is_cacheable(HttpResponse* response) {
    // TODO 1: Parse Cache-Control header from response
    // TODO 2: Check for no-store directive (never cache)
    // TODO 3: Check for private directive (not for shared caches)
    // TODO 4: Verify response status code is cacheable (200, 301, etc.)
    // TODO 5: Check for Set-Cookie headers (usually not cacheable)
```

```
    // TODO 6: Return true only if all caching requirements met
    // Hint: Use cache_control_parse() for header evaluation
    // Hint: Status codes 200, 203, 300, 301, 410 are typically cacheable
}

void* cache_cleanup_thread(void* arg) {
    // TODO 1: Cast argument to CacheEngine pointer
    // TODO 2: Loop while engine->running flag is true
    // TODO 3: Sleep for cleanup_interval seconds
    // TODO 4: Acquire write lock for cache modification
    // TODO 5: Scan cache entries for expired TTL values
    // TODO 6: Remove expired entries from hash table and LRU list
    // TODO 7: Update cache size and eviction statistics
    // TODO 8: Release write lock and continue loop
    // Hint: Use time(NULL) for current time comparison
    // Hint: Batch multiple removals in single lock acquisition
}
```

Milestone Checkpoints:

After implementing the cache engine core, verify functionality with these checkpoints:

1. **Cache Key Generation**: Create requests with different URIs, methods, and headers. Verify that identical requests generate identical keys and different requests generate different keys.
2. **Basic Caching**: Send identical GET requests through the proxy. First request should be forwarded to backend, subsequent requests should be served from cache (verify with backend access logs).
3. **Cache-Control Respect**: Send requests for resources with different cache-control headers (`no-store`, `max-age=60`, etc.). Verify that uncacheable responses are not cached and cached responses respect TTL limits.
4. **Vary Header Handling**: Configure backend to return `Vary: Accept-Encoding` and send requests with different Accept-Encoding headers. Verify that different encodings are cached separately.
5. **LRU Eviction**: Configure small cache size and send requests for more content than cache can hold. Verify that least recently used entries are evicted first.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Cache never hits	Key generation inconsistency	Log generated keys for identical requests	Ensure header normalization and consistent ordering
Memory usage grows unbounded	Missing eviction or TTL cleanup	Check LRU list size and cleanup thread	Verify size limit enforcement and cleanup thread operation
Stale content served	TTL calculation error	Log expiry times vs current time	Fix cache_control_calculate_expiry implementation
Wrong content for client	Missing Vary support	Check if response has Vary header	Include Vary headers in key generation
Cache corruption/crashes	Race condition in concurrent access	Run with thread sanitizer	Fix lock ordering and ensure consistent locking

SSL Termination Component

> **Milestone(s):** Milestone 5 (SSL Termination) - handles HTTPS connections by terminating TLS at the proxy and forwarding decrypted HTTP to backends

Think of SSL termination like a high-security checkpoint at a government building. Visitors arrive with various forms of encrypted identification (SSL certificates), guards at the checkpoint verify their credentials and decrypt their intentions (TLS handshake), then escort them inside using simple internal protocols (plain HTTP to backends). The guards must handle multiple types of credentials (SNI for different domains) and maintain strict security protocols while ensuring smooth traffic flow.

The `SSLTermination` component sits at the network edge, accepting encrypted TLS connections from clients and converting them into plain HTTP connections to backend servers. This design choice centralizes certificate management, reduces computational load on backend servers, and enables the proxy to inspect and route HTTP traffic effectively. However, it also creates a critical security boundary that must be implemented with extreme care to prevent vulnerabilities.

TLS Context Management

The foundation of SSL termination lies in proper TLS context management, which involves setting up cryptographic contexts, loading certificates and private keys, and configuring security parameters. Think of a TLS context as a cryptographic blueprint that defines how the proxy will handle encrypted connections - it specifies which certificates to present, which cipher suites to accept, and how to validate client connections.

The `SSLTermination` component maintains multiple TLS contexts to support different domains and security requirements. Each context represents a complete cryptographic configuration for a specific domain or set of domains. The component must handle context creation during startup, dynamic certificate reloading for renewals, and context selection based on incoming connection characteristics.

Field	Type	Description
contexts	`HashTable*`	Maps domain names to SSL_CTX structures
default_context	`SSL_CTX*`	Fallback context for unmatched SNI requests
cert_store	`X509_STORE*`	Certificate authority store for validation
cipher_list	`char[1024]`	Configured cipher suite preference string
min_tls_version	`int`	Minimum TLS version (TLS_1_2 or TLS_1_3)
max_tls_version	`int`	Maximum TLS version for compatibility
session_cache	`SSL_SESSION_CACHE*`	Session resumption cache for performance
context_mutex	`pthread_rwlock_t`	Synchronizes context access and updates
cert_reload_thread	`pthread_t`	Background thread monitoring certificate changes
reload_interval	`int`	Certificate file monitoring interval in seconds
running	`bool`	Controls background certificate monitoring

The certificate loading process involves several critical security validations. The component must verify that certificate files are readable, private keys match their corresponding certificates, certificate chains are complete and valid, and file permissions restrict access appropriately. Certificate validation extends beyond basic file parsing to include expiration date checking, key usage validation, and certificate chain verification against trusted authorities.

Decision: OpenSSL vs BoringSSL vs Custom TLS Implementation

- **Context**: Need TLS implementation for production reverse proxy handling potentially thousands of concurrent connections
- **Options Considered**: OpenSSL (mature, feature-complete), BoringSSL (Google's fork focused on security), Custom implementation (full control)
- **Decision**: OpenSSL with careful version management and security patches
- **Rationale**: OpenSSL provides comprehensive TLS support, extensive documentation, and broad platform compatibility. While BoringSSL offers enhanced security, OpenSSL's maturity and ecosystem support outweigh the benefits for this implementation
- **Consequences**: Enables full TLS feature support and easy certificate management but requires staying current with security patches and careful API usage to avoid common pitfalls

Option	Pros	Cons	Chosen?
OpenSSL	Comprehensive features, extensive docs, broad compatibility	Large attack surface, complex API, frequent security updates	✓ Yes
BoringSSL	Enhanced security focus, simplified API, Google backing	Limited documentation, fewer features, potential compatibility issues	No
Custom TLS	Complete control, minimal dependencies, tailored security	Enormous development effort, high risk of security bugs, maintenance burden	No

The context initialization process follows a carefully orchestrated sequence to ensure security and reliability:

1. **Library Initialization**: Initialize OpenSSL library components including random number generators, error strings, and algorithm tables
2. **Context Creation**: Create SSL_CTX structures for each configured domain using appropriate TLS methods (TLS_server_method for modern compatibility)
3. **Certificate Loading**: Load X.509 certificates from PEM files, validating format and extracting subject information
4. **Private Key Loading**: Load corresponding private keys, ensuring they match their certificates through cryptographic validation
5. **Certificate Chain Verification**: Validate complete certificate chains from leaf certificates to trusted root authorities
6. **Cipher Suite Configuration**: Configure allowed cipher suites prioritizing modern, secure algorithms like AES-GCM and ChaCha20-Poly1305
7. **Protocol Version Limits**: Set minimum and maximum TLS versions, typically requiring TLS 1.2 or higher for security
8. **Session Cache Setup**: Configure session resumption cache to improve performance for returning clients
9. **SNI Callback Registration**: Register Server Name Indication callback for dynamic certificate selection
10. **Security Parameter Validation**: Verify all security parameters meet organizational security policies

Certificate reloading presents unique challenges in a production environment where the proxy cannot afford downtime. The component implements a sophisticated certificate monitoring system that detects file system changes and performs atomic context updates.

⚠️ **Pitfall: Certificate and Private Key Mismatch** Loading certificates and private keys from separate files can result in cryptographic mismatches that only surface during TLS handshakes. The component must verify that each private key corresponds to its certificate by performing a cryptographic validation immediately after loading. Failing to validate this relationship results in TLS handshake failures that are difficult to diagnose in production environments.

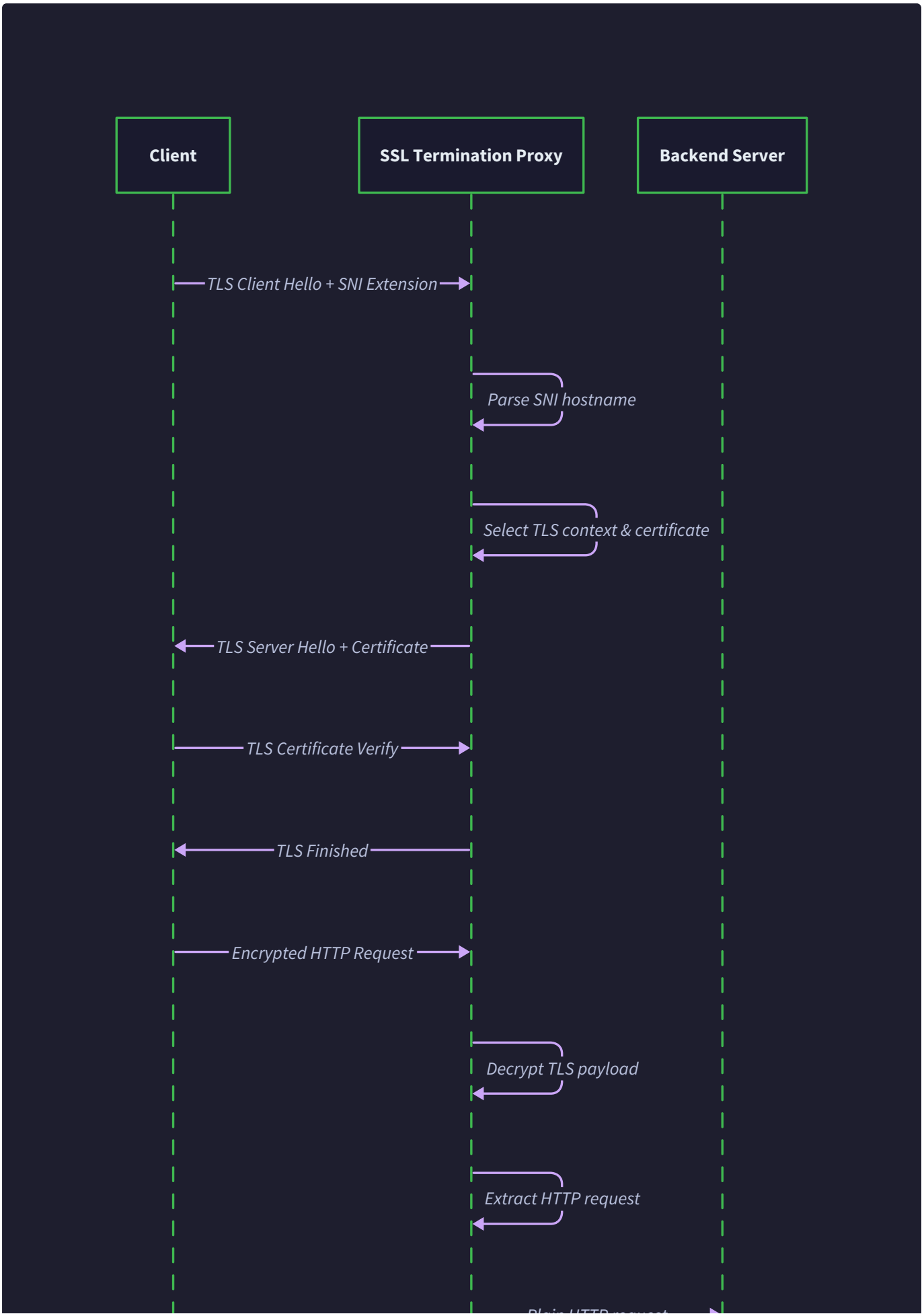
The certificate monitoring system operates through a dedicated background thread that periodically checks certificate file modification times and validates new certificates before performing atomic swaps. This approach ensures that certificate renewals (common with automated systems like Let's Encrypt) don't require proxy restarts or service interruptions.

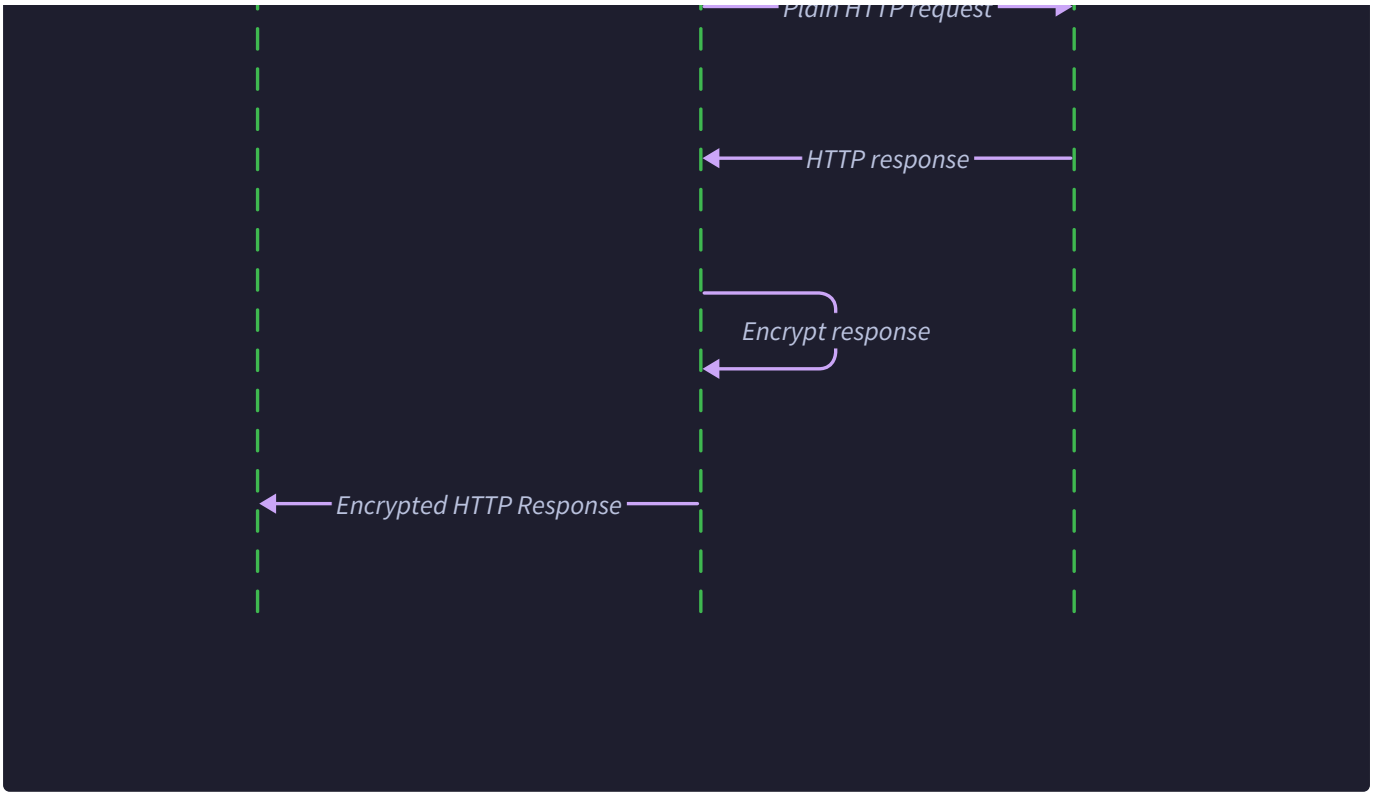
Certificate storage security requires careful attention to file system permissions and memory management. Private keys must be stored in files readable only by the proxy process user, and private key material in memory should be cleared when contexts are destroyed. The component implements secure memory allocation for key material and ensures that sensitive data doesn't persist in process memory dumps or swap files.

Server Name Indication

Server Name Indication (SNI) enables a single proxy instance to serve multiple domains with different SSL certificates, much like how a multilingual receptionist can greet visitors in their preferred language. When clients initiate TLS connections, they include the hostname they're trying to reach in the SNI extension, allowing the proxy to select the appropriate certificate for that specific domain.

The SNI implementation centers around a callback mechanism that executes during the TLS handshake process. When a client sends a TLS ClientHello message containing an SNI extension, OpenSSL invokes the proxy's SNI callback function, providing the requested hostname. The proxy must then locate the appropriate TLS context for that hostname and switch the connection to use the correct certificate and configuration.


Client
SSL Termination Proxy
Backend Server
TLS Client Hello + SNI Extension
Parse SNI hostname
Select TLS context & certificate
TLS Server Hello + Certificate
TLS Certificate Verify
TLS Finished
Encrypted HTTP Request
Decrypt TLS payload
Extract HTTP request



The SNI selection algorithm handles several complex scenarios including exact hostname matches, wildcard certificate matching, and fallback behavior for unrecognized domains. Wildcard certificates (e.g., *.example.com) require careful pattern matching logic that understands DNS wildcard semantics while maintaining security boundaries.

Current State	Client Action	Proxy Response	Next State
`TLS_HANDSHAKE_INIT`	ClientHello with SNI	Select appropriate certificate context	`TLS_CONTEXT_SELECTED`
`TLS_CONTEXT_SELECTED`	Continue handshake	Send ServerHello with selected certificate	`TLS_CERTIFICATE_SENT`
`TLS_CERTIFICATE_SENT`	Certificate validation	Process client certificate if required	`TLS_KEY_EXCHANGE`
`TLS_KEY_EXCHANGE`	Key exchange completion	Generate session keys	`TLS_HANDSHAKE_COMPLETE`
`TLS_HANDSHAKE_COMPLETE`	Application data	Decrypt and forward to backend	`TLS_APPLICATION_DATA`

The hostname matching logic must handle various edge cases that occur in real-world deployments. International domain names require proper Unicode normalization and IDNA encoding. Case sensitivity variations in hostnames need consistent handling. Port numbers included in Host headers must be stripped when matching against certificate subject names.

Decision: Wildcard Certificate Matching Strategy

- **Context**: Need to support wildcard certificates (*.example.com) while maintaining security boundaries and preventing certificate misuse
- **Options Considered**: Simple string matching, regex-based matching, DNS-compliant wildcard matching
- **Decision**: DNS-compliant wildcard matching following RFC 6125 guidelines
- **Rationale**: DNS wildcards have specific semantic rules that must be followed for security. Simple string matching can create security vulnerabilities, while regex matching is overkill and potentially slower
- **Consequences**: Enables proper wildcard certificate support with strong security guarantees but requires implementing RFC-compliant matching logic

The SNI callback implementation requires thread-safe access to the certificate context map since multiple TLS handshakes may occur simultaneously. The component uses read-write locks to allow concurrent access to the context map while serializing updates during certificate reloads.

Certificate selection follows a priority order designed to provide the most specific match for each request:

1. **Exact Hostname Match**: Direct lookup in the context map for the exact requested hostname
2. **Wildcard Certificate Match**: Check for wildcard certificates that cover the requested domain following DNS wildcard rules
3. **Subject Alternative Name Match**: Examine SAN extensions in certificates for additional hostname matches
4. **Default Context Fallback**: Use the default certificate context if no specific match is found
5. **Connection Termination**: Optionally terminate connections that don't match any configured certificate

The wildcard matching algorithm implements RFC 6125 semantics, which specify that wildcards only match a single DNS label and cannot span multiple levels. For example, *.example.com matches api.example.com but not sub.api.example.com. This restriction prevents overly broad certificate matching that could create security vulnerabilities.

SNI-based certificate selection enables advanced deployment scenarios such as hosting multiple customer domains on a single proxy instance, gradual certificate migrations, and A/B testing of different TLS configurations. However, it also introduces complexity in certificate management and monitoring since each domain requires separate certificate lifecycle management.

⚠️ **Pitfall: SNI Extension Missing from Client** Older clients or certain automated tools may not include SNI extensions in their TLS handshakes, causing certificate selection to fail or fall back to default certificates. The proxy must handle these cases gracefully by providing a sensible default certificate that can serve the most common domain or by terminating connections with clear error messages that guide clients toward proper SNI support.

Certificate caching and context pooling optimize performance for high-throughput scenarios where thousands of TLS handshakes occur simultaneously. The component maintains a cache of recently used certificate contexts and pre-computes expensive cryptographic operations where possible.

SSL Termination Decisions

The SSL termination component faces numerous architectural decisions that significantly impact security, performance, and operational complexity. These decisions involve trade-offs between security posture, computational efficiency, memory usage, and implementation complexity.

Decision: TLS Version Support Policy

- **Context**: Need to balance security with client compatibility, considering that older TLS versions have known vulnerabilities while newer versions aren't universally supported
- **Options Considered**: TLS 1.0+ (maximum compatibility), TLS 1.2+ (security focused), TLS 1.3 only (cutting edge)
- **Decision**: TLS 1.2 minimum with TLS 1.3 preferred
- **Rationale**: TLS 1.2 provides strong security while maintaining broad client compatibility. TLS 1.3 offers performance and security improvements where supported. TLS 1.0/1.1 have known vulnerabilities and should be deprecated
- **Consequences**: Provides strong security posture while maintaining compatibility with 99%+ of modern clients but may reject very old clients that only support deprecated TLS versions

TLS Version	Security Level	Client Support	Performance	Chosen?
TLS 1.0/1.1	Weak (deprecated)	Universal	Poor	No
TLS 1.2+	Strong	99%+ modern clients	Good	✓ Yes
TLS 1.3 Only	Excellent	80%+ clients	Excellent	No

Cipher suite selection represents another critical security decision that affects both protection strength and computational performance. Modern cipher suites like AES-GCM and ChaCha20-Poly1305 provide authenticated encryption with excellent performance characteristics, while older suites like RC4 and DES have known vulnerabilities.

Decision: Cipher Suite Priority and Selection

- **Context**: Must choose which encryption algorithms to support, considering security strength, performance characteristics, and client compatibility
- **Options Considered**: Server preference ordering, client preference ordering, security-only suites
- **Decision**: Server preference with modern AEAD cipher priority
- **Rationale**: Server preference allows the proxy to enforce security policy while modern AEAD ciphers provide both security and performance. ChaCha20-Poly1305 offers excellent performance on systems without AES-NI hardware acceleration
- **Consequences**: Ensures strong encryption with optimal performance but may need periodic updates as cryptographic recommendations evolve

The preferred cipher suite ordering prioritizes authenticated encryption with associated data (AEAD) ciphers that provide both confidentiality and integrity protection:

1. **TLS_AES_256_GCM_SHA384**: TLS 1.3 with AES-256 in Galois/Counter Mode
2. **TLS_CHACHA20_POLY1305_SHA256**: TLS 1.3 with ChaCha20-Poly1305 for ARM/mobile optimization
3. **TLS_AES_128_GCM_SHA256**: TLS 1.3 with AES-128 for performance-sensitive scenarios
4. **ECDHE-RSA-AES256-GCM-SHA384**: TLS 1.2 with perfect forward secrecy
5. **ECDHE-RSA-CHACHA20-POLY1305**: TLS 1.2 ChaCha20 variant for non-AES hardware

Session resumption configuration balances security and performance by allowing clients to reuse cryptographic session state across multiple connections. This optimization significantly reduces CPU usage and connection establishment latency, particularly important for mobile clients and high-frequency API access patterns.

Decision: Session Resumption and Ticket Rotation

- **Context**: Session resumption improves performance by reusing TLS session state, but session tickets must be rotated regularly for security
- **Options Considered**: No session resumption (security focused), session IDs only, session tickets with rotation
- **Decision**: Session tickets with automatic rotation every 24 hours
- **Rationale**: Session tickets provide better scalability than session IDs while rotation prevents long-term session compromise. 24-hour rotation balances security with operational simplicity
- **Consequences**: Achieves optimal TLS performance with good security properties but requires implementing ticket key rotation and handling rotation edge cases

Certificate validation and chain building require sophisticated logic to handle the complexities of real-world PKI deployments. Certificate chains may include intermediate certificates, cross-signed roots, and alternative validation paths. The component must build complete validation chains while respecting certificate policies and constraints.

The certificate chain validation process involves several security-critical steps:

1. **Certificate Chain Construction**: Build complete chains from leaf certificates to trusted root authorities
2. **Signature Verification**: Validate cryptographic signatures throughout the certificate chain
3. **Validity Period Check**: Ensure all certificates are within their valid time ranges
4. **Revocation Checking**: Optionally check certificate revocation status via CRL or OCSP
5. **Policy Validation**: Verify certificate policies and key usage constraints
6. **Hostname Validation**: Confirm certificate subject names match the requested hostname

Memory management for SSL contexts and session data requires careful attention to prevent both memory leaks and security vulnerabilities. TLS session data contains sensitive cryptographic material that must be cleared when sessions end. Context switching during SNI selection must preserve memory isolation between different domains.

⚠️ **Pitfall: Inadequate Certificate Chain Validation** Many SSL implementations perform minimal certificate validation, checking only basic cryptographic signatures while ignoring critical security constraints like key usage, certificate policies, and validity periods. Production reverse proxies must implement comprehensive validation that matches browser security standards to prevent accepting compromised or misused certificates.

Performance optimization in SSL termination focuses on reducing per-connection computational overhead while maintaining security properties. Hardware acceleration through AES-NI instructions, optimized cipher suite selection, and connection pooling all contribute to scalable TLS performance.

The SSL termination component integrates with the broader proxy architecture through well-defined interfaces that isolate TLS complexity from other components. Once TLS connections are established and decrypted, they appear as standard HTTP connections to the HTTP parser and connection manager components.

Error handling in SSL termination must distinguish between client errors (invalid certificates, unsupported protocols), server configuration errors (missing certificates, expired keys), and network errors (connection timeouts, malformed packets). Each error category requires different logging levels and recovery strategies.

Common Pitfalls

⚠️ **Pitfall: Private Key Exposure in Memory Dumps** Private key material loaded into process memory can be exposed through core dumps, swap files, or memory debugging tools. The SSL termination component must use secure memory allocation functions (like `mlock()`) to prevent key material from being written to disk and explicitly clear memory regions containing sensitive data when contexts are destroyed.

⚠️ **Pitfall: Certificate Expiration Without Monitoring** Production deployments frequently experience outages due to expired certificates that weren't renewed in time. The component should implement certificate expiration monitoring that logs warnings well before certificates expire and optionally exposes metrics for external monitoring systems to alert on approaching expiration dates.

⚠️ **Pitfall: Weak Cipher Suite Configuration** Default OpenSSL cipher configurations may include weak or deprecated algorithms for compatibility reasons. Production deployments must explicitly configure cipher suite preferences to exclude vulnerable algorithms like RC4, DES, and export-grade ciphers while prioritizing modern AEAD ciphers.

⚠️ **Pitfall: Improper SNI Fallback Handling** When clients don't provide SNI extensions or request unrecognized hostnames, the proxy must handle these cases gracefully. Falling back to a default certificate that doesn't match the requested hostname can cause certificate validation errors in clients, while terminating connections may impact legitimate traffic.

⚠️ **Pitfall: Certificate Reload Race Conditions** Dynamic certificate reloading during production traffic can create race conditions where some connections use old certificates while others use new ones. The component must implement atomic context switching that ensures all new connections use updated certificates while allowing existing connections to complete with their original contexts.

Implementation Guidance

The SSL termination component requires careful integration with OpenSSL and precise handling of cryptographic material. The implementation balances security requirements with performance needs while maintaining operational simplicity.

Technology Recommendations

Component	Simple Option	Advanced Option
TLS Library	OpenSSL 1.1.1+ with basic configuration	OpenSSL 3.0+ with provider architecture
Certificate Storage	File-based PEM certificates	Hardware Security Module (HSM) integration
Session Management	In-memory session cache	Distributed session store with Redis
Certificate Monitoring	Filesystem polling with inotify	Certificate transparency log monitoring
Performance Optimization	Basic cipher suite selection	Hardware acceleration with AES-NI and AVX

Recommended File Structure

```
proxy/
  src/
    ssl/
      ssl_termination.h        ← SSL component interface
      ssl_termination.c        ← Core SSL termination logic
      ssl_context.h           ← TLS context management
      ssl_context.c           ← Context creation and certificate loading
      ssl_sni.h              ← SNI handling interface
      ssl_sni.c              ← SNI callback and hostname matching
      ssl_session.h          ← Session resumption management
      ssl_session.c          ← Session cache and ticket rotation
      ssl_utils.h            ← SSL utility functions
      ssl_utils.c            ← Certificate validation and crypto helpers
      ssl_config.h           ← SSL configuration structures
      ssl_config.c           ← Configuration parsing and validation
    tests/
      test_ssl_termination.c  ← SSL termination tests
      test_ssl_context.c      ← Context management tests
      test_ssl_sni.c         ← SNI handling tests
      ssl_test_certs/        ← Test certificates and keys
        test_server.crt
        test_server.key
        test_ca.crt
        wildcard_test.crt
        wildcard_test.key
```

Infrastructure Starter Code

```c
// ssl_utils.h - SSL utility functions

#ifndef SSL_UTILS_H

#define SSL_UTILS_H


#include <openssl/ssl.h>

#include <openssl/x509.h>

#include <openssl/x509v3.h>

#include <openssl/err.h>

#include <openssl/pem.h>

#include <time.h>

#include <stdbool.h>


// Certificate validation result structure

typedef struct {

    bool valid;

    char error_message[256];

    time_t expires_at;

    char subject_name[256];

    char issuer_name[256];

    char **san_entries;

    size_t san_count;

} CertificateInfo;


// Initialize OpenSSL library

bool ssl_utils_init(void);


// Clean up OpenSSL library

void ssl_utils_cleanup(void);


// Load certificate from PEM file with validation

X509* ssl_utils_load_certificate(const char* cert_path, CertificateInfo* info);


// Load private key from PEM file

EVP_PKEY* ssl_utils_load_private_key(const char* key_path);
```

```c
// Verify that private key matches certificate
bool ssl_utils_verify_key_cert_match(EVP_PKEY* key, X509* cert);

// Extract hostname from certificate (CN or SAN)
bool ssl_utils_extract_hostnames(X509* cert, char*** hostnames, size_t* count);

// Check if hostname matches certificate (with wildcard support)
bool ssl_utils_hostname_matches_cert(const char* hostname, X509* cert);

// Get certificate expiration time
time_t ssl_utils_get_cert_expiration(X509* cert);

// Check if certificate is expired or expiring soon
bool ssl_utils_cert_expires_soon(X509* cert, int days_threshold);

// Build complete certificate chain
STACK_OF(X509)* ssl_utils_build_cert_chain(X509* cert, const char* chain_path);

// Configure secure cipher suites
bool ssl_utils_set_secure_ciphers(SSL_CTX* ctx);

// Set up TLS version restrictions
bool ssl_utils_set_tls_versions(SSL_CTX* ctx, int min_version, int max_version);

// Free certificate info structure
void ssl_utils_free_cert_info(CertificateInfo* info);

#endif // SSL_UTILS_H
```

```c
// ssl_utils.c - Complete SSL utility implementation

#include "ssl_utils.h"

#include <string.h>

#include <stdlib.h>

#include <stdio.h>


static bool ssl_initialized = false;


bool ssl_utils_init(void) {

    if (ssl_initialized) {

        return true;

