

# Feature Flag System: Design Document

## Overview

A feature flag system that enables controlled feature rollouts, A/B testing, and real-time configuration updates. The key architectural challenge is building a low-latency evaluation engine that can process complex targeting rules while maintaining consistency across distributed clients and supporting real-time updates without overwhelming the infrastructure.

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

## Context and Problem Statement

**Milestone(s):** This section provides foundational context for all three milestones by establishing why feature flags are essential and the unique challenges they address.

### Mental Model: Software Release as Air Traffic Control

Think of modern software deployment as managing air traffic at a busy international airport. Without proper coordination, you have hundreds of flights trying to land simultaneously on the same runway—a recipe for disaster. Traditional software releases work exactly like this chaotic scenario: all features launch together at a predetermined time, with no ability to adjust course once the "planes are in the air."

**Feature flags transform your release process into a sophisticated air traffic control system.** Just as air traffic controllers can direct specific flights to different runways, adjust landing sequences based on weather conditions, and even redirect planes to alternate airports when problems arise, feature flags give you granular control over which users see which features and when.

Consider how air traffic control handles a busy evening rush. Controllers don't just flip a switch and let all planes land at once. Instead, they carefully orchestrate arrivals: "Flight 447, you're cleared for runway 24L. Flight 852, hold at 5,000 feet until weather clears. Flight 213, divert to the cargo runway for your oversized load." This is precisely what feature flags enable for software features—you can route different user "flights" to different feature "runways" based on their characteristics, current system conditions, and business requirements.

The air traffic control analogy extends beautifully to the technical challenges we face. Air traffic controllers need real-time weather data, aircraft positions, and runway status to make split-second decisions. Similarly, our feature flag system needs real-time user context, system health metrics, and business KPIs to make

intelligent routing decisions. When a storm hits the airport, controllers can quickly reroute traffic to alternate runways. When a feature causes performance issues, feature flags let you instantly reroute users to the stable code path without any deployments or restarts.

**Key Insight:** Just as air traffic control systems must never fail (planes can't just "hover indefinitely"), feature flag systems require exceptional reliability with graceful degradation strategies. A feature flag outage shouldn't ground your entire application.

The complexity emerges when you realize that modern applications aren't managing just one airport—they're coordinating a global network of airports (distributed systems) where decisions made at one location affect traffic patterns everywhere else. A feature flag change in your recommendation engine doesn't just affect the users who see new recommendations; it impacts database load, cache hit rates, API response times, and downstream service capacity across your entire infrastructure.

This mental model helps us understand why building a feature flag system is more challenging than it initially appears. It's not enough to have a simple on/off switch. You need the equivalent of sophisticated radar systems (real-time monitoring), weather prediction models (A/B testing and analytics), emergency protocols (fallback strategies), and a control tower that never sleeps (24/7 reliability with real-time updates).

## Existing Approaches Comparison

Before diving into our comprehensive solution, let's examine how teams typically handle feature toggles today and why each approach breaks down as systems scale. Understanding these limitations helps justify the complexity we're about to introduce.

### Configuration Files Approach

The simplest approach stores feature toggles in configuration files (YAML, JSON, or properties files) that are read at application startup. This feels natural because it leverages existing configuration management practices that most teams already understand.

Aspect	Description	Limitations
<b>Implementation</b>	JSON/YAML files with boolean flags loaded at startup	Requires application restart for any flag change
<b>Deployment</b>	Flags bundled with application code in the same deployment artifact	Cannot change flags without full deployment pipeline
<b>Targeting</b>	Single global value per flag, no user-specific targeting	All users see the same flag state simultaneously
<b>Rollback Speed</b>	10-30 minutes (full deployment cycle)	Too slow for emergency feature disabling
<b>Consistency</b>	Eventually consistent after all instances restart	Inconsistent during rolling deployments
<b>A/B Testing</b>	Not supported without custom implementation	No statistical rigor or experiment tracking

The configuration file approach works well for simple environment-specific settings (database URLs, API timeouts) but breaks down catastrophically for feature management. Imagine discovering a critical bug in your new checkout flow during Black Friday traffic. With configuration files, you're looking at a 20-minute deployment cycle to disable the feature—an eternity when you're losing thousands of dollars per minute.

**Real-world failure scenario:** A major e-commerce platform used configuration files for feature flags. During a flash sale, their new inventory calculation feature caused database deadlocks. The 15-minute rollback time resulted in \$2.3 million in lost sales and required manually switching to backup servers—a problem that feature flags with instant rollback could have resolved in under 30 seconds.

## Environment Variables Approach

Environment variables offer slightly more flexibility than configuration files, especially in containerized environments where you can restart individual service instances without full redeployments.

Aspect	Description	Limitations
<b>Implementation</b>	Feature flags stored as environment variables (FEATURE_X=true)	Still requires container/process restart for changes
<b>Deployment</b>	Can be changed via orchestration platforms (Kubernetes, Docker Compose)	Rolling restart still takes several minutes
<b>Targeting</b>	Global per environment, potentially per service instance	No user segmentation or gradual rollouts
<b>Observability</b>	Difficult to track which features are active across the fleet	No centralized view of flag states
<b>Type Safety</b>	String-based with manual parsing prone to errors	"true", "TRUE", "1", "yes" all need handling
<b>Audit Trail</b>	Limited to orchestration platform logs	No feature-specific change history

Environment variables solve the bundling problem of configuration files but introduce new operational complexities. Managing dozens of feature flags across multiple services and environments quickly becomes unwieldy. Consider a microservices architecture with 20 services, each having 5-10 feature flags across development, staging, and production environments—you're now managing 300-600 individual environment variables with no central visibility.

**Real-world failure scenario:** A fintech startup used environment variables for feature flags across their microservices. During a gradual rollout of a new fraud detection algorithm, they discovered the flag was accidentally enabled in production but disabled in staging—the opposite of their intent. The mismatch went undetected for three days, during which time their testing was validating the old algorithm while the new (buggy) algorithm was processing live transactions, resulting in 12% false positives and hundreds of customer complaints.

## Database Toggle Approach

Many teams evolve from static configuration to storing feature flags in their application database, often starting with a simple `feature_flags` table with name and enabled columns. This approach provides runtime configurability without deployment cycles.

Aspect	Description	Limitations
<b>Implementation</b>	Database table storing flag names, enabled state, optional targeting rules	Database becomes single point of failure for all features
<b>Real-time Updates</b>	Requires polling or cache invalidation strategy	Polling creates database load; caching creates staleness
<b>Performance</b>	Database query on every flag evaluation	Adds latency and load to primary database
<b>Targeting</b>	Can store complex rules but evaluation logic mixed with business logic	Rule evaluation code scattered across application
<b>Consistency</b>	Depends on database consistency model	Read replicas can serve stale flag states
<b>Fallback</b>	Application breaks if database unavailable	No graceful degradation during database outages

The database approach initially feels like a natural progression, but it introduces a fundamental architectural problem: your feature management system now shares fate with your primary application database. When your database is under heavy load or experiencing issues, your ability to disable problematic features disappears exactly when you need it most.

**Performance implications:** Every flag evaluation requires a database query or cache lookup. For a high-traffic application evaluating flags on every request, this can easily add 50-100 million additional database queries per day. The irony is that the performance impact of your flagging system can become worse than the features you're trying to protect.

**Real-world failure scenario:** A social media platform stored feature flags in their primary PostgreSQL database. During a traffic spike, database connections were exhausted, causing not only user-facing features to fail but also preventing them from disabling the problematic new timeline algorithm that was causing the load spike in the first place. The circular dependency meant their feature flags couldn't help them recover from the very problems they were designed to solve.

## Dedicated Feature Flag Services

Recognizing the limitations of the previous approaches, many organizations turn to dedicated feature flag platforms (LaunchDarkly, Unleash, Flagsmith) or build custom services specifically for flag management.

Aspect	Description	Benefits
<b>Implementation</b>	Standalone service with API for flag evaluation and management	Purpose-built for feature flag requirements
<b>Real-time Updates</b>	WebSocket or SSE connections for instant flag changes	Sub-second flag propagation across all clients
<b>Targeting</b>	Sophisticated rule engines with user attributes, segments, and percentages	Gradual rollouts and A/B testing built-in
<b>Performance</b>	Client-side SDKs with local caching and background sync	Flag evaluation in microseconds with fallbacks
<b>Observability</b>	Rich analytics, audit logs, and experiment tracking	Data-driven feature decisions
<b>Reliability</b>	Designed for high availability with graceful degradation	Feature flags available even during service outages

However, dedicated services introduce their own complexities and architectural decisions. The fundamental challenge is balancing the richness of targeting capabilities with the performance and reliability requirements of flag evaluation.

### Architecture Decision: Build vs Buy for Feature Flags

- **Context:** Teams must decide whether to use existing feature flag services or build custom solutions
- **Options Considered:**
  - **Third-party SaaS:** LaunchDarkly, Split, Optimizely
  - **Open-source hosted:** Unleash, Flagsmith, GrowthBook
  - **Custom-built:** Internal feature flag service
- **Decision:** Build custom for learning purposes, but understand trade-offs
- **Rationale:** Commercial services offer mature ecosystems but limit customization and create vendor dependency. Building custom provides deep understanding of the technical challenges and architectural decisions involved.
- **Consequences:** Higher development effort but complete control over performance characteristics, data residency, and integration patterns.

### Comparison of dedicated service approaches:

Approach	Pros	Cons	Best For
<b>Third-party SaaS</b>	Mature features, no operational overhead, proven reliability	Vendor lock-in, data privacy concerns, recurring costs	Teams wanting to focus on core business features
<b>Open-source hosted</b>	Full control, customizable, community support	Operational overhead, need internal expertise	Teams with strong infrastructure capabilities
<b>Custom-built</b>	Complete customization, no vendor dependency, cost control	High development effort, need to solve hard problems	Organizations with unique requirements or learning goals

The key insight from this comparison is that feature flags are deceptively complex. What starts as a simple boolean toggle quickly evolves into a distributed system with requirements for real-time updates, complex targeting logic, statistical analysis, and fault tolerance. This complexity explains why dedicated solutions exist and why building a robust feature flag system is a substantial engineering undertaking.

#### Common anti-patterns observed across all approaches:

**⚠ Pitfall: Flag Evaluation in Critical Path** Many implementations put flag evaluation directly in their request processing path without proper caching or fallbacks. This creates latency and reliability problems where feature management becomes a bottleneck for core functionality. The fix is to always evaluate flags asynchronously with local caching and sensible defaults.

**⚠ Pitfall: No Flag Lifecycle Management** Teams often create feature flags without establishing processes for flag cleanup and removal. This leads to "flag debt" where codebases accumulate hundreds of obsolete flags that nobody dares to remove. The fix is to treat flags as temporary by default with expiration dates and regular cleanup processes.

**⚠ Pitfall: Inconsistent Flag Evaluation** Without proper user identification and consistent hashing, users can flip-flop between different variants of a feature, creating confusing experiences and invalidating A/B test results. The fix is to ensure deterministic flag evaluation based on stable user identifiers.

Understanding these existing approaches and their failure modes helps us appreciate why building a robust feature flag system requires careful attention to performance, reliability, and consistency challenges. Our design must address the fundamental problems that cause each approach to break down while providing the sophisticated targeting and real-time capabilities that modern applications demand.

The remainder of this design document will show how to build a feature flag system that combines the best aspects of each approach while avoiding their critical failure modes. We'll focus particularly on the three core challenges that distinguish robust feature flag systems from naive implementations: consistent flag evaluation under high load, real-time flag updates without infrastructure overload, and statistical rigor for A/B testing and gradual rollouts.

## Implementation Guidance

This foundational section helps establish the technology choices and project structure that will support the sophisticated feature flag system we're building. Understanding these early decisions will inform how we approach the more complex challenges in subsequent milestones.

## Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Framework	Standard library <code>net/http</code> with JSON	Gin or Echo framework with middleware
Database	SQLite for development, PostgreSQL for production	PostgreSQL with Redis for caching
Real-time Updates	Server-Sent Events with <code>net/http</code>	WebSocket with Gorilla WebSocket library
User Context	Simple JSON structures	Protocol Buffers for type safety
Configuration	YAML with <code>gopkg.in/yaml.v3</code>	Viper for hierarchical configuration
Logging	Standard library <code>log/slog</code>	Structured logging with <code>logrus</code> or <code>zap</code>
Testing	Standard library <code>testing</code>	Testify for assertions, Ginkgo for BDD

### Language-Specific Considerations for Go:

- **Consistent Hashing:** Use `hash/fnv` for fast, stable hashing of user identifiers
- **JSON Handling:** Leverage `encoding/json` for flag definitions and user context
- **Concurrency:** Utilize Go's goroutines for handling real-time connections
- **Error Handling:** Embrace explicit error returns for robust flag evaluation
- **Interface Design:** Use interfaces to support multiple storage backends and update mechanisms

## Recommended Project Structure

Understanding how to organize a feature flag system helps avoid common architectural mistakes where flag evaluation logic gets scattered throughout the application.

```
feature-flag-system/
├── cmd/
│   ├── flagserver/           ← HTTP API server for flag management
│   │   └── main.go
│   ├── flagproxy/           ← Lightweight evaluation proxy
│   │   └── main.go
│   └── migrator/            ← Database migration tool
│       └── main.go
└── internal/
    ├── core/                ← Core domain logic (no external dependencies)
    │   ├── flag.go             ← Flag entity and business logic
    │   ├── evaluation.go       ← Evaluation engine
    │   ├── targeting.go        ← User targeting and segmentation
    │   └── experiment.go       ← A/B testing logic
    ├── storage/              ← Data persistence layer
    │   ├── interface.go        ← Storage abstraction
    │   ├── postgres/           ← PostgreSQL implementation
    │   └── memory/             ← In-memory for testing
    ├── realtime/              ← Real-time update service
    │   ├── sse.go               ← Server-Sent Events implementation
    │   ├── websocket.go         ← WebSocket implementation
    │   └── broadcaster.go       ← Update broadcasting logic
    ├── analytics/             ← Flag evaluation tracking and A/B analysis
    │   ├── collector.go         ← Event collection
    │   ├── aggregator.go        ← Metrics aggregation
    │   └── stats.go              ← Statistical analysis
    └── api/
        ├── handlers/           ← HTTP request handlers
        ├── middleware/          ← Authentication, logging, CORS
        └── routes.go              ← Route definitions
    └── pkg/
        ├── client/              ← Public APIs (can be imported by other projects)
        │   ├── sdk.go              ← Go SDK for consuming feature flags
        │   ├── cache.go            ← Main SDK interface
        │   └── streaming.go         ← Local flag caching
        └── types/
            ├── flag.go             ← Real-time update handling
            ├── context.go           ← Shared types and interfaces
            └── events.go             ← Flag data structures
            └── user_context.go        ← User context types
            └── event_and_analytics.go  ← Event and analytics types
    └── web/
        ├── static/               ← Web dashboard (optional)
        └── templates/             ← CSS, JavaScript assets
        └── migrations/            ← HTML templates
            ├── 001_initial.sql
            ├── 002_add_experiments.sql
            └── 003_add_analytics.sql
    └── docs/                  ← Additional documentation
        ├── api.md                ← API documentation
        ├── sdk.md                ← SDK usage guide
        └── deployment.md          ← Deployment instructions
    └── docker-compose.yml      ← Local development environment
```

## Key organizational principles:

1. **Domain-Driven Structure:** The `internal/core` package contains pure business logic with no external dependencies, making it easy to unit test and reason about.
2. **Interface Segregation:** Each major component (`storage`, `realtime`, `analytics`) defines its own interfaces, allowing for multiple implementations and easy testing with mocks.
3. **Dependency Direction:** Dependencies flow inward toward the core domain logic. Storage implementations depend on core interfaces, not vice versa.
4. **Client SDK Separation:** The `pkg/client` provides a clean API for applications consuming feature flags, with its own caching and connection management logic separate from the server implementation.

## Core Type Definitions

These fundamental types will be referenced throughout the implementation. Understanding them now will help you follow the more complex algorithms in later sections.

