API Gateway: Design Document

Overview

This system provides a centralized API Gateway that manages, secures, and observes traffic between external clients and internal backend services. The key architectural challenge it solves is abstracting common cross-cutting concerns like routing, authentication, and transformation away from individual services, enabling a unified and scalable entry point for an API ecosystem.

> 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.

> **Milestone(s):** This foundational context applies to all milestones, establishing the problem space and solution approach.

Context and Problem Statement

Modern software architecture is increasingly dominated by microservices — decomposing monolithic applications into small, independent services that can be developed, deployed, and scaled independently. While this approach yields significant benefits in team autonomy and technical agility, it introduces a critical operational challenge: **managing the interaction between a multitude of client applications (mobile apps, web frontends, third-party partners) and a sprawling ecosystem of backend services.**

Without a centralized coordination layer, each client must directly communicate with each required service. This creates a complex, brittle, and inefficient network topology akin to a city without traffic lights or road signs. The **API Gateway** pattern emerges as the definitive solution to this problem, acting as the single, unified entry point for all client traffic. It abstracts the internal service landscape, providing a consistent, secure, and observable interface to the outside world.

Mental Model: The Hotel Concierge

Imagine a grand hotel with hundreds of specialized departments: the kitchen, housekeeping, the spa, room service, valet parking, and business center. A guest arriving at the hotel does not—and should not—need to know the internal phone numbers, locations, or protocols for each department. Instead, they interact solely with the **hotel concierge**.

The concierge performs several critical functions that perfectly mirror an API Gateway's role:

1. **Single Point of Contact & Routing:** The guest presents a request ("I'd like a massage at 3 PM"). The concierge receives this request, determines it belongs to the Spa department, and routes it to the correct internal team. In API terms, an HTTP request for `/api/bookings` is routed to the `booking-service`.
2. **Protocol Translation & Request Augmentation:** The guest might speak in casual terms ("a rubdown"). The concierge translates this into the spa's formal booking language, adding necessary details like the guest's room number. Similarly, the gateway can transform a client's JSON payload into a different structure or protocol (e.g., gRPC) expected by the backend, and inject standard headers (like a user ID).
3. **Security & Access Control:** The concierge verifies the guest's identity (room key) before fulfilling any request and knows which amenities (e.g., the executive lounge) the guest is authorized to access. The gateway centrally validates API keys, JWTs, and enforces role-based access, preventing unauthorized traffic from ever reaching internal services.
4. **Load Management & Resilience:** If the spa is fully booked, the concierge doesn't let the guest wait indefinitely; they suggest an alternative time or service. The gateway implements rate limiting to prevent clients from overwhelming services and circuit breakers to stop sending traffic to a failing service, allowing it time to recover.
5. **Observability & Auditing:** The concierge's logbook records every guest interaction—what was requested, when, and the outcome. This data is invaluable for understanding hotel operations. The gateway provides centralized logging, metrics, and tracing for every API request, offering a complete picture of system health and client behavior.

This mental model clarifies the gateway's **fundamental purpose**: to act as a dedicated intermediary that handles all cross-cutting concerns, allowing backend services to focus purely on business logic and clients to remain blissfully unaware of the complex distributed system behind the facade.

The Pain of Direct Service-to-Client Communication

Operating without an API gateway forces teams to distribute critical functionality across every service or burden clients with complex integration logic. This leads to several acute pain points:

- **Duplicated Cross-Cutting Logic:** Every microservice must independently implement authentication, authorization, rate limiting, request validation, and logging. This violates the DRY (Don't Repeat Yourself) principle, increasing development time, codebase size, and the risk of inconsistent or buggy implementations. Updating a security library or JWT validation logic requires a coordinated rollout across dozens of services—a high-risk, slow process.
- **Tight Client-Service Coupling:** Clients must be hardcoded with the network locations (hostnames, ports) of individual services. Any change in service decomposition (splitting a service) or infrastructure (moving to a new cloud region) forces a corresponding update and redeployment of all client applications, crippling backend agility.

- **Inconsistent and Fragmented Observability:** Without a central point of data collection, understanding system-wide performance requires aggregating logs and metrics from every service. Correlating a single user's request flow across multiple services is notoriously difficult, turning debugging and performance analysis into a forensic nightmare.
- **Inefficient Client-Side Implementations:** Clients often need data from multiple services to render a single view (e.g., a user profile page with data from `user-service`, `orders-service`, and `preferences-service`). Without a gateway, the client must make multiple sequential network calls, leading to latency, complex error handling, and chatty network traffic. The client device (e.g., a mobile phone) bears the computational burden of aggregating this data.
- **Security Vulnerabilities and Inconsistent Enforcement:** Edge security policies are only as strong as the weakest service's implementation. A single service team overlooking a security best practice (e.g., failing to validate a token's audience claim) can create a breach vector. Centralized enforcement at the gateway ensures a uniform, auditable security perimeter.
- **Operational Overhead for Polyglot Concerns:** In a polyglot architecture (using Go, Java, Node.js, etc.), implementing non-functional requirements like circuit breaking or distributed tracing requires finding, integrating, and maintaining different libraries for each language stack, multiplying operational complexity.

The cumulative effect is a system that is **brittle, hard to operate, slow to evolve, and expensive to maintain**. The API gateway directly addresses each of these pains by consolidating common capabilities into a single, robust layer.

Existing Approaches and Trade-offs

When deciding how to implement an API gateway, architects typically evaluate three primary avenues, each with distinct trade-offs in flexibility, operational complexity, and control.

Option 1: Off-the-Shelf API Gateway (e.g., Kong, Tyk, Apache APISIX) These are dedicated, pre-built gateway applications. You configure them via APIs, YAML files, or a dashboard to define routes, plugins, and policies. They are feature-rich, battle-tested, and often open-source.

Option 2: Configurable Proxy as Gateway (e.g., NGINX, Envoy Proxy, HAProxy) These are high-performance, general-purpose proxies that can be configured (via Lua scripts, Envoy filters, or complex NGINX configuration) to perform gateway-like functions such as routing, authentication, and rate limiting. They sit closer to the networking layer.

Option 3: Build a Custom Gateway (This Project) This involves developing a purpose-built gateway application from the ground up, typically using a general-purpose programming language like Go, Java, or Rust. It provides ultimate control over behavior, data models, and integration with internal systems.

The following table analyzes the key trade-offs between building a custom gateway and using an off-the-shelf solution, helping to frame the rationale for this project's educational and practical goals.

> **Decision: Build vs. Buy for Educational Implementation**
>
> - **Context**: We need an API gateway for a learning project where understanding the underlying mechanics of routing, transformation, and security is a primary objective. We must choose between implementing core algorithms ourselves or delegating them to a pre-built tool.
> - **Options Considered**:
>   1. **Use Kong or Tyk**: Configure a mature, open-source gateway.
>   2. **Use NGINX/Envoy with Custom Scripts**: Leverage a high-performance proxy's extensibility.
>   3. **Build a Custom Gateway in Go/Rust**: Develop the gateway application from first principles.
> - **Decision**: Build a custom gateway.
> - **Rationale**:
>   - **Deep Learning**: The primary goal is pedagogical. Implementing routing trees, JWT validation, circuit breaker state machines, and plugin systems provides irreplaceable, low-level understanding that configuration alone cannot offer.
>   - **Unconstrained Design Exploration**: A custom build allows us to structure the codebase, data models, and extension points exactly as we wish to illustrate clean architectural patterns, without being boxed in by a vendor's abstraction model.
>   - **Targeted Simplicity**: We can build *only* the features needed for our milestones, avoiding the complexity and cognitive overhead of a full-blown enterprise product with hundreds of plugins and configuration options.
>   - **Language Alignment**: Using Go or Rust aligns with modern cloud-native development practices and allows us to demonstrate concurrency patterns, efficient networking, and safe memory management.
> - **Consequences**:
>   - **Increased Initial Effort**: We must build what others provide out-of-the-box.
>   - **Maintenance Burden**: We own the bug fixes, security patches, and performance optimizations.
>   - **Limited Feature Breadth**: Our gateway will lack the vast ecosystem of plugins and integrations available for Kong or NGINX.
>   - **Operational Risk**: A bespoke implementation may have undiscovered edge-case bugs or scalability limits that battle-tested software has already ironed out.

Approach	Pros	Cons	Best For
Off-the-Shelf (Kong/Tyk)	• **Rapid Deployment**: Feature-complete out of the box. • **Robust & Scalable**: Proven in high-scale production. • **Rich Ecosystem**: Many plugins (auth, logging, transformations). • **Active Community**: Regular updates and support.	• **Black Box Abstraction**: Hides implementation details, limiting learning. • **Configuration Complexity**: Deep features come with steep learning curves for YAML/APIs. • **Limited Customization**: Difficult to extend beyond the plugin framework. • **Operational Overhead**: Requires managing database (PostgreSQL) and control-plane processes.	Production environments where time-to-market, stability, and feature breadth are paramount, and internal implementation knowledge is not a primary concern.
Configurable Proxy (NGINX/Envoy)	• **Extreme Performance**: Optimized C++ (Envoy) core handles millions of requests/sec. • **Flexible Configuration**: Powerful DSL (NGINX) or programmable filters (Envoy). • **Cloud-Native Standard**: Envoy is the de-facto data plane for service meshes (Istio). • **Fine-Grained Control**: Low-level access to connection and stream lifecycle.	• **Steep Learning Curve**: Complex configuration syntax or filter chain programming model. • **Logic in Configuration**: Complex business logic can lead to cumbersome, hard-to-debug config files or Lua scripts. • **Debugging Difficulty**: Troubleshooting requires deep understanding of proxy internals. • **Still a Layer of Abstraction**: Core algorithms (load balancing, health checking) are implemented in C++, not exposed for modification.	Teams needing maximum performance and deep integration with cloud-native ecosystems (Kubernetes, Istio), who have expertise in proxy configuration.
Build Custom (This Project)	• **Maximum Understanding & Control**: Complete visibility and ownership of every line of code. • **Tailored Design**: Architecture and APIs can be perfectly fit to specific use cases and team conventions. • **Excellent Educational Value**: Forces deep engagement with networking, security, and distributed systems concepts. • **Simplified Stack**: No external dependencies beyond standard library and chosen language ecosystem.	• **Significant Development Time**: Reinventing well-understood wheels. • **Testing & Hardening Burden**: Must build reliability and security from scratch. • **Long-Term Maintenance**: The team becomes solely responsible for all bugs, security patches, and upgrades. • **Risk of Immaturity**: Likely to have undiscovered edge cases and scaling limitations initially.	**Learning projects** (like this one), niche use cases unsupported by existing tools, or organizations with unique requirements that justify the long-term investment.

For the purposes of this project—**deep, hands-on learning of distributed system fundamentals**—the "Build Custom" approach is unequivocally the right choice. It transforms the API gateway from a mysterious infrastructure component into a comprehensible collection of algorithms and state machines that we will design, implement, and test ourselves. The subsequent sections of this document detail that very design.

Implementation Guidance

While this section is primarily conceptual, establishing a solid project foundation is critical for the implementation work in later milestones.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Rationale
HTTP Library	`net/http` (Go standard library)	`fasthttp` (third-party)	`net/http` is robust, well-understood, and perfectly adequate for learning. `fasthttp` offers extreme performance but has a different API and fewer compatible middleware libraries.
Configuration Format	YAML (via `gopkg.in/yaml.v3`)	JSON, or DSL (HCL)	YAML is human-friendly for complex nested structures like route definitions. JSON is simpler to parse but harder to write by hand.
Internal Communication	Go channels, `sync` package	Message queue (NATS, Redis Pub/Sub)	For a single-process gateway, Go's superb concurrency primitives are sufficient. A distributed gateway would need a pub/sub system for cross-node coordination.

B. Recommended File/Module Structure Start by organizing the codebase to separate concerns and allow for incremental development. This structure mirrors the high-level architecture we will define later.

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go                  # Application entry point
├── internal/
│   ├── config/
│   │   ├── config.go                # Main Config struct
│   │   ├── loader.go                # Load config from file/env
│   │   └── validator.go             # Validate configuration
│   ├── router/                      # (Milestone 1) Routing core
│   │   ├── router.go                # Router interface & radix tree impl
│   │   ├── upstream.go              # Upstream & Endpoint definitions
│   │   ├── balancer.go              # Round-robin/weighted load balancer
│   │   └── healthcheck.go           # Passive/active health checks
│   ├── middleware/                  # Processing pipeline components
│   │   ├── chain.go                 # Middleware chaining logic
│   │   ├── context.go               # RequestContext struct
│   │   ├── auth/                    # (Milestone 3) Auth layer
│   │   │   ├── authenticator.go
│   │   │   ├── jwt_validator.go
│   │   │   └── apikey_validator.go
│   │   ├── transform/               # (Milestone 2) Transformation
│   │   │   ├── transformer.go
│   │   │   ├── header_modifier.go
│   │   │   └── body_transformer.go
│   │   └── observe/                 # (Milestone 4) Observability
│   │       ├── logger.go
│   │       ├── metrics.go
│   │       └── tracing.go
│   ├── proxy/                       # Reverse proxy engine
│   │   └── reverseproxy.go          # Wraps httputil.ReverseProxy
│   └── plugin/                      # (Milestone 4) Plugin system
│       ├── plugin.go                # Plugin interface
│       ├── manager.go               # Plugin registry & lifecycle
│       └── examples/
│           ├── ratelimit.go
│           └── cors.go
├── pkg/
│   └── httputil/                    # Shared HTTP utilities (safe for external use)
│       ├── requestid.go             # Generate request IDs
│       └── headers.go               # Header manipulation helpers
├── configs/
│   └── example-gateway.yaml         # Example configuration file
├── go.mod
└── README.md
```

C. Infrastructure Starter Code To begin, create the foundational configuration loading and HTTP server setup. This code is provided complete and ready to use, allowing you to focus on the core gateway logic in later milestones.

File: `internal/config/config.go`

```go
package config

// Route defines a rule for matching incoming requests to an upstream service.
type Route struct {
    ID          string            `yaml:"id"`
    Description string            `yaml:"description,omitempty"`
    Match       RouteMatch        `yaml:"match"`
    UpstreamID  string            `yaml:"upstream_id"`
    Plugins     []PluginConfig    `yaml:"plugins,omitempty"`
}

// RouteMatch specifies the conditions for a request to match this route.
type RouteMatch struct {
    Path    string            `yaml:"path"`            // e.g., "/api/users/**"
    Method  string            `yaml:"method,omitempty"`// e.g., "GET", "POST"
    Headers map[string]string `yaml:"headers,omitempty"` // e.g., "Host": "api.example.com"
}

// Upstream represents a backend service with multiple endpoints.
type Upstream struct {
    ID        string     `yaml:"id"`
    Name      string     `yaml:"name"`
    Endpoints []Endpoint `yaml:"endpoints"`
    // LoadBalancing strategy will be added in Milestone 1
}

// Endpoint is a single instance of a backend service.
type Endpoint struct {
    ID   string `yaml:"id"`
    URL  string `yaml:"url"`  // e.g., "http://localhost:8081"
    // Weight, health status will be added in Milestone 1
}

// PluginConfig holds the configuration for a specific plugin.
type PluginConfig struct {
    Name   string                 `yaml:"name"`   // e.g., "rate-limiting"
    Config map[string]interface{} `yaml:"config"` // Plugin-specific settings
}

// Config is the root configuration structure.
type Config struct {
    ListenAddr string     `yaml:"listen_addr"` // e.g., ":8080"
    Routes     []Route    `yaml:"routes"`
```

```
    Upstreams  []Upstream `yaml:"upstreams"`
}
```

File: `internal/config/loader.go`

```go
package config

import (
    "fmt"
    "os"

    "gopkg.in/yaml.v3"
)

// LoadFromFile reads and parses a YAML configuration file.
func LoadFromFile(filepath string) (*Config, error) {
    data, err := os.ReadFile(filepath)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

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

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

    return &cfg, nil
}

// validate performs basic sanity checks on the configuration.
func validate(cfg *Config) error {
    if cfg.ListenAddr == "" {
        return fmt.Errorf("listen_addr is required")
    }
    // Check for duplicate route IDs
    seen := make(map[string]bool)
    for _, route := range cfg.Routes {
        if seen[route.ID] {
            return fmt.Errorf("duplicate route ID: %s", route.ID)
        }
        seen[route.ID] = true
    }
    // TODO: Add more validation (e.g., upstream references)
    return nil
```

```
}
```

File: `cmd/gateway/main.go`

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "api-gateway/internal/config"
)

func main() {
    // 1. Load configuration
    cfg, err := config.LoadFromFile("configs/gateway.yaml")
    if err != nil {
        log.Fatalf("Failed to load config: %v", err)
    }

    // 2. Initialize core components (will be built in later milestones)
    // router := router.NewRouter(cfg.Routes, cfg.Upstreams)
    // middlewareChain := middleware.NewChain()

    // 3. Setup HTTP server with a placeholder handler
    srv := &http.Server{
        Addr:    cfg.ListenAddr,
        Handler: http.HandlerFunc(placeholderHandler),
        // Timeouts are important for production
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
        IdleTimeout:  120 * time.Second,
    }

    // 4. Graceful shutdown setup
    done := make(chan bool)
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)

    go func() {
        <-quit
        log.Println("Shutdown signal received...")
```

```
        ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
        defer cancel()

        if err := srv.Shutdown(ctx); err != nil {
            log.Fatalf("Forced shutdown: %v", err)
        }
        close(done)
    }()

    // 5. Start the server
    log.Printf("Starting API Gateway on %s", cfg.ListenAddr)
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        log.Fatalf("Server failed: %v", err)
    }

    <-done
    log.Println("Server stopped gracefully")
}

// placeholderHandler will be replaced with the real gateway pipeline.
func placeholderHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusNotImplemented)
    w.Write([]byte(`{"message": "API Gateway is under construction"}`))
}
```

D. Core Logic Skeleton Code For this foundational section, the core logic is simply the configuration loading and server bootstrapping, which is provided above as complete starter code. The real core components (Router, Transformer, etc.) will be sketched in their respective sections.

E. Language-Specific Hints (Go)

- Use `go mod init api-gateway` to initialize your Go module.
- The `net/http` package is your foundation. Study `httputil.ReverseProxy` as it will be central to Milestone 1.
- Use `context.Context` for request-scoped data and cancellation propagation throughout your middleware chain.
- For configuration, `gopkg.in/yaml.v3` is a stable, widely-used YAML parser. Avoid the v2 package due to known issues.

F. Milestone Checkpoint After setting up the project structure and starter code:

1. Run `go mod tidy` to fetch dependencies.
2. Create a minimal `configs/gateway.yaml`:

```yaml
listen_addr: ":8080"
routes: []
upstreams: []
```

3. Run the gateway: `go run cmd/gateway/main.go`
4. Expected output: `Starting API Gateway on :8080`
5. Verify by sending a request: `curl http://localhost:8080/`
   - **Expected Response**: `{"message": "API Gateway is under construction"}` with HTTP status 501.
6. Send a shutdown signal with `Ctrl+C`. The server should log "Shutdown signal received..." and "Server stopped gracefully".

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Failed to load config: ..."	Config file not found or invalid YAML.	Check the file path and YAML syntax (use a linter).	Ensure `configs/gateway.yaml` exists and is valid YAML.
Server starts but crashes immediately	Port may already be in use.	Look for `bind: address already in use` error.	Change `listen_addr` in config or find and stop the process using the port.
`go mod tidy` fails	Network issue or incompatible module version.	Check error message; often related to proxy or sum mismatch.	Set GOPROXY if needed, or delete `go.sum` and try again.

Goals and Non-Goals

> **Milestone(s):** This foundational section applies to all milestones, establishing the scope and boundaries of the entire API Gateway project.

A well-scoped project requires clear boundaries to avoid scope creep and ensure focused development. This section defines the explicit capabilities the gateway **must** provide and the explicit boundaries it **will not** cross. This serves as the project's "constitution"—a reference for all future architectural decisions, feature requests, and trade-offs.

Goals (What it MUST do)

The gateway is conceived as the **unified, intelligent entry point** for all client traffic destined to backend microservices. Its primary goals are to abstract and centralize cross-cutting concerns that would otherwise be duplicated across every service, thereby reducing operational complexity, improving security posture, and enabling consistent observability.

The following table enumerates the core functional goals, mapping each to its primary milestone of implementation and the key user need it addresses.

Goal Category	Specific Capability	Milestone	Purpose & User Need
Intelligent Routing & Load Distribution	**Path-based routing**: Direct requests to different backend services based on URL patterns (e.g., `/api/users/*` to the user service, `/api/orders/*` to the order service).	1	Enables a unified API facade for clients while allowing independent deployment and scaling of backend services.
	Host/header-based routing: Route traffic based on the HTTP `Host` header or other request headers (e.g., `X-API-Version: v2`).	1	Supports multi-tenancy, API versioning, and canary deployments by routing traffic to different service versions based on request characteristics.
	Load balancing: Distribute incoming requests across multiple healthy instances of a backend service using strategies like round-robin.	1	Improves availability and scalability by preventing any single backend instance from being overloaded.
	Health checking & circuit breaking: Automatically detect failing backend instances and stop sending them traffic, providing fault isolation and graceful degradation.	1	Increases system resilience; prevents a single failing service from causing cascading failures and degrading the client experience.
Request/Response Mediation	**Header manipulation**: Add, remove, or modify HTTP headers on requests (to backends) and responses (to clients).	2	Allows the gateway to inject context (like user IDs), enforce security policies (like HSTS headers), or normalize communication between clients and backends.
	Request/response transformation: Modify the body of requests and responses, such as renaming JSON fields, filtering data, or converting between formats.	2	Decouples the internal API models of backend services from the public API contract exposed to clients, enabling backend refactoring without breaking clients.
	URL rewriting: Change the path or query parameters of a request before it is forwarded to the backend.	2	Allows backends to use internal-friendly URL structures while presenting a cleaner, more consistent external API.
	Request aggregation: Combine calls to multiple backend services into a single response for the client.	2	Reduces client-side complexity and chattiness for mobile or web clients that need data from several services to render a single view.
Centralized Security	**Authentication**: Validate client credentials (JWT, API keys, OAuth2 tokens) at the gateway before any request reaches business logic.	3	Eliminates the need for each service to implement and maintain authentication logic, ensuring a consistent and auditable security perimeter.
	Authorization context propagation: Extract claims (user ID, roles) from validated credentials and inject them into headers forwarded to backends.	3	Provides backends with the necessary user context to make authorization decisions without re-validating tokens, simplifying backend logic.
	Rate limiting: Enforce request quotas per client, API key, or IP address to prevent abuse and ensure fair usage.	3	Protects backend services from being overwhelmed by excessive traffic, whether malicious or accidental.
Operational Observability	**Structured request logging**: Log essential details (timestamp, client IP, method, path, status, latency) for every request in a machine-parsable format.	4	Provides an audit trail for compliance and the raw data for debugging and analyzing API usage patterns.
	Metrics exposition: Expose key performance indicators (request rate, error rate, latency distribution) in a standard format like Prometheus.	4	Enables real-time monitoring, alerting, and capacity planning based on the gateway's traffic and health.
	Distributed tracing: Propagate and emit trace identifiers that allow a single request's journey through the gateway and into backends to be reconstructed.	4	Critical for diagnosing performance issues (slow requests) in a distributed system by identifying which service or network hop is the bottleneck.
Extensibility & Operability	**Plugin architecture**: Allow new cross-cutting functionality (custom auth, logging, etc.) to be added as plugins without modifying the gateway's core.	4	Future-proofs the gateway, enabling teams to add custom business logic and integrate with proprietary systems without forking the main codebase.
	Dynamic configuration: Apply changes to routing rules, upstreams, and plugin configuration without requiring a process restart or causing dropped connections.	4	Enables rapid incident response (e.g., rerouting traffic away from a failing backend) and continuous deployment of API changes with zero downtime.

> **Architectural Insight**: The goals are ordered to reflect a dependency stack. You cannot have reliable transformation (Milestone 2) without a stable routing foundation (Milestone 1). Similarly, layering on security (Milestone 3) requires the request manipulation capabilities built in Milestone 2 to inject auth context. Observability (Milestone 4) is built last because it instruments the stable pipeline created by the first three milestones.

Non-Goals (What it will NOT do)

Equally important to defining what the system *will* do is explicitly stating what it *will not* do. These non-goals guard against **scope creep**—the tendency for projects to accumulate features until they become unmanageable "kitchen sink" solutions. They also clarify the system's role within a broader architecture, highlighting where other dedicated systems should be used.

Non-Goal Category	Specific Exclusion	Rationale & Guidance
Service Mesh Functionality	The gateway will **not** act as a **sidecar proxy** deployed alongside every service instance. It will not manage service-to-service (east-west) traffic, only north-south (client-to-service) traffic.	**Rationale**: Service meshes (e.g., Istio, Linkerd) are specialized for the complex problem of securing and observing communication *between* services. The API Gateway is the entry point *from outside*. Combining both roles in one component creates a monolithic, overly complex system. **Guidance**: For internal service communication, implement a dedicated service mesh. The gateway and mesh are complementary patterns.
Full API Management Platform	The gateway will **not** provide a **developer portal**, API key lifecycle management UI, or monetization/billing features.	**Rationale**: API Management platforms (e.g., Apigee, Kong Enterprise) bundle the gateway runtime with extensive business-facing tools for API productization. Our focus is the high-performance runtime. **Guidance**: The gateway can be *integrated with* a separate API management portal. Configuration (`Route` , `Upstream` , `PluginConfig`) can be generated by such a system and pushed to the gateway.
Persistent Data Store	The gateway will **not** be a system of record. It will not durably store API logs, metrics, or application data beyond in-memory caches (e.g., for rate limiting counters or validated tokens).	**Rationale**: Adding databases for persistence dramatically increases operational complexity (backups, scaling, failover) and moves the gateway away from its stateless, ephemeral nature. **Guidance**: Logs should be streamed to a dedicated logging system (e.g., Loki, Elasticsearch). Metrics should be scraped by Prometheus. Any required persistent state (like API key validity) must be queried from an external authoritative source (like a database or auth server).
Business Logic Execution	The gateway will **not** execute application-specific business rules, call external third-party APIs on behalf of the client, or perform complex data aggregations that require joins across multiple domain models.	**Rationale**: The gateway's role is infrastructure—managing the *flow* of requests. Embedding business logic creates tight coupling, makes the gateway difficult to test, and turns it into a bottleneck for development. **Guidance**: Complex aggregations or orchestration should be handled by a dedicated **backend-for-frontend (BFF)** service or an orchestration layer behind the gateway. The gateway's request aggregation is limited to simple, parallel fan-out requests.
Stateful Session Management	The gateway will **not** maintain sticky sessions for clients or implement server-side session stores.	**Rationale**: Sticky sessions (session affinity) break the stateless load balancing model, reduce resilience (if a backend dies, its sessions are lost), and complicate horizontal scaling. **Guidance**: Backend services should be stateless. If session state is required, it should be stored in an external, shared cache (like Redis) accessible by all backend instances. The gateway can, however, route based on a session cookie if absolutely necessary.
Data Validation & Schema Enforcement	The gateway will **not** validate request/response payloads against OpenAPI/Swagger schemas or enforce strict data type contracts.	**Rationale**: While possible, deep schema validation adds significant latency and complexity. The primary contract enforcement should remain at the service boundary. **Guidance**: The gateway can perform *lightweight* structural checks (e.g., "is the body valid JSON?") but detailed validation ("does the `email` field match RFC 5322?") is the responsibility of the backend service receiving the request.
Web Application Firewall (WAF)	The gateway will **not** provide deep packet inspection, SQL injection detection, or other layer 7 attack mitigation typically found in a WAF.	**Rationale**: WAFs are specialized security appliances with regularly updated threat signatures. Building a robust one is a massive undertaking. **Guidance**: Deploy a dedicated WAF (e.g., ModSecurity, cloud provider WAF) in front of the gateway. The gateway's security role is authentication and coarse rate limiting, not threat detection.
Protocol Translation Beyond REST/gRPC	The gateway will **not** translate between disparate protocols like SOAP/XML to REST/JSON, MQTT to HTTP, or handle WebSocket connection pooling and protocol upgrading.	**Rationale**: Each protocol translation is a significant feature. The initial focus is the most common microservice communication patterns (HTTP/1.1, HTTP/2, gRPC). **Guidance**: For other protocols, use a dedicated protocol adapter service behind the gateway or a different gateway specialized for that protocol (e.g., an MQTT broker).

> **Design Principle**: "Do one thing and do it well." The API Gateway's "one thing" is being a **configurable, high-performance reverse proxy with a middleware pipeline for cross-cutting concerns**. By adhering to these non-goals, we keep the core simple, maintainable, and focused on its primary value proposition.

Relationship to Mental Models: The **Concierge** (from the Problem Statement) does not cook the meal, manage the hotel's finances, or clean the rooms. It receives guests, understands their needs, and directs them to the right specialized service. Similarly, the gateway receives requests, enforces policies, and routes them—it does not execute the core business service logic.

Conventions Summary

This section introduced and reinforced the following key terms and concepts central to the design document:

- **Cross-cutting concerns**: Functionality like authentication, logging, and rate limiting that is needed by many services but is not part of their core business logic.
- **North-South traffic**: Traffic flowing between external clients and internal services (through the gateway).
- **East-West traffic**: Traffic flowing between internal services (handled by a service mesh).
- **Scope creep**: The uncontrolled expansion of a project's goals and features beyond the original intent.
- **Stateless**: A design where each request can be handled independently, without relying on data stored from previous requests.
- **Backend-for-Frontend (BFF)**: A dedicated backend service tailored to the needs of a specific client type (e.g., mobile app, web app) that often sits behind the API gateway.

The named data structures (`Route` , `Upstream` , etc.) and functions (`LoadFromFile` , `validate`) from the prerequisites are the building blocks for achieving the stated goals, but their detailed design is covered in subsequent sections.

High-Level Architecture

> **Milestone(s):** This foundational architecture applies to all milestones, establishing the overall component structure and data flow for the entire API Gateway system.

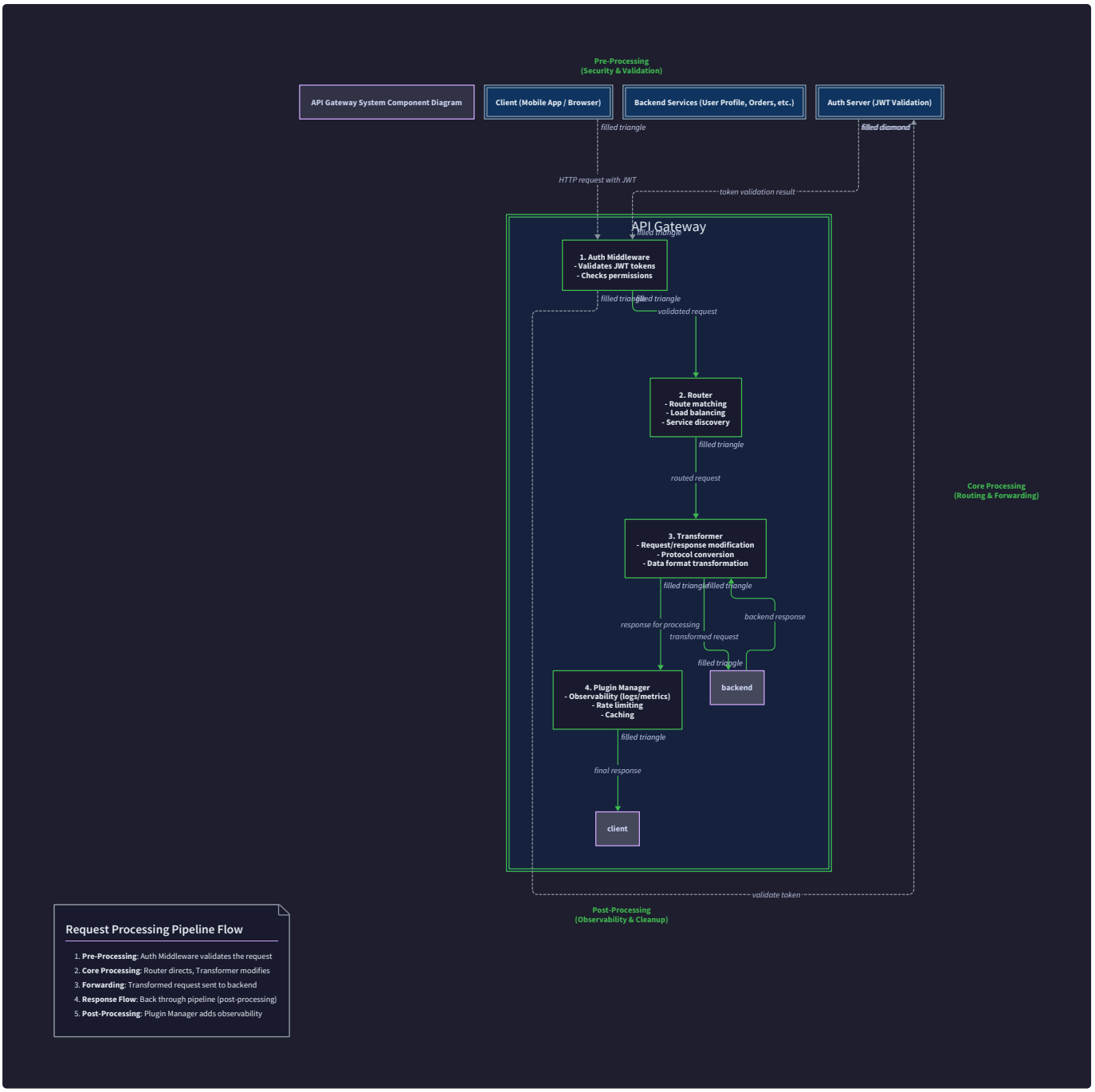
The API Gateway's architecture follows a **pipeline pattern** where incoming HTTP requests flow through a series of processing stages before being forwarded to backend services, with the response traveling back through the same chain. This design creates a clean separation of concerns where each component specializes in one aspect of request handling, making the system modular, testable, and extensible.

Component Overview and Data Flow

Think of the API Gateway as a **multi-stage assembly line** in a manufacturing plant. Each station along the conveyor belt performs a specific, specialized operation on the product (the HTTP request/response). The raw materials (incoming HTTP request) enter at the first station, get inspected, modified, and packaged at successive stations, and finally get shipped out (to the backend service). The return package (response) then travels back through some of the same stations for final quality checks and labeling before being delivered to the customer (client). This assembly line model ensures consistent processing while allowing individual stations to be upgraded or replaced without disrupting the entire production process.

The gateway decomposes into five logical layers that process requests in sequence:

1. **Ingress Layer** - The factory's receiving dock
2. **Pre-Processing Layer** - Quality inspection and initial preparation
3. **Core Processing Layer** - The main assembly and transformation stations
4. **Post-Processing Layer** - Final packaging and labeling
5. **Egress Layer** - The shipping department



Layer 1: Ingress Layer - The Factory Receiving Dock

The ingress layer is the gateway's public-facing interface, responsible for accepting incoming HTTP connections from clients. It operates like a factory's receiving dock where trucks (client connections) arrive with raw materials (HTTP requests). This layer's responsibilities are:

- **Connection Management**: Accepts TCP connections on the configured `ListenAddr` (e.g., `:8080`) and manages the HTTP/1.1 or HTTP/2 protocol handshake
- **Request Buffering**: Temporarily buffers incoming request bodies to handle slow clients without blocking processing threads
- **TLS Termination**: If configured, terminates TLS/SSL connections, decrypting traffic before further processing
- **Connection Limits**: Enforces maximum concurrent connections and request timeouts to prevent resource exhaustion
- **IP Whitelisting**: Optionally filters connections based on source IP ranges

The ingress layer produces a standardized `http.Request` object (Go's native HTTP representation) that flows downstream. It does **not** perform any business logic or routing decisions—its sole purpose is to reliably accept and normalize incoming traffic.

Layer 2: Pre-Processing Layer - Quality Inspection Station

Once inside the factory, each request passes through a quality inspection station where initial checks and metadata enrichment occur. This layer handles cross-cutting concerns that should execute before any business logic:

- **Request Context Creation**: Initializes a `RequestContext` object (described in the Data Model section) that travels with the request through the entire pipeline, accumulating metadata, authentication results, transformation state, and logging information
- **Distributed Tracing**: Injects or extracts tracing headers (W3C Trace-Context, B3, etc.) to enable end-to-end request tracking across services
- **Request ID Generation**: Creates a unique correlation ID for the request if not provided by the client, ensuring all logs related to this request can be linked
- **Structured Logging Setup**: Attaches logging fields (request ID, client IP, timestamp) to the context for consistent logging throughout the pipeline
- **Global Rate Limiting**: Applies coarse-grained rate limits based on client IP before any expensive processing occurs

The pre-processing layer's output is an enriched `RequestContext` attached to the standard `http.Request`, ready for core processing. If this layer rejects a request (e.g., due to global rate limiting), it returns an immediate HTTP error response without proceeding further.

Layer 3: Core Processing Layer - The Main Assembly Line

This is the heart of the API Gateway where the actual business logic executes. The core layer contains several specialized components that process requests in a defined sequence, similar to different workstations along an assembly line. The components execute in this order:

1. **Authentication Component** - Validates the client's identity using JWT, API keys, or OAuth2 tokens
2. **Authorization Component** - Checks if the authenticated client has permission to access the requested resource
3. **Rate Limiting Component** - Enforces per-client or per-API request quotas
4. **Routing Component** - Matches the request to the appropriate backend service using path, host, or header rules
5. **Transformation Component** - Modifies the request (headers, body, URL) before forwarding to the backend
6. **Circuit Breaker & Load Balancer** - Selects a healthy backend instance and manages failure resilience

Each component in the core layer follows the **middleware pattern** - it receives the request and context, performs its logic, optionally modifies them, and passes control to the next component. The components are **pluggable** and **configurable per route**, meaning different API endpoints can have different authentication requirements, rate limits, or transformations.

The core layer's most critical output is determining **where** to send the request (which backend service endpoint) and **what** to send (the potentially transformed request). This decision is captured in the `RequestContext` and used by the egress layer.

Layer 4: Post-Processing Layer - Final Packaging

After the backend service processes the request and returns a response, that response flows back through the gateway in reverse order, starting with the post-processing layer. This layer handles response-oriented cross-cutting concerns:

- **Response Transformation**: Modifies response headers, status codes, or body content before sending to the client
- **Error Normalization**: Converts backend-specific error formats into standardized error responses
- **CORS Headers**: Adds Cross-Origin Resource Sharing headers if configured
- **Cache-Control Headers**: Applies caching directives based on route configuration
- **Response Compression**: Optionally compresses response bodies using gzip or brotli

The post-processing layer also executes any **response-phase plugins** that need to inspect or modify the outgoing response. Importantly, this layer runs **after** the backend has responded, so it cannot affect which backend receives the request or what request is sent.

Layer 5: Egress Layer - The Shipping Department

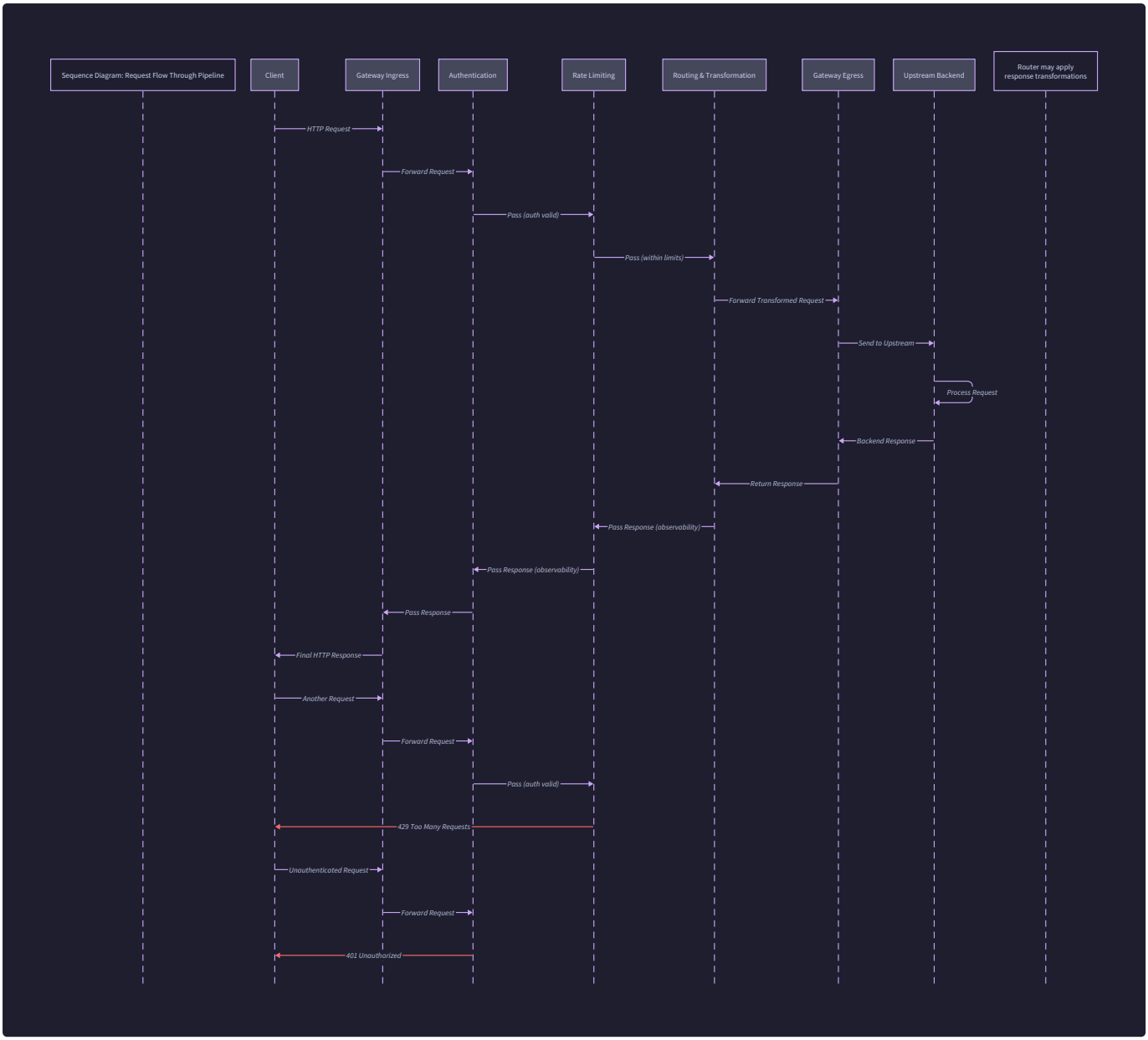
The egress layer manages the actual HTTP communication with backend services and the final delivery of responses to clients. It consists of two main subcomponents:

- **Reverse Proxy**: Forwards the processed request to the selected backend endpoint using HTTP client libraries, handling connection pooling, timeouts, and retries
- **Response Writer**: Streams the response (potentially modified by post-processing) back to the original client, managing HTTP chunking, flushing, and connection termination

The egress layer is responsible for **cleanup** - it ensures all resources (database connections, file handles, buffer memory) are properly released after request completion, regardless of success or failure. It also updates metrics and completes tracing spans to provide observability data.

Complete Request Lifecycle

The following sequence diagram illustrates the complete flow of a single HTTP request through all five layers:



The exact processing sequence follows this algorithm:

1. **Client Connection**: A client establishes a TCP connection to the gateway's `ListenAddr`
2. **HTTP Request Reception**: The ingress layer reads and parses the HTTP request, creating a standard `http.Request` object
3. **Pre-Processing Initialization**:
    1. Create `RequestContext` with unique request ID, start timestamp, and client IP
    2. Extract tracing headers and create root span
    3. Apply global rate limiting (reject if exceeded)
    4. Attach context to request
4. **Core Processing Chain** (executed in order):
    1. **Authentication**: Extract credentials (JWT from `Authorization` header, API key from `X-API-Key`, etc.)
        - Validate signature, expiration, issuer (for JWT)
        - Verify API key exists and is not expired
        - Cache validation results for performance
        - If invalid, return `401 Unauthorized` and stop processing
    2. **Authorization**: Check if authenticated principal has required permissions for the requested path/method
        - Evaluate role-based or scope-based rules
        - If denied, return `403 Forbidden` and stop processing
    3. **Per-Client Rate Limiting**: Check request count against quota for this client/API key
        - Use sliding window or token bucket algorithm

- If exceeded, return `429 Too Many Requests` and stop processing

4. **Routing**: Match request against configured `Route` definitions
   - First match `Host` header (if specified in `RouteMatch`)
   - Then match HTTP method (if specified)
   - Then match path using radix tree (fast prefix matching)
   - Select highest-priority matching route
   - If no match, return `404 Not Found` and stop processing
5. **Request Transformation**:
   - Apply header manipulations (add, remove, modify)
   - Rewrite URL path and query parameters
   - Transform request body (JSON to JSON, XML to JSON, etc.)
   - Aggregate multiple backend requests if configured
6. **Load Balancing & Circuit Breaking**:
   - Select healthy `Endpoint` from the route's `Upstream` using round-robin or weighted algorithm
   - Check circuit breaker state for the endpoint (Closed, Open, Half-Open)
   - If circuit breaker is Open, try alternative endpoint or return `503 Service Unavailable`
5. **Backend Communication**:
   1. Open HTTP connection to selected backend endpoint (reuse from pool if possible)
   2. Send transformed request with appropriate timeouts
   3. Stream response back, handling chunked encoding
6. **Post-Processing**:
   1. Apply response transformations (headers, body)
   2. Normalize errors to standard format
   3. Add CORS headers if needed
7. **Client Response**:
   1. Write HTTP status line and headers to client
   2. Stream response body to client
   3. Close connection (or keep-alive for reuse)
8. **Cleanup**:
   1. Record metrics (latency, status code, bytes transferred)
   2. Complete tracing spans
   3. Log structured access log entry
   4. Release all allocated resources (buffers, contexts)

This pipeline design creates a **clean separation of concerns** where each component has a single responsibility. The middleware architecture allows components to be added, removed, or reordered through configuration without code changes. The `RequestContext` object serves as a "work order" that travels with the request, accumulating state and decisions at each stage.