    }



    SSL_load_error_strings();

    SSL_library_init();

    OpenSSL_add_all_algorithms();



    ssl_initialized = true;

    return true;

}


void ssl_utils_cleanup(void) {

    if (!ssl_initialized) {

        return;

    }



    EVP_cleanup();

    ERR_free_strings();

    ssl_initialized = false;

}


X509* ssl_utils_load_certificate(const char* cert_path, CertificateInfo* info) {

    FILE* cert_file = fopen(cert_path, "r");

    if (!cert_file) {
```

```
        if (info) {
            snprintf(info->error_message, sizeof(info->error_message),
                    "Cannot open certificate file: %s", cert_path);
            info->valid = false;
        }
        return NULL;
    }

    X509* cert = PEM_read_X509(cert_file, NULL, NULL, NULL);
    fclose(cert_file);

    if (!cert) {
        if (info) {
            snprintf(info->error_message, sizeof(info->error_message),
                    "Cannot parse certificate file: %s", cert_path);
            info->valid = false;
        }
        return NULL;
    }

    if (info) {
        info->valid = true;
        info->expires_at = ssl_utils_get_cert_expiration(cert);

        // Extract subject name
        X509_NAME* subject = X509_get_subject_name(cert);
        X509_NAME_oneline(subject, info->subject_name, sizeof(info->subject_name));

        // Extract issuer name
        X509_NAME* issuer = X509_get_issuer_name(cert);
        X509_NAME_oneline(issuer, info->issuer_name, sizeof(info->issuer_name));
```

```
        // Extract SAN entries
        ssl_utils_extract_hostnames(cert, &info->san_entries, &info->san_count);
    }

    return cert;
}

EVP_PKEY* ssl_utils_load_private_key(const char* key_path) {
    FILE* key_file = fopen(key_path, "r");
    if (!key_file) {
        return NULL;
    }

    EVP_PKEY* key = PEM_read_PrivateKey(key_file, NULL, NULL, NULL);
    fclose(key_file);

    return key;
}

bool ssl_utils_verify_key_cert_match(EVP_PKEY* key, X509* cert) {
    if (!key || !cert) {
        return false;
    }

    EVP_PKEY* cert_key = X509_get_pubkey(cert);
    if (!cert_key) {
        return false;
    }

    int result = EVP_PKEY_cmp(key, cert_key);
    EVP_PKEY_free(cert_key);

    return result == 1;
```

```
}

bool ssl_utils_extract_hostnames(X509* cert, char*** hostnames, size_t* count) {
    STACK_OF(GENERAL_NAME)* san_names = NULL;
    san_names = X509_get_ext_d2i(cert, NID_subject_alt_name, NULL, NULL);

    if (!san_names) {
        *hostnames = NULL;
        *count = 0;
        return true; // Not an error, just no SAN extension
    }

    int san_count = sk_GENERAL_NAME_num(san_names);
    *hostnames = malloc(san_count * sizeof(char*));
    *count = 0;

    for (int i = 0; i < san_count; i++) {
        GENERAL_NAME* name = sk_GENERAL_NAME_value(san_names, i);
        if (name->type == GEN_DNS) {
            char* hostname = (char*)ASN1_STRING_get0_data(name->d.dNSName);
            (*hostnames)[*count] = strdup(hostname);
            (*count)++;
        }
    }

    sk_GENERAL_NAME_pop_free(san_names, GENERAL_NAME_free);
    return true;
}

time_t ssl_utils_get_cert_expiration(X509* cert) {
    ASN1_TIME* not_after = X509_get_notAfter(cert);
    struct tm tm;
```

```
    if (!ASN1_TIME_to_tm(not_after, &tm)) {
        return 0;
    }

    return mktime(&tm);
}

bool ssl_utils_set_secure_ciphers(SSL_CTX* ctx) {
    const char* cipher_list =
        "ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20:!aNULL:!MD5:!DSS";

    return SSL_CTX_set_cipher_list(ctx, cipher_list) == 1;
}

bool ssl_utils_set_tls_versions(SSL_CTX* ctx, int min_version, int max_version) {
    if (SSL_CTX_set_min_proto_version(ctx, min_version) != 1) {
        return false;
    }

    if (SSL_CTX_set_max_proto_version(ctx, max_version) != 1) {
        return false;
    }

    return true;
}
```

Core Logic Skeleton Code

```c
// ssl_termination.h - SSL Termination component interface

#ifndef SSL_TERMINATION_H

#define SSL_TERMINATION_H


#include "ssl_utils.h"

#include "../config.h"

#include "../hashtable.h"

#include <pthread.h>


typedef struct {

    HashTable* contexts;           // Maps domain names to SSL_CTX*

    SSL_CTX* default_context;     // Fallback context

    char cipher_list[1024];       // Configured cipher suites

    int min_tls_version;          // Minimum TLS version

    int max_tls_version;          // Maximum TLS version

    pthread_rwlock_t context_mutex; // Protects context map

    pthread_t cert_reload_thread; // Certificate monitoring thread

    int reload_interval;          // Reload check interval

    bool running;                 // Component running state

} SSLTermination;


// Create and initialize SSL termination component

SSLTermination* ssl_termination_create(ProxyConfig* config);


// Start SSL termination component (starts background threads)

bool ssl_termination_start(SSLTermination* ssl_term);


// Stop SSL termination component and cleanup resources

void ssl_termination_stop(SSLTermination* ssl_term);


// Destroy SSL termination component

void ssl_termination_destroy(SSLTermination* ssl_term);


// Create SSL connection wrapper for client socket
```

```
SSL* ssl_termination_accept_connection(SSLTermination* ssl_term, int client_fd);

// SNI callback for certificate selection
int ssl_termination_sni_callback(SSL* ssl, int* ad, void* arg);

// Reload certificates from disk (for certificate renewal)
bool ssl_termination_reload_certificates(SSLTermination* ssl_term);

#endif // SSL_TERMINATION_H
```

```c
// ssl_termination.c - Core SSL termination logic

#include "ssl_termination.h"

#include "../logger.h"

#include <unistd.h>

#include <sys/stat.h>


SSLTermination* ssl_termination_create(ProxyConfig* config) {

    // TODO 1: Allocate SSLTermination structure and initialize fields

    // TODO 2: Initialize OpenSSL library using ssl_utils_init()

    // TODO 3: Create hash table for certificate contexts

    // TODO 4: Initialize read-write lock for thread-safe context access

    // TODO 5: Load and validate all configured certificates and private keys

    // TODO 6: Create SSL_CTX for each certificate domain

    // TODO 7: Configure cipher suites and TLS versions for each context

    // TODO 8: Set up default context for unmatched SNI requests

    // TODO 9: Register SNI callback for dynamic certificate selection

    // TODO 10: Validate that all contexts are properly configured

    // Hint: Use config->ssl_cert_path and config->ssl_key_path for certificate loading

    // Hint: Call ssl_utils_verify_key_cert_match() to ensure key/cert pairs match

}


bool ssl_termination_start(SSLTermination* ssl_term) {

    // TODO 1: Set running flag to true

    // TODO 2: Create certificate reload monitoring thread

    // TODO 3: Start background thread with cert_reload_monitor function

    // TODO 4: Return true if all threads started successfully

    // Hint: Use pthread_create() to start the certificate monitoring thread

}


SSL* ssl_termination_accept_connection(SSLTermination* ssl_term, int client_fd) {

    // TODO 1: Create new SSL connection using default context

    // TODO 2: Set file descriptor for SSL connection

    // TODO 3: Configure SSL connection for server mode


```

```
    // TODO 4: Set SNI callback data to point to ssl_term
    // TODO 5: Perform SSL handshake with client
    // TODO 6: Handle handshake errors gracefully
    // TODO 7: Return established SSL connection or NULL on failure
    // Hint: Use SSL_new(), SSL_set_fd(), SSL_set_accept_state(), SSL_accept()
    // Hint: Check SSL_get_error() for detailed error information on failure
}

int ssl_termination_sni_callback(SSL* ssl, int* ad, void* arg) {
    // TODO 1: Cast arg back to SSLTermination pointer
    // TODO 2: Get requested hostname from SSL connection using SSL_get_servername()
    // TODO 3: Acquire read lock on context mutex for thread safety
    // TODO 4: Look up SSL_CTX for requested hostname in contexts hash table
    // TODO 5: If exact match found, set SSL context using SSL_set_SSL_CTX()
    // TODO 6: If no exact match, try wildcard certificate matching
    // TODO 7: If still no match, use default context (already set)
    // TODO 8: Release read lock on context mutex
    // TODO 9: Return SSL_TLSEXT_ERR_OK on success or SSL_TLSEXT_ERR_ALERT_FATAL on error
    // Hint: Use hashtable_get() to look up contexts by hostname
    // Hint: Implement wildcard matching following DNS wildcard rules (*.domain.com)
}

bool ssl_termination_reload_certificates(SSLTermination* ssl_term) {
    // TODO 1: Acquire write lock on context mutex to prevent concurrent access
    // TODO 2: Iterate through all certificate files checking modification times
    // TODO 3: For each modified certificate, load and validate new certificate
    // TODO 4: Create new SSL_CTX with updated certificate and key
    // TODO 5: Replace old context in hash table with new context atomically
    // TODO 6: Free old SSL_CTX after replacement to prevent memory leaks
    // TODO 7: Log certificate reload events for operational visibility
    // TODO 8: Release write lock on context mutex
    // TODO 9: Return true if all reloads successful, false if any failed
    // Hint: Use stat() to check file modification times
```

```
    // Hint: Keep track of last reload time to avoid unnecessary work
}
```

Milestone Checkpoint

After implementing SSL termination, verify functionality with these steps:

Basic SSL Connection Test:

```bash
# Test HTTPS connection with self-signed certificate
openssl s_client -connect localhost:8443 -servername test.example.com
# Should show certificate details and establish connection
```

SNI Functionality Test:

```bash
# Test different hostnames resolve to different certificates
openssl s_client -connect localhost:8443 -servername api.example.com
openssl s_client -connect localhost:8443 -servername web.example.com
# Should show different certificate subjects for different hostnames
```

Expected Behavior:

- HTTPS connections establish successfully with proper certificate presentation
- SNI selects appropriate certificates based on requested hostname
- Proxy decrypts HTTPS traffic and forwards plain HTTP to backends
- Certificate reload works without restarting the proxy process
- Logs show SSL handshake success/failure events with clear error messages

Signs of Problems:

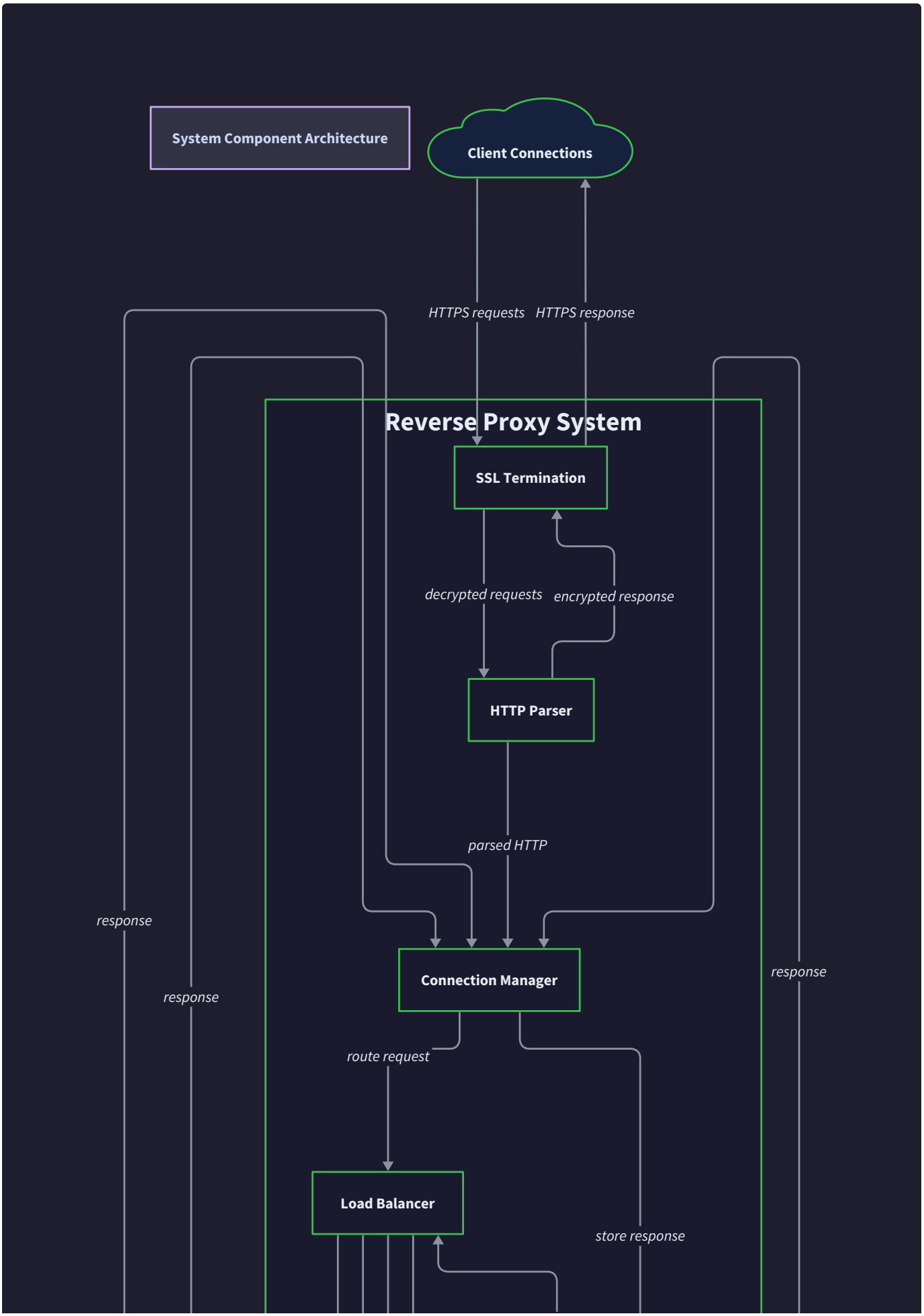
- "SSL handshake failed" errors indicate certificate or configuration issues
- "Certificate verification failed" suggests certificate chain problems
- "SNI callback failed" indicates hostname matching logic errors
- Memory leaks during certificate reloading suggest cleanup issues

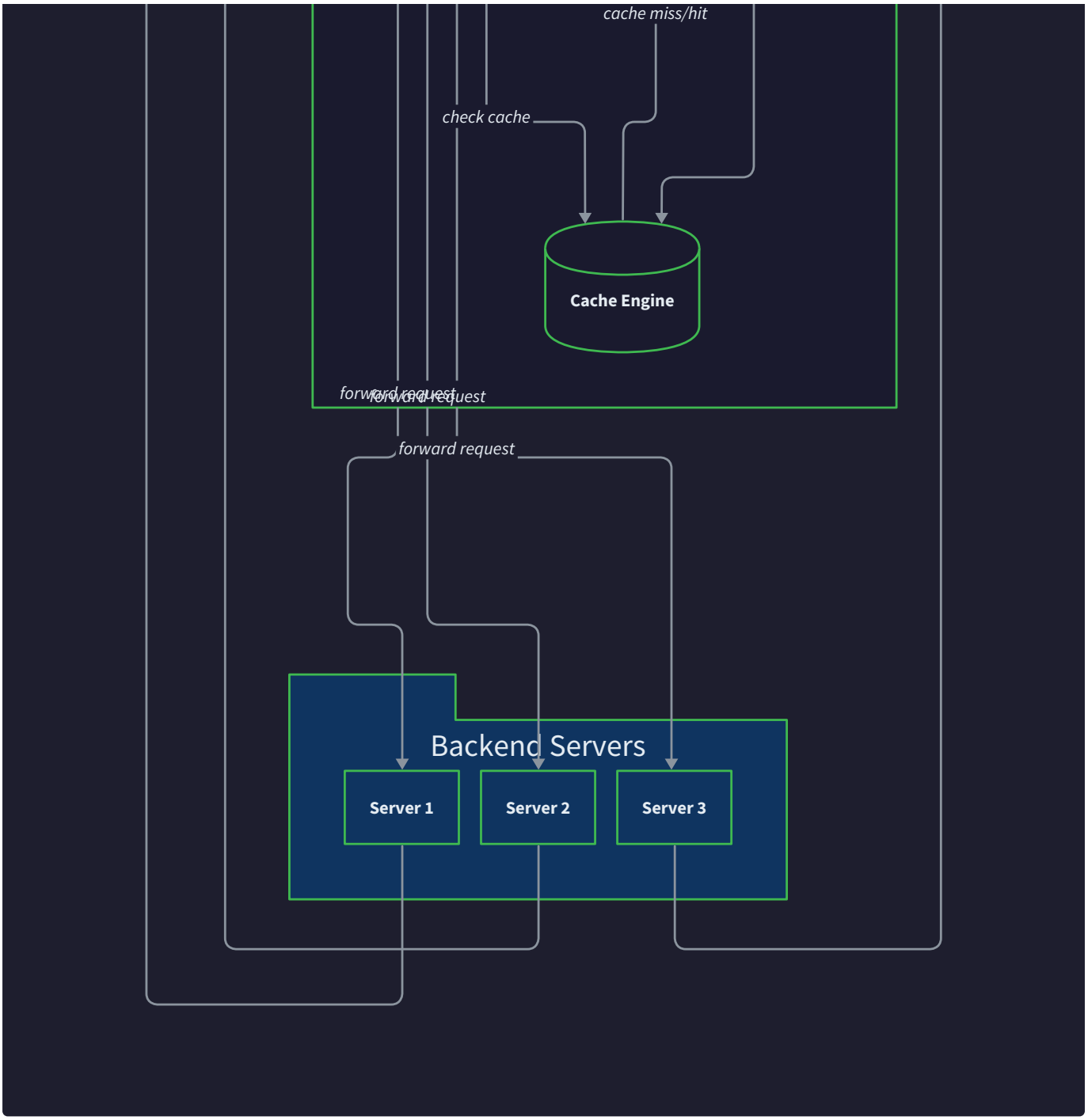
Interactions and Data Flow

> **Milestone(s):** All milestones - understanding component interactions is essential for HTTP proxy core (Milestone 1), load balancing (Milestone 2), connection pooling (Milestone 3), caching (Milestone 4), and SSL termination (Milestone 5).

Think of a reverse proxy as a sophisticated air traffic control system. Just as air traffic controllers coordinate multiple aircraft, runways, and weather services to safely guide planes from departure to destination, a reverse proxy orchestrates multiple components - HTTP parsers, connection managers, load balancers, cache engines, and SSL terminators - to guide client requests through a complex journey to backend servers and back. Each component has specialized responsibilities, but they must communicate seamlessly to deliver a cohesive service. The air traffic analogy helps us understand that timing, coordination, and clear communication protocols between components are absolutely critical for system reliability and performance.

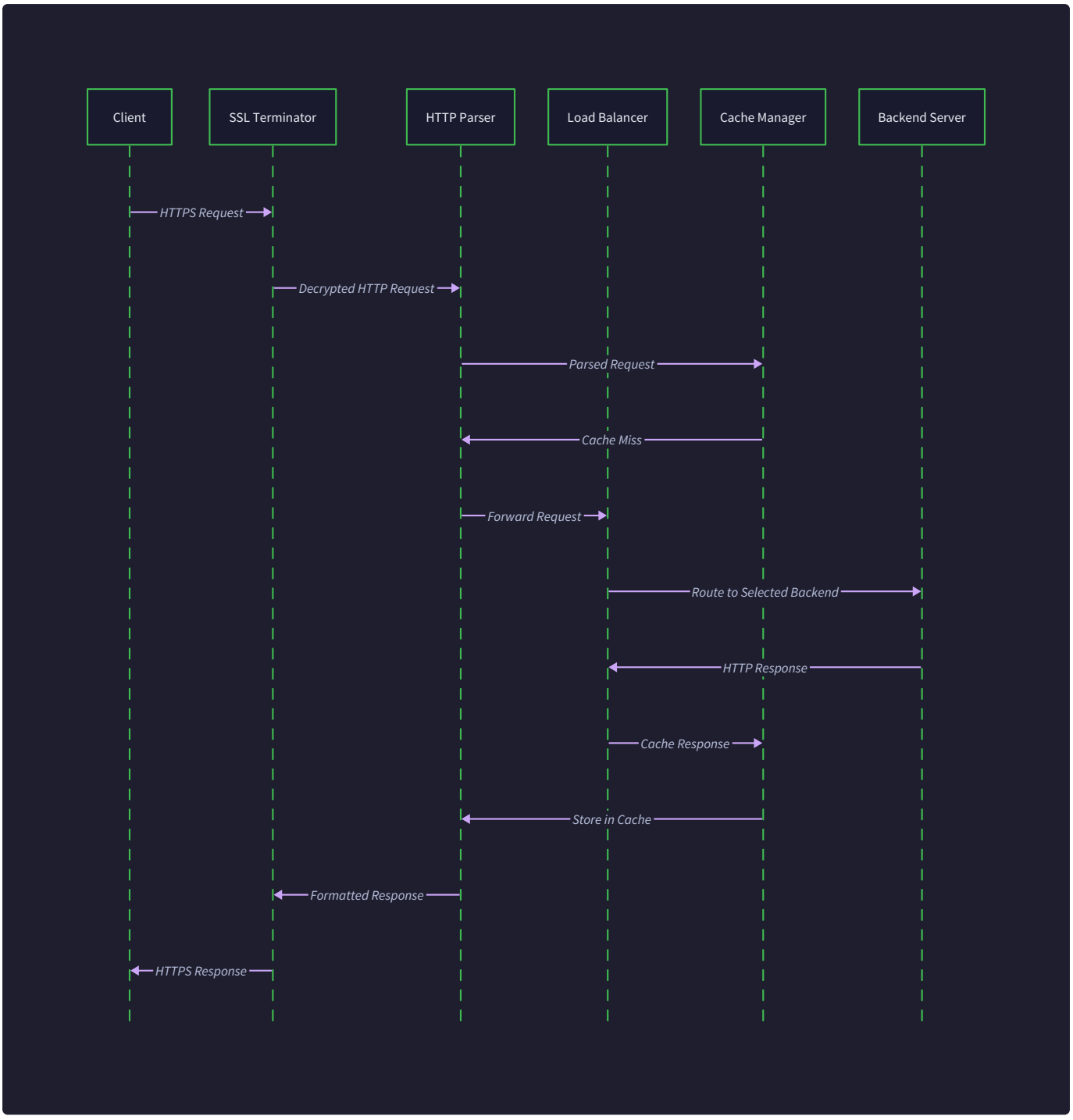
The interaction patterns between reverse proxy components follow well-defined protocols, much like how air traffic control uses standardized communication procedures. Each component exposes specific interfaces, maintains its own internal state, and participates in the larger request processing workflow. Understanding these interactions is crucial because the reverse proxy's performance, reliability, and correctness depend entirely on how effectively these components collaborate.


System Component Architecture
Client Connections
HTTPS requests
HTTPS response
Reverse Proxy System
SSL Termination
decrypted requests
encrypted response
HTTP Parser
parsed HTTP
response
response
response
Connection Manager
route request
Load Balancer
store response



Request Processing Flow

The request processing flow represents the complete journey of an HTTP request from client arrival to response delivery, traversing through multiple components in a carefully orchestrated sequence. Think of this flow as a factory assembly line where each station (component) performs specialized operations on the product (HTTP request) before passing it to the next station. However, unlike a simple linear assembly line, the reverse proxy flow includes decision points, parallel processing paths, and feedback loops that make it more sophisticated.



The request processing begins when a client establishes a connection to the reverse proxy's listening socket. This initial connection may be either a plain HTTP connection or an HTTPS connection requiring SSL termination. The distinction is crucial because it determines which component handles the initial connection establishment and how the request data flows through the system.

Phase 1: Connection Establishment and Initial Processing

Step 1: Client Connection Arrival

When a client attempts to connect, the reverse proxy's main event loop detects the incoming connection on the listening socket. The proxy examines the destination port to determine whether this is an HTTP connection (typically port 80) or an HTTPS connection (typically port 443). This determination affects the entire subsequent processing pipeline because HTTPS connections require SSL termination before HTTP parsing can begin.

For HTTP connections, the `ConnectionManager` directly accepts the client socket using `connection_manager_accept_client()`, creating a new `Connection` structure initialized in the `CONNECTION_IDLE` state. The connection includes buffers for request and response data, timing information, and state tracking fields that guide subsequent processing decisions.

For HTTPS connections, the `SSLTermination` component first accepts the connection using `ssl_termination_accept_connection()`, which performs the TLS handshake. During this handshake, the SSL termination component examines the Server Name Indication (SNI) extension to select the appropriate SSL certificate for the requested hostname. The `ssl_termination_sni_callback()` function handles this certificate selection by looking up the hostname in the SSL context hash table and returning the corresponding SSL context.

Step 2: SSL Termination Processing (HTTPS only)

For HTTPS connections, the SSL termination process involves several critical steps that must complete successfully before HTTP parsing can begin. The TLS handshake includes cipher suite negotiation, where the client and server agree on encryption algorithms. The `SSLTermination` component enforces minimum TLS versions and secure cipher suites to maintain security standards.

Once the TLS handshake completes successfully, the SSL termination component creates an encrypted channel with the client. All subsequent data from the client arrives encrypted and must be decrypted using the established SSL context before passing to other components. The SSL termination component maintains the mapping between client file descriptors and their corresponding SSL contexts, enabling efficient decryption of incoming data.

The decrypted HTTP data is then passed to the `ConnectionManager` for further processing, effectively converting the HTTPS connection into an internal HTTP data stream that the rest of the proxy can handle uniformly.

Step 3: HTTP Request Parsing

Once the connection is established (and decrypted if necessary), the `ConnectionManager` transitions the connection to the `CONNECTION_READING_REQUEST` state and begins accumulating HTTP request data. As data arrives on the client socket, the connection manager appends it to the connection's request buffer using `buffer_append()`.

The HTTP parsing process is stream-based, meaning it processes data incrementally as it arrives rather than waiting for the complete request. The `HttpParser` component uses a state machine approach with states defined in the `HttpParserState` enumeration. The parser begins in the `HTTP_PARSING_REQUEST_LINE` state and processes the incoming data byte by byte.

The `http_parser_process()` function examines each byte and makes state transitions based on the current parser state and the incoming character. For example, when parsing the request line, the parser accumulates characters until it encounters a space character, indicating the end of the HTTP method. It then transitions to parsing the URI, accumulates characters until the next space, and continues until it completes the request line with a CRLF sequence.

Header parsing follows a similar pattern, where the parser accumulates header name and value pairs until it encounters the empty line that separates headers from the message body. The parser handles various HTTP complexities including case-insensitive header names, multi-line header values, and proper handling of the `Content-Length` and `Transfer-Encoding` headers that determine body parsing strategy.

Step 4: Request Validation and Preprocessing

After successful HTTP parsing, the `ConnectionManager` performs request validation to ensure the parsed request meets basic correctness requirements. This validation includes verifying that required headers are present, checking URI format validity, and ensuring HTTP version compatibility.

The connection manager also performs request preprocessing by adding standard reverse proxy headers. The `X-Forwarded-For` header is added with the client's IP address, allowing backend servers to identify the original client. The `Via` header is added to indicate that the request passed through the reverse proxy, following HTTP specification requirements for proxy identification.

During preprocessing, the connection manager examines the `Connection` header to determine if the client requested keep-alive behavior. This information is stored in the connection's `keep_alive` field and influences connection lifecycle management decisions later in the processing flow.

Phase 2: Cache Lookup and Backend Selection

Step 5: Cache Engine Consultation

Before forwarding the request to backend servers, the reverse proxy consults the `CacheEngine` to determine if a cached response can satisfy the request. The cache engine performs this lookup using `cache_engine_lookup()`, which generates a cache key based on the request characteristics.

The cache key generation process considers multiple request attributes including the HTTP method, complete URI (including query parameters), and specific headers that affect response content. The `Vary` header from previously cached responses influences which request headers are included in the cache key calculation, ensuring that responses cached for different client capabilities are served appropriately.

If a cache entry is found, the cache engine validates its freshness using `cache_is_fresh()`. This validation examines the entry's expiration time, calculated from `Cache-Control` directives like `max-age` and response headers like `Expires`. Fresh cache entries can be served immediately, bypassing backend server communication entirely.

For stale cache entries, the cache engine can generate conditional requests using `cache_create_conditional()`. These conditional requests include `If-None-Match` headers with the cached response's ETag or `If-Modified-Since` headers with the cached response's last modification time. Conditional requests allow backend servers to respond with `304 Not Modified` if the content hasn't changed, saving bandwidth and processing time.

Step 6: Load Balancer Backend Selection

When a cache miss occurs or the request cannot be satisfied from cache, the `LoadBalancer` component selects an appropriate backend server using `loadbalancer_select_backend()`. The selection process depends on the configured load balancing algorithm and the current health status of backend servers.

The load balancer first filters the backend server list to exclude servers marked as unhealthy by the health checking system. Unhealthy servers are identified by failed health check attempts or recent request failures that exceed the configured failure threshold.

For healthy servers, the selection algorithm depends on the configured `LoadBalancingAlgorithm`. Round-robin selection uses the `rr_current_index` field to track the next server in rotation, incrementing atomically after each selection to ensure thread-safe operation across concurrent requests. Least-connections selection examines the `connection_counts` array to identify the backend server with the minimum number of active connections.

Weighted round-robin selection is more complex, using the `current_weights` array to implement the weighted round-robin algorithm. Each server's current weight is incremented by its static weight on each selection, and the server with the highest current weight is chosen. The selected server's current weight is then decremented by the sum of all server weights, ensuring that servers with higher weights are selected proportionally more often over time.

The IP hash algorithm provides session affinity by computing a hash of the client's IP address and using this hash to consistently select the same backend server for requests from the same client. This approach ensures that clients with session state requirements are always directed to the same backend server.

Phase 3: Backend Communication and Connection Management

Step 7: Backend Connection Acquisition

After backend server selection, the `ConnectionManager` acquires a connection to the selected backend using `connection_manager_acquire_backend()`. This function implements connection pooling by first checking if any idle

connections to the target backend server exist in the connection pool.

Connection pools are maintained per backend server in the `backend_pools` array, with each `ConnectionPool` structure tracking idle and active connections. The pool uses a LIFO (Last-In, First-Out) strategy for connection reuse, which provides better cache locality because recently used connections are more likely to have warm CPU caches and established network paths.

When reusing a pooled connection, the connection manager validates the connection's health by checking if the socket is still readable without data available (indicating a connection reset) and verifying that the connection hasn't exceeded its maximum idle time. Invalid connections are discarded and replaced with new connections.

If no suitable pooled connections exist, the connection manager creates a new connection to the backend server. This involves establishing a TCP socket, setting appropriate socket options like `TCP_NODELAY` to minimize latency, and configuring non-blocking I/O mode for integration with the event-driven architecture.

Step 8: Request Forwarding

With a backend connection acquired, the connection manager forwards the HTTP request to the backend server. The forwarding process involves serializing the parsed HTTP request back into the standard HTTP wire format, but with modifications appropriate for backend communication.

The forwarded request includes the modified headers added during preprocessing, ensuring that backend servers receive client identification information. The connection manager may modify or remove certain headers that are only relevant for client-proxy communication, such as `Proxy-Authorization` headers.

Request forwarding handles various HTTP complexities including chunked transfer encoding and request bodies. For requests with `Content-Length` headers, the connection manager ensures that the specified number of bytes are forwarded to the backend. For chunked requests, the proxy forwards the chunk headers and data while tracking the chunking state to detect the end of the request body.

During forwarding, the connection transitions to the `CONNECTION_FORWARDING` state, and the connection manager configures epoll events to monitor both the client connection for additional request data and the backend connection for response data.

Step 9: Response Processing and Parsing

When the backend server begins sending response data, the connection manager transitions to the `CONNECTION_READING_RESPONSE` state and begins accumulating response data in the connection's response buffer. Response parsing follows a similar pattern to request parsing, using the HTTP parser component to incrementally process the response data.

The HTTP response parser extracts the status line, response headers, and message body. Special attention is paid to caching-related headers including `Cache-Control`, `Expires`, `ETag`, `Last-Modified`, and `Vary`. These headers determine whether the response can be cached and how long it remains valid.

Response parsing also handles various HTTP response formats including chunked transfer encoding and keep-alive connection management. The parser tracks the response completeness to determine when the entire response has been received from the backend server.

Phase 4: Response Caching and Client Delivery

Step 10: Cache Storage Decision

After receiving the complete response from the backend server, the `CacheEngine` evaluates whether the response should be cached using `cache_is_cacheable()`. This evaluation examines the response status code, HTTP method, and cache-control directives to determine cacheability.

Cacheable responses are typically successful GET requests (status 200) with explicit caching directives or responses without explicit no-cache directives. The cache engine respects `Cache-Control: no-store` directives by never storing such responses, and handles `Cache-Control: private` directives by considering the proxy's role as a shared cache.

For cacheable responses, the cache engine calculates the expiration time using `cache_control_calculate_expiry()`, which considers `max-age` directives, `Expires` headers, and heuristic expiration calculations for responses without explicit expiration information.

The cache storage process using `cache_engine_store()` involves generating the same cache key used during lookup, creating a `CacheEntry` structure with the response data and metadata, and inserting the entry into both the hash table for fast lookup and the LRU list for eviction management.

Step 11: Response Delivery to Client

The final phase involves delivering the response to the client through the appropriate channel. For HTTP connections, the connection manager writes the response data directly to the client socket. For HTTPS connections, the response data must first pass through SSL encryption using the established SSL context.

Response delivery handles various client capabilities including HTTP version differences and connection management preferences. The proxy respects the client's `Connection` header preferences, maintaining keep-alive connections when requested and possible, or closing connections when appropriate.

During response delivery, the connection transitions to the `CONNECTION_WRITING_RESPONSE` state and configures epoll events to monitor the client socket for write readiness. The delivery process handles partial writes and flow control, ensuring that response data is delivered reliably even when client connections have limited receive buffers.

Step 12: Connection Cleanup and Recycling

After completing response delivery, the connection manager performs cleanup and recycling operations. Backend connections are returned to the connection pool using `connection_manager_release_backend()` if they remain healthy and the pool has capacity. This recycling process updates connection statistics and resets connection state for future reuse.

Client connections are either maintained for additional requests (HTTP keep-alive) or closed based on the HTTP version, client preferences, and error conditions. Keep-alive connections transition back to the `CONNECTION_IDLE` state and remain registered for read events to detect additional incoming requests.

The connection manager also updates various statistics including request counts, response times, and error rates. These statistics support monitoring, debugging, and performance optimization activities.

Inter-Component Communication

The inter-component communication patterns define how the reverse proxy's components exchange information, coordinate activities, and maintain consistency across the distributed system architecture. Think of inter-component communication as the nervous system of the reverse proxy, carrying control signals, data messages, and status updates between specialized organs (components) that must work together to maintain the system's health and functionality.

Understanding these communication patterns is essential because they determine system performance characteristics, reliability behaviors, and debugging approaches. The communication patterns also define the system's scalability limits and guide optimization strategies for high-performance deployments.

Communication Architecture and Patterns

The reverse proxy uses an event-driven communication architecture where components interact through well-defined interfaces rather than direct memory sharing or global variables. This architecture provides isolation between components, enabling independent testing, debugging, and optimization of each component while maintaining clear contracts for inter-component collaboration.

Message-Based Communication

Components communicate primarily through structured messages passed via function calls rather than shared memory regions. This approach provides type safety, clear ownership semantics, and explicit error handling paths that improve system reliability and

maintainability.

Message Type	Source Component	Target Component	Purpose	Data Included
HTTP Request	Connection Manager	HTTP Parser	Request parsing initiation	Raw HTTP data buffer
Parsed Request	HTTP Parser	Connection Manager	Parsing completion notification	HttpRequest structure
Backend Selection	Connection Manager	Load Balancer	Backend server selection	HttpRequest, client info
Selected Backend	Load Balancer	Connection Manager	Backend selection result	BackendServer pointer
Cache Lookup	Connection Manager	Cache Engine	Cache hit evaluation	HttpRequest, cache key
Cache Entry	Cache Engine	Connection Manager	Cache lookup result	CacheEntry or null
SSL Handshake	Connection Manager	SSL Termination	TLS connection establishment	Client socket, SNI hostname
Decrypted Data	SSL Termination	Connection Manager	Decrypted HTTP data	Buffer with plain HTTP

Event-Driven Coordination

The communication architecture uses event-driven coordination where components register for specific events and respond asynchronously when those events occur. The `ConnectionManager` serves as the central event coordinator, using epoll-based event monitoring to detect socket events and coordinating responses across multiple components.

Event-driven coordination enables high concurrency because components don't block waiting for responses from other components. Instead, components initiate operations and continue processing other tasks while waiting for asynchronous completion notifications.

Component Interface Specifications

Each component exposes well-defined interfaces that specify exactly how other components can interact with it. These interfaces include function signatures, parameter requirements, return value semantics, and error handling behaviors.

Connection Manager Interfaces

The `ConnectionManager` provides the primary coordination interfaces used by other components to participate in request processing workflows.

Interface Method	Parameters	Return Value	Description	Error Conditions
`connection_manager_accept_client`	`ConnectionManager*`, `int client_fd`	`Connection*`	Accept new client connection and initialize state	Returns NULL on memory allocation failure
`connection_manager_acquire_backend`	`ConnectionManager*`, `BackendServer*`, `int timeout_ms`	`Connection*`	Acquire backend connection from pool or create new	Returns NULL on connection failure or timeout
`connection_manager_release_backend`	`ConnectionManager*`, `Connection*`, `bool keep_alive`	`void`	Return backend connection to pool or close	No error return - best effort cleanup
`connection_manager_handle_event`	`ConnectionManager*`, `Connection*`, `uint32_t events`	`void`	Process connection state transition events	Logs errors but continues processing
`connection_manager_close_connection`	`ConnectionManager*`, `Connection*`	`void`	Clean up connection and release resources	No error return - best effort cleanup

HTTP Parser Interfaces

The `HttpParser` component provides stream-based parsing interfaces that support incremental processing of HTTP data as it arrives from network connections.

Interface Method	Parameters	Return Value	Description	Error Conditions
`http_parser_create`	`void`	`HttpParser*`	Initialize new parser instance	Returns NULL on memory allocation failure
`http_parser_process`	`HttpParser*`, `char* data`, `size_t length`	`int`	Process incoming HTTP data incrementally	Returns negative on parsing errors
`http_request_create`	`void`	`HttpRequest*`	Create empty HTTP request structure	Returns NULL on memory allocation failure
`http_request_add_header`	`HttpRequest*`, `char* name`, `char* value`	`bool`	Add header to request structure	Returns false on memory allocation failure
`http_request_get_header`	`HttpRequest*`, `char* name`	`char*`	Retrieve header value by name	Returns NULL if header not found

Load Balancer Interfaces

The `LoadBalancer` component provides backend selection interfaces that implement various distribution algorithms and health checking logic.

Interface Method	Parameters	Return Value	Description	Error Conditions
`loadbalancer_select_backend`	`LoadBalancer*`, `HttpRequest*`	`BackendServer*`	Select backend using configured algorithm	Returns NULL if no healthy backends available
`loadbalancer_update_connection_count`	`LoadBalancer*`, `BackendServer*`, `int delta`	`void`	Update active connection count for backend	No error return - uses atomic operations
`select_backend_round_robin`	`LoadBalancer*`	`BackendServer*`	Round-robin backend selection implementation	Returns NULL if no healthy backends
`select_backend_least_connections`	`LoadBalancer*`	`BackendServer*`	Least-connections selection implementation	Returns NULL if no healthy backends
`health_check_backend`	`LoadBalancer*`, `BackendServer*`	`bool`	Perform health check against backend	Returns false on health check failure

Cache Engine Interfaces

The `CacheEngine` component provides caching interfaces that implement HTTP semantics for cache storage, retrieval, and invalidation operations.

Interface Method	Parameters	Return Value	Description	Error Conditions
`cache_engine_lookup`	`CacheEngine*`, `HttpRequest*`	`CacheEntry*`	Find cached response for request	Returns NULL on cache miss
`cache_engine_store`	`CacheEngine*`, `char* key`, `HttpResponse*`	`bool`	Store response in cache if cacheable	Returns false if not cacheable or storage failure
`cache_engine_invalidate`	`CacheEngine*`, `char* key`	`bool`	Remove specific entry from cache	Returns false if key not found
`cache_generate_key`	`HttpRequest*`	`char*`	Generate cache key for request	Returns NULL on memory allocation failure
`cache_is_fresh`	`CacheEntry*`, `time_t current_time`	`bool`	Check if cached entry is still fresh	No error conditions

SSL Termination Interfaces

The `SSLTermination` component provides TLS-related interfaces for connection establishment, certificate management, and data encryption/decryption operations.

Interface Method	Parameters	Return Value	Description	Error Conditions
`ssl_termination_accept_connection`	`SSLTermination*`, `int client_fd`	`SSL*`	Perform TLS handshake for client	Returns NULL on handshake failure
`ssl_termination_sni_callback`	`SSL*`, `int* alert`, `void* context`	`int`	SNI hostname callback for certificate selection	Returns error codes for certificate selection failures
`ssl_termination_reload_certificates`	`SSLTermination*`	`bool`	Reload certificates from disk without restart	Returns false on certificate loading errors
`ssl_utils_load_certificate`	`char* cert_path`, `CertificateInfo*`	`X509*`	Load certificate from PEM file	Returns NULL on file or parsing errors
`ssl_utils_verify_key_cert_match`	`EVP_PKEY*`, `X509*`	`bool`	Verify private key matches certificate	Returns false on key/certificate mismatch

Data Flow Coordination

The data flow coordination mechanisms ensure that HTTP request and response data moves efficiently between components while maintaining data integrity and proper error handling. The coordination patterns handle various complexities including partial data availability, flow control, and cleanup after errors.

Request Data Flow

Request data flows from client connections through multiple processing stages, with each component transforming or enriching the data before passing it to the next stage.

Flow Stage	Input Data	Component	Output Data	Transformation Applied
Raw Network Data	TCP byte stream	Connection Manager	Buffered HTTP data	Socket reading, buffering
HTTP Parsing	Buffered HTTP data	HTTP Parser	HttpRequest structure	Protocol parsing, header extraction
Request Preprocessing	HttpRequest structure	Connection Manager	Enhanced HttpRequest	Header addition, validation
Backend Selection	Enhanced HttpRequest	Load Balancer	BackendServer assignment	Algorithm-based selection
Cache Consultation	Enhanced HttpRequest	Cache Engine	CacheEntry or cache miss	Key generation, lookup
Backend Forwarding	Enhanced HttpRequest	Connection Manager	Serialized HTTP	Request serialization, transmission

Response Data Flow

Response data flows from backend servers back to clients, with caching and SSL encryption stages modifying the data path based on configuration and client capabilities.

Flow Stage	Input Data	Component	Output Data	Transformation Applied
Backend Response	TCP byte stream	Connection Manager	Buffered response data	Socket reading, buffering
Response Parsing	Buffered response data	HTTP Parser	HttpResponse structure	Protocol parsing, header extraction
Cache Storage	HttpResponse structure	Cache Engine	Stored CacheEntry	Cacheability evaluation, storage
Client Delivery	HttpResponse structure	Connection Manager	Serialized HTTP	Response serialization
SSL Encryption	Serialized HTTP	SSL Termination	Encrypted byte stream	TLS encryption (HTTPS only)
Client Transmission	Encrypted/plain data	Connection Manager	Network transmission	Socket writing, flow control

Error Propagation Patterns

Error conditions must be properly propagated between components to ensure that failures are handled gracefully and don't corrupt system state or cause resource leaks.

Error Source	Detection Point	Propagation Path	Recovery Action
Client connection failure	Connection Manager	Direct error return	Connection cleanup, client notification
HTTP parsing error	HTTP Parser	Error return code	Bad request response, connection close
Backend selection failure	Load Balancer	NULL return value	Service unavailable response
Backend connection failure	Connection Manager	Connection timeout	Backend retry or error response
Cache storage failure	Cache Engine	Boolean failure return	Continue without caching
SSL handshake failure	SSL Termination	NULL SSL context	Connection close, SSL error logging

Synchronization and Thread Safety

The inter-component communication must handle concurrent access patterns safely because the reverse proxy processes multiple requests simultaneously across different threads. Each component implements appropriate synchronization mechanisms to protect shared data structures and coordinate access to system resources.

Component-Level Synchronization

Each component implements internal synchronization to protect its data structures from concurrent access corruption while minimizing lock contention that could impact performance.

Component	Synchronization Mechanism	Protected Resources	Lock Granularity
Connection Manager	Per-connection state locks	Connection structures, socket operations	Individual connections
Load Balancer	Reader-writer locks	Backend server list, health status	Backend array and individual servers
Cache Engine	Hash table locks, LRU locks	Cache entries, eviction lists	Hash buckets and LRU operations
SSL Termination	Context map locks	SSL contexts, certificate reloading	SSL context hash table
HTTP Parser	No synchronization needed	Parser state is per-request	Not applicable - stateless

Cross-Component Coordination

Cross-component interactions require careful coordination to avoid deadlocks and ensure consistent system behavior when multiple components must collaborate to complete operations.

The primary coordination mechanism uses the event-driven architecture where components don't hold locks while calling other components. Instead, components complete their internal operations, release any held locks, and then invoke other component methods without holding internal locks.

> **Critical Design Insight**: The event-driven architecture eliminates most cross-component deadlock risks because components don't hold internal locks while making external calls. This design choice trades some potential performance for significantly improved reliability and debuggability.

Component Startup and Shutdown Coordination

The system startup and shutdown sequences require careful coordination to ensure that components initialize in the correct order and that shutdown occurs gracefully without losing in-flight requests or corrupting persistent state.

Startup Sequence

Order	Component	Initialization Actions	Dependencies
1	Configuration Loading	Parse config file, validate settings	File system access
2	SSL Termination	Load certificates, initialize OpenSSL	Certificate files, OpenSSL library
3	Cache Engine	Initialize hash tables, start cleanup threads	Memory allocation
4	Load Balancer	Initialize backend list, start health checking	Backend server network access
5	Connection Manager	Initialize epoll, create listening sockets	Network socket permissions
6	Main Event Loop	Start request processing	All other components ready

Shutdown Sequence

Order	Component	Shutdown Actions	Cleanup Requirements
1	Main Event Loop	Stop accepting new connections	Complete current request processing
2	Connection Manager	Drain existing connections, close sockets	Wait for request completion timeouts
3	Load Balancer	Stop health checking, update backend status	Join health check threads
4	Cache Engine	Stop cleanup threads, sync cached data	Join cleanup threads
5	SSL Termination	Stop certificate reloading, cleanup SSL contexts	Join certificate threads
6	Resource Cleanup	Free memory, close files, release system resources	Final system resource release

Implementation Guidance

The inter-component communication implementation requires careful attention to interface design, error handling, and performance optimization. The following guidance provides concrete implementation strategies for building robust component interactions.

Technology Recommendations

Communication Aspect	Simple Approach	Advanced Approach
Function Interfaces	Direct function calls with error returns	Function pointers with callback interfaces
Data Serialization	Direct structure passing	Protocol buffers or JSON serialization
Event Coordination	Epoll-based event loops	Custom event dispatch framework
Error Handling	Return codes and errno	Structured error objects with context
Inter-thread Communication	Mutex and condition variables	Lock-free queues and atomic operations

Recommended File Structure