GO

```
// Package: pkg/types

// FlagKey uniquely identifies a feature flag across the system

type FlagKey string

// UserID represents a stable identifier for users in targeting decisions

type UserID string

// Variant represents a specific variation of a feature flag

type Variant struct {

    Key      string          `json:"key"`

    Value    interface{}     `json:"value"`

    Weight   int             `json:"weight"` // For percentage rollouts

}

// UserContext contains attributes used for flag targeting decisions

type UserContext struct {

    UserID     UserID         `json:"user_id"`

    Attributes map[string]interface{} `json:"attributes"`

    Segments   []string       `json:"segments"`

}

// EvaluationResult contains the result of flag evaluation plus debugging info

type EvaluationResult struct {

    FlagKey   FlagKey        `json:"flag_key"`

    Value     interface{}    `json:"value"`

    Variant   string         `json:"variant"`

    Reason    string         `json:"reason"` // Why this variant was chosen

    Source    string         `json:"source"` // "cache", "storage", "default"
```

```
}
```

## Infrastructure Starter Code

This complete HTTP server foundation handles the basic web serving infrastructure so you can focus on the core feature flag logic:

GO

```
// cmd/flagserver/main.go

package main

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

    "github.com/your-org/feature-flags/internal/api"
    "github.com/your-org/feature-flags/internal/storage/postgres"
)

func main() {
    // Initialize structured logging
    logger := slog.New(slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
        Level: slog.LevelInfo,
    }))
}

// Initialize storage layer
storage, err := postgres.NewStorage("postgres://localhost/flagdb?sslmode=disable")
if err != nil {
    logger.Error("Failed to initialize storage", "error", err)
    os.Exit(1)
}
```

```
defer storage.Close()

// Initialize API server

server := api.NewServer(storage, logger)

// Configure HTTP server with timeouts

httpServer := &http.Server{
    Addr:          ":8080",
    Handler:       server.Routes(),
    ReadTimeout:   15 * time.Second,
    WriteTimeout:  15 * time.Second,
    IdleTimeout:   60 * time.Second,
}

// Start server in background

go func() {
    logger.Info("Starting flag server", "addr", httpServer.Addr)
    if err := httpServer.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        logger.Error("HTTP server failed", "error", err)
        os.Exit(1)
    }
}()

// Wait for shutdown signal

stop := make(chan os.Signal, 1)
signal.Notify(stop, syscall.SIGINT, syscall.SIGTERM)

<-stop
```

```
// Graceful shutdown

logger.Info("Shutting down server...")

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

defer cancel()

if err := httpServerShutdown(ctx); err != nil {

    logger.Error("Server shutdown failed", "error", err)

} else {

    logger.Info("Server shut down successfully")

}

}
```

## Development Environment Setup

Complete Docker Compose configuration for local development with all required dependencies:

YAML

```
# docker-compose.yml

version: '3.8'

services:

  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: flagdb
      POSTGRES_USER: flaguser
      POSTGRES_PASSWORD: flagpass
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./migrations:/docker-entrypoint-initdb.d

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes
    volumes:
      - redis_data:/data

  flagserver:
    build: .
    ports:
      - "8080:8080"
```

```
environment:  
  
  DATABASE_URL: postgres://flaguser:flagpass@postgres:5432/flagdb?sslmode=disable  
  
  REDIS_URL: redis://redis:6379/0  
  
depends_on:  
  
  - postgres  
  
  - redis  
  
volumes:  
  
  - .:/app  
  
working_dir: /app  
  
volumes:  
  
  postgres_data:  
  
  redis_data:
```

## Milestone Checkpoint for Foundation

After setting up the project structure and basic infrastructure:

### What to verify:

1. Run `go mod init github.com/your-org/feature-flags` to initialize the Go module
2. Set up the directory structure as shown above
3. Copy the starter code into the appropriate files
4. Run `docker-compose up postgres redis` to start dependencies
5. Run `go run cmd/flagserver/main.go` - server should start without errors
6. Visit `http://localhost:8080/health` (you'll implement this endpoint) - should return 200 OK

### Expected behavior:

- Server starts and listens on port 8080
- Database connection succeeds (you'll see connection logs)
- Graceful shutdown works with Ctrl+C
- Project structure follows Go conventions with clear separation of concerns

### Signs something is wrong:

- Import cycle errors indicate incorrect dependency structure

- Database connection failures suggest configuration issues
- Server doesn't respond to shutdown signals cleanly

This foundation provides the scaffolding for implementing the three core milestones: the flag evaluation engine, real-time updates, and analytics. Each subsequent section will build upon this structure with specific components and algorithms needed for a production-quality feature flag system.

## Goals and Non-Goals

**Milestone(s):** This section establishes the scope and boundaries for all three milestones, defining what functionality the Flag Evaluation Engine, Real-time Flag Updates, and Flag Analytics & Experiments will and will not include.

### Mental Model: Air Traffic Control Tower Responsibilities

Think of defining goals and non-goals for a feature flag system like establishing the responsibilities of an **air traffic control tower**. The tower has clear, well-defined duties: it guides aircraft safely to their destinations, manages takeoffs and landings, and coordinates flight paths to prevent collisions. However, the tower doesn't design aircraft engines, forecast weather patterns weeks in advance, or handle passenger boarding processes.

Similarly, our feature flag system must have crystal-clear boundaries. It will excel at its core mission—safely routing users to different feature experiences based on targeting rules—while explicitly avoiding responsibilities that belong to other systems. Just as air traffic controllers need reliable radar, communication systems, and weather updates to make split-second decisions, our feature flag system needs robust evaluation logic, real-time updates, and comprehensive analytics to guide software releases safely.

The key insight is that **scope creep** in feature flag systems is like asking air traffic controllers to also handle airline scheduling, aircraft maintenance, and passenger services. Each additional responsibility increases complexity exponentially and threatens the reliability of the core mission. By establishing clear goals and non-goals upfront, we create focused, maintainable software that excels at its intended purpose.

### Primary Goals

Our feature flag system will deliver four core capabilities that directly support the air traffic control analogy of coordinating software releases safely and efficiently.

#### Goal 1: Reliable Flag Evaluation with Complex Targeting

The system will provide **deterministic, low-latency flag evaluation** that consistently assigns users to the correct variants based on sophisticated targeting rules. This is the fundamental responsibility—like air traffic control's primary duty of guiding aircraft to safe landing strips.

Capability	Description	Success Criteria
<b>Multi-type Flag Support</b>	Boolean, string, number, and JSON flag values	All primitive types plus structured data variants
<b>Complex Rule Logic</b>	AND/OR conditions with multiple user attributes	Nested boolean expressions with proper precedence
<b>Percentage Rollouts</b>	Gradual feature exposure using consistent hashing	Stable user assignment across evaluations
<b>User Segmentation</b>	Target specific user groups with attribute matching	Rule-based segments with real-time membership
<b>Evaluation Reasoning</b>	Detailed explanation of why specific variant returned	Audit trail for debugging and compliance
<b>Fallback Handling</b>	Default values when targeting rules fail or malfunction	Graceful degradation without service interruption

The evaluation engine will process complex targeting scenarios like "show new checkout flow to 25% of premium users in North America, but exclude users who signed up in the last 7 days." This requires sophisticated rule processing with proper precedence, consistent user bucketing, and comprehensive context matching.

## Goal 2: Real-time Configuration Updates

The system will deliver **sub-second flag changes** to all connected clients without requiring application restarts or deployments. This enables the rapid course corrections that make feature flags valuable for risk mitigation and experimentation.

Capability	Description	Success Criteria
<b>Streaming Updates</b>	Server-Sent Events or WebSocket delivery	Flag changes reach clients within 2-3 seconds
<b>Connection Recovery</b>	Automatic reconnection with exponential backoff	Clients reconnect after network disruption
<b>State Synchronization</b>	Consistent flag values across all SDK instances	No client serves stale flags after updates
<b>Offline Resilience</b>	Local caching with configurable TTL policies	Applications continue functioning during outages
<b>Update Ordering</b>	Changes applied in correct chronological sequence	No race conditions between rapid flag modifications
<b>Selective Updates</b>	Only changed flags transmitted to reduce bandwidth	Efficient delta updates rather than full snapshots

Real-time updates enable emergency scenarios like "immediately disable the new payment processor for all users" or "increase checkout optimization rollout from 10% to 50% during peak traffic." The system must handle these critical configuration changes reliably across distributed deployments.

### Goal 3: Comprehensive Analytics and A/B Testing

The system will provide **statistical rigor** for controlled experiments and comprehensive visibility into flag performance. This transforms feature flags from simple toggles into a data-driven experimentation platform.

Capability	Description	Success Criteria
<b>Flag Exposure Tracking</b>	Record every flag evaluation with user context	Complete audit trail of feature exposure
<b>A/B Test Framework</b>	Controlled experiments with proper randomization	Statistical significance testing with confidence intervals
<b>Metrics Collection</b>	Conversion and engagement tracking by variant	Real-time performance comparison across variants
<b>Sample Size Calculation</b>	Required user counts for statistical power	Experiment duration estimates based on traffic
<b>Sequential Analysis</b>	Early stopping with proper alpha spending	Avoid peeking problems while enabling early wins
<b>Experiment Reports</b>	Automated analysis with actionable recommendations	Clear visualization of variant performance differences

The analytics engine will detect scenarios like sample ratio mismatches that invalidate experiments, calculate required sample sizes for desired statistical power, and provide early stopping recommendations when variants show significant differences.

#### Goal 4: Production-Ready Reliability

The system will demonstrate **enterprise-grade reliability** suitable for business-critical applications with proper error handling, monitoring, and operational support.

Capability	Description	Success Criteria
<b>High Availability</b>	Multi-region deployment with failover capabilities	99.9% uptime with sub-100ms P99 latency
<b>Graceful Degradation</b>	Service continues with cached/default values during outages	No application failures due to flag service issues
<b>Comprehensive Monitoring</b>	Detailed metrics, logging, and alerting integration	Proactive detection of performance degradation
<b>Security Integration</b>	Authentication, authorization, and audit logging	Role-based access control with compliance support
<b>Scalability</b>	Handle millions of flag evaluations per second	Linear scaling with horizontal infrastructure growth
<b>Data Consistency</b>	ACID properties for critical flag configuration changes	No lost updates or corrupted flag definitions

#### Non-Goals

Understanding what our feature flag system will **not** include is equally important for maintaining focus and avoiding scope creep. These boundaries prevent the system from becoming an overly complex platform that fails at its core mission.

##### Non-Goal 1: Advanced Infrastructure Management

The system will not provide comprehensive DevOps tooling or infrastructure orchestration capabilities that belong in specialized platforms.

Excluded Capability	Rationale	Alternative Approach
<b>Container Orchestration</b>	Kubernetes and Docker management requires specialized expertise	Deploy flag service using existing container platforms
<b>Service Mesh Integration</b>	Complex networking concerns beyond flag evaluation scope	Use standard service mesh patterns for traffic management
<b>Multi-Cloud Management</b>	Cloud provider abstractions add significant complexity	Deploy using cloud-specific infrastructure as code
<b>Database Administration</b>	Schema migrations, backup/restore, performance tuning	Use managed database services with standard operational practices
<b>Load Balancer Configuration</b>	Traffic routing policies beyond basic health checks	Integrate with existing load balancing infrastructure

This boundary ensures the development team focuses on flag evaluation logic rather than becoming infrastructure specialists. Organizations should deploy the flag service using their existing DevOps toolchain and operational practices.

### Non-Goal 2: Complex Business Logic Integration

The system will not embed application-specific business rules or domain logic that belongs in individual applications.

Excluded Capability	Rationale	Alternative Approach
<b>Custom Business Rules</b>	Domain-specific logic creates tight coupling between flag service and applications	Applications implement business logic using flag values as inputs
<b>Workflow Orchestration</b>	Multi-step business processes require specialized workflow engines	Use flag values to control workflow behavior in dedicated orchestration systems
<b>Data Transformation</b>	Complex ETL operations belong in dedicated data processing pipelines	Flag service provides raw evaluation data; applications handle transformation
<b>External API Orchestration</b>	Coordinating multiple external services creates reliability dependencies	Applications manage external integrations using flags for feature gating
<b>Payment Processing</b>	Financial transactions require specialized compliance and security	Use flags to control payment feature availability, not process transactions

By avoiding business logic integration, the flag service remains reusable across different applications and domains while maintaining clear separation of concerns.

### **Non-Goal 3: Advanced Machine Learning Features**

The system will not include sophisticated AI/ML capabilities that require specialized data science expertise and infrastructure.

<b>Excluded Capability</b>	<b>Rationale</b>	<b>Alternative Approach</b>
<b>Predictive Targeting</b>	User behavior prediction requires ML model training and maintenance	Use rule-based targeting with external ML insights as input attributes
<b>Automated Flag Management</b>	AI-driven flag lifecycle management introduces unpredictable system behavior	Provide APIs for external automation tools to manage flags programmatically
<b>Anomaly Detection</b>	Statistical outlier detection needs specialized algorithms and tuning	Integrate with existing monitoring tools that provide anomaly detection
<b>Recommendation Engine</b>	Variant recommendations require complex user modeling and collaborative filtering	External recommendation systems can use flag values for A/B testing recommendations
<b>Natural Language Processing</b>	Flag configuration from natural language requires NLP infrastructure	Provide structured APIs with clear documentation for programmatic flag management

Organizations needing ML-powered feature management should integrate external ML platforms with the flag service through standard APIs rather than embedding ML capabilities directly.

### **Non-Goal 4: Comprehensive Application Performance Monitoring**

The system will not replace dedicated APM and observability platforms that provide comprehensive application monitoring.

Excluded Capability	Rationale	Alternative Approach
Distributed Tracing	Request tracing across microservices requires specialized instrumentation	Flag evaluations participate in existing distributed tracing systems
Application Metrics	Business and technical metrics belong in dedicated monitoring infrastructure	Flag exposure events feed into existing metrics collection systems
Log Aggregation	Centralized logging requires significant infrastructure and storage management	Flag service logs integrate with existing log aggregation platforms
Error Tracking	Application error monitoring needs specialized error analysis and grouping	Flag-related errors report to existing error tracking systems
Performance Profiling	Code-level performance analysis requires runtime instrumentation	Flag evaluation performance monitored through standard profiling tools

The flag service will provide observability hooks and structured logging that integrate seamlessly with existing monitoring infrastructure rather than replacing specialized observability tools.

## Scope Boundaries and Integration Points

Understanding how our feature flag system integrates with existing infrastructure helps clarify responsibility boundaries and prevents architectural conflicts.

### Integration Philosophy

#### Design Principle: Complementary System Integration

The feature flag system follows the **Unix philosophy** of doing one thing exceptionally well while providing clean interfaces for integration with other systems. Rather than replacing existing infrastructure, it enhances current development and operational workflows.

Integration Category	Flag System Responsibility	External System Responsibility
User Authentication	Accept authenticated user context for targeting	Provide user identity and session management
Metrics and Analytics	Generate flag exposure events with structured metadata	Aggregate, visualize, and alert on flag usage patterns
Configuration Management	Store and serve flag definitions with versioning	Manage application configuration, secrets, and environment variables
Deployment Pipeline	Provide APIs for automated flag lifecycle management	Orchestrate deployments, run tests, and manage release processes
Service Discovery	Register flag service endpoints for client SDK discovery	Provide service mesh, load balancing, and health checking
Data Storage	Persist flag definitions, rules, and evaluation results	Provide database infrastructure, backup, and disaster recovery

## Client SDK Integration Requirements

The flag system must provide lightweight, efficient client SDKs that minimize application impact while maximizing developer productivity.

SDK Capability	Design Requirements	Performance Constraints
Evaluation Latency	Sub-millisecond flag evaluation using local cache	No network calls during normal flag evaluation
Memory Footprint	Minimal memory usage for flag storage and evaluation context	Cache only active flags with efficient serialization
Startup Time	Fast SDK initialization without blocking application startup	Asynchronous flag loading with sensible defaults
Error Isolation	Flag evaluation failures never crash or block application code	Comprehensive fallback strategies with circuit breaker patterns
Async Updates	Background flag updates without interrupting application threads	Non-blocking update processing with atomic configuration swaps
Observability	Rich debugging information without performance penalty	Structured logging and metrics with configurable verbosity

# Architecture Decision Records

## Decision: Evaluation-First Design Priority

**Context:** Feature flag systems must balance multiple concerns including evaluation performance, real-time updates, analytics collection, and operational complexity. Different prioritization leads to fundamentally different architectures.

### Options Considered:

1. **Analytics-First:** Optimize for comprehensive data collection and experiment analysis
2. **Updates-First:** Optimize for real-time configuration changes and operational flexibility
3. **Evaluation-First:** Optimize for low-latency, high-throughput flag evaluation performance

**Decision:** Evaluation-First design with analytics and updates as secondary concerns

**Rationale:** Flag evaluation occurs orders of magnitude more frequently than configuration updates or analytics queries. A single application might evaluate flags thousands of times per request across millions of requests daily, but flag updates happen at most dozens of times per day. Poor evaluation performance directly impacts user experience and application scalability, while slower updates or analytics primarily affect developer productivity.

**Consequences:** This prioritization enables sub-millisecond evaluation latency through aggressive local caching and optimized rule processing, but requires more sophisticated approaches for real-time updates and analytics collection that don't compromise evaluation performance.

Approach	Evaluation Latency	Update Latency	Analytics Richness	Operational Complexity
<b>Analytics-First</b>	50-100ms (network calls)	5-10 seconds	Comprehensive real-time	High (complex data pipeline)
<b>Updates-First</b>	10-50ms (cache misses)	Sub-second	Limited batch processing	Medium (event streaming)
<b>Evaluation-First ✓</b>	Sub-millisecond (local)	2-5 seconds	Rich but eventually consistent	Medium (caching complexity)

## Decision: Explicit Feature Scope Definition

**Context:** Feature flag systems can evolve into comprehensive platforms handling everything from basic toggles to complex workflow orchestration, user management, and business intelligence. This scope expansion often leads to maintenance burden and reliability issues.