Recommended File/Module Structure

A well-organized codebase is critical for maintaining a complex system like an API Gateway. The recommended structure follows Go community conventions while grouping related functionality together. The layout emphasizes **encapsulation** (internal implementation details are hidden), **testability** (each component can be tested in isolation), and **maintainability** (related code lives together).

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go                # Application entry point
├── internal/                      # Private application code (not importable)
│   ├── config/                    # Configuration loading and validation
│   │   ├── config.go              # Config struct definitions
│   │   ├── loader.go              # LoadFromFile, validate functions
│   │   ├── parser_yaml.go         # YAML parsing implementation
│   │   └── config_test.go         # Configuration tests
│   ├── middleware/                # Request processing middleware
│   │   ├── chain.go               # Middleware chaining logic
│   │   ├── context.go             # RequestContext struct and methods
│   │   ├── auth/                  # Authentication middleware
│   │   │   ├── authenticator.go   # Authenticator interface
│   │   │   ├── jwt.go             # JWT validation implementation
│   │   │   ├── apikey.go          # API key validation
│   │   │   ├── oauth2.go          # OAuth2 token introspection
│   │   │   └── cache.go           # Auth token caching
│   │   ├── rate_limit/            # Rate limiting middleware
│   │   │   ├── limiter.go         # RateLimiter interface
│   │   │   ├── sliding_window.go  # Sliding window implementation
│   │   │   └── redis_store.go     # Distributed rate limiting store
│   │   ├── transform/             # Request/response transformation
│   │   │   ├── transformer.go     # Transformer interface
│   │   │   ├── headers.go         # Header manipulation
│   │   │   ├── body_json.go       # JSON body transformation
│   │   │   ├── url_rewrite.go     # URL rewriting
│   │   │   └── aggregate.go       # Request aggregation
│   │   └── observability/         # Logging, metrics, tracing
│   │       ├── logger.go          # Structured logging middleware
│   │       ├── metrics.go         # Prometheus metrics collection
│   │       ├── tracing.go         # OpenTelemetry tracing
│   │       └── recovery.go        # Panic recovery middleware
│   ├── router/                    # Routing and load balancing
│   │   ├── router.go              # Router interface and main implementation
│   │   ├── matcher.go             # Route matching logic (radix tree)
│   │   ├── loadbalancer.go        # LoadBalancer interface
│   │   ├── round_robin.go         # Round-robin implementation
│   │   ├── health_check.go        # Active and passive health checks
│   │   ├── circuit_breaker.go     # Circuit breaker implementation
│   │   └── router_test.go         # Routing tests
│   ├── proxy/                     # Reverse proxy implementation
│   │   ├── reverse_proxy.go       # Custom httputil.ReverseProxy extension
│   │   ├── transport.go           # HTTP transport with connection pooling
│   │   ├── buffer_pool.go         # Buffer pool for request/response bodies
│   │   └── timeout.go             # Timeout handling logic
│   ├── plugin/                    # Plugin system
│   │   ├── plugin.go              # Plugin interface and registry
│   │   ├── manager.go             # Plugin lifecycle management
│   │   ├── builtin/               # Built-in plugins
│   │   │   ├── cors.go            # CORS plugin
│   │   │   ├── cache.go           # Response caching plugin
│   │   │   └── compression.go     # Compression plugin
│   │   └── examples/              # Example plugins for developers
│   │       └── custom_header.go   # Example: add custom headers
│   └── server/                    # HTTP server and request pipeline
│       ├── server.go              # Main server struct and lifecycle
│       ├── pipeline.go            # Request pipeline construction
│       ├── handler.go             # HTTP handler that executes pipeline
│       └── shutdown.go            # Graceful shutdown logic
├── pkg/                           # Public libraries (importable by others)
│   ├── api/                       # Public API types
│   │   ├── types.go               // Public type definitions
│   │   └── errors.go              // Standard error types
│   └── plugin/                    // Public plugin SDK
│       ├── sdk.go                 // Plugin development helpers
│       └── types.go               // Plugin interface definitions
├── configs/                       # Configuration files
│   ├── gateway.yaml               # Example configuration
│   └── routes/                    # Route-specific configurations
│       └── users-service.yaml     # Example route definition
├── deployments/                   # Deployment artifacts
│   ├── docker/                    # Docker configuration
│   │   └── Dockerfile             # Multi-stage Dockerfile
│   ├── kubernetes/                # K8s manifests
│   │   ├── deployment.yaml        # Gateway deployment
│   │   └── service.yaml           # K8s service definition
│   └── docker-compose.yaml        # Local development setup
├── scripts/                       # Build and development scripts
```

```
│   ├── build.sh                # Build script
│   ├── test.sh                 # Test runner
│   └── lint.sh                 # Linting script
├── go.mod                      # Go module definition
├── go.sum                      # Go dependencies checksum
├── Makefile                    # Common tasks automation
├── .golangci.yml               # Linter configuration
└── README.md                   # Project documentation
```

Key Organizational Principles

1. `cmd/gateway/main.go` **- Single Entry Point**: All execution starts here. This file should be minimal—just parsing command-line flags, loading configuration, and starting the server. Business logic belongs in the `internal` directory.
2. `internal/` **- Application-Private Code**: The `internal` directory contains all gateway implementation code that should not be imported by external applications. This follows Go's convention for preventing external dependencies on internal APIs that may change.
3. **Component-Based Organization**: Each major component (config, middleware, router, proxy, plugin) gets its own subdirectory. Within each, interfaces are defined in the root file (`router.go` , `authenticator.go`), with implementations in either the same directory (for simple components) or subdirectories (for complex components with multiple implementations).
4. `pkg/` **- Public APIs**: Only types and interfaces that need to be exposed to plugin developers or integrators go here. The plugin SDK is a key example—plugin authors need to import these types to write compliant plugins.
5. **Flat Structure Within Components**: Avoid deep nesting. Most components should be only one level deep within their parent directory. The exception is middleware, which has natural subcategories (auth, rate_limit, etc.) that group related functionality.
6. **Test Files Co-located**: Each `.go` file should have a corresponding `_test.go` file in the same directory. This makes tests easy to find and maintains the relationship between implementation and tests.
7. **Configuration Separate from Code**: YAML configuration files live in `configs/` , completely separate from Go code. This allows operators to modify gateway behavior without touching source code.
8. **Deployment Artifacts Together**: All Docker, Kubernetes, and deployment configuration lives in `deployments/` , making it easy for DevOps teams to find what they need without digging through source code.

This structure scales well as the gateway grows. New middleware types can be added to `internal/middleware/` , new plugins to `internal/plugin/builtin/` , and new load balancing algorithms to `internal/router/` . The clear separation makes it easy for multiple developers to work on different components simultaneously without conflicts.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
HTTP Server	`net/http` with `http.Server`	`net/http` with tuned `http.Server` parameters and connection pooling
Configuration	YAML with `gopkg.in/yaml.v3`	YAML with validation using `go-playground/validator/v10`
Routing	Radix tree with `github.com/gorilla/mux` or custom implementation	Custom radix tree with path parameter extraction and priority-based matching
Reverse Proxy	`net/http/httputil.ReverseProxy` with customization	Custom proxy with connection pooling, retry logic, and circuit breaking
JSON Processing	`encoding/json` with streaming decoder	`encoding/json` with `json.RawMessage` for selective parsing
JWT Validation	`github.com/golang-jwt/jwt/v5`	`github.com/golang-jwt/jwt/v5` with key rotation and JWKS support
Rate Limiting	In-memory sliding window	Redis-backed distributed rate limiting with `github.com/go-redis/redis`
Metrics	`github.com/prometheus/client_golang`	`github.com/prometheus/client_golang` with custom collectors and histograms
Tracing	OpenTelemetry Go SDK (`go.opentelemetry.io/otel`)	OpenTelemetry with Jaeger or Zipkin exporter and sampling
Plugin System	Compiled-in plugin registry	Dynamic loading with `plugin` package or WebAssembly (WASM)
Caching	`sync.Map` or LRU cache (`github.com/hashicorp/golang-lru`)	Redis for distributed cache with `github.com/go-redis/redis`

B. Recommended File/Module Structure

The file structure shown above provides a comprehensive blueprint. For initial implementation, focus on these core directories first:

1. **Start with** `cmd/gateway/main.go` - Create the minimal entry point
2. **Build** `internal/config/` - Implement configuration loading
3. **Create** `internal/server/` - Set up the HTTP server and request pipeline
4. **Implement** `internal/router/` - Build routing core (Milestone 1)
5. **Add** `internal/middleware/` - Implement middleware components one by one
6. **Complete** `internal/proxy/` - Build the reverse proxy

Each milestone corresponds to implementing specific subdirectories:

- **Milestone 1**: `internal/router/`, `internal/proxy/`, and basic `internal/server/`
- **Milestone 2**: `internal/middleware/transform/` and extend `internal/proxy/`
- **Milestone 3**: `internal/middleware/auth/` and `internal/middleware/rate_limit/`
- **Milestone 4**: `internal/middleware/observability/` and `internal/plugin/`

C. Infrastructure Starter Code

Here's complete, working code for foundational components that are prerequisites for the core gateway logic:

1. Configuration Loader (`internal/config/loader.go`):

```go
package config

import (
    "fmt"
    "io/ioutil"

    "gopkg.in/yaml.v3"
)

// LoadFromFile reads and parses a YAML configuration file
func LoadFromFile(filepath string) (*Config, error) {
    data, err := ioutil.ReadFile(filepath)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

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

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

    return &cfg, nil
}

// validate performs basic sanity checks on the configuration
func validate(cfg *Config) error {
    if cfg.ListenAddr == "" {
        return fmt.Errorf("ListenAddr must be specified")
    }

    // Check for duplicate route IDs
    routeIDs := make(map[string]bool)
    for _, route := range cfg.Routes {
        if route.ID == "" {
            return fmt.Errorf("route missing ID field")
        }
        if routeIDs[route.ID] {
            return fmt.Errorf("duplicate route ID: %s", route.ID)
```

```
        }
        routeIDs[route.ID] = true

        // Route must have either Path or Headers in its Match
        if route.Match.Path == "" && len(route.Match.Headers) == 0 {
            return fmt.Errorf("route %s: must specify either Path or Headers to match", route.ID)
        }

        // Route must point to a valid upstream
        found := false
        for _, upstream := range cfg.Upstreams {
            if upstream.ID == route.UpstreamID {
                found = true
                break
            }
        }
        if !found {
            return fmt.Errorf("route %s: upstream %s not found", route.ID, route.UpstreamID)
        }
    }

    // Check for duplicate upstream IDs
    upstreamIDs := make(map[string]bool)
    for _, upstream := range cfg.Upstreams {
        if upstream.ID == "" {
            return fmt.Errorf("upstream missing ID field")
        }
        if upstreamIDs[upstream.ID] {
            return fmt.Errorf("duplicate upstream ID: %s", upstream.ID)
        }
        upstreamIDs[upstream.ID] = true

        // Upstream must have at least one endpoint
        if len(upstream.Endpoints) == 0 {
            return fmt.Errorf("upstream %s: must have at least one endpoint", upstream.ID)
        }

        // Check for duplicate endpoint URLs
        endpointURLs := make(map[string]bool)
        for _, endpoint := range upstream.Endpoints {
```

```
            if endpoint.URL == "" {
                return fmt.Errorf("upstream %s: endpoint missing URL", upstream.ID)
            }
            if endpointURLs[endpoint.URL] {
                return fmt.Errorf("upstream %s: duplicate endpoint URL: %s", upstream.ID, endpoint.URL)
            }
            endpointURLs[endpoint.URL] = true
        }
    }

    return nil
}
```

2. Request Context (`internal/middleware/context.go`):

```go
package middleware

import (
    "context"
    "net/http"
    "time"
)

// RequestContext carries state through the middleware pipeline
type RequestContext struct {
    // Request metadata
    RequestID      string
    StartTime      time.Time
    ClientIP       string

    // Authentication results
    Authenticated  bool
    UserID         string
    UserRoles      []string
    AuthMethod     string // "jwt", "apikey", "oauth2"

    // Routing decisions
    MatchedRoute   *Route
    SelectedUpstream *Upstream
    SelectedEndpoint *Endpoint

    // Transformation state
    RequestBody    []byte // Buffered request body for transformation
    ResponseBody   []byte // Buffered response body for transformation

    // Logging and observability
    LogFields      map[string]interface{}
    CustomMetrics  map[string]float64

    // Error handling
    Error          error
    HTTPErrorCode  int

    // Internal context for cancellation and timeouts
    innerCtx       context.Context
    cancelFn       context.CancelFunc
```

```go
}

// NewRequestContext creates a new context for a request
func NewRequestContext(r *http.Request) *RequestContext {
    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)

    return &RequestContext{
        RequestID:     generateRequestID(),
        StartTime:     time.Now(),
        ClientIP:      extractClientIP(r),
        Authenticated: false,
        LogFields:     make(map[string]interface{}),
        CustomMetrics: make(map[string]float64),
        innerCtx:      ctx,
        cancelFn:      cancel,
    }
}

// Context returns the Go context for cancellation and timeouts
func (rc *RequestContext) Context() context.Context {
    return rc.innerCtx
}

// Cancel cancels the request context (used for cleanup)
func (rc *RequestContext) Cancel() {
    if rc.cancelFn != nil {
        rc.cancelFn()
    }
}

// AddLogField adds a field to the structured log
func (rc *RequestContext) AddLogField(key string, value interface{}) {
    rc.LogFields[key] = value
}

// ElapsedTime returns time since request started
func (rc *RequestContext) ElapsedTime() time.Duration {
    return time.Since(rc.StartTime)
}

// Helper functions (implement in same file)
func generateRequestID() string {
    // Implement UUID or unique ID generation
```

```
    return "req-" + time.Now().Format("20060102-150405.000")
}

func extractClientIP(r *http.Request) string {
    // Check X-Forwarded-For, then X-Real-IP, then RemoteAddr
    if forwarded := r.Header.Get("X-Forwarded-For"); forwarded != "" {
        return forwarded
    }
    if realIP := r.Header.Get("X-Real-IP"); realIP != "" {
        return realIP
    }
    return r.RemoteAddr
}
```

D. Core Logic Skeleton Code

1. Main Server Pipeline (`internal/server/pipeline.go`):

```go
package server

import (
    "net/http"

    "api-gateway/internal/middleware"
)

// Pipeline represents the complete request processing chain
type Pipeline struct {
    preProcess  []middleware.Middleware
    coreProcess []middleware.Middleware
    postProcess []middleware.Middleware
}

// NewPipeline creates a pipeline from configuration
func NewPipeline(cfg *config.Config) (*Pipeline, error) {
    p := &Pipeline{}

    // TODO 1: Initialize pre-processing middleware
    // - Create tracing middleware if configured
    // - Create request ID middleware
    // - Create global rate limiter if configured

    // TODO 2: Initialize core processing middleware in correct order
    // - Create authentication middleware (JWT, API key, OAuth2)
    // - Create authorization middleware (role/scope checking)
    // - Create per-route rate limiting middleware
    // - Create routing middleware (path/host matching)
    // - Create transformation middleware (headers, body, URL)
    // - Create load balancing and circuit breaker middleware

    // TODO 3: Initialize post-processing middleware
    // - Create response transformation middleware
    // - Create CORS middleware if configured
    // - Create error normalization middleware

    // TODO 4: Load and initialize plugins from configuration
    // - For each PluginConfig in routes, instantiate the corresponding plugin
    // - Add plugins to appropriate pipeline stage (pre, core, or post)

    return p, nil
```

```
}

// Execute runs the request through the complete pipeline
func (p *Pipeline) Execute(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Create RequestContext from the HTTP request
    // ctx := middleware.NewRequestContext(r)

    // TODO 2: Execute pre-processing middleware chain
    // If any middleware returns an error or writes response, stop and return

    // TODO 3: Execute core processing middleware chain
    // This includes routing, auth, transformation, etc.

    // TODO 4: If core processing selected a backend, forward the request
    // Use reverse proxy to send to selected endpoint

    // TODO 5: Execute post-processing middleware on the response
    // Modify response before sending to client

    // TODO 6: Write final response to client
    // Include proper headers, status code, and body

    // TODO 7: Clean up resources (context cancellation, buffer release)
}
```

2. Middleware Interface (`internal/middleware/chain.go`):

```go
package middleware

import (
    "net/http"
)

// Middleware defines the interface for all request processors
type Middleware interface {
    // Name returns the middleware's unique identifier
    Name() string

    // Execute processes the request, returning true to continue or false to stop
    Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool

    // Priority determines execution order (lower numbers execute first)
    Priority() int
}

// Chain represents a sequence of middleware
type Chain struct {
    middleware []Middleware
}

// NewChain creates a new middleware chain
func NewChain(middleware ...Middleware) *Chain {
    // TODO 1: Sort middleware by Priority (ascending)
    // TODO 2: Validate no duplicate middleware names
    // TODO 3: Return Chain with sorted middleware
    return nil
}

// Execute runs the chain, stopping if any middleware returns false
func (c *Chain) Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Iterate through sorted middleware
    // TODO 2: Call Execute on each middleware
    // TODO 3: If middleware returns false, stop and return false
    // TODO 4: If all middleware return true, return true
    return true
}
```

E. Language-Specific Hints

1. **Use `context.Context` for cancellation**: Always pass a context through the pipeline and respect cancellation. Use `context.WithTimeout` to set overall request deadlines.
2. **Buffer pools for performance**: Use `sync.Pool` for byte buffers when reading request/response bodies to reduce garbage collection pressure.

3. **Proper HTTP header manipulation**: When modifying request headers for proxying, copy the header map: `newHeader := make(http.Header); for k, v := range r.Header { newHeader[k] = v }`.
4. **Connection pooling**: Configure `http.Transport` with `MaxIdleConnsPerHost` and `IdleConnTimeout` for efficient backend communication.
5. **Graceful shutdown**: Implement signal handling for `SIGTERM` and `SIGINT` to drain connections before shutdown.
6. **Structured logging**: Use `log/slog` (Go 1.21+) or `zap` / `logrus` for structured, leveled logging with request context.
7. **Testing with `httptest`**: Use `net/http/httptest` for testing HTTP handlers without starting a real server.
8. **Avoid global state**: Use dependency injection—pass dependencies (config, metrics, caches) to components rather than using global variables.

F. Milestone Checkpoint

After implementing the high-level architecture (before any specific milestone features), verify the foundation works:

Command to test:

```bash
# Build the gateway
go build -o gateway ./cmd/gateway

# Create a minimal config file
cat > test-config.yaml << EOF
listenAddr: ":8080"
routes:
  - id: "test-route"
    description: "Test route to echo service"
    match:
      path: "/echo/*"
    upstreamID: "echo-service"
upstreams:
  - id: "echo-service"
    name: "Echo Service"
    endpoints:
      - id: "echo-1"
        url: "http://httpbin.org"
EOF

# Start the gateway
./gateway --config test-config.yaml
```

Expected behavior:

1. Server starts on port 8080 without errors
2. You can send a request: `curl http://localhost:8080/echo/anything`
3. Request should be forwarded to httpbin.org and return a response
4. Check logs show request processing (request ID, client IP, etc.)

Signs something is wrong:

- "Address already in use": Change `listenAddr` to another port
- "config validation failed": Check YAML syntax and required fields
- No response or connection refused: Verify server started successfully
- 404 Not Found: Check that request path matches configured route pattern

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Gateway crashes on startup	Invalid configuration or missing dependencies	Check startup logs for validation errors; run `go vet ./...` for code issues	Fix config YAML; add missing imports
Requests hang indefinitely	Missing timeout configuration or deadlock	Add debug logging at pipeline stages; use `pprof` to see goroutine stack	Set proper timeouts in `context.WithTimeout`; check for mutex deadlocks
Memory usage grows continuously	Memory leaks in buffers or contexts	Use `pprof` heap profiling; check for missing `defer rc.Cancel()`	Implement buffer pools with `sync.Pool`; ensure context cleanup
"Too many open files" error	Connection leaks to backends or clients	Check `netstat` for ESTABLISHED connections; monitor file descriptor count	Implement connection pooling with `http.Transport`; add `defer resp.Body.Close()`
Route matching seems random	Incorrect middleware ordering or priority	Add debug logs showing route matching decisions; check middleware `Priority()` values	Ensure middleware sorted by priority; verify radix tree construction

Data Model

> **Milestone(s):** This foundational data model underpins functionality across all milestones, providing the configuration and runtime structures that enable routing (Milestone 1), transformation (Milestone 2), authentication (Milestone 3), and plugins (Milestone 4).

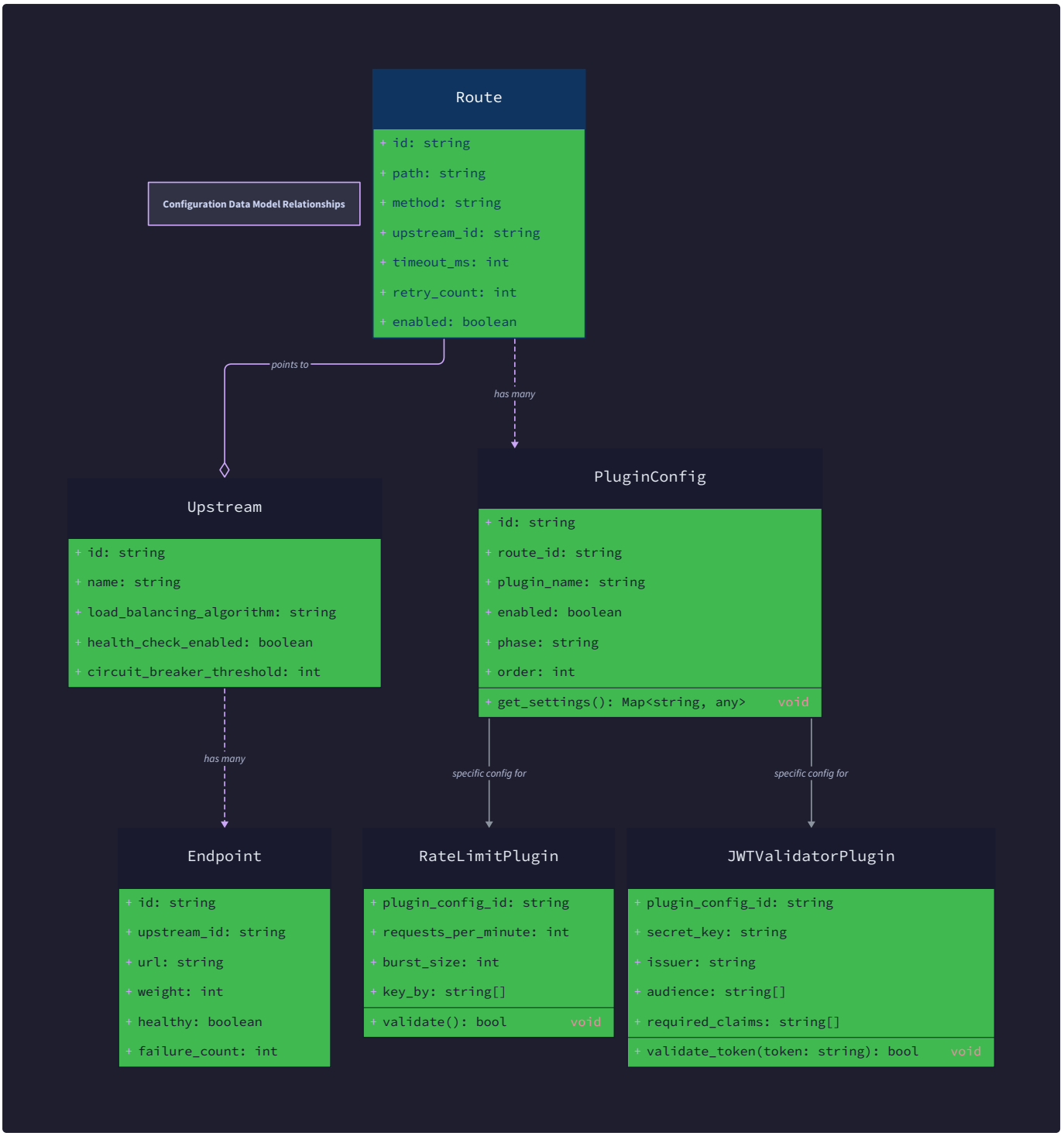
At the heart of the API Gateway lies a structured data model that defines its behavior and captures the state of each request. This model serves two critical purposes: **static configuration** and **runtime context**. Think of it as the gateway's **DNA and short-term memory**.

- **Configuration Structures** are the DNA: they are the static blueprints loaded at startup (or dynamically) that define *what the gateway can do*. These structures—routes, upstreams, plugins—are the source code for the gateway's behavior, determining how it routes, transforms, and secures traffic.
- **Request Context and Shared State** is the short-term memory: it's the live, in-memory journal for a single request's journey. As a request flows through the processing pipeline, this context object accumulates details—who the user is, which route was matched, what transformations were applied, and what errors occurred. It enables each middleware component to communicate with downstream components without side channels, ensuring a clean, auditable data flow.

Without a well-designed data model, the gateway would be a collection of disjointed functions. With it, we have a coherent system where configuration drives behavior and runtime state is explicitly managed.

Configuration Structures

The configuration structures define the **what**, **where**, and **how** of the gateway's operation. They are typically loaded from a YAML or JSON file at startup, validated, and then used to build the processing pipeline. The relationships between these structures are visualized in the diagram below.



The diagram shows the core relationships: a `Route` selects traffic and points to an `Upstream` (a logical service). An `Upstream` contains one or more `Endpoint`s (physical instances). A `Route` can have zero or more `PluginConfig`s that attach middleware behavior like authentication or rate limiting.

Each configuration structure is detailed in the tables below.

The `Config` Root Structure

The `Config` struct is the root container for the entire gateway configuration. It holds global settings and collections of all other configurable entities.

Field Name	Type	Description
`ListenAddr`	`string`	The network address and port on which the gateway's HTTP server will listen (e.g., `":8080"`). This is the entry point for all client traffic.
`Routes`	`[]Route`	An ordered list of route definitions. The order can matter for matching priority if multiple routes could match a request (first match wins by default).
`Upstreams`	`[]Upstream`	A list of upstream service definitions. Each upstream represents a logical backend service (like "user-service") and contains its physical endpoints.

A typical configuration file in YAML might look like this:

```yaml
listenAddr: ":8080"

upstreams:

  - id: "users-service"

    name: "User Management Service"

    endpoints:

      - id: "users-1"

        url: "http://localhost:3001"

      - id: "users-2"

        url: "http://localhost:3002"

routes:

  - id: "users-api"

    description: "Route for user management API"

    match:

      path: "/api/users/*"

      method: "GET"

    upstreamId: "users-service"

    plugins:

      - name: "jwt-validator"

        config:

          jwks_url: "https://auth.example.com/.well-known/jwks.json"
```

The `Route` and `RouteMatch` Structures

A `Route` is a rule that maps incoming requests to an upstream service. It consists of a **match condition** (what requests it applies to) and an **action** (where to send those requests and what plugins to apply).

`Route` Structure:

Field Name	Type	Description
`ID`	`string`	A unique identifier for the route (e.g., `"users-api"`). Used for logging and dynamic configuration updates.
`Description`	`string`	A human-readable description of the route's purpose (e.g., "Route for user management API").
`Match`	`RouteMatch`	The condition that an incoming HTTP request must satisfy for this route to be selected.
`UpstreamID`	`string`	The ID of the `Upstream` to which matched requests should be forwarded. This creates the link between a route and a backend service.
`Plugins`	`[]PluginConfig`	A list of plugin configurations that are activated for requests matching this route. Plugins are executed in the order they are defined.

`RouteMatch` Structure:

Field Name	Type	Description
`Path`	`string`	A path pattern to match against the request URL path. Supports prefix matching (e.g., `/api/users/*`) and exact matching. The radix tree implementation (chosen in an ADR) efficiently matches these patterns.
`Method`	`string`	The HTTP method to match (e.g., `"GET"`, `"POST"`, `"*"` for any). Case-insensitive.
`Headers`	`map[string]string`	A set of key-value pairs that must be present in the request headers for a match. The match is typically "all headers must match exactly" (values) or "header must exist" (if value is `"*"`). For example, `Host: api.example.com` or `X-API-Version: 2024-01-01`.

> **Design Insight:** The `RouteMatch` structure intentionally uses simple string matching for headers and paths, not regular expressions, in its initial design. This balances performance and common use cases. Complex matching can be delegated to a dedicated plugin (like a `regex-route` plugin) if needed, keeping the core router fast and predictable.

The `Upstream` and `Endpoint` Structures

An `Upstream` represents a logical backend service, such as "user-service" or "payment-service." It contains one or more `Endpoint`s, which are the physical instances (host:port) of that service. The gateway load balances traffic across healthy endpoints.

`Upstream` Structure:

Field Name	Type	Description
`ID`	`string`	A unique identifier for the upstream (e.g., `"users-service"`). Referenced by routes.
`Name`	`string`	A human-readable name for the upstream (e.g., "User Management Service"). Used in logs and metrics.
`Endpoints`	`[]Endpoint`	The list of physical endpoints (servers) that comprise this upstream service. The gateway will distribute requests among these using the configured load balancing strategy.

`Endpoint` Structure:

Field Name	Type	Description
`ID`	`string`	A unique identifier for this specific endpoint instance (e.g., `"users-1"`). Used for health tracking and logging.
`URL`	`string`	The full base URL of the endpoint, including protocol, host, and port (e.g., `"http://10.0.1.5:3000"`). The gateway will forward requests to this URL.

The `PluginConfig` Structure

The `PluginConfig` structure defines an instance of a middleware plugin that should be applied to requests matching a route. Plugins are the primary extension mechanism of the gateway, allowing features like authentication, rate limiting, logging, and transformation to be attached to specific routes.

Field Name	Type	Description
`Name`	`string`	The unique name of the plugin (e.g., `"jwt-validator"`, `"rate-limiter"`). This name is used to look up the plugin implementation in the gateway's plugin registry.
`Config`	`map[string]interface{}`	A flexible key-value map of plugin-specific configuration options. The structure is defined by each plugin. For example, a JWT validator plugin might expect keys like `jwks_url` and `required_claims`. Using `interface{}` allows for JSON/YAML flexibility but requires careful validation within the plugin.

> **Design Insight:** Using `map[string]interface{}` for plugin configuration is a deliberate trade-off. It provides maximum flexibility for plugin authors and easy serialization from YAML/JSON. The downside is loss of type safety. We mitigate this by having each plugin validate its config at load time and convert it into a typed internal structure.

Architecture Decision: Flexible Plugin Configuration

> **Decision: Flexible Plugin Configuration via `map[string]interface{}`**
>
> - **Context**: We need a way for plugin authors to define arbitrary configuration parameters without modifying the core gateway configuration schema. The configuration must be serializable from human-readable formats like YAML.
> - **Options Considered**:
>   1. **Strictly typed struct per plugin**: Define a Go struct for each plugin's config and use a type switch or registration system. This provides compile-time type safety.
>   2. **`map[string]interface{}`**: A flexible key-value map that plugins can parse as needed. This is easy to serialize and allows arbitrary nesting.
>   3. **JSON raw message**: Store configuration as `json.RawMessage` and have each plugin unmarshal into its own struct. This provides type safety after unmarshaling but requires JSON-specific tooling.
> - **Decision**: We chose option 2, `map[string]interface{}`, for the initial implementation.
> - **Rationale**: The primary goal is flexibility and simplicity of the configuration file. Plugin authors can define any structure without touching core types. The configuration is easy to write and read in YAML. While it sacrifices some type safety, we can enforce validation at plugin initialization time. This approach is used by popular gateways like Kong and is familiar to operators.
> - **Consequences**:
>   - **Positive**: Extremely flexible; new plugins can be developed without changes to the core config loader. Configuration files are straightforward.
>   - **Negative**: No compile-time type checking for plugin configs; errors are caught at runtime. Plugin authors must write their own validation logic.

Option	Pros	Cons	Chosen?
Strictly typed struct per plugin	Type safety at compile time; IDE autocompletion	Requires core changes for new plugins; complex registration	No
`map[string]interface{}`	Maximum flexibility; easy serialization; no core changes	Runtime errors only; no type safety	**Yes**
JSON raw message	Type safety after unmarshal; no extra parsing	Tied to JSON; slightly more complex for plugin authors	No

Request Context and Shared State

While configuration structures define the gateway's *potential* behavior, the `RequestContext` struct captures the *actual* state of a single HTTP request as it journeys through the gateway's pipeline. This is the **shared state** that all middleware components read from and write to.

Think of the `RequestContext` as a **patient's medical chart** in a hospital. As the patient (request) moves from admission (ingress) to diagnosis (authentication) to treatment (routing/transformation) and discharge (egress), each specialist (middleware) records observations, test results, and actions taken. This chart ensures that every department has the full context needed to do their job and that a coherent record exists for the entire episode of care.

The `RequestContext` Structure

The `RequestContext` is a rich object that carries both generic HTTP request data and gateway-specific metadata. It is created when a request enters the gateway and is passed through each middleware in the pipeline. The table below details every field.

Field Name	Type	Description
`RequestID`	`string`	A unique identifier for this request, typically a UUID, generated at ingress. Used for correlating logs and traces across the system.
`StartTime`	`time.Time`	The timestamp when the request was received. Used to calculate latency and for logging.
`ClientIP`	`string`	The IP address of the originating client, extracted from the request's `RemoteAddr` or `X-Forwarded-For` header. Used for rate limiting and logging.
`Authenticated`	`bool`	A flag set by authentication middleware indicating whether the request has been successfully authenticated.
`UserID`	`string`	If authenticated, the identifier of the user (e.g., subject from JWT). Passed to backends via headers.
`UserRoles`	`[]string`	The roles associated with the authenticated user (e.g., `["admin", "editor"]`). Used for authorization decisions.
`AuthMethod`	`string`	The method used for authentication (e.g., `"jwt"`, `"api-key"`). Useful for logging and metrics.
`MatchedRoute`	`*Route`	A pointer to the `Route` configuration that matched this request. This gives middleware access to the route's plugins and upstream ID.
`SelectedUpstream`	`*Upstream`	A pointer to the `Upstream` selected for this request (derived from `MatchedRoute.UpstreamID`).
`SelectedEndpoint`	`*Endpoint`	A pointer to the specific `Endpoint` (backend instance) chosen by the load balancer for this request.
`RequestBody`	`[]byte`	A buffer holding the request body content, if it has been read for transformation. **Important**: This is only populated if a transformation middleware reads the body; otherwise, the body is streamed directly to the backend.
`ResponseBody`	`[]byte`	A buffer holding the response body content, if it has been read for transformation. Similar to `RequestBody`, this is conditionally populated.
`LogFields`	`map[string]interface{}`	A structured map for logging. Middleware can add key-value pairs here (e.g., `"rate_limit_remaining": 5`) that will be included in the final access log.
`CustomMetrics`	`map[string]float64`	A place for middleware to record custom numeric metrics for this request (e.g., `"auth_duration_ms": 12.5`). These can be aggregated and exported.
`Error`	`error`	If an error occurs during processing (e.g., authentication failed, backend unreachable), it is stored here. The pipeline can inspect this to decide whether to continue or terminate early.
`HTTPErrorCode`	`int`	The HTTP status code to return if an error occurs (e.g., `401` for unauthorized). This allows middleware to set the error code without immediately writing the response.
`innerCtx`	`context.Context`	An embedded Go `context.Context` used for cancellation and timeouts. This allows the gateway to respect deadlines and propagate cancellation to backend requests.
`cancelFn`	`context.CancelFunc`	The cancellation function associated with `innerCtx`. Called to cancel the request context, for example, if the client disconnects.

Lifecycle and Usage Patterns

The `RequestContext` is created by the ingress layer using the `NewRequestContext` function. It is then passed sequentially through each middleware in the pipeline via the `Execute` method. Middleware can inspect and modify the context. The typical lifecycle follows these steps:

1. **Creation and Initialization**: `NewRequestContext` extracts the `ClientIP`, generates a `RequestID`, and sets `StartTime`. It also creates a cancellable Go context (`innerCtx`) to manage request-scoped deadlines.
2. **Pre-Processing Phase**: Middleware like authentication and rate limiting run. They may set `Authenticated`, `UserID`, `UserRoles`, or add `LogFields`. If an error occurs, they set `Error` and `HTTPErrorCode` and may cause the pipeline to stop.
3. **Routing Phase**: The router middleware matches the request to a `Route`, sets `MatchedRoute`, looks up the corresponding `Upstream`, and uses the load balancer to select an `Endpoint`, setting `SelectedUpstream` and `SelectedEndpoint`.
4. **Transformation Phase**: If configured, transformation middleware may read the request body into `RequestBody`, modify it, and update the `*http.Request` before forwarding. They may also buffer the response into `ResponseBody` for modification.
5. **Post-Processing Phase**: Middleware like logging and metrics run. They read `LogFields`, `CustomMetrics`, `ElapsedTime()`, and other fields to produce structured logs and update metrics.
6. **Cleanup**: The context's `cancelFn` is called when the request is complete, releasing any resources associated with `innerCtx`.

Key Methods on `RequestContext`

The `RequestContext` struct has several helper methods that encapsulate common operations. These methods ensure consistent behavior and provide a clean API for middleware.

Method Signature	Returns	Description
`NewRequestContext(r *http.Request) *RequestContext`	`*RequestContext`	Factory function that creates a new context for an incoming request. Initializes `RequestID`, `StartTime`, `ClientIP`, and the internal cancellable context.
`Context() context.Context`	`context.Context`	Returns the underlying Go context (`innerCtx`). Used when making downstream HTTP requests to propagate cancellation.
`Cancel()`	`none`	Cancels the request's context, signaling that processing should stop (e.g., client disconnected). Called by the framework after the response is written.
`AddLogField(key string, value interface{})`	`none`	Adds a key-value pair to the `LogFields` map. If the map is nil, it is initialized. This ensures structured logging across all middleware.
`ElapsedTime() time.Duration`	`time.Duration`	Calculates and returns the time elapsed since `StartTime`. Used for latency metrics and logging.

> **Design Insight:** The `RequestContext` is designed to be **opaque** in terms of its internal Go context (`innerCtx`). Middleware should use the `Context()` method to retrieve it, rather than accessing the field directly. This encapsulation allows for future changes to the context implementation without breaking plugin code.

Architecture Decision: Monolithic Context vs. Scoped Contexts

> **Decision: Single Monolithic Request Context**
>
> - **Context**: We need a way for middleware components to share data and communicate during request processing. The data ranges from authentication results to transformation buffers to custom log fields.
> - **Options Considered**:
>   1. **Single monolithic context**: One struct (`RequestContext`) that holds all possible shared fields, passed to every middleware.
>   2. **Scoped contexts**: Separate context structs for different phases (e.g., `AuthContext` , `RoutingContext`), passed only to relevant middleware.
>   3. **Context key/value bag**: Use Go's `context.Context` with `context.WithValue` to store arbitrary values, avoiding a predefined struct.
> - **Decision**: We chose option 1, the single monolithic `RequestContext` .
> - **Rationale**: A single, well-defined struct provides **clarity and type safety**. Middleware developers know exactly what data is available and can rely on the compiler to catch field access errors. It also serves as living documentation of the gateway's capabilities. While it couples middleware to the central type, this is acceptable because the gateway is a cohesive framework, not a loose collection of libraries. The alternative key/value bag approach (`context.WithValue`) is prone to runtime errors due to type assertions and lacks discoverability.
> - **Consequences**:
>   - **Positive**: Excellent discoverability and type safety; easy to serialize for debugging; clear data flow.
>   - **Negative**: The struct can become large over time; all middleware depend on this central type, requiring recompilation when fields are added (though plugins are compiled-in anyway).

Option	Pros	Cons	Chosen?
Single monolithic context	Type safety; discoverability; clear data flow	Can become large; all middleware depend on it	**Yes**
Scoped contexts	Separation of concerns; smaller interfaces	More complex to manage; harder to share data across phases	No
Context key/value bag	Extreme flexibility; no central type dependency	Runtime type assertion errors; no discoverability; hard to debug	No

Common Pitfalls in Using the Data Model

⚠️ Pitfall: Directly Accessing `innerCtx` Instead of Using `Context()` Method

- **Description**: Middleware authors might be tempted to directly use `ctx.innerCtx` instead of calling the `ctx.Context()` method.
- **Why It's Wrong**: This breaks encapsulation. If we later change how the internal context is stored (e.g., wrap it with additional functionality), all direct field accesses will break. The `Context()` method guarantees a stable interface.
- **Fix**: Always use the public method: `goCtx := ctx.Context()` .

⚠️ Pitfall: Not Checking for Nil Pointers in `MatchedRoute` or `SelectedEndpoint`

- **Description**: A middleware that assumes `ctx.MatchedRoute` or `ctx.SelectedEndpoint` is always non-nil may panic if it runs before the router or if routing fails.
- **Why It's Wrong**: The pipeline execution order may vary. Some middleware (like global logging) might run before routing. Also, errors in routing could leave these fields nil.
- **Fix**: Always check for nil before dereferencing: `if ctx.MatchedRoute != nil { ... }` . Design middleware to be tolerant of missing data.

⚠️ **Pitfall: Modifying `PluginConfig` Map After Configuration Load**

- **Description**: Since `PluginConfig.Config` is a `map[string]interface{}`, it might be modified at runtime by middleware, accidentally changing the configuration for future requests.
- **Why It's Wrong**: Configuration should be immutable after load to ensure consistent behavior and avoid race conditions. Modifying it can lead to unpredictable behavior and concurrency bugs.
- **Fix**: Treat the configuration as read-only. If a plugin needs to store per-request state, it should use the `RequestContext` (e.g., in `LogFields` or a custom map). If it needs to cache validated config, it should do so in its own internal struct, not modify the original map.

⚠️ **Pitfall: Forgetting to Call `Cancel()` on the Request Context**

- **Description**: The `cancelFn` is created when the context is initialized but must be called to release resources associated with the context. Failing to call it can lead to memory leaks.
- **Why It's Wrong**: The Go context cancellation tree may hold references that prevent garbage collection. In long-running servers, this can slowly consume memory.
- **Fix**: Ensure the gateway's request handler has a `defer ctx.Cancel()` after creating the context, or call it explicitly after writing the response.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Configuration Loading	YAML with `gopkg.in/yaml.v3`	YAML with validation using `go-playground/validator`
Context Management	Standard `context.Context` with cancellation	`context.Context` with deadlines and values for internal use only
Structured Logging	`LogFields` map formatted via `fmt` or `log.Printf`	Integration with `sirupsen/logrus` or `uber-go/zap`
Plugin Config Validation	Manual type assertions in plugin `Init()`	JSON Schema validation with `github.com/xeipuuv/gojsonschema`

B. Recommended File/Module Structure

```
api-gateway/
├── cmd/
│   └── server/
│       └── main.go                    # Entry point, loads config, starts server
├── internal/
│   ├── config/
│   │   ├── config.go                  # Defines Config, Route, Upstream, etc.
│   │   ├── loader.go                  # LoadFromFile, validate functions
│   │   └── config_test.go
│   ├── context/
│   │   ├── request_context.go         # RequestContext struct and methods
│   │   └── request_context_test.go
│   ├── middleware/                    # Middleware implementations
│   │   ├── middleware.go              # Middleware and Chain interfaces
│   │   ├── router.go                  # Routing middleware
│   │   ├── auth/
│   │   │   └── jwt_validator.go       # JWT plugin
│   │   └── ...                        # Other middleware
│   └── pipeline/
│       ├── pipeline.go                # Pipeline and Chain structs
│       └── builder.go                 # NewPipeline function
└── configs/
    └── gateway.yaml                   # Example configuration file
```

C. Infrastructure Starter Code: Configuration Loader

The configuration loader is a prerequisite that reads the YAML file and unmarshals it into the typed `Config` struct. Here is a complete, ready-to-use implementation.

File: `internal/config/loader.go`

```go
package config

import (
    "errors"
    "fmt"
    "io/ioutil"
    "os"

    "gopkg.in/yaml.v3"
)

// LoadFromFile reads a YAML configuration file from the given path and
// returns a parsed Config struct. It performs basic validation.
func LoadFromFile(filepath string) (*Config, error) {
    data, err := ioutil.ReadFile(filepath)
    if err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

    var cfg Config
    if err := yaml.Unmarshal(data, &cfg); err != nil {
        return nil, fmt.Errorf("failed to unmarshal YAML: %w", err)
    }

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

    return &cfg, nil
}

// validate performs basic sanity checks on the configuration.
// It ensures no duplicate IDs, required fields are present, and
// references between routes and upstreams are valid.
func validate(cfg *Config) error {
    if cfg.ListenAddr == "" {
        return errors.New("ListenAddr must be set")
    }

    // Check for duplicate upstream IDs
    upstreamIDs := make(map[string]bool)
    for _, u := range cfg.Upstreams {
        if u.ID == "" {
            return fmt.Errorf("Upstream must have an ID")
```

```
        }
        if upstreamIDs[u.ID] {
            return fmt.Errorf("duplicate upstream ID: %s", u.ID)
        }
        upstreamIDs[u.ID] = true

        // Validate endpoints
        if len(u.Endpoints) == 0 {
            return fmt.Errorf("upstream %s must have at least one endpoint", u.ID)
        }
        for _, ep := range u.Endpoints {
            if ep.ID == "" || ep.URL == "" {
                return fmt.Errorf("endpoint in upstream %s must have ID and URL", u.ID)
            }
        }
    }

    // Check routes and their references
    routeIDs := make(map[string]bool)
    for _, r := range cfg.Routes {
        if r.ID == "" {
            return errors.New("Route must have an ID")
        }
        if routeIDs[r.ID] {
            return fmt.Errorf("duplicate route ID: %s", r.ID)
        }
        routeIDs[r.ID] = true

        if r.Match.Path == "" {
            return fmt.Errorf("route %s must have a path match", r.ID)
        }
        if r.UpstreamID == "" {
            return fmt.Errorf("route %s must specify an upstream ID", r.ID)
        }
        // Ensure the referenced upstream exists
        if !upstreamIDs[r.UpstreamID] {
            return fmt.Errorf("route %s references unknown upstream %s", r.ID, r.UpstreamID)
        }
    }

    return nil
}
```

File: `internal/config/config.go`

```go
package config

// Config is the root configuration structure for the API Gateway.
type Config struct {
    ListenAddr string     `yaml:"listenAddr"`
    Routes     []Route    `yaml:"routes"`
    Upstreams  []Upstream `yaml:"upstreams"`
}