```
proxy/
├── src/
│   ├── main.c                    ← Main event loop and component coordination
│   ├── interfaces/               ← Component interface definitions
│   │   ├── connection_manager.h  ← Connection manager interface
│   │   ├── http_parser.h         ← HTTP parser interface
│   │   ├── load_balancer.h       ← Load balancer interface
│   │   ├── cache_engine.h        ← Cache engine interface
│   │   └── ssl_termination.h     ← SSL termination interface
│   ├── core/                     ← Core coordination logic
│   │   ├── proxy_server.c        ← Main server coordination
│   │   ├── event_dispatcher.c    ← Event routing between components
│   │   └── error_handling.c      ← Cross-component error handling
│   └── utils/                    ← Shared utilities
│       ├── buffer.c              ← Buffer management
│       ├── hashtable.c           ← Hash table implementation
│       └── logger.c              ← Logging system
```

Core Event Dispatcher Implementation

```c
#include <sys/epoll.h>

#include <errno.h>

#include <unistd.h>

#include "interfaces/connection_manager.h"

#include "interfaces/ssl_termination.h"


// Event dispatcher coordinates between components

typedef struct {

    int epoll_fd;

    ConnectionManager* conn_mgr;

    SSLTermination* ssl_term;

    bool running;

} EventDispatcher;


// Initialize event dispatcher with component references

EventDispatcher* event_dispatcher_create(ConnectionManager* conn_mgr,

                                         SSLTermination* ssl_term) {

    EventDispatcher* dispatcher = malloc(sizeof(EventDispatcher));

    if (!dispatcher) return NULL;



    dispatcher->epoll_fd = epoll_create1(EPOLL_CLOEXEC);

    if (dispatcher->epoll_fd == -1) {

        free(dispatcher);

        return NULL;

    }



    dispatcher->conn_mgr = conn_mgr;

    dispatcher->ssl_term = ssl_term;

    dispatcher->running = true;



    return dispatcher;

}
```

```c
// Main event processing loop - coordinates all component interactions
void event_dispatcher_run(EventDispatcher* dispatcher) {
    struct epoll_event events[1024];

    while (dispatcher->running) {
        int event_count = epoll_wait(dispatcher->epoll_fd, events, 1024, 1000);

        if (event_count == -1) {
            if (errno == EINTR) continue;
            logger_log(ERROR, __FILE__, __LINE__, "epoll_wait failed: %s", strerror(errno));
            break;
        }

        // TODO 1: Process each epoll event
        // TODO 2: Determine if event is for client connection, backend connection, or listening socket
        // TODO 3: Route event to appropriate component (ConnectionManager or SSLTermination)
        // TODO 4: Handle component coordination for complex operations (cache + backend)
        // TODO 5: Update event monitoring based on connection state changes
        // Hint: Use epoll event data to store Connection* pointers for fast lookup
    }
}
```

Inter-Component Message Structures

```c
// Standard message header for all inter-component communication
typedef struct {
    uint32_t message_type;
    uint32_t message_length;
    uint64_t transaction_id;  // For request tracing
    time_t timestamp;
} MessageHeader;

// Cache lookup request message
typedef struct {
    MessageHeader header;
    char cache_key[512];
    HttpRequest* request;
    bool conditional_request;
} CacheLookupMessage;

// Backend selection request message
typedef struct {
    MessageHeader header;
    HttpRequest* request;
    struct sockaddr_in client_addr;
    LoadBalancingAlgorithm preferred_algorithm;
} BackendSelectionMessage;

// Generic component response message
typedef struct {
    MessageHeader header;
    int result_code;
    char error_message[256];
    void* result_data;
} ComponentResponse;
```

Request Processing Coordination

```c
// Coordinate complete request processing across all components
int proxy_process_request(ProxyServer* server, Connection* client_conn) {
    HttpRequest* request = NULL;
    BackendServer* backend = NULL;
    Connection* backend_conn = NULL;
    CacheEntry* cached_response = NULL;
    int result = -1;

    // TODO 1: Parse HTTP request using HttpParser component
    // TODO 2: Check cache using CacheEngine component - return cached response if fresh
    // TODO 3: Select backend using LoadBalancer component
    // TODO 4: Acquire backend connection using ConnectionManager
    // TODO 5: Forward request to backend server
    // TODO 6: Process backend response and update cache
    // TODO 7: Send response to client (with SSL encryption if needed)
    // TODO 8: Clean up all resources and update connection pools
    // Hint: Use goto cleanup pattern to ensure proper resource cleanup on any error

cleanup:
    // Always clean up resources regardless of success/failure
    if (backend_conn) {
        connection_manager_release_backend(server->conn_mgr, backend_conn,
                                           result == 0 && client_conn->keep_alive);
    }
    if (request) http_request_destroy(request);
    return result;
}
```

Component Error Handling Coordination

```c
// Centralized error handling for cross-component operations
typedef enum {
    PROXY_ERROR_NONE = 0,
    PROXY_ERROR_PARSE_FAILED,
    PROXY_ERROR_BACKEND_UNAVAILABLE,
    PROXY_ERROR_BACKEND_TIMEOUT,
    PROXY_ERROR_CACHE_FAILURE,
    PROXY_ERROR_SSL_HANDSHAKE_FAILED,
    PROXY_ERROR_CLIENT_DISCONNECTED
} ProxyErrorCode;

// Handle errors that span multiple components
void proxy_handle_error(ProxyServer* server, Connection* client_conn,
                        ProxyErrorCode error_code, const char* error_details) {
    HttpResponse* error_response = NULL;

    // TODO 1: Log error with appropriate severity level
    // TODO 2: Create appropriate HTTP error response based on error code
    // TODO 3: Update component statistics (failed requests, error counts)
    // TODO 4: Send error response to client (with SSL if needed)
    // TODO 5: Clean up any partial state in components
    // TODO 6: Close client connection if error is unrecoverable
    // Hint: Different error codes require different HTTP status codes and cleanup actions
}
```

Component Health Monitoring

```c
// Monitor component health and coordination effectiveness

typedef struct {

    uint64_t requests_processed;

    uint64_t cache_hits;

    uint64_t cache_misses;

    uint64_t backend_failures;

    uint64_t ssl_handshake_failures;

    double avg_request_time;

    time_t last_update;

} ProxyStats;


// Collect statistics from all components

void proxy_update_statistics(ProxyServer* server, ProxyStats* stats) {

    // TODO 1: Collect statistics from ConnectionManager (request counts, timing)

    // TODO 2: Collect statistics from CacheEngine (hit rates, evictions)

    // TODO 3: Collect statistics from LoadBalancer (backend health, distribution)

    // TODO 4: Collect statistics from SSLTermination (handshake success rates)

    // TODO 5: Calculate derived metrics (success rates, latency percentiles)