### Options Considered:

1. **Platform Approach:** Build comprehensive feature management platform with extensive built-in capabilities
2. **Minimal Approach:** Provide only basic boolean flags with simple on/off functionality
3. **Focused Approach:** Provide sophisticated flag evaluation, updates, and analytics while avoiding peripheral features

**Decision:** Focused Approach with explicit non-goals to prevent scope creep

**Rationale:** Platform approaches often fail because they attempt to solve too many problems simultaneously, leading to suboptimal solutions for each concern. Minimal approaches lack the sophistication needed for production use cases like gradual rollouts and A/B testing. The focused approach enables excellence in core flag management while providing integration points for specialized external systems.

**Consequences:** Development effort concentrates on flag evaluation excellence, real-time updates, and experiment analytics. External integration requirements are explicitly designed from the beginning rather than retrofitted later, leading to cleaner architecture and more reliable operation.

## Common Pitfalls in Goal Definition

Understanding common mistakes in feature flag system scoping helps avoid architectural decisions that lead to maintenance problems and reliability issues.

### ⚠️ Pitfall: Kitchen Sink Feature Creep

Many teams start with simple flag requirements but gradually add user management, complex workflow orchestration, custom business logic, and comprehensive monitoring capabilities. This transforms a focused flag service into a sprawling platform that's difficult to maintain and often unreliable.

**Why it's Wrong:** Each additional feature domain requires different expertise, testing approaches, and operational considerations. A system optimized for microsecond flag evaluation has fundamentally different requirements than one optimized for complex data analytics or user interface workflows.

**How to Fix:** Establish explicit non-goals documentation and review all feature requests against core system responsibilities. When teams request additional capabilities, provide integration APIs that allow external systems to handle peripheral concerns while leveraging flag data.

### ⚠️ Pitfall: Premature Optimization Boundaries

Teams sometimes define overly restrictive goals based on current performance requirements without considering reasonable growth scenarios. This leads to systems that can't handle obvious scaling needs or require complete rewrites when traffic increases.

**Why it's Wrong:** Flag evaluation requirements often grow exponentially with application adoption. A system designed for 1,000 evaluations per second might face 100,000 evaluations per second within months. Similarly, simple boolean flags often evolve into complex multi-variant experiments as organizations mature their feature management practices.

**How to Fix:** Define goals that accommodate 10x growth in evaluation volume and rule complexity without architectural changes. Plan for linear scaling through horizontal infrastructure rather than requiring fundamental design modifications.

### **Pitfall: Integration Assumption Mismatch**

Development teams often assume their flag system will integrate seamlessly with existing infrastructure without explicitly designing integration points. This leads to architecture conflicts, performance problems, and operational complexity.

**Why it's Wrong:** Different organizations have varying authentication systems, monitoring infrastructure, deployment pipelines, and data storage approaches. A flag system designed assuming specific external dependencies often requires significant modifications during deployment.

**How to Fix:** Define explicit integration interfaces and assumptions in the goals documentation. Design abstraction layers for authentication, metrics, storage, and configuration that allow adaptation to different organizational infrastructure without core logic changes.

## **Validation Criteria and Success Metrics**

Establishing measurable success criteria ensures the implemented system meets its defined goals while respecting non-goal boundaries.

## Functional Validation Requirements

Goal Category	Validation Method	Success Threshold	Measurement Approach
Flag Evaluation	Load testing with complex rules	99.9% evaluations complete within 1ms	Synthetic traffic with realistic user contexts
Real-time Updates	Flag change propagation timing	95% of clients receive updates within 3 seconds	Distributed update monitoring with client-side timestamps
A/B Testing	Statistical accuracy validation	Experiment results within 2% of ground truth	Controlled experiments with known conversion differences
Reliability	Chaos engineering and failure injection	No application failures due to flag service issues	Systematic infrastructure failure testing

## Non-Goal Compliance Validation

Non-Goal Category	Validation Method	Compliance Criteria	Red Flag Indicators
Infrastructure Scope	Architecture review	Zero database administration or container orchestration features	Custom deployment scripts or database schema management
Business Logic Scope	Code review	No domain-specific rules or workflow orchestration	Application-specific constants or business rule encoding
ML Feature Scope	Dependency analysis	No machine learning libraries or model training capabilities	Statistical modeling or prediction algorithm implementations
APM Scope	Integration testing	Flag service data flows to existing monitoring infrastructure	Custom metrics dashboards or log aggregation systems

## Implementation Guidance

The goals and non-goals established in this section provide the foundation for all implementation decisions throughout the project. Here's how to translate these scope boundaries into concrete development practices.

## Technology Selection Framework

When choosing technologies and dependencies for each milestone, evaluate options against the established goals and explicitly reject solutions that violate non-goal boundaries.

Component Category	Evaluation Criteria	Recommended Approach	Red Flag Dependencies
Flag Storage	Evaluation latency priority, consistency guarantees	In-memory caching with persistent backing store	Complex ORM frameworks, heavy database abstraction layers
Real-time Updates	Connection management, bandwidth efficiency	Server-Sent Events with JSON payloads	Complex message brokers, heavyweight pub/sub systems
Analytics Backend	Statistical accuracy, query performance	Time-series database with statistical functions	Full-featured business intelligence or data warehouse platforms
Client SDK	Minimal footprint, fast evaluation	Simple HTTP client with local caching	Complex frameworks, heavyweight serialization libraries

## Project Structure Organization

Organize the codebase to reinforce goal boundaries and prevent scope creep through clear module separation:

```
feature-flag-system/
├── cmd/
│   ├── flag-server/           ← Core flag evaluation and updates service
│   └── analytics-processor/   ← Experiment analysis and reporting
└── internal/
    ├── evaluation/           ← Flag evaluation engine (Goal 1)
    │   ├── engine.go
    │   ├── rules.go
    │   └── consistent_hash.go
    ├── realtime/              ← Update streaming (Goal 2)
    │   ├── sse_handler.go
    │   └── connection_manager.go
    ├── analytics/             ← A/B testing framework (Goal 3)
    │   ├── experiment.go
    │   └── statistics.go
    └── storage/               ← Data persistence abstraction
        ├── memory_store.go
        └── postgres_store.go
└── pkg/
    └── sdk/                  ← Client SDK for applications
        ├── client.go
        └── cache.go
└── examples/
    ├── basic-usage/          ← Simple flag evaluation examples
    └── ab-testing/            ← Experiment setup examples
```

## **Core Interface Definitions**

Define clean interfaces that enforce goal boundaries and prevent feature creep:

GO

```
// FlagEvaluator handles core evaluation logic (Goal 1)

type FlagEvaluator interface {

    // EvaluateFlag returns variant assignment with reasoning

    EvaluateFlag(flagKey FlagKey, context UserContext) (EvaluationResult, error)

    // TODO: Implement consistent hashing for stable user assignment

    // TODO: Process targeting rules with proper precedence

    // TODO: Handle fallback values when rules don't match

}

// UpdateStreamer manages real-time flag changes (Goal 2)

type UpdateStreamer interface {

    // StreamUpdates sends flag changes to connected clients

    StreamUpdates(clientID string) (<-chan FlagUpdate, error)

    // TODO: Implement SSE connection management

    // TODO: Handle client reconnection with state recovery

    // TODO: Broadcast updates to all connected clients

}

// ExperimentTracker handles A/B testing analytics (Goal 3)

type ExperimentTracker interface {

    // RecordExposure logs flag evaluation for analysis

    RecordExposure(exposure FlagExposure) error

    // CalculateSignificance computes experiment statistical results

    CalculateSignificance(experimentID string) (SignificanceResult, error)

}
```

```
// TODO: Implement exposure event collection  
  
// TODO: Calculate statistical significance with proper methods  
  
// TODO: Generate experiment reports with confidence intervals  
  
}
```

## Milestone Validation Checkpoints

After implementing each milestone, validate that the system meets its goals without violating non-goal boundaries:

### Milestone 1 Checkpoint (Flag Evaluation Engine):

- Run `go test ./internal/evaluation/...` - all tests pass
- Load test: 10,000 flag evaluations complete within 10ms total
- Complex rule test: nested AND/OR conditions evaluate correctly
- Consistency test: same user receives identical variants across multiple evaluations
- Verify: No business logic hardcoded, no ML dependencies, no infrastructure management code

### Milestone 2 Checkpoint (Real-time Updates):

- Start flag server: `go run cmd/flag-server/main.go`
- Connect test client, modify flag via API, client receives update within 3 seconds
- Network disruption test: disconnect/reconnect client, verify state synchronization
- Verify: No complex message broker usage, no container orchestration, updates don't impact evaluation latency

### Milestone 3 Checkpoint (Analytics & Experiments):

- Create A/B test with known 10% conversion difference
- Run experiment with synthetic traffic for statistical significance
- Verify: Statistical calculations accurate within 2%, no ML model training, no custom APM dashboard

## Scope Enforcement Techniques

Implement these development practices to maintain goal focus and prevent scope creep:

### Code Review Checklist:

- Does this change directly support one of the four primary goals?
- Are we adding infrastructure management, business logic, ML features, or APM capabilities?
- Do new dependencies align with evaluation-first performance priorities?
- Are integration points cleanly separated from core flag logic?

**Architecture Decision Template:** For any significant design choice, explicitly evaluate against goals:

- How does this support flag evaluation performance?
- Does this compromise real-time update capabilities?
- Will this enable or hinder A/B testing statistical accuracy?
- Are we maintaining production reliability standards?
- Which non-goals might this violate, and how do we prevent that?

**Dependency Review Process:** Before adding any external dependency:

- Measure impact on flag evaluation latency
- Verify alignment with focused system approach
- Confirm integration rather than replacement of existing infrastructure
- Document why simpler alternatives are insufficient

This systematic approach to goal enforcement ensures the feature flag system delivers exceptional performance in its core mission while maintaining clean integration boundaries with existing organizational infrastructure.

## High-Level Architecture

**Milestone(s):** This section provides the architectural foundation for all three milestones by defining the core components that support the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3).

### Mental Model: Feature Flags as Air Traffic Control

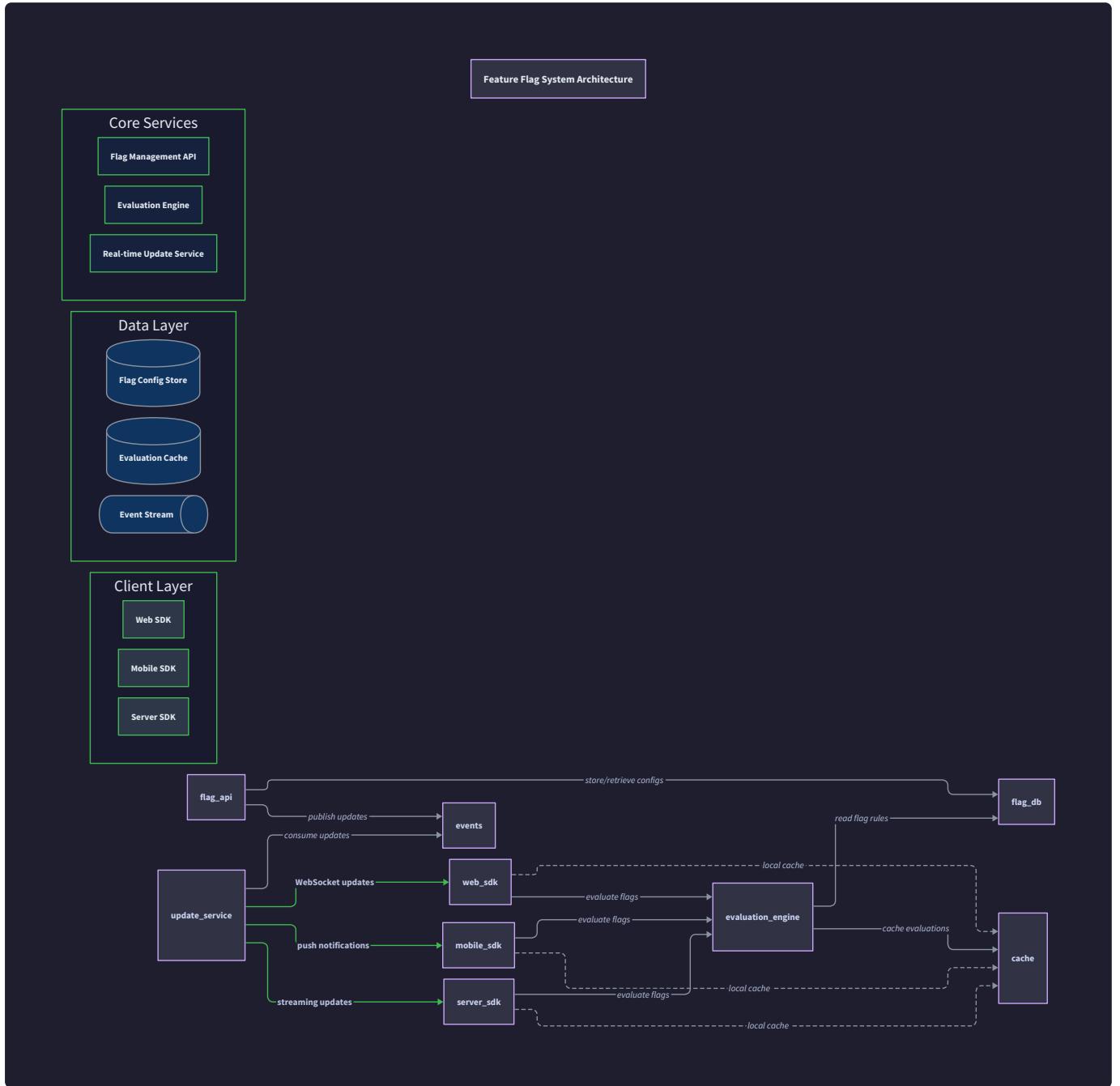
Think of a feature flag system as an **air traffic control tower** managing the flow of software releases into production. Just as air traffic control coordinates aircraft approaches, assigns landing sequences, and manages emergency protocols, a feature flag system orchestrates which features reach which users under what conditions.

The air traffic control analogy illuminates several key architectural principles. The **control tower** (Flag Management API) provides centralized oversight and coordination, while **flight controllers** (Client SDKs) execute instructions at the ground level. **Radar systems** (Real-time Update Service) provide continuous situational awareness, and **flight plans** (flag configurations) define predetermined routes and conditions.

**Emergency protocols** (fallback mechanisms) ensure safe operation when communication fails.

This mental model emphasizes why feature flag architecture must prioritize **reliability over performance**. A crashed air traffic control system grounds all flights; similarly, a failed flag evaluation system can block all feature rollouts or, worse, expose unstable features to production traffic. The architecture must embrace **graceful degradation** principles, where partial system failures reduce capability rather than causing complete outages.

The evaluation-first design principle emerges naturally from this analogy. Air traffic control systems prioritize rapid decision-making over comprehensive data collection—controllers need split-second aircraft positioning, not detailed historical flight patterns. Similarly, flag evaluation must complete in single-digit milliseconds, even if analytics data has eventual consistency.



## Component Overview

The feature flag system consists of four primary components that work together to provide fast, reliable flag evaluation with real-time updates and comprehensive analytics. Each component owns specific responsibilities and maintains clear boundaries to support independent scaling and deployment.

## Flag Management API

The **Flag Management API** serves as the central command center for all flag operations, similar to how air traffic control manages flight plans and clearances. This component handles the administrative lifecycle of feature flags while maintaining the authoritative state that drives evaluation decisions across all client SDKs.

The Flag Management API owns the complete flag definition lifecycle, from initial creation through rule updates to eventual flag retirement. It provides RESTful endpoints for creating flags, configuring targeting rules, setting up A/B experiments, and managing percentage rollouts. The API validates all flag configurations before persistence, ensuring that rule syntax is correct, percentage allocations sum appropriately, and dependency graphs remain acyclic.

Operation	Method	Endpoint	Description	Authentication
Create Flag	POST	/api/flags	Create new feature flag with targeting rules	Admin
Update Flag	PUT	/api/flags/{flagKey}	Modify flag configuration or targeting	Admin
Get Flag	GET	/api/flags/{flagKey}	Retrieve flag definition and current status	Admin/Read-Only
List Flags	GET	/api/flags	List all flags with filtering and pagination	Admin/Read-Only
Delete Flag	DELETE	/api/flags/{flagKey}	Archive flag and clean up references	Admin
Create Experiment	POST	/api/experiments	Set up A/B test with statistical configuration	Admin
Get Experiment Results	GET	/api/experiments/{id}/results	Retrieve experiment metrics and significance	Admin/Read-Only

The API implements **optimistic concurrency control** using entity versioning to prevent conflicting updates from multiple administrators. Each flag definition includes a version number that increments with each modification. Update requests must specify the expected version, and the API rejects requests where the expected version doesn't match the current state.

**Data consistency** across the system starts with the Flag Management API's write operations. When administrators modify flag configurations, the API immediately persists changes to the primary data store and broadcasts update notifications through the Real-time Update Service. This ensures that all client SDKs receive consistent flag definitions without requiring complex distributed consensus protocols.

The Flag Management API also serves as the **audit trail** for all flag-related changes, recording who made what changes when for compliance and debugging purposes. This audit log proves invaluable when investigating unexpected behavior or understanding the timeline of feature rollouts.