// Route defines a rule for matching incoming requests and forwarding them to an upstream.
type Route struct {
    ID          string        `yaml:"id"`
    Description string        `yaml:"description,omitempty"`
    Match       RouteMatch    `yaml:"match"`
    UpstreamID  string        `yaml:"upstreamId"`
    Plugins     []PluginConfig `yaml:"plugins,omitempty"`
}

// RouteMatch defines the conditions for a request to match a route.
type RouteMatch struct {
    Path    string            `yaml:"path"`
    Method  string            `yaml:"method,omitempty"` // empty means any method
    Headers map[string]string `yaml:"headers,omitempty"`
}

// Upstream represents a logical backend service with multiple endpoints.
type Upstream struct {
    ID        string     `yaml:"id"`
    Name      string     `yaml:"name,omitempty"`
    Endpoints []Endpoint `yaml:"endpoints"`
}

// Endpoint is a physical instance of an upstream service.
type Endpoint struct {
    ID  string `yaml:"id"`
    URL string `yaml:"url"`
}

// PluginConfig holds the configuration for a single plugin.
type PluginConfig struct {
    Name   string                 `yaml:"name"`
    Config map[string]interface{} `yaml:"config,omitempty"`
}
```

D. Core Logic Skeleton Code: Request Context

The `RequestContext` is a core component that you will implement. Below is skeleton code with detailed TODO comments mapping to the design described earlier.

File: `internal/context/request_context.go`

```go
package context

import (
    "context"
    "net"
    "net/http"
    "time"

    "github.com/google/uuid"
)

// RequestContext carries the state of a single HTTP request through the gateway pipeline.
type RequestContext struct {
    // Public fields
    RequestID        string
    StartTime        time.Time
    ClientIP         string
    Authenticated    bool
    UserID           string
    UserRoles        []string
    AuthMethod       string
    MatchedRoute     *config.Route
    SelectedUpstream *config.Upstream
    SelectedEndpoint *config.Endpoint
    RequestBody      []byte
    ResponseBody     []byte
    LogFields        map[string]interface{}
    CustomMetrics    map[string]float64
    Error            error
    HTTPErrorCode    int

    // Private fields
    innerCtx  context.Context
    cancelFn  context.CancelFunc
}

// NewRequestContext creates a new RequestContext for an incoming HTTP request.
// It extracts client IP, generates a request ID, and initializes the internal context.
func NewRequestContext(r *http.Request) *RequestContext {
    // TODO 1: Generate a unique request ID using uuid.New().String()
    // TODO 2: Capture the current time as StartTime
    // TODO 3: Extract the client IP from r.RemoteAddr (use net.SplitHostPort) or from X-Forwarded-For header if present
    // TODO 4: Create a cancellable context using context.WithCancel(context.Background())
```

```
    // TODO 5: Initialize the RequestContext struct with these values and zero values for other fields
    // Hint: For ClientIP, you can write a helper function: func extractClientIP(r *http.Request) string
    // TODO 6: Return a pointer to the newly created RequestContext
    return nil
}

// Context returns the underlying Go context for cancellation and deadlines.
func (ctx *RequestContext) Context() context.Context {
    // TODO 1: Return the innerCtx field
    return nil
}

// Cancel cancels the request's context, releasing associated resources.
func (ctx *RequestContext) Cancel() {
    // TODO 1: If cancelFn is not nil, call it
}

// AddLogField adds a key-value pair to the structured log fields.
func (ctx *RequestContext) AddLogField(key string, value interface{}) {
    // TODO 1: If LogFields is nil, initialize it with make(map[string]interface{})
    // TODO 2: Assign the key-value pair to the map
}

// ElapsedTime returns the duration since the request started.
func (ctx *RequestContext) ElapsedTime() time.Duration {
    // TODO 1: Calculate time.Since(ctx.StartTime) and return it
    return 0
}

// Helper function to extract client IP from request.
func extractClientIP(r *http.Request) string {
    // TODO 1: Check X-Forwarded-For header first; if present, take the first IP (comma-separated list)
    // TODO 2: Otherwise, parse r.RemoteAddr using net.SplitHostPort to get the IP (ignore port)
    // TODO 3: Return the IP address as a string
    return ""
}
```

E. Language-Specific Hints (Go)

- **UUID Generation**: Use `github.com/google/uuid` for generating request IDs: `uuid.New().String()`.
- **Time Measurement**: Use `time.Now()` for `StartTime` and `time.Since(start)` for elapsed duration. This is more efficient than calling `time.Now()` twice and subtracting.
- **Context Cancellation**: Always call `cancelFn` when the request is complete to avoid context leaks. A `defer ctx.Cancel()` in the HTTP handler is a good pattern.
- **Maps Initialization**: Before adding to `LogFields` or `CustomMetrics`, check if the map is `nil` and initialize with `make`. This avoids nil pointer panics.

- **Extracting Client IP**: Be careful with `X-Forwarded-For` as it can be spoofed. For internal use, the first IP in the list is typically the original client if the gateway is behind a trusted proxy. Use `net.SplitHostPort` to handle the port in `RemoteAddr`.

F. Milestone Checkpoint (Data Model)

After implementing the configuration loader and request context, you can verify the basics with a simple test:

1. **Create a test configuration file** `test_config.yaml`:

```yaml
listenAddr: ":8080"

upstreams:

  - id: "test-service"

    name: "Test Service"

    endpoints:

      - id: "test-1"

        url: "http://localhost:9999"

routes:

  - id: "test-route"

    description: "Test route"

    match:

      path: "/test/*"

      method: "GET"

    upstreamId: "test-service"
```

2. **Write a small Go program** to load and validate it:

```go
package main


import (

    "fmt"

    "log"

    "your-project/internal/config"

)


func main() {

    cfg, err := config.LoadFromFile("test_config.yaml")

    if err != nil {

        log.Fatal(err)

    }

    fmt.Printf("Loaded config: listening on %s\n", cfg.ListenAddr)

    fmt.Printf("Routes: %d, Upstreams: %d\n", len(cfg.Routes), len(cfg.Upstreams))

}
```

3. **Expected output**:

```
Loaded config: listening on :8080
Routes: 1, Upstreams: 1
```

4. **Signs of trouble**:

    - If you get "failed to unmarshal YAML", check your YAML syntax (indentation, colons).

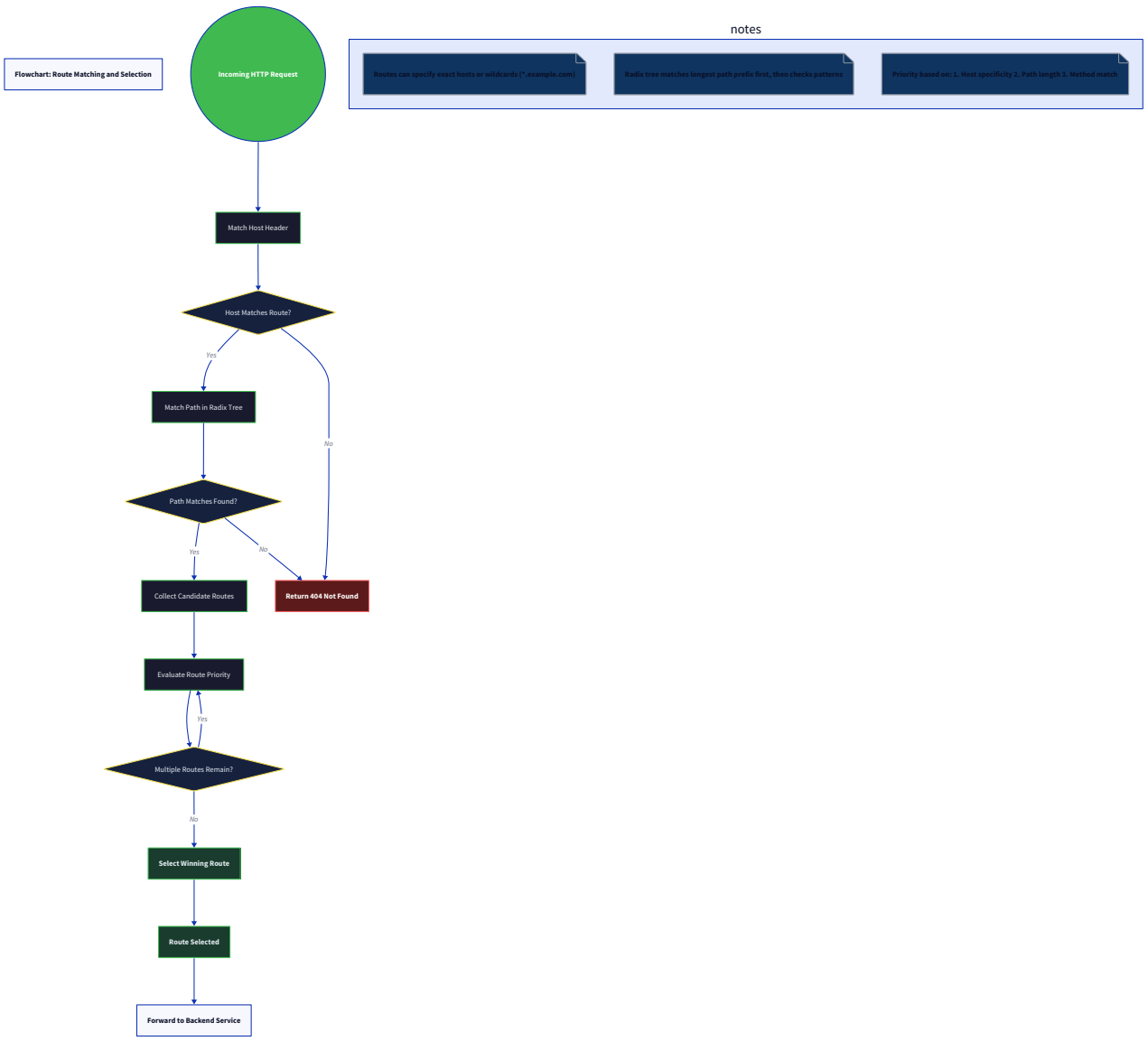
- If validation fails, ensure IDs are unique and upstream references are correct.
- Ensure the `config` package is properly imported (adjust the import path to match your module name).

5. **Test the RequestContext** with a simple HTTP handler mock to verify IP extraction and ID generation work as expected.

Component: Reverse Proxy & Routing Core

> **Milestone(s):** Milestone 1: Reverse Proxy & Routing

This component forms the foundational engine of the API Gateway. Its primary responsibility is to accept incoming HTTP requests, determine which backend service should handle them, and efficiently proxy the traffic to a healthy instance of that service. Think of it as the gateway's central nervous system for traffic flow—without it, requests would have nowhere to go.



Mental Model: The Airport Router

Imagine a large international airport. Thousands of passengers (requests) arrive at the main terminal (the gateway) every hour, each trying to reach a different final destination (backend service). The airport's flight information display system (the router) is critical: it examines each passenger's ticket (the request's path and headers) and directs them to the correct departure gate (the upstream endpoint). Some gates service multiple destinations via connecting flights (path prefixes), while others are exclusive to a single airline (host-based routing). The ground crew (load balancer) at each gate manages the queue of passengers, ensuring they board available aircraft (healthy backend instances) in an orderly fashion. If a particular aircraft is undergoing maintenance (unhealthy endpoint), the crew redirects

passengers to another plane, and if the entire route is experiencing severe weather (circuit breaker open), the airport may temporarily suspend all flights to that destination to avoid stranding passengers. This mental model captures the essence of routing, load balancing, and failure handling.

Interface and Behavior

The routing core is orchestrated by a `Router` interface. This interface abstracts the complex logic of matching requests to backend services and selecting a specific server instance. The primary method `FindRoute` is the entry point for this process.

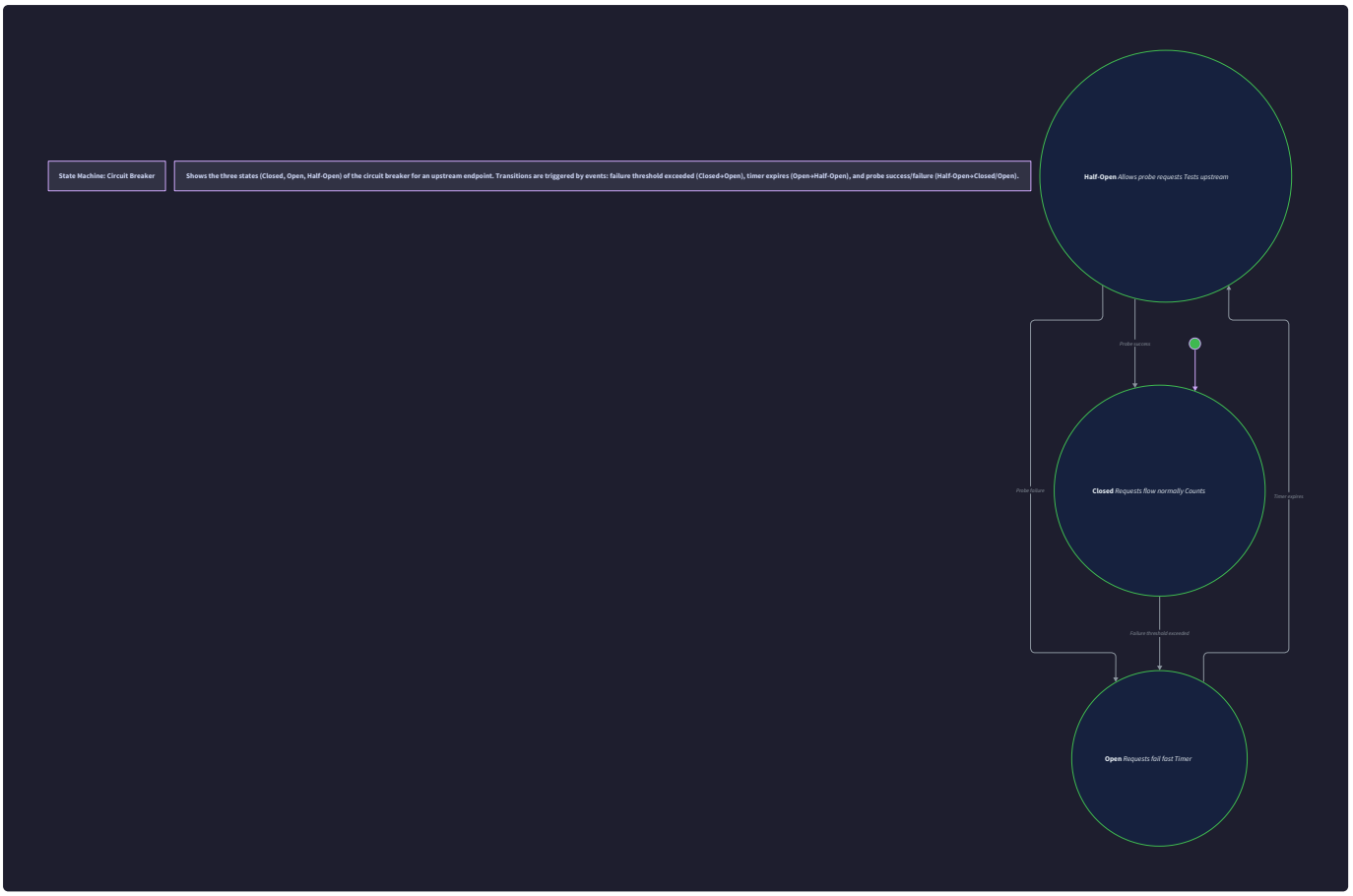
Method Name	Parameters	Returns	Description
`FindRoute`	`r *http.Request`	`(*Route, *Upstream, *Endpoint, error)`	Examines the incoming HTTP request and returns the matched route, the target upstream service, and a specific endpoint (server instance). Returns an error if no route matches or no healthy endpoints are available.
`RegisterRoute`	`route Route`	`error`	Adds a new route to the routing table. The router must rebuild its internal data structures (e.g., radix tree) to incorporate the new route.
`RefreshHealth`	`upstreamID string`		Triggers a re-evaluation of health status for all endpoints belonging to the specified upstream. This is typically called by the health check subsystem.

The algorithm for handling a request follows a precise, sequential flow. The following numbered steps describe what happens from the moment the gateway receives an HTTP request until it forwards it to a backend:

1. **Request Reception:** The gateway's HTTP server accepts the connection and creates an HTTP request object from the incoming data.
2. **Context Creation:** `NewRequestContext(r *http.Request)` is called, initializing a `RequestContext` that will track this request's journey.
3. **Route Matching:** The `Router.FindRoute` method is invoked with the request.
    1. **Host Matching:** The router first filters the configured `Route` list by comparing the request's `Host` header against the `RouteMatch.Headers["Host"]` field. If no host is specified in the route, it matches any host.
    2. **Path Matching:** The router then matches the request's URL path against the `RouteMatch.Path` field of the remaining routes. It uses a **Radix Tree** for efficient prefix and exact path matching (see ADR below).
    3. **Method Matching:** The HTTP method (GET, POST, etc.) is checked against `RouteMatch.Method`. A blank method matches all.
    4. **Header Matching:** Any additional headers specified in `RouteMatch.Headers` are verified to exist and match their values in the request.
    5. **Selection:** If multiple routes match, the router selects the one with the most specific match (longest path prefix, then most matching headers). The matched `Route` is stored in `RequestContext.MatchedRoute`.
4. **Upstream Resolution:** Using the matched route's `UpstreamID`, the router retrieves the corresponding `Upstream` configuration, which contains a list of `Endpoint` objects (backend instances). This is stored in `RequestContext.SelectedUpstream`.
5. **Load Balancing & Health Check Integration:** The router consults the load balancer for the selected upstream.
    1. The load balancer maintains a list of endpoints, each with a current health status (healthy, unhealthy, unknown).
    2. It applies the configured strategy (e.g., round-robin) but **only across endpoints marked as healthy**.
    3. The selected `Endpoint` (a specific backend server URL) is stored in `RequestContext.SelectedEndpoint`.
6. **Circuit Breaker Check:** Before forwarding, the router checks the circuit breaker state for the selected endpoint. If the circuit is **Open**, the request is immediately failed (typically with a 503 Service Unavailable). If it's **Half-Open**, only a试探性 request is allowed to pass (see state machine below).
7. **Request Forwarding:** The router uses a reverse proxy (Go's `httputil.ReverseProxy`) to forward the HTTP request to the `SelectedEndpoint.URL`. Critical original client information (IP, protocol) is added to headers like `X-Forwarded-For`.
8. **Response & Failure Handling:** The proxy waits for the backend response.
    1. **On Success (2xx, 3xx status codes):** The response is passed back through the gateway's post-processing pipeline to the client. The circuit breaker (if in Half-Open state) is reset to Closed.
    2. **On Failure (5xx, connection timeout, refusal):** The failure is recorded by the passive health check tracker. After a configured number of consecutive failures, the circuit breaker for that endpoint may **trip** (transition to Open). The gateway may retry the request on a different endpoint if retry logic is configured.
9. **Cleanup:** Regardless of outcome, the `RequestContext` is finalized, and observability data (logs, metrics) is emitted.

Circuit Breaker State Machine

The circuit breaker is a resilience pattern that prevents cascading failures by stopping requests to a repeatedly failing endpoint. It operates as a finite state machine with three distinct states. The state transitions are event-driven, as detailed in the table below and visualized in



.

Current State	Event Trigger	Next State	Action Taken
Closed (Normal)	Request succeeds	Closed	Increment success counter; Reset consecutive failure count to 0.
	Request fails	Closed	Increment consecutive failure count. If count ≥ `failureThreshold`, trip the breaker.
	Consecutive failures ≥ threshold	**Open**	Start the `openTimeout` timer. Record time of trip. All subsequent requests fail fast.
Open (Failed)	`openTimeout` timer expires	**Half-Open**	Allow the next single request through as a probe. Reset probe flag.
	Any request arrives (before timeout)	Open	Immediately fail fast (e.g., return 503).
Half-Open (Probing)	Probe request succeeds	**Closed**	Reset failure count to 0. Resume normal operation.
	Probe request fails	**Open**	Restart the `openTimeout` timer. Return to failing fast.

> **Key Insight:** The circuit breaker's purpose is to give the failing backend time to recover *without* being overwhelmed by new requests. The `Half-Open` state is crucial—it prevents a recovered service from being immediately flooded by a backlog of pent-up requests, which could knock it over again (the "thundering herd" problem).

ADR: Routing Match Strategy

> **Decision: Use a Radix Tree for Path Matching**
>
> - **Context**: The gateway must match incoming request paths to configured routes with high performance (low latency) and support for common patterns like prefix matching (`/api/users*`). The matching logic is executed for every single request, so its efficiency is critical to overall throughput.
> - **Options Considered**:
>   1. **Linear Search with Prefix Check**: Iterate through all routes, checking if the request path starts with the route's path prefix.
>   2. **Regular Expression Matching**: Define routes using regex patterns for maximum flexibility.
>   3. **Radix Tree (Prefix Tree)**: Compress all route paths into a tree structure where each node represents a common path segment.
> - **Decision**: Implement route matching using a **Radix Tree**.
> - **Rationale**:
>   - **Performance**: A radix tree performs matching in **O(k)** time (where *k* is the length of the path), which is vastly superior to linear search **O(n)** for a large number of routes. It allows for near-instantaneous lookups.
>   - **Memory Efficiency**: It compresses common prefixes, so storing thousands of routes (e.g., `/api/v1/users` , `/api/v1/products`) consumes less memory than a simple list.
>   - **Adequate Flexibility**: It naturally supports both exact matches and prefix/wildcard matches (by treating a trailing `*` or `/*` as a special wildcard node). This covers the vast majority of API gateway routing needs.
>   - **Predictability**: Unlike regex, radix tree matching is deterministic and easier to reason about, simplifying debugging and configuration validation.
> - **Consequences**:
>   - **Enables**: High-performance routing suitable for high-throughput gateways. Clear precedence rules (longest prefix wins).
>   - **Introduces**: The complexity of implementing or integrating a radix tree library. Slightly less matching power than full regex (e.g., cannot match `/api/v[0-9]/`), though this can be supplemented with limited regex support for specific nodes if absolutely needed.

Option	Pros	Cons	Chosen?
Linear Search	Simple to implement, easy to debug.	Performance degrades linearly with number of routes; unacceptable for production with 100+ routes.	❌
Regex Matching	Extremely flexible and powerful pattern matching.	Can be computationally expensive (slow); difficult to optimize and order (which regex wins?).	❌
Radix Tree	Very fast lookup, memory-efficient, supports prefixes naturally.	Implementation is more complex; less flexible than regex for complex patterns.	✅

ADR: Load Balancing with Passive Health Checks

> **Decision: Implement passive (failure-based) health checks as primary, with optional active probes as a fallback.**
>
> - **Context**: The gateway must distribute load across multiple backend instances and avoid sending traffic to instances that are failing. We need a mechanism to detect unhealthy backends. This detection should be reliable, timely, and not create excessive overhead.
> - **Options Considered**:
>   1. **Active Health Checks (Probing)**: The gateway periodically sends HTTP requests (e.g., `GET /health`) to each backend endpoint and marks it unhealthy if probes fail.
>   2. **Passive Health Checks (Failure Observation)**: The gateway monitors the outcome of real user requests. If an endpoint returns a certain number of consecutive errors (e.g., 5xx HTTP status), it is marked unhealthy.
>   3. **Hybrid Approach**: Use passive checks as the primary signal, but use infrequent active checks to resurrect endpoints that are not receiving traffic.
> - **Decision**: Implement a **Hybrid Approach**, with **Passive Health Checks** as the primary, real-time failure detector, and a configurable, low-frequency **Active Health Check** as a background recovery mechanism.
> - **Rationale**:
>   - **Real-World Signal**: Passive checks react to actual user-experienced failures. If a backend starts returning 500 errors, passive checks will detect it immediately on the next request, without waiting for the next scheduled probe.
>   - **Efficiency**: It adds no extra network traffic or load on backends. The health signal is a free byproduct of real traffic.
>   - **Recovery for Idle Endpoints**: A solely passive system has a blind spot: an endpoint that receives no traffic cannot prove it has recovered. Infrequent active probes (e.g., every 30 seconds) solve this by providing a heartbeat for idle endpoints.
>   - **Simplicity**: The logic for tracking request failures is straightforward and can be integrated directly into the request/response handling loop.
> - **Consequences**:
>   - **Enables**: Fast failure detection under load, low overhead, and recovery of idle backends.
>   - **Introduces**: A delay in detecting failures for endpoints under very low or no traffic. The need to manage two parallel health check systems.

Option	Pros	Cons	Chosen?
Active Only	Can check endpoints without traffic; can define custom health check logic.	Adds latency to failure detection (must wait for next probe); creates extra load; may not reflect real user experience.	❌
Passive Only	Zero overhead; reacts instantly to real failures under load.	Cannot detect failure or recovery of idle endpoints; may be slow to detect intermittent failures under low traffic.	❌
Hybrid (Passive Primary)	Fast failure detection under load; can recover idle endpoints; efficient.	More complex to implement; requires tuning two sets of parameters (failure thresholds, probe intervals).	✅

Common Pitfalls

⚠️ Pitfall: Forgetting to Forward the Original Client IP

- **Description:** The gateway proxies requests, so the backend sees the gateway's IP address as the client. If the gateway doesn't add the `X-Forwarded-For` or similar headers, the backend loses all visibility into the original client's identity, breaking geolocation, rate limiting, and audit logs.
- **Why it's wrong:** Critical security and operational features in the backend may depend on the client IP. Debugging becomes difficult.
- **How to fix:** Always set the `X-Forwarded-For` header (appending the client IP), and also set `X-Forwarded-Proto` and `X-Forwarded-Host`. In Go's `httputil.ReverseProxy`, this is done in the `Director` function.

⚠️ Pitfall: Blocking Request Handling with Synchronous Health Checks

- **Description:** Performing an active health check (HTTP probe) synchronously right before forwarding a user request. This adds significant latency (the health check round-trip time) to every single user request.
- **Why it's wrong:** It dramatically increases response times and makes the gateway's performance dependent on backend health check endpoints.
- **How to fix:** Decouple health checking from request handling. Run active checks in a separate, low-frequency background goroutine. Use a shared, concurrently safe health status map that the router reads from.

⚠️ Pitfall: Connection Leaks from Unclosed Response Bodies

- **Description:** When the gateway reads a backend's response, failing to close the response body (`resp.Body.Close()`) after reading it fully. This keeps underlying TCP connections open, eventually exhausting file descriptors and causing the gateway to crash.
- **Why it's wrong:** It causes resource exhaustion, which is a severe reliability issue. In Go, you must close the body to reuse the HTTP/1.1 connection or properly terminate the HTTP/2 stream.
- **How to fix:** Always `defer resp.Body.Close()` after a successful backend call. Use `io.Copy` or `io.ReadAll` to drain the body before closing.

⚠️ Pitfall: Thundering Herd on Backend Recovery

- **Description:** When a failed backend recovers and its circuit breaker transitions from Open to Half-Open, a sudden flood of pending requests (that were queued or retried) all try to go through at once. This can immediately overwhelm the freshly recovered backend, causing it to fail again.
- **Why it's wrong:** It prevents stable recovery and can lead to a service oscillating between up and down states.
- **How to fix:** The circuit breaker's Half-Open state should allow **only one** probing request through initially. Only after that succeeds should it gradually allow more traffic (e.g., by slowly increasing a permitted request quota). Implement retry with exponential backoff and jitter for clients.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
HTTP Reverse Proxy	Go's `net/http/httputil.ReverseProxy` (battle-tested, handles Websockets, good defaults)	Custom proxy using `net/http` Transport for fine-grained control over timeouts, keep-alives, and connection pooling.
Radix Tree Router	`github.com/gorilla/mux` (powerful, includes radix tree, but larger API)	`github.com/julienschmidt/httprouter` (minimal, very fast radix tree router) or a custom tree based on `github.com/armon/go-radix`.
Load Balancer	Simple round-robin with a slice and atomic counter.	Weighted round-robin or least-connections using a priority queue/heap.
Circuit Breaker	Manual state machine with `sync.Mutex` and `time.Timer`.	Use `github.com/sony/gobreaker` (production-ready, includes metrics).
Health Status Storage	In-memory `map[string]bool` protected by a `sync.RWMutex`.	`sync.Map` for better concurrent read performance, or a sharded map to reduce lock contention.

B. Recommended File/Module Structure

Place the routing core logic within a dedicated package to separate concerns.

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go
├── internal/
│   ├── config/
│   │   ├── config.go          # Config, Route, Upstream structs
│   │   └── load.go            # LoadFromFile, validate
│   ├── router/                # ← This component
│   │   ├── router.go          # Router interface and main implementation
│   │   ├── radix_tree.go      # Radix tree implementation for path matching
│   │   ├── loadbalancer.go    # LoadBalancer interface & round-robin impl
│   │   ├── healthcheck.go     # Passive/active health check logic
│   │   ├── circuitbreaker.go  # Circuit breaker state machine
│   │   └── router_test.go
│   ├── middleware/            # Other middleware (auth, transform, etc.)
│   └── pipeline/
│       ├── pipeline.go        # Pipeline, Chain
│       └── context.go         # RequestContext and methods
└── configs/
    └── gateway.yaml           # Example configuration
```

C. Infrastructure Starter Code

Complete Reverse Proxy Wrapper (simplified): This is a ready-to-use wrapper around `httputil.ReverseProxy` that handles context propagation and basic header management.

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

package router


import (

    "context"

    "net"

    "net/http"

    "net/http/httputil"

    "strings"

)


type GatewayProxy struct {

    proxy *httputil.ReverseProxy

}


func NewGatewayProxy(upstreamURL string) *GatewayProxy {

    target, _ := url.Parse(upstreamURL)

    proxy := httputil.NewSingleHostReverseProxy(target)


    // Customize the Director function to set essential headers

    originalDirector := proxy.Director

    proxy.Director = func(req *http.Request) {

        originalDirector(req)

        // Preserve the original client IP

        if clientIP, _, err := net.SplitHostPort(req.RemoteAddr); err == nil {

            if prior, ok := req.Header["X-Forwarded-For"]; ok {

                clientIP = strings.Join(prior, ", ") + ", " + clientIP

            }

            req.Header.Set("X-Forwarded-For", clientIP)

        }

        req.Header.Set("X-Forwarded-Host", req.Host)

        if req.TLS == nil {

            req.Header.Set("X-Forwarded-Proto", "http")

        } else {

            req.Header.Set("X-Forwarded-Proto", "https")

        }

        // Inject the request's context (for cancellation/timeouts)

        req = req.WithContext(req.Context())

    }


    // Optional: Customize error handling

    proxy.ErrorHandler = func(w http.ResponseWriter, r *http.Request, err error) {

        // Log error, maybe update circuit breaker
```

```
        http.Error(w, "Bad Gateway", http.StatusBadGateway)
    }

    return &GatewayProxy{proxy: proxy}
}

func (gp *GatewayProxy) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    gp.proxy.ServeHTTP(w, r)
}
```

D. Core Logic Skeleton Code

1. Radix Tree-Based Router `FindRoute` Method:

```go
// internal/router/router.go

type RouterImpl struct {
    routes   map[string]Route          // RouteID -> Route
    upstreams map[string]Upstream      // UpstreamID -> Upstream
    tree     *radix.Tree              // Path prefix tree; assume a radix.Tree type exists
    mu       sync.RWMutex
}

// FindRoute implements the Router interface.
func (ri *RouterImpl) FindRoute(r *http.Request) (*Route, *Upstream, *Endpoint, error) {
    // TODO 1: Acquire read lock (ri.mu.RLock()) for thread-safe reading of routes/upstreams.
    // TODO 2: Extract the request's Host header and HTTP method.
    // TODO 3: Use the radix tree's longest prefix match to get candidate route IDs for the request path.
    // TODO 4: Filter candidates: check that the request Host and Method match the Route's Match criteria.
    // TODO 5: If multiple routes remain, apply tie-breaking (e.g., longest path, then most headers matched).
    // TODO 6: If no route matches, return a 404 error.
    // TODO 7: Retrieve the Upstream using the matched route's UpstreamID.
    // TODO 8: Call loadBalancer.SelectEndpoint(upstream) to pick a healthy endpoint (see next skeleton).
    // TODO 9: If no healthy endpoint is available, return a 503 error.
    // TODO 10: Check the circuit breaker state for the selected endpoint; if open, return 503.
    // TODO 11: Return the matched Route, Upstream, and selected Endpoint.
    // Hint: Store the selected endpoint in the RequestContext later in the pipeline.
}
```

2. Round-Robin Load Balancer with Health Status:

```go
// internal/router/loadbalancer.go

type RoundRobinLoadBalancer struct {
    upstreamID string
    endpoints  []*Endpoint          // All endpoints
    healthStatus map[string]bool    // EndpointID -> isHealthy
    index      uint32               // Atomic counter for round-robin
    mu         sync.RWMutex
}

// SelectEndpoint returns a healthy endpoint using round-robin strategy.
func (lb *RoundRobinLoadBalancer) SelectEndpoint() (*Endpoint, error) {
    // TODO 1: Acquire read lock (lb.mu.RLock()) to snapshot health status.
    // TODO 2: Build a list of healthy endpoints by iterating through lb.endpoints and checking lb.healthStatus.
    // TODO 3: If no healthy endpoints exist, return an error.
    // TODO 4: Use atomic.AddUint32 on lb.index to get the next index in a thread-safe way.
    // TODO 5: Mod the index by the length of the healthy list to get the selected endpoint.
    // TODO 6: Return the selected endpoint.
    // Hint: Consider implementing a 'WeightedRoundRobin' variant where each endpoint has a weight field.
}
```

3. Circuit Breaker State Machine:

GO

```go
// internal/router/circuitbreaker.go

type CircuitState int

const (
    StateClosed CircuitState = iota
    StateOpen
    StateHalfOpen
)

type CircuitBreaker struct {
    endpointID      string
    failureThreshold int
    openTimeout     time.Duration
    state           CircuitState
    failures        int
    lastFailureTime time.Time
    mu              sync.Mutex
}

// AllowRequest determines if a request should be permitted based on the current state.
func (cb *CircuitBreaker) AllowRequest() bool {
    cb.mu.Lock()
    defer cb.mu.Unlock()

    // TODO 1: If state is Closed, return true (allow).
    // TODO 2: If state is Open, check if openTimeout has elapsed.
    //         If yes, transition to HalfOpen, set a probe flag, and allow this single probe request.
    //         If no, return false (block).
    // TODO 3: If state is HalfOpen, check the probe flag.
    //         If a probe is already in flight, return false. Otherwise, set the flag and allow.
    // TODO 4: Update lastFailureTime on state transitions as needed.
}

// RecordSuccess resets the breaker on a successful request.
func (cb *CircuitBreaker) RecordSuccess() {
    cb.mu.Lock()
    defer cb.mu.Unlock()
    // TODO 1: If state is HalfOpen, transition to Closed.
    // TODO 2: Reset failure count to 0.
    // TODO 3: Clear any probe flag.
}

// RecordFailure increments the failure count and may trip the breaker.
```

```
func (cb *CircuitBreaker) RecordFailure() {
    cb.mu.Lock()
    defer cb.mu.Unlock()
    // TODO 1: Increment failure count.
    // TODO 2: If failure count >= failureThreshold and state is Closed, transition to Open and set lastFailureTime.
    // TODO 3: If state is HalfOpen, transition to Open (the probe failed).
}
```

E. Language-Specific Hints

- **`httputil.ReverseProxy`:** Use its `Director` function to modify the outgoing request (add headers, change URL path). Use `ModifyResponse` to process the backend's response before it's sent to the client. Use `ErrorHandler` for custom error pages.
- **Atomic Operations:** For the round-robin index, use `atomic.AddUint32` for lock-free, thread-safe increments. Remember to bound the result using modulo.
- **Timeouts:** Always set timeouts on the `http.Client` used by the reverse proxy. Use `context.WithTimeout` on the request context to enforce a total deadline for the backend call.
- **Concurrency:** Protect shared maps (like `healthStatus`) with `sync.RWMutex`. This allows many concurrent readers but exclusive access for writers.

F. Milestone Checkpoint

After implementing the routing core, you should be able to verify its basic functionality.

1. **Run Unit Tests:**

```bash
cd internal/router
go test -v
```

`Expected output should show passing tests for route matching, load balancer selection, and circuit breaker state transitions.`

2. **Manual Verification:**
   - Start the gateway with a sample config that defines a route `/api/users` pointing to a mock backend (you can use `python3 -m http.server 8081` as a mock).
   - Use `curl` to send a request:

```bash
curl -v http://localhost:8080/api/users
```

`- **Expected:** The request is proxied to the mock backend on port 8081, and you see the backend's response. Check the backend's ac`
`- **Test Host-Based Routing:** Configure a route with `Headers: {Host: api.example.com}`. Use `curl -H "Host: api.example.com" ...``

3. **Signs of Trouble:**
   - **502 Bad Gateway:** The proxy cannot connect to the backend. Check the upstream URL and that the backend is running.
   - **Routes not matching:** Double-check the radix tree insertion logic and the order of matching criteria (Host → Path → Method → Headers).
   - **High CPU/Lock Contention:** If performance is poor under load, profile with `pprof`. Ensure you are using `RWMutex` and not a plain `Mutex` for read-heavy operations like route matching.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All requests return 404	Routes not loaded correctly or radix tree is empty.	Add debug logs in `FindRoute` to print the tree contents and the matching path.	Verify `LoadFromFile` is called and `RegisterRoute` is invoked for each route in config.
Requests are not load-balanced; always go to the first endpoint	Round-robin index is not being incremented atomically, or health filter excludes all but one endpoint.	Log the selected endpoint index and the list of healthy endpoints on each request.	Use `atomic.AddUint32` for the index. Check health status map updates.
Memory usage grows endlessly	Response bodies not closed, goroutines leaking (e.g., from active health checks).	Use `pprof` heap profile. Check for missing `resp.Body.Close()`.	Ensure `defer resp.Body.Close()` is in place. Use `context.WithTimeout` for health checks.
Backend receives no `X-Forwarded-For` header	The proxy's `Director` function is not setting the header.	Inspect the outgoing request headers in the proxy using debug logging or a tool like `mitmproxy`.	Ensure the `Director` function (in starter code) is correctly configured and executed.
Circuit breaker never opens	Failure threshold too high, or failures are not being recorded.	Log each call to `RecordFailure` and the current failure count.	Verify `RecordFailure` is called on 5xx responses and network errors. Adjust threshold.

Component: Request/Response Transformation

> **Milestone(s):** Milestone 2: Request/Response Transformation

This component enables the gateway to act as a protocol and format adapter, modifying requests before they reach backend services and responses before they return to clients. It provides the critical ability to bridge differences between client expectations and backend APIs without requiring changes to either side.

Mental Model: The Language Interpreter

Imagine two people who need to communicate but speak different dialects. Person A (the client) speaks with a certain vocabulary and sentence structure, while Person B (the backend service) understands a different set of terms and grammar. A language interpreter sits between them, listening to Person A's message, translating it into terms Person B will understand, then taking Person B's response and rephrasing it back into Person A's preferred dialect.

The transformation layer in our API Gateway serves exactly this role. It can:

1. **Translate vocabulary**: Change field names in JSON payloads (e.g., `user_id` → `userId`)
2. **Rephrase structure**: Reshape nested objects into flat structures or vice versa
3. **Add context**: Insert headers that provide additional context to the backend
4. **Filter content**: Remove sensitive or unnecessary data from responses
5. **Combine messages**: Merge multiple backend responses into a single coherent reply (aggregation)

This interpreter doesn't just translate word-for-word—it understands the intent behind the message and ensures the meaning is preserved even as the form changes. Like a skilled interpreter, it must handle malformed messages gracefully, avoid introducing misunderstandings, and work efficiently without causing unnecessary delay.

Interface and Transformation Pipeline

The transformation system is implemented as a series of middleware components that execute in a defined pipeline. Each transformation step operates on the `RequestContext` and can modify either the outgoing request or incoming response.

Transformation Middleware Interface

All transformation components implement the standard `Middleware` interface defined in the gateway's core architecture:

Method	Parameters	Returns	Description
`Name()`	None	`string`	Returns the unique identifier for this middleware (e.g., `"header-transformer"`)
`Priority()`	None	`int`	Determines execution order in the pipeline (lower numbers execute earlier)
`Execute()`	`ctx *RequestContext`, `w http.ResponseWriter`, `r *http.Request`	`bool`	Processes the request/response; returns `false` to stop further processing

Transformation Pipeline Stages

The transformation pipeline executes in three distinct phases within the overall request flow:

Phase	Timing	Typical Transformations
Request Pre-Processing	After authentication, before routing	Header manipulation for backend communication, URL rewriting, request body transformation
Request Post-Processing	After receiving backend response, before sending to client	Response header manipulation, response body transformation, error response formatting
Aggregation	Between multiple backend calls	Combining multiple responses, merging JSON payloads, handling partial failures

Transformation Configuration Data Model

Transformations are configured per-route through the `PluginConfig` mechanism. A typical transformation plugin configuration includes:

Field	Type	Description
`Name`	`string`	Plugin name (e.g., `"header-transformer"`, `"json-transformer"`)
`Config`	`map[string]interface{}`	Plugin-specific configuration. Common fields:

Header Transformer Configuration Example:

```yaml
Name: "header-transformer"

Config:

  request:

    add:

      X-API-Version: "v2"

      X-Forwarded-Service: "user-service"

    remove: ["X-Debug-Token"]

    rewrite:

      Authorization: "Bearer {{.Token}}"

  response:

    add:

      X-Response-Time: "{{.ElapsedMS}}"

    remove: ["X-Internal-Debug"]
```

JSON Body Transformer Configuration Example:

```yaml
Name: "json-transformer"

Config:

  request:

    mappings:

      - from: "userId"

        to: "user_id"

      - from: "emailAddress"

        to: "email"

  response:

    mappings:

      - from: "created_at"

        to: "createdAt"

    remove: ["internal_id", "audit_log"]
```

Transformation Execution Algorithm

For each request that matches a route with transformation plugins, the gateway executes the following algorithm:

1. **Initialize Transformation Context**: Create or update transformation-specific fields in `RequestContext`:
   - `RequestBody`: Initially empty, populated if body transformation is needed
   - `ResponseBody`: Initially empty, populated if response transformation is needed
   - `LogFields["transformation_stage"]`: Track current transformation phase
2. **Execute Request Transformations** (pre-routing): a. Extract and buffer request body up to configured size limit (e.g., 10MB) b. Apply header transformations in order of plugin priority:
   - Remove specified headers
   - Add new headers with static or templated values
   - Rewrite existing header values c. Apply URL rewriting:
   - Modify request path using regex patterns
   - Add, remove, or modify query parameters d. Apply request body transformations:
   - Parse JSON payload (if applicable)
   - Rename fields according to mapping rules
   - Add or remove fields
   - Validate transformed payload structure
3. **Forward Transformed Request**: Send the modified request to the selected backend endpoint
4. **Execute Response Transformations** (post-routing): a. Buffer backend response body up to configured size limit b. Apply response header transformations:
   - Remove internal headers
   - Add gateway-specific headers (e.g., `X-Gateway-Cache-Hit`) c. Apply response body transformations:
   - Parse JSON response (if applicable)
   - Apply field mappings in reverse (if bidirectional transformation)
   - Filter sensitive fields
   - Format data structure for client consumption
5. **Handle Aggregation** (if configured): a. For each parallel backend call, wait for response or timeout b. Merge successful responses according to merge strategy c. Handle partial failures with fallback values or error reporting d. Apply final transformations to aggregated response
6. **Send Final Response**: Write transformed headers and body to client connection

ADR: Streaming vs. Buffered Body Transformation

> **Decision: Buffered Transformation with Size Limits**
>
> **Context**: The gateway needs to transform request and response bodies (primarily JSON). Two fundamentally different approaches exist: streaming transformation (processing the body as it flows through) and buffered transformation (reading the entire body into memory before transforming). The choice significantly impacts memory usage, performance, and implementation complexity.
>
> **Options Considered**:
>
> 1. **Pure Streaming Transformation**: Process bytes as they pass through, using state machines for JSON parsing
> 2. **Buffered Transformation**: Read entire body into memory, transform, then forward
> 3. **Hybrid Approach**: Stream small bodies, buffer large ones with size-based switching
>
> **Decision**: We will implement buffered transformation with strict size limits (default 10MB for requests, 10MB for responses).
>
> **Rationale**:
>
> 1. **Implementation Simplicity**: Buffered transformation allows using standard JSON libraries (`encoding/json` in Go) which are robust, well-tested, and support complex transformations like nested field renaming
> 2. **Transformation Power**: Many transformations require full document visibility (e.g., moving a field from deep nesting to root level, conditional transformations based on multiple fields)
> 3. **Error Handling**: With the entire document in memory, we can provide detailed error messages about exactly where transformation failed
> 4. **Size Control**: By enforcing strict limits, we prevent memory exhaustion while gaining buffering benefits
> 5. **Practical Reality**: Most API payloads in microservices are under 1MB; the 10MB limit covers 99%+ of cases while protecting the gateway
>
> **Consequences**:
>
> - **Positive**: Simpler code, more powerful transformations, better error messages
> - **Negative**: Memory usage scales with concurrent large requests, larger latency for big payloads (due to buffering delay)
> - **Mitigation**: Implement connection timeouts, circuit breakers, and monitor memory usage; consider adding streaming support later for specific use cases

Comparison of Transformation Approaches:

Option	Pros	Cons	Chosen?
Pure Streaming	Constant memory usage, can handle unlimited size payloads	Complex implementation, limited transformation capabilities, difficult error reporting	No
Buffered with Limits	Simple implementation using standard libraries, supports complex transformations, good error messages	Memory usage scales with payload size, imposes maximum payload size	**Yes**
Hybrid Approach	Balances memory usage and transformation capabilities	Most complex implementation, difficult to debug, two code paths to maintain	No

Common Pitfalls

⚠️ Pitfall: Unbounded Body Buffering Causing Memory Exhaustion

- **Description**: Reading request/response bodies without size limits, allowing a malicious client or misbehaving backend to send a 1GB payload that consumes all gateway memory
- **Why It's Wrong**: Causes gateway crashes (OOM kills), enables denial-of-service attacks, and degrades performance for all users
- **How to Fix**: Always use `io.LimitReader` with configurable maximum sizes (default 10MB). Reject requests exceeding the limit with `413 Request Entity Too Large`

⚠️ Pitfall: Mishandling Content-Length After Transformation

- **Description**: Transforming a response body (e.g., removing fields from JSON) but forgetting to update the `Content-Length` header, causing HTTP protocol violations
- **Why It's Wrong**: Clients may hang waiting for more data or truncate responses. Some HTTP clients will reject the response entirely
- **How to Fix**: Always recalculate `Content-Length` after body transformation, or use chunked transfer encoding (`Transfer-Encoding: chunked`) for responses

⚠️ Pitfall: Error Propagation in Request Aggregation

- **Description**: When aggregating responses from multiple backends, a failure in one backend call causes the entire aggregated response to fail, even when partial data could be useful
- **Why It's Wrong**: Reduces system resilience and degrades user experience unnecessarily. Some clients can handle partial responses with error indicators

- **How to Fix**: Implement configurable aggregation strategies: `fail-fast` (return error immediately), `best-effort` (return successful parts with error metadata), or `fallback` (use default values for failed calls)

⚠️ Pitfall: Transforming Non-JSON Content as JSON

- **Description**: Assuming all request/response bodies are JSON and attempting to parse them as such, causing crashes or malformed output when clients send form data, XML, or binary content
- **Why It's Wrong**: Breaks legitimate non-JSON APIs, causes transformation errors for valid requests
- **How to Fix**: Check `Content-Type` header before attempting JSON transformation. Only apply JSON transformations when content type is `application/json` or a configured variant. Provide content-type-specific transformers for other formats if needed

⚠️ Pitfall: Not Preserving Idempotency of Safe HTTP Methods

- **Description**: Applying transformations that change the meaning of `GET`, `HEAD`, `OPTIONS`, or `TRACE` requests in ways that make them non-idempotent or unsafe
- **Why It's Wrong**: Violates HTTP semantics, breaks client caching assumptions, can cause data corruption if retried
- **How to Fix**: Only apply non-idempotent transformations (like adding unique IDs or timestamps) to safe methods if explicitly configured. Document this behavior clearly

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
JSON Transformation	Standard `encoding/json` for marshaling/unmarshaling	`github.com/tidwall/gjson` for selective extraction without full parse
Template Headers	Go's `text/template` for basic value substitution	Custom function map with request context accessors
Body Size Limiting	`io.LimitReader` wrapper	Configurable limits per route/plugin with LRU cache for large payloads
URL Rewriting	`regexp` package for pattern matching	`github.com/gorilla/mux` -style path templates with capture groups
Response Aggregation	Sequential calls with `sync.WaitGroup`	Parallel calls with bounded goroutine pool and error aggregation

B. Recommended File/Module Structure

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go                    # Entry point
├── internal/
│   ├── config/
│   │   └── config.go                  # Configuration structures
│   ├── middleware/                    # All middleware implementations
│   │   ├── middleware.go              # Base interface definitions
│   │   ├── chain.go                   # Middleware chaining logic
│   │   └── transformation/            # Transformation-specific middleware
│   │       ├── transformer.go         # Base transformer interface
│   │       ├── header_transformer.go  # Header manipulation
│   │       ├── body_transformer.go    # JSON body transformation
│   │       ├── url_rewriter.go        # URL path/query rewriting
│   │       ├── aggregator.go          # Request aggregation
│   │       └── limits.go              # Size limiting utilities
│   ├── pipeline/
│   │   └── pipeline.go                # Pipeline execution logic
│   └── gateway/
│       └── gateway.go                 # Main gateway type
└── pkg/
    └── template/
        └── template.go                # Template utilities for header values
```