    // TODO 6: Update monitoring endpoints for external systems

    // Hint: Use atomic operations for statistics updates to avoid locks

}
```

Milestone Checkpoints

After implementing the inter-component communication system, verify correct operation using these checkpoints:

1. **Component Isolation Testing**: Each component should be testable independently by providing mock implementations of other component interfaces. Create mock implementations that return predictable results and verify that each component handles the mock responses correctly.
2. **Request Flow Tracing**: Implement transaction ID tracking that follows a single request through all components. Add logging at each component boundary and verify that the transaction ID appears in all log entries for a single request.
3. **Error Propagation Verification**: Inject errors at each component interface and verify that errors propagate correctly to the client without corrupting system state or causing resource leaks.
4. **Concurrent Request Testing**: Process multiple concurrent requests and verify that component interactions don't cause race conditions, deadlocks, or data corruption. Monitor for proper connection pool usage and cache consistency.

Common Implementation Pitfalls

⚠️ **Pitfall: Holding Internal Locks During External Calls** Components should never hold their internal locks while calling methods on other components. This creates deadlock risks when components need to interact bidirectionally. Always complete internal operations and release locks before making external component calls.

⚠️ **Pitfall: Inconsistent Error Handling Across Components** Each component must handle errors consistently and provide meaningful error information to callers. Don't ignore error return values or fail to propagate errors up the call stack. Implement structured error handling that provides context about which component failed and why.

⚠️ **Pitfall: Resource Ownership Confusion** Clearly define which component owns each resource and is responsible for cleanup. Use consistent patterns like "creator owns" or "last user owns" throughout the system. Document ownership rules in interface specifications.

⚠️ **Pitfall: Race Conditions in Component Statistics** Statistics updates across components can create race conditions if not properly synchronized. Use atomic operations for simple counters and appropriate locking for complex statistics calculations. Consider using per-thread statistics with periodic aggregation to reduce contention.

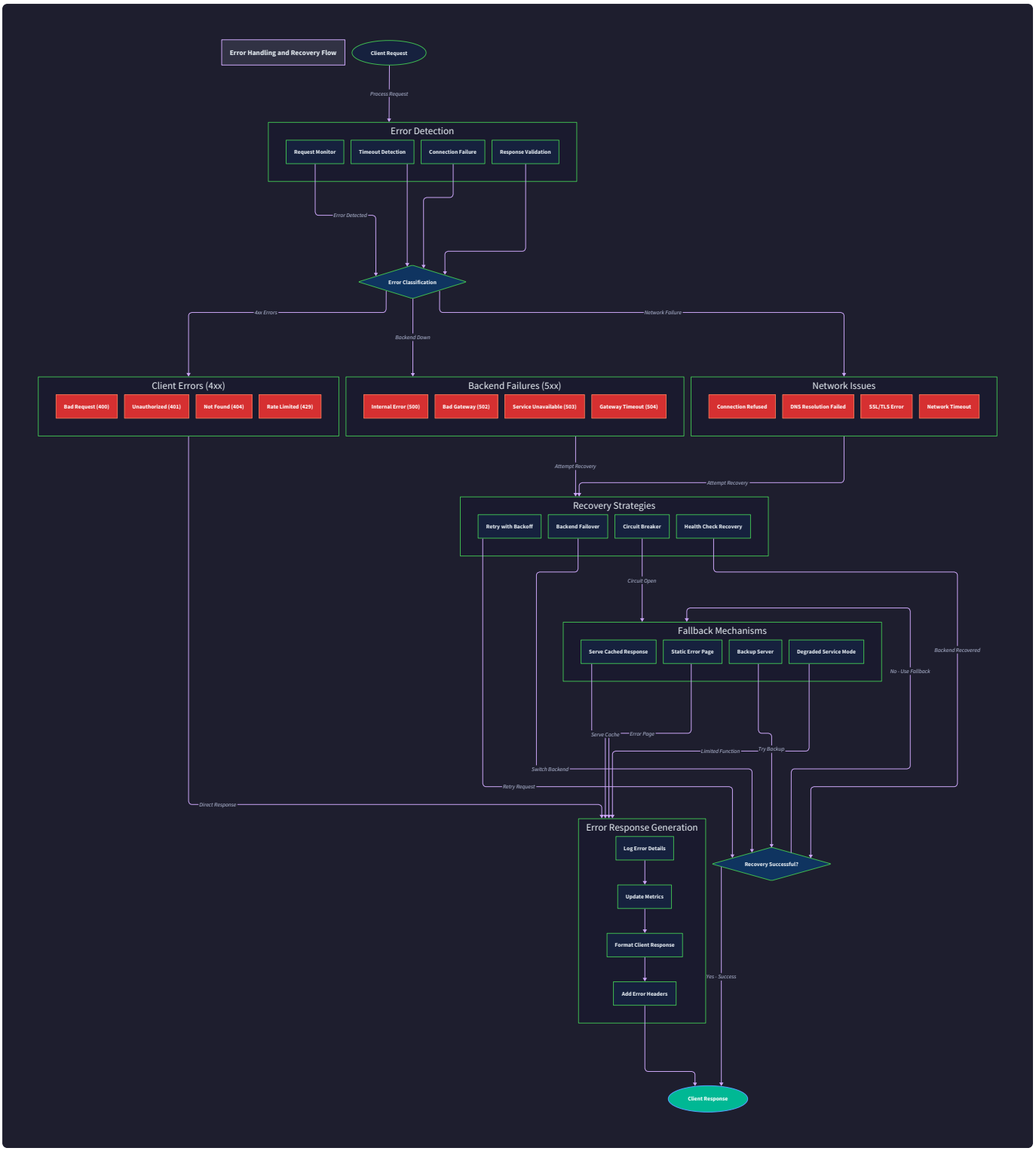
Error Handling and Edge Cases

> **Milestone(s):** All milestones - comprehensive error handling is essential across HTTP proxy core (Milestone 1), load balancing (Milestone 2), connection pooling (Milestone 3), caching (Milestone 4), and SSL termination (Milestone 5).

Think of error handling in a reverse proxy like an air traffic control system managing multiple failure scenarios simultaneously. Just as air traffic controllers must handle plane mechanical failures, weather emergencies, runway closures, and communication blackouts while keeping other flights operating safely, a reverse proxy must gracefully handle client disconnections, backend server failures, network partitions, and SSL certificate problems while continuing to serve other requests. The key insight is that failures are not exceptional cases—they are normal operating conditions that require systematic detection, classification, and recovery strategies.

Building robust error handling requires understanding that failures cascade through components in predictable patterns. When a backend server fails, it affects the load balancer's backend selection, the connection manager's pooled connections, and potentially cached responses that originated from that server. Each failure mode requires specific detection mechanisms, recovery strategies, and graceful degradation approaches to maintain service availability.

The challenge lies in designing error handling that provides strong guarantees while maintaining performance. Error detection must be fast enough not to impact request latency, recovery mechanisms must activate quickly to minimize service disruption, and fallback strategies must preserve as much functionality as possible during partial system failures.



Failure Modes and Detection

Understanding failure modes requires systematically analyzing each point where the reverse proxy interacts with external systems or manages internal state. Think of failure detection like a comprehensive medical diagnostic system—each component must monitor its own vital signs while also coordinating with other components to detect system-wide health issues. The goal is early detection before failures cascade into service outages.

Client Connection Failures represent the most common failure category, occurring when clients disconnect unexpectedly, send malformed requests, or violate protocol specifications. These failures require immediate detection to prevent resource leaks and unnecessary backend processing.

Failure Mode	Detection Method	Symptoms	Root Cause
Client Disconnect During Request	`EPOLLHUP` or `EPOLLERR` events	Connection state shows `CONNECTION_READING_REQUEST` but socket closed	Client closed browser, network timeout, mobile device sleep
Malformed HTTP Request	HTTP parser returns `HTTP_PARSING_ERROR`	Parser unable to extract method, URI, or headers	Client bug, protocol violation, manual crafting
Request Size Exceeding Limits	Buffer allocation failure or size checks	Request buffer exceeds configured maximum	Large file upload, DoS attempt, misconfigured client
Slow Client Attack	Timer wheel timeout on request reading	Connection remains in `CONNECTION_READING_REQUEST` beyond timeout	Intentional slowloris attack or very slow network
SSL Handshake Failure	SSL_accept returns error codes	TLS negotiation fails during connection establishment	Wrong certificate, cipher mismatch, protocol version incompatibility

The detection strategy for client failures centers on the connection state machine monitoring and timeout management. The `ConnectionManager` tracks each connection's state and uses the `TimerWheel` to detect connections that remain in transitional states beyond configured thresholds.

Backend Server Failures require more sophisticated detection because they affect multiple concurrent requests and the overall service capacity. Backend failures can be immediate (connection refused) or gradual (increasing response times, partial responses).

Failure Mode	Detection Method	Symptoms	Impact
Backend Server Down	Connection establishment failure, `ECONNREFUSED`	`connect()` system call fails immediately	All requests to this backend fail
Backend Network Partition	Connection timeout during establishment	`connect()` hangs then times out after `SO_SNDTIMEO`	Backend appears down but may be healthy
Backend Slow Response	Response timeout using timer wheel	Connection stuck in `CONNECTION_READING_RESPONSE`	Requests queue up, response times increase
Backend Partial Response	Incomplete response parsing, unexpected connection close	`HttpParser` detects incomplete response or connection closes mid-response	Data corruption, partial service failure
Backend Health Check Failure	Active health check request failure	HTTP request to health check endpoint returns non-200 status	Backend overloaded or failing internally

The `LoadBalancer` component implements comprehensive backend failure detection through both passive monitoring (observing request failures) and active health checking (periodic probe requests). The health checking system maintains failure counters and implements exponential backoff to avoid overwhelming failing backends.

> **Key Insight**: Backend failure detection must balance responsiveness with stability. Too aggressive failure detection leads to healthy backends being marked down due to temporary network hiccups. Too conservative detection allows failing backends to impact user requests.

Connection Pool Failures occur when the connection pooling mechanism itself experiences problems, such as resource exhaustion, connection leaks, or pool corruption.

Failure Mode	Detection Method	Symptoms	Prevention
Connection Pool Exhaustion	Pool creation fails when `active_count + idle_count >= max_connections`	New requests wait indefinitely for connections	Monitor pool utilization, implement connection limits
Stale Connection Reuse	Backend connection fails immediately after retrieval from pool	First request on reused connection gets `EPIPE` or `ECONNRESET`	Validate connections before reuse, implement keep-alive timeouts
Connection Leak	Pool size grows continuously, connections never returned	`active_count` increases without corresponding decreases	Audit connection lifecycle, ensure all code paths release connections
Pool Lock Contention	High latency on connection acquisition	Threads block on `pool_mutex` for extended periods	Implement pool sharding, reduce critical section duration

The `ConnectionPool` implements connection validation by sending a minimal probe (like TCP `MSG_PEEK`) before returning connections from the idle pool. Stale connections are discarded and new connections established as needed.

Caching System Failures impact performance rather than correctness, but require careful handling to avoid serving stale or corrupted cached responses.

Failure Mode	Detection Method	Symptoms	Recovery Strategy
Cache Memory Exhaustion	Memory allocation failures during cache storage	`cache_engine_store()` returns false, new entries not cached	Emergency cache eviction, memory pressure relief
Cache Corruption	Checksum validation failure, invalid cache entry structure	Cache lookup returns malformed response	Invalidate corrupted entries, fall back to backend
TTL Heap Corruption	Heap invariant violations during TTL processing	Expired entries not evicted, fresh entries evicted prematurely	Rebuild TTL heap, restart cache cleanup thread
Cache Lock Deadlock	Cache operations hang indefinitely	Threads block on `cache_rwlock` or `lru_mutex`	Detect deadlock using timeouts, restart cache subsystem

The `CacheEngine` implements defensive programming by validating cache entry integrity before returning cached responses. Corrupted entries are immediately purged and requests fall back to backend servers.

SSL/TLS Failures require special handling because they involve cryptographic operations and certificate management with specific security implications.

Failure Mode	Detection Method	Symptoms	Security Impact
Certificate Expiration	Certificate validity period check during context creation	SSL context creation fails, `ssl_utils_load_certificate()` reports expiration	Clients receive certificate errors, connections fail
Private Key Mismatch	Key-certificate validation during SSL context setup	`ssl_utils_verify_key_cert_match()` returns false	SSL handshake failures, authentication problems
SNI Certificate Missing	SNI callback unable to find matching certificate	`ssl_termination_sni_callback()` returns `SSL_TLSEXT_ERR_ALERT_FATAL`	Wrong certificate served, client warnings
Cipher Suite Negotiation Failure	SSL handshake fails during cipher selection	TLS handshake aborts with cipher-related errors	Connection establishment fails, compatibility issues
Certificate Chain Validation Failure	Chain verification fails during certificate loading	`ssl_utils_load_certificate()` unable to build trust chain	Browser security warnings, connection failures

The `SSLTermination` component implements certificate monitoring through background validation threads that check certificate expiration dates and reload certificates when files change on disk.

> **Architecture Decision: Failure Detection Strategy**
>
> - **Context**: Need to balance early failure detection with system stability and performance
> - **Options Considered**:
>   1. Reactive detection (detect failures only when they impact requests)
>   2. Proactive detection (active monitoring and health checking)
>   3. Hybrid approach (combine passive observation with active probing)
> - **Decision**: Hybrid approach with component-specific strategies
> - **Rationale**: Reactive detection alone misses gradual degradation and allows cascade failures. Pure proactive detection creates excessive monitoring overhead. Hybrid approach provides early warning while minimizing false positives.
> - **Consequences**: Requires coordination between components for failure state sharing, but provides robust early detection with acceptable overhead.

Recovery and Fallback Strategies

Recovery strategies must address the fundamental challenge of maintaining service availability while dealing with component failures. Think of this like a hospital emergency response system—when one department fails, other departments must absorb the load while the failed department recovers, all without compromising patient care. The key principle is graceful degradation: the system should lose functionality gradually rather than failing catastrophically.

Client Connection Recovery focuses on clean resource management and appropriate error responses that help clients understand and respond to failure conditions.

When client disconnections are detected during request processing, the recovery strategy prioritizes preventing resource leaks and canceling unnecessary backend processing. The `ConnectionManager` implements a comprehensive cleanup sequence that releases all resources associated with the failed client connection.

Recovery Scenario	Detection Point	Recovery Actions	Client Response
Client Disconnect During Request Reading	`EPOLLHUP` event on client socket	1. Cancel request parsing 2. Release client connection 3. Clean up request buffers 4. Log disconnect for monitoring	None (client already disconnected)
Client Disconnect During Backend Processing	`EPOLLHUP` during `CONNECTION_FORWARDING`	1. Cancel backend request 2. Return backend connection to pool 3. Release client connection 4. Update backend statistics	None (client already disconnected)
Malformed Request Parsing	Parser enters `HTTP_PARSING_ERROR` state	1. Generate HTTP 400 Bad Request response 2. Include specific error details 3. Close connection (HTTP/1.0) or reset stream (HTTP/2)	HTTP 400 with error description
Request Size Limit Exceeded	Buffer allocation fails or size check triggers	1. Generate HTTP 413 Payload Too Large 2. Close connection immediately 3. Log oversized request attempt	HTTP 413 with size limit information
Client Timeout During Request	Timer wheel timeout fires	1. Generate HTTP 408 Request Timeout 2. Close connection 3. Release all associated resources	HTTP 408 with timeout duration

The client error response generation follows HTTP specification guidelines to provide actionable feedback. Error responses include specific details about the failure condition and suggested client actions when appropriate.

Backend Server Recovery requires sophisticated strategies because backend failures affect multiple concurrent client requests and the overall system capacity. The recovery approach must minimize impact on active requests while restoring service capacity.

When backend servers fail, the `LoadBalancer` immediately removes them from the active pool and redistributes their load across remaining healthy backends. This requires careful coordination between the load balancer and connection manager to handle in-flight requests appropriately.

Recovery Scenario	Detection Trigger	Immediate Actions	Long-term Recovery
Backend Connection Refused	`connect()` fails with `ECONNREFUSED`	1. Mark backend unhealthy 2. Select alternative backend 3. Retry request on new backend	1. Continue health checking 2. Restore to pool when healthy 3. Adjust health check frequency
Backend Response Timeout	Timer wheel timeout on response reading	1. Close backend connection 2. Return connection failure to load balancer 3. Generate HTTP 504 Gateway Timeout for client	1. Increase backend timeout threshold 2. Monitor for systematic slowness 3. Consider backend capacity issues
Backend Partial Response	Parser detects incomplete response	1. Close corrupted connection 2. Remove from connection pool 3. Retry request if safe (GET, HEAD) 4. Generate HTTP 502 Bad Gateway if not retriable	1. Monitor for data corruption patterns 2. Investigate backend health 3. Consider network issues
All Backends Down	No healthy backends available	1. Return HTTP 503 Service Unavailable 2. Include Retry-After header 3. Continue health checking all backends	1. Implement emergency backend discovery 2. Alert operations team 3. Consider maintenance mode

The backend recovery system implements intelligent retry logic that distinguishes between retriable requests (idempotent operations like GET and HEAD) and non-retriable requests (potentially state-changing operations like POST and PUT). Retriable

requests automatically retry on alternative backends, while non-retriable requests return errors to avoid duplicate processing.

> **Critical Design Principle**: Backend recovery must never amplify failures. When backends are struggling, the recovery system should reduce load rather than increase it through aggressive retries.

Connection Pool Recovery ensures that connection pooling continues to provide performance benefits even when individual connections fail or pools become corrupted.

The `ConnectionPool` implements a self-healing design that automatically adapts to changing backend conditions. When stale connections are detected, the pool discards them and establishes fresh connections. When pool exhaustion occurs, the system gracefully degrades to creating direct connections while working to restore pool capacity.

Recovery Scenario	Trigger Condition	Immediate Response	Pool Restoration
Stale Connection Detection	Connection validation fails before reuse	1. Discard stale connection 2. Attempt to create new connection 3. Update pool statistics	1. Reduce idle timeout to prevent staleness 2. Increase validation frequency 3. Monitor backend keep-alive settings
Pool Exhaustion	`active_count >= max_connections` and no idle connections	1. Block new requests with timeout 2. Create direct connection if possible 3. Monitor for connection returns	1. Increase pool size if backend can handle it 2. Investigate connection leaks 3. Optimize connection lifecycle
Pool Lock Contention	High latency on mutex acquisition	1. Implement connection request queuing 2. Use timeout on lock acquisition 3. Fall back to direct connections	1. Implement pool sharding 2. Reduce critical section size 3. Consider lock-free data structures
Connection Leak Detection	Pool size grows without bound	1. Force connection cleanup 2. Log connection allocation traces 3. Implement emergency pool reset	1. Audit all connection acquisition sites 2. Add connection lifecycle tracking 3. Implement automatic leak detection

The pool recovery system maintains detailed statistics about connection usage patterns to identify systemic issues. When connection leaks are detected, the system implements emergency pool draining that forcibly closes all connections and rebuilds the pool from scratch.

Cache System Recovery focuses on maintaining performance benefits while ensuring data integrity during cache failures.

When cache system failures occur, the primary strategy is immediate fallback to backend servers while attempting to restore cache functionality in the background. This ensures that service remains available even if caching performance benefits are temporarily lost.

Recovery Scenario	Failure Detection	Immediate Fallback	Cache Restoration
Cache Memory Exhaustion	Memory allocation failures during storage	1. Disable new cache entries 2. Trigger emergency eviction 3. Fall back to backend for all requests	1. Implement aggressive LRU eviction 2. Reduce cache size limits 3. Monitor memory usage patterns
Cache Entry Corruption	Checksum validation failure	1. Purge corrupted entry immediately 2. Serve request from backend 3. Log corruption details	1. Investigate corruption patterns 2. Implement cache entry versioning 3. Add redundant storage
Cache Lock Deadlock	Timeout on cache operations	1. Bypass cache for affected requests 2. Restart cache subsystem 3. Serve from backend	1. Implement deadlock detection 2. Redesign locking hierarchy 3. Add lock timeout mechanisms
TTL Processing Failure	Heap corruption or processing errors	1. Disable TTL-based eviction 2. Implement manual cache clearing 3. Continue serving cached entries	1. Rebuild TTL heap from scratch 2. Implement heap invariant checking 3. Add TTL processing monitoring

The cache recovery system implements a "circuit breaker" pattern that automatically disables caching when error rates exceed configurable thresholds. This prevents cache failures from impacting request processing while allowing time for automatic recovery.

SSL/TLS Recovery requires special handling due to security implications and the need to maintain encrypted communications.

When SSL termination failures occur, the system must balance security requirements with service availability. The recovery strategy prioritizes security over availability—it's better to refuse connections than to serve them with compromised encryption.

Recovery Scenario	Security Impact	Immediate Response	Certificate Recovery
Certificate Expiration	High - clients receive security warnings	1. Disable affected SSL contexts 2. Return HTTP 503 for HTTPS requests 3. Alert operations immediately	1. Install renewed certificates 2. Reload SSL contexts 3. Resume HTTPS service
Private Key Compromise	Critical - potential man-in-the-middle attacks	1. Immediately disable all SSL contexts 2. Refuse all HTTPS connections 3. Generate new key pairs	1. Generate new private keys 2. Request new certificates 3. Update all SSL contexts
SNI Certificate Missing	Medium - wrong certificate served	1. Fall back to default certificate 2. Log SNI hostname mismatches 3. Continue service with warnings	1. Install missing certificates 2. Update SNI configuration 3. Validate certificate coverage
Cipher Suite Negotiation Failure	Medium - reduced security or connection failures	1. Expand cipher suite support 2. Log negotiation failures 3. Maintain secure defaults	1. Update cipher configuration 2. Test client compatibility 3. Balance security with compatibility

The SSL recovery system implements automatic certificate monitoring that checks for expiration dates and triggers renewal processes before certificates expire. Certificate reloading happens atomically to avoid service interruption during updates.

Architecture Decision: Error Recovery Philosophy

- **Context**: Need to balance service availability with operational safety during failures
- **Options Considered**:
  1. Fail-fast: Stop processing immediately when errors are detected
  2. Best-effort: Continue processing with degraded functionality
  3. Graceful degradation: Systematically reduce functionality while maintaining core service
- **Decision**: Graceful degradation with fail-fast for security-critical failures
- **Rationale**: Web services must maintain availability during partial failures, but security violations require immediate protection. Graceful degradation preserves user experience while systematic recovery restores full functionality.
- **Consequences**: Requires sophisticated error classification and recovery coordination, but provides optimal balance of availability and safety.

Common Pitfalls in Error Handling

⚠️ Pitfall: Resource Leaks During Error Conditions

The most common error handling mistake is failing to properly clean up resources when errors occur. This happens because error handling code paths are tested less frequently than success paths, leading to subtle resource leaks that only manifest under stress.

The specific problem occurs when error handling code returns early without calling cleanup functions, or when cleanup functions themselves can fail and propagate errors. For example, if `connection_manager_acquire_backend()` fails after allocating a `Connection` structure but before adding it to the pool, the connection memory and file descriptor leak unless the error path explicitly releases them.

To avoid this pitfall, implement the "RAII pattern" even in C by using consistent cleanup functions and ensuring every resource acquisition has a corresponding release in all code paths. Structure error handling with goto labels that consolidate cleanup operations:

```c
ProxyErrorCode proxy_process_request(ProxyServer* server, Connection* conn) {
    HttpRequest* request = NULL;
    BackendServer* backend = NULL;
    Connection* backend_conn = NULL;

    request = http_request_create();
    if (!request) {
        goto cleanup_connection;
    }

    backend = loadbalancer_select_backend(server->load_balancer, request);
    if (!backend) {
        goto cleanup_request;
    }

    backend_conn = connection_manager_acquire_backend(server->conn_manager, backend, 5000);
    if (!backend_conn) {
        goto cleanup_request;
    }

    // Process request...

    connection_manager_release_backend(server->conn_manager, backend_conn, true);
    http_request_destroy(request);
    return PROXY_ERROR_NONE;

cleanup_request:
    http_request_destroy(request);
cleanup_connection:
    connection_manager_close_connection(server->conn_manager, conn);
    return PROXY_ERROR_BACKEND_UNAVAILABLE;
}
```

⚠️ Pitfall: Cascade Failure Amplification

A critical mistake is designing error handling that amplifies failures rather than containing them. This occurs when error recovery logic creates additional load on already-stressed systems, causing localized failures to become system-wide outages.

The specific anti-pattern involves aggressive retry logic that doesn't implement backoff or circuit breaker patterns. When a backend server starts responding slowly, naive retry logic sends more requests to the struggling server, making the problem worse. Similarly, when health checks detect a failing server, continuing to send health check requests at normal frequency can prevent the server from recovering.

To prevent cascade failures, implement exponential backoff for retries, circuit breaker patterns for failing backends, and reduced health check frequency for known-unhealthy servers:

```c
bool health_check_backend(LoadBalancer* lb, BackendServer* backend) {
    time_t now = time(NULL);
    time_t since_last_failure = now - backend->last_failure_time;

    // Exponential backoff for failed backends
    if (backend->failure_count > 0) {
        time_t backoff_delay = (1 << min(backend->failure_count, 6)) * lb->health_check_interval;
        if (since_last_failure < backoff_delay) {
            return false; // Skip health check during backoff period
        }
    }

    // Proceed with health check...
}
```

⚠️ Pitfall: Inconsistent Error State Between Components

Components can get into inconsistent states where one component believes a resource is available while another component knows it has failed. This happens because error propagation between components is asynchronous and components may cache state that becomes stale during failures.

The specific problem occurs when the `LoadBalancer` marks a backend as healthy based on successful connection establishment, but the `ConnectionManager` discovers that all connections to that backend are actually failing due to application-level issues. The load balancer continues selecting the failed backend because it hasn't received updated failure information.

To maintain consistent error state, implement a centralized error reporting system where all components report failures to a shared state manager:

```c
typedef struct {
    BackendServer* backend;
    ProxyErrorCode error_code;
    time_t error_time;
    char error_details[256];
} ErrorReport;

void proxy_handle_error(ProxyServer* server, Connection* conn, ProxyErrorCode error, char* details) {
    // Update component-specific state
    switch (error) {
        case PROXY_ERROR_BACKEND_TIMEOUT:
            loadbalancer_report_backend_error(server->load_balancer, conn->backend_server, error);
            connection_manager_invalidate_backend_pool(server->conn_manager, conn->backend_server);
            break;
        case PROXY_ERROR_SSL_HANDSHAKE_FAILED:
            ssl_termination_report_error(server->ssl_termination, error, details);
            break;
    }

    // Update global error statistics
    proxy_update_statistics(server, error, details);

    // Log error for monitoring
    logger_log(ERROR, __FILE__, __LINE__, "Request failed: %s", details);
}
```

⚠️ Pitfall: Blocking Operations in Error Handlers

Error handling code often performs blocking operations like DNS lookups, file I/O, or network requests, which can cause the entire event processing thread to stall. This is particularly dangerous because errors tend to occur in clusters, so blocking error handling can quickly exhaust all processing threads.

The specific problem occurs when error handling code tries to immediately reload certificates from disk, perform DNS resolution for alternative backends, or send error notifications over the network. These blocking operations prevent the error handler from processing other events, leading to cascading failures.

To avoid blocking in error handlers, implement asynchronous error processing using background threads or work queues:

```c
typedef struct {
    ProxyErrorCode error_code;
    char error_details[512];
    time_t error_time;
    Connection* failed_connection;
} ErrorWorkItem;

void* error_processing_thread(void* arg) {
    ErrorWorkItem* item;
    while (queue_dequeue(error_queue, (void**)&item)) {
        switch (item->error_code) {
            case PROXY_ERROR_SSL_HANDSHAKE_FAILED:
                // Non-blocking: schedule certificate reload
                ssl_termination_schedule_reload(ssl_termination);
                break;
            case PROXY_ERROR_BACKEND_UNAVAILABLE:
                // Non-blocking: update backend health asynchronously
                loadbalancer_schedule_health_check(load_balancer, item->failed_connection->backend_server);
                break;
        }
        free(item);
    }
    return NULL;
}
```

⚠️ Pitfall: Security Information Leakage in Error Messages

Error messages often inadvertently expose sensitive information about the system's internal structure, configuration, or data. This information can be valuable to attackers attempting to exploit the system.

The specific problem occurs when error messages include internal IP addresses, file paths, database connection strings, or detailed stack traces that reveal implementation details. For example, returning "Connection failed to backend server 192.168.1.100:3306" exposes internal network topology.

To prevent information leakage, implement error sanitization that provides useful feedback to legitimate users while hiding sensitive details:

```c
void generate_client_error_response(Connection* conn, ProxyErrorCode error, char* internal_details) {
    HttpResponse* response = http_response_create();

    switch (error) {
        case PROXY_ERROR_BACKEND_UNAVAILABLE:
            response->status = 503;
            http_response_add_header(response, "Content-Type", "text/plain");
            buffer_append(response->body, "Service temporarily unavailable", 28);
            // Internal details logged but not sent to client
            logger_log(ERROR, __FILE__, __LINE__, "Backend failure: %s", internal_details);
            break;
        case PROXY_ERROR_BACKEND_TIMEOUT:
            response->status = 504;
            http_response_add_header(response, "Content-Type", "text/plain");
            buffer_append(response->body, "Gateway timeout", 15);
            break;
    }

    // Send sanitized response to client
    connection_send_response(conn, response);
    http_response_destroy(response);
}
```

Implementation Guidance

Error handling implementation requires careful coordination between all proxy components to ensure consistent failure detection and recovery. The key challenge is implementing robust error handling without significantly impacting performance or code complexity.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Logging	Standard C stdio with log levels	Structured logging with syslog integration
Error Propagation	Return codes with errno	Custom error types with context
Timeout Management	Simple timer threads	Timer wheel with O(1) operations
Resource Cleanup	Manual cleanup with goto labels	RAII-style cleanup macros
Error Recovery	Immediate fallback strategies	Circuit breaker pattern with state machines
Error Monitoring	Basic counters and logs	Metrics collection with alerting

File Structure for Error Handling

```
src/
├── error/
│   ├── error_types.h         ← Error code definitions and structures
│   ├── error_handler.c       ← Central error coordination
│   ├── error_reporter.c      ← Error logging and monitoring
│   └── recovery_manager.c    ← Recovery strategy implementation
├── utils/
│   ├── cleanup_macros.h      ← RAII-style cleanup helpers
│   └── timeout_manager.c     ← Centralized timeout handling
└── monitoring/
    ├── metrics.c             ← Performance and error metrics
    └── health_monitor.c      ← System health monitoring
```

Core Error Handling Infrastructure

The error handling system requires a centralized error coordinator that receives error reports from all components and orchestrates appropriate recovery actions:

```c
// error/error_types.h

typedef enum {

    PROXY_ERROR_NONE = 0,

    PROXY_ERROR_PARSE_FAILED,

    PROXY_ERROR_BACKEND_UNAVAILABLE,

    PROXY_ERROR_BACKEND_TIMEOUT,

    PROXY_ERROR_CACHE_FAILURE,

    PROXY_ERROR_SSL_HANDSHAKE_FAILED,

    PROXY_ERROR_CLIENT_DISCONNECTED,

    PROXY_ERROR_MEMORY_EXHAUSTED,

    PROXY_ERROR_CONFIG_INVALID

} ProxyErrorCode;


typedef struct {

    ProxyErrorCode code;

    char message[512];

    char component[64];

    time_t timestamp;

    uint64_t request_id;

    Connection* failed_connection;

    BackendServer* failed_backend;

} ErrorContext;


typedef struct {

    ErrorContext* errors[1024];

    size_t error_count;

    size_t error_capacity;

    pthread_mutex_t error_mutex;

    pthread_t recovery_thread;

    bool running;

} ErrorHandler;


// Central error handling initialization
```

```
ErrorHandler* error_handler_create();

void error_handler_start(ErrorHandler* handler);

void error_handler_report(ErrorHandler* handler, ErrorContext* error);

void error_handler_stop(ErrorHandler* handler);

void error_handler_destroy(ErrorHandler* handler);
```

Resource Cleanup Macros

To prevent resource leaks during error conditions, implement cleanup macros that ensure consistent resource management:

```c
// utils/cleanup_macros.h

#define CLEANUP_BUFFER(buf) do { \

    if (buf) { \

        buffer_destroy(buf); \

        buf = NULL; \

    } \

} while(0)


#define CLEANUP_CONNECTION(conn_mgr, conn) do { \

    if (conn) { \

        connection_manager_close_connection(conn_mgr, conn); \

        conn = NULL; \

    } \

} while(0)


#define CLEANUP_HTTP_REQUEST(req) do { \

    if (req) { \

        http_request_destroy(req); \

        req = NULL; \

    } \

} while(0)


#define CLEANUP_BACKEND_CONNECTION(conn_mgr, conn, keep_alive) do { \

    if (conn) { \

        connection_manager_release_backend(conn_mgr, conn, keep_alive); \

        conn = NULL; \

    } \

} while(0)
```

Error Recovery Skeleton

The core error recovery logic coordinates between components to implement graceful degradation:

```c
// error/recovery_manager.c

typedef struct {
    LoadBalancer* load_balancer;
    ConnectionManager* conn_manager;
    CacheEngine* cache_engine;
    SSLTermination* ssl_termination;
    pthread_mutex_t recovery_mutex;
} RecoveryManager;

RecoveryManager* recovery_manager_create(LoadBalancer* lb, ConnectionManager* cm,
                                         CacheEngine* cache, SSLTermination* ssl) {
    // TODO 1: Allocate RecoveryManager structure
    // TODO 2: Initialize component references
    // TODO 3: Initialize recovery_mutex
    // TODO 4: Set up recovery state tracking
    // Hint: Store references to all components for coordinated recovery
}

void recovery_manager_handle_backend_failure(RecoveryManager* mgr, BackendServer* backend,
                                             ProxyErrorCode error) {
    // TODO 1: Lock recovery_mutex to ensure atomic recovery actions
    // TODO 2: Mark backend as unhealthy in load balancer
    // TODO 3: Invalidate all pooled connections to failed backend
    // TODO 4: Clear cache entries that originated from failed backend
    // TODO 5: Schedule health check retry with exponential backoff
    // TODO 6: Update failure statistics and monitoring metrics
    // TODO 7: Release recovery_mutex
    // Hint: Use loadbalancer_mark_backend_unhealthy(), connection_manager_invalidate_backend_pool()
}

void recovery_manager_handle_ssl_failure(RecoveryManager* mgr, ProxyErrorCode error,
                                         char* hostname) {
    // TODO 1: Determine if failure affects specific hostname or all SSL contexts
```

```
    // TODO 2: If certificate expired, schedule immediate certificate reload
    // TODO 3: If SNI failure, add missing certificate to reload queue
    // TODO 4: If cipher negotiation failure, log client compatibility issue
    // TODO 5: Update SSL failure statistics for monitoring
    // Hint: Use ssl_termination_schedule_reload() for certificate issues
}

void recovery_manager_handle_cache_failure(RecoveryManager* mgr, ProxyErrorCode error,
                                           char* cache_key) {
    // TODO 1: Determine scope of cache failure (single entry vs entire cache)
    // TODO 2: If memory exhaustion, trigger emergency cache eviction
    // TODO 3: If corruption detected, invalidate affected cache entries
    // TODO 4: If lock contention, implement cache operation timeouts
    // TODO 5: Enable cache bypass mode if failures exceed threshold
    // TODO 6: Schedule cache subsystem restart if necessary
    // Hint: Use cache_engine_invalidate() and cache_engine_clear() for cleanup
}
```

Timeout Management Implementation

Centralized timeout management prevents resource exhaustion and provides consistent timeout behavior:

```c
// utils/timeout_manager.c

typedef struct {

    Connection* connection;

    time_t timeout_time;

    ProxyErrorCode timeout_error;

    void (*timeout_callback)(Connection*, ProxyErrorCode);

} TimeoutEntry;


typedef struct {

    TimerWheel* timer_wheel;

    pthread_t timeout_thread;

    bool running;

} TimeoutManager;


TimeoutManager* timeout_manager_create(time_t granularity) {

    // TODO 1: Allocate TimeoutManager structure

    // TODO 2: Create timer wheel with specified granularity

    // TODO 3: Initialize timeout processing thread

    // TODO 4: Set running flag to true

    // Hint: Use timer_wheel_create() with appropriate slot count and duration

}


void timeout_manager_add_connection(TimeoutManager* mgr, Connection* conn,

                                    int timeout_ms, ProxyErrorCode error_code) {

    // TODO 1: Calculate absolute timeout time from current time + timeout_ms

    // TODO 2: Create TimeoutEntry with connection and error code

    // TODO 3: Add timeout entry to timer wheel at calculated time slot

    // TODO 4: Store timer wheel reference in connection for later removal

    // Hint: Use time(NULL) + (timeout_ms / 1000) for timeout calculation

}


void* timeout_processing_thread(void* arg) {

    TimeoutManager* mgr = (TimeoutManager*)arg;
```

```
    while (mgr->running) {
        // TODO 1: Sleep for timer wheel granularity period
        // TODO 2: Process current timer wheel slot for expired timeouts
        // TODO 3: For each expired timeout, call timeout callback
        // TODO 4: Advance timer wheel to next slot
        // TODO 5: Handle any timer wheel errors or corruption
        // Hint: Use timer_wheel_process_current_slot() and timer_wheel_advance()
    }
    return NULL;
}
```

Milestone Checkpoint: Error Handling Verification

After implementing error handling infrastructure, verify the system correctly detects and recovers from common failure scenarios:

1. **Backend Failure Testing**: Start the proxy with multiple backend servers, then shut down one backend. Verify that:
   - The load balancer immediately stops selecting the failed backend
   - In-flight requests to the failed backend return appropriate error codes
   - New requests distribute only among healthy backends
   - Health checks continue and detect when the backend recovers
2. **Client Disconnection Testing**: Start a request but disconnect the client before completion. Verify that:
   - The connection is properly cleaned up within 1 second
   - The backend request is canceled if not yet sent
   - No file descriptors or memory leak occurs
   - Backend connections are returned to the pool if applicable
3. **SSL Certificate Failure Testing**: Configure SSL termination with an expired certificate. Verify that:
   - HTTPS connections are refused rather than served with expired certificates
   - Certificate reload functionality works when valid certificates are provided
   - SNI continues working for other valid certificates
   - Error logs contain appropriate security alert information
4. **Resource Exhaustion Testing**: Send requests at high rate to exhaust connection pools. Verify that:
   - New requests receive HTTP 503 responses rather than hanging indefinitely
   - Connection pool recovery occurs when load decreases
   - No file descriptor leaks occur during exhaustion periods
   - Error monitoring correctly reports resource exhaustion conditions

Debugging Error Handling Issues

Symptom	Likely Cause	Diagnosis	Fix
File descriptor leaks during errors	Missing cleanup in error paths	Use `lsof -p <pid>` to track FD growth	Add CLEANUP_CONNECTION macros to all error paths
Requests hang during backend failures	Missing timeout on backend operations	Check connection state, look for FORWARDING state persistence	Implement timeout_manager_add_connection() for all backend operations
Error responses contain internal details	No error sanitization	Review error response content in logs	Implement generate_client_error_response() with message filtering
Recovery actions cause more failures	Cascade failure amplification	Monitor error rates during recovery periods	Add exponential backoff and circuit breaker patterns
Components have inconsistent failure state	Async error propagation issues	Compare backend health state across load balancer and connection manager	Implement centralized error reporting with proxy_handle_error()

The error handling implementation forms the foundation for a robust reverse proxy that maintains service availability during various failure conditions while protecting against security vulnerabilities and resource exhaustion.

Testing Strategy

> **Milestone(s):** All milestones - comprehensive testing ensures correct implementation of HTTP proxy core (Milestone 1), load balancing (Milestone 2), connection pooling (Milestone 3), caching (Milestone 4), and SSL termination (Milestone 5).

Think of testing a reverse proxy as conducting a full orchestra rehearsal. Just as a conductor must verify that each section plays correctly in isolation (violins, brass, percussion) and then ensure they harmonize together during the complete symphony, we need to test individual components independently and verify their coordinated behavior in realistic scenarios. A single missed note in one section can disrupt the entire performance, just as a bug in HTTP parsing or connection pooling can cascade through the entire request processing pipeline.

The testing strategy for our reverse proxy follows a three-tiered approach that mirrors how complex distributed systems are validated in production environments. We start with **unit testing** to verify individual component behavior in isolation, progress to **integration testing** to validate component interactions and data flow, and culminate with **milestone verification checkpoints** that simulate real-world usage patterns. Each tier builds confidence in different aspects of system correctness while providing increasingly realistic validation scenarios.

Unit Testing Approach

Unit testing for a reverse proxy focuses on isolating each component's core logic while carefully mocking external dependencies like network I/O, file system operations, and time-dependent behavior. Think of unit testing as examining each instrument in our orchestra separately - we want to verify that the violin section can play their parts correctly before worrying about how they blend with the brass section.

The foundation of effective unit testing lies in **dependency injection** and **interface abstraction**. Each component receives its dependencies through constructor parameters rather than creating them directly, allowing tests to substitute mock implementations that behave predictably. This approach enables us to test error scenarios that would be difficult to reproduce with real network connections, such as partial reads, connection timeouts, or SSL handshake failures.

HTTP Parser Unit Testing Strategy:

The `HttpParser` component requires particularly thorough unit testing due to its responsibility for correctly interpreting the HTTP protocol specification. Tests must verify parsing behavior across all `HttpParserState` transitions, handle malformed input gracefully, and correctly process edge cases like chunked transfer encoding and keep-alive connections.

Test Category	Test Cases	Input Data	Expected Behavior
Request Line Parsing	Valid methods, URIs, versions	`GET /path HTTP/1.1\r\n`	State transitions to `HTTP_PARSING_HEADERS`
Header Parsing	Standard headers, custom headers	`Content-Type: application/json\r\n`	Headers stored in `HashTable`
Content-Length Body	Fixed-length request body	`Content-Length: 5\r\n\r\nhello`	Body buffer contains exact bytes
Chunked Encoding	Variable-sized chunks	`5\r\nhello\r\n0\r\n\r\n`	Assembled body without chunk markers
Malformed Input	Invalid syntax, oversized headers	`INVALID REQUEST LINE`	State transitions to `HTTP_PARSING_ERROR`
Incremental Parsing	Partial packet delivery	Split input across multiple calls	Maintains state across `http_parser_process` calls

The parser testing framework uses **mock buffers** that simulate network packet boundaries, ensuring the parser correctly handles scenarios where HTTP messages arrive in fragments. Tests inject specific byte sequences at precise boundaries to verify state machine robustness.