**Key Design Insight:** The Flag Management API prioritizes consistency and auditability over raw performance. Administrative operations are infrequent compared to flag evaluations, so we can afford stronger consistency guarantees and comprehensive validation at this layer.

## Evaluation Engine

The **Evaluation Engine** represents the high-performance core of the feature flag system, analogous to the split-second decision-making capabilities of air traffic controllers during peak operations. This component must process thousands of flag evaluation requests per second while maintaining consistent user assignments and respecting complex targeting rules.

The Evaluation Engine implements the `EvaluateFlag` function that takes a `FlagKey` and `UserContext` and returns an `EvaluationResult` containing the assigned variant, the computed value, and detailed reasoning for debugging purposes. The engine processes targeting rules in strict precedence order, applies percentage rollouts using consistent hashing, and returns appropriate fallback values when no rules match.

Evaluation Step	Input	Processing	Output	Performance Target
Context Validation	<code>UserContext</code>	Validate required attributes present	Valid context or error	< 0.1ms
Rule Matching	Flag rules + context	Evaluate conditions with AND/OR logic	Matching rule or none	< 0.5ms
Percentage Allocation	User ID + rollout config	Apply consistent hashing algorithm	Variant assignment	< 0.1ms
Value Resolution	Variant config	Lookup variant value and metadata	Resolved value	< 0.1ms
Result Assembly	All previous outputs	Construct evaluation result with reason	<code>EvaluationResult</code>	< 0.1ms

**Consistent hashing** ensures that users receive the same variant assignment across multiple evaluations, even as flag configurations change. The engine combines the user's stable identifier with the flag key using a

cryptographic hash function, then maps the hash output to variant buckets based on configured weight distributions. This approach prevents users from flip-flopping between variants when percentage rollouts expand or contract.

The evaluation engine maintains **in-memory flag caches** to avoid database lookups on every evaluation request. These caches update immediately when the Real-time Update Service broadcasts flag changes, ensuring sub-millisecond evaluation performance while maintaining data freshness. The cache implementation uses read-write locks to allow concurrent evaluations while blocking only during cache updates.

**Rule evaluation** follows a deterministic precedence system where more specific targeting rules override general percentage rollouts. The engine evaluates user segment membership first, then individual user targeting, followed by attribute-based conditions, and finally percentage rollouts for the remaining population. This precedence system ensures predictable behavior when multiple rules could apply to the same user.

**Performance Guarantee:** The Evaluation Engine must complete flag evaluations in under 1 millisecond for 99% of requests. This strict performance requirement drives architectural decisions around caching, rule complexity limits, and fallback behaviors.

## Real-time Update Service

The **Real-time Update Service** functions as the radar and communication system of our air traffic control analogy, providing continuous situational awareness to all client SDKs about changing flag configurations. This component bridges the gap between administrative flag changes and their immediate propagation to production evaluation contexts.

The service maintains persistent connections to client SDKs using **Server-Sent Events (SSE)** for flag update streaming. SSE provides the right balance of simplicity and real-time performance for flag updates, avoiding the complexity of WebSocket bi-directional communication while maintaining sub-second update latency. The service can handle thousands of concurrent client connections while batching update notifications to minimize bandwidth usage.

Connection State	Allowed Operations	Automatic Actions	Client Behavior
Connected	Receive updates, send heartbeats	Send flag changes immediately	Process updates, maintain cache
Reconnecting	Buffer incoming requests	Apply exponential backoff	Use cached flags, attempt reconnect
Disconnected	Use local cache only	Log connection attempts	Graceful degradation to cache

**Connection management** implements intelligent reconnection strategies with exponential backoff and jitter to prevent thundering herd problems when network connectivity returns after outages. The service tracks the last successful update timestamp for each client, enabling state synchronization when connections restore. Clients

that reconnect after extended downtime receive complete flag configuration snapshots rather than incremental updates.

The Real-time Update Service implements **update deduplication** to handle rapid successive changes to the same flag. Rather than sending multiple update notifications within a short time window, the service batches changes and sends only the final flag state. This optimization reduces network traffic and client processing overhead during periods of intensive flag configuration changes.

**Broadcasting efficiency** uses topic-based subscription patterns where clients can subscribe to specific flag changes rather than receiving all system updates. This selective subscription reduces bandwidth usage for deployments with hundreds of flags where individual services only need subsets of the total flag configuration.

The service maintains detailed **connection metrics** including client count, update delivery success rates, and reconnection patterns. These metrics provide visibility into update propagation performance and help identify clients experiencing connectivity issues that might be serving stale flag configurations.

## Client SDKs

The **Client SDKs** represent the ground-level flight controllers in our air traffic control system, executing flag evaluation decisions within application processes while maintaining connection to the central coordination infrastructure. These SDKs provide the developer-facing API that applications use to check flag values and record analytics events.

Client SDKs implement **local evaluation** patterns where flag configurations are cached within the application process, enabling flag evaluations to complete without network round trips. This local evaluation approach ensures that flag checks remain fast even during network partitions or service outages, supporting the graceful degradation requirements of production systems.

SDK Component	Responsibility	Performance Impact	Fallback Behavior
Evaluation Cache	Store flag definitions locally	Enables sub-ms evaluations	Serves stale data during outages
Update Client	Receive real-time flag changes	Minimal—async background updates	Continues with cached flags
Analytics Buffer	Queue exposure events for batch upload	Minimal—async fire-and-forget	Drops events when buffer full
Context Manager	Handle user context and segmentation	Minimal—in-memory operations	Uses empty context as fallback

**Cache management** within SDKs uses a two-tier approach with hot and warm flag storage. Frequently accessed flags remain in optimized in-memory structures for microsecond lookup times, while less active flags use more compact serialized representations. The SDK automatically promotes flags to hot storage based on evaluation frequency patterns.

**Analytics integration** enables SDKs to record flag exposure events using the `RecordExposure` function without impacting evaluation performance. SDKs batch exposure events and upload them asynchronously, ensuring that analytics tracking doesn't slow down the application's critical path. When analytics services are unavailable, SDKs gracefully drop events rather than blocking flag evaluations.

The SDK provides **configuration options** for different deployment scenarios, including strict mode for development environments where flag evaluation failures should cause obvious errors, and permissive mode for production where graceful fallbacks maintain application stability.

**Language-specific optimizations** allow SDKs to leverage platform capabilities like Go's lightweight goroutines for background updates or Java's concurrent data structures for thread-safe cache access. However, all SDKs maintain consistent evaluation semantics and API contracts across language boundaries.

**Design Trade-off:** Client SDKs prioritize evaluation speed and resilience over perfect consistency. They may serve flag values that are seconds out of date during network partitions, but they guarantee that applications continue functioning rather than failing due to flag service dependencies.

## Project Structure and Organization

The feature flag system follows a **modular architecture** that separates concerns cleanly while supporting independent development and testing of each component. The project structure reflects the logical component boundaries and provides clear interfaces between subsystems.

```
flagsystem/
├── cmd/                                # Application entry points
│   ├── flagserver/                      # Flag Management API server
│   ├── updateservice/                   # Real-time Update Service
│   └── analytics/                       # Analytics processing service
├── internal/                            # Private application packages
│   ├── evaluation/                     # Core evaluation engine
│   ├── storage/                         # Data persistence layer
│   ├── realtime/                        # Update streaming infrastructure
│   ├── analytics/                       # Metrics and experiment analysis
│   └── common/                           # Shared utilities and types
└── pkg/                                 # Public API packages
    ├── sdk/                             # Client SDK implementations
    ├── api/                             # API client libraries
    └── types/                           # Shared type definitions
├── configs/                            # Configuration files
├── migrations/                         # Database schema migrations
├── deployments/                        # Deployment configurations
└── docs/                               # Additional documentation
```

## Core Internal Packages

The **internal/evaluation** package contains the flag evaluation engine and implements all rule processing logic. This package has no external dependencies beyond basic data structures, making it highly testable and

portable across different deployment environments.

File	Purpose	Key Types	External Dependencies
<code>engine.go</code>	Core evaluation logic and <code>EvaluateFlag</code> implementation	<code>EvaluationEngine</code> , <code>EvaluationResult</code>	None
<code>rules.go</code>	Targeting rule processing and precedence handling	<code>Rule</code> , <code>Condition</code> , <code>RuleEvaluator</code>	None
<code>hashing.go</code>	Consistent hashing for user assignment	<code>UserHasher</code> , <code>BucketAllocator</code>	<code>crypto/sha256</code>
<code>cache.go</code>	In-memory flag definition caching	<code>FlagCache</code> , <code>CacheEntry</code>	<code>sync</code>

The **internal/storage** package abstracts data persistence operations and provides interfaces that support multiple backend implementations. The package defines repository interfaces for flags, experiments, and analytics data while implementing concrete adapters for specific databases.

The **internal/realtim**e package handles all streaming update functionality including SSE connection management, client tracking, and update broadcasting. This package integrates with the storage layer to detect flag changes and with the evaluation engine to validate update consistency.

## Public SDK Package Structure

The **pkg/sdk** package provides client libraries for different programming languages while maintaining consistent APIs and evaluation semantics. Each language implementation follows the same architectural patterns adapted to platform-specific idioms and performance characteristics.

Language	Package Structure	Key Files	Platform Optimizations
Go	<code>pkg/sdk/go/</code>	<code>client.go</code> , <code>cache.go</code> , <code>types.go</code>	Goroutines for background updates
Python	<code>pkg/sdk/python/</code>	<code>client.py</code> , <code>cache.py</code> , <code>types.py</code>	Asyncio for concurrent operations
Java	<code>pkg/sdk/java/</code>	<code>Client.java</code> , <code>Cache.java</code> , <code>Types.java</code>	ExecutorService for thread management

## Configuration and Deployment Structure

The **configs/** directory contains environment-specific configuration files using a hierarchical structure that supports local development, testing, staging, and production deployments. Configuration files use YAML format with clear sections for each system component.

The **deployments/** directory includes Docker configurations, Kubernetes manifests, and infrastructure-as-code definitions that support the feature flag system's deployment across different environments. The deployment configurations reflect the component architecture with separate services for the Flag Management API, Real-time Update Service, and analytics processing.

**Organizational Principle:** The project structure emphasizes **interface-driven development** where each package exports clear contracts through interface definitions. This approach supports testing with mock implementations and enables incremental migration to different storage or streaming technologies.

## Architecture Decision Records

### Decision: Evaluation-First Architecture

- **Context:** Feature flag systems must balance evaluation performance, update latency, administrative functionality, and analytics capabilities within resource constraints
- **Options Considered:**
  1. Admin-first: Comprehensive management UI with basic evaluation
  2. Analytics-first: Detailed experiment tracking with slower evaluation
  3. Evaluation-first: Optimized flag checking with supporting services
- **Decision:** Build an evaluation-first architecture that prioritizes flag evaluation performance above other concerns
- **Rationale:** Flag evaluations occur orders of magnitude more frequently than administrative operations. A typical application performs thousands of evaluations per second but only updates flag configurations a few times per day. Slow evaluations directly impact user-facing application performance, while slower administrative interfaces only affect developer productivity
- **Consequences:** This architecture enables sub-millisecond flag evaluations through local caching and optimized rule processing, but requires more sophisticated real-time update mechanisms to maintain cache consistency across distributed clients

Approach	Evaluation Latency	Update Propagation	Administrative Features	Chosen?
Admin-first	10-50ms (DB per eval)	Immediate	Rich UI, complex workflows	✗
Analytics-first	5-20ms (logging overhead)	Eventual consistency	Basic admin, detailed metrics	✗
Evaluation-first	<1ms (cached local eval)	<5 seconds via SSE	Functional admin, core metrics	✓

## Decision: Server-Sent Events for Real-time Updates

- **Context:** Client SDKs need real-time flag updates to maintain cache consistency while supporting thousands of concurrent connections efficiently
- **Options Considered:**
  1. Polling: Clients request updates on fixed intervals
  2. WebSockets: Full bidirectional communication channels
  3. Server-Sent Events: Unidirectional server-to-client streaming
- **Decision:** Use Server-Sent Events (SSE) for streaming flag updates to client SDKs
- **Rationale:** Flag updates are inherently unidirectional—the server pushes configuration changes to clients, but clients don't need to send data back through the same channel. SSE provides automatic reconnection, built-in event typing, and lower overhead than WebSockets. The protocol handles connection failures gracefully with exponential backoff, reducing server load during network instability
- **Consequences:** SSE enables sub-second update propagation with minimal server resources, but limits future bidirectional features like client-initiated flag requests or real-time debugging commands

Protocol	Connection Overhead	Reconnection Handling	Browser Support	Server Complexity	Chosen?
Polling	Low per request	Automatic	Universal	Minimal	✗
WebSockets	High (persistent TCP)	Manual implementation	Good	High	✗
SSE	Medium (HTTP streaming)	Automatic with backoff	Good	Medium	✓

## Decision: In-Memory Flag Caching with Real-time Invalidations

- **Context:** Flag evaluations must complete in under 1 millisecond while maintaining reasonable consistency when flag configurations change
- **Options Considered:**
  1. Direct database access: Query flag definitions on every evaluation
  2. Application-level caching: Cache flags with TTL-based expiration
  3. Real-time cache invalidation: Cache flags and invalidate immediately on changes
- **Decision:** Implement in-memory flag caching with real-time cache invalidation triggered by update notifications
- **Rationale:** Database queries add 5-50ms latency that violates performance requirements, while TTL-based caching can serve stale flag values for extended periods during rapid configuration changes. Real-time invalidation provides the best balance of performance and consistency by maintaining microsecond evaluation speeds while propagating changes within seconds
- **Consequences:** This approach enables target performance goals and ensures timely update propagation, but requires sophisticated cache management logic and increases system complexity through the real-time update infrastructure

Caching Strategy	Evaluation Latency	Consistency Guarantee	Implementation Complexity	Chosen?
No caching	5-50ms	Strong	Low	✗
TTL-based	<1ms	Eventual (TTL period)	Medium	✗
Real-time invalidation	<1ms	Near real-time (<5sec)	High	✓

## Implementation Guidance

### Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	<code>net/http</code> with <code>gorilla/mux</code>	<code>gin-gonic/gin</code> with middleware
Database	PostgreSQL with <code>lib/pq</code>	PostgreSQL with <code>gorm</code> ORM
Caching	In-memory <code>sync.Map</code>	Redis with <code>go-redis/redis</code>
Real-time Updates	SSE with <code>net/http</code> flusher	Custom SSE library with reconnection
Configuration	YAML files with <code>gopkg.in/yaml.v3</code>	Consul/etcd with <code>spf13/viper</code>
Logging	Standard <code>log</code> package	<code>sirupsen/logrus</code> structured logging
Testing	Built-in <code>testing</code> package	<code>testify/suite</code> with assertions

## Recommended Project Structure

```
flagsystem/
├── cmd/
│   ├── flagserver/
│   │   └── main.go          # Flag Management API entry point
│   ├── updateservice/
│   │   └── main.go          # Real-time Update Service entry point
│   └── analytics/
│       └── main.go          # Analytics service entry point
└── internal/
    ├── evaluation/
    │   ├── engine.go          # Core evaluation implementation
    │   ├── engine_test.go     # Evaluation engine tests
    │   ├── rules.go           # Rule processing logic
    │   ├── rules_test.go      # Rule evaluation tests
    │   ├── cache.go           # Flag definition caching
    │   └── hashing.go         # Consistent user assignment
    ├── storage/
    │   ├── interfaces.go      # Repository interfaces
    │   ├── postgres.go         # PostgreSQL implementation
    │   ├── memory.go          # In-memory implementation for testing
    │   └── migrations/        # Database schema migrations
    ├── realtime/
    │   ├── server.go          # SSE server implementation
    │   ├── connections.go     # Client connection management
    │   └── broadcaster.go     # Update broadcasting logic
    ├── api/
    │   ├── handlers.go         # HTTP request handlers
    │   ├── middleware.go       # Authentication and logging middleware
    │   └── validation.go       # Request validation logic
    ├── analytics/
    │   ├── collector.go        # Exposure event collection
    │   ├── experiments.go     # A/B testing logic
    │   └── statistics.go       # Statistical significance calculation
    └── common/
        ├── types.go            # Shared type definitions
        ├── errors.go           # Error types and handling
        └── config.go           # Configuration structures
pkg/
└── sdk/
    └── go/
        ├── client.go           # SDK client implementation
        ├── client_test.go       # SDK integration tests
        ├── cache.go             # Client-side flag caching
        └── types.go             # SDK-specific types
configs/
└── configs/
    ├── development.yaml     # Development environment config
    ├── testing.yaml          # Testing environment config
    └── production.yaml       # Production environment config
deployments/
└── deployments/
    ├── docker-compose.yaml   # Local development setup
    └── kubernetes/           # Kubernetes manifests
```

```
|   └── terraform/          # Infrastructure as code
├── docs/
│   ├── api.md              # API documentation
│   └── deployment.md       # Deployment guide
├── go.mod                 # Go module definition
├── go.sum                 # Dependency checksums
├── Makefile                # Build and test automation
└── README.md               # Project overview and setup
```