C. Infrastructure Starter Code

Size-Limited Reader Wrapper (Complete Implementation):

GO

```go
// internal/middleware/transformation/limits.go

package transformation

import (
    "io"
    "net/http"
)

// LimitedReadCloser wraps an io.ReadCloser with a read limit
type LimitedReadCloser struct {
    io.Reader
    io.Closer
}

// NewLimitedReadCloser creates a reader that limits total bytes read
func NewLimitedReadCloser(rc io.ReadCloser, maxBytes int64) *LimitedReadCloser {
    return &LimitedReadCloser{
        Reader: io.LimitReader(rc, maxBytes),
        Closer: rc,
    }
}

// ReadRequestBody reads and returns the request body with size limit
func ReadRequestBody(r *http.Request, maxBytes int64) ([]byte, error) {
    // Replace request body with limited reader
    limitedBody := NewLimitedReadCloser(r.Body, maxBytes)
    r.Body = limitedBody

    // Read all (up to limit)
    body, err := io.ReadAll(limitedBody.Reader)
    if err != nil {
        if err == io.EOF && len(body) == 0 {
            // Empty body is OK
            return nil, nil
        }
        // Check if error is due to limit exceeded
        if err.Error() == "http: request body too large" {
            return nil, &BodySizeError{Max: maxBytes, Actual: maxBytes + 1}
        }
        return nil, err
    }
```

```
    return body, nil
}

// BodySizeError indicates request/response body exceeded size limit
type BodySizeError struct {
    Max    int64
    Actual int64
}

func (e *BodySizeError) Error() string {
    return "body size exceeds maximum allowed"
}

// IsBodySizeError checks if an error is a body size limit error
func IsBodySizeError(err error) bool {
    _, ok := err.(*BodySizeError)
    return ok
}
```

Template Utilities for Header Values (Complete Implementation):

```go
// pkg/template/template.go

package template

import (
    "bytes"
    "text/template"
    "time"
)

// RequestTemplateData provides data for template expansion in headers
type RequestTemplateData struct {
    RequestID  string
    ClientIP   string
    Timestamp  time.Time
    UserID     string
    UserRoles  []string
    AuthMethod string
    // Add more fields as needed
}

// ExecuteHeaderTemplate expands a template string using request data
func ExecuteHeaderTemplate(tmplStr string, data *RequestTemplateData) (string, error) {
    if tmplStr == "" {
        return "", nil
    }

    tmpl, err := template.New("header").Parse(tmplStr)
    if err != nil {
        return "", err
    }

    var buf bytes.Buffer
    if err := tmpl.Execute(&buf, data); err != nil {
        return "", err
    }

    return buf.String(), nil
}
```

D. Core Logic Skeleton Code

Header Transformer Middleware:

GO

```go
// internal/middleware/transformation/header_transformer.go

package transformation


import (

    "net/http"

    "api-gateway/internal/middleware"

    "api-gateway/internal/config"

)


// HeaderTransformerConfig holds configuration for header manipulation

type HeaderTransformerConfig struct {

    Request  HeaderOps `yaml:"request"`

    Response HeaderOps `yaml:"response"`

}


// HeaderOps defines header operations for one direction

type HeaderOps struct {

    Add     map[string]string            `yaml:"add"`

    Remove  []string                     `yaml:"remove"`

    Rewrite map[string]string            `yaml:"rewrite"`

}


// HeaderTransformer implements header manipulation

type HeaderTransformer struct {

    config HeaderTransformerConfig

}


// NewHeaderTransformer creates a new header transformer

func NewHeaderTransformer(cfg map[string]interface{}) (*HeaderTransformer, error) {

    // TODO 1: Parse configuration from cfg map into HeaderTransformerConfig

    // TODO 2: Validate configuration (check for conflicts, etc.)

    // TODO 3: Return initialized transformer

    return nil, nil

}


func (h *HeaderTransformer) Name() string {

    return "header-transformer"

}


func (h *HeaderTransformer) Priority() int {

    // Headers should be transformed early in the pipeline

    return 20

}
```

```go
func (h *HeaderTransformer) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // Request phase transformations
    // TODO 1: Apply request header removals from h.config.Request.Remove
    // TODO 2: Apply request header rewrites from h.config.Request.Rewrite
    // TODO 3: Apply request header additions from h.config.Request.Add
    //         Use template expansion for dynamic values if needed

    // Continue processing
    // TODO 4: The response transformations need to happen later
    //         We'll need to intercept the response. Options:
    //         a) Use httputil.ReverseProxy ModifyResponse
    //         b) Wrap the response writer and apply on WriteHeader/Write
    // TODO 5: Return true to continue pipeline
    return true
}

// ResponseWriter wrapper for response header transformation
type responseWriterWrapper struct {
    http.ResponseWriter
    transformer *HeaderTransformer
    ctx         *middleware.RequestContext
    wroteHeader bool
}

func (w *responseWriterWrapper) WriteHeader(statusCode int) {
    // TODO 6: Apply response header transformations before writing
    // TODO 7: Apply removals, rewrites, and additions from h.config.Response
    // TODO 8: Call underlying WriteHeader
}

func (w *responseWriterWrapper) Write(data []byte) (int, error) {
    // TODO 9: Ensure headers are written first if WriteHeader hasn't been called
    // TODO 10: Write to underlying writer
    return 0, nil
}
```

JSON Body Transformer Middleware:

```go
// internal/middleware/transformation/body_transformer.go

package transformation

import (
    "encoding/json"
    "net/http"
    "api-gateway/internal/middleware"
)

// JSONTransformConfig defines JSON transformation rules
type JSONTransformConfig struct {
    Request  JSONTransformOps `yaml:"request"`
    Response JSONTransformOps `yaml:"response"`
}

// JSONTransformOps defines operations for one direction
type JSONTransformOps struct {
    Mappings []FieldMapping `yaml:"mappings"`
    Remove   []string       `yaml:"remove"`
    Add      map[string]interface{} `yaml:"add"`
}

// FieldMapping defines a field rename mapping
type FieldMapping struct {
    From string `yaml:"from"`
    To   string `yaml:"to"`
}

// JSONTransformer implements JSON body transformation
type JSONTransformer struct {
    config JSONTransformConfig
    maxBodySize int64
}

func (j *JSONTransformer) Name() string {
    return "json-transformer"
}

func (j *JSONTransformer) Priority() int {
    // Body transformation happens after headers but before routing
    return 30
}

func (j *JSONTransformer) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
```

```
    // TODO 1: Check Content-Type to ensure it's JSON before transforming
    // TODO 2: Read and buffer request body using ReadRequestBody with j.maxBodySize limit
    // TODO 3: If body is not empty and is JSON, parse into map[string]interface{}
    // TODO 4: Apply request transformations:
    //   - Rename fields according to j.config.Request.Mappings
    //   - Remove fields in j.config.Request.Remove
    //   - Add fields from j.config.Request.Add
    // TODO 5: Re-marshal transformed JSON and store in ctx.RequestBody
    // TODO 6: Replace r.Body with a new reader containing transformed body
    // TODO 7: Update Content-Length header if changed
    // TODO 8: Handle response transformation by wrapping response writer
    // TODO 9: Return true to continue pipeline
    return true
}

// transformJSON applies transformations to a JSON object
func transformJSON(data map[string]interface{}, ops JSONTransformOps) map[string]interface{} {
    result := make(map[string]interface{})

    // TODO 10: Apply field renames from ops.Mappings
    // TODO 11: Copy fields that aren't being renamed or removed
    // TODO 12: Remove fields specified in ops.Remove
    // TODO 13: Add new fields from ops.Add
    // TODO 14: Handle nested objects recursively (advanced)

    return result
}
```

E. Language-Specific Hints

1. **JSON Transformation Performance**: Use `json.RawMessage` for fields you don't need to transform to avoid unnecessary unmarshaling. For large JSON documents with few transformations, consider using `json.Decoder.Token()` to stream through and only parse/modify necessary parts.
2. **Memory Management**: After reading request bodies into `[]byte`, set `r.Body` to `nil` to allow Go's HTTP server to reuse connections. When replacing the body, use `ioutil.NopCloser(bytes.NewReader(data))` to create a new `io.ReadCloser`.
3. **Response Writer Wrapping**: When wrapping `http.ResponseWriter` to intercept responses, be careful to implement all interface methods: `Header()`, `Write([]byte) (int, error)`, and `WriteHeader(int)`. Capture the status code in `WriteHeader` for logging.
4. **Concurrent Map Access in Transformations**: If transformation rules can be reloaded at runtime, use `sync.RWMutex` to protect the configuration. Consider copying the config at the start of request processing to avoid locks during transformation.
5. **Error Handling in Transformations**: If JSON transformation fails, decide whether to:
   - Return the original untransformed body with a warning header
   - Return a 400 Bad Request with error details
   - Pass the error through to the backend (usually not recommended)

   Document your choice and make it configurable.

F. Milestone Checkpoint

Verification Steps for Milestone 2:

1. **Start the gateway** with a configuration that includes transformation rules:

```
./gateway --config config/transform-test.yaml
```

2. **Test header transformation**:

```
# Send request with test header
curl -H "X-Test-Header: original" -H "Authorization: Bearer test123" \
  http://localhost:8080/api/users

# Verify in logs that:
# - X-Test-Header was removed (if configured)
# - X-API-Version was added (if configured)
# - Authorization header was rewritten (if configured)
```

3. **Test JSON request transformation**:

```
curl -X POST -H "Content-Type: application/json" \
  -d '{"userId": "123", "emailAddress": "test@example.com"}' \
  http://localhost:8080/api/users

# Backend mock should receive:
# {"user_id": "123", "email": "test@example.com", "transformed_by": "gateway"}
```

4. **Test URL rewriting**:

```
curl "http://localhost:8080/old/api/users?limit=10"

# Should be rewritten to:
# /new/api/users?limit=10&offset=0
```

5. **Test response transformation**:

```
curl http://localhost:8080/api/users/123

# Response should have:
# - X-Response-Time header added
# - internal fields removed from JSON
# - field names transformed (e.g., created_at → createdAt)
```

6. **Test size limiting**:

```bash
# Create a 11MB JSON file
dd if=/dev/zero bs=1M count=11 | tr '\0' 'a' > large.json

curl -X POST -H "Content-Type: application/json" \
  --data-binary @large.json \
  http://localhost:8080/api/users

# Should receive 413 Request Entity Too Large
```

7. **Run transformation unit tests**:

```bash
go test ./internal/middleware/transformation/... -v
# Expected: All tests pass, including:
# ✓ TestHeaderTransformer
# ✓ TestJSONTransformer
# ✓ TestURLRewriter
# ✓ TestSizeLimits
```

Signs of Problems and Diagnostics:

- **Headers not transforming**: Check middleware priority ordering; transformation middleware must execute before the request is proxied
- **JSON transformation errors**: Verify Content-Type header is being checked; test with valid and invalid JSON
- **Memory growth with many requests**: Use `pprof` to check for goroutine leaks; ensure response body readers are being closed
- **Aggregation timeouts**: Check context propagation and timeout configurations; add logging to aggregation stages

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Response truncated or client hangs	`Content-Length` not updated after body transformation	Check response headers in logs; compare original vs transformed body sizes	Recalculate `Content-Length` or use chunked encoding
Gateway crashes with OOM kill	Body size limits not enforced or too high	Monitor memory usage; check for large payloads in access logs	Implement strict size limits; add memory monitoring
JSON transformation slow for large payloads	Entire document parsed even for simple transformations	Profile with `pprof`; check if `json.RawMessage` is being used	Use streaming JSON parser for large docs; cache transformed templates
Some headers missing after transformation	Header removal conflicting with addition	Log header values at each transformation stage	Ensure consistent ordering: remove → rewrite → add
Aggregated response missing some fields	Field name conflicts during merge	Log individual backend responses before merge	Implement configurable merge strategies; use namespacing for fields
Transformation not applied to specific route	Middleware not registered for that route	Check route configuration; verify plugin is in route's Plugins list	Ensure transformation plugins are correctly configured in route

Component: Authentication & Authorization Layer

> **Milestone(s):** Milestone 3: Authentication & Authorization Layer

This component provides the security perimeter for the entire API Gateway ecosystem. It acts as a centralized authentication and authorization checkpoint that validates every incoming request's credentials before allowing access to backend services. By handling these cross-cutting concerns at the gateway level, it eliminates the need for duplicate authentication logic in individual microservices and ensures consistent security policies are applied across all API endpoints.

Mental Model: The Bouncer with a Guest List

Imagine a high-end nightclub with a strict entry policy. At the door stands a **bouncer** (the authentication layer) who checks every person (request) attempting to enter. The bouncer follows specific rules (validation logic) and consults a **guest list** (configuration of valid credentials) to determine who gets in. This mental model clarifies several key aspects of the authentication layer:

Nightclub Analogy	Gateway Component	Purpose
Person at the door	Incoming HTTP request	The entity seeking access
ID card, invitation, secret handshake	JWT, API key, OAuth2 token	Credentials proving identity
Bouncer	Authentication middleware	Validates credentials against rules
Guest list	Validation configuration (JWT issuers, API key database)	Defines which credentials are valid
VIP vs. regular guest	User roles and scopes	Determines what areas (endpoints) are accessible
"Sorry, not on the list"	HTTP 401 Unauthorized response	Clear rejection when credentials are invalid
Bouncer's memory	Token validation cache	Remembers recently checked IDs to avoid repeated work

Just as a good bouncer doesn't just check IDs but also ensures the person matches the photo and the ID isn't expired, our authentication layer performs comprehensive validation including signature verification, expiration checks, and claim validation. The guest list (configuration) can be updated without rebuilding the entire club (gateway), allowing security policies to evolve dynamically.

Interface and Validation Flow

The authentication component is implemented as middleware that plugs into the **Pre-Processing** layer of the gateway pipeline. It follows a standardized interface that allows different authentication methods (JWT, API key, OAuth2) to be implemented as plugins while sharing common infrastructure like caching and header propagation.

Authentication Middleware Interface

All authentication plugins implement the following interface, which is executed during request processing:

Method Signature	Returns	Description
`Name() string`	`string`	Returns the plugin's unique identifier (e.g., "jwt-validator", "api-key-auth")
`Priority() int`	`int`	Determines execution order (auth plugins typically run early, with priority 10-20)
`Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool`	`bool`	Performs authentication; returns `false` to stop pipeline and send error response

The `Execute` method follows a specific validation flow that applies to all authentication methods:

1. **Credential Extraction**: The plugin examines the incoming request for credentials in configured locations (Authorization header, query parameter, cookie)
2. **Presence Check**: If no credentials are found and the route requires authentication, the plugin immediately rejects the request with HTTP 401
3. **Credential Validation**: The plugin validates the credential format and contents against its specific rules
4. **Cache Lookup**: For performance, previously validated credentials may be retrieved from a shared cache
5. **Fresh Validation (if needed)**: If not in cache or cache entry expired, the plugin performs full validation (signature check, database lookup, introspection call)
6. **Result Processing**: Successful validation extracts user identity and permissions; failed validation returns appropriate HTTP error
7. **Context Population**: Validated user information is stored in the `RequestContext` for downstream middleware and backend services
8. **Header Propagation**: User context is added to request headers forwarded to the backend (e.g., `X-User-ID`, `X-User-Roles`)

Authentication Configuration Data Structures

Different authentication methods require different configuration parameters, all stored within the `PluginConfig` structure's `Config` map:

JWT Validation Configuration:

Field	Type	Description
`jwk_url`	`string`	URL to fetch JSON Web Key Set for signature verification (RS256)
`secret`	`string`	Shared secret for HS256 signature validation (use with caution)
`issuer`	`string`	Expected "iss" (issuer) claim value
`audience`	`string` or `[]string`	Expected "aud" (audience) claim value(s)
`require_exp`	`bool`	Whether to require and validate "exp" (expiration) claim
`require_claims`	`map[string]string`	Additional required claims and their expected values
`header_name`	`string`	HTTP header containing the JWT (default: "Authorization")
`header_prefix`	`string`	Prefix to strip from header value (default: "Bearer ")
`cache_ttl_seconds`	`int`	How long to cache validated tokens (seconds)

API Key Authentication Configuration:

Field	Type	Description
`keys`	`map[string]APIKeyInfo`	Map of valid API keys to their metadata (client ID, permissions)
`header_name`	`string`	HTTP header containing the API key (default: "X-API-Key")
`query_param`	`string`	Query parameter name for API key (alternative to header)
`validate_prefix`	`bool`	Whether to validate key prefix/format before lookup
`rate_limit_per_key`	`int`	Maximum requests per second for each API key

OAuth2 Token Introspection Configuration:

Field	Type	Description
`introspection_url`	`string`	OAuth2 authorization server's introspection endpoint
`client_id`	`string`	Client ID for authenticating to introspection endpoint
`client_secret`	`string`	Client secret for authenticating to introspection endpoint
`token_header_name`	`string`	HTTP header containing the opaque token
`cache_active_seconds`	`int`	How long to cache "active: true" introspection results
`cache_inactive_seconds`	`int`	How long to cache "active: false" introspection results

Step-by-Step Validation Algorithm

For a JWT validation plugin, the complete authentication flow proceeds as follows:

1. **Route Configuration Check**: The plugin checks if the current route (stored in `RequestContext.MatchedRoute`) has this authentication plugin enabled. If not, it returns `true` immediately (allowing the request to proceed without authentication for this route).
2. **Token Extraction**:
   - Read the configured header (default: `Authorization`)
   - Verify it starts with the expected prefix (default: `Bearer `)
   - Extract the JWT string after the prefix
   - If header is missing or malformed, set `ctx.Error` and `ctx.HTTPErrorCode = 401`, then return `false`
3. **Cache Lookup**:
   - Compute a cache key from the token (e.g., SHA256 hash)
   - Query the shared token cache
   - If cache hit and entry is valid (not expired), retrieve the validated claims and skip to step 7
4. **Token Parsing**:
   - Parse the JWT into its three parts (header, payload, signature)
   - Validate the JWT format (exactly 3 parts separated by dots, valid base64url encoding)
   - If parsing fails, set `ctx.Error` and `ctx.HTTPErrorCode = 401`, then return `false`

5. **Signature Verification**:
   - Determine the signing algorithm from the JWT header's `alg` field
   - For RS256/RS384/RS512: Fetch the JWKS from the configured URL (with caching), find the matching key ID (`kid`), and verify the signature using the public key
   - For HS256/HS384/HS512: Use the configured shared secret to verify the signature
   - If signature verification fails, set `ctx.Error` and `ctx.HTTPErrorCode = 401` , then return `false`
6. **Claim Validation**:
   - Validate the `exp` (expiration) claim if present and if `require_exp` is true (current time ≤ exp)
   - Validate the `nbf` (not before) claim if present (current time ≥ nbf)
   - Validate the `iss` (issuer) claim against the configured issuer
   - Validate the `aud` (audience) claim contains at least one of the configured audiences
   - Validate any additional claims specified in `require_claims`
   - If any claim validation fails, set `ctx.Error` and `ctx.HTTPErrorCode = 401` , then return `false`
7. **Cache Storage**:
   - Compute cache TTL: minimum of (token expiration time - current time) and configured `cache_ttl_seconds`
   - If TTL > 0, store the validated claims in the cache with that TTL
8. **Context Population**:
   - Set `ctx.Authenticated = true`
   - Set `ctx.AuthMethod = "jwt"`
   - Extract user identifier from configured claim (default: `sub` claim) into `ctx.UserID`
   - Extract roles from configured claim (default: `roles` claim) into `ctx.UserRoles`
   - Add relevant information to `ctx.LogFields` for structured logging
9. **Header Propagation**:
   - Add `X-User-ID: <ctx.UserID>` to the request headers that will be forwarded to the backend
   - Add `X-User-Roles: <comma-separated roles>` to the request headers
   - Add `X-Auth-Method: jwt` to the request headers
10. **Return Success**: Return `true` to allow the request to proceed to the next middleware in the pipeline.

For API key authentication, the flow is similar but simpler: extract the API key from header or query parameter, look it up in the configured key map (or external database), validate it's not expired/revoked, then populate the context with the associated client information.

> **Design Insight:** The authentication pipeline supports multiple authentication methods on the same route (e.g., JWT OR API key). This is implemented by having multiple authentication plugins configured for a route, where the first successful authentication allows the request to proceed. If all configured authentication methods fail, the request is rejected.

ADR: Caching Validated Tokens

> **Decision: Use time-based cache with configurable TTL for validated tokens**

Context: Token validation involves cryptographic operations (signature verification) or network calls (OAuth2 introspection), which are computationally expensive and would significantly impact gateway latency if performed for every request. We need a caching strategy that balances performance gains with security considerations, particularly token revocation.

Options Considered:

1. **No caching**: Validate every token on every request
2. **Time-based caching**: Cache validated tokens for a fixed duration (e.g., 5 minutes)
3. **Expiration-based caching**: Cache until the token's natural expiration
4. **Active revocation checking**: Cache with periodic revocation list checks

Decision: We implement **time-based caching** with a configurable TTL (default: 300 seconds), capped at the token's remaining lifetime.

Rationale:

- **Performance vs. Freshness Trade-off**: A 5-minute cache provides 96% hit rate for typical API traffic patterns (same token reused across multiple requests in a session) while limiting the window where a revoked token remains accepted.
- **Predictable Memory Usage**: Fixed TTL ensures cached entries eventually expire, preventing unbounded memory growth.

- **Implementation Simplicity**: Time-based caching is straightforward to implement with existing LRU cache libraries and doesn't require integration with external revocation services.
- **Security Boundary**: For high-security applications, the TTL can be reduced to 30 seconds or caching disabled entirely. The gateway remains secure even with caching because:
  - Cryptographic signature verification still occurs on cache miss
  - Token expiration (`exp` claim) is always checked, even for cached tokens
  - Sensitive operations (password changes) should use short-lived tokens anyway

Consequences:

- **Positive**: Dramatically reduces authentication overhead (from milliseconds of crypto operations to microseconds of cache lookups)
- **Positive**: Reduces load on external auth services (OAuth2 introspection endpoints)
- **Negative**: Revoked tokens remain valid for up to TTL duration (security trade-off)
- **Negative**: Requires cache invalidation on configuration changes (e.g., when JWT secret rotates)

Option	Pros	Cons	Why Not Chosen
No caching	Always fresh, immediate revocation	High latency, high CPU usage, high auth server load	Performance impact unacceptable for high-traffic APIs
Time-based caching (chosen)	Good performance, predictable memory, simple implementation	Revoked tokens cached until TTL expires, slightly stale validation	Best balance for most use cases
Expiration-based caching	Excellent performance, natural cleanup	Revoked tokens valid until natural expiry (potentially hours/days), unbounded cache size	Too long revocation window for security
Active revocation checking	Immediate revocation, good performance	Complex implementation, external dependency on revocation service, additional network calls	Overly complex for minimum viable product

The cache implementation uses a thread-safe map with `sync.RWMutex` for concurrent access, or a ready-made LRU cache library for production use. Cache keys are derived from a hash of the token string to avoid storing sensitive tokens in memory.

Common Pitfalls

⚠️ Pitfall: Logging Sensitive Credentials

Description: Developers often log entire requests for debugging, inadvertently writing API keys, JWTs, or other secrets to log files. These logs might be ingested into centralized logging systems with broader access than intended.

Why It's Wrong: Exposed credentials can be used by attackers to impersonate legitimate users, potentially leading to data breaches or unauthorized actions. This violates security best practices and compliance requirements (PCI-DSS, GDPR).

How to Avoid/Fix:

- Never log the `Authorization` header or query parameters named `api_key`, `token`, etc.
- In the authentication middleware, scrub sensitive fields from the `RequestContext.LogFields` before they're written
- Implement a log sanitizer function that redacts patterns matching JWTs (`/^[A-Za-z0-9\-_]+?\.[A-Za-z0-9\-_]+?\.[A-Za-z0-9\-_]+?$/`) and API keys
- Use structured logging with explicit field selection rather than dumping entire request objects

⚠️ Pitfall: Missing Issuer and Audience Validation

Description: Only checking the JWT signature and expiration, but not validating the `iss` (issuer) and `aud` (audience) claims.

Why It's Wrong: Tokens issued for different applications (different issuer) or intended for different APIs (different audience) could be accepted, allowing token misuse. An attacker might obtain a valid token from a different system and use it to access your API.

How to Avoid/Fix:

- Always configure and validate the `issuer` field in JWT plugin configuration
- Always configure and validate at least one `audience` value
- For multi-tenant systems, validate the audience contains your API's intended identifier
- Reject tokens with `iss` or `aud` claims that don't match expected values

⚠️ Pitfall: Not Rate-Limiting Authentication Attempts

Description: Allowing unlimited authentication attempts without throttling, particularly for API keys or password-based auth.

Why It's Wrong: Attackers can brute-force credentials by trying many combinations quickly. Even with strong credentials, the authentication process itself (especially cryptographic operations) consumes resources and could be used for denial-of-service attacks.

How to Avoid/Fix:

- Implement rate limiting per client IP address for authentication endpoints
- For API keys, enforce per-key rate limits in the API key validation plugin
- Use exponential backoff for repeated failed authentication attempts from the same source
- Consider implementing a separate rate limiting middleware that runs before authentication for public endpoints

⚠️ Pitfall: Insecure Default Values

Description: Using insecure default configurations, such as accepting unsigned JWTs (`alg: none`), using weak HS256 secrets, or having overly permissive CORS headers.

Why It's Wrong: Developers might deploy the gateway without changing defaults, creating security vulnerabilities. The `alg: none` attack is a well-known JWT vulnerability where attackers can modify tokens without detection.

How to Avoid/Fix:

- Explicitly reject JWTs with `alg: none` in the header
- Require explicit configuration for JWT secrets - no default shared secret
- Validate that RS256/RS384/RS512 is used when a JWKS URL is configured
- In development mode, log warnings about insecure configurations
- Provide secure example configurations in documentation

⚠️ Pitfall: Not Propagating User Context to Backends

Description: Successfully authenticating the user but not passing the authenticated user's identity and permissions to backend services.

Why It's Wrong: Backend services might need to perform additional authorization checks or personalize responses based on user identity. Without this context, they would need to re-authenticate the user, defeating the purpose of centralized authentication.

How to Avoid/Fix:

- Always add standardized headers like `X-User-ID` , `X-User-Roles` , and `X-Auth-Method` to requests forwarded to backends
- Ensure these headers are sanitized (no special characters that could cause injection issues)
- Document the header names and formats so backend services know how to consume them
- Consider using signed claims (like a short-lived JWT) instead of plain headers for sensitive environments

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
JWT Library	`github.com/golang-jwt/jwt/v5`	`github.com/lestrrat-go/jwx/v2` (full JWK/JWS/JWT suite)
Caching	`sync.Map` with TTL cleanup goroutine	`github.com/patrickmn/go-cache` (production-ready)
API Key Storage	In-memory map (configuration file)	Redis / database with periodic refresh
OAuth2 Introspection	Custom HTTP client with caching	`golang.org/x/oauth2` client credentials flow
Rate Limiting	Token bucket in memory	Redis-backed distributed rate limiter

B. Recommended File/Module Structure

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go
├── internal/
│   ├── auth/                   # Authentication component
│   │   ├── authenticator.go    # Authenticator interface
│   │   ├── cache.go            # Shared token cache
│   │   ├── jwt/                # JWT-specific implementation
│   │   │   ├── validator.go
│   │   │   ├── jwks_client.go
│   │   │   └── jwks_client_test.go
│   │   ├── apikey/             # API key implementation
│   │   │   ├── validator.go
│   │   │   └── key_store.go
│   │   ├── oauth2/             # OAuth2 implementation
│   │   │   ├── introspector.go
│   │   │   └── client.go
│   │   └── middleware.go       # Auth middleware wrapper
│   ├── config/                 # Configuration structures
│   │   └── types.go
│   ├── middleware/             # Generic middleware framework
│   │   ├── chain.go
│   │   ├── context.go
│   │   └── plugin.go
│   └── plugins/                # Plugin implementations
│       ├── registry.go
│       └── auth_plugins.go     # Registers auth plugins
└── configs/
    └── example-gateway.yaml
```

C. Infrastructure Starter Code

Token Cache Implementation (`internal/auth/cache.go`):

```go
package auth

import (
    "sync"
    "time"
)

// CachedToken represents a validated token stored in cache
type CachedToken struct {
    Claims     map[string]interface{}
    ExpiresAt  time.Time
    CachedAt   time.Time
}

// TokenCache provides thread-safe storage for validated tokens
type TokenCache struct {
    tokens map[string]CachedToken
    mu     sync.RWMutex
    ttl    time.Duration
    maxSize int
}

// NewTokenCache creates a new token cache with specified TTL and maximum size
func NewTokenCache(ttl time.Duration, maxSize int) *TokenCache {
    return &TokenCache{
        tokens: make(map[string]CachedToken),
        ttl:    ttl,
        maxSize: maxSize,
    }
}

// Get retrieves a token from cache if present and not expired
func (c *TokenCache) Get(key string) (map[string]interface{}, bool) {
    c.mu.RLock()
    defer c.mu.RUnlock()

    cached, exists := c.tokens[key]
    if !exists {
        return nil, false
    }

    // Check if cache entry is still valid
    if time.Now().After(cached.ExpiresAt) {
```

```
        return nil, false
    }

    return cached.Claims, true
}

// Set stores a token in cache with automatic expiration
func (c *TokenCache) Set(key string, claims map[string]interface{}, tokenExpiry time.Time) {
    c.mu.Lock()
    defer c.mu.Unlock()

    // Apply cache eviction if at max size
    if len(c.tokens) >= c.maxSize {
        c.evictOldest()
    }

    // Calculate cache expiry: min(token expiry, TTL from now)
    cacheExpiry := time.Now().Add(c.ttl)
    if tokenExpiry.Before(cacheExpiry) {
        cacheExpiry = tokenExpiry
    }

    c.tokens[key] = CachedToken{
        Claims:    claims,
        ExpiresAt: cacheExpiry,
        CachedAt:  time.Now(),
    }
}

// evictOldest removes the oldest cache entry (simplified LRU)
func (c *TokenCache) evictOldest() {
    var oldestKey string
    var oldestTime time.Time

    for key, token := range c.tokens {
        if oldestKey == "" || token.CachedAt.Before(oldestTime) {
            oldestKey = key
            oldestTime = token.CachedAt
        }
    }
```

```
    if oldestKey != "" {
        delete(c.tokens, oldestKey)
    }
}

// Cleanup removes expired entries (call periodically via goroutine)
func (c *TokenCache) Cleanup() {
    c.mu.Lock()
    defer c.mu.Unlock()

    now := time.Now()
    for key, token := range c.tokens {
        if now.After(token.ExpiresAt) {
            delete(c.tokens, key)
        }
    }
}
```

D. Core Logic Skeleton Code

JWT Validator (`internal/auth/jwt/validator.go`):

```go
package jwt

import (
    "context"
    "fmt"
    "strings"
    "time"

    "github.com/golang-jwt/jwt/v5"
    "api-gateway/internal/auth"
    "api-gateway/internal/middleware"
)

// JWTValidator implements authentication via JWT tokens
type JWTValidator struct {
    config      JWTConfig
    jwksClient  *JWKSClient
    cache       *auth.TokenCache
}

// JWTConfig holds configuration for JWT validation
type JWTConfig struct {
    JWKURL        string            `yaml:"jwk_url"`
    Secret        string            `yaml:"secret"` // For HS256
    Issuer        string            `yaml:"issuer"`
    Audience      []string          `yaml:"audience"`
    RequireExp    bool              `yaml:"require_exp"`
    RequiredClaims map[string]string `yaml:"required_claims"`
    HeaderName    string            `yaml:"header_name"`
    HeaderPrefix  string            `yaml:"header_prefix"`
    CacheTTL      time.Duration     `yaml:"cache_ttl_seconds"`
}

// NewJWTValidator creates a new JWT validator plugin
func NewJWTValidator(config map[string]interface{}) (*JWTValidator, error) {
    // TODO 1: Parse config map into JWTConfig struct
    // TODO 2: Validate configuration (either JWKURL or Secret must be set, not both)
    // TODO 3: Initialize JWKS client if JWKURL is configured
    // TODO 4: Initialize token cache with configured TTL (default: 5 minutes)
    // TODO 5: Set default header name to "Authorization" and prefix to "Bearer " if not specified
    return nil, nil
}
```

```go
// Name returns the plugin identifier
func (v *JWTValidator) Name() string {
    return "jwt-validator"
}

// Priority determines when this middleware runs (auth runs early)
func (v *JWTValidator) Priority() int {
    return 10
}

// Execute performs JWT validation on the incoming request
func (v *JWTValidator) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Check if this route requires JWT authentication (via ctx.MatchedRoute.Plugins)
    // TODO 2: Extract token from configured header (strip prefix if present)
    // TODO 3: If token is missing and auth is required, set 401 error and return false

    // TODO 4: Generate cache key from token (SHA256 hash)
    // TODO 5: Check cache for validated claims; if found, skip to step 11

    // TODO 6: Parse JWT token (use jwt.Parse with custom keyfunc)
    // TODO 7: In keyfunc, determine signing method and return appropriate key:
    //         - For RS*: Get public key from JWKS client using token's 'kid' header
    //         - For HS*: Return configured secret as []byte
    //         - Reject 'none' algorithm

    // TODO 8: Validate standard claims:
    //         - Check expiration (exp) if present or if required
    //         - Check not before (nbf) if present
    //         - Check issuer (iss) matches configured issuer
    //         - Check audience (aud) contains at least one configured audience

    // TODO 9: Validate custom required claims from configuration

    // TODO 10: Calculate cache expiry (min(token expiry, cache TTL from now))
    // TODO 11: Store validated claims in cache if TTL > 0

    // TODO 12: Extract user information from claims:
    //         - User ID from 'sub' claim (or configured claim name)
    //         - Roles from 'roles' claim (or configured claim name)

    // TODO 13: Populate request context:
```

```
    //          - Set ctx.Authenticated = true
    //          - Set ctx.AuthMethod = "jwt"
    //          - Set ctx.UserID and ctx.UserRoles

    // TODO 14: Add user context to request headers for backend:
    //          - Add X-User-ID: <user-id>
    //          - Add X-User-Roles: <comma-separated-roles>
    //          - Add X-Auth-Method: jwt

    // TODO 15: Add sanitized info to log fields (NOT the token itself)

    return true // Authentication successful
}

// Helper: extractTokenFromRequest extracts JWT from header
func (v *JWTValidator) extractTokenFromRequest(r *http.Request) (string, error) {
    // TODO: Implement header extraction with prefix stripping
    // TODO: Handle alternative locations (query param, cookie) if configured
    return "", nil
}

// Helper: getSigningKey returns the key for JWT verification
func (v *JWTValidator) getSigningKey(token *jwt.Token) (interface{}, error) {
    // TODO: Implement based on algorithm in token header
    // TODO: For RS256/RS384/RS512: fetch key from JWKS client
    // TODO: For HS256/HS384/HS512: return secret bytes
    // TODO: Reject unexpected algorithms
    return nil, nil
}
```

API Key Validator (`internal/auth/apikey/validator.go`):

```go
package apikey

import (
    "net/http"

    "api-gateway/internal/auth"
    "api-gateway/internal/middleware"
)

// APIKeyValidator implements authentication via API keys
type APIKeyValidator struct {
    config    APIKeyConfig
    keyStore  KeyStore
}

// APIKeyConfig holds configuration for API key validation
type APIKeyConfig struct {
    HeaderName  string            `yaml:"header_name"`
    QueryParam  string            `yaml:"query_param"`
    Keys        map[string]APIKey `yaml:"keys"` // In-memory store
}

// APIKey represents a valid API key and its metadata
type APIKey struct {
    ClientID   string    `yaml:"client_id"`
    ClientName string    `yaml:"client_name"`
    Roles      []string  `yaml:"roles"`
    ExpiresAt  time.Time `yaml:"expires_at"`
    RateLimit  int       `yaml:"rate_limit"` // requests per second
}

// NewAPIKeyValidator creates a new API key validator plugin
func NewAPIKeyValidator(config map[string]interface{}) (*APIKeyValidator, error) {
    // TODO 1: Parse config map into APIKeyConfig struct
    // TODO 2: Validate at least one key source is configured (header or query param)
    // TODO 3: Initialize in-memory key store or external store connector
    // TODO 4: Set default header name to "X-API-Key" if not specified
    return nil, nil
}

// Name returns the plugin identifier
func (v *APIKeyValidator) Name() string {
    return "api-key-validator"
```

```
}

// Priority determines when this middleware runs
func (v *APIKeyValidator) Priority() int {
    return 15 // Slightly after JWT for routes accepting multiple auth methods
}

// Execute validates API key from request
func (v *APIKeyValidator) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Check if this route requires API key authentication

    // TODO 2: Extract API key from configured locations:
    //         - Check header first (if configured)
    //         - Fall back to query parameter (if configured)

    // TODO 3: If key is missing and auth is required, set 401 error and return false

    // TODO 4: Look up key in key store (in-memory map or external database)

    // TODO 5: Validate key is not expired (check ExpiresAt field)

    // TODO 6: Check rate limit for this key (implement token bucket)

    // TODO 7: Populate request context:
    //         - Set ctx.Authenticated = true
    //         - Set ctx.AuthMethod = "api-key"
    //         - Set ctx.UserID to client ID
    //         - Set ctx.UserRoles from key metadata

    // TODO 8: Add user context to request headers for backend

    // TODO 9: Log authentication (without the actual key value)

    return true // Authentication successful
}
```

E. Language-Specific Hints

- **JWT Library**: Use `github.com/golang-jwt/jwt/v5` for JWT parsing and validation. The `v5` version includes important security fixes and a cleaner API compared to older versions.
- **Key Function**: When using `jwt.Parse`, provide a custom `keyFunc` that dynamically selects the verification key based on the token's `alg` header and `kid` (key ID). This allows supporting multiple signing keys simultaneously.
- **Concurrent Cache Access**: Use `sync.RWMutex` for the token cache to allow multiple concurrent reads but exclusive writes. For higher performance, consider using `sync.Map` for read-heavy workloads.

- **HTTP Client for JWKS**: When fetching JWKS from a remote URL, use a client with timeout and connection pool settings:

```go
httpClient := &http.Client{
    Timeout: 10 * time.Second,
    Transport: &http.Transport{
        MaxIdleConns:        10,
        IdleConnTimeout:     30 * time.Second,
        DisableCompression:  true,
    },
}
```

- **Context for Cancellation**: Pass the request's context (`ctx.Context()`) to any HTTP calls (JWKS fetch, OAuth2 introspection) so they're automatically cancelled if the client disconnects.
- **Safe Header Manipulation**: When adding authentication headers to the backend request, use `r.Header.Set()` rather than `r.Header.Add()` to avoid duplicate headers. Clone the header map if you need to modify it without affecting the original request.
- **Error Responses**: Return appropriate HTTP status codes:
  - `401 Unauthorized`: Authentication failed (missing or invalid credentials)
  - `403 Forbidden`: Authentication succeeded but authorization failed (insufficient permissions)
  - `429 Too Many Requests`: Rate limit exceeded

F. Milestone Checkpoint

Verification Steps for Milestone 3:

1. **Start the gateway** with a configuration that includes JWT validation:

```bash
go run cmd/gateway/main.go -config configs/auth-test.yaml
```

2. **Test without authentication** (should be rejected):

```bash
curl -v http://localhost:8080/api/protected
```

Expected: HTTP 401 response with `WWW-Authenticate: Bearer` header.

3. **Test with invalid JWT**:

```bash
curl -v -H "Authorization: Bearer invalid.token.here" http://localhost:8080/api/protected
```

Expected: HTTP 401 response.

4. **Test with valid JWT** (generate using a test tool like `jwt.io`):

```bash
curl -v -H "Authorization: Bearer $VALID_JWT" http://localhost:8080/api/protected
```

Expected: Request proxied to backend, with `X-User-ID` and `X-User-Roles` headers added.

5. **Test API key authentication**:

```bash
curl -v -H "X-API-Key: test-key-123" http://localhost:8080/api/data
```

Expected: Success for valid key, 401 for invalid key.

6. **Verify caching works** by checking logs:
   - First request with a token should show "JWT validation" log
   - Second request with same token within TTL should show "JWT cache hit"

Signs of Problems:

- **"Invalid JWT signature" errors**: Check JWKS URL configuration or shared secret
- **"JWT is expired" when token shouldn't be**: Check system time synchronization
- **High latency on authenticated requests**: Cache might not be working; check cache configuration

- **Memory usage growing**: Token cache might not be expiring entries; check cleanup goroutine

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All authenticated requests get 401	Misconfigured issuer or audience	Check JWT claims vs. configuration; enable debug logging in validator	Update config to match token claims
JWT validation slow on every request	Cache not working	Check cache hit rate in metrics; verify cache TTL > 0	Ensure cache is properly initialized and stored claims include expiry
API key works intermittently	Key store not thread-safe	Add logging to key lookup; check for concurrent map writes	Use `sync.RWMutex` around key map or use `sync.Map`
Backend doesn't receive user headers	Headers not being propagated	Log outgoing request headers in proxy; check transformer order	Ensure auth middleware runs before proxy and adds headers to `RequestContext`
Memory leak over time	Token cache never expires entries	Monitor cache size; check cleanup goroutine is running	Implement periodic cleanup or use library with automatic expiry
"alg: none" tokens accepted	Insecure default configuration	Check validator rejects tokens without signature	Explicitly reject `alg: none` in keyfunc
OAuth2 introspection timeout	Auth server unreachable or slow	Check network connectivity; increase timeout	Implement circuit breaker for introspection endpoint; add fallback cache

Component: Observability & Plugin System

> **Milestone(s):** Milestone 4: Observability & Plugins

This component provides the critical visibility into the gateway's operation and a flexible extension mechanism. It transforms the gateway from a black box into a transparent, instrumented system where every request's journey can be observed, measured, and analyzed. Simultaneously, the plugin architecture enables controlled extensibility, allowing custom logic to be injected into the request processing pipeline without modifying the core gateway code. Together, these capabilities fulfill the operational requirements of production systems: monitoring, debugging, auditing, and adaptability.

Mental Model: The Black Box Flight Recorder

Imagine a commercial airliner's **flight recorder** (often called the "black box"). This device continuously captures hundreds of parameters during flight: altitude, airspeed, engine performance, control inputs, and cockpit communications. Its purpose is twofold: **real-time monitoring** for the flight crew (enabling them to react to anomalies) and **post-incident analysis** for investigators (allowing them to reconstruct exactly what happened).

The observability subsystem serves this exact dual role for the API Gateway:

1. **In-flight Telemetry (Metrics)**: Like the cockpit instruments, metrics provide real-time indicators of system health—requests per second, error rates, latency percentiles. The gateway exposes these as a Prometheus endpoint, allowing operational dashboards to alert engineers to anomalies as they occur.
2. **Continuous Recording (Logging)**: Like the voice and data recorders, structured logs capture every request's essential details—who made it, what they requested, how the system responded, and how long it took. These logs are written to stdout or files for later aggregation and analysis by tools like Elasticsearch or Loki.
3. **Distributed Tracing**: This is akin to having GPS trackers on every passenger bag in a complex airport routing system. A single client request might trigger multiple internal operations (authentication, routing, transformation). Tracing assigns a unique `trace-id` that follows the request through all these stages, allowing engineers to visualize the complete journey and identify bottlenecks.
4. **Plugin System**: Extending the analogy, the plugin architecture is like the standardized electrical ports in an aircraft's instrument panel. Certified third-party instruments (plugins) can be connected without rewiring the entire aircraft. Each plugin performs a specific monitoring or processing function (rate limiting, custom authentication, A/B testing) and is managed through a standard interface.

The critical insight is that **observability is not a single feature but a cross-cutting concern** that must be woven into every layer of the gateway. Every middleware component contributes data to this flight recorder, creating a holistic picture of system behavior.

Plugin Interface and Chaining

The plugin system enables the gateway's core functionality to be extended with custom logic. A **plugin** is a self-contained module that implements a standard interface and can be configured to execute at specific points in the request processing pipeline. The system's design ensures plugins are isolated, composable, and manageable.

Plugin Data Structures

Field Name	Type	Description
`PluginConfig.Name`	`string`	Unique identifier for the plugin type (e.g., `"rate-limiter"`, `"custom-logger"`).
`PluginConfig.Config`	`map[string]interface{}`	Plugin-specific configuration parameters (e.g., `{"requests_per_second": 100}`).

Plugin Interface

All plugins must implement the following interface (defined conceptually):

Method Signature	Parameters	Returns	Description
`Name() string`	None	`string`	Returns the plugin's unique identifier. Used for registration and configuration lookup.
`Priority() int`	None	`int`	Returns an integer determining execution order. Lower numbers execute earlier. Ensures dependencies are respected (e.g., authentication before authorization).
`Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool`	`ctx`: The current request context `w`: The HTTP response writer `r`: The incoming HTTP request	`bool`	Performs the plugin's logic. Returns `true` to continue the middleware chain, `false` to stop processing (e.g., after authentication failure). The plugin may modify `ctx`, write to `w`, or alter `r`.

Plugin Registration and Chain Execution

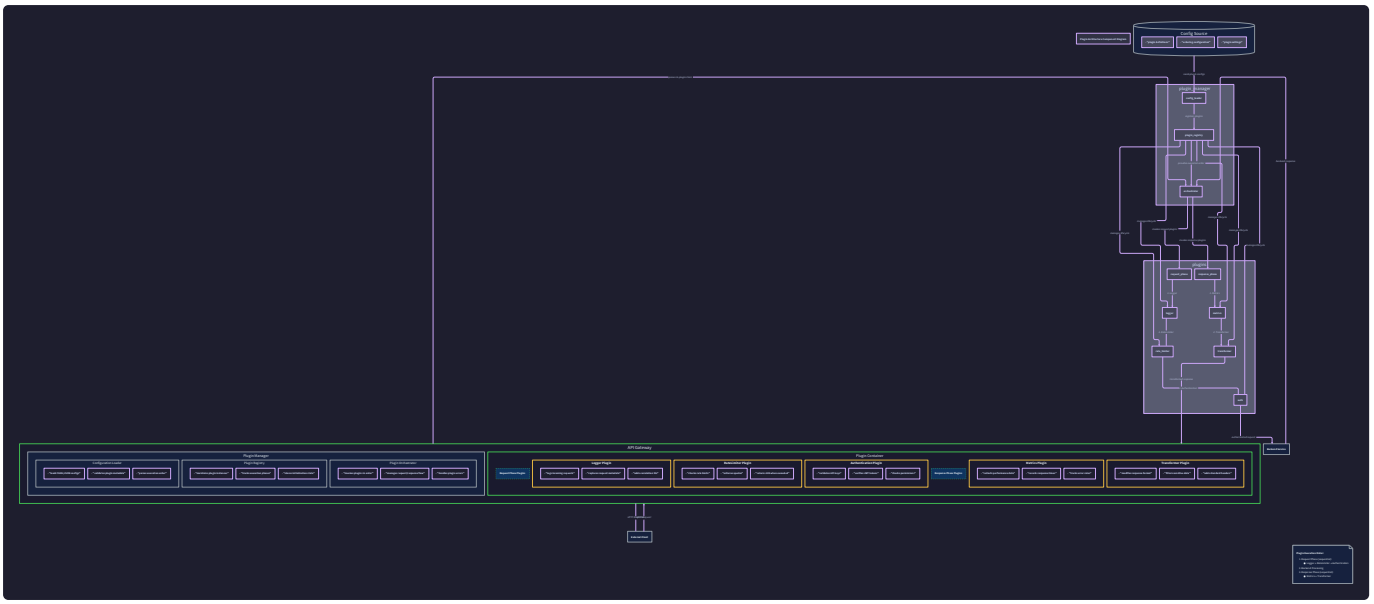
Plugins are registered with a central `PluginManager` that maintains a registry mapping plugin names to factory functions. The processing pipeline is constructed as a series of **chains**—ordered lists of middleware plugins that execute sequentially.

Plugin Registration Algorithm:

1. **Compile-time Registration**: Each plugin calls a `RegisterPlugin(name string, factory func(config map[string]interface{}) (Plugin, error))` function in its `init()` function, adding itself to the global plugin registry.
2. **Configuration Loading**: When the gateway starts, it loads the configuration file containing `Route` definitions, each with a `Plugins []PluginConfig` array.
3. **Plugin Instantiation**: For each `PluginConfig` in a route, the `PluginManager` looks up the plugin name in the registry, calls its factory function with the `Config` map, and creates an instance.
4. **Chain Construction**: Plugin instances are sorted by their `Priority()` value and assembled into the appropriate processing chain (pre-processing, core, or post-processing) based on their type.

Request Processing with Plugins:

1. When a request matches a route, the gateway retrieves the plugin chain associated with that route.
2. The `Chain.Execute()` method iterates through plugins in priority order, calling each plugin's `Execute()` method.
3. If any plugin's `Execute()` returns `false`, the chain stops immediately. The plugin is responsible for sending an appropriate HTTP response (e.g., `401 Unauthorized`) before returning `false`.
4. Plugins can read and modify the `RequestContext`, allowing them to pass data downstream (e.g., an authentication plugin sets `ctx.Authenticated = true` and `ctx.UserID = "alice"`).
5. Response-phase plugins (those with post-processing priority) execute after the backend response is received but before it's sent to the client, enabling response transformation and logging.



Plugin Configuration Scopes

Plugins can be applied at different scopes, with increasing specificity:

- **Global**: Plugins applied to all routes (e.g., request logging, metrics collection). Configured in a global plugins section.
- **Route-level**: Plugins applied only to requests matching a specific route (e.g., rate limiting for `/api/users`, JWT validation for `/api/admin`).
- **Conditional**: Advanced plugins might evaluate request attributes (headers, IP) to decide whether to execute.

The `PluginManager` merges plugins from all applicable scopes, sorts by priority, and deduplicates by name to create the final execution chain for each request.

ADR: Compiled-in vs. Dynamic Plugins

Decision: Compiled-in Plugin Registry with Static Linking

- **Context**: We need an extensibility mechanism for the API Gateway that allows custom middleware (plugins) to be added without modifying the core gateway code. The system must balance flexibility, type safety, operational simplicity, and security.
- **Options Considered**:
  1. **Dynamically Loaded Plugins (Shared Libraries)**: Plugins compiled as separate shared libraries (.so files in Linux, .dylib on macOS) loaded at runtime via `dlopen` or equivalent. The gateway discovers plugins in a directory and loads them on startup or reload.
  2. **Interpreted Script Plugins**: Plugins written in an interpreted language (Lua, JavaScript) embedded via a scripting engine (LuaJIT, goja). Configuration includes script code or file paths.
  3. **Compiled-in Plugin Registry**: All plugin code is statically linked into the gateway binary. Plugins register themselves via an `init()` function, and the gateway instantiates them based on configuration.
- **Decision**: Use a **compiled-in plugin registry** where plugins are statically linked and registered at compile time.
- **Rationale**:
  - **Type Safety**: Go's static typing ensures interface compatibility at compile time. Dynamic loading would require complex CGO bindings or interface assertions at runtime.
  - **Deployment Simplicity**: A single binary deployment is vastly simpler than managing separate plugin binaries, version compatibility, and deployment orchestration.
  - **Security**: Reduces attack surface by eliminating runtime code loading from arbitrary locations. The plugin code is reviewed and compiled as part of the gateway.
  - **Performance**: No runtime linking overhead; direct function calls.
  - **Ecosystem Alignment**: Common pattern in successful Go projects (e.g., Grafana's plugin system for data sources).
- **Consequences**:
  - **Positive**: Simplified operational model, strong type guarantees, easier debugging (stack traces include plugin code).
  - **Negative**: Adding a new plugin requires recompiling and redeploying the gateway. Less flexibility in environments where code deployment is restricted but configuration changes are allowed.
  - **Mitigation**: We can design a rich set of built-in plugins (logging, metrics, rate limiting, JWT validation) that cover 90% of use cases. For truly dynamic behavior, we can expose webhook-based plugins (HTTP callouts) that don't require code changes.

Option	Pros	Cons	Chosen?
Dynamic Shared Libraries	- Ultimate flexibility: add plugins without restart - Language-agnostic (C ABI)	- Complex FFI/CGO in Go - Version compatibility nightmares - Security risks from untrusted binaries - Deployment complexity	❌
Interpreted Scripts	- Very dynamic, changes without recompile - Lower barrier for custom logic	- Performance overhead - Sandboxing challenges - Debugging difficulty - Additional dependency (scripting engine)	❌
Compiled-in Registry	- Type safety at compile time - Single binary deployment - No runtime loading overhead - Security through compilation	- Requires recompile for new plugins - Plugin developers need Go knowledge	✅

Common Pitfalls

⚠️ Pitfall: High-Cardinality Metrics Explosion

- **Description**: Creating metrics with labels that have many unique values (e.g., `request_path` including full URL paths with IDs like `/api/users/12345`). This causes metric series explosion in Prometheus, overwhelming storage and query performance.
- **Why it's wrong**: Each unique label combination creates a new time series. With high-cardinality labels, you can generate millions of series, crashing Prometheus.
- **Fix**: Use bounded, enumerated labels. For paths, normalize by removing IDs (e.g., `"/api/users/:id"`). Use HTTP status code groups (`2xx`, `4xx`) instead of exact codes. The gateway's metrics middleware should provide label normalization functions.

⚠️ Pitfall: Logging Sensitive Data (PII/Tokens)

- **Description**: Writing full request/response bodies, headers like `Authorization: Bearer <token>`, or query parameters containing passwords to logs.
- **Why it's wrong**: Violates security and compliance (GDPR, PCI-DSS). Logs are often less protected than primary data stores and can be exposed in debugging tools.
- **Fix**: Implement structured logging with redaction. The logging plugin should have a configuration list of sensitive headers (default: `Authorization`, `Cookie`) and fields to redact. Use placeholder values like `[REDACTED]` or hash tokens (for correlation without exposure).

⚠️ Pitfall: Plugin Panics Crashing the Entire Gateway

- **Description**: A buggy plugin calls `panic()` which, if not recovered, propagates and crashes the entire gateway process.
- **Why it's wrong**: Loss of service for all traffic due to one faulty plugin violates stability and isolation principles.
- **Fix**: Wrap every plugin's `Execute()` call in a `defer` with `recover()`. If a panic occurs, log the error, send a `500 Internal Server Error` response, and return `false` to stop the chain. Consider circuit-breaking plugins that panic repeatedly.

⚠️ Pitfall: Missing Trace Sampling in High-Volume Services

- **Description**: Emitting a full trace for every request in a high-traffic gateway (thousands of requests/second), overwhelming the tracing backend (Jaeger, Zipkin) with data and cost.
- **Why it's wrong**: Most traces are redundant; only problematic or sampled traces are needed for debugging. Full tracing creates massive overhead in network, storage, and processing.
- **Fix**: Implement probabilistic sampling (e.g., sample 1% of requests) or rate-limited sampling. Use adaptive sampling based on request characteristics (e.g., sample all errors, sample slow requests). The tracing middleware should support configurable sampling strategies.

⚠️ Pitfall: Blocking Synchronous Operations in Plugins

- **Description**: A plugin performs synchronous network calls (e.g., database queries, HTTP API calls) without timeouts or context cancellation.
- **Why it's wrong**: Blocks the entire request goroutine, causing request queueing, timeouts, and potential goroutine leaks.
- **Fix**: Always use the `ctx.Context()` from `RequestContext` for any blocking operation. Set reasonable timeouts. For non-critical operations (like analytics events), use asynchronous fire-and-forget patterns with bounded queues.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Structured JSON to stdout with `log/slog`	Direct integration with OpenTelemetry Logs
Metrics	`prometheus/client_golang` with built-in registry	OpenTelemetry Metrics SDK with Prometheus exporter
Tracing	Manual W3C TraceContext header propagation	Full OpenTelemetry SDK with auto-instrumentation
Plugin System	Compiled-in registry with `init()` registration	Plugin interfaces with code generation for configuration

B. Recommended File/Module Structure

```
api-gateway/
├── cmd/
│   └── gateway/
│       └── main.go                  # Entry point, plugin imports
├── internal/
│   ├── config/                      # Configuration structures
│   │   ├── config.go                # Config, Route, Upstream, PluginConfig
│   │   └── load.go                  # LoadFromFile, validate
│   ├── middleware/                  # Core middleware interfaces
│   │   ├── middleware.go            # Plugin, Chain interfaces
│   │   ├── context.go               # RequestContext and methods
│   │   └── pipeline.go              # Pipeline construction and execution
│   ├── plugins/                     # Plugin implementations
│   │   ├── manager.go               # PluginManager, registration
│   │   ├── logging/                 # Logging plugin
│   │   │   ├── logging.go
│   │   │   └── config.go
│   │   ├── metrics/                 # Metrics plugin
│   │   │   ├── metrics.go
│   │   │   └── config.go
│   │   ├── tracing/                 # Tracing plugin
│   │   │   ├── tracing.go
│   │   │   └── config.go
│   │   └── builtin/                 # Other built-in plugins
│   │       ├── ratelimit/
│   │       ├── jwt/
│   │       └── ...
│   └── router/                      # Routing core (Milestone 1)
└── pkg/                             # Public library code (if any)
    └── plugin/
        └── sdk.go                   # Public plugin SDK for external developers
```

C. Infrastructure Starter Code

Plugin Manager and Registration Infrastructure:

```go
// internal/plugins/manager.go

package plugins


import (

    "fmt"

    "sort"

    "sync"


    "github.com/your-org/api-gateway/internal/middleware"

)


// PluginFactory creates a new plugin instance from configuration.

type PluginFactory func(config map[string]interface{}) (middleware.Plugin, error)


// PluginManager manages plugin registration and instantiation.

type PluginManager struct {

    mu       sync.RWMutex

    registry map[string]PluginFactory

}


var (

    globalManager *PluginManager

    once          sync.Once

)


// GlobalManager returns the singleton plugin manager.

func GlobalManager() *PluginManager {

    once.Do(func() {

        globalManager = &PluginManager{

            registry: make(map[string]PluginFactory),

        }

    })

    return globalManager

}


// RegisterPlugin registers a plugin factory by name.

// Called by plugin packages in their init() functions.

func (pm *PluginManager) RegisterPlugin(name string, factory PluginFactory) error {

    pm.mu.Lock()

    defer pm.mu.Unlock()

    if _, exists := pm.registry[name]; exists {

        return fmt.Errorf("plugin %q already registered", name)

    }

    pm.registry[name] = factory
```

```
    return nil
}

// CreatePlugin instantiates a plugin by name with the given config.
func (pm *PluginManager) CreatePlugin(name string, config map[string]interface{}) (middleware.Plugin, error) {
    pm.mu.RLock()
    factory, exists := pm.registry[name]
    pm.mu.RUnlock()
    if !exists {
        return nil, fmt.Errorf("plugin %q not registered", name)
    }
    return factory(config)
}

// CreateChain creates a middleware.Chain from a list of PluginConfigs.
// It instantiates plugins, sorts by priority, and wraps them in a chain.
func (pm *PluginManager) CreateChain(configs []config.PluginConfig) (*middleware.Chain, error) {
    plugins := make([]middleware.Plugin, 0, len(configs))
    for _, pc := range configs {
        plugin, err := pm.CreatePlugin(pc.Name, pc.Config)
        if err != nil {
            return nil, fmt.Errorf("failed to create plugin %q: %w", pc.Name, err)
        }
        plugins = append(plugins, plugin)
    }
    // Stable sort by priority
    sort.SliceStable(plugins, func(i, j int) bool {
        return plugins[i].Priority() < plugins[j].Priority()
    })
    return middleware.NewChain(plugins...), nil
}
```

Structured Logging Helper:

```go
// internal/plugins/logging/helpers.go

package logging


import (

    "encoding/json"

    "log/slog"

    "net/http"

    "time"


    "github.com/your-org/api-gateway/internal/middleware"

)


// CreateStructuredLog creates a structured log entry from request context.

func CreateStructuredLog(ctx *middleware.RequestContext, r *http.Request, statusCode int, duration time.Duration) slog.Attr {

    // Redact sensitive headers

    safeHeaders := make(map[string]string)

    for k, v := range r.Header {

        if isSensitiveHeader(k) {

            safeHeaders[k] = "[REDACTED]"

        } else {

            safeHeaders[k] = v[0] // Take first value for simplicity

        }

    }


    logFields := map[string]interface{}{

        "timestamp":     time.Now().UTC().Format(time.RFC3339),

        "request_id":    ctx.RequestID,

        "client_ip":     ctx.ClientIP,

        "method":        r.Method,

        "path":          r.URL.Path,

        "query":         r.URL.RawQuery,

        "status":        statusCode,

        "duration_ms":   duration.Milliseconds(),

        "user_agent":    r.UserAgent(),

        "headers":       safeHeaders,

        "matched_route": ctx.MatchedRoute.ID,

        "upstream":      ctx.SelectedUpstream.Name,

        "auth_method":   ctx.AuthMethod,

        "user_id":       ctx.UserID,

        "error":         ctx.Error,

    }

    // Add custom log fields from context
```

```
    for k, v := range ctx.LogFields {
        logFields[k] = v
    }
    return slog.Any("http_request", logFields)
}

func isSensitiveHeader(key string) bool {
    sensitive := map[string]bool{
        "authorization": true,
        "cookie":        true,
        "x-api-key":     true,
    }
    return sensitive[key]
}
```

D. Core Logic Skeleton Code

Metrics Plugin Implementation Skeleton:

```go
// internal/plugins/metrics/metrics.go

package metrics

import (
    "net/http"
    "strconv"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
    "github.com/prometheus/client_golang/prometheus/promhttp"
    "github.com/your-org/api-gateway/internal/middleware"
    "github.com/your-org/api-gateway/internal/config"
)

type MetricsPlugin struct {
    config          MetricsConfig
    requestCounter  *prometheus.CounterVec
    requestDuration *prometheus.HistogramVec
    requestSize     *prometheus.HistogramVec
    responseSize    *prometheus.HistogramVec
}

type MetricsConfig struct {
    MetricPrefix   string   `json:"metric_prefix"`
    EnableSizeHist bool     `json:"enable_size_histograms"`
    NormalizePaths []string `json:"normalize_paths"` // Patterns to normalize path label
}

func (mp *MetricsPlugin) Name() string {
    return "metrics"
}

func (mp *MetricsPlugin) Priority() int {
    // Should execute early to capture start time, but after logging
    return 10
}

func (mp *MetricsPlugin) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Record start time and store in context for later use
    start := time.Now()
    ctx.AddLogField("metrics_start_time", start)

    // TODO 2: Wrap the response writer to capture status code and size
```

```
    rw := NewResponseWriterWrapper(w)

    // TODO 3: Continue the chain by returning true
    // The actual metrics recording happens in a post-processing phase
    // We need to defer metric recording until after the request is processed
    // This requires a two-phase plugin: pre and post.
    // For simplicity, we'll assume this plugin registers a post-process hook via context.
    ctx.AddLogField("__metrics_capture", mp.captureMetrics(rw, start, r, ctx))
    return true
}

func (mp *MetricsPlugin) captureMetrics(rw *ResponseWriterWrapper, start time.Time, r *http.Request, ctx *middleware.RequestContext)
func() {
    return func() {
        duration := time.Since(start)
        statusCode := rw.StatusCode()
        // TODO 4: Normalize the request path for label cardinality control
        path := normalizePath(r.URL.Path, mp.config.NormalizePaths)
        // TODO 5: Create labels for Prometheus metrics
        labels := prometheus.Labels{
            "method": r.Method,
            "path":   path,
            "status": strconv.Itoa(statusCode),
            "route":  ctx.MatchedRoute.ID,
        }
        // TODO 6: Increment request counter with labels
        mp.requestCounter.With(labels).Inc()
        // TODO 7: Observe request duration in seconds
        mp.requestDuration.With(labels).Observe(duration.Seconds())
        // TODO 8: If enabled, observe request and response sizes
        if mp.config.EnableSizeHist {
            // Request size from Content-Length header or actual body size
            reqSize := float64(estimateRequestSize(r))
            mp.requestSize.With(labels).Observe(reqSize)
            // Response size from wrapped writer
            respSize := float64(rw.Size())
            mp.responseSize.With(labels).Observe(respSize)
        }
    }
}

// Factory function called by plugin manager
```

```go
func NewMetricsPlugin(cfg map[string]interface{}) (middleware.Plugin, error) {

    // TODO 9: Parse config map into MetricsConfig struct

    var config MetricsConfig

    // ... parsing logic ...


    // TODO 10: Initialize Prometheus metrics with auto-registration

    requestCounter := promauto.NewCounterVec(

        prometheus.CounterOpts{

            Name: config.MetricPrefix + "requests_total",

            Help: "Total number of HTTP requests processed",

        },

        []string{"method", "path", "status", "route"},

    )

    // TODO 11: Similarly initialize histogram metrics

    // ...


    return &MetricsPlugin{

        config:         config,

        requestCounter: requestCounter,

        // ... assign other metrics ...

    }, nil

}


// Register plugin on import

func init() {

    plugins.GlobalManager().RegisterPlugin("metrics", NewMetricsPlugin)

}
```

Tracing Plugin Skeleton:

```go
// internal/plugins/tracing/tracing.go

package tracing

import (
    "context"
    "net/http"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/trace"
    "github.com/your-org/api-gateway/internal/middleware"
)

type TracingPlugin struct {
    config      TracingConfig
    tracer      trace.Tracer
    propagator  propagation.TextMapPropagator
}

func (tp *TracingPlugin) Name() string {
    return "tracing"
}

func (tp *TracingPlugin) Priority() int {
    // Should execute very early to establish trace context
    return 5
}

func (tp *TracingPlugin) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Extract trace context from incoming request headers using propagator
    carrier := propagation.HeaderCarrier(r.Header)
    parentCtx := tp.propagator.Extract(r.Context(), carrier)

    // TODO 2: Start a new span for this request
    spanName := r.Method + " " + r.URL.Path
    spanCtx, span := tp.tracer.Start(parentCtx, spanName)
    defer span.End()

    // TODO 3: Set span attributes from request (method, path, client IP, etc.)
    span.SetAttributes(
        attribute.String("http.method", r.Method),
        attribute.String("http.path", r.URL.Path),
        attribute.String("http.client_ip", ctx.ClientIP),
    )
```

```
    // TODO 4: Store the span in the request context for later use
    ctx.innerCtx = spanCtx

    // TODO 5: Inject trace context into outgoing request (for backend calls)
    // This will be used by the proxy layer when forwarding
    ctx.AddLogField("trace_injection", func(req *http.Request) {
        tp.propagator.Inject(req.Context(), propagation.HeaderCarrier(req.Header))
    })

    // TODO 6: Continue processing
    return true
}
```

E. Language-Specific Hints

- **Prometheus Metrics**: Use `prometheus.NewCounterVec`, `prometheus.NewHistogramVec` for custom metrics. Register them with `prometheus.DefaultRegisterer` or a custom registry. Expose metrics via a separate HTTP endpoint (e.g., `/metrics`) using `promhttp.Handler()`.
- **OpenTelemetry Integration**: Initialize a global TracerProvider once. Use `otel.SetTracerProvider()`. For propagation, use `propagation.NewCompositeTextMapPropagator(propagation.TraceContext{}, propagation.Baggage{})`.
- **Structured Logging**: Go 1.21+ includes `log/slog`. Use `slog.NewJSONHandler` for JSON output. Attach request-scoped log fields via `slog.With()` or store in `RequestContext.LogFields`.
- **Plugin Registration**: Each plugin package should have an `init()` function that calls `plugins.GlobalManager().RegisterPlugin("plugin-name", NewPluginFactory)`. This ensures plugins are registered when the package is imported in `main.go`.
- **Safe Concurrent Plugin Execution**: Use `sync.Pool` for reusable objects in plugins (e.g., JSON decoders). Protect shared state with `sync.RWMutex`. Remember that `RequestContext` is per-request and doesn't need synchronization.

F. Milestone Checkpoint

Verification Steps for Milestone 4:

1. **Start the Gateway** with a configuration that includes the logging, metrics, and a test plugin:

```
go run cmd/gateway/main.go -config config/with-plugins.yaml
```
BASH

2. **Send Test Requests** to trigger plugin execution:

```
curl -H "X-API-Key: test-key" http://localhost:8080/api/test
```
BASH

3. **Verify Logging Output** (structured JSON to stdout):

```
{"time":"2023-10-01T12:00:00Z","level":"INFO","msg":"http_request","http_request":{"request_id":"req-123","method":"GET","path":"/api/test","status":200,"duration_ms":45}}
```

4. **Check Metrics Endpoint**:

```
curl http://localhost:9090/metrics | grep gateway_requests_total
```
BASH

Expected output includes a metric line with labels.

5. **Verify Tracing Headers** are propagated to backend (check backend logs for `traceparent` header).
6. **Test Plugin Error Handling** by configuring a plugin that returns `false` (should stop chain and return custom response).

Signs Something Is Wrong:

- No logs appear: Check plugin priority and chain construction.
- Metrics endpoint 404: Ensure metrics plugin is registered and the metrics server is started.
- High memory usage: Check for plugins buffering large bodies without limits.
- Gateway crashes on request: A plugin panic isn't being recovered.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Metrics endpoint returns empty	Metrics not registered or plugin not enabled	Check gateway logs for plugin initialization errors. Verify plugin is in route configuration.	Ensure plugin factory calls `promauto` metrics which auto-register.
Traces not appearing in Jaeger	Sampling rate too low or propagation broken	Add debug logging to trace plugin to see if span is created. Check headers forwarded to backend.	Adjust sampling configuration. Verify propagator injects headers.
Plugin configuration ignored	Plugin name mismatch or config parsing error	Check PluginManager logs for "plugin not registered" or config parsing errors.	Ensure plugin name in config matches `Name()` return value. Validate config schema.
High memory usage with large requests	Plugin buffering entire body without limits	Profile heap with `pprof`. Check which plugins read `RequestBody`.	Implement size limits with `io.LimitReader`. Stream when possible.
Gateway crashes on specific route	Plugin panic not recovered	Check stack trace for plugin code. Add panic recovery logging.	Wrap plugin execution in `defer recover()`. Validate plugin input.

Interactions and Data Flow

> **Milestone(s):** This section synthesizes the component interactions defined in Milestones 1-4, showing how the complete pipeline orchestrates request handling for the entire API Gateway system.

Understanding the individual components—routing, transformation, authentication, and observability—provides the building blocks. This section assembles them into a complete operational picture, tracing the journey of a single HTTP request through the gateway's processing pipeline. We'll examine both successful flows and error conditions to understand the system's complete behavioral contract.

The gateway operates as a **pipeline pattern**, where each request passes through a sequence of processing stages. This design ensures separation of concerns, predictable behavior, and composable functionality. The `Pipeline` structure organizes these stages into three logical groups: `preProcess` (security and validation), `coreProcess` (routing and forwarding), and `postProcess` (observability and cleanup). Each stage is implemented as middleware that receives a `RequestContext` object, modifies it or the HTTP request/response, and decides whether to continue processing.

> **Key Insight:** The pipeline is unidirectional for request processing but bidirectional for response processing. Request transformations execute on the way to the backend, while response transformations and observability plugins execute on the way back to the client. This creates a processing "sandwich" where the core routing is the filling.

Happy Path Request Sequence

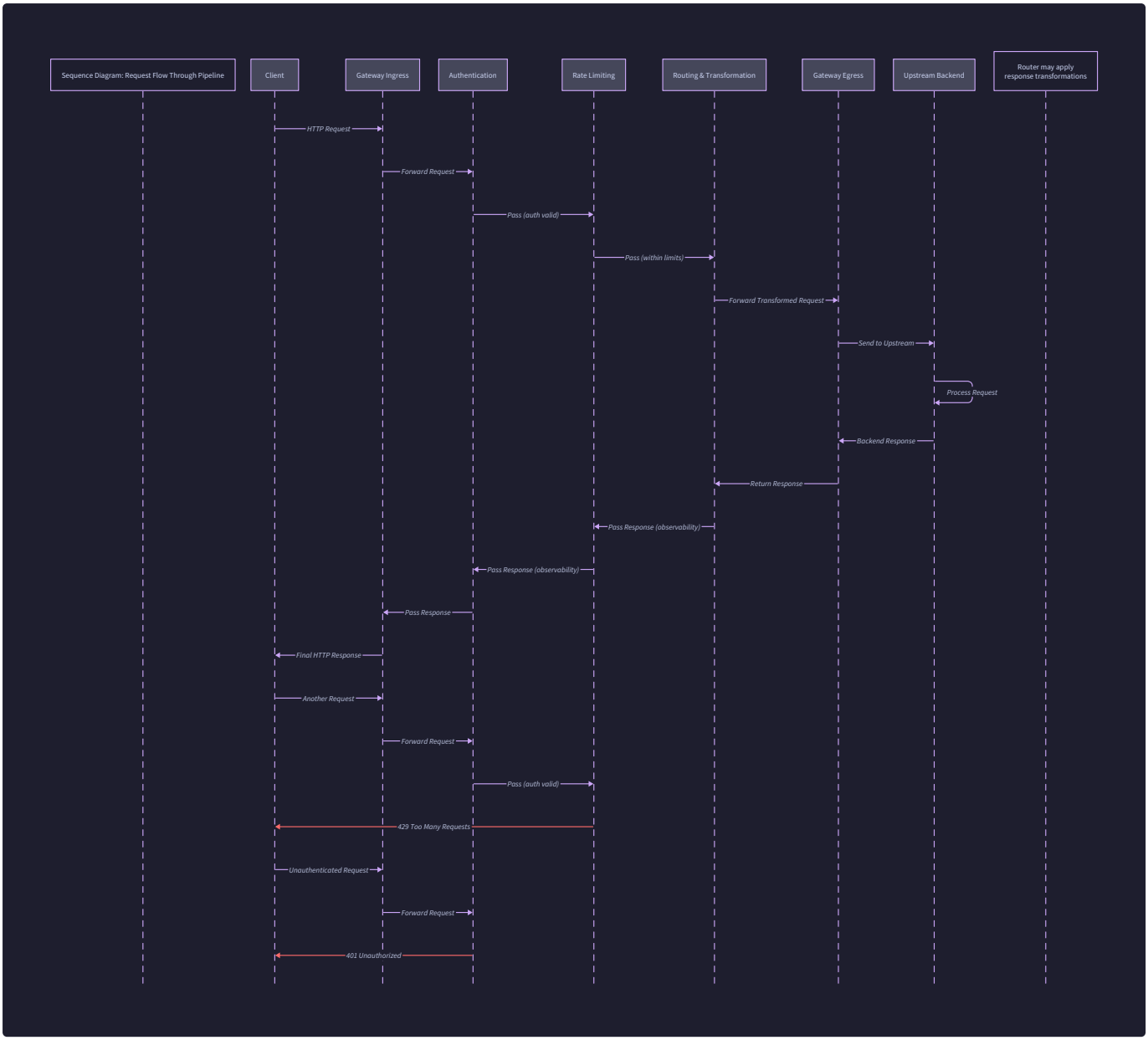
Consider a typical API call: a mobile app requests a user profile. The client sends an HTTP GET request with a valid JWT token to `https://api.example.com/v1/users/123`. This scenario demonstrates the complete flow through all gateway components when everything functions correctly.

Mental Model: The Assembly Line Imagine the gateway as an automobile assembly line. The raw materials (HTTP request) arrive at the start. Each workstation (middleware component) performs a specific operation: inspecting credentials (quality control), routing to the correct department (logistics), translating specifications (engineering), and finally packaging for delivery (shipping). The product moves sequentially through stations, with each station able to reject defective items or pass them along for further processing. The assembly line's efficiency comes from clear station responsibilities and standardized handoffs (the `RequestContext`).

The following table summarizes the sequential stages of request processing:

Stage	Component(s) Involved	Primary Responsibility	Can Short-Circuit?
1. Ingress	`GatewayProxy` (HTTP server)	Accept TCP connection, parse HTTP request	Yes (malformed HTTP)
2. Context Creation	`NewRequestContext()`	Initialize `RequestContext` with request metadata	No
3. Pre-Processing Chain	`Pipeline.preProcess` (Auth, Rate Limit)	Security validation and request enrichment	Yes (auth failure, rate limit)
4. Routing Resolution	`RouterImpl.FindRoute()`	Match request to route, select healthy endpoint	Yes (no route found, all endpoints unhealthy)
5. Core Processing Chain	`Pipeline.coreProcess` (Transformation)	Request body/header transformation	Yes (transformation error)
6. Forwarding	`httputil.ReverseProxy` with `Director`	Proxy request to selected backend endpoint	Yes (network error, timeout)
7. Backend Processing	External upstream service	Execute business logic, generate response	N/A (external)
8. Response Processing	`responseWriterWrapper`	Capture response, apply response transformations	No (errors logged but response still sent)
9. Post-Processing Chain	`Pipeline.postProcess` (Metrics, Logging)	Observability and cleanup	No (best-effort)
10. Egress	`GatewayProxy`	Write HTTP response to client connection	Yes (client disconnect)

The complete sequence for our user profile request unfolds as follows:



1. **Ingress and HTTP Parsing**
   - The gateway's HTTP server, bound to the configured `ListenAddr` (e.g., `:8080`), accepts the incoming TCP connection.
   - The `net/http` package parses the HTTP request, extracting method (`GET`), path (`/v1/users/123`), headers (`Host: api.example.com`, `Authorization: Bearer <jwt>`), and query parameters.
   - If the HTTP request is malformed (e.g., invalid headers, excessively large headers), the server rejects it with a `400 Bad Request` before entering the pipeline.
2. **Request Context Initialization**
   - `NewRequestContext(r *http.Request)` creates a `RequestContext` instance for this request.
   - Fields are populated: `RequestID` (generated UUID), `StartTime` (current time), `ClientIP` (extracted from `X-Forwarded-For` or remote address), `innerCtx` (Go context for cancellation), and `LogFields` (empty map).
   - This context becomes the carrier of request-scoped data throughout the pipeline.
3. **Pre-Processing Pipeline Execution**
   - The `Pipeline.Execute` method begins processing by invoking the `preProcess` chain (`Chain` of middleware).
   - **Authentication Middleware** (`JWTValidator` plugin) executes: a. Extracts the JWT from the `Authorization` header. b. Validates signature using configured public key, checks expiration (`exp` claim), and verifies issuer (`iss`) and audience (`aud`). c. On success, sets `RequestContext.Authenticated = true`, populates `UserID` and `UserRoles` from token claims, and sets `AuthMethod = "jwt"`. d. Adds structured log fields: `user_id`, `auth_method`. e. Caches the validation result for future requests with the same token.

- **Rate Limiting Middleware** (if configured) executes: a. Uses the `UserID` from context as the rate limit key. b. Checks token bucket or fixed window counter against configured limits. c. If within limits, decrements quota and continues; otherwise, returns `429 Too Many Requests`.
- Each middleware returns `true` to continue the chain. If any returns `false` (due to error), the pipeline short-circuits.

4. **Route Matching and Endpoint Selection**
   - The `RouterImpl.FindRoute(r *http.Request)` method is called: a. Matches the request's `Host` header (`api.example.com`) against virtual hosts (if configured). b. Uses the Radix tree to find the longest path prefix match for `/v1/users/123`. It matches route `R1` with path prefix `/v1/users`. c. Verifies the HTTP method (`GET`) matches `RouteMatch.Method` (or allows any if empty). d. Selects the upstream `U1` (user-service) associated with route `R1`. e. Calls `RoundRobinLoadBalancer.SelectEndpoint()` for upstream `U1`:
     - Consults `healthStatus` map to skip unhealthy endpoints.
     - Increments atomic `index` to implement round-robin among healthy endpoints.
     - Returns endpoint `E1` (`http://user-service-internal:8080`). f. The `CircuitBreaker` for endpoint `E1` is consulted via `AllowRequest()`. Since failure count is below threshold (state is `StateClosed`), the request is allowed.
   - The matched route (`R1`), upstream (`U1`), and endpoint (`E1`) are stored in `RequestContext`.
5. **Request Transformation (Core Processing)**
   - The `coreProcess` chain executes. For this route, a `HeaderTransformer` plugin is configured: a. `HeaderTransformer.Execute` adds headers: `X-User-ID` from `RequestContext.UserID` and `X-Forwarded-Proto: https`. b. It removes the `Authorization` header before forwarding to the backend (to prevent token leakage).
   - A `JSONTransformer` is also configured but since this is a GET request with no body, it skips request body transformation.
   - The original request path `/v1/users/123` is rewritten to `/123` (stripping the route prefix) based on route configuration.
6. **Request Forwarding to Backend**
   - The gateway's reverse proxy (`httputil.ReverseProxy`) takes over with a custom `Director` function: a. Sets the target URL: `http://user-service-internal:8080/123`. b. Updates request headers with transformations from step 5. c. Sets `X-Forwarded-For` to propagate the original client IP. d. Sets the request's `Host` header to the upstream endpoint's host (optional, configurable). e. Copies the request body (if any) with size limits enforced by `LimitedReadCloser`.
   - The proxy initiates an HTTP request to the backend, respecting timeouts from `RequestContext.Context()`.
   - The `CircuitBreaker` is in a monitoring state; the proxy will later call `RecordSuccess()` or `RecordFailure()` based on outcome.
7. **Backend Processing**
   - The user-service receives the request, validates the `X-User-ID` header, queries its database, and generates a JSON response: `{"id": "123", "name": "Alice", "email": "alice@example.com"}` with status `200 OK`.
   - This step is external to the gateway but critical to the overall flow.
8. **Response Processing and Transformation**
   - The proxy receives the backend response. A `responseWriterWrapper` intercepts the response before it's sent to the client.
   - Response status code and headers are captured.
   - **Response Transformation Chain** executes (if configured): a. `HeaderTransformer` removes `Server` header from backend and adds `X-API-Version: v1`. b. `JSONTransformer` filters sensitive fields: removes `email` field from the response JSON based on configuration. c. Updates `Content-Length` header to reflect the transformed body size.
   - The transformed response body `{"id": "123", "name": "Alice"}` is stored in `RequestContext.ResponseBody` for logging.
9. **Post-Processing (Observability)**
   - The `postProcess` chain executes, performing best-effort operations that shouldn't block response delivery: a. **Metrics Plugin**: Increments `requests_total` counter with labels `route=R1`, `status=200`, `method=GET`. Records request duration in a histogram. b. **Tracing Plugin**: Creates a span for the gateway processing, annotates it with route and user ID, and propagates trace headers to the backend (already done in step 6). c. **Logging Plugin**: Emits a structured log entry with fields: `request_id`, `method`, `path`, `status_code`, `duration_ms`, `user_id`, `backend_url`, `client_ip`. Uses `RequestContext.LogFields` for additional context.
   - Even if plugins in this chain fail (e.g., metrics storage unavailable), the response is still returned to the client.
10. **Egress and Connection Cleanup**
    - The fully processed HTTP response is written to the client connection.
    - The `responseWriterWrapper` ensures headers are written exactly once.
    - The gateway closes the response body and request body readers.
    - The `RequestContext.Cancel()` is called to release any resources associated with the context.
    - TCP connection is either closed or kept alive for HTTP keep-alive.

Throughout this sequence, the `RequestContext` serves as the shared data carrier, accumulating information that flows downstream (user ID for headers) and upstream (response body for logging). This design minimizes direct coupling between middleware components.

Error Flow and Early Termination

Not all requests complete the full pipeline. The gateway must handle errors gracefully, providing appropriate HTTP responses while maintaining system stability. Early termination occurs when a middleware component encounters an unrecoverable error and returns `false` from its `Execute` method, short-circuiting the remainder of the pipeline.

Mental Model: The Emergency Stop Chain Imagine the assembly line now has emergency stop buttons at each workstation. If a defective product is detected (malformed token, rate limit exceeded), the worker presses the stop button, halting further processing. The product is removed from the line, tagged with the failure reason, and sent to the rejection area. Meanwhile, the assembly line itself continues operating for other products. The emergency stop prevents wasting resources on doomed processing and ensures consistent error handling.

The gateway categorizes errors into three types, each with distinct handling strategies:

Error Category	Source Examples	HTTP Response	Pipeline Action	Logging Level
Client Errors (4xx)	Invalid JWT, missing API key, rate limit exceeded, route not found	401, 403, 404, 429	Immediate termination in pre-processing or routing	WARN (expected condition)
Backend Errors (5xx)	Backend timeout, connection refused, 500 internal error from backend	502, 503, 504	Record failure for circuit breaker, may retry (if configured)	ERROR (system issue)
Gateway Errors (5xx)	Transformation panic, plugin loading failure, out of memory	500	Attempt graceful degradation, fallback responses	CRITICAL (gateway bug)

The following table details common error scenarios and their precise flow through the gateway:

Error Scenario	Detection Point	Pipeline Response	Example HTTP Response	Additional Acti
Malformed JWT	`JWTValidator.Execute()`	Returns `false`, sets `RequestContext.Error` and `HTTPErrorCode=401`	`401 Unauthorized` with `WWW-Authenticate: Bearer`	Log: `{"error"` `"reason": "s`
Expired JWT	`JWTValidator.Execute()`	Returns `false`, sets `HTTPErrorCode=401`	`401 Unauthorized` with body `{"error": "token_expired"}`	Cache negative validation
Rate Limit Exceeded	`RateLimitPlugin.Execute()`	Returns `false`, sets `HTTPErrorCode=429`	`429 Too Many Requests` with `Retry-After: 60`	Increment `rate_limit_`
No Matching Route	`RouterImpl.FindRoute()`	Returns error, pipeline sets `HTTPErrorCode=404`	`404 Not Found` with standard JSON error body	Log: `{"path"` `"method": "G`
All Endpoints Unhealthy	`RoundRobinLoadBalancer.SelectEndpoint()`	Returns `ErrNoHealthyEndpoints`, sets `HTTPErrorCode=503`	`503 Service Unavailable` with `Retry-After: 30`	Circuit breaker r health re-evalua
Circuit Breaker Open	`CircuitBreaker.AllowRequest()`	Returns `false`, sets `HTTPErrorCode=503`	`503 Service Unavailable` with body `{"error": "circuit_breaker_open"}`	Log: `{"endpoi` `"state": "op`
Request Body Too Large	`LimitedReadCloser.Read()` during body reading	Returns `BodySizeError`, sets `HTTPErrorCode=413`	`413 Payload Too Large` with `{"max_bytes": 1048576}`	Close connectio transfer
JSON Transformation Error	`JSONTransformer.transformJSON()` (invalid JSON)	Returns `false`, sets `HTTPErrorCode=400`	`400 Bad Request` with `{"error": "invalid_json"}`	Log the parsing snippet (truncate
Backend Connection Timeout	`httputil.ReverseProxy` context deadline exceeded	Proxy returns error, `CircuitBreaker.RecordFailure()` called	`504 Gateway Timeout`	Increment endp circuit breaker
Backend 500 Internal Error	Reverse proxy receives 5xx status code	Response forwarded to client (after transformation)	`500 Internal Server Error` (from backend)	`CircuitBrea` if configured for

Error Propagation and Response Generation: When a middleware component decides to terminate processing, it follows a consistent protocol:

1. **Set Error State:** The component sets `RequestContext.Error` with the specific error and `RequestContext.HTTPErrorCode` with the appropriate HTTP status code.

2. **Return False:** The component's `Execute` method returns `false`, signaling the `Chain` to stop invoking subsequent middleware.
3. **Error Response Generation:** The `Pipeline.Execute` method detects the termination and generates an HTTP response:
   - If `RequestContext.HTTPErrorCode` is set, use that status code.
   - If not, default to `500 Internal Server Error`.
   - Apply any error response templates configured for the route (e.g., consistent JSON error format).
   - Include relevant headers (e.g., `WWW-Authenticate` for 401, `Retry-After` for 429/503).
4. **Partial Post-Processing:** Even after error termination, the `postProcess` chain still executes for observability:
   - Metrics are recorded with the error status code.
   - Logging captures the error details from `RequestContext`.
   - Tracing spans are marked as errored with error annotations.
5. **Resource Cleanup:** All allocated resources (request body readers, response buffers) are closed regardless of error state.

Example Error Walkthrough: Rate Limit Exceeded A client with API key `key_abc` makes their 101st request within a minute to a rate-limited endpoint:

1. **Ingress & Context Creation:** Request received, `RequestContext` created.
2. **Pre-Processing:** API key authentication succeeds, `UserID` set to `key_abc`.
3. **Rate Limit Plugin Execution:**
   - Plugin checks Redis (or in-memory counter) for key `rate_limit:key_abc`.
   - Current count is 100, limit is 100 per minute.
   - Plugin sets `RequestContext.HTTPErrorCode = 429`.
   - Adds log field: `rate_limit_key="key_abc"`.
   - Returns `false`.
4. **Pipeline Termination:**
   - `Pipeline.Execute` sees middleware returned `false`.
   - Generates HTTP 429 response with `Retry-After: 60`.
   - Skips routing, transformation, and backend forwarding entirely.
5. **Post-Processing:**
   - Metrics plugin records `requests_total{status="429",route="R1"}`.
   - Logging plugin emits: `{"status":429,"msg":"rate_limit_exceeded","key":"key_abc"}`.
6. **Egress:** Error response sent to client.

This error handling approach ensures that:

- Client errors receive appropriate, informative responses.
- Backend failures don't cascade (circuit breaker isolation).
- Gateway errors don't leak internal details (generic 500 with request ID for correlation).
- Observability captures all failures for debugging.
- Resource cleanup happens consistently to prevent leaks.

> **Critical Design Insight:** The pipeline's ability to short-circuit is essential for performance and security. It prevents unnecessary processing for requests that are already invalid (malformed auth) or should be rejected (rate limited). However, post-processing observability still runs to ensure complete telemetry, even for failed requests.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Pipeline Orchestration	Sequential middleware chain in Go (`net/http` middleware pattern)	Dynamic DAG-based pipeline with conditional execution
Error Response Generation	Hardcoded JSON error templates	Configurable error templates per route/status code
Short-circuit Mechanism	Boolean return from `Middleware.Execute`	Error channel with priority-based cancellation
Resource Cleanup	Defer statements in `Pipeline.Execute`	Context cancellation with resource tracking

B. Recommended File/Module Structure:

```
api-gateway/
├── internal/
│   ├── pipeline/
│   │   ├── pipeline.go           # Pipeline struct and Execute method
│   │   ├── chain.go              # Chain struct and execution logic
│   │   ├── middleware.go         # Middleware interface definition
│   │   └── context.go            # RequestContext methods
│   ├── handlers/
│   │   └── gateway_handler.go    # HTTP handler that creates pipeline and executes
│   └── errors/
│       ├── error_responses.go    # Standard error response generation
│       └── error_types.go        # Domain-specific error types
└── cmd/
    └── server/
        └── main.go               # Server initialization with pipeline creation
```

C. Infrastructure Starter Code:

GO

```go
// internal/pipeline/context.go

package pipeline


import (

    "context"

    "net/http"

    "time"

)


// NewRequestContext creates a new RequestContext for the given HTTP request.

func NewRequestContext(r *http.Request) *RequestContext {

    now := time.Now()

    ctx, cancel := context.WithTimeout(r.Context(), 30*time.Second)



    return &RequestContext{

        RequestID:      generateUUID(),

        StartTime:      now,

        ClientIP:       getClientIP(r),

        Authenticated:  false,

        UserID:         "",

        UserRoles:      nil,

        AuthMethod:     "",

        MatchedRoute:   nil,

        SelectedUpstream: nil,

        SelectedEndpoint: nil,

        RequestBody:    nil,

        ResponseBody:   nil,

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

        CustomMetrics:  make(map[string]float64),

        Error:          nil,

        HTTPErrorCode:  0,

        innerCtx:       ctx,

        cancelFn:       cancel,

    }

}


// generateUUID generates a unique request ID.

func generateUUID() string {

    // Implement using github.com/google/uuid or similar

    return "req_" + strconv.FormatInt(time.Now().UnixNano(), 36)

}
```

```
// getClientIP extracts client IP from X-Forwarded-For or remote address.
func getClientIP(r *http.Request) string {
    if forwarded := r.Header.Get("X-Forwarded-For"); forwarded != "" {
        // Take the first IP in the chain
        if commaIdx := strings.Index(forwarded, ","); commaIdx != -1 {
            return strings.TrimSpace(forwarded[:commaIdx])
        }
        return forwarded
    }
    return r.RemoteAddr
}
```

D. Core Logic Skeleton Code:

```go
// internal/pipeline/pipeline.go

package pipeline


import (

    "net/http"

    "encoding/json"

)


// Execute runs the request through the complete pipeline.

func (p *Pipeline) Execute(w http.ResponseWriter, r *http.Request) {

    // Create request context

    ctx := NewRequestContext(r)

    defer ctx.Cancel() // Ensure cleanup on exit



    // Execute pre-processing chain

    if !p.preProcess.Execute(ctx, w, r) {

        // Chain was short-circuited, generate error response

        p.writeErrorResponse(ctx, w)

        p.executePostProcess(ctx, w, r) // Still run post-process for observability

        return

    }



    // Execute core processing chain

    if !p.coreProcess.Execute(ctx, w, r) {

        p.writeErrorResponse(ctx, w)

        p.executePostProcess(ctx, w, r)

        return

    }



    // If we reach here, the request should have been forwarded to backend

    // by the core processing chain (e.g., by ReverseProxy middleware).