```c
// Mock buffer simulation for incremental parsing tests
typedef struct {
    char* fragments[16];     // Simulated network packets
    size_t fragment_sizes[16];
    size_t fragment_count;
    size_t current_fragment;
} MockNetworkBuffer;
```

Connection Manager Unit Testing Strategy:

Testing the `ConnectionManager` requires sophisticated mocking of the event-driven I/O subsystem. Since the connection manager coordinates between epoll events, timer wheel timeouts, and connection pool operations, tests must carefully control the timing and ordering of these interactions.

Component Under Test	Mock Dependencies	Test Focus
`connection_manager_accept_client`	Mock socket file descriptors	Verify `Connection` state initialization
`connection_manager_acquire_backend`	Mock `ConnectionPool`	Test connection reuse vs. new connection logic
`connection_manager_handle_event`	Mock epoll events	Verify state transitions for `ConnectionState`
`timeout_processor`	Mock `TimerWheel`	Test timeout detection and connection cleanup
`connection_manager_release_backend`	Mock pool statistics	Verify connection return to pool logic

The connection manager tests use **synthetic epoll events** generated by test harnesses rather than real network I/O. This approach allows tests to precisely control event ordering and timing without depending on external network conditions or introducing non-deterministic behavior.

Load Balancer Unit Testing Strategy:

Load balancer testing focuses on algorithm correctness and health check behavior under various backend availability scenarios. The key challenge is testing concurrent access patterns while maintaining deterministic test outcomes.

Algorithm	Test Scenarios	Expected Distribution	Verification Method
`LB_ROUND_ROBIN`	3 healthy backends, 100 requests	33/33/34 requests per backend	Track `rr_current_index` progression
`LB_LEAST_CONNECTIONS`	Backends with 0, 5, 10 connections	New requests to backend with 0	Verify `connection_counts` array
`LB_WEIGHTED_ROUND_ROBIN`	Weights 1:2:3, 60 requests	10/20/30 requests per backend	Track `current_weights` calculations
`LB_IP_HASH`	Same client IP, 10 requests	All requests to same backend	Hash consistency verification

Backend health checking tests use **controlled failure injection** to simulate network timeouts, connection refused errors, and slow response scenarios. Tests advance mock timers to trigger health check intervals without waiting for real time to pass.

Cache Engine Unit Testing Strategy:

Cache testing requires careful attention to timing precision and memory management. Tests must verify cache-control header parsing, TTL calculations, LRU eviction behavior, and concurrent access patterns across multiple threads.

Cache Operation	Test Scenarios	Cache State Verification
`cache_engine_lookup`	Hit, miss, expired entry	Verify `CacheEntry` retrieval
`cache_engine_store`	Cacheable, non-cacheable responses	Check `CacheStats` updates
`lru_list_move_to_head`	Access existing entry	Verify LRU order maintenance
`cache_control_parse`	Various Cache-Control directives	Validate `CacheControl` structure
Cache eviction	Fill cache beyond size limit	Verify LRU entry removal
TTL expiration	Advance mock time	Verify expired entry cleanup

Cache tests use **deterministic time mocking** to control TTL calculations and expiration behavior. Mock time functions replace system calls to `time()` and `clock_gettime()`, allowing tests to "fast forward" through cache lifetimes without actual delays.

SSL Termination Unit Testing Strategy:

SSL component testing requires sophisticated OpenSSL mocking since real certificate operations involve file system access and cryptographic computations. Tests focus on certificate loading, SNI callback behavior, and TLS context management.

SSL Function	Mock Strategy	Test Focus
`ssl_utils_load_certificate`	Mock certificate file content	Verify `CertificateInfo` parsing
`ssl_termination_sni_callback`	Mock SNI hostnames	Test context selection logic
`ssl_utils_verify_key_cert_match`	Controlled key/cert pairs	Verify cryptographic validation
Certificate reload	Mock file system notifications	Test dynamic certificate updates

SSL tests use **synthetic certificates** generated at test startup rather than loading real certificate files. This approach eliminates file system dependencies while providing controlled certificate properties for testing edge cases.

⚠️ **Pitfall: Testing with Real Network I/O** Many developers attempt to test connection manager logic using actual TCP sockets and network communication. This approach introduces non-deterministic timing behavior, makes tests fragile to network conditions, and requires complex test environment setup. Instead, use mock file descriptors and synthetic events that provide deterministic, controllable test conditions.

⚠️ **Pitfall: Ignoring Parser State Persistence** HTTP parser tests often focus only on complete, well-formed messages without testing incremental parsing behavior. Real network traffic arrives in arbitrary packet boundaries, so tests must verify that parser state correctly persists across multiple `http_parser_process` calls with fragmented input data.

Integration Testing

Integration testing validates the **orchestration** between components, ensuring that data flows correctly through the complete request processing pipeline. Think of integration testing as rehearsing sections of our orchestra together - we need to verify that when the violins finish their phrase, the brass section enters at precisely the right moment with the correct dynamics.

Integration tests operate at the **component boundary level**, using real implementations for the components under test while carefully controlling their external dependencies. Unlike unit tests that isolate individual functions, integration tests verify that components correctly implement their **interface contracts** and handle the asynchronous message passing that coordinates request processing.

Parser-Connection Manager Integration:

The integration between `HttpParser` and `ConnectionManager` centers on the **stream-based parsing contract**. The connection manager feeds incoming network data to the parser incrementally, while the parser signals completion states that drive connection state transitions.

Integration Scenario	Test Setup	Validation Points
Complete request parsing	Send well-formed HTTP request	Verify `HTTP_PARSING_COMPLETE` triggers backend forwarding
Partial request arrival	Send request in multiple fragments	Verify parser maintains state across `connection_manager_handle_event` calls
Parse error handling	Send malformed HTTP data	Verify connection transitions to `CONNECTION_CLOSING`
Keep-alive detection	Send `Connection: keep-alive` header	Verify connection remains in pool after response
Pipeline request handling	Send multiple pipelined requests	Verify correct request boundary detection

The test harness uses **mock TCP streams** that deliver data at controlled intervals, simulating realistic network packet arrival patterns. Tests verify that connection state transitions occur at precisely the correct moments relative to parser state changes.

```c
// Integration test data structure for controlled packet delivery
typedef struct {
    char* packet_data;
    size_t packet_size;
    int delivery_delay_ms;
    bool should_trigger_event;
} MockPacket;

typedef struct {
    MockPacket* packets;
    size_t packet_count;
    size_t current_packet;
    ConnectionManager* conn_mgr;
    HttpParser* parser;
} ParserConnectionIntegrationTest;
```

Load Balancer-Connection Manager Integration:

This integration validates the **backend selection and connection acquisition workflow**. The load balancer selects an appropriate backend server, while the connection manager either reuses an existing pooled connection or establishes a new connection to that backend.

Integration Workflow	Test Configuration	Validation Criteria
Backend selection with pool hit	3 backends, existing pooled connections	Verify `connection_manager_acquire_backend` reuses connection
Backend selection with pool miss	Backends with empty connection pools	Verify new connection establishment
Backend failure during acquisition	Temporarily unreachable backend	Verify failover to alternate backend
Health check triggered pool eviction	Backend marked unhealthy	Verify pooled connections removed
Weighted selection with pool status	Different pool utilization per backend	Verify selection considers both weight and availability

Integration tests use **network namespace isolation** to simulate backend server behavior without requiring actual backend processes. Test backends respond with predictable HTTP responses and can be configured to simulate various failure modes.

Cache Engine-HTTP Parser Integration:

Cache integration testing verifies **conditional request generation** and **cache-control header interpretation**. The cache engine must correctly parse HTTP headers to determine cacheability, generate appropriate cache keys, and create conditional requests for cache validation.

Cache Integration Scenario	Request Properties	Expected Cache Behavior
Fresh cache hit	Cached response within TTL	Return cached response without backend query
Stale cache validation	Cached response beyond TTL with ETag	Generate `If-None-Match` conditional request
Cache miss	No cached entry for request	Forward to backend and cache response
Non-cacheable response	Response with `Cache-Control: no-store`	Forward response without caching
Vary header handling	Response with `Vary: Accept-Encoding`	Include Accept-Encoding in cache key

Cache integration tests use **mock HTTP responses** with controlled cache-control headers and known ETags. Tests verify that cache key generation produces consistent results and that conditional requests contain the correct validation headers.

SSL Termination-Connection Manager Integration:

SSL integration testing focuses on the **TLS handshake coordination** and the **transition from encrypted client communication to plaintext backend forwarding**. This integration is particularly complex due to the asynchronous nature of TLS handshake processing.

SSL Integration Case	TLS Configuration	Validation Points
Successful handshake	Valid certificate, strong ciphers	Verify plaintext HTTP extraction
SNI-based certificate selection	Multiple certificates loaded	Verify correct certificate chosen
Client certificate validation	Mutual TLS configuration	Verify client certificate verification
TLS handshake failure	Invalid certificate configuration	Verify graceful connection termination
Mixed HTTP/HTTPS handling	Both ports listening	Verify protocol-appropriate handling

SSL integration tests use **self-signed test certificates** generated during test setup with known properties. Tests simulate various client TLS implementations to verify compatibility across different TLS library versions and cipher suite preferences.

End-to-End Request Flow Integration:

The most comprehensive integration tests trace **complete request processing pipelines** that exercise all major components in realistic sequences. These tests simulate actual reverse proxy usage patterns with multiple concurrent clients and backend servers.

End-to-End Scenario	Test Configuration	Component Interactions
Cached response serving	Cache hit with fresh entry	SSL → Parser → Cache (hit) → Response
Load-balanced backend request	Cache miss, multiple backends	SSL → Parser → Cache (miss) → LB → Backend → Response
Backend failover with retry	Primary backend failure	SSL → Parser → LB → Failure → LB (retry) → Backend → Response
SSL certificate reload	Certificate update during traffic	SSL (reload) → Continue serving with new cert
Connection pooling optimization	Multiple requests to same backend	Verify connection reuse across requests

End-to-end tests use **docker-compose environments** with real backend HTTP servers, allowing tests to verify behavior against actual HTTP implementations rather than mocks. This approach catches integration issues that might be missed with entirely synthetic test environments.

⚠️ **Pitfall: Over-Mocking in Integration Tests** Integration tests lose their value when too many components are mocked. The goal is to test real component interactions, so only external dependencies (like actual network calls) should be mocked. Using real component implementations reveals interface mismatches and timing issues that unit tests cannot detect.

⚠️ **Pitfall: Ignoring Asynchronous Timing** Many integration test failures occur due to race conditions between components that communicate asynchronously. Tests must either use synchronization primitives to control component interaction timing or employ polling strategies that wait for expected state changes rather than assuming immediate completion.

Milestone Verification Checkpoints

Milestone checkpoints provide **concrete validation criteria** that confirm each phase of implementation meets functional requirements before proceeding to the next milestone. Think of these checkpoints as **integration rehearsals** where we verify that newly implemented functionality works correctly both in isolation and in combination with previously completed components.

Each milestone checkpoint includes **automated test suites**, **manual verification procedures**, and **performance benchmarks** that validate both correctness and basic performance characteristics. The checkpoints are designed to catch integration issues early, before they compound with additional complexity from subsequent milestones.

Milestone 1: HTTP Proxy Core Verification

The HTTP proxy core checkpoint validates basic request forwarding functionality without load balancing, caching, or SSL termination. This foundational milestone must demonstrate rock-solid HTTP protocol handling and connection management before adding additional complexity.

Verification Area	Test Method	Success Criteria
HTTP/1.1 request forwarding	Automated test suite	100% of well-formed requests forwarded correctly
HTTP header preservation	Header comparison tests	All client headers preserved in backend requests
Response body integrity	Binary content verification	Byte-for-byte response accuracy
Connection state management	Connection lifecycle tests	No resource leaks after 1000 requests
Error handling	Malformed request tests	Appropriate error responses for invalid input
Keep-alive support	Connection reuse verification	Multiple requests over single client connection

Automated Test Commands:

```bash
# Run core HTTP proxy tests
make test-milestone-1
./test/integration/http_proxy_core_test

# Performance baseline test
./test/performance/proxy_benchmark -requests=1000 -concurrency=10
```

Manual Verification Procedures: The manual verification uses `curl` commands to test realistic usage scenarios that automated tests might miss:

1. **Basic Request Forwarding Test:**
   - Start proxy server: `./reverse_proxy -config=test.conf -backends=localhost:8080`
   - Start test backend: `python3 -m http.server 8080`
   - Test GET request: `curl -v http://localhost:3128/test.html`
   - Verify response matches direct backend access
   - Check proxy logs show request forwarding
2. **Header Manipulation Verification:**
   - Send request with custom headers: `curl -H "X-Test: value" http://localhost:3128/`
   - Verify backend receives `X-Forwarded-For` and `Via` headers
   - Confirm custom client headers preserved
3. **HTTP Method Support:**
   - Test POST: `curl -X POST -d "data" http://localhost:3128/api/test`
   - Test PUT: `curl -X PUT -d @file.json http://localhost:3128/upload`
   - Test DELETE: `curl -X DELETE http://localhost:3128/resource/123`
4. **Error Condition Handling:**
   - Stop backend server, verify 502 Bad Gateway response
   - Send oversized request, verify 413 Request Entity Too Large
   - Send malformed HTTP, verify connection closes gracefully

Expected Performance Baseline:

- **Throughput:** 1,000 requests/second with 10 concurrent connections
- **Latency:** 95th percentile under 10ms for localhost backends
- **Memory Usage:** Stable under 50MB resident memory after 10,000 requests
- **Connection Handling:** Support at least 100 concurrent client connections

⚠️ **Milestone 1 Common Issues:**

- **Incomplete response forwarding:** Verify both headers and body are completely transmitted
- **Connection hang on backend failure:** Ensure timeouts are implemented for backend connections
- **Memory leaks in request processing:** Use valgrind to detect buffer management issues

Milestone 2: Load Balancing Verification

Load balancing verification confirms correct request distribution across multiple backend servers and validates health checking behavior under various failure scenarios.

Load Balancing Feature	Test Method	Success Criteria
Round-robin distribution	Statistical analysis	Even distribution within 5% variance
Least-connections algorithm	Connection count tracking	Requests routed to least busy backend
Backend health checking	Controlled backend failures	Unhealthy backends removed within 30 seconds
Failover behavior	Sequential backend shutdown	Traffic redirected without client errors
Dynamic backend updates	Configuration reload	New backends available without restart
Weighted distribution	Capacity-based routing	Traffic distributed per weight ratios

Automated Test Commands:

```bash
# Run load balancing test suite
make test-milestone-2
./test/integration/load_balancer_test

# Multi-backend stress test
./test/load/multi_backend_test -backends=3 -requests=5000
```

Manual Verification Procedures:

1. **Round-Robin Distribution Test:**
   - Configure 3 backend servers on ports 8080, 8081, 8082
   - Send 30 requests: `for i in {1..30}; do curl http://localhost:3128/; done`
   - Check backend logs: each should receive ~10 requests
   - Verify request distribution is approximately even
2. **Health Check Validation:**
   - Start proxy with health checking enabled (30-second intervals)
   - Stop one backend server: `kill $BACKEND_PID`
   - Send requests and verify traffic routes only to healthy backends

- Restart backend and verify it rejoins rotation

3. **Weighted Load Balancing:**
   - Configure backends with weights 1:2:3
   - Send 60 requests and verify distribution: 10/20/30
   - Monitor `LoadBalancer` metrics for weight compliance
4. **Least-Connections Algorithm:**
   - Configure least-connections mode
   - Establish persistent connections to backends (different counts)
   - Send new requests and verify routing to least busy backend

Backend Health Check Configuration:

```ini
# test.conf load balancing section
[load_balancer]
algorithm = round_robin
health_check_interval = 30
health_check_timeout = 5
failure_threshold = 3
success_threshold = 2

[backends]
backend1 = host=localhost port=8080 weight=1
backend2 = host=localhost port=8081 weight=2
backend3 = host=localhost port=8082 weight=3
```

⚠️ **Milestone 2 Common Issues:**

- **Sticky connections disrupting distribution:** Ensure connection pooling doesn't bias routing
- **Health check false positives:** Verify health check timeout values are appropriate for network conditions
- **Race conditions in backend selection:** Check thread safety of backend state updates

Milestone 3: Connection Pooling Verification

Connection pooling checkpoint validates backend connection reuse, proper pool size management, and connection lifecycle handling under various load patterns.

Connection Pool Feature	Test Method	Success Criteria
Connection reuse	Pool utilization metrics	>80% connection reuse rate
Pool size limits	Concurrent connection test	Pool never exceeds configured maximum
Idle timeout handling	Connection aging test	Stale connections removed within timeout
Pool exhaustion handling	High concurrency test	Graceful handling when pool full
Health integration	Backend failure simulation	Broken connections purged from pool
Per-backend isolation	Multi-backend pool test	Separate pools maintained per backend

Automated Test Commands:

```bash
# Run connection pooling test suite
make test-milestone-3
./test/integration/connection_pool_test

# Pool exhaustion stress test
./test/stress/pool_exhaustion_test -max_pools=100 -concurrent=500
```

Manual Verification Procedures:

1. **Connection Reuse Validation:**
   - Configure small pool size (max 5 connections per backend)
   - Send 50 sequential requests to same backend
   - Monitor pool metrics: should show high reuse percentage
   - Check backend: should see only 5 TCP connections established
2. **Pool Limit Enforcement:**
   - Configure pool with max_connections=3
   - Generate 10 concurrent requests
   - Verify only 3 backend connections established
   - Additional requests should queue or use alternate backends
3. **Idle Timeout Testing:**
   - Send requests to establish pooled connections
   - Wait for idle_timeout period (configured in test.conf)
   - Send new request and verify fresh connection establishment
   - Check pool metrics show connection eviction occurred
4. **Pool Health Integration:**
   - Establish connections in pool
   - Force backend restart (simulates network partition)
   - Verify broken connections detected and removed
   - New requests should establish fresh connections

Connection Pool Metrics to Monitor:

```c
// Expected pool statistics after testing
typedef struct {
    size_t total_connections_created;    // Should be minimized via reuse
    size_t current_active_connections;   // Should stay within limits
    size_t current_idle_connections;     // Should reflect pool utilization
    uint64_t connection_reuse_count;     // Should be high relative to requests
    uint64_t pool_exhaustion_events;     // Should be zero under normal load
    time_t last_cleanup_time;            // Should advance with idle timeouts
} PoolMetrics;
```

⚠️ **Milestone 3 Common Issues:**

- **Connection leaks:** Ensure connections are properly returned to pool after request completion
- **Deadlocks in pool access:** Verify thread-safe pool operations under high concurrency
- **Stale connection reuse:** Implement connection validation before reuse from pool

Milestone 4: Caching Verification

Caching verification confirms correct HTTP caching behavior, cache-control header compliance, and cache performance characteristics under various response types and cache sizes.

Cache Feature	Test Method	Success Criteria
Cache hit serving	Duplicate request test	Second request served from cache
TTL expiration	Time-based cache test	Expired entries trigger backend requests
Cache-Control compliance	Header parsing test	`no-cache`, `no-store` directives respected
ETag validation	Conditional request test	`If-None-Match` requests generate 304 responses
LRU eviction	Cache overflow test	Oldest entries evicted when cache full
Vary header handling	Content negotiation test	Accept-Encoding included in cache keys

Automated Test Commands:

```bash
# Run caching test suite
make test-milestone-4
./test/integration/cache_engine_test

# Cache performance benchmark
./test/performance/cache_benchmark -cache_size=100MB -requests=10000
```

Manual Verification Procedures:

1. **Basic Cache Hit Testing:**

- Clear cache: `curl -X DELETE http://localhost:3128/admin/cache`
- Send request: `curl -v http://localhost:3128/static/image.jpg`
- Note backend access in logs and response headers
- Repeat request: should show cache hit with `X-Cache-Status: HIT`

2. **Cache-Control Compliance:**
   - Test cacheable response: `curl http://localhost:3128/api/data` (with `Cache-Control: max-age=300`)
   - Test non-cacheable: `curl http://localhost:3128/api/user` (with `Cache-Control: no-store`)
   - Verify only first response gets cached
3. **TTL Expiration Validation:**
   - Configure short TTL in test backend responses (`max-age=5`)
   - Send request and verify cache storage
   - Wait 10 seconds, repeat request
   - Verify backend access occurs due to expiration
4. **Conditional Request Testing:**
   - Send request to backend that supports ETags
   - Verify cache stores ETag value
   - After expiration, verify proxy sends `If-None-Match`
   - Backend should respond with 304, cache should serve original response

Cache Metrics Validation:

```c
// Expected cache statistics after testing
typedef struct {
    uint64_t hit_count;          // Should increase with duplicate requests
    uint64_t miss_count;         // Should equal unique cacheable requests
    uint64_t eviction_count;     // Should remain low until cache full
    size_t current_size;         // Should not exceed configured limit
    double hit_ratio;            // Should be >50% with duplicate requests
} CachePerformanceMetrics;
```

⚠️ **Milestone 4 Common Issues:**

- **Caching non-cacheable responses:** Verify POST requests and private responses excluded
- **Incorrect cache key generation:** Ensure Vary headers properly included in cache keys
- **Memory leaks in cache entries:** Verify proper cleanup of evicted cache entries

Milestone 5: SSL Termination Verification

SSL termination checkpoint validates HTTPS request handling, certificate management, SNI support, and the transition from encrypted client connections to plaintext backend communication.

SSL Feature	Test Method	Success Criteria
TLS handshake completion	HTTPS client test	Successful connection establishment
Certificate validation	Certificate chain test	Valid certificates accepted, invalid rejected
SNI hostname selection	Multi-domain test	Correct certificate chosen per hostname
Cipher suite negotiation	TLS compatibility test	Strong ciphers preferred, weak ciphers rejected
HTTP to HTTPS redirect	Mixed protocol test	HTTP requests redirected to HTTPS
Certificate reload	Dynamic cert update	New certificates loaded without restart

Automated Test Commands:

```bash
# Run SSL termination test suite
make test-milestone-5
./test/integration/ssl_termination_test

# SSL compatibility test across TLS versions
./test/ssl/tls_compatibility_test -min_version=TLS1.2
```

Manual Verification Procedures:

1. **Basic HTTPS Request Processing:**
   - Generate test certificate: `openssl req -x509 -newkey rsa:2048 -keyout test.key -out test.crt`
   - Configure SSL termination with test certificate
   - Test HTTPS request: `curl -k https://localhost:3129/test.html`
   - Verify response received and backend accessed via HTTP
2. **SNI Multi-Domain Testing:**
   - Configure certificates for multiple domains
   - Test domain1: `curl -k --resolve domain1.test:3129:127.0.0.1 https://domain1.test:3129/`
   - Test domain2: `curl -k --resolve domain2.test:3129:127.0.0.1 https://domain2.test:3129/`
   - Verify correct certificates selected via OpenSSL inspection
3. **TLS Version and Cipher Validation:**
   - Test minimum TLS version: `curl --tlsv1.2 -k https://localhost:3129/`
   - Verify weak TLS rejected: `curl --tlsv1.0 https://localhost:3129/` (should fail)
   - Check cipher selection: `openssl s_client -connect localhost:3129 -cipher HIGH`
4. **Certificate Reload Testing:**
   - Start proxy with initial certificate
   - Update certificate files on disk
   - Send reload signal or API call
   - Verify new certificate used for subsequent connections

SSL Configuration Validation:

```ini
# test.conf SSL section
[ssl]
enabled = true
cert_path = /etc/ssl/test.crt
key_path = /etc/ssl/test.key
min_tls_version = TLS_1_2
cipher_list = ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20:!aNULL:!MD5:!DSS
```

SSL Metrics to Monitor:

```c
// SSL termination performance statistics

typedef struct {

    uint64_t handshakes_completed;       // Should equal HTTPS requests

    uint64_t handshake_failures;         // Should remain low

    uint64_t certificate_reloads;        // Should match reload operations

    double avg_handshake_time_ms;        // Should be <100ms for RSA certificates

    uint32_t active_ssl_connections;     // Should not exceed connection limits

} SSLTerminationMetrics;
```

⚠️ **Milestone 5 Common Issues:**

- **Certificate/key mismatch:** Verify certificate and private key correspond via OpenSSL
- **Weak cipher acceptance:** Ensure cipher suite configuration rejects deprecated algorithms
- **Memory leaks in SSL contexts:** Properly cleanup SSL_CTX structures during certificate reload

Implementation Guidance

The testing strategy implementation requires careful coordination between test frameworks, mock systems, and build automation. The following guidance provides concrete tools and patterns for implementing the testing approach described above.

A. Technology Recommendations:

Testing Component	Simple Option	Advanced Option
Unit Test Framework	Custom assert macros with C	Google Test (C++) or Unity (C)
Mock System	Manual dependency injection	CMock or FFF (Fake Function Framework)
Integration Testing	Custom test harnesses	Testcontainers with Docker
HTTP Test Clients	curl + shell scripts	Custom HTTP client library
Performance Testing	Simple timing loops	wrk or Apache Bench (ab)
SSL Testing	OpenSSL command line tools	custom SSL test client

B. Recommended File Structure:

```
reverse-proxy/
├── src/                          # Source code
│   ├── http_parser/
│   ├── connection_manager/
│   ├── load_balancer/
│   ├── cache_engine/
│   └── ssl_termination/
├── test/                         # All testing code
│   ├── unit/                     # Unit tests
│   │   ├── test_http_parser.c
│   │   ├── test_connection_manager.c
│   │   ├── test_load_balancer.c
│   │   ├── test_cache_engine.c
│   │   └── test_ssl_termination.c
│   ├── integration/              # Integration tests
│   │   ├── test_parser_connection.c
│   │   ├── test_lb_connection.c
│   │   ├── test_cache_integration.c
│   │   └── test_ssl_integration.c
│   ├── milestone/                # Milestone checkpoints
│   │   ├── milestone1_http_core.c
│   │   ├── milestone2_load_balancing.c
│   │   ├── milestone3_connection_pooling.c
│   │   ├── milestone4_caching.c
│   │   └── milestone5_ssl_termination.c
│   ├── mocks/                    # Mock implementations
│   │   ├── mock_network.c
│   │   ├── mock_timer.c
│   │   └── mock_ssl.c
│   ├── fixtures/                 # Test data and configurations
│   │   ├── test_certificates/
│   │   ├── sample_http_requests/
│   │   └── test_configs/
│   └── scripts/                  # Test automation
│       ├── run_unit_tests.sh
│       ├── run_integration_tests.sh
│       └── milestone_verification.sh
├── tools/                        # Development tools
│   ├── test_http_server.c        # Simple backend for testing
│   └── ssl_cert_generator.sh     # Generate test certificates
└── Makefile                      # Build and test automation
```

C. Infrastructure Starter Code (COMPLETE):

Mock Network Interface (`test/mocks/mock_network.c`):

```c
#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include "mock_network.h"