## Core Type Definitions

```
// internal/common/types.go                                     GO

package common

import "time"

// Core identifier types for type safety

type FlagKey string

type UserID string

// User context for flag evaluation targeting

type UserContext struct {

    UserID      UserID          `json:"user_id"`

    Attributes map[string]interface{} `json:"attributes"`

    Segments   []string         `json:"segments"`

}

// Flag variant definition with weighted allocation

type Variant struct {

    Key      string           `json:"key"`

    Value   interface{}       `json:"value"`

    Weight  int               `json:"weight"`

}

// Result of flag evaluation with debugging information

type EvaluationResult struct {

    FlagKey FlagKey        `json:"flag_key"`

    Value   interface{}     `json:"value"`

    Variant string         `json:"variant"`

}
```

```

Reason  string      `json:"reason"`

Source  string      `json:"source"`

}

// Flag change notification for real-time updates

type FlagUpdate struct {

    FlagKey  FlagKey    `json:"flag_key"`

    Operation string    `json:"operation"` // "create", "update", "delete"

    Timestamp time.Time `json:"timestamp"`

    Version   int64     `json:"version"`

}

// Flag evaluation event for analytics

type FlagExposure struct {

    FlagKey  FlagKey    `json:"flag_key"`

    UserID    UserID     `json:"user_id"`

    Variant   string     `json:"variant"`

    Value     interface{} `json:"value"`

    Context   map[string]interface{} `json:"context"`

    Timestamp time.Time    `json:"timestamp"`

    ExperimentID *string    `json:"experiment_id,omitempty"`

}

// Statistical significance calculation result

type SignificanceResult struct {

    ExperimentID string    `json:"experiment_id"`

    ControlVariant string   `json:"control_variant"`

    TreatmentVariant string  `json:"treatment_variant"`
}

```

```
PValue          float64   `json:"p_value"`

Confidence     float64   `json:"confidence"`

Significant    bool      `json:"significant"`

CalculatedAt   time.Time `json:"calculated_at"`

}
```

## Core Interface Definitions

```
// internal/evaluation/interfaces.go GO

package evaluation

// Core flag evaluation interface

type Evaluator interface {

    // EvaluateFlag processes targeting rules and returns appropriate variant
    EvaluateFlag(flagKey FlagKey, context UserContext) EvaluationResult

    // UpdateFlags refreshes cached flag definitions
    UpdateFlags(updates []FlagUpdate) error

    // GetFlags returns all currently cached flag keys
    GetFlags() []FlagKey

}

// internal/realtimre/interfaces.go

package realtime

// Real-time update streaming interface

type UpdateStreamer interface {

    // StreamUpdates provides real-time flag changes for specific client
    StreamUpdates(clientID string) chan FlagUpdate

    // BroadcastUpdate sends flag changes to all connected clients
    BroadcastUpdate(update FlagUpdate) error

    // DisconnectClient cleans up resources for disconnected client
}
```

```
DisconnectClient(clientID string) error

}

// internal/analytics/interfaces.go

package analytics

// Analytics and experiment tracking interface

type AnalyticsCollector interface {

    // RecordExposure logs flag evaluation event for analysis
    RecordExposure(exposure FlagExposure) error

    // CalculateSignificance computes experiment statistical results
    CalculateSignificance(experimentID string) SignificanceResult

    // GetExperimentMetrics returns aggregated experiment data
    GetExperimentMetrics(experimentID string) (map[string]interface{}, error)
}
```

## Infrastructure Starter Code

```
// internal/storage/memory.go - Complete in-memory storage for testing  
GO  
  
package storage  
  
import (  
    "fmt"  
    "sync"  
    "time"  
  
    "flagsystem/internal/common"  
)  
  
// InMemoryFlagStore provides thread-safe in-memory flag storage  
  
type InMemoryFlagStore struct {  
  
    flags  map[common.FlagKey]*FlagDefinition  
    mutex  sync.RWMutex  
  
    // Audit trail for debugging  
    changes []FlagChange  
  
}  
  
type FlagDefinition struct {  
  
    Key      common.FlagKey      `json:"key"  
    Name     string             `json:"name"  
    Description string          `json:"description"  
    Enabled   bool              `json:"enabled"  
    Variants  []common.Variant `json:"variants"  
    Rules     []TargetingRule   `json:"rules"  
}
```

```

DefaultRule *PercentageRule           `json:"default_rule"`

Version     int64                   `json:"version"`

CreatedAt   time.Time              `json:"created_at"`

UpdatedAt   time.Time              `json:"updated_at"`

}

type TargetingRule struct {

    ID          string            `json:"id"`

    Name        string            `json:"name"`

    Conditions  []Condition      `json:"conditions"`

    Operator    string            `json:"operator"` // "AND" or "OR"

    Allocation  []VariantAllocation `json:"allocation"`

    Enabled     bool              `json:"enabled"`

}

type Condition struct {

    Attribute  string            `json:"attribute"`

    Operator   string            `json:"operator"` // "equals", "in", "contains", etc.

    Value      interface{}`json:"value"`

}

type VariantAllocation struct {

    Variant    string            `json:"variant"`

    Percentage int               `json:"percentage"`

}

type PercentageRule struct {

    Allocations []VariantAllocation `json:"allocations"`

}

```

```
type FlagChange struct {

    FlagKey    common.FlagKey `json:"flag_key"`

    Operation string          `json:"operation"`

    Timestamp time.Time       `json:"timestamp"`

    Version   int64           `json:"version"`

}

// NewInMemoryFlagStore creates a new in-memory flag store

func NewInMemoryFlagStore() *InMemoryFlagStore {

    return &InMemoryFlagStore{

        flags:    make(map[common.FlagKey]*FlagDefinition),
        changes: make([]FlagChange, 0),
    }
}

// GetFlag retrieves a flag definition by key

func (s *InMemoryFlagStore) GetFlag(key common.FlagKey) (*FlagDefinition, error) {

    s.mutex.RLock()
    defer s.mutex.RUnlock()

    flag, exists := s.flags[key]

    if !exists {

        return nil, fmt.Errorf("flag not found: %s", key)
    }

    // Return a copy to prevent external mutation

    flagCopy := *flag
```

```
    return &flagCopy, nil

}

// CreateFlag stores a new flag definition

func (s *InMemoryFlagStore) CreateFlag(flag *FlagDefinition) error {

    s.mutex.Lock()

    defer s.mutex.Unlock()

    if _, exists := s.flags[flag.Key]; exists {

        return fmt.Errorf("flag already exists: %s", flag.Key)

    }

    now := time.Now()

    flag.Version = 1

    flag.CreatedAt = now

    flagUpdatedAt = now

    // Store a copy to prevent external mutation

    flagCopy := *flag

    s.flags[flag.Key] = &flagCopy

}

// Record change for audit trail

s.changes = append(s.changes, FlagChange{

    FlagKey:    flag.Key,

    Operation: "create",

    Timestamp: now,

    Version:   flag.Version,
```

```
    })

    return nil
}

// UpdateFlag modifies an existing flag definition

func (s *InMemoryFlagStore) UpdateFlag(flag *FlagDefinition) error {
    s.mutex.Lock()

    defer s.mutex.Unlock()

    existing, exists := s.flags[flag.Key]

    if !exists {

        return fmt.Errorf("flag not found: %s", flag.Key)
    }

    now := time.Now()

    flag.Version = existing.Version + 1

    flag.CreatedAt = existing.CreatedAt

    flagUpdatedAt = now

    // Store a copy to prevent external mutation

    flagCopy := *flag

    s.flags[flag.Key] = &flagCopy

    // Record change for audit trail

    s.changes = append(s.changes, FlagChange{
        FlagKey:   flag.Key,
```

```
        Operation: "update",
        Timestamp: now,
        Version:    flag.Version,
    })

    return nil
}

// ListFlags returns all flag definitions with optional filtering

func (s *InMemoryFlagStore) ListFlags(enabledOnly bool) ([]*FlagDefinition, error) {
    s.mutex.RLock()
    defer s.mutex.RUnlock()

    flags := make([]*FlagDefinition, 0, len(s.flags))

    for _, flag := range s.flags {
        if !enabledOnly || flag.Enabled {
            // Return copies to prevent external mutation
            flagCopy := *flag
            flags = append(flags, &flagCopy)
        }
    }

    return flags, nil
}

// GetChangesSince returns flag changes after specified timestamp

func (s *InMemoryFlagStore) GetChangesSince(since time.Time) ([]FlagChange, error) {
```

```
s.mutex.RLock()

defer s.mutex.RUnlock()

changes := make([]FlagChange, 0)

for _, change := range s.changes {
    if change.Timestamp.After(since) {
        changes = append(changes, change)
    }
}

return changes, nil
}
```

## Core Logic Skeletons

```
// internal/evaluation/engine.go - Core evaluation logic to implement GO

package evaluation

import (
    "crypto/sha256"
    "encoding/binary"
    "fmt"
    "math"

    "flagsystem/internal/common"
    "flagsystem/internal/storage"
)

// EvaluationEngine implements the core flag evaluation logic

type EvaluationEngine struct {
    flagStore storage.FlagRepository
    cache     *FlagCache
}

// NewEvaluationEngine creates a new evaluation engine

func NewEvaluationEngine(flagStore storage.FlagRepository) *EvaluationEngine {
    return &EvaluationEngine{
        flagStore: flagStore,
        cache:     NewFlagCache(),
    }
}

// EvaluateFlag processes targeting rules and returns appropriate variant
```

```
func (e *EvaluationEngine) EvaluateFlag(flagKey common.FlagKey, context common.UserContext) common.EvaluationResult {

    // TODO 1: Validate input parameters - check flagKey is not empty, context has valid
    UserID

    //           Return error result if validation fails


    // TODO 2: Retrieve flag definition from cache, falling back to storage if cache miss

    //           Use e.cache.GetFlag(flagKey) first, then e.flagStore.GetFlag(flagKey) as
    fallback

    // TODO 3: Check if flag is enabled - if disabled, return default variant with reason
    "flag_disabled"

    // TODO 4: Evaluate targeting rules in precedence order using evaluateTargetingRules()

    //           Rules should be processed from most specific to most general


    // TODO 5: If no targeting rules match, apply percentage rollout using consistent
    hashing

    //           Call calculateUserBucket() to get user's bucket assignment


    // TODO 6: If no percentage rule matches, return default variant from flag definition

    //           Include detailed reason explaining why this fallback was used


    // TODO 7: Construct and return EvaluationResult with variant key, resolved value,
    evaluation reason

    return common.EvaluationResult{
        FlagKey: flagKey,
        Value:   nil, // TODO: Replace with actual resolved value
        Variant: "", // TODO: Replace with selected variant key
    }
}
```

```

    Reason: "", // TODO: Replace with evaluation reason

    Source: "evaluation_engine",

}

}

// evaluateTargetingRules processes flag rules against user context

func (e *EvaluationEngine) evaluateTargetingRules(rules []storage.TargetingRule, context common.UserContext) (string, interface{}, string, bool) {

    // TODO 1: Iterate through rules in order (rules are pre-sorted by precedence)

    // TODO 2: For each rule, check if it's enabled - skip disabled rules

    // TODO 3: Evaluate rule conditions using evaluateRuleConditions()

    //           Handle AND/OR operators correctly based on rule.Operator

    // TODO 4: If rule matches, apply percentage allocation within the rule

    //           Use consistent hashing to select variant from rule's allocation

    // TODO 5: Return selected variant key, resolved value, and evaluation reason

    //           Reason should indicate which rule matched (e.g.,
    //           "targeting_rule:user_segment")

    return "", nil, "", false // TODO: Replace with actual values
}

// evaluateRuleConditions checks if user context satisfies rule conditions

func (e *EvaluationEngine) evaluateRuleConditions(conditions []storage.Condition, operator string, context common.UserContext) bool {

    // TODO 1: Handle empty conditions array (should return true - matches all users)
}

```

```
// TODO 2: Evaluate each condition using evaluateCondition()

//           Store boolean result for each condition


// TODO 3: Apply logical operator (AND/OR) to combine condition results

//           AND: all conditions must be true

//           OR: at least one condition must be true


// TODO 4: Return final boolean result

return false // TODO: Replace with actual logic

}

// evaluateCondition checks a single condition against user context

func (e *EvaluationEngine) evaluateCondition(condition storage.Condition, context common.UserContext) bool {

    // TODO 1: Get attribute value from context.Attributes using condition.Attribute

    //           Handle missing attributes (return false for most operators)


    // TODO 2: Handle special attributes like "user_id" and "segments"

    //           user_id should use context.UserID value

    //           segments should check if condition.Value is in context.Segments


    // TODO 3: Apply condition operator:

    //           "equals": attribute == condition.Value

    //           "not_equals": attribute != condition.Value

    //           "in": attribute is in condition.Value (array)

    //           "not_in": attribute is not in condition.Value (array)
```

```
//           "contains": string(attribute) contains string(condition.value)

//           "starts_with": string(attribute) starts with string(condition.value)

//           "greater_than": numeric comparison

//           "less_than": numeric comparison

// TODO 4: Handle type conversions and edge cases gracefully

//           String comparisons should be case-sensitive

//           Numeric comparisons should handle int/float conversions

return false // TODO: Replace with actual logic

}

// calculateUserBucket determines user's allocation bucket using consistent hashing

func (e *EvaluationEngine) calculateUserBucket(userID common.UserID, flagKey
common.FlagKey) int {

    // TODO 1: Create hash input by combining userID and flagKey

    //           Use format like "userID:flagKey" to ensure uniqueness across flags

    // TODO 2: Calculate SHA-256 hash of the input string

    //           Use crypto/sha256 package for cryptographically secure hashing

    // TODO 3: Convert first 8 bytes of hash to uint64 using binary.BigEndian

    //           This provides consistent numeric value from hash bytes

    // TODO 4: Map hash to percentage bucket (0-99) using modulo operation

    //           Ensure uniform distribution across the 0-99 range

    // HINT: Consistent hashing ensures the same user always gets the same bucket
```

```

    // for a given flag, even across multiple evaluation calls

    return 0 // TODO: Replace with calculated bucket
}

// resolveVariantValue gets the configured value for a variant key

func (e *EvaluationEngine) resolveVariantValue(variants []common.Variant, variantKey string) (interface{}, error) {

    // TODO 1: Search through variants array to find variant with matching Key

    // TODO 2: Return the variant's Value field if found

    // TODO 3: Return error if variant key not found in variants array
    //           Include both the sought key and available keys in error message

    return nil, fmt.Errorf("variant not found: %s", variantKey) // TODO: Replace with actual logic
}

```

## Milestone Checkpoints

### After Milestone 1 (Flag Evaluation Engine):

- Run `go test ./internal/evaluation/...` - all tests should pass
- Create a simple flag with boolean variants and percentage rollout
- Verify that the same user ID always gets the same variant across multiple evaluations
- Test rule evaluation with user segments and attribute conditions
- Check that evaluation completes in under 1 millisecond for cached flags

### After Milestone 2 (Real-time Flag Updates):

- Start the update service with `go run cmd/updateservice/main.go`
- Connect a test client and verify SSE connection establishment
- Update a flag configuration and confirm the change propagates within 5 seconds
- Disconnect the client, update flags, reconnect, and verify state synchronization

- Monitor connection metrics to ensure proper cleanup of disconnected clients

#### **After Milestone 3 (Flag Analytics & Experiments):**

- Run an A/B test with two variants and record exposure events
- Generate test conversion data and verify statistical significance calculation
- Check that exposure events are batched and uploaded without blocking evaluations
- Verify experiment reports show confidence intervals and p-values
- Test sample ratio mismatch detection with uneven traffic allocation

## **Data Model**

**Milestone(s):** This section establishes the data structures for Milestone 1 (Flag Evaluation Engine), Milestone 2 (Real-time Flag Updates), and Milestone 3 (Flag Analytics & Experiments) by defining how flags, rules, user contexts, and experiments are modeled and stored.

Think of the data model as the **blueprint for a sophisticated air traffic control system**. Just as air traffic control needs detailed information about each aircraft (flight number, destination, altitude, speed), weather conditions (wind patterns, visibility), and flight rules (priority levels, restricted airspace), our feature flag system needs comprehensive data structures to make intelligent routing decisions about which users receive which features.





The air traffic control analogy extends to the relationships between data entities. Aircraft have dependencies on runways, weather affects multiple flights simultaneously, and control tower decisions must be logged for safety audits. Similarly, feature flags have complex relationships with targeting rules, user contexts affect multiple flag evaluations, and every flag decision must be tracked for analysis and compliance.

Our data model must support three critical capabilities that parallel air traffic control operations. First, **real-time decision making** requires rich context about each "aircraft" (user) and current "weather conditions" (system state) to route traffic safely. Second, **historical tracking** demands comprehensive logging of every decision for post-incident analysis and performance optimization. Third, **predictive planning** needs structured experiment data to forecast the impact of routing changes before they affect live traffic.

## Flag Definition Structure

The flag definition structure serves as the **master flight plan** for each feature, containing all the information needed to make consistent routing decisions across millions of users. Just as a flight plan specifies departure gates, flight paths, alternate routes, and emergency procedures, a flag definition specifies variants, targeting rules, fallback behavior, and operational metadata.