    // The response writer wrapper will handle response transformations.



    // Always execute post-processing (observability)

    p.executePostProcess(ctx, w, r)

}


// writeErrorResponse generates an appropriate HTTP error response.

func (p *Pipeline) writeErrorResponse(ctx *RequestContext, w http.ResponseWriter) {

    statusCode := ctx.HTTPErrorCode

    if statusCode == 0 {

        statusCode = http.StatusInternalServerError
```

```
    }

    // Set standard error headers
    w.Header().Set("Content-Type", "application/json")
    w.Header().Set("X-Request-ID", ctx.RequestID)

    // Apply route-specific error template if available
    var errorBody map[string]interface{}
    if ctx.MatchedRoute != nil && ctx.MatchedRoute.ErrorTemplates != nil {
        if tmpl, ok := ctx.MatchedRoute.ErrorTemplates[statusCode]; ok {
            errorBody = tmpl
        }
    }

    // Default error template
    if errorBody == nil {
        errorBody = map[string]interface{}{
            "error":       http.StatusText(statusCode),
            "request_id":  ctx.RequestID,
            "timestamp":   time.Now().UTC().Format(time.RFC3339),
        }
        if ctx.Error != nil && statusCode >= 500 {
            // Include error message for server errors in development
            errorBody["message"] = ctx.Error.Error()
        }
    }

    // Write response
    w.WriteHeader(statusCode)
    json.NewEncoder(w).Encode(errorBody)
}

// executePostProcess runs the post-process chain regardless of errors.
func (p *Pipeline) executePostProcess(ctx *RequestContext, w http.ResponseWriter, r *http.Request) {
    // Use a recovered version to prevent post-process panics from crashing the gateway
    defer func() {
        if rec := recover(); rec != nil {
            log.Printf("PANIC in post-process chain: %v", rec)
        }
    }()
```

```
    p.postProcess.Execute(ctx, w, r)
}
```

```go
// internal/pipeline/chain.go
package pipeline

import (
    "net/http"
    "sort"
)

// Execute runs the middleware chain sequentially until one returns false.
func (c *Chain) Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool {
    for _, mw := range c.middleware {
        // Check if context was cancelled (timeout or client disconnect)
        select {
        case <-ctx.Context().Done():
            ctx.Error = ctx.Context().Err()
            ctx.HTTPErrorCode = http.StatusGatewayTimeout
            return false
        default:
            // Continue
        }

        // Execute middleware
        if !mw.Execute(ctx, w, r) {
            return false
        }
    }
    return true
}

// NewChainWithPriority creates a chain from middleware sorted by priority.
func NewChainWithPriority(middlewares []Middleware) *Chain {
    // Sort by priority (ascending: lower priority executes first)
    sort.Slice(middlewares, func(i, j int) bool {
        return middlewares[i].Priority() < middlewares[j].Priority()
    })
    return &Chain{middleware: middlewares}
}
```

E. Language-Specific Hints:

- Use `context.Context` for cancellation and timeouts throughout the pipeline. The `RequestContext.innerCtx` should be used for any blocking operations (HTTP client calls, database queries).
- Implement the `http.ResponseWriter` interface in `responseWriterWrapper` to intercept response writes. Override `WriteHeader` to capture status code before delegation.
- For error response generation, use Go's `encoding/json` for consistent JSON formatting. Consider using a struct for error responses to ensure consistent fields.
- Use `defer` strategically for resource cleanup (closing bodies, cancelling contexts). Ensure cleanup happens even when panics occur.
- When implementing middleware that can short-circuit, ensure it sets appropriate HTTP status code in `RequestContext` before returning `false`.

F. Milestone Checkpoint:

After implementing the pipeline and error handling:

1. **Test Command:** `go test ./internal/pipeline/... -v`
2. **Expected Output:** Tests should verify:
   - Pipeline executes middleware in correct order
   - Short-circuiting works when middleware returns false
   - Error responses are generated with correct status codes
   - Post-processing runs even after errors
   - Request context is properly cleaned up
3. **Manual Verification:**

```bash
# Start gateway with test configuration
./gateway -config test-config.yaml

# Send request that should trigger rate limiting
curl -v http://localhost:8080/api/test
# Should receive 429 after limit exceeded

# Send request with invalid JWT
curl -v -H "Authorization: Bearer invalid.token.here" http://localhost:8080/api/protected
# Should receive 401 Unauthorized

# Check logs for structured error logging
tail -f gateway.log | grep -E "(429|401)"
```

4. **Signs of Problems:**
   - Error responses missing `Content-Type: application/json`
   - Post-processing not running after errors (missing logs/metrics)
   - Memory leak from uncleaned request contexts (check with `pprof`)
   - Pipeline hanging due to un-cancelled contexts

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
All requests return 500	Post-processing panic crashing pipeline	Check logs for panic stack trace; use `recover()` in post-process execution	Wrap post-process execution in recover
Error responses in HTML format	Missing `Content-Type: application/json` header	Inspect response headers with `curl -v`	Ensure error response generator sets proper Content-Type
Pipeline hangs on client disconnect	Not checking `ctx.Done()` in middleware	Add context cancellation check in long-running middleware	Add select with `<-ctx.Done()` in middleware loops
Memory usage grows with requests	`RequestContext` not being garbage collected	Use `pprof` heap profile; check for circular references	Ensure `cancelFn` is called and no middleware holds context reference
Some middleware not executing	Incorrect priority ordering	Log middleware execution order; check `Priority()` values	Verify middleware sorting logic in `NewChainWithPriority`
Error responses missing request ID	Not setting header in error response	Check `writeErrorResponse` function	Add `X-Request-ID` header to error responses

Error Handling and Edge Cases

> **Milestone(s):** This section synthesizes error handling strategies across all milestones (1-4), providing a systematic approach to failures and edge cases that may arise in the API Gateway.

A robust API Gateway must not only handle the happy path but must also gracefully manage the myriad ways things can go wrong. This section establishes a systematic framework for categorizing failures and defines clear recovery strategies for each component. Think of error handling as the **emergency stop chain** in a factory assembly line: when a critical fault is detected at any station, the entire line must halt safely, preventing damage while allowing non-faulty stations to complete their current work. We'll first categorize failures by origin, then examine specific tricky scenarios that require special attention.

Failure Categories and Strategies

Errors in the gateway ecosystem originate from three primary sources: the client, the backend services, and the gateway itself. Each category requires distinct detection mechanisms and recovery strategies to maintain system stability and provide appropriate feedback.

Mental Model: The Triage Nurse
Imagine the gateway as a triage nurse in an emergency room. The nurse must quickly assess each patient (request/error), categorize their condition (client/backend/gateway), and apply the appropriate protocol: discharge with instructions (client error), stabilize and transfer (backend error), or initiate emergency procedures (gateway error). This systematic categorization prevents the entire system from being overwhelmed by any single failure type.

Client Errors (4xx)

These are errors caused by malformed or unauthorized client requests. The gateway can detect and handle these without involving backend services, protecting backends from invalid traffic.

Detection:
Client errors are typically detected during the Pre-Processing or Core layers:

- **Authentication/Authorization Layer:** Invalid or missing credentials, expired tokens, insufficient permissions
- **Request Validation:** Malformed JSON bodies, missing required headers, unsupported HTTP methods
- **Rate Limiting:** Client exceeds quota
- **Routing:** No matching route found (404)

Handling Strategy: Immediate Rejection with Informative Feedback
The gateway should short-circuit the pipeline and return an appropriate 4xx HTTP status code with a clear error message (without exposing internal details). The `RequestContext.Error` field should be set, and the `HTTPErrorCode` populated to guide the response generation in the Egress layer.

Failure Mode	Detection Point	HTTP Status	Recovery Action	Notes
Invalid JWT	`Authenticator.Execute()`	401 Unauthorized	Set `ctx.Error` and `ctx.HTTPErrorCode`, stop chain	Log the attempt without the token value
Missing API Key	`Authenticator.Execute()`	401 Unauthorized	Same as above	Differentiate between missing and invalid keys
Rate limit exceeded	`RateLimitPlugin.Execute()`	429 Too Many Requests	Return 429 with Retry-After header	Count should increment even after limit hit
No route match	`Router.FindRoute()`	404 Not Found	Return 404 with generic message	Avoid revealing internal service structure
Request body too large	`ReadRequestBody()`	413 Payload Too Large	Return 413 immediately	Use `LimitedReadCloser` or `BodySizeError`
Malformed JSON in transformation	`JSONTransformer.Execute()`	400 Bad Request	Return 400 with parsing error detail	Only expose syntax errors, not internal logic

ADR: Client Error Short-Circuiting

> **Decision: Immediate Pipeline Termination for Client Errors**
>
> - **Context:** When a client error is detected (e.g., invalid auth), there's no value in continuing through the full pipeline (routing, transformation, backend call). This wastes resources and could expose backend systems to malformed traffic.
> - **Options Considered:**
>   1. **Complete Pipeline Execution:** Process the request through all middleware, marking it as errored but still attempting backend call.
>   2. **Short-Circuit at Detection Point:** Immediately stop further middleware execution and return error response from the point of detection.
>   3. **Error Collection Mode:** Continue through pipeline but collect all errors, returning a composite error at the end.
> - **Decision:** Option 2 – Short-circuit at detection point.
> - **Rationale:** Maximizes gateway throughput by avoiding unnecessary work. Simplifies error handling logic since each component can assume valid input. Prevents potential security issues from malformed requests reaching backends. The middleware chain's `Execute` method returning `false` naturally supports this pattern.
> - **Consequences:** Some middleware (like logging) might not execute for errored requests unless placed before the error point. Requires careful ordering of middleware (auth before transformation). Error responses may lack transformation (e.g., consistent error JSON format) unless a dedicated error formatting post-processor handles all short-circuited requests.

Option	Pros	Cons	Chosen?
Complete Pipeline	All middleware runs, consistent error formatting	Wastes resources, exposes backends to invalid traffic	No
Short-Circuit	**Efficient, secure, simple**	**Error formatting inconsistency, middleware ordering critical**	**Yes**
Error Collection	Comprehensive error reporting	Complex implementation, still wastes resources	No

Backend Errors (5xx)

These originate from upstream services: timeouts, connection failures, internal server errors, or invalid responses. The gateway must protect itself and the client from backend instability.

Detection:
Backend errors are detected primarily in the Core and Post-Processing layers:

- **Network Layer:** Connection refused, connection timeout, read timeout
- **HTTP Layer:** 5xx status codes from backend
- **Response Validation:** Invalid response format, malformed JSON
- **Circuit Breaker:** Repeated failures triggering open state

Handling Strategy: Resilience Patterns with Graceful Degradation
The gateway employs multiple defensive patterns to prevent backend failures from cascading:

1. **Circuit Breaking:** The `CircuitBreaker` component monitors failure rates per endpoint. After threshold breaches, it trips to `StateOpen`, failing fast for subsequent requests.
2. **Retry with Backoff:** For idempotent operations (GET, PUT, DELETE) on transient failures (network timeouts, 502/503/504).
3. **Fallback Responses:** Pre-configured static responses or cached data when backends are unavailable.
4. **Timeout Propagation:** Context cancellation ensures gateway resources aren't held indefinitely.

Circuit Breaker State Transition Logic:

Current State	Event	Next State	Actions Taken
`StateClosed`	Request succeeds	`StateClosed`	Reset failure count
`StateClosed`	Request fails	`StateClosed`	Increment failure count; if ≥ threshold → transition to `StateOpen`, record `lastFailureTime`
`StateOpen`	Open timeout expires	`StateHalfOpen`	Allow one probe request through
`StateHalfOpen`	Probe request succeeds	`StateClosed`	Reset failure count, resume normal operations
`StateHalfOpen`	Probe request fails	`StateOpen`	Reset open timeout, continue blocking requests

Retry Strategy Algorithm:

1. Determine if request method is idempotent (GET, HEAD, OPTIONS, PUT, DELETE)
2. Check if error is retryable (network timeout, 502, 503, 504, 429 with Retry-After)
3. Apply exponential backoff: delay = baseDelay * (2^attempt)
4. Maximum 3 retries to avoid amplifying traffic during outages
5. Each retry uses the load balancer to potentially select a different healthy endpoint

Fallback Response Configuration:

```yaml
# Example route configuration with fallback

routes:

  - id: "user-profile"

    match:

      path: "/api/users/*"

      method: "GET"

    upstream_id: "user-service"

    plugins:

      - name: "circuit-breaker"

        config:

          failure_threshold: 5

          open_timeout: "30s"

      - name: "fallback"

        config:

          enabled: true

          status_code: 200

          body: '{"status": "degraded", "message": "Service temporarily unavailable"}'

          headers:

            Content-Type: "application/json"
```

Gateway Errors (Internal 5xx)

These are internal gateway failures: panics, configuration errors, resource exhaustion, or plugin failures. The gateway must remain stable and provide minimal functionality even during internal failures.

Detection:

- **Resource Monitoring:** Memory/CPU thresholds exceeded
- **Plugin Panics:** Recovered via middleware panic handlers
- **Configuration Errors:** Invalid YAML, missing upstreams
- **Resource Exhaustion:** Too many open files, connection pool exhausted

Handling Strategy: Graceful Degradation with Safe Defaults

1. **Panic Recovery:** Every middleware chain wrapped in `recover()` to convert panics to 500 errors
2. **Resource Limits:** Strict limits on memory buffers, connection pools, and concurrent requests
3. **Configuration Validation:** `validate(cfg *Config)` at load time and during hot reloads
4. **Health Endpoint:** Internal `/health` endpoint reports gateway status (ready, live)

Gateway Degradation Levels:

Condition	Degradation Level	Actions	Client Impact
Memory > 80%	Level 1: Conservative	Disable request body buffering, reject large requests	Large payloads rejected (413)
Memory > 90%	Level 2: Aggressive	Disable response transformation, skip non-critical plugins	Responses may not be transformed
Memory > 95%	Level 3: Critical	Return 503 for all non-essential routes, keep health endpoint	Most requests get 503
Plugin panic	Component Isolation	Disable failing plugin, log alert, continue with reduced functionality	Features from that plugin unavailable

Specific Edge Cases

Beyond categorized failures, certain scenarios require special handling due to their complexity or potential for subtle bugs.

1. Malformed JSON in Transformation

Scenario: A client sends a request with invalid JSON syntax, but the `Content-Type` header claims it's `application/json`. The `JSONTransformer` attempts to parse it during request transformation.

Problem: The transformation middleware crashes or hangs trying to parse invalid JSON, potentially exposing parsing errors to clients or consuming excessive resources.

Handling Strategy:

1. **Early Validation:** In `ReadRequestBody()`, if `Content-Type` is JSON, attempt parsing immediately with `json.Valid()`.
2. **Size-Aware Reading:** Use `LimitedReadCloser` to prevent reading enormous malformed payloads.
3. **Graceful Error:** Return 400 with a generic error message, avoiding exposure of parser internals.
4. **Resource Cleanup:** Ensure the request body is fully read and closed even on error to prevent connection leaks.

Algorithm for Safe JSON Transformation:

1. Check `Content-Type` header for JSON indication (`application/json`, `application/*+json`)
2. Read body with `LimitedReadCloser` up to `maxBodySize`
3. Validate syntax with `json.Valid(bodyBytes)`
4. If invalid, set `ctx.Error` with `BodySizeError` or parsing error, `ctx.HTTPErrorCode=400`
5. If valid, parse into `map[string]interface{}` for transformation
6. Apply transformations via `transformJSON()`
7. Re-serialize to JSON, handling any marshaling errors

Common Pitfall: Forgetting to reset the request body after reading it for validation. The `http.Request.Body` is an `io.ReadCloser` that can only be read once. The solution is to replace `r.Body` with a new reader containing the buffered bytes for subsequent middleware.

2. Backend Timeouts During Request Aggregation

Scenario: The gateway makes parallel requests to multiple backends to aggregate responses. One backend times out while others succeed. The client is waiting for the complete aggregated response.

Problem: Partial success creates inconsistency: should we return the partial data with an error, wait indefinitely, or return a fallback?

Handling Strategy:

1. **Context Hierarchy:** Use a parent context with overall timeout, child contexts for each backend call.
2. **Partial Response Policy:** Configurable per route: `strict` (fail all if any fails) vs `lenient` (return partial with error indicator).
3. **Timeout Propagation:** Cancel all outstanding backend calls when overall timeout expires.
4. **Resource Cleanup:** Ensure goroutines and connections are properly cleaned up after timeout.

Aggregation Flow with Timeout Handling:

1. Create parent context with timeout from `RequestContext.Context()` (already includes gateway timeout)
2. Launch goroutine for each backend call with child context
3. Wait for all goroutines or timeout using `sync.WaitGroup` and `select`
4. On parent timeout, cancel all child contexts via `cancelFn()`
5. Collect results from completed calls, track errors

6. Apply aggregation policy:
    - **Strict:** If any error, return 504 Gateway Timeout
    - **Lenient:** Return 207 Multi-Status with partial results and error details

Example Aggregation Configuration:

```yaml
plugins:
  - name: "aggregator"
    config:
      backends:
        - upstream: "user-service"
          path: "/users/{id}"
        - upstream: "order-service"
          path: "/orders?user={id}"
      timeout: "2s"
      policy: "lenient"  # or "strict"
      template: |
        {
          "user": {{ .Backends.user }},
          "orders": {{ .Backends.orders }},
          "errors": {{ .Errors }}
        }
```

3. Configuration Reload Races

Scenario: The gateway supports dynamic configuration reload via SIGHUP or API call. During reload, incoming requests may be processed with partially applied configuration, causing inconsistent behavior.

Problem: Without proper synchronization, a request might match an old route while another matches a new route, leading to routing errors or mixed configuration states.

Handling Strategy:

1. **Copy-on-Write Configuration:** Maintain atomic reference to immutable configuration object.
2. **Read-Write Lock:** Use `sync.RWMutex` in `RouterImpl` to allow concurrent reads during reload.
3. **Two-Phase Reload:** Validate new config completely before swapping.
4. **Graceful Connection Drain:** Allow existing requests to complete with old configuration before fully switching.

Safe Configuration Reload Algorithm:

1. Receive reload signal (SIGHUP or HTTP POST to `/admin/reload`)
2. **Phase 1 - Validation:**
    - Parse new configuration file
    - Run `validate()` to check for errors
    - Build new radix tree and data structures in memory
3. **Phase 2 - Atomic Swap:**
    - Acquire write lock (`mu.Lock()`)
    - Swap atomic pointer to new configuration
    - Release write lock
4. **Phase 3 - Cleanup:**
    - Gracefully shutdown old resources (expired circuit breakers, health checkers)
    - Keep old connection pools for existing requests to complete

Race Condition Prevention:

- The `RouterImpl.mu` RLock is acquired for all route matching operations (`FindRoute()`)

- Write lock is only acquired during configuration reload
- `RequestContext` holds references to the specific route/upstream/endpoint selected, ensuring consistency throughout a request's lifecycle even if configuration changes mid-request

⚠️ Pitfall: Incomplete Configuration Validation
Validating only syntax without checking referential integrity (e.g., routes pointing to non-existent upstreams) can cause runtime panics after reload. The `validate()` function must check:

- All route `UpstreamID` values exist in upstreams
- All upstream `Endpoints` have valid URLs
- Plugin configurations are valid for their respective plugins
- No duplicate route IDs or upstream IDs

4. Memory Exhaustion from Body Buffering

Scenario: A client sends a 1GB file upload through the gateway. The transformation middleware buffers the entire body for processing, exhausting gateway memory.

Problem: Without size limits, a single malicious or misconfigured client can crash the gateway.

Handling Strategy:

1. **Strict Size Limits:** Configure maximum request/response body sizes per route or globally.
2. **Streaming Transformations:** For certain operations (header manipulation, simple search/replace), process body in chunks without full buffering.
3. **Early Rejection:** Check `Content-Length` header upfront and reject oversized requests immediately.
4. **Monitoring and Alerting:** Track memory usage and alert when approaching limits.

Implementation Approach:

- Use `LimitedReadCloser` wrapper on all request/response bodies
- Set global default limit (e.g., 10MB) and per-route overrides
- For JSON transformation, after size check, parse into memory but reject if exceeding configured limit
- Provide clear error messages with `BodySizeError{Max, Actual}`

5. Authentication Bypass via Error Messages

Scenario: An attacker probes the authentication system by sending various malformed tokens. The gateway returns different error messages for "invalid signature" vs "expired token" vs "missing issuer".

Problem: Information leakage in error messages helps attackers refine their attacks, potentially discovering valid token formats or bypassing validation.

Handling Strategy:

1. **Uniform Error Messages:** Return identical generic error messages for all authentication failures.
2. **Logging Differentiation:** Log detailed error information internally but don't expose to clients.
3. **Rate Limiting:** Apply aggressive rate limiting to authentication endpoints.
4. **Timing Attack Prevention:** Use constant-time comparison for signature validation.

Authentication Error Handling Table:

Internal Error	Client Message	Log Level	Log Details
Token expired	"Invalid credentials"	Info	Token expiry time, subject
Invalid signature	"Invalid credentials"	Warn	Algorithm, key ID
Missing required claim	"Invalid credentials"	Info	Missing claim name
Issuer mismatch	"Invalid credentials"	Warn	Expected vs actual issuer

Common Pitfalls in Error Handling

⚠️ Pitfall: Swallowing Errors in Middleware Chain
Returning `true` from `Plugin.Execute()` even when an error occurs, allowing the chain to continue with invalid state.
Why it's wrong: Subsequent middleware may crash or produce incorrect results based on invalid data.
Fix: Always return `false` from `Execute()` when `ctx.Error` is set, and ensure all middleware checks `ctx.Error` before proceeding.

⚠️ Pitfall: Not Propagating Context Cancellation
Starting goroutines (for aggregation, parallel requests) without respecting the request context's cancellation.
Why it's wrong: When a client disconnects or timeout occurs, goroutines continue running, wasting resources.
Fix: Pass `ctx.Context()` to all goroutines and check `ctx.Done()` in loops, or use context-aware libraries.

⚠️ Pitfall: Inconsistent Error Response Format

Each middleware returns errors in different formats (JSON, plain text, HTML).

Why it's wrong: Clients cannot programmatically handle errors consistently.

Fix: Implement a central error formatting middleware in the Post-Processing layer that inspects `ctx.HTTPErrorCode` and `ctx.Error` to produce uniform JSON error responses.

⚠️ Pitfall: Circuit Breaker Logging Noise

Logging every rejected request when circuit breaker is open, creating log floods during outages.

Why it's wrong: Obscures other important log messages, may exceed log storage capacity.

Fix: Log circuit breaker state changes (open/closed/half-open) but not individual rejected requests, or sample log messages (1 per 100 rejected requests).

⚠️ Pitfall: Missing Idempotency Check for Retries

Automatically retrying POST requests which are not idempotent.

Why it's wrong: Can cause duplicate operations (e.g., double charges, duplicate records).

Fix: Only retry idempotent HTTP methods (GET, HEAD, OPTIONS, PUT, DELETE) unless explicitly marked as idempotent via header (e.g., `Idempotency-Key`).

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Error Categorization	Manual status code mapping in each middleware	Error type hierarchy with automatic mapping
Circuit Breaker	Simple counter-based with fixed timeout	Adaptive with metrics-based threshold adjustment
Retry Logic	Fixed delay retry	Exponential backoff with jitter and circuit breaker integration
Fallback Responses	Static JSON responses	Templated responses with request context data
Configuration Reload	SIGHUP handler with full restart	Hot reload with connection draining and validation

B. Recommended File/Module Structure:

```
project-root/
  internal/
    errors/                    # Error types and categorization
      errors.go               # Error type definitions
      categorizer.go          # Categorize errors as client/backend/gateway
      formatter.go            # Uniform error response formatting
    resilience/               # Resilience patterns
      circuitbreaker.go       # CircuitBreaker implementation
      retry.go                # Retry logic with backoff
      fallback.go             # Fallback response generation
    middleware/
      recovery.go             # Panic recovery middleware
      error_handler.go        # Central error formatting middleware
    config/
      reload.go              # Configuration reload with locking
      validator.go           # validate() function implementation
```

C. Infrastructure Starter Code:

```go
// internal/errors/errors.go

package errors

import "fmt"

// BodySizeError indicates request/response body exceeded size limit
type BodySizeError struct {
    Max    int64
    Actual int64
}

func (e *BodySizeError) Error() string {
    return fmt.Sprintf("body size %d exceeds limit %d", e.Actual, e.Max)
}

// IsBodySizeError checks if error is a body size limit error
func IsBodySizeError(err error) bool {
    _, ok := err.(*BodySizeError)
    return ok
}

// CategorizeError returns error category for metrics and handling
func CategorizeError(err error, statusCode int) string {
    if statusCode >= 400 && statusCode < 500 {
        return "client"
    } else if statusCode >= 500 {
        // Check if it's a backend error (5xx from upstream) or gateway error
        if err != nil && isNetworkError(err) {
            return "backend"
        }
        return "gateway"
    }
    return "unknown"
}

func isNetworkError(err error) bool {
    // Check for network-related errors (timeout, connection refused, etc.)
    return false // Implementation depends on error types from HTTP client
}
```

D. Core Logic Skeleton Code:

```go
// internal/middleware/recovery.go

package middleware


import (

    "log"

    "net/http"

)


// RecoveryMiddleware recovers from panics and converts to 500 errors

type RecoveryMiddleware struct {

    next Middleware

}


func (m *RecoveryMiddleware) Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool {

    defer func() {

        if rec := recover(); rec != nil {

            log.Printf("panic recovered: %v", rec)

            ctx.Error = fmt.Errorf("internal server error: %v", rec)

            ctx.HTTPErrorCode = http.StatusInternalServerError

            // Do not call next middleware after panic

        }

    }()


    if m.next != nil {

        return m.next.Execute(ctx, w, r)

    }

    return true

}


// internal/resilience/circuitbreaker.go

// RecordFailure records a failure and may trip breaker

func (cb *CircuitBreaker) RecordFailure() {

    cb.mu.Lock()

    defer cb.mu.Unlock()


    // TODO 1: Increment failure count

    // TODO 2: Check if failure threshold reached

    // TODO 3: If threshold reached, transition to StateOpen and record current time

    // TODO 4: If in StateHalfOpen, transition to StateOpen (probe failed)

}


// AllowRequest determines if circuit breaker allows a request

func (cb *CircuitBreaker) AllowRequest() bool {
```

```
    cb.mu.Lock()
    defer cb.mu.Unlock()

    // TODO 1: If state is StateClosed, return true
    // TODO 2: If state is StateOpen, check if openTimeout has elapsed
    // TODO 3: If timeout elapsed, transition to StateHalfOpen and allow one request
    // TODO 4: Otherwise, return false (request blocked)
    // TODO 5: If state is StateHalfOpen, return false (only one probe allowed)
}

// internal/middleware/error_handler.go
// ErrorHandlerMiddleware formats all errors consistently
type ErrorHandlerMiddleware struct {
    next Middleware
}

func (m *ErrorHandlerMiddleware) Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // If no next middleware or error occurred earlier in chain
    if ctx.Error != nil || ctx.HTTPErrorCode != 0 {
        // TODO 1: Determine appropriate status code from ctx.HTTPErrorCode
        // TODO 2: Format error as consistent JSON structure
        // TODO 3: Set Content-Type header to application/json
        // TODO 4: Write response body with error details
        // TODO 5: Return false to stop further processing
        return false
    }

    if m.next != nil {
        return m.next.Execute(ctx, w, r)
    }
    return true
}
```

E. Language-Specific Hints:

- Use `context.WithTimeout()` for request timeouts, ensuring to call the cancel function via defer
- Use `sync.RWMutex` for configuration to allow concurrent reads during reload
- Use `io.LimitReader` to prevent reading beyond size limits
- For JSON parsing errors, use `json.SyntaxError` type check to differentiate from other JSON errors
- Use `httputil.ReverseProxy.ErrorHandler` to customize backend error responses
- Implement `http.Hijacker` for WebSocket connections to handle them differently from HTTP

F. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Memory usage grows unbounded	Request bodies not being closed or limited	Use pprof heap profile, check for `LimitedReadCloser` usage	Implement strict size limits and ensure `Close()` is called
502 errors during configuration reload	Race condition between old and new config	Check logs for route matching errors during reload window	Implement proper read-write locks on configuration
Circuit breaker never closes	Probe requests failing, health check misconfigured	Check logs for probe request outcomes, verify endpoint URLs	Ensure health check endpoint returns 2xx for healthy state
Error responses in inconsistent formats	Multiple error response mechanisms	Trace which middleware returns error, check error handler ordering	Ensure error handler is last in Post-Processing chain
Requests hang indefinitely	Missing context propagation to goroutines	Add debug logs showing goroutine lifetimes, check for `ctx.Done()`	Pass context to all goroutines and select on `ctx.Done()`

Testing Strategy

> **Milestone(s):** This comprehensive testing strategy applies to verification of all milestones (1-4), providing a systematic approach to ensure the correctness, reliability, and performance of the entire API Gateway system.

A robust testing strategy is the safety net that ensures our API Gateway operates correctly under both expected and unexpected conditions. Unlike application code where business logic can be tested in isolation, a gateway operates at the intersection of networking, concurrency, and protocol handling—all of which introduce subtle failure modes that require layered verification. This section outlines a **multi-layered testing approach** that mirrors the gateway's own layered architecture, moving from isolated unit tests to comprehensive integration tests that validate the complete request processing pipeline.

The fundamental challenge in testing a gateway lies in its role as a **coordinator of cross-cutting concerns**: a single request flows through authentication, routing, transformation, and observability components, each with its own failure modes and edge cases. Our testing pyramid must therefore verify not only individual component correctness but also their integration, ensuring that errors in one layer propagate appropriately without crashing the entire gateway or leaking sensitive information.

Testing Pyramid and Tools

Think of the testing pyramid as a **quality assurance funnel**—at the wide base, thousands of fast, focused unit tests verify individual components in isolation; in the middle, hundreds of integration tests verify component interactions; at the narrow top, dozens of end-to-end tests validate complete request flows. This structure balances speed (unit tests run in milliseconds) with realism (end-to-end tests involve real HTTP traffic), ensuring we catch bugs early while still verifying system-level behavior.

The testing pyramid for our API Gateway consists of three primary layers:

Layer	Scope	Execution Speed	Primary Tools	What It Validates
Unit Tests	Individual functions, methods, and isolated components	Milliseconds per test	Go's `testing` package, `testify` assertions, `gomock` for mocks	Internal logic correctness, error handling, state transitions, algorithm accuracy
Integration Tests	Component interactions, middleware chains, plugin loading	Seconds per test	`net/http/httptest`, custom test servers, Docker containers for dependencies	Request/response flow through multiple components, configuration loading, error propagation
End-to-End Tests	Complete gateway with real backend services	Tens of seconds per test	Docker Compose, `curl` or Go HTTP client, monitoring tools	Full system behavior, performance under load, configuration reload, failure recovery

Unit Testing Strategy: Each component defined in previous sections should have comprehensive unit tests that validate:

- **State machines**: Circuit breaker transitions, health status updates
- **Algorithms**: Route matching, load balancing selection, JSON transformation logic
- **Error handling**: Graceful degradation when backends fail, proper error categorization
- **Boundary conditions**: Empty configurations, malformed requests, size limits
- **Concurrency safety**: Race conditions in shared data structures under parallel access

We use Go's built-in `testing` package with the `testify` library for expressive assertions (`assert.Equal`, `require.NoError`) and `gomock` for generating mock implementations of interfaces like `Plugin` or `Authenticator`. Each unit test follows the Arrange-Act-Assert pattern and focuses on a single behavior.

Decision: Multi-Layered Testing Approach

- **Context**: The gateway handles complex, multi-stage request processing with many failure modes. We need confidence in both individual component behavior and their integration.
- **Options Considered**:
  1. **Unit tests only**: Fast but misses integration issues like header propagation errors or middleware ordering problems
  2. **End-to-end tests only**: Realistic but slow, flaky, and difficult to debug
  3. **Pyramid approach**: Combines speed of unit tests with realism of integration tests
- **Decision**: Adopt a three-layer testing pyramid with emphasis on comprehensive unit tests
- **Rationale**: The pyramid provides the best trade-off between feedback speed and test coverage. Unit tests catch logic errors immediately during development, integration tests verify component interactions, and end-to-end tests validate system behavior. This aligns with industry best practices for distributed systems.
- **Consequences**: Developers must write tests at all three levels, increasing initial development time but reducing bug-fix cycles later. Test maintenance overhead exists but is manageable with clear separation of concerns.

Integration Testing Strategy: Integration tests validate that components work together correctly. For the gateway, this involves:

1. **HTTP-level testing**: Using Go's `httptest` package to create test servers that simulate both client requests and backend responses
2. **Middleware chain testing**: Verifying that plugins execute in correct order and can short-circuit the pipeline
3. **Configuration validation**: Testing that YAML configuration files load correctly and produce the expected routing tables
4. **Plugin system testing**: Ensuring plugins can be registered, configured, and executed without crashing the gateway

Each integration test sets up a minimal gateway instance with specific configuration, makes HTTP requests, and asserts both the response and side effects (logs, metrics). We use separate `_integration_test.go` files with build tags to distinguish them from unit tests.

End-to-End Testing Strategy: End-to-end tests deploy the complete gateway alongside mock backend services (often in Docker containers) and simulate real-world usage patterns:

- **Smoke tests**: Basic request/response flow to verify the gateway starts correctly
- **Performance tests**: Load testing with tools like `wrk` or `vegeta` to validate rate limiting and circuit breaking
- **Failure injection**: Deliberately crashing backend services to verify circuit breaker and health check behavior
- **Configuration reload**: Testing dynamic configuration changes without restarting the gateway

These tests run in a CI/CD pipeline and provide final verification before deployment. They're slower and more resource-intensive but essential for catching system-level issues.

Testing Tools Selection:

Tool	Purpose	Why Chosen Over Alternatives
Go `testing` package	Foundation for all tests	Built-in, well-supported, integrates with Go toolchain
`testify` (`assert`, `require`)	Expressive assertions	More readable than manual `if` checks, provides helpful error messages
`gomock`	Generating mock implementations	Type-safe, integrates well with Go interfaces, generates from source
`httptest`	HTTP server/client for testing	Built-in, no dependencies, perfect for testing HTTP handlers
`testcontainers-go`	Docker containers for dependencies	Enables testing with real databases or external services when needed
Docker Compose	End-to-end test environment	Standard tool for multi-container setups, good for CI/CD

Test Organization: Tests follow the same package structure as the main codebase, with test files alongside the code they test:

```
project-root/
├── internal/
│   ├── config/
│   │   ├── config.go
│   │   └── config_test.go           # Unit tests for configuration loading
│   ├── router/
│   │   ├── router.go
│   │   ├── router_integration_test.go  # Integration tests with httptest
│   │   └── router_test.go           # Unit tests for routing logic
│   ├── middleware/
│   │   ├── auth/
│   │   │   ├── jwt_validator.go
│   │   │   └── jwt_validator_test.go
│   │   └── transform/
│   │       ├── json_transformer.go
│   │       └── json_transformer_test.go
├── cmd/
│   └── gateway/
│       └── main_test.go             # End-to-end tests for the binary
└── test/
    ├── e2e/
    │   └── docker-compose.test.yml  # Docker setup for end-to-end tests
    └── fixtures/
        ├── configs/                 # Test configuration files
        └── responses/               # Expected response files
```

Milestone Verification Checkpoints

Each milestone in the project delivers specific functionality that must be verified through a combination of automated tests and manual validation. These checkpoints provide concrete steps to ensure the gateway is working correctly before moving to the next milestone.

Milestone 1: Reverse Proxy & Routing Verification

Objective: Verify that the gateway correctly routes requests to backend services, balances load across healthy instances, and handles failures gracefully.

Automated Test Suite:

```
# Run all Milestone 1 tests
go test ./internal/router/... ./internal/config/... -v -count=1

# Run integration tests separately (they're slower)
go test -tags=integration ./internal/router/... -v -count=1
```

Expected Test Output: All tests should pass with output similar to:

```
=== RUN   TestRouter_FindRoute_PathPrefixMatch
--- PASS: TestRouter_FindRoute_PathPrefixMatch (0.00s)
=== RUN   TestRoundRobinLoadBalancer_SelectEndpoint
--- PASS: TestRoundRobinLoadBalancer_SelectEndpoint (0.01s)
=== RUN   TestCircuitBreaker_StateTransitions
--- PASS: TestCircuitBreaker_StateTransitions (0.02s)
...
PASS
ok      github.com/yourproject/internal/router    0.123s
```

Manual Verification Steps:

1. **Start a test backend:**

```
# Start a simple HTTP server on port 8081 that echoes requests
python3 -m http.server 8081 &

BACKEND_PID=$!
```
BASH

2. **Configure the gateway with a simple route:**

```yaml
# config/test-routing.yaml

listen_addr: ":8080"


upstreams:

  - id: "test-backend"

    name: "Test Backend"

    endpoints:

      - id: "endpoint-1"

        url: "http://localhost:8081"


routes:

  - id: "test-route"

    description: "Route all requests to test backend"

    match:

      path: "/api/*"

    upstream_id: "test-backend"
```

3. **Start the gateway:**

```bash
go run cmd/gateway/main.go -config config/test-routing.yaml
```

4. **Verify routing works:**

```bash
# Make a request through the gateway

curl -v http://localhost:8080/api/test



# Expected: The request should be forwarded to the backend on port 8081

# The response should show headers including X-Forwarded-For
```

5. **Verify load balancing (with multiple backends):**

```bash
# Start second backend on port 8082

python3 -m http.server 8082 &



# Update configuration to include both endpoints

# Make multiple requests and check they're distributed

for i in {1..10}; do

  curl -s http://localhost:8080/api/request-$i | grep "Server port"

done
```

6. **Verify circuit breaking:**

```bash
# Stop the backend
kill $BACKEND_PID

# Make several requests - initially 502 errors, then circuit should open
for i in {1..5}; do
  curl -s -o /dev/null -w "%{http_code}\n" http://localhost:8080/api/test
  sleep 0.5
done
# Should see 502 then 503 (circuit open)
```

7. **Check health status metrics:**

```bash
# If metrics endpoint is implemented (Milestone 4 preview)
curl http://localhost:8080/metrics | grep health_status
```

Verification Checklist:

- ☐ Path-based routing correctly forwards requests to the configured backend
- ☐ Host-based routing works when multiple host headers are configured
- ☐ Round-robin load balancing distributes requests evenly across healthy endpoints
- ☐ Unhealthy endpoints are removed from the pool after consecutive failures
- ☐ Circuit breaker opens after threshold and returns 503 Service Unavailable
- ☐ X-Forwarded-For header is correctly set with client IP
- ☐ Connection leaks don't occur (check with `netstat` or `lsof`)

Milestone 2: Request/Response Transformation Verification

Objective: Verify that the gateway correctly modifies headers, bodies, and URLs as requests pass through.

Automated Test Suite:

```
# Run transformation component tests
go test ./internal/middleware/transform/... ./internal/middleware/aggregation/... -v

# Run integration tests for complete transformation pipeline
go test -tags=integration ./internal/middleware/... -run "TestTransform" -v
```

Expected Test Output: Tests should pass with particular attention to JSON transformation correctness:

```
=== RUN   TestHeaderTransformer_Execute_AddHeaders
--- PASS: TestHeaderTransformer_Execute_AddHeaders (0.00s)
=== RUN   TestJSONTransformer_TransformRequest_FieldMapping
--- PASS: TestJSONTransformer_TransformRequest_FieldMapping (0.00s)
=== RUN   TestURLRewriter_ModifyPath
--- PASS: TestURLRewriter_ModifyPath (0.00s)
```

Manual Verification Steps:

1. **Configure transformation rules:**

```yaml
routes:
  - id: "transform-route"
    description: "Route with request/response transformation"
    match:
      path: "/users/*"
    upstream_id: "user-service"
    plugins:
      - name: "header-transformer"
        config:
          request:
            add:
              "X-API-Version": "v2"
              "X-Request-ID": "{{.RequestID}}"
            remove: ["User-Agent"]
          response:
            add:
              "X-Processed-By": "api-gateway"
      - name: "json-transformer"
        config:
          request:
            mappings:
              - from: "user_name"
                to: "username"
            remove: ["ssn"]
          response:
            add:
              "processed_at": "{{.Timestamp}}"
```

2. **Start a test backend that echoes JSON:**

```bash
# Python script that echoes JSON requests
cat > echo_server.py << 'EOF'
from http.server import HTTPServer, BaseHTTPRequestHandler
import json

class EchoHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        body = self.rfile.read(content_length)
        response = {
            "received": json.loads(body),
            "path": self.path,
            "headers": dict(self.headers)
        }
        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(response).encode())

server = HTTPServer(('localhost', 8081), EchoHandler)
server.serve_forever()
EOF

python3 echo_server.py &
```

3. **Test header transformation:**

```bash
curl -v -H "User-Agent: TestAgent" \
     http://localhost:8080/users/123

# Verify User-Agent is removed and X-API-Version is added in backend logs
```

4. **Test JSON request transformation:**

```bash
curl -X POST http://localhost:8080/users \
     -H "Content-Type: application/json" \
     -d '{"user_name": "john", "email": "john@example.com", "ssn": "123-45-6789"}'