// Mock network implementation for testing without real sockets

typedef struct {

    char* buffer_data;

    size_t buffer_size;

    size_t read_position;

    size_t write_position;

    bool closed;

    int mock_errno;

} MockSocket;


static MockSocket* mock_sockets[1024] = {0};

static int next_mock_fd = 1000;


int mock_socket_create(size_t buffer_size) {

    int fd = next_mock_fd++;

    MockSocket* sock = malloc(sizeof(MockSocket));

    sock->buffer_data = malloc(buffer_size);

    sock->buffer_size = buffer_size;

    sock->read_position = 0;

    sock->write_position = 0;

    sock->closed = false;

    sock->mock_errno = 0;

    mock_sockets[fd - 1000] = sock;

    return fd;

}


ssize_t mock_read(int fd, void* buf, size_t count) {

    MockSocket* sock = mock_sockets[fd - 1000];

    if (!sock || sock->closed) {
```

```
        errno = EBADF;
        return -1;
    }

    if (sock->mock_errno != 0) {
        errno = sock->mock_errno;
        return -1;
    }

    size_t available = sock->write_position - sock->read_position;
    size_t to_read = (count < available) ? count : available;

    if (to_read == 0) {
        errno = EAGAIN;
        return -1;
    }

    memcpy(buf, sock->buffer_data + sock->read_position, to_read);
    sock->read_position += to_read;
    return to_read;
}

ssize_t mock_write(int fd, const void* buf, size_t count) {
    MockSocket* sock = mock_sockets[fd - 1000];
    if (!sock || sock->closed) {
        errno = EBADF;
        return -1;
    }

    if (sock->mock_errno != 0) {
        errno = sock->mock_errno;
        return -1;
```

```
    }

    size_t available = sock->buffer_size - sock->write_position;
    size_t to_write = (count < available) ? count : available;

    if (to_write == 0) {
        errno = EAGAIN;
        return -1;
    }

    memcpy(sock->buffer_data + sock->write_position, buf, to_write);
    sock->write_position += to_write;
    return to_write;
}

void mock_socket_inject_data(int fd, const char* data, size_t size) {
    MockSocket* sock = mock_sockets[fd - 1000];
    if (sock && sock->write_position + size <= sock->buffer_size) {
        memcpy(sock->buffer_data + sock->write_position, data, size);
        sock->write_position += size;
    }
}

void mock_socket_set_error(int fd, int error_code) {
    MockSocket* sock = mock_sockets[fd - 1000];
    if (sock) {
        sock->mock_errno = error_code;
    }
}

void mock_socket_close(int fd) {
    MockSocket* sock = mock_sockets[fd - 1000];
    if (sock) {
```

```
        sock->closed = true;
        free(sock->buffer_data);
        free(sock);
        mock_sockets[fd - 1000] = NULL;
    }
}
```

Test HTTP Server (`tools/test_http_server.c`):

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <signal.h>

#include <time.h>


// Simple HTTP server for integration testing

typedef struct {

    int port;

    int delay_ms;

    char* response_body;

    char* custom_headers;

    bool should_fail;

} TestServerConfig;


static volatile bool server_running = true;


void signal_handler(int sig) {

    server_running = false;

    printf("Test server shutting down...\n");

}


void send_http_response(int client_fd, TestServerConfig* config) {

    char response[4096];

    time_t now = time(NULL);



    if (config->should_fail) {

        // Simulate server error

        close(client_fd);

        return;

    }
```

```
    if (config->delay_ms > 0) {
        usleep(config->delay_ms * 1000);
    }

    snprintf(response, sizeof(response),
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/plain\r\n"
        "Content-Length: %zu\r\n"
        "Cache-Control: max-age=300\r\n"
        "ETag: \"test-etag-%ld\"\r\n"
        "%s"
        "\r\n"
        "%s",
        strlen(config->response_body),
        now,
        config->custom_headers ? config->custom_headers : "",
        config->response_body
    );

    send(client_fd, response, strlen(response), 0);
    close(client_fd);
}

int main(int argc, char* argv[]) {
    TestServerConfig config = {
        .port = 8080,
        .delay_ms = 0,
        .response_body = "Test server response",
        .custom_headers = NULL,
        .should_fail = false
    };
```

```
// Parse command line arguments
for (int i = 1; i < argc; i++) {
    if (strncmp(argv[i], "--port=", 7) == 0) {
        config.port = atoi(argv[i] + 7);
    } else if (strncmp(argv[i], "--delay=", 8) == 0) {
        config.delay_ms = atoi(argv[i] + 8);
    } else if (strncmp(argv[i], "--fail", 6) == 0) {
        config.should_fail = true;
    }
}

signal(SIGINT, signal_handler);
signal(SIGTERM, signal_handler);

int server_fd = socket(AF_INET, SOCK_STREAM, 0);
int opt = 1;
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(config.port);

bind(server_fd, (struct sockaddr*)&address, sizeof(address));
listen(server_fd, 10);

printf("Test HTTP server listening on port %d\n", config.port);

while (server_running) {
    fd_set read_fds;
    FD_ZERO(&read_fds);
```

```
        FD_SET(server_fd, &read_fds);


        struct timeval timeout = {.tv_sec = 1, .tv_usec = 0};
        int activity = select(server_fd + 1, &read_fds, NULL, NULL, &timeout);


        if (activity > 0 && FD_ISSET(server_fd, &read_fds)) {
            int client_fd = accept(server_fd, NULL, NULL);
            if (client_fd >= 0) {
                char buffer[1024];
                recv(client_fd, buffer, sizeof(buffer), 0); // Read request
                send_http_response(client_fd, &config);
            }
        }
    }


    close(server_fd);
    return 0;
}
```

D. Core Logic Skeleton Code:

Unit Test Template (`test/unit/test_http_parser.c`):

```c
#include "http_parser.h"
#include "test_framework.h"

// HTTP Parser unit test template
void test_http_parser_request_line_parsing() {
    // TODO 1: Create HttpParser instance using http_parser_create()
    // TODO 2: Prepare test input: "GET /path HTTP/1.1\r\n"
    // TODO 3: Call http_parser_process() with test input
    // TODO 4: Verify parser state transitions to HTTP_PARSING_HEADERS
    // TODO 5: Verify HttpRequest fields populated correctly (method, uri, version)
    // TODO 6: Clean up parser and request structures
    // Hint: Use mock_socket_inject_data() for controlled input delivery
}

void test_http_parser_chunked_encoding() {
    // TODO 1: Create parser and prepare chunked request body
    // TODO 2: Process input incrementally: "5\r\nhello\r\n0\r\n\r\n"
    // TODO 3: Verify state transitions: BODY -> CHUNKED_SIZE -> CHUNKED_DATA -> COMPLETE
    // TODO 4: Verify assembled body contains "hello" without chunk markers
    // TODO 5: Verify Content-Length calculation matches actual body
    // Hint: Test partial delivery of chunk size and data separately
}

void test_http_parser_malformed_input() {
    // TODO 1: Create parser and prepare invalid HTTP input
    // TODO 2: Test cases: invalid method, malformed headers, oversized requests
    // TODO 3: Verify parser transitions to HTTP_PARSING_ERROR state
    // TODO 4: Verify error handling doesn't crash or leak memory
    // TODO 5: Verify parser can be reset and reused after errors
    // Hint: Use valgrind to detect memory leaks in error paths
}

int main() {
    test_http_parser_request_line_parsing();
```

```
    test_http_parser_chunked_encoding();
    test_http_parser_malformed_input();


    printf("HTTP Parser unit tests completed\n");
    return 0;
}
```

Integration Test Template (`test/integration/test_cache_integration.c`):

```c
#include "cache_engine.h"
#include "http_parser.h"
#include "test_framework.h"

// Cache Engine integration test template
void test_cache_conditional_request_generation() {
    // TODO 1: Create CacheEngine and populate with test response (with ETag)
    // TODO 2: Create new HttpRequest for same resource after TTL expiry
    // TODO 3: Call cache_engine_lookup() and verify it returns stale entry
    // TODO 4: Verify cache_create_conditional() generates If-None-Match header
    // TODO 5: Simulate 304 Not Modified response from backend
    // TODO 6: Verify cache_update_from_304() refreshes entry TTL
    // Hint: Use mock time functions to control TTL calculations
}

void test_cache_vary_header_handling() {
    // TODO 1: Create responses with "Vary: Accept-Encoding" header
    // TODO 2: Store responses for same URL with different Accept-Encoding values
    // TODO 3: Verify cache_generate_key() includes Accept-Encoding in key
    // TODO 4: Verify cache lookups return correct response for each encoding
    // TODO 5: Verify cache doesn't serve gzip response for non-gzip request
    // Hint: Test with "gzip, deflate" vs "identity" Accept-Encoding values
}

void test_cache_size_limit_enforcement() {
    // TODO 1: Create CacheEngine with small size limit (e.g., 1MB)
    // TODO 2: Store responses until cache size approaches limit
    // TODO 3: Store additional response that exceeds limit
    // TODO 4: Verify LRU eviction removes oldest entries
    // TODO 5: Verify cache size stays within configured limit
    // TODO 6: Verify evicted entries are completely cleaned up
    // Hint: Monitor cache_engine_stats() for size and eviction metrics
}
```

```c
int main() {
    test_cache_conditional_request_generation();
    test_cache_vary_header_handling();
    test_cache_size_limit_enforcement();

    printf("Cache Engine integration tests completed\n");
    return 0;
}
```

E. Language-Specific Hints:

C Development Tips:

- Use `valgrind --leak-check=full` to detect memory leaks in connection pooling
- Use `gdb` with breakpoints to debug state machine transitions
- Compile with `-fsanitize=address` to detect buffer overflows
- Use `strace` to monitor actual system calls during network testing
- Use `tcpdump` to capture real network traffic for integration verification

Mock Time Management:

```c
// Override system time functions for deterministic testing
static time_t mock_current_time = 1640995200; // 2022-01-01 00:00:00

time_t time(time_t* tloc) {
    if (tloc) *tloc = mock_current_time;
    return mock_current_time;
}

void advance_mock_time(time_t seconds) {
    mock_current_time += seconds;
}
```

Thread Safety Testing:

```c
// Use pthread barriers to synchronize test threads

pthread_barrier_t test_barrier;


void setup_concurrent_test(int thread_count) {

    pthread_barrier_init(&test_barrier, NULL, thread_count + 1);

}


void wait_for_threads() {

    pthread_barrier_wait(&test_barrier); // All threads start simultaneously

}
```

F. Milestone Checkpoint Implementation:

Milestone 1 Checkpoint Script (`test/scripts/milestone1_verification.sh`):

```bash
#!/bin/bash
set -e

echo "=== Milestone 1: HTTP Proxy Core Verification ==="

# Start test backend
./tools/test_http_server --port=8080 &
BACKEND_PID=$!
sleep 2

# Start proxy server
./reverse_proxy --config=test/fixtures/milestone1.conf &
PROXY_PID=$!
sleep 3

# Test basic request forwarding
echo "Testing basic request forwarding..."
RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:3128/test)
if [ "$RESPONSE" != "200" ]; then
    echo "FAIL: Expected 200, got $RESPONSE"
    exit 1
fi

# Test header preservation
echo "Testing header preservation..."
curl -H "X-Test-Header: test-value" http://localhost:3128/headers > /tmp/response.txt
if ! grep -q "X-Forwarded-For" /tmp/response.txt; then
    echo "FAIL: X-Forwarded-For header missing"
    exit 1
fi

# Test error handling
echo "Testing error handling..."
kill $BACKEND_PID
sleep 1
```

```
RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:3128/test)
if [ "$RESPONSE" != "502" ]; then
    echo "FAIL: Expected 502 Bad Gateway, got $RESPONSE"
    exit 1
fi

# Cleanup
kill $PROXY_PID
echo "SUCCESS: Milestone 1 verification passed"
```

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Parser hangs on input	Infinite loop in state machine	Add debug prints in `http_parser_process`	Check state transition logic for missing break statements
Connection pool exhaustion	Connections not returned to pool	Monitor pool metrics and connection lifecycle	Ensure `connection_manager_release_backend` called
Cache hit ratio is 0%	Incorrect cache key generation	Log cache keys for identical requests	Verify `cache_generate_key` includes all relevant headers
SSL handshake fails	Certificate/key mismatch	Check certificate with `openssl x509 -text`	Verify certificate and key correspond with `ssl_utils_verify_key_cert_match`
Memory usage grows continuously	Resource leaks in error paths	Run with valgrind during error injection	Add proper cleanup in all error handling branches
Backend health checks failing	Network timing issues	Monitor health check request/response timing	Adjust health check timeout values in configuration

Debugging Guide

> **Milestone(s):** All milestones - comprehensive debugging knowledge is essential across HTTP proxy core (Milestone 1), load balancing (Milestone 2), connection pooling (Milestone 3), caching (Milestone 4), and SSL termination (Milestone 5).

Think of debugging a reverse proxy like being a detective investigating a complex crime scene. Unlike debugging a simple application where problems typically have one cause, reverse proxy issues often involve multiple moving parts: network connections, protocol parsing, thread synchronization, and distributed systems interactions. Each component can fail in subtle ways that cascade through the system, creating symptoms that appear far from their root cause. A connection timeout might stem from DNS resolution issues, SSL handshake failures, backend overload, or even a subtle HTTP parser bug that corrupts request headers.

The key insight for reverse proxy debugging is understanding the **request lifecycle dependencies**. Every HTTP request flows through multiple components in a specific sequence, and each component maintains state that can become corrupted. Unlike a

stateless function where inputs directly map to outputs, reverse proxy debugging requires understanding how connection state, cache entries, SSL contexts, and load balancer statistics interact across multiple threads and connections.

Common Bug Patterns

This section catalogs the most frequently encountered issues when building reverse proxies, organized by component and symptom. Each pattern includes the observable behavior, underlying cause, diagnostic steps, and resolution strategy.

HTTP Parser Component Issues

⚠️ Pitfall: Partial Request Buffering

The HTTP parser receives data in chunks from TCP sockets, but HTTP requests don't always arrive as complete units. A common mistake is assuming `recv()` calls return complete HTTP messages, leading to parser failures when requests span multiple network packets.

Symptom	Likely Cause	Diagnostic Steps	Resolution
Parser returns `HTTP_PARSING_ERROR` randomly	Treating partial data as complete requests	Log buffer contents before parsing, check if request headers end with `\r\n\r\n`	Implement proper buffering with `buffer_append()` until complete headers received
Missing request headers	Parser processes incomplete header section	Monitor `buffer_length` vs expected header size	Buffer data until double CRLF found before parsing
Request body truncated	Not handling `Content-Length` properly	Compare bytes read vs `Content-Length` header value	Continue reading until full body received based on `Content-Length`

> **Key Insight**: HTTP is a stream protocol delivered over TCP. The protocol boundaries (message start/end) don't align with TCP packet boundaries. Always buffer until you have a complete HTTP message unit.

⚠️ Pitfall: Chunked Transfer Encoding Bugs

Chunked transfer encoding allows HTTP bodies to be sent without knowing the total size upfront. Parsers must handle the chunk size parsing, data reading, and trailer processing correctly.

State Transition Error	Symptom	Root Cause	Fix
`HTTP_PARSING_CHUNKED_SIZE` → `HTTP_PARSING_ERROR`	Parser fails on chunk size line	Not parsing hexadecimal chunk size correctly	Use `strtol(chunk_line, NULL, 16)` for hex conversion
Infinite loop in `HTTP_PARSING_CHUNKED_DATA`	Connection hangs reading chunk data	Chunk size calculation error, reading wrong number of bytes	Verify `bytes_remaining` matches parsed chunk size
Missing final chunk	Response appears truncated	Not detecting zero-size final chunk `0\r\n\r\n`	Check for `chunk_size == 0` transition to `HTTP_PARSING_COMPLETE`

⚠️ Pitfall: Request Smuggling Vulnerabilities

Request smuggling occurs when the proxy and backend server disagree on request boundaries, typically due to inconsistent `Content-Length` and `Transfer-Encoding` header handling.

```
Example vulnerable request:
POST /api/data HTTP/1.1
Host: example.com
Content-Length: 44
Transfer-Encoding: chunked

5c
POST /admin/delete HTTP/1.1
Host: example.com
Content-Length: 15

malicious_data
0
```

Vulnerability Pattern	Detection Method	Prevention Strategy
Dual `Content-Length` headers	Multiple `Content-Length` values in request	Reject requests with multiple `Content-Length` headers
`Content-Length` + `Transfer-Encoding`	Both headers present simultaneously	Prefer `Transfer-Encoding` when both present, or reject request
Invalid chunk encoding	Malformed chunk size or missing CRLF	Strict chunk format validation before forwarding

Connection Manager Issues

⚠️ Pitfall: File Descriptor Exhaustion

The connection manager can exhaust system file descriptors if connections aren't properly closed or if the connection pool grows unbounded.

Resource Leak Pattern	Symptom	Detection Command	Resolution
Client connections not closed	`accept()` fails with `EMFILE`	`lsof -p <proxy_pid> | wc -l`	Ensure `connection_manager_close_connection()` called on all error paths
Backend pool connections accumulate	Backend connection count grows continuously	Monitor `ConnectionPool.idle_count` and `active_count`	Implement idle timeout cleanup in `timeout_processor()`
Epoll events not removed	`epoll_wait()` returns events for closed FDs	`strace -e epoll_ctl` to see unmatched additions/deletions	Call `epoll_ctl(EPOLL_CTL_DEL)` before closing file descriptors

⚠️ Pitfall: Connection State Race Conditions

Multiple threads can modify connection state simultaneously, leading to use-after-free bugs or double-close errors when proper synchronization is missing.

Race Condition	Observable Behavior	Root Cause	Synchronization Fix
Double connection close	Segmentation fault in `close()` or `free()`	Two threads call `connection_manager_close_connection()`	Use atomic compare-and-swap for connection state transitions
Use-after-free on connection object	Random crashes accessing connection fields	Connection freed while another thread uses it	Reference counting with atomic operations
Backend selection vs connection close	Wrong backend receives request	Load balancer selects backend while connection manager closes it	Hold backend reference with proper cleanup ordering

Load Balancer Issues

⚠️ Pitfall: Backend Health Check False Positives

Health checks might report backends as healthy when they're actually overloaded or partially failing, leading to request failures despite "healthy" status.

False Positive Pattern	Symptom	Root Cause	Improved Detection
TCP connect succeeds but HTTP fails	Requests timeout despite healthy backends	Health check only tests TCP connectivity	Send actual HTTP request in `health_check_backend()`
Backend responds to health check but not requests	Intermittent request failures	Health check endpoint different from request handling	Use same endpoint/port for health checks as real requests
Health check too infrequent	Failures detected after many requests fail	Long intervals between health checks	Reduce `health_check_interval` and implement passive failure detection

⚠️ Pitfall: Load Balancer Algorithm Edge Cases

Load balancing algorithms can behave unexpectedly when backend weights change, connections finish, or servers are added/removed dynamically.

Algorithm Issue	Unexpected Behavior	Edge Case	Solution
Round-robin index out of bounds	Segmentation fault in backend selection	Backend removed while `rr_current_index` points to it	Reset index when backend array changes
Weighted round-robin starvation	Some backends never selected	Weight calculation overflow or zero weights	Normalize weights and handle zero-weight backends
Least-connections stale data	Requests sent to overloaded backends	Connection counts not updated on failures	Update counts immediately in error handlers

Cache Engine Issues

⚠️ Pitfall: Cache Coherence Problems

Cached responses can become stale or inconsistent when cache invalidation doesn't account for all scenarios where cached data should be purged.

Coherence Issue	Symptom	Root Cause	Invalidation Strategy
Stale responses served after backend update	Clients see old data after server changes	TTL too long or no invalidation on errors	Invalidate cache entries on 5xx responses from backend
Cache poisoning with error responses	404 or 500 responses cached and served repeatedly	Caching non-cacheable responses	Check `cache_is_cacheable()` includes status code validation
Vary header ignored in cache key	Wrong cached response for different request variants	Cache key doesn't include `Vary` header fields	Include `Vary` header values in `cache_generate_key()`

⚠️ Pitfall: Cache Memory Management

Cache engines can consume unbounded memory if eviction policies don't work correctly or if large responses are cached without size limits.

Memory Issue	Observable Behavior	Monitoring Metric	Memory Control
Cache size grows beyond limit	Proxy memory usage increases continuously	Monitor `CacheEngine.size_current` vs `size_limit`	Implement LRU eviction in `cache_engine_store()`
Large response caching	Single large file consumes entire cache	Track individual entry size in `CacheEntry.entry_size`	Reject responses larger than percentage of cache limit
Cache fragmentation	Available cache space but unable to store entries	Monitor successful vs failed cache storage attempts	Implement cache compaction or use memory pools

SSL Termination Issues

⚠️ Pitfall: Certificate Validation and SNI Problems

SSL termination can fail silently or present wrong certificates when Server Name Indication (SNI) handling or certificate loading has bugs.

SSL Issue	Client Observable Behavior	Server-Side Symptom	Resolution
Wrong certificate served	Browser certificate warning for different domain	SNI callback not triggered or incorrect context selection	Verify `ssl_termination_sni_callback()` hostname matching
Certificate chain incomplete	SSL handshake fails with "unknown CA"	Missing intermediate certificates	Load full certificate chain in `ssl_utils_load_certificate()`
Private key mismatch	SSL handshake fails with "bad certificate"	Certificate and key don't correspond	Use `ssl_utils_verify_key_cert_match()` during loading

⚠️ Pitfall: TLS Performance and Security Issues

SSL termination performance can degrade due to poor cipher selection, missing session resumption, or inefficient TLS context management.

Performance Issue	Observable Metric	Root Cause	Optimization
High SSL handshake latency	Increased connection establishment time	Weak key exchange algorithms	Configure ECDHE ciphers in `ssl_utils_set_secure_ciphers()`
Excessive CPU usage for SSL	High CPU utilization on SSL threads	No session resumption, repeated full handshakes	Enable TLS session tickets and caching
Memory usage grows with connections	SSL context memory increases	Creating new SSL_CTX per connection instead of reusing	Share SSL contexts across connections for same domain

Debugging Techniques and Tools

This section provides systematic approaches for diagnosing reverse proxy issues, from initial symptom observation through root cause identification to verification of fixes.

Systematic Debugging Approach

Mental Model: The Debugging Funnel

Think of reverse proxy debugging like a medical diagnosis process. Start with observable symptoms (the patient's complaints), gather more data through systematic observation (vital signs and tests), form hypotheses about root causes (differential diagnosis), test hypotheses with targeted experiments (specific tests), and verify the cure works (follow-up monitoring). Each step narrows down the possible causes until you isolate the specific problem.

The key insight is that reverse proxy bugs often manifest far from their source. A client timeout might be caused by DNS resolution delays, SSL handshake failures, backend overload, cache corruption, or thread synchronization bugs. The debugging process must systematically eliminate possibilities rather than jumping to conclusions based on surface symptoms.

Diagnostic Data Collection

Before attempting to fix any issue, establish comprehensive visibility into proxy operation. This involves capturing data at multiple layers of the system.

Connection-Level Diagnostics

Data Point	Collection Method	Information Revealed
Active connection count	Monitor `ConnectionManager` statistics	Whether issue is connection exhaustion
Connection state distribution	Count connections in each `ConnectionState`	Which processing stage has bottlenecks
Connection duration histogram	Track time from creation to closure	Whether connections are hanging
Bytes transferred per connection	Sum `bytes_read` and `bytes_written`	Whether transfers are completing
Backend connection pool utilization	Monitor `ConnectionPool.idle_count` vs `max_connections`	Whether backend pooling is effective

Request-Level Diagnostics

Diagnostic Technique	Implementation	Troubleshooting Value
Request ID tracing	Generate unique ID per request, log in all components	Follow request flow across components
Timing breakdown	Measure time in each processing stage	Identify bottleneck components
Header inspection	Log all request/response headers	Debug protocol-level issues
Cache hit/miss tracking	Log cache decisions with reasons	Understand cache effectiveness
Backend selection logging	Log load balancer decisions	Debug routing issues

Component-Specific Debugging Tools

HTTP Parser Debugging

When HTTP parser issues occur, the key is understanding exactly what data the parser received and how it interpreted the protocol boundaries.

Parser Issue	Debug Data to Collect	Analysis Method
Parse failures	Raw buffer contents before parsing	Check for valid HTTP format, protocol violations
Incomplete requests	Buffer state at each `http_parser_process()` call	Verify incremental parsing state machine
Header corruption	Header hash table contents after parsing	Compare parsed values with raw buffer
Body length errors	`Content-Length` vs actual body bytes received	Check for chunked encoding vs fixed length

Load Balancer Debugging

Load balancer issues require understanding the decision-making process and backend server state over time.

Debug Data Collection	Purpose	Implementation
Backend selection history	Track which backend chosen for each request	Log backend ID and selection algorithm result
Health check timeline	Record all health check attempts and results	Log successful/failed checks with timestamps
Connection count tracking	Monitor active connections per backend	Sample connection counts periodically
Weight adjustment history	Track dynamic weight changes	Log weight updates with reasons

Cache Engine Debugging

Cache-related issues often involve understanding why specific entries were or weren't cached, and how cache invalidation decisions were made.

Cache Debug Information	Collection Method	Analysis Purpose
Cache key generation	Log keys generated for each request	Debug cache miss issues
Cache-Control parsing	Log parsed cache directives	Understand caching decisions
TTL calculations	Log computed expiration times	Debug premature or delayed expiration
Eviction decisions	Log LRU operations and size-based evictions	Understand cache memory management
Cache hit/miss reasons	Log why each lookup succeeded or failed	Optimize cache effectiveness

Memory and Resource Leak Detection

Reverse proxies are long-running servers that must manage resources carefully to avoid gradual degradation.

Memory Leak Detection Strategy

Detection Method	Tool/Technique	What It Reveals
Process memory monitoring	Track RSS/VSZ over time with `ps` or `/proc/pid/status`	Overall memory growth trends
Allocation tracking	Use Valgrind or AddressSanitizer	Specific allocation sites that aren't freed
Connection object counting	Monitor `ConnectionManager` object counts	Whether connections are accumulating
Cache size monitoring	Track `CacheEngine.size_current` over time	Whether cache eviction is working
SSL context counting	Count SSL contexts in `SSLTermination`	Whether TLS contexts are leaking

File Descriptor Leak Detection

Resource Type	Monitoring Command	Normal vs Leak Pattern
Socket file descriptors	`lsof -p <pid> -a -i`	Should correlate with active connections
Regular file descriptors	`lsof -p <pid> -a -f -- /`	Should remain constant for config/log files
Pipe/eventfd descriptors	`lsof -p <pid> -a -t REG`	Should match thread communication channels
Total FD count	`ls /proc/<pid>/fd | wc -l`	Should stay within reasonable bounds

Network-Level Debugging

Since reverse proxies are network intermediaries, network-level debugging provides critical visibility into connection establishment, data transfer, and protocol interactions.

Traffic Analysis Techniques

Analysis Level	Tool	Information Provided
Packet capture	`tcpdump -i any -w proxy-traffic.pcap`	Raw network data for protocol analysis
Connection tracking	`ss -tupln` and `netstat -an`	Current socket states and listen ports
Traffic statistics	`iftop` or `nethogs`	Bandwidth utilization per connection
DNS resolution	`dig` and `nslookup` for backend hostnames	DNS resolution delays or failures

SSL/TLS Debugging

TLS issues require specialized analysis tools since the traffic is encrypted and handshake failures can have many causes.

TLS Debug Technique	Command/Tool	Diagnostic Value
Handshake analysis	`openssl s_client -connect proxy:443 -debug`	Step-by-step handshake progression
Certificate validation	`openssl verify -CApath /etc/ssl/certs cert.pem`	Certificate chain validation issues
Cipher negotiation	`nmap --script ssl-enum-ciphers -p 443 proxy`	Available cipher suites
SNI testing	`openssl s_client -servername domain.com -connect proxy:443`	SNI hostname handling

Performance Profiling and Bottleneck Analysis

Performance issues in reverse proxies often involve identifying which component or operation is the limiting factor under load.

CPU Profiling Strategy

Profiling Target	Tool/Method	Analysis Focus
Function-level CPU usage	`perf record -g` + `perf report`	Which functions consume most CPU time
Lock contention	`perf lock record` + `perf lock report`	Mutex/rwlock blocking behavior
System call overhead	`strace -c -p <pid>`	System call frequency and timing
Thread activity	`htop` with thread view	Per-thread CPU utilization

I/O Performance Analysis

I/O Pattern	Monitoring Method	Performance Implications
Socket read/write efficiency	Monitor `EAGAIN` frequency in logs	Whether non-blocking I/O is optimal
Disk I/O for caching	`iostat` to monitor disk utilization	Whether cache storage is bottleneck
DNS resolution performance	Time DNS lookups with `dig`	Whether backend resolution is slow
Backend response times	Log request duration per backend	Whether specific backends are slow

Systematic Root Cause Analysis

When debugging complex issues, use a systematic approach to avoid missing important causes or making incorrect assumptions.

The Five Whys Technique for Reverse Proxy Issues

Level	Question	Example Analysis
1st Why	Why is the symptom occurring?	"Clients are getting 503 errors"
2nd Why	Why is that immediate cause happening?	"Load balancer returns no healthy backends"
3rd Why	Why is that deeper cause happening?	"Health checks are failing for all backends"
4th Why	Why is that root cause occurring?	"Health check timeout is too short"
5th Why	Why was that configuration chosen?	"Default timeout not adjusted for network latency"

Hypothesis Testing Framework

For each suspected root cause, design specific tests that can confirm or refute the hypothesis.