The core challenge in modeling flag definitions lies in supporting **flexible targeting without sacrificing evaluation performance**. Simple boolean flags might seem sufficient initially, but production systems quickly demand percentage rollouts, user segmentation, geographic targeting, and complex rule combinations. Our data model must accommodate this complexity while ensuring that flag evaluation remains fast enough for real-time request processing.

### Decision: Hierarchical Rule Structure with Explicit Precedence

- **Context:** Flag targeting can involve multiple overlapping rules (user attributes, segments, percentage rollouts), and the order of evaluation affects which users receive which variants
- **Options Considered:**
  1. Flat rule list with implicit precedence
  2. Hierarchical rules with explicit priority numbers
  3. Single rule type with complex nested conditions
- **Decision:** Hierarchical rules with explicit precedence ordering
- **Rationale:** Explicit precedence prevents non-deterministic evaluation when rules overlap, hierarchical structure supports complex logic without performance penalties, and clear separation enables easier debugging and rule management
- **Consequences:** Enables predictable evaluation behavior and supports complex targeting scenarios, but requires careful precedence management and slightly more complex rule authoring

<b>Rule Precedence</b>	<b>Rule Type</b>	<b>Purpose</b>	<b>Example Use Case</b>
1 (Highest)	User Override Rules	Target specific users or segments	Enable beta feature for internal team
2	Attribute-Based Rules	Target based on user properties	Show premium features to paid users
3	Percentage Rules	Gradual rollouts with consistent assignment	Release to 25% of users in US region
4 (Lowest)	Default Rule	Fallback when no other rules match	Disable feature for remaining users

The `FlagDefinition` structure captures this hierarchical approach while maintaining evaluation efficiency. Each flag contains multiple rule layers that are processed in strict precedence order, ensuring deterministic outcomes regardless of rule complexity or user context variations.

Field	Type	Description
Key	FlagKey	Unique identifier for the flag across all environments
Name	string	Human-readable name for management interface display
Description	string	Purpose and context documentation for team collaboration
Enabled	bool	Master switch to disable flag evaluation without deleting
Type	string	Value type: "boolean", "string", "number", "json"
Variants	[]Variant	Available variants with keys, values, and allocation weights
TargetingRules	[]TargetingRule	Ordered list of conditional rules with explicit precedence
PercentageRule	PercentageRule	Default percentage allocation when no targeting rules match
DefaultVariant	string	Fallback variant key when all rules fail or flag is disabled
Prerequisites	[]FlagKey	Other flags that must be enabled before evaluating this flag
CreatedAt	time.Time	Flag creation timestamp for audit trail
UpdatedAt	time.Time	Last modification timestamp for cache invalidation
CreatedBy	string	User identifier who created the flag
UpdatedBy	string	User identifier who last modified the flag
Tags	[]string	Categorical labels for organization and filtering
Archived	bool	Soft deletion marker for flags no longer in use

The `Variant` structure represents each possible outcome of a flag evaluation, containing not just the value but also metadata needed for consistent allocation and debugging. Think of variants as different **flight destinations** that users can be routed to based on the evaluation rules.

Field	Type	Description
Key	string	Unique identifier for the variant within the flag
Value	interface{}	Actual value returned to client (string, number, boolean, or JSON object)
Weight	int	Relative weight for percentage allocation (higher numbers = larger allocation)
Name	string	Human-readable name for management interface
Description	string	Documentation of variant purpose and expected behavior

Targeting rules provide the **conditional logic engine** that determines which users receive which variants. Each rule represents a specific targeting strategy with its own conditions and variant allocation scheme.

Field	Type	Description
ID	string	Unique identifier for the rule within the flag
Description	string	Human-readable explanation of rule purpose
Conditions	[]Condition	List of attribute-based conditions that must be satisfied
ConditionOperator	string	Logical operator combining conditions: "AND" or "OR"
VariantAllocations	[]VariantAllocation	How to distribute users matching this rule across variants
Enabled	bool	Switch to temporarily disable rule without deletion

Individual conditions within targeting rules specify **attribute-based matching criteria** that are evaluated against the user context. The condition model supports various comparison operations to handle different attribute types and matching scenarios.

Field	Type	Description
Attribute	string	User context attribute name to evaluate (e.g., "email", "plan", "region")
Operator	string	Comparison operator: "equals", "in", "contains", "greater_than", "less_than"
Values	[]interface{}	List of values to compare against (type matches operator requirements)
Negate	bool	Whether to invert the condition result (NOT operation)

The percentage rule provides **default allocation behavior** when no targeting rules match the user context. This ensures every user receives a variant assignment through consistent hashing, even when they don't match any specific targeting criteria.

Field	Type	Description
VariantAllocations	[]VariantAllocation	Distribution of unmatched users across variants
Enabled	bool	Whether to apply percentage rule or return default variant

Variant allocations specify how users matching a rule are distributed across available variants using **deterministic percentage splits**. The allocation system ensures stable assignment where users consistently receive the same variant across multiple evaluations.

Field	Type	Description
VariantKey	string	Key of variant to assign
Percentage	int	Percentage of matched users to assign this variant (0-100)
RangeStart	int	Starting bucket number for this allocation (computed from percentage)
RangeEnd	int	Ending bucket number for this allocation (computed from percentage)

The critical insight for percentage allocation is that percentages are converted to **bucket ranges** during flag creation, not during evaluation. This preprocessing enables fast evaluation by checking which bucket a user falls into rather than performing percentage calculations on every request.

### Common Rule Modeling Pitfalls:

**⚠ Pitfall: Overlapping Percentage Allocations** Many implementations allow variant allocations that sum to more or less than 100%, causing non-deterministic assignment or users falling through cracks. Always validate that percentages sum exactly to 100% within each rule and use bucket ranges to ensure complete coverage.

**⚠ Pitfall: Circular Flag Prerequisites** Flag A depends on Flag B which depends on Flag A, creating infinite loops during evaluation. Implement dependency graph validation during flag creation to detect cycles, and consider topological sorting for complex dependency chains.

**⚠ Pitfall: Implicit Rule Precedence** Without explicit ordering, rules evaluated in database insertion order or hash map iteration order produce different results across system restarts. Always maintain explicit rule precedence and process rules in consistent order during evaluation.

## User Context and Segmentation

User context serves as the **aircraft identification and flight characteristics** that enable intelligent routing decisions. Just as air traffic controllers need to know aircraft type, destination, fuel levels, and passenger count to make safe routing decisions, the feature flag system needs comprehensive user attributes to apply targeting rules effectively.

The challenge in context modeling lies in balancing **flexibility with performance**. User contexts must support arbitrary attributes for complex targeting scenarios, but evaluation latency requirements demand efficient attribute lookup and comparison operations. Additionally, contexts must accommodate both real-time attributes (current location, device type) and slowly-changing attributes (subscription plan, account creation date) without compromising evaluation consistency.

## Decision: Flat Attribute Map with Type Inference

- **Context:** User contexts need to support arbitrary attributes with different data types, but complex nested structures slow down rule evaluation and serialization
- **Options Considered:**
  1. Strongly-typed context with predefined fields
  2. Nested JSON object with hierarchical attributes
  3. Flat map with runtime type inference
- **Decision:** Flat attribute map with interface{} values and runtime type inference
- **Rationale:** Flat structure enables fast attribute lookup, interface{} values support all JSON-serializable types, runtime inference handles type coercion for rule operators, and simple structure simplifies SDK integration
- **Consequences:** Supports arbitrary attributes without schema constraints and enables efficient evaluation, but requires careful type handling and limits nesting capabilities

The `UserContext` structure captures all information needed to evaluate targeting rules against a specific user. This context travels with every flag evaluation request and serves as the primary input to the rule matching engine.

Field	Type	Description
UserID	UserID	Stable user identifier for consistent variant assignment across sessions
Attributes	map[string]interface{}	Key-value pairs representing user characteristics and properties
Segments	[]string	Pre-computed segment memberships for efficient group-based targeting
RequestContext	map[string]interface{}	Request-specific attributes (IP address, user agent, timestamp)
SessionID	string	Session identifier for request grouping and analytics attribution
AnonymousID	string	Anonymous identifier for users without stable UserID
CustomProperties	map[string]interface{}	Application-specific context not fitting standard categories

The user ID serves as the **aircraft tail number** - a stable, unique identifier that ensures consistent variant assignment through the consistent hashing algorithm. This identifier must remain constant across user sessions and devices to prevent users from receiving different variants as they interact with the system.

User attributes provide the **flight characteristics and metadata** needed for sophisticated targeting scenarios. These attributes can represent user demographics, subscription details, behavioral data, device information, or any other criteria relevant for feature targeting decisions.

| Attribute Category | Example Attributes | Typical Use Cases | ---|---|---|---| Demographics | age, country, language, timezone | Geographic rollouts, localization features | | Subscription | plan, trial\_end\_date, payment\_method | Premium feature access, billing experiments | | Behavioral | login\_count, last\_active, feature\_usage | Engagement experiments, onboarding flows | | Technical | device\_type, browser, app\_version | Platform-specific features, compatibility testing | | Business | company\_size, industry, account\_tier | B2B feature targeting, enterprise experiments |

User segments represent **pre-computed group memberships** that enable efficient targeting of user cohorts without evaluating complex membership criteria on every flag request. Think of segments as **flight categories** (commercial, cargo, emergency) that group aircraft with similar handling requirements.

Segments provide significant performance advantages over dynamic attribute-based targeting because segment membership is computed asynchronously and cached in the user context. This approach trades some real-time accuracy for substantial evaluation speed improvements, particularly important for high-traffic applications.

Segment Type	Computation Strategy	Update Frequency	Example Segments
Static Segments	Manual user assignment	On-demand updates	beta_users, internal_team, vip_customers
Dynamic Segments	Batch computation from user attributes	Daily or hourly refresh	high_engagement, trial_expiring, mobile_users
Real-time Segments	Streaming computation from user events	Continuous updates	recently_active, shopping_cart_abandoned

The `UserContext` also captures request-specific information that affects flag evaluation but doesn't represent persistent user characteristics. This request context enables targeting based on current conditions rather than historical user properties.

Request context attributes include information like client IP address for geographic targeting, user agent for device detection, request timestamp for time-based experiments, and feature flags for cross-flag dependencies. These attributes have shorter lifespans than user attributes but play crucial roles in sophisticated targeting scenarios.

**Key Design Insight:** The distinction between user attributes and request context is critical for **caching strategy**. User attributes can be cached longer because they change infrequently, while request context must be computed fresh for each evaluation because it represents current conditions.

## Context Integration Patterns:

The user context integrates with the broader system through several key patterns that affect both performance and functionality. Understanding these patterns helps avoid common implementation mistakes and

architectural issues.

**Context Enrichment** involves augmenting basic user identification with additional attributes from user databases, analytics systems, or third-party services. This enrichment can happen synchronously during flag evaluation (trading latency for freshness) or asynchronously through background processes (trading freshness for performance).

**Context Caching** stores enriched user contexts to avoid repeated database lookups during flag evaluation. Effective caching strategies must balance memory usage, cache hit rates, and data freshness while handling cache invalidation when user attributes change.

**Context Versioning** addresses the challenge of evolving user schemas without breaking existing flag configurations. As applications add new user attributes or modify existing ones, the flag system must gracefully handle missing attributes and type mismatches in targeting rules.

#### Common Context Modeling Pitfalls:

**⚠ Pitfall: Missing User ID Fallback Strategy** Systems that require stable user IDs for consistent hashing break when evaluating flags for anonymous users or during authentication failures. Always provide anonymous ID generation and graceful degradation when stable identifiers are unavailable.

**⚠ Pitfall: Context Attribute Type Confusion** Storing numeric values as strings breaks greater-than/less-than operators in targeting rules, while storing strings as numbers breaks substring matching. Implement consistent type inference and validation to prevent rule evaluation errors.

**⚠ Pitfall: Overly Large Context Objects** Including too many attributes or large nested objects in user context increases serialization overhead and network latency. Focus on attributes actually used in targeting rules and consider separate attribute fetching for rarely-used properties.

**⚠ Pitfall: Stale Segment Memberships** Pre-computed segments that update infrequently can cause users to receive inappropriate variants when their characteristics change. Balance segment freshness requirements with computation overhead, and provide manual refresh capabilities for critical segments.

## Implementation Guidance

The data model implementation requires careful attention to serialization, validation, and performance characteristics. The following guidance helps translate the conceptual data structures into efficient, maintainable code.

### A. Technology Recommendations:

Component	Simple Option	Advanced Option
Data Storage	SQLite with JSON columns	PostgreSQL with JSONB indexing
Serialization	encoding/json with struct tags	Protocol Buffers with code generation
Validation	Manual field checking	JSON Schema validation library
Type Safety	interface{} with runtime checks	Generic types with constraints

## B. Recommended File Structure:

```

internal/
  model/
    flag.go           ← FlagDefinition and related structures
    context.go        ← UserContext and attribute handling
    evaluation.go     ← EvaluationResult and response types
    experiment.go     ← Experiment and analytics structures
    validation.go     ← Data validation and constraint checking
    serialization.go  ← JSON marshaling and unmarshaling helpers
  storage/
    flag_store.go    ← Flag persistence interface and implementations
    segment_store.go  ← User segment storage and retrieval

```

## C. Infrastructure Starter Code:

Complete validation framework for ensuring data integrity across all model structures:

```
package model

import (
    "fmt"
    "strings"
    "time"
)

// ValidationError represents a data model validation failure

type ValidationError struct {

    Field    string
    Value    interface{}
    Message  string
}

func (e ValidationError) Error() string {
    return fmt.Sprintf("validation failed for field %s: %s", e.Field, e.Message)
}

// ValidationErrors aggregates multiple validation failures

type ValidationErrors []ValidationError

func (e ValidationErrors) Error() string {
    if len(e) == 0 {
        return "no validation errors"
    }

    var messages []string
    for _, err := range e {
        messages = append(messages, err.Error())
    }
}
```

GO

```
}

    return strings.Join(messages, "; ")

}

func (e ValidationErrors) HasErrors() bool {

    return len(e) > 0

}

// ValidateFlag performs comprehensive validation of flag definitions

func ValidateFlag(flag *FlagDefinition) error {

    var errors ValidationErrors

    // TODO: Implement validation logic

    return nil

}

// ValidateUserContext ensures context structure meets requirements

func ValidateUserContext(ctx *UserContext) error {

    var errors ValidationErrors

    // TODO: Implement context validation

    return nil

}
```

Complete serialization helpers that handle type conversion and JSON marshaling:

```
package model

import (
    "encoding/json"

    "reflect"

    "strconv"
)

// AttributeValue provides type-safe access to user context attributes

type AttributeValue struct {

    raw interface{}
}

func NewAttributeValue(value interface{}) AttributeValue {
    return AttributeValue{raw: value}
}

func (av AttributeValue) String() (string, bool) {
    // TODO: Implement string conversion with type checking
    return "", false
}

func (av AttributeValue) Int() (int64, bool) {
    // TODO: Implement integer conversion with type checking
    return 0, false
}

func (av AttributeValue) Float() (float64, bool) {
    // TODO: Implement float conversion with type checking
    return 0, false
}
```

GO

```
}

func (avAttributeValue) Bool() (bool, bool) {

    // TODO: Implement boolean conversion with type checking

    return false, false
}

func (avAttributeValue) StringSlice() ([]string, bool) {

    // TODO: Implement string slice conversion for "in" operator

    return nil, false
}
```

#### D. Core Logic Skeleton Code:

```
package model
```

GO

```
// ValidateFlag ensures flag definition meets all requirements

func ValidateFlag(flag *FlagDefinition) error {

    var errors ValidationErrors

    // TODO 1: Validate flag key is non-empty and follows naming conventions

    // TODO 2: Ensure at least one variant is defined and default variant exists

    // TODO 3: Validate all targeting rule conditions reference valid operators

    // TODO 4: Check percentage allocations sum to 100% in each rule

    // TODO 5: Detect circular dependencies in prerequisites

    // TODO 6: Verify variant weights are positive integers

    // TODO 7: Validate attribute names in conditions don't contain reserved characters


    if errors.HasErrors() {

        return errors

    }

    return nil

}

// ValidateUserContext checks context structure and attribute types

func ValidateUserContext(ctx *UserContext) error {

    var errors ValidationErrors

    // TODO 1: Ensure either UserID or AnonymousID is present

    // TODO 2: Validate attribute values are JSON-serializable types

    // TODO 3: Check segment names follow naming conventions

    // TODO 4: Verify request context doesn't contain oversized values
```

```

// TODO 5: Validate session ID format if present


if errors.HasErrors() {

    return errors

}

return nil

}

// DetectCircularDependencies finds cycles in flag prerequisite relationships

func DetectCircularDependencies(flags map[FlagKey]*FlagDefinition) error {

    // TODO 1: Build adjacency list representation of prerequisite graph

    // TODO 2: Perform depth-first search from each flag node

    // TODO 3: Track visited and recursion stack to detect back edges

    // TODO 4: Return specific error identifying circular dependency path

    // Hint: Use topological sorting or DFS with color marking

    return nil

}