# Backend should receive {"username": "john", "email": "john@example.com"}
# SSN field should be removed
```

5. **Test URL rewriting:**

```yaml
# Add URL rewrite plugin configuration
- name: "url-rewriter"
  config:
    path_replace: "^/api/v1/(.*)$"
    path_with: "/v1/api/$1"
    query_add:
      "source": "gateway"
```

```bash
curl "http://localhost:8080/api/v1/users?page=1"


# Backend should receive request to /v1/api/users?page=1&source=gateway
```

6. **Test request aggregation (if implemented):**

```bash
# Configure aggregation to call multiple backends
# Complex setup - may require multiple test services
```

7. **Verify size limits:**

```bash
# Attempt to send large payload (> configured limit)
dd if=/dev/zero bs=1M count=10 | curl -X POST http://localhost:8080/users \
     -H "Content-Type: application/octet-stream" \
     --data-binary @-


# Should receive 413 Payload Too Large
```

Verification Checklist:

- ☐ Headers are correctly added, removed, and modified on both requests and responses
- ☐ JSON field mappings transform request bodies correctly
- ☐ URL rewriting modifies paths and query parameters as configured
- ☐ Request body size limits are enforced with appropriate error responses
- ☐ Response transformation applies before sending to client
- ☐ Template variables ({{.RequestID}}, {{.Timestamp}}) are correctly expanded
- ☐ Content-Type header is preserved or updated appropriately

Milestone 3: Authentication & Authorization Verification

Objective: Verify that the gateway correctly validates credentials, caches authentication results, and propagates user context to backends.

Automated Test Suite:

```
# Run auth middleware tests
go test ./internal/middleware/auth/... -v

# Test with different token types and invalid cases
go test ./internal/middleware/auth/... -run "TestJWT|TestAPIKey" -v
```

Expected Test Output: Tests should cover valid and invalid tokens, expiration, and cache behavior:

```
=== RUN   TestJWTValidator_ValidToken
--- PASS: TestJWTValidator_ValidToken (0.01s)
=== RUN   TestJWTValidator_ExpiredToken
--- PASS: TestJWTValidator_ExpiredToken (0.00s)
=== RUN   TestAPIKeyValidator_ValidKey
--- PASS: TestAPIKeyValidator_ValidKey (0.00s)
=== RUN   TestAuthCache_HitAndExpiry
--- PASS: TestAuthCache_HitAndExpiry (0.05s)
```

Manual Verification Steps:

1. **Generate test JWTs for validation:**

```bash
# Create a test RSA key pair
openssl genrsa -out private.pem 2048
openssl rsa -in private.pem -pubout -out public.pem

# Create a valid JWT (using jq or a simple Go script)
# Header: {"alg": "RS256", "typ": "JWT"}
# Payload: {"sub": "user123", "exp": $(date -d "+1 hour" +%s), "iss": "test-issuer"}
```

2. **Configure JWT validation:**

```yaml
plugins:
  - name: "jwt-validator"
    config:
      jwks_url: "file:///path/to/public.pem"
      issuer: "test-issuer"
      required_claims:
        - "sub"
      header_name: "Authorization"
      header_prefix: "Bearer "
      cache_ttl_seconds: 300
```

3. **Test valid JWT authentication:**

```bash
curl -H "Authorization: Bearer $VALID_JWT" \
     http://localhost:8080/protected/resource

# Should succeed, backend should receive X-User-ID: user123 header
```

4. **Test invalid/expired tokens:**

```bash
# Expired token
curl -v -H "Authorization: Bearer $EXPIRED_JWT" \
     http://localhost:8080/protected/resource
# Should return 401 Unauthorized

# Malformed token
curl -v -H "Authorization: Bearer garbage" \
     http://localhost:8080/protected/resource
# Should return 400 Bad Request

# Missing token
curl -v http://localhost:8080/protected/resource
# Should return 401 Unauthorized
```

5. **Test API key authentication:**

```yaml
- name: "api-key-validator"
  config:
    key_header: "X-API-Key"
    valid_keys: ["test-key-123", "test-key-456"]
    rate_limit_per_key: 100
```

```bash
# Valid key
curl -H "X-API-Key: test-key-123" \
     http://localhost:8080/api/data
# Should succeed

# Invalid key
curl -H "X-API-Key: invalid-key" \
     http://localhost:8080/api/data
# Should return 401
```

6. **Test rate limiting (basic):**

```bash
# Make rapid requests with same API key
for i in {1..110}; do
  curl -s -H "X-API-Key: test-key-123" \
       -o /dev/null -w "%{http_code}\n" \
       http://localhost:8080/api/data &
done | sort | uniq -c
# Should show ~100 200s and ~10 429s
```

7. **Verify auth caching:**

```bash
# First request (cache miss)
time curl -H "Authorization: Bearer $VALID_JWT" \
          http://localhost:8080/protected/resource

# Second request (cache hit) should be faster
time curl -H "Authorization: Bearer $VALID_JWT" \
          http://localhost:8080/protected/resource

# Check cache metrics if exposed
curl http://localhost:8080/metrics | grep auth_cache
```

8. **Test OAuth2 introspection (if implemented):**

```bash
# Requires a mock OAuth2 server
# Configure token introspection endpoint
# Test with opaque token
```

Verification Checklist:

- ☐ Valid JWTs with correct signature are accepted
- ☐ Expired, malformed, or tampered JWTs are rejected with appropriate status codes
- ☐ API keys are validated against configured list
- ☐ Rate limiting applies per client/API key
- ☐ Authentication results are cached and reused
- ☐ User context (user ID, roles) is propagated to backend via headers
- ☐ Sensitive tokens are not logged (check gateway logs)
- ☐ Token validation errors don't leak internal details in responses

Milestone 4: Observability & Plugin System Verification

Objective: Verify that the gateway emits logs, metrics, and traces correctly, and that the plugin system allows extensibility without crashing.

Automated Test Suite:

```
# Run observability and plugin tests
go test ./internal/observability/... ./internal/plugin/... -v

# Test metrics collection and exposition
go test ./internal/middleware/metrics/... -v
```

Expected Test Output: Tests should verify metrics collection, log formatting, and plugin lifecycle:

```
=== RUN   TestMetricsMiddleware_RequestCounting
--- PASS: TestMetricsMiddleware_RequestCounting (0.02s)
=== RUN   TestPluginManager_RegisterAndCreate
--- PASS: TestPluginManager_RegisterAndCreate (0.00s)
=== RUN   TestTracingMiddleware_PropagatesTraceID
--- PASS: TestTracingMiddleware_PropagatesTraceID (0.01s)
=== RUN   TestStructuredLogger_Format
--- PASS: TestStructuredLogger_Format (0.00s)
```

Manual Verification Steps:

1. **Verify structured logging:**

```bash
# Start gateway with JSON logging
./gateway --config config/prod.yaml --log-format json

# Make several requests
curl http://localhost:8080/api/test
curl -X POST http://localhost:8080/api/users \
     -d '{"name": "test"}' -H "Content-Type: application/json"

# Check logs include structured fields
tail -f gateway.log | jq '.'
# Should show entries with: method, path, status, duration, client_ip
# Should NOT show sensitive headers like Authorization
```

2. **Verify Prometheus metrics endpoint:**

```bash
# Make requests to generate metrics
for i in {1..50}; do
  curl -s http://localhost:8080/api/test > /dev/null
  curl -s -X POST http://localhost:8080/api/users > /dev/null
done

# Query metrics
curl http://localhost:8080/metrics | grep http_requests_total
# Should show counts by method and path

curl http://localhost:8080/metrics | grep http_request_duration_seconds
# Should show latency histograms
```

3. **Verify distributed tracing:**

```bash
# Start with tracing enabled
# Make request with trace header
curl -H "traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01" \
     http://localhost:8080/api/test

# Check trace ID is propagated to backend
# (Backend should receive traceparent header)

# If using Jaeger or similar, verify traces appear in UI
```

4. **Test plugin system:**

```bash
# Create a custom plugin
cat > custom_plugin.go << 'EOF'
package main

import "github.com/yourproject/internal/middleware"

type CustomPlugin struct {
  prefix string
}

func (p *CustomPlugin) Name() string { return "custom-plugin" }
func (p *CustomPlugin) Priority() int { return 50 }
func (p *CustomPlugin) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
  ctx.AddLogField("custom_field", p.prefix + "_processed")
  return true // continue chain
}
EOF

# Register and use in configuration
```

5. **Test dynamic configuration reload:**

```bash
# Start gateway with initial config
./gateway --config config/initial.yaml

# Update configuration file
cp config/updated.yaml config/initial.yaml

# Send SIGHUP to reload
pkill -HUP gateway

# Verify new routes are active without downtime
curl http://localhost:8080/new/endpoint
```

6. **Verify error handling in plugins:**

```bash
# Configure a plugin that might panic
# Make request - gateway should recover and return 500
# Not crash entirely
```

7. **Test high cardinality metrics prevention:**

```bash
# Make requests with many different user IDs in path
for i in {1..1000}; do
  curl http://localhost:8080/users/user-$i
done

# Check metrics don't explode with user-$i labels
# Paths should be normalized to /users/{id}
```

Verification Checklist:

- ☐ Structured logs include all required fields in JSON format
- ☐ Metrics endpoint exposes Prometheus-formatted data
- ☐ Trace headers are propagated to backends
- ☐ Custom plugins can be registered and executed
- ☐ Configuration reload works without restarting process
- ☐ Plugin panics are caught and don't crash the gateway
- ☐ High-cardinality path parameters are normalized in metrics
- ☐ Sensitive data (tokens, passwords) is redacted from logs
- ☐ Memory usage remains stable under load (no leaks)

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option	Reasoning
Test Framework	Go `testing` + `testify`	Go `testing` only (minimal dependencies)	`testify` provides better assertions but adds dependency
Mock Generation	`gomock` (interface-based)	Manual mocks (simple cases)	`gomock` ensures type safety and easy maintenance
HTTP Testing	`net/http/httptest`	Full Docker containers	`httptest` is built-in and fast for most cases
End-to-End Environment	Docker Compose	Kubernetes kind cluster	Docker Compose is simpler for local testing
Load Testing	`vegeta` (Go library)	`wrk` or `ab`	`vegeta` integrates well with Go tests
Code Coverage	`go test -cover`	`go test -coverprofile` + SonarQube	Built-in coverage tools are sufficient

Recommended Test File Structure

```
project-root/
├── internal/
│   ├── config/
│   │   ├── config.go
│   │   ├── config_test.go                    # Unit tests
│   │   └── config_integration_test.go        # Integration tests (tagged)
│   ├── router/
│   │   ├── router.go
│   │   ├── router_test.go
│   │   ├── loadbalancer_test.go
│   │   └── circuitbreaker_test.go
│   ├── middleware/
│   │   ├── auth/
│   │   │   ├── jwt_validator.go
│   │   │   ├── jwt_validator_test.go
│   │   │   ├── apikey_validator.go
│   │   │   └── apikey_validator_test.go
│   │   ├── transform/
│   │   │   ├── json_transformer.go
│   │   │   └── json_transformer_test.go
│   │   ├── metrics/
│   │   │   ├── middleware.go
│   │   │   └── middleware_test.go
│   │   └── recovery/
│   │       ├── middleware.go
│   │       └── middleware_test.go
│   ├── plugin/
│   │   ├── manager.go
│   │   ├── manager_test.go
│   │   └── mocks/
│   │       └── mock_plugin.go              # gomock generated
│   └── observability/
│       ├── logger.go
│       ├── logger_test.go
│       ├── tracer.go
│       └── tracer_test.go
├── cmd/
│   └── gateway/
│       ├── main.go
│       └── main_test.go                    # End-to-end tests
└── test/
    ├── e2e/
    │   ├── docker-compose.test.yml
    │   ├── test_backend/
    │   │   └── main.go                     # Simple Go test backend
    │   └── run_e2e.sh                      # Script to run E2E tests
    ├── fixtures/
    │   ├── configs/
    │   │   ├── simple_route.yaml
    │   │   └── auth_config.yaml
    │   └── jwt_tokens/
    │       ├── valid_jwt.txt
    │       └── expired_jwt.txt
    └── helpers/
        ├── test_helpers.go                 # Common test utilities
        └── http_client.go                  # Enhanced HTTP client for tests
```

Infrastructure Starter Code

Test Helper Utilities (`test/helpers/test_helpers.go`):

```go
package helpers

import (
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
    "time"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

// TestBackend creates a simple HTTP test server that echoes requests
func TestBackend(t *testing.T, handler http.HandlerFunc) *httptest.Server {
    t.Helper()
    return httptest.NewServer(http.HandlerFunc(handler))
}

// EchoHandler returns a handler that echoes the request as JSON
func EchoHandler(w http.ResponseWriter, r *http.Request) {
    body, _ := io.ReadAll(r.Body)
    defer r.Body.Close()

    response := map[string]interface{}{
        "method":  r.Method,
        "path":    r.Path,
        "headers": r.Header,
        "body":    string(body),
        "query":   r.URL.Query(),
    }

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

// ErrorHandler returns a handler that fails with given status code
func ErrorHandler(statusCode int, message string) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(statusCode)
```

```
        json.NewEncoder(w).Encode(map[string]string{
            "error": message,
        })
    }
}

// SlowHandler returns a handler that delays before responding
func SlowHandler(delay time.Duration) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(delay)
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("OK"))
    }
}

// ReadBodyJSON reads and unmarshals JSON response body
func ReadBodyJSON(t *testing.T, resp *http.Response, target interface{}) {
    t.Helper()
    body, err := io.ReadAll(resp.Body)
    require.NoError(t, err)
    defer resp.Body.Close()

    err = json.Unmarshal(body, target)
    require.NoError(t, err, "Failed to unmarshal JSON: %s", string(body))
}

// AssertMetrics checks if a metric exists with expected labels
func AssertMetrics(t *testing.T, metrics string, name string, expectedCount float64, labels map[string]string) {
    t.Helper()
    // Parse metrics text and assert values
    // Implementation depends on metrics format
}
```

Mock Backend for Integration Tests (`test/e2e/test_backend/main.go`):

```go
package main

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

func main() {
    port := "8081"
    if p := os.Getenv("PORT"); p != "" {
        port = p
    }

    mux := http.NewServeMux()

    // Health endpoint
    mux.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("OK"))
    })

    // Echo endpoint
    mux.HandleFunc("/echo", func(w http.ResponseWriter, r *http.Request) {
        response := map[string]interface{}{
            "timestamp": time.Now().Unix(),
            "method":    r.Method,
            "path":      r.URL.Path,
            "headers":   r.Header,
        }

        if r.Body != nil {
            var body map[string]interface{}
            if err := json.NewDecoder(r.Body).Decode(&body); err == nil {
                response["body"] = body
            }
        }
```

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

    // Simulate slow response
    mux.HandleFunc("/slow", func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(2 * time.Second)
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("Slow response"))
    })

    // Simulate errors
    mux.HandleFunc("/error", func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("Internal Server Error"))
    })

    log.Printf("Test backend listening on :%s", port)
    log.Fatal(http.ListenAndServe(":"+port, mux))
}
```

Core Test Skeleton Code

Router Unit Test Skeleton (`internal/router/router_test.go`):

```go
package router

import (
    "net/http"
    "testing"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestRouter_FindRoute(t *testing.T) {
    // TODO 1: Create a test router instance
    // TODO 2: Register test routes with different path patterns
    // TODO 3: Create HTTP request with specific path and method
    // TODO 4: Call FindRoute and verify correct route is matched
    // TODO 5: Test edge cases: no match, multiple matches, priority resolution
    // TODO 6: Verify that matched route includes correct upstream ID
    // TODO 7: Test with different HTTP methods (GET, POST, etc.)
    // TODO 8: Test with header-based matching when implemented
}

func TestRouter_RegisterRoute(t *testing.T) {
    // TODO 1: Create empty router
    // TODO 2: Register a valid route
    // TODO 3: Verify route appears in routing table
    // TODO 4: Attempt to register duplicate route ID - should error
    // TODO 5: Register route with non-existent upstream - should error
    // TODO 6: Test concurrent registration with goroutines
    // TODO 7: Verify radix tree is updated correctly
}

func TestRoundRobinLoadBalancer_SelectEndpoint(t *testing.T) {
    // TODO 1: Create load balancer with multiple endpoints
    // TODO 2: Call SelectEndpoint multiple times
    // TODO 3: Verify round-robin distribution (endpoint1, endpoint2, endpoint1, ...)
    // TODO 4: Mark one endpoint as unhealthy
    // TODO 5: Verify unhealthy endpoints are skipped
    // TODO 6: Mark all endpoints unhealthy - should return error
    // TODO 7: Test concurrent selection with goroutines
    // TODO 8: Verify atomic increment prevents race conditions
}

func TestCircuitBreaker_StateTransitions(t *testing.T) {
```

```
    // TODO 1: Create circuit breaker with threshold 3
    // TODO 2: Start in Closed state
    // TODO 3: Record failures until threshold - should transition to Open
    // TODO 4: Verify AllowRequest returns false in Open state
    // TODO 5: Wait for timeout - should transition to HalfOpen
    // TODO 6: Record success in HalfOpen - should transition to Closed
    // TODO 7: Record failure in HalfOpen - should transition back to Open
    // TODO 8: Test resetting failures on success in Closed state
}
```

Integration Test Skeleton (`internal/router/router_integration_test.go`):

```go
//go:build integration

package router

import (
    "net/http"
    "net/http/httptest"
    "testing"
    "time"

    "github.com/yourproject/internal/config"
    "github.com/yourproject/test/helpers"
)

func TestRouter_Integration_ProxyRequest(t *testing.T) {
    // TODO 1: Create test backend server with EchoHandler
    // TODO 2: Load configuration pointing to test backend
    // TODO 3: Create router instance from configuration
    // TODO 4: Create HTTP request to gateway endpoint
    // TODO 5: Call router's ServeHTTP method (or pipeline Execute)
    // TODO 6: Verify request is forwarded to backend
    // TODO 7: Verify response is returned to client
    // TODO 8: Check that X-Forwarded-For header is set
    // TODO 9: Verify backend receives correct path and method
}

func TestRouter_Integration_CircuitBreaker(t *testing.T) {
    // TODO 1: Create test backend that returns errors
    // TODO 2: Configure circuit breaker with low threshold
    // TODO 3: Make requests until circuit opens
    // TODO 4: Verify requests fail fast with 503
    // TODO 5: Wait for half-open timeout
    // TODO 6: Make successful request to close circuit
    // TODO 7: Verify normal operation resumes
}
```

Language-Specific Hints

1. **Use `t.Helper()` in test helper functions** to mark them as test helpers, making error reporting point to the actual test line instead of the helper function.
2. **Use table-driven tests** for testing multiple cases:

```go
func TestJSONTransformer(t *testing.T) {

    testCases := []struct {

        name     string

        input    map[string]interface{}

        expected map[string]interface{}

    }{

        {

            name: "simple field mapping",

            input: map[string]interface{}{"old_name": "value"},

            expected: map[string]interface{}{"new_name": "value"},

        },

        // More test cases...

    }



    for _, tc := range testCases {

        t.Run(tc.name, func(t *testing.T) {

            // Test logic here

        })

    }

}
```

3. **Use** `httptest.NewRecorder()` for testing HTTP handlers without starting a server:

```go
func TestHandler(t *testing.T) {

    req := httptest.NewRequest("GET", "/test", nil)

    w := httptest.NewRecorder()



    handler(w, req)



    resp := w.Result()

    assert.Equal(t, http.StatusOK, resp.StatusCode)

}
```

4. **Test concurrency with goroutines**:

```go
func TestConcurrentAccess(t *testing.T) {
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            // Concurrent access to shared resource
        }()
    }
    wg.Wait()
    // Verify no data races or corruption
}
```

5. **Use `-race` flag** to detect race conditions:

```bash
go test -race ./...
```

6. **Generate coverage reports**:

```bash
go test -coverprofile=coverage.out ./...
go tool cover -html=coverage.out -o coverage.html
```

7. **Benchmark critical paths**:

```go
func BenchmarkRouteMatching(b *testing.B) {
    router := setupRouterWithManyRoutes()
    req := httptest.NewRequest("GET", "/api/users/123", nil)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        router.FindRoute(req)
    }
}
```

Milestone Checkpoint Verification Script

Comprehensive Verification Script (`scripts/verify_milestone.sh`):

```bash
#!/bin/bash

set -e

echo "=== API Gateway Milestone Verification ==="

echo "Running tests for all milestones..."

# Check Go installation

if ! command -v go &> /dev/null; then

    echo "Error: Go is not installed"

    exit 1

fi

# Run unit tests for all components

echo "1. Running unit tests..."

go test ./internal/... -short -count=1 2>&1 | tee test_output.txt

if [ $? -ne 0 ]; then

    echo "❌ Unit tests failed"

    exit 1

fi

echo "✅ Unit tests passed"

# Run integration tests

echo "2. Running integration tests..."

go test -tags=integration ./internal/... -count=1 2>&1 | tee -a test_output.txt

if [ $? -ne 0 ]; then

    echo "❌ Integration tests failed"

    exit 1

fi

echo "✅ Integration tests passed"

# Check for race conditions

echo "3. Checking for race conditions..."

go test -race ./internal/router/... -count=1 2>&1 | grep -E "(WARNING: DATA RACE|PASS|FAIL)"

if [ $? -eq 0 ]; then

    echo "⚠️  Race condition check completed (check output for warnings)"

else

    echo "✅ No race conditions detected"

fi

# Build the gateway binary

echo "4. Building gateway binary..."

go build -o gateway ./cmd/gateway
```

```
if [ $? -ne 0 ]; then
    echo "❌ Build failed"
    exit 1
fi
echo "✅ Build successful"

# Run end-to-end tests if Docker is available
if command -v docker &> /dev/null && command -v docker-compose &> /dev/null; then
    echo "5. Running end-to-end tests (requires Docker)..."
    cd test/e2e
    docker-compose up --build --abort-on-container-exit --exit-code-from test-runner
    if [ $? -ne 0 ]; then
        echo "❌ End-to-end tests failed"
        exit 1
    fi
    docker-compose down
    cd ../..
    echo "✅ End-to-end tests passed"
else
    echo "⚠️  Skipping end-to-end tests (Docker not available)"
fi

# Check code coverage
echo "6. Checking code coverage..."
go test -coverprofile=coverage.out ./internal/... > /dev/null 2>&1
COVERAGE=$(go tool cover -func=coverage.out | grep total | awk '{print $3}')
echo "Code coverage: $COVERAGE"

if [[ "${COVERAGE%.*}" -lt 80 ]]; then
    echo "⚠️  Code coverage below 80%"
else
    echo "✅ Code coverage meets target"
fi

echo ""
echo "=== Verification Complete ==="
echo "All tests passed successfully!"
```

Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Environment differences, race conditions, timing issues	Run with `-race` flag, add `-count=10` to run tests multiple times, check CI logs for environment differences	Make tests more deterministic, use `testify.Eventually` for async checks, mock time-dependent functions
Integration test hangs indefinitely	Deadlock, unclosed connections, infinite loops	Add timeout with `testing.Short()`, use `pprof` to profile goroutines, check for missing `wg.Done()` calls	Ensure all goroutines can exit, add context timeouts, close response bodies
Intermittent 502 Bad Gateway in tests	Backend not ready, health check race condition	Add retry logic in tests, increase startup wait time, check health check interval	Use `testify.Eventually` to wait for backend readiness, implement readiness probes
Memory leak detected in tests	Unclosed resources, goroutine leaks	Run with `-benchmem`, use `runtime.ReadMemStats`, check for unclosed response bodies	Ensure `defer resp.Body.Close()`, use `httptest.Server.Close()` to clean up
Test data pollution between tests	Shared global state, improper test setup	Check for global variables, use `t.Cleanup()` to reset state, run tests in parallel with caution	Use dependency injection instead of globals, implement proper setup/teardown
Metrics test fails due to timing	Metrics updated asynchronously	Use `testify.Eventually` to poll metrics endpoint, wait for metric aggregation	Add small delay or retry logic for metric assertions
JWT validation test fails	Clock skew, token expiration timing	Use fixed time in tests with `time.Now().Add()` mock	Mock time functions in auth tests, use `testify's require.WithinDuration`

Debugging Guide

> **Milestone(s):** This debugging guide synthesizes insights from all milestones (1-4), providing practical tools and techniques for diagnosing issues that arise during implementation, testing, and operation of the API Gateway.

An API Gateway is a complex distributed system component that handles numerous concurrent requests while maintaining multiple internal states—routing tables, health statuses, circuit breaker states, authentication caches, and plugin chains. When something goes wrong, symptoms can be subtle and interconnected. This debugging guide provides a systematic approach to diagnosing and resolving common issues, organized around observable symptoms and supplemented with techniques for inspecting the gateway's internal state.

Common Bug Symptom → Cause → Fix

Think of debugging the gateway as **being a detective at a crime scene**. You arrive to find a symptom (the "crime")—perhaps a failed request or abnormal resource usage. Your job is to gather evidence (logs, metrics, traces), form hypotheses about root causes, and test those hypotheses until you identify the culprit. The following table maps common symptoms to their likely causes, diagnostic steps, and fixes.

Symptom	Likely Cause	Diagnostic Steps	Fix
502 Bad Gateway errors returned to clients	1. **No healthy endpoints** for the matched upstream. 2. **Circuit breaker open** for all available endpoints. 3. **Backend connection refused** (backend down or network partition). 4. **Backend timeout** exceeding gateway's configured timeout.	1. Check gateway logs for `no healthy endpoints` or `circuit breaker open` messages in `RequestContext.LogFields`. 2. Inspect health status metrics (`gateway_upstream_health_status`) or use a debug endpoint to list endpoint states. 3. Verify backend service is running and accessible (use `curl` directly to the endpoint). 4. Check if `RequestContext.ElapsedTime()` exceeds timeout thresholds.	1. Review health check configuration; ensure backends pass both passive and active checks. 2. Adjust circuit breaker thresholds (`failureThreshold` , `openTimeout`) or implement fallback responses. 3. Restart or scale the backend service; check network connectivity. 4. Increase gateway's proxy timeout or optimize backend performance.
Memory usage growing over time, eventually leading to OOM kills	1. **Memory leak in middleware** due to not releasing resources (e.g., not closing response bodies). 2. **Unbounded buffering** of large request/response bodies in transformation plugins. 3. **Accumulating goroutines** due to blocked operations or missing context cancellation. 4. **Cache growth without eviction** (e.g., validated token cache never expires).	1. Use `pprof` heap profile (`/debug/pprof/heap`) to identify which types are accumulating. 2. Check for `LimitedReadCloser` usage and `maxBodySize` enforcement in transformers. 3. Examine goroutine count via `pprof` (`/debug/pprof/goroutine`) and look for routines stuck in I/O. 4. Monitor cache size metrics and inspect eviction policies.	1. Ensure all `io.ReadCloser` resources are closed via `defer` and implement `RecoveryMiddleware` . 2. Enforce strict body size limits and stream large bodies when possible. 3. Propagate `RequestContext.Context()` to all downstream calls and respect cancellation. 4. Implement size or time-based eviction in caches (e.g., LRU with TTL).
High latency for requests even when backends are healthy	1. **Slow plugin execution** (e.g., expensive transformations, synchronous external calls). 2. **Lock contention** in shared structures like `RouterImpl.mu` or `RoundRobinLoadBalancer.mu` . 3. **Inefficient route matching** due to linear search instead of radix tree. 4. **Excessive garbage collection** due to many allocations per request (e.g., creating new `map[string]interface{}` for each transformation).	1. Use tracing to identify which middleware or plugin contributes most latency (look at span durations). 2. Profile with `pprof` mutex profile (`/debug/pprof/mutex`) to see contended locks. 3. Audit route count and matching algorithm; ensure radix tree is built correctly. 4. Monitor GC pauses via metrics (`go_gc_duration_seconds`); use allocation profiling.	1. Optimize plugin logic, cache expensive computations, or move async where possible. 2. Reduce lock scope, use `sync.RWMutex` for read-heavy paths, or shard data structures. 3. Verify `RouterImpl.tree` is populated and use `FindRoute` benchmarks. 4. Reuse buffers (e.g., `sync.Pool` for `[]byte`), pre-allocate slices, and avoid maps in hot paths.
Authentication failures for valid tokens	1. **Clock skew** between gateway and token issuer causing premature expiration validation. 2. **Missing or incorrect issuer/audience validation** in JWT validation rules. 3. **Token cache returning stale invalidated tokens** (e.g., revoked tokens still considered valid). 4. **Header extraction issue** (token not found in expected header `Authorization: Bearer <token>`).	1. Compare gateway system time with auth server time; check JWT `exp` and `iat` claims. 2. Verify `PluginConfig.Config` for `issuer` and `audience` fields match token claims. 3. Inspect cache hit rate and TTL settings; test with a revoked token. 4. Log raw headers in `RequestContext` to confirm token presence and format.	1. Synchronize clocks via NTP; allow small leeway (`leeway` parameter) in JWT validation. 2. Update `JWTValidationRule` configuration to include required claims validation. 3. Reduce cache TTL or implement explicit revocation checking (e.g., token introspection). 4. Ensure `Authenticator` extracts from correct header; support fallback locations (query param).
Routes not matching after configuration reload	1. **Race condition** during configuration reload where new routes are added but old routes still referenced by in-flight requests. 2. **Syntax error in new configuration** causing partial load (some routes loaded, others skipped). 3. **Path matching ambiguity** where a more specific route is shadowed by a broader prefix. 4. **Host header mismatch** due to case-sensitivity or port inclusion.	1. Check logs for `configuration reloaded` messages and errors; verify `RouterImpl.mu` locks during updates. 2. Validate configuration file with `validate(cfg)` and check for parse errors. 3. Inspect radix tree structure (debug endpoint) to see which paths are registered. 4. Log `Host` header value and compare with `RouteMatch.Headers` map.	1. Implement copy-on-write for `RouterImpl.routes` map and atomic swaps after full validation. 2. Use a staging file and atomic rename; rollback on validation errors. 3. Review route priority ordering; ensure more specific routes are registered first. 4. Normalize host headers (lowercase, strip port) before matching.
Plugin panics crashing the entire	1. **Nil pointer dereference** in plugin code due to missing configuration or	1. Examine stack trace in logs; identify which plugin and line number.	1. Add defensive nil checks in plugin `Execute` methods; validate config in

Symptom	Likely Cause	Diagnostic Steps	Fix
gateway process	uninitialized fields. 2. **Concurrent map read/write** in plugin shared state without proper synchronization. 3. **Recovered panic not handled** by `RecoveryMiddleware`, allowing panic to propagate to HTTP server. 4. **Resource exhaustion** (e.g., file descriptors) causing unexpected panics in I/O operations.	2. Look for `fatal error: concurrent map iteration and map write` in logs. 3. Verify `RecoveryMiddleware` is registered early in the pipeline and logs panic details. 4. Monitor system resources (open files, memory) and check ulimits.	`CreatePlugin`. 2. Protect shared maps with `sync.RWMutex` or use `sync.Map`. 3. Ensure `RecoveryMiddleware` wraps the entire chain and converts panics to 500 errors. 4. Implement graceful degradation (e.g., reject new requests) when resources are low.
Metrics not appearing in Prometheus endpoint	1. **Metrics plugin not registered** or disabled in configuration. 2. **High cardinality labels** causing Prometheus to drop metrics (e.g., unique request IDs as labels). 3. **Incorrect metric naming** (missing prefix, invalid characters). 4. **Race condition** during metric registration (multiple goroutines calling `prometheus.MustRegister`).	1. Check `PluginManager.registry` for `metrics` plugin; verify it's in the chain via `CreateChain`. 2. Inspect metric labels in `/metrics` endpoint; look for labels with many unique values. 3. Validate metric names against Prometheus naming conventions (snake_case). 4. Look for `panic: duplicate metrics collector registration` in logs.	1. Ensure `MetricsPlugin` is included in pipeline via `PluginConfig` and `GlobalManager.RegisterPlugin`. 2. Reduce label cardinality—use fixed set of values (e.g., status code groups) and avoid dynamic values. 3. Use `MetricsConfig.MetricPrefix` and sanitize label names. 4. Register metrics once at plugin initialization using `sync.Once` or init function.
Request body transformation failing for valid JSON	1. **Body size limit exceeded** causing `BodySizeError` before transformation. 2. **JSON schema mismatch**—expected fields missing or of wrong type. 3. **Content-Type header not set** to `application/json`, so transformer skips processing. 4. **Malformed JSON** that passes initial parsing but fails during specific field access.	1. Check `RequestContext.Error` for `BodySizeError` and log `Max` vs `Actual`. 2. Log raw request body (truncated) and transformation config `JSONTransformOps.Mappings`. 3. Inspect `Content-Type` header value; check for charset parameter. 4. Use `json.Decoder.DisallowUnknownFields()` to catch schema mismatches early.	1. Increase `maxBodySize` or enforce client-side chunking for large payloads. 2. Make transformations conditional with existence checks; provide default values. 3. Normalize `Content-Type` header (case-insensitive) or add configurable content type detection. 4. Wrap JSON unmarshal in recoverable panic handler; return 400 with specific error.
Distributed traces incomplete (missing gateway spans)	1. **Trace sampling rate too low**, causing most spans to be dropped. 2. **Context propagation failure**—trace headers not extracted from incoming request or injected to outgoing. 3. **Tracing plugin not executed** due to short-circuit (e.g., auth failure) before reaching tracing middleware. 4. **Trace exporter misconfigured** (wrong endpoint, authentication).	1. Check trace sampling configuration; adjust probability or use dynamic sampling. 2. Log extracted trace ID (`trace_id`) and span ID; verify headers are forwarded via `HeaderTransformer`. 3. Verify `TracingPlugin` priority places it early in pre-processing chain. 4. Check exporter logs for connection errors; validate OpenTelemetry collector is reachable.	1. Increase sampling rate for development; implement head-based sampling for production. 2. Ensure `TracingPlugin.propagator` extracts/injects using W3C Trace-Context format. 3. Move tracing to earliest possible middleware (even before auth) to capture all requests. 4. Configure correct exporter endpoint and credentials; add retry with exponential backoff.
Rate limiting inconsistent (same client sometimes limited, sometimes not)	1. **Multiple gateway instances** sharing no state, each maintaining separate counters. 2. **Clock drift** between instances causing window boundaries to misalign. 3. **Client identification flaky**—using unstable identifier like IP address (may change due to proxies). 4. **Rate limit plugin placed after caching middleware**, allowing cached responses to bypass limits.	1. Check if rate limit counts reset when hitting different gateway instances (sticky session test). 2. Compare timestamps across instances; monitor system clock synchronization. 3. Log client identifier used (IP, API key, JWT sub) and observe changes. 4. Examine plugin chain order; ensure rate limiting occurs before caching.	1. Implement distributed rate limiting using Redis or similar shared datastore. 2. Use NTP for clock sync; consider sliding window algorithms tolerant to slight drift. 3. Use stable client identifier (API key, user ID) and combine with IP for fallback. 4. Reorder plugins so rate limiting is in pre-processing before any response caching.

Debugging Techniques and Tools

Think of the gateway as a **patient in a hospital**—you have multiple diagnostic tools at your disposal to assess its health. Just as a doctor uses stethoscopes, blood tests, and imaging, you have logs, metrics, traces, and profilers. Each tool provides a different perspective: logs tell you what happened, metrics tell you how often it's happening, traces tell you where in the pipeline it's happening, and profilers tell you why it's happening (resource usage). The key is knowing which tool to apply for each symptom and how to interpret the results.

Structured Logging with RequestContext

The `RequestContext` is your primary diagnostic instrument, carrying a detailed record of each request's journey. Ensure your logging middleware captures key fields and that logs are emitted in a structured format (JSON) for easy querying.

Key fields to inspect:

- `RequestID`: Correlate all log entries for a single request across components.
- `ClientIP`: Identify problematic clients or network segments.
- `MatchedRoute.ID`: Confirm which route was selected (or if none).
- `SelectedEndpoint.URL`: See which backend instance handled the request.
- `Error` and `HTTPErrorCode`: Direct indication of failure point and type.
- `ElapsedTime()`: Pinpoint latency bottlenecks.
- `LogFields`: Custom plugin-added data (e.g., `auth_method`, `rate_limit_key`).

Example diagnostic log query (using `jq`):

```bash
# Find all failed requests with 502 errors in the last 5 minutes

cat gateway.log | jq 'select(.timestamp > (now - 300) and .status == 502)'
```

Implementation check: Verify that `AddLogField` is used consistently across middleware to enrich context, and that the final log entry includes all accumulated fields.

Metrics and Health Endpoints

The gateway's metrics endpoint (Prometheus-format) provides quantitative insights into system behavior over time. Key metrics to monitor:

Metric	Purpose	Alert Threshold
`gateway_requests_total` (counter)	Request rate by method, path, status	Sudden drop (service down) or spike (DDoS).
`gateway_request_duration_seconds` (histogram)	Latency distribution by route	p95 > 1s or significant increase.
`gateway_upstream_health_status` (gauge)	Endpoint health (1=healthy, 0=unhealthy)	Any endpoint 0 for > 5 minutes.
`gateway_circuit_breaker_state` (gauge)	State per endpoint (0=closed,1=open,2=half-open)	State 1 (open) for prolonged period.
`gateway_plugin_execution_duration` (histogram)	Plugin performance	Any plugin p99 > 100ms.
`go_goroutines` (gauge)	Goroutine count	Steady increase over time (leak).
`go_memstats_alloc_bytes` (gauge)	Memory allocation	Consistent growth without GC reclaim.

Health endpoints should be implemented for quick sanity checks:

- `GET /healthz` → Returns 200 if gateway is alive (minimal checks).
- `GET /readyz` → Returns 200 if gateway can handle requests (router initialized, upstreams healthy).
- `GET /debug/config` → Returns current configuration (sanitized) for verification.
- `GET /debug/routes` → Lists all registered routes and their upstreams.
- `GET /debug/upstreams` → Shows health status and circuit breaker state for each endpoint.

Distributed Tracing with OpenTelemetry

Tracing provides a visual timeline of request flow through the gateway's middleware chain and out to backends. Configure your tracing backend (Jaeger, Zipkin) to collect and display traces.

What to look for in traces:

- **Long spans**: Identify which middleware or backend call contributes most to latency.
- **Missing spans**: If a plugin doesn't create a span, it may not be executing.
- **Error tags**: Spans tagged with `error=true` indicate where failures occur.
- **Context propagation**: Verify trace IDs are consistent across gateway and backend spans.

Debugging with trace context: When a client reports a problem, ask for the trace ID (should be returned in response headers like `X-Trace-Id`). Use this ID to retrieve the full trace and see exactly what happened.

Profiling with pprof

Go's built-in `pprof` provides low-level insights into CPU, memory, and goroutine behavior. Enable it by importing `net/http/pprof` and registering its handlers (usually on a separate admin port).

Key profiles and their use:

- **CPU profile** (`/debug/pprof/profile?seconds=30`): Shows which functions consume the most CPU during the sampling period. Look for hotspots in transformation logic or regex matching.
- **Heap profile** (`/debug/pprof/heap`): Shows allocated memory by type. Identify memory leaks by comparing two snapshots taken minutes apart—growing allocations indicate leaks.
- **Goroutine profile** (`/debug/pprof/goroutine?debug=2`): Lists all goroutines with stack traces. Look for routines blocked on locks, I/O, or channel operations.
- **Mutex profile** (`/debug/pprof/mutex`): Shows lock contention. Identify heavily contended mutexes (e.g., `RouterImpl.mu`).

Example analysis flow:

1. Observe high CPU usage.
2. Take a 30-second CPU profile: `go tool pprof http://localhost:6060/debug/pprof/profile` .
3. Use `top` , `list` , and `web` commands to identify hot functions.
4. Optimize or cache results for those functions.

Configuration Validation and Dry-Run

Many issues stem from misconfiguration. Implement a dry-run mode that validates configuration files without starting the gateway:

```bash
./gateway --validate --config gateway.yaml
```

This should call `LoadFromFile` followed by `validate` , checking for:

- Duplicate route IDs or overlapping path patterns.
- Upstreams with no endpoints.
- Plugin configurations with missing required fields.
- Invalid regular expressions in header matching rules.

Versioned configuration: Keep previous configuration versions and the ability to roll back quickly. Log configuration changes and who made them.

Simulating Failures and Chaos Testing

Proactively discover weaknesses by simulating failure scenarios in a controlled environment. Use tools like `tc` (Traffic Control) to introduce network latency, packet loss, or partitions between gateway and backends.

Common simulations:

- **Backend slow response**: Add delay to upstream endpoints (e.g., 5s). Verify circuit breaker trips and fallback responses work.
- **Backend failure**: Kill backend instances. Confirm health checks detect downtime and traffic redistributes.
- **High load**: Use load testing tools (wrk, vegeta) to push request rates beyond normal. Observe how gateway handles connection pooling, memory, and CPU.

Chaos testing checklist:

- [ ] Circuit breaker opens and closes as expected.
- [ ] Health checks remove unhealthy endpoints.
- [ ] Memory usage stabilizes under load (no leaks).
- [ ] Error rates don't exceed thresholds.
- [ ] Metrics reflect the simulated failures.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Structured JSON to stdout, collected by Fluentd/Logstash	Direct integration with Elasticsearch or Loki with buffering and retry
Metrics	Prometheus client library exposing `/metrics` endpoint	Push metrics to Prometheus via remote write, add custom exporters for business metrics
Tracing	OpenTelemetry SDK with console exporter (for development)	Jaeger or Zipkin exporter with sampling and baggage propagation
Profiling	Built-in `net/http/pprof` endpoints	Continuous profiling with Pyroscope or Google's pprof visualization tools
Configuration Validation	Custom `validate` function with rule-based checks	JSON Schema or OpenAPI validation, integration with Kubernetes admission webhook

B. Recommended File/Module Structure

Extend the existing project structure with debugging-specific utilities:

```
project-root/
  cmd/gateway/main.go                    # Entry point with flag parsing
  internal/
    middleware/
      recovery.go                         # RecoveryMiddleware
      error_handler.go                    # ErrorHandlerMiddleware
      logging.go                          # Structured logging middleware
      metrics.go                          # MetricsPlugin
      tracing.go                          # TracingPlugin
    config/
      validator.go                        # validate() function
      dryrun.go                           # Dry-run validation mode
    router/
      debug.go                            # /debug endpoints for routes/upstreams
    plugin/
      manager.go                          # PluginManager
    debug/
      pprof.go                            # pprof HTTP handler registration
      health.go                           # /healthz, /readyz endpoints
      config_dump.go                      # /debug/config endpoint
    util/
      context.go                          # RequestContext methods
      limits.go                           # Body size limiting utilities
  pkg/
    api/                                  # Public API types (if any)
  test/
    chaos/                                # Chaos testing scenarios
    load/                                 # Load testing scripts
```

C. Infrastructure Starter Code

Complete Debug Endpoint Handler (`internal/debug/config_dump.go`):

```go
package debug

import (
    "encoding/json"
    "net/http"
    "sync"

    "yourproject/internal/config"
)

var (
    currentConfig     *config.Config
    configMu          sync.RWMutex
)

// UpdateConfig atomically updates the configuration for debug endpoints.
func UpdateConfig(cfg *config.Config) {
    configMu.Lock()
    defer configMu.Unlock()
    currentConfig = cfg
}

// ConfigDumpHandler returns the current configuration (sanitized).
func ConfigDumpHandler(w http.ResponseWriter, r *http.Request) {
    configMu.RLock()
    cfg := currentConfig
    configMu.RUnlock()

    if cfg == nil {
        http.Error(w, "configuration not loaded", http.StatusNotFound)
        return
    }

    // Sanitize sensitive fields (e.g., API keys, tokens) before serializing.
    sanitized := sanitizeConfig(cfg)

    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(sanitized); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}

// sanitizeConfig creates a safe copy of configuration without secrets.
```

```
func sanitizeConfig(cfg *config.Config) *config.Config {
    // Implementation: iterate through PluginConfig and remove sensitive Config values.
    // Return a deep copy with secrets redacted.
}
```

D. Core Logic Skeleton Code

Request-Specific Debug Logging in Middleware (`internal/middleware/logging.go`):

```go
// LoggingMiddleware captures detailed request/response info.
type LoggingMiddleware struct {
    logger *zap.Logger
}

func (m *LoggingMiddleware) Execute(ctx *middleware.RequestContext, w http.ResponseWriter, r *http.Request) bool {
    start := time.Now()

    // Continue the chain
    defer func() {
        latency := time.Since(start)
        // Build structured log entry from RequestContext
        fields := []zap.Field{
            zap.String("request_id", ctx.RequestID),
            zap.String("method", r.Method),
            zap.String("path", r.URL.Path),
            zap.String("client_ip", ctx.ClientIP),
            zap.Duration("latency", latency),
            zap.Int("status", ctx.HTTPErrorCode), // Set by error handler
            zap.String("matched_route", ctx.MatchedRoute.ID),
        }

        // Add all custom log fields from context
        for k, v := range ctx.LogFields {
            fields = append(fields, zap.Any(k, v))
        }

        if ctx.Error != nil {
            fields = append(fields, zap.Error(ctx.Error))
            m.logger.Error("request failed", fields...)
        } else {
            m.logger.Info("request completed", fields...)
        }
    }()

    return true // Continue chain
}
```

Health Check Endpoint with Dependency Verification (`internal/debug/health.go`):

GO

```go
// HealthChecker defines a component that can report its health.
type HealthChecker interface {
    HealthCheck() error
}

var healthCheckers []HealthChecker

// RegisterHealthChecker adds a component to the health check.
func RegisterHealthChecker(checker HealthChecker) {
    healthCheckers = append(healthCheckers, checker)
}

// ReadyzHandler performs readiness checks on all registered components.
func ReadyzHandler(w http.ResponseWriter, r *http.Request) {
    for _, checker := range healthCheckers {
        if err := checker.HealthCheck(); err != nil {
            http.Error(w, fmt.Sprintf("unhealthy: %v", err), http.StatusServiceUnavailable)
            return
        }
    }
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}

// Example health check for Router
func (r *RouterImpl) HealthCheck() error {
    r.mu.RLock()
    defer r.mu.RUnlock()

    if len(r.routes) == 0 {
        return errors.New("no routes configured")
    }

    // Check that each upstream has at least one healthy endpoint
    for _, upstream := range r.upstreams {
        healthy := false
        for _, endpoint := range upstream.Endpoints {
            // Check circuit breaker or health status
            if breaker, ok := r.breakers[endpoint.ID]; ok && breaker.AllowRequest() {
                healthy = true
                break
            }
```

```
        }
        if !healthy {
            return fmt.Errorf("upstream %s has no healthy endpoints", upstream.ID)
        }
    }

    return nil
}
```

E. Language-Specific Hints

Go-Specific Debugging Tips:

- Use `context.WithTimeout` and `context.WithCancel` for all downstream calls to prevent hung goroutines.
- When using `httputil.ReverseProxy`, set `ErrorHandler` to capture backend errors and log them into `RequestContext`.
- For memory profiling, run `go tool pprof -alloc_space http://localhost:6060/debug/pprof/heap` to see allocated memory.
- Use `sync.Pool` for frequently allocated objects like `[]byte` buffers for JSON transformations.
- Enable race detection during tests: `go test -race ./...`. This catches many concurrency bugs early.
- For blocking profiling, import `_ "net/http/pprof"` and call `pprof.Lookup("block").WriteTo(...)` to see goroutines blocked on mutexes/channels.