Hypothesis	Test Design	Expected Result if Correct	Expected Result if Incorrect
"Backend overload causing timeouts"	Send requests directly to backend, bypassing proxy	Direct requests also timeout	Direct requests succeed
"SSL handshake failures"	Test same domain with HTTP vs HTTPS	HTTPS fails, HTTP succeeds	Both fail or both succeed
"Cache serving stale data"	Send cache-busting headers (`Cache-Control: no-cache`)	Fresh data returned	Same stale data returned
"Load balancer algorithm bug"	Manually specify backend in request	Specified backend works	All backends fail

Implementation Guidance

Technology Recommendations

Debugging Component	Simple Option	Advanced Option
Logging Framework	`fprintf(stderr, ...)` with timestamp	Structured logging with log levels and rotation
Memory Debugging	Valgrind memcheck	AddressSanitizer with custom allocator
Network Analysis	tcpdump + Wireshark	Custom packet capture with libpcap
Performance Profiling	gprof with `-pg` compilation	perf with flamegraph visualization
Configuration Debugging	Static config file parsing	Dynamic config reload with validation

Debugging Infrastructure Code

Complete Logging System Implementation:

C

```c
// logger.h
#include <stdio.h>
#include <time.h>
#include <stdarg.h>
#include <pthread.h>

typedef enum {
    LOG_DEBUG = 0,
    LOG_INFO = 1,
    LOG_WARN = 2,
    LOG_ERROR = 3
} LogLevel;

typedef struct {
    LogLevel min_level;
    FILE* output_file;
    pthread_mutex_t log_mutex;
    bool include_thread_id;
    bool include_component;
} Logger;

// Initialize logging system with specified minimum level and output file
bool logger_init(LogLevel min_level, FILE* output);

// Log formatted message with component, file, and line information
void logger_log(LogLevel level, const char* component, int line, const char* format, ...);

// Specialized logging macros for convenience
#define LOG_DEBUG_MSG(component, ...) logger_log(LOG_DEBUG, component, __LINE__, __VA_ARGS__)
#define LOG_INFO_MSG(component, ...) logger_log(LOG_INFO, component, __LINE__, __VA_ARGS__)
#define LOG_WARN_MSG(component, ...) logger_log(LOG_WARN, component, __LINE__, __VA_ARGS__)
#define LOG_ERROR_MSG(component, ...) logger_log(LOG_ERROR, component, __LINE__, __VA_ARGS__)

// logger.c
static Logger g_logger = {0};
```

```
bool logger_init(LogLevel min_level, FILE* output) {
    g_logger.min_level = min_level;
    g_logger.output_file = output ? output : stderr;
    g_logger.include_thread_id = true;
    g_logger.include_component = true;

    if (pthread_mutex_init(&g_logger.log_mutex, NULL) != 0) {
        return false;
    }

    return true;
}

void logger_log(LogLevel level, const char* component, int line, const char* format, ...) {
    if (level < g_logger.min_level) {
        return;
    }

    pthread_mutex_lock(&g_logger.log_mutex);

    // Generate timestamp
    time_t now = time(NULL);
    struct tm* local_time = localtime(&now);
    char timestamp[32];
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", local_time);

    // Get thread ID
    pthread_t thread_id = pthread_self();

    // Format level string
    const char* level_strings[] = {"DEBUG", "INFO", "WARN", "ERROR"};
```

```
    // Print log prefix
    fprintf(g_logger.output_file, "[%s] [%s] [%lu:%s:%d] ",
            timestamp, level_strings[level], (unsigned long)thread_id, component, line);

    // Print actual message
    va_list args;
    va_start(args, format);
    vfprintf(g_logger.output_file, format, args);
    va_end(args);

    fprintf(g_logger.output_file, "\n");
    fflush(g_logger.output_file);

    pthread_mutex_unlock(&g_logger.log_mutex);
}
```

Complete Error Context Tracking:

```c
// error_handler.h

#include "logger.h"

#include "proxy_types.h"


typedef enum {

    PROXY_ERROR_NONE = 0,

    PROXY_ERROR_PARSE_FAILED = 1,

    PROXY_ERROR_BACKEND_UNAVAILABLE = 2,

    PROXY_ERROR_BACKEND_TIMEOUT = 3,

    PROXY_ERROR_CACHE_FAILURE = 4,

    PROXY_ERROR_SSL_HANDSHAKE_FAILED = 5,

    PROXY_ERROR_CLIENT_DISCONNECTED = 6,

    PROXY_ERROR_MEMORY_ALLOCATION = 7,

    PROXY_ERROR_CONNECTION_LIMIT = 8,

    PROXY_ERROR_CONFIG_INVALID = 9

} ProxyErrorCode;


typedef struct {

    ProxyErrorCode code;

    char message[512];

    char component[64];

    time_t timestamp;

    uint64_t request_id;

    Connection* failed_connection;

    BackendServer* failed_backend;

    pthread_t thread_id;

    int system_errno;

} ErrorContext;


typedef struct {

    ErrorContext errors[1024];

    size_t error_count;

    size_t error_capacity;
```

```
    pthread_mutex_t error_mutex;
    pthread_t recovery_thread;
    bool running;
} ErrorHandler;

// Initialize centralized error handling system
ErrorHandler* error_handler_create(void);

// Report error to central handler for recovery coordination
void error_handler_report(ErrorHandler* handler, ErrorContext* context);

// Generate sanitized error responses for clients
void generate_client_error_response(Connection* conn, ProxyErrorCode code, const char* details);

// error_handler.c
ErrorHandler* error_handler_create(void) {
    ErrorHandler* handler = malloc(sizeof(ErrorHandler));
    if (!handler) return NULL;

    memset(handler, 0, sizeof(ErrorHandler));
    handler->error_capacity = 1024;

    if (pthread_mutex_init(&handler->error_mutex, NULL) != 0) {
        free(handler);
        return NULL;
    }

    handler->running = true;
    return handler;
}

void error_handler_report(ErrorHandler* handler, ErrorContext* context) {
    if (!handler || !context) return;
```

```
    pthread_mutex_lock(&handler->error_mutex);

    if (handler->error_count < handler->error_capacity) {
        // Store error context
        memcpy(&handler->errors[handler->error_count], context, sizeof(ErrorContext));
        handler->error_count++;

        // Log error with full context
        LOG_ERROR_MSG("error_handler",
                      "Error %d in %s: %s (request_id=%lu, connection=%p, backend=%p, errno=%d)",
                      context->code, context->component, context->message,
                      context->request_id, context->failed_connection,
                      context->failed_backend, context->system_errno);
    }

    pthread_mutex_unlock(&handler->error_mutex);
}

void generate_client_error_response(Connection* conn, ProxyErrorCode code, const char* details) {
    if (!conn) return;

    // Generate appropriate HTTP error response
    const char* status_text = "Internal Server Error";
    int status_code = 500;

    switch (code) {
        case PROXY_ERROR_BACKEND_UNAVAILABLE:
            status_code = 503;
            status_text = "Service Unavailable";
            break;
        case PROXY_ERROR_BACKEND_TIMEOUT:
            status_code = 504;
```

```
            status_text = "Gateway Timeout";
            break;
        case PROXY_ERROR_PARSE_FAILED:
            status_code = 400;
            status_text = "Bad Request";
            break;
        case PROXY_ERROR_SSL_HANDSHAKE_FAILED:
            status_code = 400;
            status_text = "Bad Request";
            break;
        default:
            break;
    }

    // Sanitize details - never expose internal information
    const char* safe_details = "The server encountered an error processing your request.";

    char response[1024];
    snprintf(response, sizeof(response),
             "HTTP/1.1 %d %s\r\n"
             "Content-Type: text/plain\r\n"
             "Content-Length: %zu\r\n"
             "Connection: close\r\n"
             "\r\n"
             "%s",
             status_code, status_text, strlen(safe_details), safe_details);

    // Send error response to client
    send(conn->client_fd, response, strlen(response), 0);
    LOG_INFO_MSG("error_response", "Sent %d %s to client (error: %s)",
                 status_code, status_text, details);
```

```
}
```

Core Logic Debugging Skeletons

Request Tracing Implementation:

```c
// request_tracer.h

typedef struct {

    uint64_t request_id;

    char client_ip[INET_ADDRSTRLEN];

    char method[16];

    char uri[512];

    time_t start_time;

    time_t parse_complete_time;

    time_t backend_selected_time;

    time_t cache_lookup_time;

    time_t backend_request_time;

    time_t response_complete_time;

    BackendServer* selected_backend;

    bool cache_hit;

    ProxyErrorCode error_code;

} RequestTrace;


RequestTrace* request_trace_create(Connection* conn, HttpRequest* request) {

    // TODO 1: Allocate RequestTrace structure

    // TODO 2: Generate unique request ID using atomic counter

    // TODO 3: Extract client IP from connection socket address

    // TODO 4: Copy HTTP method and URI from request

    // TODO 5: Set start_time to current timestamp

    // TODO 6: Initialize timing fields to 0

    // TODO 7: Initialize error_code to PROXY_ERROR_NONE

    // TODO 8: Log request start with all initial context

}


void request_trace_mark_milestone(RequestTrace* trace, const char* milestone) {

    // TODO 1: Get current timestamp

    // TODO 2: Set appropriate timing field based on milestone string

    // TODO 3: Log milestone reached with elapsed time since start

    // TODO 4: If this is an error milestone, set error_code appropriately
```

```
}

void request_trace_complete(RequestTrace* trace) {
    // TODO 1: Set response_complete_time to current timestamp
    // TODO 2: Calculate total request duration
    // TODO 3: Log complete request summary with all timing breakdowns
    // TODO 4: Log backend selection, cache hit status, and any errors
    // TODO 5: Free RequestTrace structure
}
```

Connection State Debugging:

```c
// connection_debug.h

typedef struct {

    Connection* connection;

    ConnectionState previous_state;

    ConnectionState current_state;

    time_t transition_time;

    const char* transition_reason;

    size_t bytes_processed;

    int system_error;

} ConnectionStateTrace;


void debug_log_connection_state_change(Connection* conn, ConnectionState new_state,

                                       const char* reason) {

    // TODO 1: Check if connection state actually changed

    // TODO 2: Create ConnectionStateTrace with previous and new states

    // TODO 3: Log state transition with timing and reason

    // TODO 4: If transitioning to error state, log additional context

    // TODO 5: Update connection's last_activity timestamp

    // TODO 6: Store trace in connection debugging history

}


void debug_dump_connection_state(Connection* conn) {

    // TODO 1: Log current connection state and timing information

    // TODO 2: Log buffer states (request_buffer and response_buffer positions)

    // TODO 3: Log backend server information if connected

    // TODO 4: Log any pending timeout information

    // TODO 5: Log recent state transition history

}
```

Milestone Checkpoints

Milestone 1 Debugging Verification:

After implementing basic HTTP proxy functionality, verify debugging capabilities:

1. **Start proxy with debug logging enabled**: `./proxy --log-level=DEBUG --log-file=debug.log`
2. **Send malformed HTTP request** to test parser error handling:

```bash
echo -e "GET /test HTTP/1.1\r\nInvalid-Header-Missing-Colon\r\n\r\n" | nc localhost 8080
```

Expected: Parser error logged with specific buffer contents and error reason.

3. **Monitor connection state transitions** with legitimate request:

```bash
curl -v http://localhost:8080/test
```

Expected: State transitions logged from `CONNECTION_IDLE` → `CONNECTION_READING_REQUEST` → `CONNECTION_FORWARDING` → etc.

4. **Verify error response generation** when backend unavailable:
   - Stop backend server
   - Send request through proxy
   - Expected: 503 Service Unavailable response with sanitized error message

Milestone 2 Load Balancer Debugging:

1. **Test backend selection logging**:

```bash
# Send multiple requests and verify round-robin distribution

for i in {1..10}; do curl http://localhost:8080/test; done
```

Expected: Log shows requests distributed evenly across healthy backends.

2. **Test health check failure detection**:
   - Stop one backend server
   - Wait for health check interval
   - Send requests
   - Expected: Failed backend marked unhealthy and excluded from selection.

3. **Test connection count tracking**:
   - Send concurrent requests
   - Expected: Connection counts per backend logged and updated correctly.

Milestone 3-5 Advanced Debugging:

Each subsequent milestone should verify debugging capabilities for new components:

- Cache hit/miss logging with reasons
- SSL handshake failure diagnosis
- Performance profiling under load

Common Debugging Command Patterns

Resource Monitoring Commands:

```bash
# Monitor file descriptor usage
watch -n 1 'lsof -p $(pgrep proxy) | wc -l'

# Monitor memory usage
watch -n 1 'cat /proc/$(pgrep proxy)/status | grep VmRSS'

# Monitor network connections
watch -n 1 'ss -tupln | grep :8080'

# Monitor thread activity
htop -H -p $(pgrep proxy)

# Capture network traffic
tcpdump -i any -w proxy-debug.pcap host backend-server

# Profile CPU usage
perf record -g -p $(pgrep proxy) sleep 10
perf report --stdio
```

These debugging techniques and tools provide comprehensive visibility into reverse proxy operation, enabling systematic diagnosis of issues from simple configuration errors to complex race conditions and performance bottlenecks.

Future Extensions

> **Milestone(s):** All milestones - the extensibility architecture established during the core implementation (Milestones 1-5) enables future growth without requiring fundamental redesigns.

Think of a reverse proxy like a Swiss Army knife that starts with basic tools but has slots for additional specialized implements. The base platform provides the essential mechanisms (HTTP parsing, connection management, request routing), while the extension system allows new tools to be added without redesigning the entire handle. Each extension leverages the existing infrastructure while adding specialized capabilities, just as a magnifying glass attachment uses the knife's existing frame while providing new optical functionality.

Strategic Extension Framework

The reverse proxy's architecture naturally accommodates future enhancements through several key extensibility patterns. The **plugin architecture** allows new functionality to be added without modifying core components, while the **configuration-driven approach** enables features to be enabled or disabled dynamically. The **event-driven design** provides hooks where extensions can intercept and modify request processing, and the **component isolation** ensures that new features don't destabilize existing functionality.

Decision: Extension Architecture Pattern

- **Context**: Future features require integration points without destabilizing the core proxy functionality or requiring architectural rewrites
- **Options Considered**: Monolithic expansion, Plugin system with dynamic loading, Event-driven hooks with static compilation
- **Decision**: Event-driven hooks with static compilation and configuration-based activation
- **Rationale**: Provides flexibility without the complexity and security risks of dynamic plugin loading, while maintaining compile-time type safety and performance
- **Consequences**: Extensions require recompilation but gain full access to core data structures and zero-overhead integration

Extension Pattern	Implementation Complexity	Runtime Overhead	Security Risk	Flexibility
Monolithic Expansion	Low	None	Low	Limited
Dynamic Plugin System	High	Moderate	High	Maximum
Event-Driven Hooks	Moderate	Minimal	Low	High

Rate Limiting and Throttling

Rate limiting acts like a nightclub bouncer who controls the flow of patrons to prevent overcrowding. The bouncer counts how many people have entered recently, checks against established limits, and either allows entry or asks visitors to wait. Similarly, rate limiting tracks request patterns per client and enforces configurable limits to protect backend servers from overload.

The rate limiting extension integrates with the existing `HttpParser` and `ConnectionManager` components by adding request tracking before the load balancing stage. Each incoming request triggers rate limit evaluation based on client IP address, request path patterns, or custom header values. The system maintains sliding window counters that track request volumes across different time intervals, enabling both burst protection and sustained rate enforcement.

Rate Limiting Data Structures:

Structure	Field	Type	Description
`RateLimiter`	rules	`RateLimitRule**`	Array of configured rate limiting rules
`RateLimiter`	rule_count	`size_t`	Number of active rate limiting rules
`RateLimiter`	client_counters	`HashTable*`	Per-client request counters indexed by IP
`RateLimiter`	sliding_windows	`SlidingWindow**`	Time-based request counting windows
`RateLimiter`	cleanup_thread	`pthread_t`	Background thread for counter cleanup
`RateLimiter`	limiter_mutex	`pthread_rwlock_t`	Synchronization for counter updates
`RateLimitRule`	pattern	`char[512]`	URL pattern or client identifier pattern
`RateLimitRule`	requests_per_second	`uint32_t`	Maximum requests allowed per second
`RateLimitRule`	burst_size	`uint32_t`	Maximum burst requests before rate limiting
`RateLimitRule`	time_window_seconds	`uint32_t`	Time window for rate calculation
`ClientCounter`	client_id	`char[64]`	Client identifier (IP address or custom key)
`ClientCounter`	request_count	`uint32_t`	Current request count in time window
`ClientCounter`	last_request_time	`time_t`	Timestamp of most recent request
`ClientCounter`	burst_tokens	`uint32_t`	Available burst request tokens

The rate limiter extension hooks into the request processing pipeline at three key points: **request arrival** (to increment counters), **backend selection** (to enforce limits before load balancing), and **response generation** (to add rate limit headers). When a request exceeds configured limits, the system generates an HTTP 429 Too Many Requests response with appropriate Retry-After headers, preventing the request from reaching backend servers.

> The critical insight for rate limiting is that enforcement must occur as early as possible in the request pipeline to maximize protection effectiveness. Waiting until after parsing or load balancing wastes processing resources on requests that will ultimately be rejected.

Rate Limiting Integration Points:

Integration Point	Component	Hook Function	Purpose
Request Arrival	`ConnectionManager`	`rate_limiter_check_request()`	Early request validation
Backend Selection	`LoadBalancer`	`rate_limiter_pre_balance()`	Pre-routing enforcement
Response Headers	`HttpParser`	`rate_limiter_add_headers()`	Rate limit status communication
Counter Cleanup	Background Thread	`rate_limiter_cleanup_expired()`	Memory management

Web Application Firewall (WAF)

A Web Application Firewall functions like an expert security guard who examines every visitor's belongings and behavior patterns, looking for signs of malicious intent. The guard knows common attack signatures (like weapons or suspicious tools) and behavioral patterns (like someone trying to access restricted areas repeatedly). When threats are detected, the guard can block entry, strip dangerous items, or alert security management.

The WAF extension operates on fully parsed HTTP requests, examining headers, URL patterns, request bodies, and parameter values against configurable rule sets. It integrates after the `HttpParser` component completes request parsing but before the `LoadBalancer` selects a backend server, ensuring that malicious requests never reach application servers.

WAF Architecture Components:

Component	Responsibility	Integration Point
Rule Engine	Pattern matching and threat detection	Post-parsing, pre-routing
Signature Database	Known attack patterns and payloads	Static configuration with hot reload
Anomaly Detector	Statistical analysis of request patterns	Background analysis thread
Response Generator	Security error page generation	Request termination point
Audit Logger	Security event logging and alerting	Cross-cutting concern

The WAF rule engine processes requests through multiple detection layers: **signature matching** (comparing request content against known attack patterns), **statistical anomaly detection** (identifying unusual request characteristics), **rate-based detection** (spotting suspicious request volumes), and **behavioral analysis** (tracking client interaction patterns over time).

WAF Rule Types:

Rule Type	Detection Method	Example Pattern	Action Options
SQL Injection	Pattern matching	`UNION SELECT.*FROM`	Block, sanitize, log
XSS Attack	Content scanning	`<script>.*</script>`	Block, escape, log
Path Traversal	URL analysis	`\.\.\/` sequences	Block, normalize, log
Command Injection	Parameter scanning	Shell metacharacters	Block, sanitize, log
Rate Anomaly	Statistical analysis	Request volume spikes	Throttle, block, log

When the WAF detects threats, it can take various actions: **blocking** (returning 403 Forbidden responses), **sanitizing** (cleaning request content and forwarding), **logging** (recording security events while allowing requests), or **challenging** (requiring additional authentication). The system maintains detailed audit logs that include threat classifications, rule triggers, client information, and response actions.

Monitoring and Observability

Monitoring a reverse proxy resembles managing a busy restaurant kitchen where you need visibility into every station's performance. The head chef needs real-time information about order volumes, preparation times, ingredient availability, and staff performance. Similarly, proxy monitoring provides comprehensive visibility into request flows, component performance, error rates, and resource utilization across all system layers.

The observability extension integrates with every core component through instrumentation hooks that collect metrics without impacting request processing performance. The system employs **push-based metrics** (actively sending data to monitoring systems), **pull-based metrics** (exposing endpoints for metric collection), and **structured logging** (machine-readable log formats for analysis).

Monitoring Data Model:

Metric Category	Component Source	Key Metrics	Collection Method
Request Metrics	`HttpParser`	Parse time, request size, protocol version	Per-request instrumentation
Connection Metrics	`ConnectionManager`	Pool utilization, connection lifetime, timeouts	Connection lifecycle hooks
Load Balancing	`LoadBalancer`	Backend selection time, health check results	Algorithm execution hooks
Cache Performance	`CacheEngine`	Hit/miss ratios, eviction rates, storage usage	Cache operation instrumentation
SSL Performance	`SSLTermination`	Handshake duration, cipher negotiation, SNI usage	TLS event callbacks

The monitoring system exposes metrics through multiple interfaces: **Prometheus endpoints** (for scraping-based collection), **StatsD integration** (for push-based metric delivery), **structured JSON logs** (for centralized log aggregation), and **health check endpoints** (for load balancer health monitoring). Each interface provides different metric granularities and update frequencies to support various monitoring architectures.

Observable Events:

Event Type	Trigger Condition	Included Data	Monitoring Use Case
Request Started	Client connection accepted	Timestamp, client IP, request ID	Request tracing
Backend Selected	Load balancer chooses server	Backend ID, selection algorithm, health status	Load distribution analysis
Cache Hit/Miss	Cache lookup completed	Cache key, hit status, TTL remaining	Cache performance tuning
Error Occurred	Any component error	Error code, component, stack trace	Error rate monitoring
Connection Pooled	Backend connection returned	Pool size, connection age, reuse count	Pool efficiency analysis

Authentication and Authorization

Authentication and authorization work like a sophisticated embassy security system with multiple checkpoints. The first checkpoint verifies visitor identity through passport examination (authentication), while subsequent checkpoints determine which areas the visitor can access based on their diplomatic status and clearance level (authorization). The system maintains visitor records, tracks access patterns, and can revoke access privileges dynamically.

The auth extension integrates early in the request pipeline, immediately after HTTP parsing but before cache lookup or backend selection. This positioning ensures that unauthorized requests consume minimal system resources while authorized requests benefit from full proxy optimizations including caching and connection pooling.

Authentication Integration Architecture:

Integration Point	Purpose	Component Interaction	Data Flow
Request Validation	Identity verification	`HttpParser` → Auth Extension	Headers → Auth decision
Cache Key Modification	User-aware caching	Auth Extension → `CacheEngine`	User context → Cache key
Backend Selection	User-based routing	Auth Extension → `LoadBalancer`	User role → Backend pool
Response Headers	Auth status communication	Auth Extension → `HttpParser`	Auth result → Response headers

The authentication system supports multiple verification methods: **JWT token validation** (for stateless authentication), **session cookie verification** (for traditional web applications), **API key authentication** (for service-to-service communication), and **OAuth2**

token introspection (for delegated authorization). Each method integrates with external identity providers while maintaining local caching for performance.

Authorization Decision Engine:

Decision Factor	Data Source	Evaluation Method	Caching Strategy
User Identity	Auth token/session	Token validation/lookup	In-memory with TTL
Request Path	HTTP request URL	Pattern matching	Rule compilation
HTTP Method	Request method header	Exact matching	Static configuration
User Roles	Identity provider	Role membership check	User session cache
Resource Permissions	Permission database	ACL evaluation	Permission result cache

When authorization fails, the system can respond with different strategies: **401 Unauthorized** (for missing authentication), **403 Forbidden** (for insufficient permissions), **redirect to login** (for web applications), or **custom error pages** (for branded experiences). The auth extension maintains audit logs for security compliance, tracking authentication attempts, authorization decisions, and access pattern anomalies.

Content Transformation

Content transformation functions like a universal translator and format converter at an international conference. Just as the translator converts languages and cultural references to ensure effective communication between parties, content transformation adapts response formats, protocols, and encodings to match client capabilities and requirements.

The transformation extension operates on HTTP responses after they return from backend servers but before they reach the client. This positioning allows the proxy to modify content without impacting backend server logic while ensuring that transformations respect caching and connection management optimizations.

Transformation Pipeline Architecture:

Transformation Stage	Purpose	Processing Order	Configuration
Content Negotiation	Format selection	1st - immediately after backend response	Client Accept headers
Protocol Translation	Version conversion	2nd - after format decision	HTTP version capabilities
Compression	Bandwidth optimization	3rd - after content finalization	Client encoding support
Security Headers	Response hardening	4th - final header addition	Security policy configuration

The content transformation system supports multiple transformation types: **protocol conversion** (HTTP/1.1 to HTTP/2), **format translation** (JSON to XML or vice versa), **image optimization** (resizing and compression), **response compression** (gzip, brotli), and **security header injection** (HSTS, CSP, CORP headers).

Transformation Rules:

Rule Type	Trigger Condition	Input Data	Output Modification	Performance Impact
Protocol Downgrade	HTTP/2 client, HTTP/1.1 backend	Response stream	Header format conversion	Low
JSON to XML	Accept: application/xml header	Response body	Format transformation	High
Image Resize	Image content type + query params	Binary response body	Resized image data	Very High
Compression	Client accepts gzip/brotli	Response body	Compressed body + headers	Moderate
Security Headers	All responses	Response headers	Additional security headers	Minimal

Content transformation integrates with the `CacheEngine` to ensure that transformed content is cached appropriately. The cache key generation includes transformation parameters so that different client capabilities receive correctly cached responses. For example, a gzipped response and an uncompressed response for the same resource are cached as separate entries.

Service Mesh Integration

Service mesh integration transforms the reverse proxy into a service mesh sidecar proxy, similar to how a personal assistant coordinates all communications and interactions for a busy executive. The assistant handles scheduling, protocol translation, security verification, and relationship management, allowing the executive to focus on core business decisions while ensuring all interactions follow organizational policies and procedures.

The service mesh extension adds distributed tracing, service discovery, traffic policies, and inter-service security. It integrates with the existing `LoadBalancer` component for dynamic service discovery and with the `ConnectionManager` for connection security and observability.

Service Mesh Components:

Component	Responsibility	Integration Point	External Dependencies
Service Discovery	Backend server enumeration	`LoadBalancer` backend configuration	Kubernetes API, Consul, etcd
Traffic Policy Engine	Request routing and shaping	`LoadBalancer` selection algorithm	Policy configuration store
Distributed Tracing	Request correlation across services	All component instrumentation	Jaeger, Zipkin, OpenTelemetry
mTLS Manager	Inter-service authentication	`SSLTermination` certificate management	Certificate authority integration

The service mesh extension enables **progressive traffic deployment** (canary releases and blue-green deployments), **circuit breaker patterns** (automatic failure isolation), **retry and timeout policies** (configurable resilience patterns), and **traffic splitting** (A/B testing and gradual rollouts).

Service Mesh Policies:

Policy Type	Configuration Scope	Enforcement Point	Dynamic Updates
Traffic Routing	Per-service destination	Backend selection	Real-time via API
Retry Behavior	Per-service or per-endpoint	Connection failure handling	Configuration reload
Timeout Values	Per-service or per-operation	Request processing	Real-time via API
Circuit Breaker	Per-backend server	Health checking integration	Real-time via API
Load Balancing	Per-service backend pool	Load balancer algorithm	Configuration reload

Multi-Tenancy Support

Multi-tenancy resembles managing a large apartment building where different tenants share common infrastructure (elevators, utilities, security) while maintaining complete isolation of their private spaces and resources. The building management system ensures that tenant A cannot access tenant B's apartment, utilities are billed correctly, and common areas remain available to all authorized residents.

The multi-tenancy extension adds **tenant isolation** (separate resource pools and configurations), **tenant-aware routing** (directing requests to appropriate backend clusters), **resource quotas** (preventing tenant resource exhaustion), and **audit isolation** (separate logging and monitoring per tenant).

Multi-Tenant Architecture:

Isolation Layer	Implementation Method	Resource Separation	Configuration Management
Network	Virtual routing tables	Separate backend pools per tenant	Tenant-specific load balancer rules
Cache	Namespace prefixing	Isolated cache partitions	Per-tenant cache policies
SSL	Certificate management	Tenant-specific certificates	SNI-based tenant resolution
Monitoring	Metric tagging	Separate metric namespaces	Tenant-aware dashboards
Rate Limiting	Client classification	Per-tenant rate limits	Tenant-specific policies

The multi-tenancy system determines tenant identity through multiple methods: **subdomain analysis** (tenant1.example.com), **URL path prefixes** (/tenant1/api/...), **custom headers** (X-Tenant-ID), or **SSL certificate subject names**. Once identified, all proxy components apply tenant-specific configurations and resource allocations.

Design Extensibility Features

The current reverse proxy architecture accommodates future extensions through several key design patterns that were established during the core implementation phases.

Decision: Extension Hook Architecture

- **Context**: Future features require integration points without modifying core component logic or destabilizing existing functionality
- **Options Considered**: Callback functions, Event publishing/subscription, Component inheritance
- **Decision**: Event publishing with typed message interfaces and subscriber registration
- **Rationale**: Provides loose coupling between extensions and core components while maintaining type safety and compile-time validation
- **Consequences**: Extensions can observe and modify request processing without requiring changes to core components, but extension interactions must be carefully coordinated

Extension Integration Patterns:

Pattern	Usage Scenario	Implementation Method	Trade-offs
Pre-processing Hooks	Request modification before routing	Event subscription at request parsing completion	Low latency, limited context
Post-processing Hooks	Response modification before client delivery	Event subscription at response generation	Full context, higher latency
Component Replacement	Alternative algorithm implementations	Interface inheritance with factory selection	Maximum flexibility, complexity
Configuration Extensions	New configuration sections	Configuration parser plugin registration	Easy integration, limited runtime changes

The `EventDispatcher` component, introduced during the core architecture phase, provides the foundation for extension integration. Extensions register event handlers for specific processing stages, receiving typed messages with full request context and the ability to modify processing behavior.

Extension Event Types:

Event Type	Trigger Point	Message Data	Extension Capabilities
`RequestParsed`	After HTTP parsing completion	`HttpRequest*`, `Connection*`	Header modification, request blocking
`BackendSelected`	After load balancer decision	`BackendServer*`, `HttpRequest*`	Backend override, request transformation
`CacheLookup`	Before cache key generation	`HttpRequest*`, cache key	Cache key modification, bypass decisions
`ResponseReceived`	After backend response arrival	`HttpResponse*`, `BackendServer*`	Content transformation, header injection
`ConnectionEstablished`	After client connection acceptance	`Connection*`, client address	Connection rejection, metadata attachment

The configuration system supports extension-specific sections through a plugin registration mechanism. Extensions provide configuration schema definitions that integrate with the main configuration parser, enabling complex extension settings while maintaining configuration validation and hot-reload capabilities.

Performance Considerations for Extensions

Extensions must balance functionality with performance impact, particularly since the reverse proxy operates in the critical path of application traffic. The architecture provides several mechanisms to minimize extension overhead while maximizing functionality.

Performance Optimization Strategies:

Strategy	Application Area	Performance Gain	Implementation Complexity
Event Filtering	Extension activation	Skip unused extensions	Low
Asynchronous Processing	Heavy computations	Non-blocking request path	High
Result Caching	Expensive operations	Amortized computation cost	Moderate
Bulk Operations	Batch processing	Reduced per-request overhead	Moderate

Extensions that perform expensive operations (content transformation, complex authentication) implement **asynchronous processing patterns** where the extension initiates background work and registers completion callbacks. This approach prevents extension processing from blocking the main request pipeline while ensuring that responses incorporate extension results.

The extension system includes **performance budgets** that limit the maximum processing time extensions can consume per request. When extensions exceed their allocated time budget, the system can skip optional extensions, use cached results, or fail gracefully while maintaining core proxy functionality.

Common Extension Pitfalls

⚠️ **Pitfall: Memory Leaks in Extension Data** Extensions that allocate memory for request processing (authentication tokens, transformation buffers, cached results) must properly integrate with the connection lifecycle to ensure cleanup. Extension data attached to `Connection` structures requires cleanup callbacks registered during attachment.

⚠️ **Pitfall: Blocking Operations in Extension Hooks** Extensions that perform synchronous I/O operations (database lookups, external API calls, file system access) in event handlers block the entire request processing pipeline. Extensions requiring external data must use asynchronous patterns with completion callbacks.