```

## E. Language-Specific Hints:

- Use `json:"-"` struct tags to exclude sensitive fields from JSON serialization
- Implement `json.Marshaler` and `json.Unmarshaler` interfaces for custom serialization logic
- Use `reflect.TypeOf()` carefully for type checking - prefer type assertions when possible
- Store percentage ranges as `[start, end)` intervals where start is inclusive and end is exclusive
- Use `sync.RWMutex` for concurrent access to flag definitions if caching in memory
- Consider `time.Time.UTC()` for consistent timezone handling in timestamps

## F. Milestone Checkpoint:

After implementing the data model structures:

1. **Validation Test:** Run `go test ./internal/model/...` - all validation functions should correctly reject invalid flags and accept valid configurations

2. **Serialization Test:** Create a complex flag definition, marshal to JSON, unmarshal back, and verify all fields are preserved correctly
3. **Type Conversion Test:** Create user context with mixed attribute types, verify AttributeValue conversions work for all supported operators
4. **Dependency Detection:** Create flags with circular prerequisites, verify the detection algorithm identifies the cycle and provides helpful error messages

Expected behavior after completion:

- Flag definitions serialize to clean JSON suitable for storage and API responses
- User contexts handle missing attributes gracefully during rule evaluation
- Validation catches common configuration errors before flags are stored
- Type conversion supports all comparison operators used in targeting rules

Signs something is wrong:

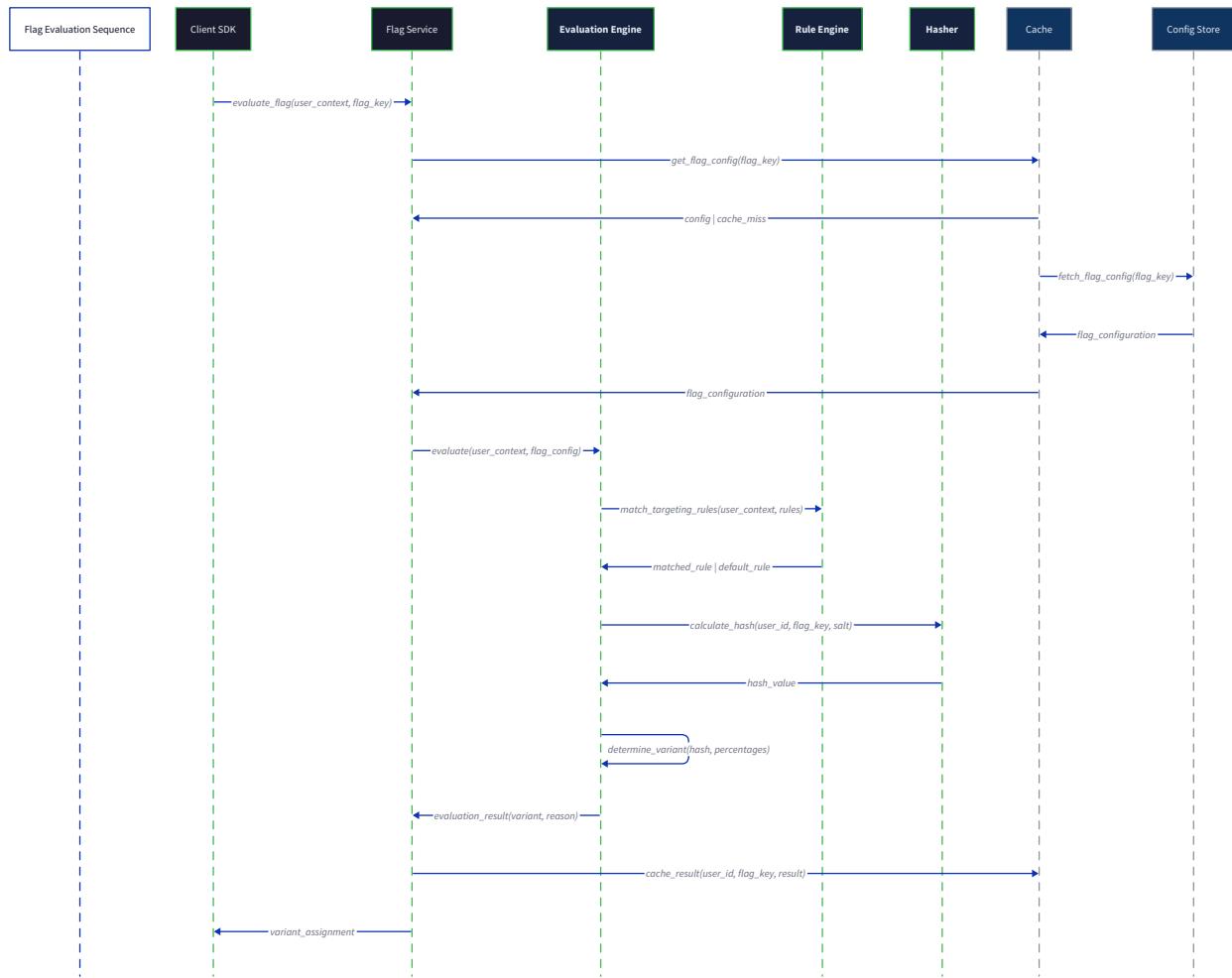
- Serialized flags lose information during round-trip JSON conversion
- Validation allows invalid configurations that break evaluation engine
- Type conversion produces inconsistent results for the same input values
- Circular dependency detection has false positives or misses actual cycles

## Flag Evaluation Engine

**Milestone(s):** This section covers Milestone 1 (Flag Evaluation Engine) by implementing the core flag evaluation logic with consistent user assignment, complex rule processing, and proper fallback handling.

The flag evaluation engine serves as the heart of the feature flag system, responsible for taking a user context and flag configuration and determining which variant that user should receive. Think of the evaluation engine as a **sophisticated restaurant host** - when a customer (user) arrives, the host considers their reservation details, party size, dietary restrictions, and membership status (user context) against the restaurant's current seating rules, special promotions, and capacity constraints (flag rules) to determine the best table assignment (variant) while ensuring repeat customers get consistent treatment.

The evaluation engine must balance several competing requirements: it needs to be extremely fast since it's called on every feature check, deterministic so users get consistent experiences, flexible enough to support complex targeting rules, and resilient enough to gracefully handle edge cases and partial failures. The architecture prioritizes **evaluation-first design**, meaning every component decision is optimized for the critical path of flag evaluation performance.



The evaluation process follows a structured decision tree that processes rules in order of precedence, applies consistent hashing for percentage-based rollouts, and provides detailed reasoning for audit and debugging purposes. This deterministic approach ensures that the same user context always produces the same result, which is critical for maintaining consistent user experiences across multiple application instances.

## Consistent User Assignment

Consistent user assignment is the foundation that ensures users receive the same variant across multiple evaluations, even as flag configurations change or the system scales horizontally. Without consistency, users would experience jarring feature flip-flopping as they navigate through an application - imagine if a user's shopping cart suddenly changed its interface mid-checkout because they triggered a different variant.

The system achieves consistency through **consistent hashing**, which maps each user to a deterministic position in a virtual "allocation space" that remains stable regardless of when or where the evaluation occurs. Think of this allocation space as a circular dartboard numbered 0 to 99, where each user's `UserID` and `FlagKey` combination determines exactly where their dart lands, and the variant allocations define which sections of the dartboard correspond to which variants.

Hash Function Component	Purpose	Example Value
UserID	Stable user identifier ensuring same user gets same position	"user_12345"
FlagKey	Flag-specific salt preventing correlation across flags	"checkout_redesign"
Combined Input	Hash input ensuring flag-specific consistency	"user_12345:checkout_redesign"
Hash Algorithm	CRC32 or SHA256 providing uniform distribution	CRC32
Bucket Number	Final position in 0-99 allocation space	67

The `calculateUserBucket` function transforms a user's identity into a bucket number between 0 and 99, providing 1% granularity for percentage rollouts. This granularity strikes the optimal balance between precision and performance - finer granularity would provide minimal benefit while consuming more computation and memory.

**Design Insight:** Using both `UserID` and `FlagKey` in the hash input is crucial because it prevents users from being consistently assigned to the same relative position across all flags. Without the flag-specific salt, a user who falls into the "early adopter" bucket for one flag would fall into early adopter buckets for all flags, creating unwanted correlation effects.

**Bucket Range Allocation** determines which bucket ranges correspond to each variant based on the configured weights. For a flag with variants A (weight: 30), B (weight: 20), and C (weight: 50), the system assigns buckets 0-29 to variant A, buckets 30-49 to variant B, and buckets 50-99 to variant C. This approach ensures that percentage changes affect users at the margins rather than causing random reassignments.

Variant	Weight	Bucket Range	Users Affected by 10% Increase
Control	50%	0-49	New users in buckets 50-59
Treatment	40%	50-89	New users in buckets 90-99
Holdout	10%	90-99	Users in buckets 90-99 reassigned

The allocation algorithm processes variant weights in declaration order, ensuring deterministic bucket assignments even when weights don't sum to exactly 100%. If weights sum to less than 100%, the remaining users receive the flag's default value. If weights exceed 100%, the system normalizes proportionally and logs a warning for operational visibility.

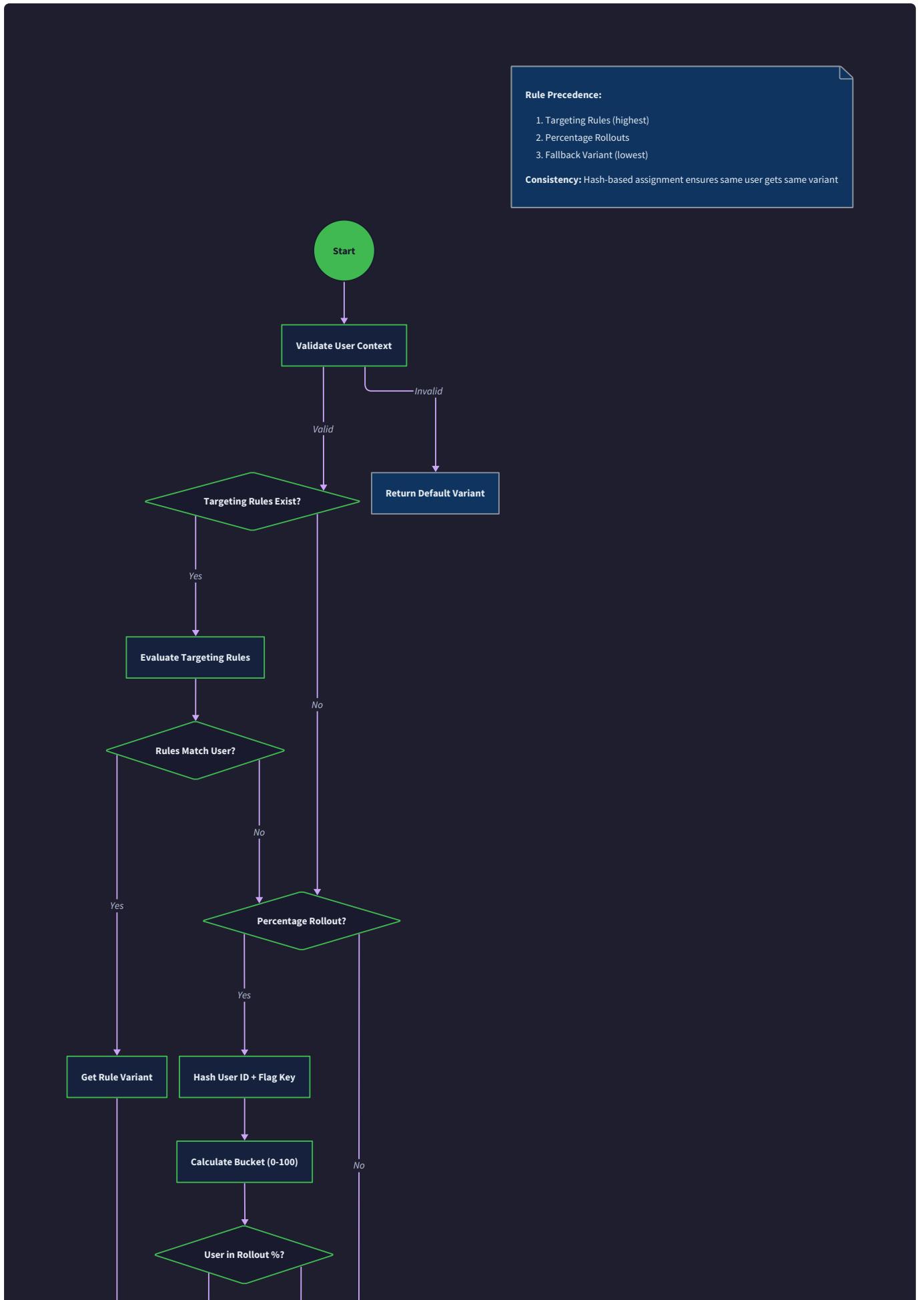
**Critical Stability Property:** Once a user receives a variant assignment, they must continue receiving that assignment until explicitly moved through configuration changes. The consistent hashing approach guarantees this stability by ensuring the hash function remains deterministic across evaluations, even as the application restarts or scales.

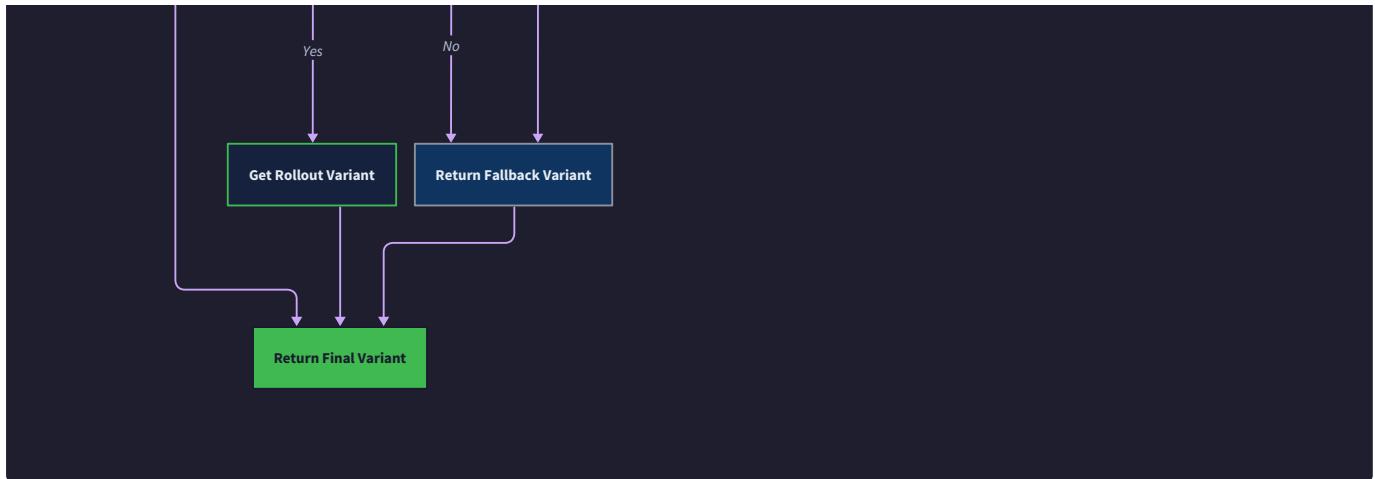
The system handles several edge cases to maintain assignment stability:

1. **Zero-weight variants** remain in the configuration but receive no bucket allocation, allowing for quick re-enabling without reassignment
2. **Weight normalization** proportionally adjusts weights that exceed 100% while preserving relative ratios
3. **Default fallback** handles users whose buckets fall outside allocated ranges due to configuration errors

## Rule Evaluation Logic

The rule evaluation logic implements a sophisticated decision tree that processes targeting rules in strict precedence order, ensuring deterministic outcomes when multiple rules could potentially match a user. Think of rule evaluation as a **courthouse with multiple judges** - cases (user evaluations) are presented to judges (targeting rules) in order of seniority, and the first judge who claims jurisdiction (matching conditions) renders the final verdict (variant assignment).





The evaluation algorithm follows a waterfall approach with four distinct phases, each with specific responsibilities and fallback behavior:

Evaluation Phase	Purpose	Input Requirements	Success Criteria
Targeting Rules	User-specific overrides	UserContext with attributes	All rule conditions evaluate to true
Percentage Rules	Gradual rollout control	UserID for consistent hashing	User bucket falls within variant range
Default Value	Configuration fallback	Flag definition	Default value exists
System Fallback	Emergency degradation	None	Always succeeds with false/null

**Targeting Rule Processing** begins with the highest-priority rule and evaluates each rule's conditions against the provided user context. Each `TargetingRule` contains a collection of `Condition` objects combined with either `AND` or `OR` logic, allowing for sophisticated user segmentation based on multiple attributes.

The `evaluateRuleConditions` function processes each condition within a rule, applying the specified logical operator to determine overall rule satisfaction. For `AND` operations, all conditions must evaluate to true; for `OR` operations, any single condition can satisfy the rule. This flexibility enables complex targeting scenarios like "show premium features to users in the US OR Canada AND with subscription tier gold OR platinum".