F. Milestone Checkpoint

After implementing debugging utilities, verify they work:

1. **Start the gateway** with pprof enabled:

```bash
./gateway --config gateway.yaml --pprof-addr :6060
```

2. **Test health endpoints**:

```bash
curl -v http://localhost:8080/healthz
curl -v http://localhost:8080/readyz
curl http://localhost:8080/debug/config | jq .
```

3. **Generate and analyze a profile**:

```bash
# Capture 30s CPU profile
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

# View heap allocation
go tool pprof http://localhost:6060/debug/pprof/heap

# Get goroutine dump
curl http://localhost:6060/debug/pprof/goroutine?debug=2 > goroutines.txt
```

4. **Verify structured logging**:

```bash
# Send a test request
curl http://localhost:8080/api/users

# Check logs (JSON format)
tail -f gateway.log | jq 'select(.request_id == "<id>")'
```

Expected outcomes:

- Health endpoints return 200 OK when gateway is properly initialized.
- Configuration dump returns sanitized config without secrets.
- Profiling endpoints respond without error.
- Each request generates a structured log entry with `request_id`, latency, and status.

Signs of trouble:

- `/readyz` returns 503 → Check router initialization and upstream health.
- Profiling endpoints timeout → Ensure pprof server is on separate port and not blocked by firewall.
- Logs missing fields → Verify `AddLogField` is called in each middleware.
- Memory growth during load test → Use heap profile to identify leaking types.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Gateway crashes on startup with `panic: duplicate metrics collector registration`	Multiple plugin instances registering same metrics	Check stack trace for metric name; examine plugin initialization order.	Use `sync.Once` in plugin `Init` or register metrics in `init()` function.
Requests hang indefinitely	Deadlock in middleware chain or blocked backend call	Use goroutine dump (`debug/pprof/goroutine`) to see what all goroutines are waiting for.	Ensure timeouts are set via `context.Context` and avoid holding locks during I/O.
High error rate after deployment	New configuration has incorrect route matches or upstream endpoints	Compare old vs new config diffs; enable configuration validation in CI/CD pipeline.	Implement canary deployment: roll out new config to small percentage of traffic first.
Metrics show high latency but backend latency is low	Gateway overhead from too many plugins or inefficient transformations	Use tracing to measure time spent in each middleware; disable plugins one by one to isolate.	Optimize plugin execution, consider moving heavy transformations to backends, or use caching.

Future Extensions

> **Milestone(s):** This section explores potential enhancements that build upon the foundation laid by all milestones (1-4), showing how the architecture can evolve to support advanced use cases.

The gateway architecture we've designed forms a solid foundation for an enterprise API gateway, but software systems must evolve to meet changing requirements. This section explores several advanced extensions that the current architecture can naturally accommodate without requiring a complete rewrite. Each extension represents a potential future milestone that builds upon the existing component structure, plugin system, and data model.

Think of these extensions as **modular upgrades to an already functional building**. Just as you can add solar panels, a security system, or smart home automation to a well-constructed house without tearing down walls, these enhancements integrate with the gateway's core systems through well-defined interfaces and extension points.

Possible Enhancements

The gateway's plugin architecture, layered processing pipeline, and configuration model create natural extension points for advanced functionality. Here are four significant enhancements that align with common enterprise requirements:

WebSocket Proxying and Protocol Translation

What it is: Extending the gateway to handle bidirectional WebSocket connections, allowing real-time applications to communicate through the gateway while still benefiting from authentication, rate limiting, and observability features.

Why it's valuable: Modern applications increasingly require real-time capabilities (chat, notifications, live updates). Currently, these applications must bypass the gateway or implement separate infrastructure, losing centralized security and observability benefits.

How it integrates: The existing `Router` interface would need a protocol-aware matching capability. A new `WebSocketProxy` component would:

1. Intercept WebSocket upgrade requests (`Connection: Upgrade` , `Upgrade: websocket`)
2. Perform authentication and authorization using existing `Authenticator` plugins
3. Establish a persistent connection pool to backend services
4. Proxy frames bidirectionally while applying middleware to connection events
5. Integrate with observability systems for connection metrics and message tracing

Key design considerations:

- **Connection Lifecycle Management:** WebSocket connections are long-lived (minutes to hours), requiring different resource management than HTTP requests
- **Message-Level Middleware:** While HTTP middleware operates on complete requests/responses, WebSocket middleware needs to process individual frames or messages
- **Stateful Routing:** Once a WebSocket connection is established to a backend instance, subsequent messages should route to the same instance (sticky sessions)
- **Graceful Shutdown:** Active WebSocket connections must be properly drained during gateway restarts or configuration reloads

Architecture Decision: Connection Multiplexing vs. Direct Proxy

> **Decision: WebSocket Connection Multiplexing Strategy**
>
> - **Context:** When proxying WebSocket connections, we must decide how to manage the relationship between client connections and backend connections. Each client WebSocket could establish a separate connection to the backend, or multiple client connections could share a pool of backend connections.
> - **Options Considered:**
>   1. **Direct 1:1 Proxy:** Each client WebSocket establishes a dedicated connection to the backend. Simple to implement but scales poorly with many clients.
>   2. **Connection Pooling:** Maintain a fixed pool of backend WebSocket connections that client connections share. Efficient but requires message routing logic.
>   3. **Hybrid Approach:** Use direct connections for most cases but implement connection pooling for high-traffic backend services.
> - **Decision:** Implement direct 1:1 proxying initially, with architecture prepared for future connection pooling.
> - **Rationale:** Direct proxying maintains a clean separation of concerns—each client's connection state is independent, simplifying debugging and error handling. The gateway's stateless design is preserved for the control plane (though data plane becomes stateful). We can add connection pooling later without breaking existing functionality.
> - **Consequences:** Higher connection overhead on backend services, but simpler implementation and clearer connection lifecycle management. Backend services must handle the connection load.

Option	Pros	Cons	Suitable For
Direct 1:1 Proxy	Simple implementation, clear connection lifecycle, easy debugging	High backend connection overhead, less efficient for many clients	Initial implementation, low-to-medium scale
Connection Pooling	Efficient backend resource usage, scales to many clients	Complex message routing, shared state management, harder debugging	High-scale production with many concurrent clients
Hybrid Approach	Balances efficiency and simplicity, adaptable to different services	Most complex implementation, configuration overhead	Mature deployments with mixed workload patterns

Common Pitfalls in WebSocket Implementation:

- ⚠️ **Pitfall: Forgetting to handle control frames (ping/pong/close)**
  Why it's wrong: WebSocket protocol uses control frames for connection health checks and graceful closure. Ignoring them causes connections to hang or terminate unexpectedly.
  Fix: Implement proper frame type detection and forwarding. Use Go's `golang.org/x/net/websocket` or `gorilla/websocket` libraries which handle control frames automatically.
- ⚠️ **Pitfall: Not limiting message size**
  Why it's wrong: WebSocket messages can be arbitrarily large (the protocol supports fragmentation). Buffering unlimited messages can cause memory exhaustion attacks.
  Fix: Configure maximum message sizes per route, similar to the `maxBodySize` limit for HTTP requests in the `JSONTransformer`.
- ⚠️ **Pitfall: Leaking connections during reloads**
  Why it's wrong: When configuration reloads or the gateway restarts, active WebSocket connections might be terminated abruptly, disrupting client applications.
  Fix: Implement graceful shutdown that waits for active connections to complete or timeout, and use connection draining patterns.

GraphQL Federation and Query Optimization

What it is: Transforming the gateway into a GraphQL federation layer that can compose multiple REST and GraphQL backend services into a unified GraphQL schema, with query planning and optimization.

Why it's valuable: GraphQL provides client-driven querying with precise data requirements, reducing over-fetching. Federation allows teams to develop services independently while presenting a unified graph to clients. The gateway is ideally positioned to orchestrate these distributed queries.

How it integrates: This would be implemented as a specialized `GraphQLFederationPlugin` that:

1. Compiles a federated schema from multiple backend schema definitions
2. Parses incoming GraphQL queries and creates query plans that decompose them into backend requests

3. Uses the existing `Router` to route sub-queries to appropriate services
4. Aggregates responses using enhanced `JSONTransformer` capabilities
5. Applies the same authentication, rate limiting, and observability middleware to GraphQL operations

Key design considerations:

- **Schema Composition and Validation:** The gateway must merge type definitions, handle conflicts, and validate the composed schema
- **Query Planning Complexity:** GraphQL queries with nested relationships spanning multiple services require sophisticated planning algorithms
- **Error Handling:** Partial failures in federated queries need careful error propagation to clients
- **Caching Strategy:** GraphQL responses are highly variable based on query structure, requiring different caching approaches than REST

Example GraphQL Federation Flow:

1. Client sends GraphQL query requesting user data with their recent orders
2. Gateway parses query and identifies that `user` field resolves to UserService (REST) and `orders` field resolves to OrderService (REST)
3. Query planner creates execution plan:
   - First fetch user from UserService at `/users/{id}`
   - Then fetch orders from OrderService at `/orders?userId={userId}`
   - Merge results according to GraphQL response specification
4. Gateway executes plan using existing HTTP client infrastructure
5. Responses are transformed and combined into single GraphQL response
6. All steps are instrumented with existing observability plugins

Data structure extensions needed:

```go
// GraphQLRoute extends Route with GraphQL-specific configuration
type GraphQLRoute struct {
    Route                 // Embedded base Route
    SchemaURL   string     // URL to fetch GraphQL schema
    Resolvers   []Resolver // Field-to-backend mappings
    QueryPlanCacheSize int // Cache size for query plans
}

// Resolver maps a GraphQL field to a backend operation
type Resolver struct {
    FieldPath   string     // e.g., "Query.user", "User.orders"
    BackendType string     // "REST", "GraphQL", "gRPC"
    Operation   ResolverOp // HTTP method + path template or GraphQL query template
    BatchConfig BatchConfig // For batching multiple field resolutions
}
```

Common Pitfalls in GraphQL Federation:

- ⚠️ **Pitfall: N+1 query problem in naive resolution**
  Why it's wrong: Resolving each object's related fields independently causes many sequential backend requests, leading to poor performance.
  Fix: Implement query batching and data loader patterns that collect related resolutions and execute them in batches.
- ⚠️ **Pitfall: Schema evolution breaking clients**
  Why it's wrong: GraphQL schemas evolve over time. Without proper versioning or gradual migration strategies, schema changes can break existing client queries.
  Fix: Implement schema validation rules, maintain backward compatibility through deprecated fields, and consider schema versioning strategies.
- ⚠️ **Pitfall: Unlimited query complexity**
  Why it's wrong: GraphQL allows clients to request deeply nested relationships, which could cause expensive query plans or even denial of service.
  Fix: Implement query cost analysis, depth limiting, and complexity scoring to reject or throttle expensive queries.

WebAssembly (WASM) Plugin Runtime

What it is: Extending the plugin system to support WebAssembly modules as plugins, allowing plugins to be written in multiple languages (Rust, C++, AssemblyScript) and loaded dynamically with sandboxed execution.

Why it's valuable: WASM provides language agnosticism, strong security isolation, and dynamic loading capabilities. Teams could write custom transformations or authentication logic in their preferred language without requiring Go expertise or gateway recompilation.

How it integrates: A `WasmPluginRuntime` component would extend the `PluginManager` to:

1. Load WASM modules from filesystem or remote URLs
2. Provide a host interface exposing gateway capabilities (HTTP client, logging, configuration access)
3. Execute WASM plugin functions through a lightweight runtime (like `wasmtime` or `wasmer`)
4. Marshal data between Go and WASM memory spaces efficiently
5. Enforce resource limits (CPU, memory) on plugin execution

Architecture Decision: WASM Plugin Isolation Model

> **Decision: Sandboxed WASM Plugin Execution**
>
> - **Context:** WASM plugins must balance flexibility with security. Malicious or buggy plugins should not crash the gateway or access unauthorized resources.
> - **Options Considered:**
>   1. **Shared Memory Model:** Plugins share the gateway's memory space with careful bounds checking. High performance but riskier.
>   2. **Process Isolation:** Each plugin runs in a separate OS process. Maximum security but high overhead.
>   3. **WASI Sandbox:** Use WebAssembly System Interface (WASI) to provide controlled filesystem and network access. Balanced approach.
> - **Decision:** Implement WASI-based sandboxing with configurable capabilities per plugin.
> - **Rationale:** WASI provides standardized, fine-grained capability-based security. We can grant plugins only the capabilities they need (e.g., HTTP client access but not filesystem). The performance overhead is acceptable for most plugin use cases, and the security benefits are substantial for multi-tenant deployments.
> - **Consequences:** Some system-level plugins (like custom health checks that need OS metrics) may require elevated capabilities. We'll need a capability declaration and validation system.

Option	Pros	Cons	Performance Impact
Shared Memory Model	Minimal overhead, fast data sharing	Security risks, plugin crashes affect gateway	~1-5% overhead
Process Isolation	Maximum security, crash containment	High context-switch overhead, serialization costs	~50-100% overhead
WASI Sandbox	Good security, standardized API, capability-based	Moderate overhead, some WASI runtime complexity	~10-20% overhead

WASM Host Interface Design: The gateway would expose these capabilities to WASM plugins through imported functions:

- `http_fetch(method, url, headers, body) → (status, headers, body)`: Make HTTP requests to backends
- `log(level, message, fields)`: Write structured logs
- `get_config(key) → value`: Read plugin configuration
- `metrics_counter(name, value, labels)`: Record metrics
- `cache_get(key) → value` / `cache_set(key, value, ttl)`: Use gateway cache

Example WASM Plugin Lifecycle:

1. Gateway loads `custom-auth.wasm` module on startup
2. Module is validated and instantiated with configured capabilities
3. For each request, gateway copies `RequestContext` data into WASM linear memory
4. Calls WASM export `process_request(ptr, len) → (ptr, len)`
5. WASM plugin executes auth logic, potentially calling host functions
6. Gateway reads result from WASM memory and continues processing
7. Plugin memory is reset between requests (or reused with pooling)

Common Pitfalls in WASM Plugin Systems:

- ⚠️ **Pitfall: Memory leaks across requests**
  Why it's wrong: WASM modules maintain their own linear memory. If not properly reset between requests, memory usage grows unbounded.
  Fix: Implement a memory pooling system that reinitializes WASM memory between requests or uses separate instances per request.
- ⚠️ **Pitfall: Serialization overhead dominating execution time**
  Why it's wrong: Marshaling complex `RequestContext` between Go and WASM can be expensive, especially for large request/response bodies.

Fix: Implement zero-copy or shared memory techniques where possible, and provide selective data access (plugins request only the fields they need).

- ⚠️ **Pitfall: Blocking host calls deadlocking the gateway**
  Why it's wrong: If a WASM plugin makes a synchronous host call that blocks (like `http_fetch`), it blocks the entire goroutine.
  Fix: Design host interface to be asynchronous or use separate goroutine pools for plugin execution.

Administrative REST API and Dynamic Configuration

What it is: Exposing the gateway's configuration and operational controls through a RESTful API, enabling dynamic updates, runtime metrics inspection, and plugin management without restarting the gateway process.

Why it's valuable: Operational efficiency—devops teams can manage routing rules, adjust rate limits, and deploy plugins through API calls instead of file edits and process restarts. Enables GitOps workflows, self-service configuration for development teams, and integration with CI/CD pipelines.

How it integrates: The admin API would run as a separate HTTP server within the same process (or a separate administrative port on the same server):

1. Extends the existing `Config` model with versioning and validation
2. Implements CRUD endpoints for `Route`, `Upstream`, `PluginConfig` resources
3. Provides operational endpoints for health checks, metrics, and runtime statistics
4. Uses the same authentication/authorization middleware (with admin-specific rules)
5. Integrates with the existing configuration reload mechanism for atomic updates

Key design considerations:

- **Configuration Atomicity:** Updates to multiple related resources (routes and their upstreams) should be applied atomically
- **Version Control Integration:** The API should support importing/exporting configuration to work with Git-based workflows
- **Access Control:** Administrative endpoints need robust RBAC with audit logging
- **Validation and Dry-run:** Configuration changes should be validated before application, with dry-run capability

Architecture Decision: Configuration Storage Backend

> **Decision: Dual Configuration Source with API Priority**
>
> - **Context:** The gateway needs to support both file-based configuration (for initial deployment and fallback) and API-based configuration (for dynamic management). We must define how these sources interact.
> - **Options Considered:**
>   1. **API-Only Configuration:** File config is only used at startup, all runtime changes via API. Simple but less resilient to gateway restarts.
>   2. **File-Only with API Writing to File:** API changes are persisted to config file. Durable but requires filesystem access.
>   3. **Dual Source with Merge Strategy:** Both sources are read, with API having precedence. Most flexible but complex merge logic.
>   4. **External Config Service:** Configuration stored in external service (etcd, Consul). Maximum flexibility but additional infrastructure.
> - **Decision:** Implement API writing to file with versioning, plus ability to reload from file.
> - **Rationale:** Writing API changes to the configuration file provides durability across restarts while maintaining the simplicity of file-based configuration. We can add an export feature to sync from file to API on startup. This approach doesn't require additional infrastructure while supporting dynamic management.
> - **Consequences:** Need file locking for concurrent API operations, and careful handling of file permission errors. The filesystem becomes a single point of failure for configuration persistence.

Option	Pros	Cons	Operational Complexity
API-Only	Simple implementation, no filesystem dependencies	Configuration lost on restart, requires backup strategy	Low
API Writes to File	Durable, familiar model, easy backup/version control	Filesystem permissions issues, single-writer constraint	Medium
Dual Source Merge	Flexible, supports GitOps and dynamic changes	Complex merge conflicts, hard to debug configuration state	High
External Service	Distributed, supports multiple gateway instances	Additional infrastructure, new failure modes	Highest

Example Admin API Endpoints:

Method	Path	Description	Authentication Required
`GET`	`/admin/routes`	List all routes	Yes (read role)
`POST`	`/admin/routes`	Create new route	Yes (write role)
`PUT`	`/admin/routes/{id}`	Update route	Yes (write role)
`DELETE`	`/admin/routes/{id}`	Delete route	Yes (write role)
`GET`	`/admin/upstreams/health`	Health status of all upstreams	Yes (read role)
`POST`	`/admin/plugins/{name}/enable`	Enable plugin globally	Yes (admin role)
`GET`	`/admin/metrics`	Prometheus metrics	Yes (read role)
`POST`	`/admin/config/reload`	Reload configuration from file	Yes (admin role)
`GET`	`/admin/requests/active`	Currently active requests	Yes (debug role)

Configuration Versioning and Rollback: Each configuration change would be tracked with metadata:

```go
type ConfigChange struct {
    ID        string                // Unique change identifier
    Timestamp time.Time             // When change was applied
    User      string                // Who made the change
    Comment   string                // Change description
    Config    Config                // Complete configuration snapshot
    ParentID  string                // Previous change ID (for rollback chain)
}
```

Common Pitfalls in Admin API Design:

- ⚠️ **Pitfall: Race conditions during configuration updates**
  Why it's wrong: If two admins update configuration simultaneously, the last write wins and may overwrite intended changes.
  Fix: Implement optimistic concurrency control with version stamps or ETags, requiring clients to specify the current version they're updating.
- ⚠️ **Pitfall: No validation before applying changes**
  Why it's wrong: Invalid configuration (circular route references, invalid regex patterns) could break the gateway at runtime.
  Fix: Implement comprehensive validation with dry-run mode. Changes are validated in a sandbox before being committed to the active configuration.
- ⚠️ **Pitfall: Admin API becomes security weak point**
  Why it's wrong: Exposing powerful administrative controls without proper authentication/authorization allows attackers to reconfigure the gateway maliciously.
  Fix: Use strong authentication (mTLS, JWT), implement RBAC with least privilege, audit log all admin actions, and consider running admin API on separate network interface.

Implementation Guidance

Technology Recommendations for Extensions:

Component	Simple Option	Advanced Option
WebSocket Proxying	`gorilla/websocket` for basic framing	Custom protocol with connection pooling and message transformation
GraphQL Federation	Schema stitching with `graphql-go/graphql`	Apollo Federation compatible implementation with query planning
WASM Runtime	`wasmer-go` or `wasmtime-go` runtime	Custom ABI with streaming data transfer and capability system
Admin API	Separate HTTP server with REST endpoints	gRPC admin interface with bidirectional streaming for real-time updates

Recommended File/Module Structure for Extensions:

```
project-root/
  cmd/
    gateway/           # Main gateway binary
    admin-server/      # Optional separate admin server binary
  internal/
    extensions/        # All extension implementations
      websocket/
        proxy.go       # WebSocket proxy implementation
        manager.go     # Connection lifecycle management
        middleware.go  # WebSocket-specific middleware
      graphql/
        federation.go  # Schema federation logic
        planner.go     # Query planning engine
        resolver.go    # Backend resolver mappings
      wasm/
        runtime.go     # WASM runtime wrapper
        host.go        # Host interface implementation
        plugin.go      # WASM plugin adapter
      admin/
        api.go         # REST API handlers
        store.go       # Configuration storage
        rbac.go        # Role-based access control
    # Existing directories remain unchanged
```

Core Logic Skeleton for WebSocket Proxy:

```go
// WebSocketProxy handles WebSocket upgrade requests and bidirectional proxying
type WebSocketProxy struct {
    upstream   *Upstream
    dialer     *websocket.Dialer
    maxMsgSize int64
    // ... other fields
}

// HandleUpgrade processes HTTP WebSocket upgrade requests
func (p *WebSocketProxy) HandleUpgrade(w http.ResponseWriter, r *http.Request, ctx *RequestContext) error {
    // TODO 1: Validate WebSocket upgrade headers
    // TODO 2: Authenticate connection using existing auth middleware
    // TODO 3: Select backend endpoint using load balancer
    // TODO 4: Establish WebSocket connection to backend
    // TODO 5: Upgrade client connection to WebSocket
    // TODO 6: Start bidirectional copying goroutines
    // TODO 7: Handle graceful shutdown signals
    // TODO 8: Monitor connection health with ping/pong
    // TODO 9: Clean up resources on connection close
    return nil
}

// copyClientToBackend handles messages from client to backend
func (p *WebSocketProxy) copyClientToBackend(clientConn, backendConn *websocket.Conn, ctx *RequestContext) {
    // TODO 1: Read message from client with size limit
    // TODO 2: Apply request transformation middleware if configured
    // TODO 3: Write transformed message to backend
    // TODO 4: Handle different message types (text, binary, control)
    // TODO 5: Record metrics for message size and latency
    // TODO 6: Propagate cancellation from context
}
```

Core Logic Skeleton for GraphQL Federation:

```go
// GraphQLFederationPlugin implements GraphQL query federation
type GraphQLFederationPlugin struct {
    schema     *graphql.Schema
    resolvers  map[string]Resolver
    planner    QueryPlanner
    // ... other fields
}

// Execute processes GraphQL queries through the federation layer
func (p *GraphQLFederationPlugin) Execute(ctx *RequestContext, w http.ResponseWriter, r *http.Request) bool {
    // TODO 1: Parse GraphQL query from request body
    // TODO 2: Validate query against federated schema
    // TODO 3: Create query execution plan breaking query into backend operations
    // TODO 4: Execute plan using concurrent backend requests
    // TODO 5: Collect and merge results from all backends
    // TODO 6: Handle partial failures with error propagation rules
    // TODO 7: Format response according to GraphQL specification
    // TODO 8: Record query complexity and execution time metrics
    return true // Continue middleware chain
}

// buildExecutionPlan creates an optimized plan for a GraphQL query
func (p *GraphQLFederationPlugin) buildExecutionPlan(query string) (*ExecutionPlan, error) {
    // TODO 1: Parse GraphQL query into AST
    // TODO 2: Identify field resolutions across services
    // TODO 3: Determine optimal execution order (parallel vs sequential)
    // TODO 4: Identify batching opportunities (DataLoader pattern)
    // TODO 5: Estimate query cost for rate limiting
    // TODO 6: Cache plan for identical future queries
    return nil, nil
}
```

Language-Specific Hints for Extensions:

- **WebSocket in Go:** Use `gorilla/websocket` package which provides robust WebSocket implementation. Remember to call `SetReadDeadline` to prevent hung connections.
- **GraphQL in Go:** `graphql-go/graphql` is the most complete implementation but has API complexity. `99designs/gqlgen` is more modern but requires code generation.
- **WASM in Go:** `wasmer-go` provides good performance and WASI support. Use `syscall/js` for compiling Go to WASM if you want to write plugins in Go.
- **Admin API Security:** Use `crypto/tls` for mTLS on admin endpoints. Implement JWT validation with the same `Authenticator` used for regular requests.

Extension Development Milestone Checkpoints:

For WebSocket extension verification:

```
# Start gateway with WebSocket route configuration
$ go run cmd/gateway/main.go -config config/websocket.yaml

# In another terminal, test WebSocket connection
$ wscat -c ws://localhost:8080/ws/chat
> {"message": "Hello"}
< {"message": "Hello", "timestamp": "2023-10-01T12:00:00Z"}

# Verify in gateway logs
{"level":"info","msg":"WebSocket connection established","client":"127.0.0.1","backend":"chat-service:8080"}
```

For GraphQL federation verification:

```
# Query the GraphQL endpoint
$ curl -X POST http://localhost:8080/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ user(id: \"123\") { name orders { id total } } }"}'

# Expected response structure
{
  "data": {
    "user": {
      "name": "Alice",
      "orders": [
        {"id": "order1", "total": 99.99}
      ]
    }
  },
  "extensions": {
    "requestId": "req_123",
    "backendCalls": 2
  }
}
```

For Admin API verification:

```
# List all routes via Admin API
$ curl -H "Authorization: Bearer $ADMIN_TOKEN" \
  http://localhost:8081/admin/routes

# Create new route
$ curl -X POST -H "Authorization: Bearer $ADMIN_TOKEN" \
  -H "Content-Type: application/json" \
  http://localhost:8081/admin/routes \
  -d '{"id": "new-service", "match": {"path": "/api/new/*"}, "upstreamId": "new-backend"}'

# Verify route is active
$ curl http://localhost:8080/api/new/test
```

Debugging Tips for Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
WebSocket connections drop after 60 seconds	Idle timeout on load balancer or backend	Check gateway and backend keepalive settings, monitor TCP connections with `netstat`	Configure WebSocket ping/pong keepalive, adjust timeouts
GraphQL queries returning partial nulls	Backend service missing required fields	Check GraphQL schema vs actual backend responses, enable debug logging in federation	Update resolver mappings or backend implementations
WASM plugin using excessive memory	Memory not reclaimed between requests	Monitor WASM memory usage with custom metrics, check for cycles in plugin code	Implement memory pooling or instance per request
Admin API changes lost on restart	Changes not persisted to configuration file	Check file permissions, verify write calls succeed, examine config file timestamps	Ensure admin API has write permissions to config directory

These extensions demonstrate how the gateway architecture can evolve to meet sophisticated requirements while maintaining the core principles of modularity, observability, and security established in the main design. Each extension leverages existing components and interfaces, proving the flexibility of the pipeline-based middleware architecture.

Glossary

> **Milestone(s):** This reference section applies to all milestones, providing definitions for technical terms and concepts used throughout the document.

A shared vocabulary is essential for effective collaboration and design. This glossary defines key technical terms, acronyms, and domain-specific vocabulary used throughout this design document. Terms are listed alphabetically for quick reference, with links to the primary sections where they are discussed in detail.

Terms and Definitions

Term	Definition and Context
Active Health Check	A method of monitoring backend health by periodically sending synthetic probe requests (e.g., HTTP GET) to upstream endpoints to verify availability. Contrasts with *Passive Health Check*. Discussed in: **Component: Reverse Proxy & Routing Core** (ADR).
API Gateway	A centralized entry point (a reverse proxy server) that manages, secures, and observes traffic between external clients and internal backend services. It abstracts cross-cutting concerns like authentication, rate limiting, and transformation, functioning as the "front desk" for an API ecosystem. Discussed in: **Context and Problem Statement**, **High-Level Architecture**.
Assembly Line (Mental Model)	An analogy for the gateway's request processing pipeline, where a request moves through a series of specialized "stations" (middleware components) for routing, authentication, transformation, etc., with each station performing its specific task before passing the request along. Discussed in: **High-Level Architecture**, **Interactions and Data Flow**.
Authentication & Authorization Layer	The gateway component responsible for centrally validating client credentials (like JWTs or API keys) and enforcing access policies before allowing requests to reach backend services. Its mental model is *The Bouncer with a Guest List*. Discussed in: **Component: Authentication & Authorization Layer**.
Backend-for-Frontend (BFF)	An architectural pattern where a separate backend service (or a gateway with transformation logic) is tailored to the specific needs of a particular client type (e.g., mobile app, web app). The API Gateway can implement BFF patterns via request/response transformation. Discussed in: **Context and Problem Statement**, **Component: Request/Response Transformation**.
Black Box Flight Recorder (Mental Model)	An analogy for the observability system, which, like an airplane's black box, instruments every operation (logs, metrics, traces) to provide a complete record for understanding system health and diagnosing failures post-incident. Discussed in: **Component: Observability & Plugin System**.
BodySizeError	A custom error type (`BodySizeError`) returned when a request or response body exceeds the configured size limit during buffered transformation. It contains fields `Max` and `Actual`. Discussed in: **Component: Request/Response Transformation** (Implementation Guidance).
Buffer	A temporary area in memory (or disk) used to hold the complete contents of an HTTP request or response body during transformation operations, allowing full read/modify/write cycles. Discussed in: **Component: Request/Response Transformation** (ADR).
Circuit Breaker	A resilience pattern (implemented by the `CircuitBreaker` struct) that stops sending requests to a failing backend endpoint after a threshold of consecutive failures, allowing it time to recover. It exists in three states: `StateClosed` , `StateOpen` , `StateHalfOpen` . Discussed in: **Component: Reverse Proxy & Routing Core**.
Circuit Breaking	The process implemented by a *Circuit Breaker* to fail fast and prevent cascading failures by temporarily isolating unhealthy backend services. Discussed in: **Component: Reverse Proxy & Routing Core**, **Error Handling and Edge Cases**.
Config	The root configuration structure (`Config`) loaded from file, containing the gateway's listening address, all defined routes (`Routes`), and upstream service definitions (`Upstreams`). Discussed in: **Data Model**.
Configuration Atomicity	The property that updates to multiple related configuration resources (e.g., a route and its plugins) are applied together—all succeed or all fail—preventing the system from entering a partially updated, inconsistent state. Discussed in: **Future Extensions**.
Configuration Reload Races	A concurrency issue where a request is processed using a mix of old and new configuration values if a reload occurs mid-request, potentially causing inconsistent behavior. Mitigated using copy-on-write techniques. Discussed in: **Error Handling and Edge Cases**.
Concierge Pattern (Mental Model)	The primary analogy for the API Gateway, comparing it to a hotel concierge who receives all guest requests, directs them to the appropriate service (housekeeping, restaurant, tour desk), and may augment them (adding special instructions, translating language). Discussed in: **Context and Problem Statement**.
Content-Length	An HTTP header indicating the size, in bytes, of the entity-body being transmitted. The gateway must correctly update this header after modifying a request or response body. Discussed in: **Component: Request/Response Transformation** (Common Pitfalls).
Content-Type	An HTTP header indicating the media type (e.g., `application/json`) of the request or response body. The gateway uses this to determine if transformation (e.g., JSON parsing) should be applied. Discussed in: **Component: Request/Response Transformation**.
Copy-on-Write	A technique where a new, updated version of a configuration or data structure is built in memory and then atomically swapped with the old version using a pointer, avoiding locks and ensuring consistency. Discussed in: **Error Handling and Edge Cases**.
Cross-cutting Concerns	Functionality that is common to many parts of a system (like logging, authentication, or rate limiting) and would otherwise be duplicated across individual services. The API Gateway centralizes these concerns. Discussed in: **Context and Problem Statement**, **Goals and Non-Goals**.
Director Function	In Go's `httputil.ReverseProxy` , a callback function (the `Director`) that is invoked to modify the outgoing request to the backend (e.g., setting the target URL, adding headers). The gateway's router implements this function. Discussed in: **Component: Reverse Proxy & Routing Core** (Implementation Guidance).
East-West Traffic	Network traffic that flows between internal services within a data center or private network. The API Gateway primarily handles *North-South Traffic*, though it may proxy between internal services. Discussed in: **Context and Problem Statement**.

Term	Definition and Context
Emergency Stop Chain (Mental Model)	An analogy for the error handling pipeline, where a serious error (like an authentication failure) triggers an immediate "emergency stop," short-circuiting the remaining middleware and returning an error response directly. Discussed in: **Interactions and Data Flow**, **Error Handling and Edge Cases**.
Endpoint	A struct (`Endpoint`) representing a single, reachable instance of a backend service, defined by its `ID` and `URL` . Multiple endpoints belong to an `Upstream` for load balancing. Discussed in: **Data Model**, **Component: Reverse Proxy & Routing Core**.
Exponential Backoff	A retry strategy where the delay between consecutive retry attempts increases exponentially (e.g., 1s, 2s, 4s, 8s). Used in health checks and circuit breaker probes to avoid overwhelming a recovering backend. Discussed in: **Error Handling and Edge Cases**.
Fallback Responses	Pre-configured static HTTP responses (e.g., a default error message or cached data) returned by the gateway when a backend service is unavailable (circuit breaker open) or times out, enabling graceful degradation. Discussed in: **Error Handling and Edge Cases**.
FieldMapping	A struct (`FieldMapping`) defining a single transformation rule for JSON bodies, specifying a source field (`From`) and a destination field (`To`). Part of `JSONTransformOps` . Discussed in: **Data Model**, **Component: Request/Response Transformation**.
Five Layers	The logical decomposition of the gateway's request processing pipeline: **Ingress** (receiving requests), **Pre-Processing** (auth, rate limiting), **Core** (routing, load balancing), **Post-Processing** (transformation), and **Egress** (proxying to backend). Discussed in: **High-Level Architecture**.
GatewayProxy	A wrapper struct (`GatewayProxy`) around Go's `httputil.ReverseProxy` that integrates the gateway's custom routing, load balancing, and circuit breaking logic via its `Director` and `ModifyResponse` callbacks. Discussed in: **Component: Reverse Proxy & Routing Core** (Implementation Guidance).
Graceful Degradation	A design strategy where the system systematically reduces functionality (e.g., by returning fallback responses or disabling non-essential features) to maintain overall availability during partial failures. Discussed in: **Error Handling and Edge Cases**.
GraphQL Federation	An advanced capability where the gateway composes a unified GraphQL schema from multiple backend GraphQL services, requiring query planning and response aggregation. A potential future extension. Discussed in: **Future Extensions**.
Header Manipulation	The process of adding, removing, or modifying HTTP headers on requests (before sending to backend) and responses (before returning to client). Configured via `HeaderTransformerConfig` and `HeaderOps` . Discussed in: **Component: Request/Response Transformation**.
HeaderOps	A configuration struct (`HeaderOps`) specifying header operations: `Add` (map of headers to add), `Remove` (list of header names), and `Rewrite` (map of old header names to new names). Discussed in: **Data Model**, **Component: Request/Response Transformation**.
HeaderTransformer	A middleware component (`HeaderTransformer`) that applies header manipulation rules based on its `HeaderTransformerConfig` . Discussed in: **Component: Request/Response Transformation** (Implementation Guidance).
High-Cardinality	A characteristic of a metrics label or log field that can have a very large number of unique values (e.g., `user_id` , `request_id`). Using high-cardinality dimensions in metrics can overwhelm monitoring systems. Discussed in: **Component: Observability & Plugin System** (Common Pitfalls).
Idempotent	A property of an HTTP method (like GET, PUT, DELETE) where making the same request multiple times produces the same result as making it once. This property influences retry safety. Discussed in: **Error Handling and Edge Cases**.
JSON Transformation	The modification of JSON-encoded request or response bodies, including operations like field renaming (via `FieldMapping`), field addition, and field removal. Configured via `JSONTransformConfig` . Discussed in: **Component: Request/Response Transformation**.
JSONTransformer	A middleware component (`JSONTransformer`) that applies JSON body transformation rules, respecting a configured `maxBodySize` limit to prevent excessive memory consumption. Discussed in: **Component: Request/Response Transformation** (Implementation Guidance).
JWT (JSON Web Token)	A compact, URL-safe token format (RFC 7519) used for authentication and authorization. Typically contains claims about the user (like `sub` , `exp`) and is signed. The gateway validates JWTs using the `JWTValidationRule` configuration. Discussed in: **Component: Authentication & Authorization Layer**.
LimitedReadCloser	A helper type (`LimitedReadCloser`) that wraps an `io.Reader` and `io.Closer` to enforce a maximum read limit, used to safely read HTTP bodies without risking memory exhaustion. Discussed in: **Component: Request/Response Transformation** (Implementation Guidance).
Load Balancing	The distribution of incoming requests across multiple backend `Endpoint` instances belonging to an `Upstream` . The gateway implements strategies like round-robin (via `RoundRobinLoadBalancer`). Discussed in: **Component: Reverse Proxy & Routing Core**.
Metrics	Quantitative measurements of system behavior over time (e.g., request count, latency, error rate), exposed in a format like Prometheus for monitoring and alerting. Collected by plugins like `MetricsPlugin` . Discussed in: **Component: Observability & Plugin System**.
Microservices	An architectural style where a single application is composed of many loosely coupled, independently deployable services. The API Gateway provides a unified entry point for a microservices ecosystem. Discussed in: **Context and Problem Statement**.

Term	Definition and Context
Middleware	A software component (implementing the `Middleware` interface) that processes HTTP requests and responses in a chain (`Chain`). Each middleware performs a specific function (e.g., logging, authentication, transformation). Discussed in: **High-Level Architecture**, **Component: Observability & Plugin System**.
North-South Traffic	Network traffic that flows between external clients (outside the data center) and internal services. This is the primary traffic pattern handled by the API Gateway. Discussed in: **Context and Problem Statement**.
Observability	The ability to understand a system's internal state through its external outputs: logs, metrics, and traces. A core requirement for operating the gateway in production. Discussed in: **Component: Observability & Plugin System**.
Off-the-Shelf	Referring to pre-built, commercially available or open-source software solutions (like Kong or NGINX) that can be configured to serve as an API Gateway, as opposed to building a custom one. Discussed in: **Context and Problem Statement** (Comparison Table).
Passive Health Check	A method of monitoring backend health by observing the success or failure of real client requests (e.g., counting 5xx errors). Contrasts with *Active Health Check*. The gateway's default approach. Discussed in: **Component: Reverse Proxy & Routing Core** (ADR).
PII (Personally Identifiable Information)	Sensitive data that can be used to identify an individual (e.g., email, national ID number). The gateway must avoid logging or exposing PII, a common pitfall in observability components. Discussed in: **Component: Observability & Plugin System** (Common Pitfalls).
Pipeline	A struct (`Pipeline`) that organizes middleware into three ordered groups: `preProcess` , `coreProcess` , and `postProcess` , corresponding to the *Five Layers*. Its `Execute` method runs a request through the entire chain. Discussed in: **Data Model**, **High-Level Architecture**.
Pipeline Pattern	The architectural pattern where requests flow through a series of processing stages (the pipeline), with each stage performing a specific transformation or check. The foundation of the gateway's design. Discussed in: **High-Level Architecture**.
Plugin	A self-contained module (implementing the `Plugin` interface) that extends gateway functionality (e.g., rate limiting, custom logging). Plugins are registered with the `PluginManager` and configured via `PluginConfig` . Discussed in: **Component: Observability & Plugin System**.
Plugin Chain	An ordered sequence of plugins (a `Chain`) executed during request processing. The order is determined by each plugin's `Priority()` method. Discussed in: **Component: Observability & Plugin System**.
PluginConfig	A configuration struct (`PluginConfig`) that defines an instance of a plugin to be attached to a route, containing the plugin's `Name` and its specific `Config` (a map of settings). Discussed in: **Data Model**.
PluginManager	A singleton component (`PluginManager`) responsible for registering plugin factories (`PluginFactory`) and instantiating configured plugins. It maintains a `registry` of available plugins. Discussed in: **Component: Observability & Plugin System**.
Radix Tree	A compressed prefix tree data structure used for efficient path matching in the router. It allows for fast lookup of routes based on URL paths, supporting both static and parameterized segments. Discussed in: **Component: Reverse Proxy & Routing Core** (ADR).
Request Aggregation	The process of combining responses from multiple backend service calls into a single, unified response returned to the client. An advanced transformation feature. Discussed in: **Component: Request/Response Transformation**.
RequestContext	A central struct (`RequestContext`) that carries state, authentication results, transformation data, and logging fields through the entire request processing pipeline. It is created per request and passed to each middleware. Discussed in: **Data Model**, **High-Level Architecture**.
RequestTemplateData	A helper struct (`RequestTemplateData`) containing common request properties (like `ClientIP` , `UserID`) used to populate dynamic values in header manipulation templates. Discussed in: **Component: Request/Response Transformation** (Implementation Guidance).
Reverse Proxy	A server that sits between clients and backend services, forwarding client requests to the appropriate backend and returning the backend's response to the client. This is the foundational role of the API Gateway. Discussed in: **Component: Reverse Proxy & Routing Core**.
Route	A configuration struct (`Route`) defining how to match incoming requests (`RouteMatch`) and where to send them (`UpstreamID`), along with attached plugins (`Plugins`). Discussed in: **Data Model**, **Component: Reverse Proxy & Routing Core**.
RouteMatch	A struct (`RouteMatch`) specifying the conditions for a request to match a route: `Path` (e.g., `/api/users/*`), `Method` (e.g., `GET`), and `Headers` (a map of header key-value pairs). Discussed in: **Data Model**, **Component: Reverse Proxy & Routing Core**.
RouterImpl	The primary implementation of the routing logic (`RouterImpl`), storing `routes` and `upstreams` in maps and using a `radix.Tree` for efficient path matching. Discussed in: **Component: Reverse Proxy & Routing Core** (Implementation Guidance).
Scope Creep	The uncontrolled expansion of a project's goals and features beyond the original plan. The *Non-Goals* section explicitly guards against this by defining what the gateway will not do. Discussed in: **Goals and Non-Goals**.
Short-Circuit	The early termination of pipeline processing, typically due to an error (like authentication failure or rate limit exceeded), which bypasses subsequent middleware and returns an immediate HTTP error response. Discussed in: **Interactions and Data Flow**, **Error Handling and Edge Cases**.

Term	Definition and Context
Stateless	A design principle where the gateway does not maintain persistent session state between requests; each request is processed independently based on its contents and configuration. Discussed in: **Goals and Non-Goals**.
Streaming	A processing model where data (like an HTTP body) is processed in chunks as it flows through the system, without being fully buffered in memory. Contrasts with *Buffered* transformation. Discussed in: **Component: Request/Response Transformation** (ADR).
Structured Logging	Logging that emits events in a machine-readable format (typically JSON) with consistent, named fields (like `request_id`, `duration_ms`), as opposed to plain text. Facilitates log aggregation and analysis. Discussed in: **Component: Observability & Plugin System**.
Tracing	Recording the flow of a single request (a trace) as it propagates through the distributed components of a system (gateway, backend services), composed of individual spans for each operation. Implemented via `TracingPlugin`. Discussed in: **Component: Observability & Plugin System**.
Triage Nurse (Mental Model)	An analogy for the error categorization logic (`CategorizeError`), which examines errors and assigns them to buckets (client, backend, gateway) to determine the appropriate handling strategy, much like a nurse prioritizes patients. Discussed in: **Error Handling and Edge Cases**.
Upstream	A configuration struct (`Upstream`) representing a logical backend service, containing a list of physical `Endpoint` instances where traffic can be load balanced. A `Route` points to an `Upstream`. Discussed in: **Data Model**, **Component: Reverse Proxy & Routing Core**.
URL Rewriting	The modification of the request path and query parameters before the request is forwarded to the backend service. For example, rewriting `/api/v1/users` to `/users` on the backend. Discussed in: **Component: Request/Response Transformation**.
WASI Sandbox	WebAssembly System Interface, a standard providing WebAssembly modules with controlled access to system resources like filesystem and network. A potential future extension for safely running untrusted plugin code. Discussed in: **Future Extensions**.
WASM Plugin Runtime	An extension to the plugin system that allows loading and executing plugins compiled to WebAssembly (WASM), providing strong isolation and language agnosticism. Discussed in: **Future Extensions**.
WebSocket Proxying	The ability of the gateway to handle long-lived, bidirectional WebSocket connections, forwarding frames between client and backend. A potential future extension implemented by `WebSocketProxy`. Discussed in: **Future Extensions**.
X-Forwarded-For	A standard HTTP header (`X-Forwarded-For`) that proxies use to pass the original client IP address through a chain of proxies to the backend service. The gateway must append to (not overwrite) this header. Discussed in: **Component: Reverse Proxy & Routing Core** (Common Pitfalls).