⚠️ **Pitfall: Thread Safety in Extension State** Extensions that maintain global state (caches, configuration, metrics) must implement proper synchronization when accessed from multiple worker threads. The proxy's event-driven architecture ensures that request processing can occur concurrently across multiple connections.

⚠️ **Pitfall: Extension Ordering Dependencies** Multiple extensions processing the same request can create ordering dependencies where Extension A's output becomes Extension B's input. The extension system requires explicit ordering configuration to ensure deterministic behavior.

⚠️ **Pitfall: Configuration Validation Gaps** Extensions that introduce new configuration sections must implement comprehensive validation to prevent runtime failures. Invalid extension configuration should be detected during startup rather than during request processing.

Implementation Guidance

The reverse proxy's extension architecture leverages the event-driven foundation established during core development. Extensions integrate through well-defined interfaces that provide access to request context while maintaining performance and stability guarantees.

Technology Recommendations for Extensions:

Extension Type	Simple Implementation	Advanced Implementation
Request Filtering	Direct header inspection	Pattern matching with compiled regex
Content Transformation	String replacement	Streaming transformation with SAX parsing
Authentication	Static token validation	JWT with cryptographic verification
Monitoring	Simple counter increments	Time-series metrics with statistical aggregation
Rate Limiting	Token bucket algorithm	Sliding window with distributed state

Extension Integration File Structure:

```
proxy/
├── core/                    ← Core proxy components
│   ├── http_parser.c
│   ├── connection_manager.c
│   └── load_balancer.c
├── extensions/              ← Extension implementations
│   ├── rate_limiter/
│   │   ├── rate_limiter.h   ← Extension interface
│   │   ├── rate_limiter.c   ← Core rate limiting logic
│   │   └── sliding_window.c ← Helper components
│   ├── waf/
│   │   ├── waf.h
│   │   ├── rule_engine.c
│   │   └── signature_db.c
│   └── auth/
│       ├── auth_extension.h
│       ├── jwt_validator.c
│       └── session_manager.c
├── extension_framework/     ← Extension support infrastructure
│   ├── event_dispatcher.h   ← Event system interfaces
│   ├── extension_manager.c  ← Extension lifecycle management
│   └── extension_config.c   ← Configuration integration
└── config/                  ← Extension configuration
    ├── rate_limits.conf
    ├── waf_rules.conf
    └── auth_policies.conf
```

Extension Registration Infrastructure:

```c
// Extension interface definition
typedef struct Extension {
    char name[64];                          // Extension identifier
    bool (*init)(ProxyConfig* config);      // Extension initialization
    void (*destroy)(void);                  // Extension cleanup
    EventHandler* handlers;                 // Event handler array
    size_t handler_count;                   // Number of handlers
} Extension;

// Event handler definition for extensions
typedef struct EventHandler {
    EventType event_type;                   // Which event to handle
    int priority;                           // Handler execution order
    bool (*handler_func)(EventData* data);  // Handler implementation
    bool enabled;                           // Runtime enable/disable
} EventHandler;

// Extension manager for lifecycle control
typedef struct ExtensionManager {
    Extension* extensions[MAX_EXTENSIONS];  // Loaded extensions
    size_t extension_count;                 // Number of loaded extensions
    HashTable* event_handlers;              // Handlers by event type
    pthread_rwlock_t extensions_lock;       // Thread-safe registration
    bool extensions_enabled;                // Global extension toggle
} ExtensionManager;
```

Core Extension Registration Functions:

```c
// Register extension with the proxy system
bool extension_manager_register(ExtensionManager* manager, Extension* ext) {
    // TODO: Validate extension interface completeness
    // TODO: Check for name conflicts with existing extensions
    // TODO: Initialize extension with proxy configuration
    // TODO: Register event handlers in dispatcher
    // TODO: Add extension to manager's extension array
    // Hint: Use write lock during registration to prevent races
}

// Initialize all registered extensions during proxy startup
bool extension_manager_init_all(ExtensionManager* manager, ProxyConfig* config) {
    // TODO: Iterate through registered extensions
    // TODO: Call each extension's init function with config
    // TODO: Disable extensions that fail initialization
    // TODO: Sort event handlers by priority for each event type
    // TODO: Log extension initialization results
}

// Dispatch event to all registered handlers
bool extension_manager_dispatch_event(ExtensionManager* manager, EventType type,
                                      EventData* data) {
    // TODO: Look up handlers for the specified event type
    // TODO: Execute handlers in priority order
    // TODO: Stop processing if any handler returns false (blocks request)
    // TODO: Update extension performance metrics
    // TODO: Handle handler exceptions gracefully
}
```

Rate Limiting Extension Example:

```c
// Rate limiting extension implementation

typedef struct RateLimitExtension {

    RateLimiter* limiter;                  // Core rate limiting logic

    pthread_t cleanup_thread;              // Background counter cleanup

    bool thread_running;                   // Thread lifecycle flag

} RateLimitExtension;


// Extension initialization function

bool rate_limit_extension_init(ProxyConfig* config) {

    // TODO: Parse rate limiting configuration from config file

    // TODO: Initialize rate limiter with configured rules

    // TODO: Start background cleanup thread for expired counters

    // TODO: Register event handlers for request arrival and response

    // TODO: Set up performance monitoring for rate limit decisions

}


// Request arrival event handler for rate limiting

bool rate_limit_handle_request(EventData* event_data) {

    // TODO: Extract client identifier from request (IP, user ID, API key)

    // TODO: Look up rate limit rules that apply to this request

    // TODO: Check current request count against configured limits

    // TODO: Update request counters for this client

    // TODO: Return false to block request if limits exceeded, true to allow

    // Hint: Generate HTTP 429 response with Retry-After header for blocked requests

}
```

Extension Configuration Integration:

```c
// Extension configuration parser registration

typedef struct ExtensionConfigParser {

    char section_name[64];                 // Configuration section identifier

    bool (*parse_section)(cJSON* section, void** config); // Parser function

    bool (*validate_config)(void* config); // Configuration validator

    void (*free_config)(void* config);     // Configuration cleanup

} ExtensionConfigParser;


// Register configuration parser for extension

bool config_register_extension_parser(ConfigManager* manager,

                                      ExtensionConfigParser* parser) {

    // TODO: Validate parser function pointers are non-null

    // TODO: Check for section name conflicts

    // TODO: Add parser to configuration manager's parser registry

    // TODO: Enable hot-reload support for extension configuration

}
```

Milestone Checkpoint - Extension Framework: After implementing the extension framework infrastructure, verify functionality by:

1. **Extension Registration Test**: Create a simple test extension that logs requests and verify it loads correctly during proxy startup
2. **Event Dispatch Test**: Send HTTP requests and confirm that extension event handlers receive appropriate event data
3. **Configuration Integration Test**: Add extension-specific configuration and verify parsing and validation work correctly
4. **Performance Test**: Measure request processing latency with and without extensions to ensure minimal overhead
5. **Thread Safety Test**: Run concurrent requests while loading/unloading extensions to verify thread safety

Expected behavior: Extensions should integrate seamlessly without impacting core proxy functionality, and extension failures should not crash the proxy process.

The extension architecture provides a robust foundation for evolving the reverse proxy to meet changing requirements while maintaining the performance, reliability, and security characteristics established during core development.

Glossary

> **Milestone(s):** All milestones - this glossary provides definitions for technical terms, acronyms, and domain-specific vocabulary used throughout the reverse proxy implementation.

A comprehensive understanding of reverse proxy terminology is essential for successfully implementing the HTTP proxy core, load balancing, connection pooling, caching, and SSL termination components. This glossary serves as your technical dictionary throughout the implementation journey.

Core Reverse Proxy Concepts

Reverse Proxy A server that sits in front of one or more backend servers, intercepting requests from clients and forwarding them to the appropriate backend. Unlike a forward proxy that acts on behalf of clients, a reverse proxy acts on behalf of the servers. Think of it as a skilled receptionist at a large company who receives all visitor requests and directs them to the right department based on availability and specialization.

Forward Proxy A proxy server that acts on behalf of clients, forwarding their requests to servers on the internet. The server sees the proxy's IP address rather than the client's original IP address. This is the traditional "proxy" that users configure in their browsers.

Backend Server An upstream server that actually processes client requests and generates responses. These are the real application servers that do the work, while the reverse proxy handles the complexity of client communication, load balancing, and optimization.

Upstream Server Another term for backend server, emphasizing that it's "upstream" in the request processing flow from the reverse proxy's perspective.

SSL Termination The process of decrypting SSL/TLS connections at the reverse proxy level, then forwarding the decrypted HTTP requests to backend servers over plain HTTP. This centralizes certificate management and reduces computational load on backend servers.

TLS Termination Modern terminology for SSL termination, as TLS (Transport Layer Security) has superseded SSL (Secure Sockets Layer) in practice, though the terms are often used interchangeably.

HTTP Protocol and Parsing

HTTP/1.1 The widely-supported HTTP protocol version that uses text-based headers and supports persistent connections through the `Connection: keep-alive` header. Requests and responses are processed sequentially on each connection.

HTTP/2 A binary protocol that multiplexes multiple request-response streams over a single connection. It includes header compression and server push capabilities, requiring more complex parsing logic than HTTP/1.1.

Stream-Based Parsing An incremental parsing approach that processes HTTP messages as data arrives over the network, rather than buffering the entire message before parsing. This is essential for handling large requests and responses without excessive memory usage.

State Machine A parsing approach where the parser maintains its current state (parsing request line, headers, body, etc.) and transitions between states based on input data. This enables robust handling of partial reads and malformed input.

Request Smuggling An attack that exploits differences in how front-end proxies and backend servers parse HTTP messages, particularly around Content-Length and Transfer-Encoding headers. Proper parsing and validation prevent these vulnerabilities.

Chunked Transfer Encoding An HTTP mechanism for sending message bodies in chunks when the total size is unknown at transmission start. Each chunk is prefixed with its size in hexadecimal, followed by the data and a trailing CRLF.

Content-Length Header HTTP header specifying the exact size of the message body in bytes. When present, it determines how much data to read for the complete message body.

Transfer-Encoding Header HTTP header indicating how the message body is encoded for transfer. The most common value is "chunked" for chunked transfer encoding.

Keep-Alive Connection HTTP/1.1 feature allowing multiple requests and responses to be sent over a single TCP connection, reducing the overhead of connection establishment and teardown.

Pipeline Depth The number of HTTP requests sent on a connection before waiting for responses. HTTP/1.1 pipelining allows multiple requests to be sent without waiting for each response, though it's rarely used in practice due to head-of-line blocking

issues.

Connection Management

Connection Pooling The practice of maintaining persistent connections to backend servers and reusing them for multiple requests. This eliminates the TCP handshake overhead for each request and improves performance.

Connection Lifecycle The states and transitions connections go through from establishment to cleanup, including idle, reading request, forwarding, reading response, writing response, and closing states.

Event-Driven Architecture An architecture pattern using asynchronous I/O and event loops to handle many concurrent connections efficiently. Events like data arrival, connection completion, and timeouts trigger appropriate handlers.

LIFO Strategy Last-In First-Out connection reuse strategy where the most recently used connection is selected first from the pool. This improves CPU cache locality and can reduce connection establishment overhead.

Connection State The current operational phase of a connection, such as `CONNECTION_IDLE`, `CONNECTION_READING_REQUEST`, `CONNECTION_FORWARDING`, `CONNECTION_READING_RESPONSE`, `CONNECTION_WRITING_RESPONSE`, or `CONNECTION_CLOSING`.

Resource Leak A programming error where allocated system resources (file descriptors, memory, connections) are not properly released, eventually exhausting available resources and causing system failure.

Timeout Management The practice of enforcing time limits on network operations to prevent connections from hanging indefinitely and consuming system resources.

Timer Wheel An efficient data structure for managing timeouts with O(1) insertion and deletion operations. It uses a circular array of time slots, each containing operations that expire at that time.

Non-Blocking I/O I/O operations that return immediately if no data is available, allowing a single thread to handle many connections by processing only those that are ready for I/O operations.

Edge-Triggered Events Event notification mode where events are delivered only when the state changes (e.g., from no data available to data available), requiring the application to process all available data before waiting for the next event.

Level-Triggered Events Event notification mode where events are delivered whenever the condition is true (e.g., data is available), making it easier to program but potentially less efficient than edge-triggered mode.

Load Balancing

Round-Robin A load balancing algorithm that distributes requests by cycling through backend servers in order. Each server receives an equal number of requests over time, regardless of their current load or capacity.

Least-Connections A load balancing algorithm that routes requests to the backend server with the fewest active connections. This helps balance load more dynamically than round-robin when requests have varying processing times.

Weighted Distribution Load balancing that accounts for different server capacities by assigning weights to each server. Servers with higher weights receive proportionally more requests than those with lower weights.

IP Hash A load balancing method that uses a hash of the client's IP address to consistently route requests from the same client to the same backend server, providing session affinity.

Session Affinity The practice of routing requests from the same client session to the same backend server, ensuring that session state stored on the server remains accessible across requests.

Sticky Sessions Another term for session affinity, emphasizing that client sessions "stick" to particular backend servers for the duration of the session.

Health Checking The process of periodically testing backend servers to determine their availability and readiness to handle requests. Failed health checks remove servers from the active pool.

Failure Threshold The number of consecutive health check failures required before marking a backend server as unhealthy and removing it from the load balancing rotation.

Success Threshold The number of consecutive health check successes required before marking a previously failed backend server as healthy and returning it to the load balancing rotation.

Graceful Degradation The ability to continue operating with reduced functionality when some backend servers fail, rather than experiencing complete system failure.

Connection Count The number of active connections currently being handled by a backend server, used by least-connections load balancing algorithms to make routing decisions.

Backend Selection The process of choosing which available backend server should handle an incoming request based on the configured load balancing algorithm and server health status.

Caching

Cache Hit A request for which a valid cached response exists and can be returned to the client without contacting the backend server. This improves response time and reduces backend load.

Cache Miss A request for which no cached response exists or the cached response is stale, requiring the proxy to forward the request to a backend server.

TTL (Time To Live) The duration for which a cached entry remains valid before it expires and must be refreshed from the backend server.

TTL Expiration The process of cached entries becoming stale due to exceeding their time-to-live limits, requiring revalidation or refresh from the backend.

Cache Key A unique identifier generated from request attributes (method, URL, headers) used to store and retrieve cached responses. Proper key generation is crucial for cache correctness.

Cache-Control Header HTTP header containing directives that specify caching behavior, including whether responses can be cached, for how long, and under what conditions.

ETag (Entity Tag) A unique identifier assigned to a specific version of a resource, used for cache validation through conditional requests to determine if cached content is still current.

Last-Modified Header HTTP header indicating when a resource was last changed, used for cache validation through conditional requests with If-Modified-Since headers.

Conditional Request HTTP requests that include headers like If-Modified-Since or If-None-Match to allow servers to respond with 304 Not Modified if the cached version is still current.

304 Not Modified HTTP status code indicating that a cached response is still valid, sent in response to conditional requests when the resource hasn't changed since the cached version.

Vary Header HTTP response header indicating which request headers were used to generate the response, affecting cache key generation to ensure proper cache segmentation.

LRU (Least Recently Used) A cache eviction policy that removes the least recently accessed entries when the cache reaches its capacity limit, keeping frequently accessed content in cache longer.

Cache Invalidation The process of removing or marking cached entries as invalid, either due to TTL expiration, explicit purge requests, or backend error conditions.

Cache Validation The process of checking whether a cached response is still valid by sending conditional requests to the backend server.

No-Cache Directive Cache-Control directive indicating that cached responses must be validated with the origin server before being served to clients.

No-Store Directive Cache-Control directive indicating that responses must not be stored in any cache, typically used for sensitive or highly dynamic content.

Max-Age Directive Cache-Control directive specifying the maximum time in seconds that a response may be cached before it becomes stale and requires revalidation.

SSL/TLS and Security

TLS Context The cryptographic configuration and state information for SSL/TLS connections, including certificates, private keys, cipher preferences, and protocol versions.

SSL Context Legacy term for TLS context, still commonly used even though TLS has replaced SSL in modern implementations.

SNI (Server Name Indication) A TLS extension that allows clients to specify which hostname they're connecting to, enabling a single IP address to serve multiple SSL certificates for different domains.

Certificate Chain The sequence of certificates from the server's leaf certificate up to a trusted root CA certificate, required for clients to verify the server's authenticity.

Wildcard Certificate An SSL certificate valid for multiple subdomains of a domain (e.g., *.example.com covers api.example.com, www.example.com, etc.).

Subject Alternative Name (SAN) X.509 certificate extension that allows a single certificate to be valid for multiple hostnames, providing an alternative to wildcard certificates.

Cipher Suite A combination of encryption algorithms used in TLS connections, including key exchange, authentication, bulk encryption, and message authentication algorithms.

Perfect Forward Secrecy A security property ensuring that session keys cannot be derived from long-term keys, so compromising long-term keys doesn't compromise past communication sessions.

AEAD Cipher Authenticated Encryption with Associated Data cipher that provides both confidentiality and authenticity in a single operation, preferred in modern TLS implementations.

TLS Handshake The initial negotiation process between client and server to establish encryption parameters, exchange certificates, and derive session keys for secure communication.

Certificate Validation The process of verifying that a server's certificate is valid, properly signed by a trusted CA, not expired, and matches the requested hostname.

Session Resumption TLS optimization that allows clients and servers to reuse previously established session parameters, reducing handshake overhead for subsequent connections.

Cipher Preference The order in which cipher suites are preferred during TLS negotiation, typically prioritizing stronger, more secure cipher suites over weaker alternatives.

Error Handling and Recovery

Cascade Failure A failure propagation pattern where the failure of one component triggers failures in dependent components, potentially causing widespread system failure.

Circuit Breaker A design pattern that prevents operations when error rates exceed configured thresholds, allowing systems to fail fast and recover more quickly.

Exponential Backoff A retry strategy where the delay between retry attempts increases exponentially, reducing load on failing systems and improving recovery chances.

Error Sanitization The practice of removing sensitive information (internal paths, stack traces, configuration details) from error messages shown to clients.

Graceful Shutdown The process of cleanly stopping a server by finishing in-flight requests, closing connections properly, and releasing resources before termination.

Health Check Endpoint A specific URL path that backend servers expose to indicate their readiness to handle requests, used by load balancers for health monitoring.

Retry Logic Automated mechanisms for retrying failed operations with appropriate delays and limits to handle transient failures without overwhelming systems.

Fallback Mechanism Alternative behavior implemented when primary systems fail, such as serving cached content when backends are unavailable.

Dead Letter Queue A storage mechanism for requests that cannot be processed successfully after multiple retry attempts, allowing for later analysis or manual intervention.

Bulkhead Pattern An isolation strategy that prevents failures in one part of the system from affecting other parts, similar to watertight compartments in ships.

Performance and Monitoring

Latency The time delay between sending a request and receiving a response, typically measured in milliseconds for HTTP operations.

Throughput The number of requests processed per unit time, typically measured in requests per second (RPS) for reverse proxy performance.

Connection Multiplexing The ability to handle multiple requests simultaneously over a single connection, particularly important in HTTP/2 implementations.

Keep-Alive Timeout The duration a connection remains open waiting for additional requests before being closed to free up resources.

Request Rate Limiting Controlling the number of requests accepted from clients over a specific time period to prevent overload and ensure fair resource usage.

Response Time Percentiles Statistical measures (P50, P95, P99) indicating the response time below which a certain percentage of requests are served, useful for understanding performance distribution.

Memory Footprint The amount of system memory used by the reverse proxy, important for capacity planning and resource optimization.

File Descriptor Usage The number of file descriptors (handles for files, sockets, etc.) used by the proxy, as each connection typically requires at least one file descriptor.

CPU Utilization The percentage of available CPU resources being used by the proxy, affecting its ability to handle concurrent connections and requests.

Bandwidth Utilization The amount of network bandwidth consumed by the proxy for client and backend communications, important for capacity planning.

Testing and Development

Unit Testing Testing individual components in isolation using mock implementations of dependencies to verify correct behavior under various conditions.

Integration Testing Testing interactions between multiple components to ensure they work correctly together and handle data flow properly.

Milestone Verification Predefined checkpoints that validate implementation progress, ensuring each milestone's acceptance criteria are met before proceeding.

Synthetic Testing Using artificially generated test data and scenarios to validate system behavior under controlled conditions.

Load Testing Testing system performance under high request volumes to identify bottlenecks and capacity limits.

Stress Testing Testing system behavior under extreme conditions beyond normal operating parameters to identify failure modes and recovery behavior.

Mock Implementation Fake components that simulate real behavior for testing purposes, allowing isolation of the component under test.

Test Harness A framework that automates test execution, result collection, and validation across multiple test scenarios.

Deterministic Testing Tests designed to produce predictable, repeatable outcomes regardless of execution timing or environment variations.

Regression Testing Re-running existing tests after code changes to ensure new modifications don't break previously working functionality.

Architecture and Design Patterns

Component Architecture A design approach that breaks the system into discrete, loosely-coupled components with well-defined responsibilities and interfaces.

Dependency Injection A design pattern where components receive their dependencies through constructor parameters or setters rather than creating them directly.

Observer Pattern A design pattern where components register to receive notifications about events or state changes in other components.

Factory Pattern A creational pattern that provides an interface for creating objects without specifying their concrete classes.

Singleton Pattern A design pattern ensuring only one instance of a class exists, commonly used for global configuration and resource managers.

State Pattern A behavioral pattern that allows objects to change their behavior based on internal state changes, useful for connection and parser state management.

Strategy Pattern A behavioral pattern that defines a family of algorithms (like load balancing strategies) and makes them interchangeable at runtime.

Facade Pattern A structural pattern that provides a simplified interface to a complex subsystem, hiding implementation details from clients.

Network and System Programming

Epoll A Linux system call for efficient I/O event notification that can monitor many file descriptors for events without the overhead of polling each one individually.

Kqueue A BSD system call similar to epoll, providing scalable event notification for file descriptors and other system events.

File Descriptor An operating system handle representing an open file, socket, or other I/O resource, with each process having a limited number available.

Socket Programming Network programming using BSD socket APIs to create, bind, listen, accept, connect, send, and receive data over TCP/UDP connections.

TCP Nodelay A socket option that disables Nagle's algorithm, reducing latency by immediately sending small packets rather than waiting to accumulate more data.

SO_REUSEADDR A socket option allowing immediate reuse of local addresses, particularly useful for server sockets that need to restart quickly.

Buffer Management Techniques for efficiently handling data buffers during network I/O operations, including allocation, reuse, and memory management strategies.

Memory Pool A memory management technique that pre-allocates large blocks of memory and suballocates from them to reduce allocation overhead and fragmentation.

Extension and Plugin Architecture

Extension Architecture A framework for adding new functionality to the reverse proxy without modifying core components, enabling modularity and customization.

Plugin System A modular architecture allowing runtime loading of additional features through dynamically loaded libraries or modules.

Event-Driven Hooks Callback mechanisms that allow extensions to intercept and modify request processing at specific points in the pipeline.

Rate Limiting Controlling request frequency from clients to prevent overload and ensure fair resource usage across multiple clients.

Web Application Firewall (WAF) Security extension that examines HTTP requests for malicious patterns and blocks potentially harmful traffic.

Authentication Extension Module responsible for verifying client identity and authorization before allowing access to backend services.

Monitoring Extension Module that collects metrics, logs, and observability data about proxy operations and performance.

Multi-Tenancy Architecture pattern for serving multiple isolated customers or applications from a single proxy instance with proper resource isolation.

Service Mesh Integration Features for integrating with distributed system coordination platforms that provide service discovery, load balancing, and observability.

Content Transformation Extensions that modify response content, headers, or format before delivering to clients, such as compression or format conversion.

Acronyms and Abbreviations

HTTP: HyperText Transfer Protocol **HTTPS**: HyperText Transfer Protocol Secure
SSL: Secure Sockets Layer **TLS**: Transport Layer Security **TCP**: Transmission Control Protocol **UDP**: User Datagram Protocol **DNS**: Domain Name System **IP**: Internet Protocol **URL**: Uniform Resource Locator **URI**: Uniform Resource Identifier **API**: Application Programming Interface **JSON**: JavaScript Object Notation **XML**: eXtensible Markup Language **MIME**: Multipurpose Internet Mail Extensions **CORS**: Cross-Origin Resource Sharing **CDN**: Content Delivery Network **DDoS**: Distributed Denial of Service **CA**: Certificate Authority **CRL**: Certificate Revocation List **OCSP**: Online Certificate Status Protocol **RSA**: Rivest-Shamir-Adleman (encryption algorithm) **AES**: Advanced Encryption Standard **SHA**: Secure Hash Algorithm **HMAC**: Hash-based Message

Authentication Code **CSRF**: Cross-Site Request Forgery **XSS**: Cross-Site Scripting **SQL**: Structured Query Language **REST**: Representational State Transfer **SOAP**: Simple Object Access Protocol **gRPC**: Google Remote Procedure Call **WebSocket**: Web Socket Protocol **SPDY**: SPeeDY (HTTP/2 predecessor) **QUIC**: Quick UDP Internet Connections **MTU**: Maximum Transmission Unit **MSS**: Maximum Segment Size **TTL**: Time To Live **QoS**: Quality of Service **SLA**: Service Level Agreement **RTO**: Request Time Out **RTT**: Round Trip Time **ACK**: Acknowledgment **SYN**: Synchronize **FIN**: Finish **PSH**: Push **RST**: Reset **URG**: Urgent

Implementation Guidance

This section provides practical guidance for implementing reverse proxy components with proper terminology usage throughout the codebase.

A. Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Parser	Manual state machine with `HttpParser`	ANTLR/Yacc generated parser
Event Loop	Basic epoll with `EventDispatcher`	libevent/libuv wrapper
SSL/TLS	OpenSSL with `SSLTermination`	BoringSSL or wolfSSL
Logging	Simple file logging with `LogLevel`	Structured logging with syslog
Configuration	INI file parsing with `ProxyConfig`	YAML/JSON with schema validation
Testing	Custom test framework	CUnit/Unity testing framework

B. Recommended File Structure

The codebase should be organized to reflect the component architecture and terminology used throughout this glossary:

```
reverse-proxy/
├── src/
│   ├── core/
│   │   ├── proxy_server.c          ← Main ProxyServer implementation
│   │   ├── proxy_server.h          ← Public ProxyServer interface
│   │   └── proxy_config.c          ← ProxyConfig loading and validation
│   ├── http/
│   │   ├── http_parser.c           ← HttpParser state machine
│   │   ├── http_request.c          ← HttpRequest manipulation
│   │   └── http_response.c         ← HttpResponse handling
│   ├── connection/
│   │   ├── connection_manager.c    ← ConnectionManager with event-driven I/O
│   │   ├── connection_pool.c       ← ConnectionPool per backend
│   │   └── timer_wheel.c           ← TimerWheel timeout management
│   ├── load_balancer/
│   │   ├── load_balancer.c         ← LoadBalancer algorithms
│   │   ├── backend_server.c        ← BackendServer health checking
│   │   └── health_checker.c        ← Health checking implementation
│   ├── cache/
│   │   ├── cache_engine.c          ← CacheEngine with LRU eviction
│   │   ├── cache_control.c         ← CacheControl header parsing
│   │   └── lru_list.c             ← LRUList implementation
│   ├── ssl/
│   │   ├── ssl_termination.c       ← SSLTermination with SNI support
│   │   └── ssl_utils.c             ← Certificate loading utilities
│   ├── error/
│   │   ├── error_handler.c         ← ErrorHandler for recovery coordination
│   │   └── recovery_manager.c      ← RecoveryManager for failure handling
│   └── utils/
│       ├── buffer.c                ← Buffer management utilities
│       ├── hashtable.c             ← HashTable implementation
│       └── logger.c                ← Logging with LogLevel support
├── include/
│   └── reverse_proxy/              ← Public header files
├── tests/
│   ├── unit/                       ← Unit tests for individual components
│   ├── integration/                ← Integration tests for component interactions
│   └── mocks/                      ← MockSocket and test utilities
├── config/
│   └── proxy.conf                  ← Example ProxyConfig file
└── docs/
    └── terminology.md              ← This glossary for reference
```

C. Terminology Usage in Code

When implementing components, use the exact terminology from this glossary in comments, variable names, and function names. This creates consistency between the design documentation and implementation:

```c
// Example: Connection state management using proper terminology
typedef enum {
    CONNECTION_IDLE,              // Connection waiting for new request
    CONNECTION_READING_REQUEST,   // Receiving HTTP request from client
    CONNECTION_FORWARDING,        // Forwarding request to backend server
    CONNECTION_READING_RESPONSE,  // Receiving response from backend
    CONNECTION_WRITING_RESPONSE,  // Sending response back to client
    CONNECTION_CLOSING           // Graceful connection shutdown
} ConnectionState;

// Example: Load balancing algorithm enumeration
typedef enum {
    LB_ROUND_ROBIN,              // Round-robin backend selection
    LB_LEAST_CONNECTIONS,        // Least-connections algorithm
    LB_WEIGHTED_ROUND_ROBIN,     // Weighted distribution
    LB_IP_HASH                   // IP hash for session affinity
} LoadBalancingAlgorithm;
```

D. Common Implementation Terminology Mistakes

Incorrect Term	Correct Term	Explanation
"Upstream proxy"	"Reverse proxy"	Avoids confusion with forward proxies
"SSL connection"	"TLS connection"	Uses modern terminology
"Connection reuse"	"Connection pooling"	More specific technical term
"Request caching"	"Response caching"	Clarifies what is actually cached
"Server selection"	"Backend selection"	Distinguishes from client/proxy server
"Health monitoring"	"Health checking"	Standard industry terminology
"Certificate reload"	"Certificate rotation"	Different concepts
"Error recovery"	"Failure recovery"	More precise terminology

E. Documentation and Comment Guidelines

Use consistent terminology in all code comments and documentation:

```c
/**
 * connection_manager_acquire_backend - Acquire backend connection from pool
 * @manager: ConnectionManager instance
 * @backend: Target BackendServer for connection
 * @timeout_ms: Connection timeout in milliseconds
 *
 * Attempts to acquire a connection from the connection pool for the specified
 * backend server. If no idle connections are available, creates a new connection
 * up to the pool's maximum limit. Uses LIFO strategy for cache locality.
 *
 * Returns: Connection pointer on success, NULL on failure
 */
Connection* connection_manager_acquire_backend(ConnectionManager* manager,
                                               BackendServer* backend,
                                               int timeout_ms);
```

F. Debugging Terminology Reference

When debugging reverse proxy issues, use precise terminology to communicate problems effectively:

Problem Area	Correct Terminology	Avoid
Connection issues	"Connection pool exhausted"	"Out of connections"
SSL problems	"TLS handshake failure"	"SSL error"
Parsing errors	"HTTP parser state machine error"	"Bad request"
Load balancing	"Backend server health check failure"	"Server down"
Caching	"Cache miss due to TTL expiration"	"Cache not working"
Performance	"High connection establishment latency"	"Slow connections"

This terminology consistency ensures clear communication between team members and accurate problem diagnosis throughout the development process.