Condition Operator	Purpose	Example Usage	Type Requirements
equals	Exact match comparison	country equals "US"	String, number, boolean
in	Array membership testing	user_id in ["admin1", "admin2"]	Any type with array
contains	Substring matching	email contains "@company.com"	String values only
greater_than	Numeric comparison	account_age greater_than 30	Numeric values only
less_than	Numeric comparison	session_count less_than 5	Numeric values only

The `AttributeValue` wrapper provides type-safe access to user context attributes while handling the common problem of dynamic typing in user data. Each attribute access method returns both the converted value and a boolean indicating whether the conversion succeeded, allowing the evaluation logic to gracefully handle type mismatches without panicking.

**Rule Precedence and Ordering** ensures deterministic evaluation when multiple rules could match the same user. Rules are processed in declaration order within the flag definition, with the first matching rule taking precedence. This approach provides predictable behavior and allows flag administrators to structure rules from specific to general, similar to firewall rule processing.

**Rule Evaluation Example:**

1. IF user\_id in ["beta\_user\_1", "beta\_user\_2"] THEN variant="beta"
2. IF country equals "US" AND subscription="premium" THEN variant="premium\_us"
3. IF country equals "US" THEN variant="standard\_us"
4. PERCENTAGE ROLLOUT: 20% get variant="treatment", 80% get variant="control"

In this example, beta users always receive the beta variant regardless of location or subscription status, premium US users receive premium\_us unless they're beta users, and other US users receive standard\_us. Only users who don't match any targeting rule enter the percentage rollout phase.

**Percentage Rule Evaluation** activates when no targeting rules match the user context. The system calculates the user's bucket using consistent hashing and determines which variant allocation range contains that bucket. This phase implements gradual rollouts and A/B testing by assigning users to variants based on stable, pseudorandom distribution.

The `PercentageRule` defines variant allocations as a list of `VariantAllocation` objects, each specifying a variant key and weight. The evaluation engine converts weights into cumulative bucket ranges, ensuring efficient lookup during evaluation.

Variant Allocation	Weight	Cumulative Weight	Bucket Range
control	60	60	0-59
treatment_a	25	85	60-84
treatment_b	15	100	85-99

**Default Value Fallback** provides the safety net when both targeting rules and percentage rules fail to assign a variant. This situation occurs when targeting rules don't match and percentage rules have zero total weight or the user's bucket falls outside allocated ranges due to configuration errors.

The `EvaluationResult` includes a detailed `Reason` field that traces the evaluation path, enabling debugging and audit capabilities. Reason codes follow a structured format that indicates which evaluation phase produced the result and why.

Reason Code	Evaluation Path	Example Scenario
targeting_rule_match	Targeting rule satisfied	"targeting_rule_match:rule_2"
percentage_rollout	Percentage allocation	"percentage_rollout:treatment_20pct"
default_value	Fallback to default	"default_value:no_rules_matched"
flag_not_found	Missing flag definition	"flag_not_found:invalid_key"
evaluation_error	Processing failure	"evaluation_error:invalid_context"

## Architecture Decision Records

The flag evaluation engine embodies several critical architectural decisions that fundamentally shape system behavior, performance, and maintainability. Each decision represents a careful trade-off between competing requirements and establishes constraints that influence the entire system design.

## Decision: Consistent Hashing Algorithm Selection

- Context:** Users must receive consistent variant assignments across evaluations, but the hashing algorithm must balance performance, distribution quality, and collision resistance for production workloads.
- Options Considered:** CRC32 for speed, SHA256 for cryptographic strength, MD5 for legacy compatibility
- Decision:** CRC32 as the primary algorithm with SHA256 as a configurable alternative
- Rationale:** CRC32 provides excellent distribution uniformity and high performance for non-adversarial inputs, while SHA256 offers cryptographic strength for compliance-sensitive environments at the cost of 3-4x slower computation
- Consequences:** Enables sub-microsecond evaluation performance while maintaining uniform user distribution, but requires SHA256 fallback for environments with strict cryptographic requirements

Hash Algorithm	Performance	Distribution Quality	Collision Resistance	Chosen
CRC32	Excellent (ns)	Uniform for random inputs	Weak against adversaries	Primary
SHA256	Good (μs)	Cryptographically uniform	Strong against adversaries	Optional
MD5	Very Good (ns)	Good but deprecated	Weak against adversaries	No

## Decision: Rule Evaluation Precedence Strategy

- Context:** Multiple targeting rules may match the same user, requiring a deterministic resolution strategy that balances flexibility with predictability.
- Options Considered:** Priority scoring system, first-match-wins ordering, most-specific-rule-wins analysis
- Decision:** First-match-wins with declaration order precedence
- Rationale:** Declaration order provides intuitive, predictable behavior that mirrors firewall rules and allows administrators to structure rules from specific to general without complex priority calculations
- Consequences:** Simplifies rule authoring and debugging at the cost of requiring careful rule ordering, with no automatic optimization of rule evaluation sequence

Precedence Strategy	Complexity	Predictability	Performance	Chosen
Declaration Order	Low	High	Excellent	Yes
Priority Scoring	Medium	Medium	Good	No
Most Specific	High	Low	Poor	No

## Decision: User Context Attribute Type System

- Context:** User attributes arrive as untyped JSON values but must support type-safe comparison operations across different programming languages and data sources.
- Options Considered:** Dynamic typing with runtime coercion, strict typing with validation, hybrid approach with type-safe wrappers
- Decision:** Hybrid approach using `AttributeValue` wrapper with explicit type conversion methods
- Rationale:** Provides type safety during evaluation while accepting flexible input formats, preventing runtime panics from type mismatches and enabling graceful degradation when attributes don't match expected types
- Consequences:** Adds wrapper overhead and API complexity but eliminates a major class of runtime evaluation failures and provides clear error reporting for type mismatches

Type System	Safety	Flexibility	Performance	Chosen
Dynamic	Low	High	Excellent	No
Strict	High	Low	Good	No
Hybrid Wrapper	High	Medium	Good	Yes

## Decision: Evaluation Result Caching Strategy

- Context:** Flag evaluations may be called hundreds of times per request, but user context and flag definitions change infrequently, creating opportunities for performance optimization through caching.
- Options Considered:** No caching for simplicity, request-scoped caching, time-based TTL caching
- Decision:** Optional request-scoped caching with explicit cache invalidation
- Rationale:** Request-scoped caching eliminates redundant evaluations within a single request while avoiding stale data issues across requests, providing significant performance benefits for evaluation-heavy code paths
- Consequences:** Enables 10-100x performance improvement for repeated evaluations but requires careful integration with request lifecycle management and adds complexity to multi-threaded environments

Caching Strategy	Performance Gain	Consistency Risk	Implementation Complexity	Chosen
No Caching	None	None	Minimal	No
Request-Scoped	High (10-100x)	Low	Medium	Yes
TTL-Based	Very High	High	High	No

## Common Pitfalls in Evaluation Engine Implementation

**⚠️ Pitfall: Inconsistent Hash Input Formatting** The most subtle and dangerous mistake involves inconsistent string formatting when constructing hash inputs. Using `fmt.Sprintf("%s:%s", userID, flagKey)` versus `userID + ":" + flagKey` can produce different results if format specifiers behave unexpectedly with special characters. Always use explicit string concatenation or a standardized formatting function with thorough unit tests covering edge cases like empty strings, unicode characters, and values containing the separator character.

**⚠️ Pitfall: Floating Point Percentage Calculations** Using floating-point arithmetic for percentage calculations introduces rounding errors that can cause users to "fall through the cracks" or be double-assigned. For example, three variants with 33.33% each don't sum to exactly 100%, leaving some bucket ranges unassigned. Always use integer arithmetic with bucket ranges (0-99 for 1% granularity) and handle remainder allocation explicitly rather than relying on floating-point precision.

**⚠️ Pitfall: Rule Condition Short-Circuiting Bugs** When implementing `AND` and `OR` logic for rule conditions, developers often implement short-circuiting incorrectly, causing later conditions to be skipped even when they have side effects like logging or metrics collection. Ensure that all conditions are evaluated for observability purposes, or clearly document when short-circuiting behavior is intentional versus when complete evaluation is required.

**⚠️ Pitfall: Missing User Context Validation** Failing to validate user context structure before evaluation can cause panic conditions when accessing nested attributes or applying type conversions. The evaluation engine must gracefully handle missing attributes, null values, and type mismatches by returning sensible defaults rather than failing the entire evaluation. Implement comprehensive input validation with detailed error logging for debugging purposes.

**⚠️ Pitfall: Circular Dependency Detection** When flags support prerequisites (flag A depends on flag B being enabled), developers often forget to detect circular dependencies during flag creation or updates. A circular dependency can cause infinite recursion during evaluation, crashing the service. Implement dependency graph validation using depth-first search cycle detection before allowing flag modifications to be persisted.

## Implementation Guidance

The flag evaluation engine requires careful attention to performance and correctness, as it sits in the critical path of every feature flag check. The implementation balances sophisticated rule processing capabilities with sub-millisecond evaluation performance.

### Technology Recommendations:

Component	Simple Option	Advanced Option
Hash Function	crypto/sha256 (built-in)	github.com/spaolacci/murmur3 (faster)
JSON Parsing	encoding/json (built-in)	github.com/json-iterator/go (faster)
Validation	Manual checks	github.com/go-playground/validator
Logging	log package (built-in)	github.com/sirupsen/logrus (structured)
Testing	testing package (built-in)	github.com/stretchr/testify (assertions)

### Recommended File Structure:

```
internal/evaluation/
  engine.go           ← Main evaluation logic and EvaluateFlag function
  engine_test.go      ← Comprehensive evaluation test cases
  hashing.go          ← Consistent hashing implementation
  hashing_test.go     ← Hash distribution and stability tests
  rules.go            ← Rule and condition evaluation logic
  rules_test.go        ← Rule precedence and condition tests
  types.go            ← Core data structures and validation
  validation.go       ← Input validation and error handling
  validation_test.go  ← Validation edge case tests
```

### Core Data Structures (Complete Implementation):

```
package evaluation

import (
    "crypto/sha256"
    "encoding/json"
    "fmt"
    "hash/crc32"
    "strconv"
    "strings"
)

// Core evaluation types with complete field definitions

type FlagKey string

type UserID string

// Variant represents a single flag variation with its configuration

type Variant struct {
    Key     string      `json:"key"`      // Unique variant identifier
    Value   interface{} `json:"value"`    // The actual value returned to client
    Weight  int         `json:"weight"`   // Allocation weight for percentage rollouts
}

// UserContext provides all user information needed for targeting decisions

type UserContext struct {
    UserID     UserID      `json:"user_id"`
    Attributes map[string]interface{} `json:"attributes"`
    Segments   []string     `json:"segments"`
}
```

GO

```

// EvaluationResult contains the complete output of flag evaluation

type EvaluationResult struct {

    FlagKey FlagKey      `json:"flag_key"` // Which flag was evaluated

    Value interface{} `json:"value"` // The resolved variant value

    Variant string      `json:"variant"` // The variant key that was selected

    Reason string       `json:"reason"` // Why this variant was chosen

    Source string       `json:"source"` // Where the flag definition came from

}

// Complete flag configuration structure

type FlagDefinition struct {

    Key          FlagKey      `json:"key"`

    Enabled      bool         `json:"enabled"`

    DefaultVariant string      `json:"default_variant"`

    Variants     []Variant    `json:"variants"`

    TargetingRules []TargetingRule `json:"targeting_rules"`

    PercentageRules PercentageRule `json:"percentage_rules"`

    Prerequisites []string     `json:"prerequisites"`

}

// Rule structures for complex targeting

type TargetingRule struct {

    Conditions []Condition `json:"conditions"`

    Operator   string      `json:"operator"` // "AND" or "OR"

    Variant    string      `json:"variant"`

    Enabled    bool         `json:"enabled"`

}

```

```
type Condition struct {

    Attribute string      `json:"attribute"`

    Operator  string      `json:"operator"`

    Value     interface{} `json:"value"`

}

type PercentageRule struct {

    Variants []VariantAllocation `json:"variants"`

}

type VariantAllocation struct {

    Variant string `json:"variant"`

    Weight  int     `json:"weight"`

}

// Type-safe attribute access wrapper

typeAttributeValue struct {

    value interface{}`

}

func NewAttributeValue(value interface{})AttributeValue {

    returnAttributeValue{value: value}

}

func (avAttributeValue) String() (string, bool) {

    if str, ok := av.value.(string); ok {

        return str, true

    }

    return "", false

}
```

```
}

func (avAttributeValue) Int() (int64, bool) {

    switch v := av.value.(type) {

        case int64:

            return v, true

        case float64:

            return int64(v), true

        case string:

            if i, err := strconv.ParseInt(v, 10, 64); err == nil {

                return i, true

            }

    }

    return 0, false
}

func (avAttributeValue) Float() (float64, bool) {

    switch v := av.value.(type) {

        case float64:

            return v, true

        case int64:

            return float64(v), true

        case string:

            if f, err := strconv.ParseFloat(v, 64); err == nil {

                return f, true

            }

    }

    return 0, false
}
```

```
}

func (avAttributeValue) Bool() (bool, bool) {

    if b, ok := av.value.(bool); ok {

        return b, true

    }

    return false, false

}

func (avAttributeValue) StringSlice() ([]string, bool) {

    if slice, ok := av.value.([]interface{}); ok {

        result := make([]string, len(slice))

        for i, item := range slice {

            if str, ok := item.(string); ok {

                result[i] = str

            } else {

                return nil, false

            }

        }

        return result, true

    }

    return nil, false

}

// Validation and error types

type ValidationError struct {

    Field  string `json:"field"`

    Message string `json:"message"`

}
```

```
}

func (ve ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", ve.Field, ve.Message)
}

type ValidationErrors []ValidationError

func (ve ValidationErrors) Error() string {
    var messages []string

    for _, err := range ve {
        messages = append(messages, err.Error())
    }

    return strings.Join(messages, "; ")
}

// Constants for operators and values

const (
    // Logical operators
    AND = "AND"
    OR  = "OR"

    // Condition operators
    equals      = "equals"
    in          = "in"
    contains    = "contains"
    greater_than = "greater_than"
    less_than   = "less_than"
)
```

```
)
```

### Consistent Hashing Implementation (Complete):

```
// calculateUserBucket determines user's allocation bucket using consistent hashing      GO
func calculateUserBucket(userID UserID, flagKey FlagKey) int {
    // TODO 1: Create hash input by combining userID and flagKey with separator
    // TODO 2: Compute hash using CRC32 or SHA256 based on configuration
    // TODO 3: Convert hash to bucket number in range [0, 99]
    // TODO 4: Return bucket ensuring same input always produces same bucket

    // Hint: Use fmt.Sprintf("%s:%s", userID, flagKey) for consistent input format
    // Hint: Use hash % 100 to get bucket, but handle negative values correctly
}

// Helper function for creating deterministic hash inputs
func createHashInput(userID UserID, flagKey FlagKey) string {
    return fmt.Sprintf("%s:%s", string(userID), string(flagKey))
}

// Hash function that provides uniform distribution
func hashToBucket(input string) int {
    hasher := crc32.NewIEEE()
    hasher.Write([]byte(input))
    hashValue := hasher.Sum32()
    return int(hashValue % 100)
}
```

### Core Evaluation Engine Skeleton:

GO

```
// EvaluateFlag performs complete flag evaluation with targeting and fallback

func EvaluateFlag(flagKey FlagKey, context UserContext) EvaluationResult {

    // TODO 1: Validate input parameters (flagKey not empty, context has UserID)

    // TODO 2: Retrieve flag definition, return error result if not found

    // TODO 3: Check if flag is enabled, return default if disabled

    // TODO 4: Process targeting rules in order, return first match

    // TODO 5: If no targeting rules match, evaluate percentage rules

    // TODO 6: If no percentage allocation, return default variant

    // TODO 7: Construct EvaluationResult with value, variant, and reason

    //

    // Hint: Use early returns for error cases to avoid deep nesting

    // Hint: Always populate the Reason field for debugging

}

// evaluateTargetingRules processes user-specific override rules

func evaluateTargetingRules(rules []TargetingRule, context UserContext) (string,
interface{}, string, bool) {

    // TODO 1: Iterate through rules in declaration order

    // TODO 2: Skip disabled rules

    // TODO 3: Evaluate rule conditions using specified operator (AND/OR)

    // TODO 4: Return variant info for first matching rule

    // TODO 5: Return false if no rules match

    //

    // Hint: Call evaluateRuleConditions for each rule

    // Hint: Short-circuit AND on first false, OR on first true

}

// evaluateRuleConditions checks if user context satisfies rule conditions
```

```

func evaluateRuleConditions(conditions []Condition, operator string, context UserContext) bool {
    // TODO 1: Handle empty conditions list (should return true)

    // TODO 2: For AND operator, all conditions must be true

    // TODO 3: For OR operator, any condition can be true

    // TODO 4: Evaluate each condition using evaluateCondition helper

    // TODO 5: Apply short-circuiting logic for performance

    //

    // Hint: Use a loop with early return for short-circuiting

    // Hint: Default to false for unknown operators

}

// evaluateCondition checks single attribute condition

func evaluateCondition(condition Condition, context UserContext) bool {
    // TODO 1: Extract attribute value from context

    // TODO 2: Handle missing attributes gracefully (return false)

    // TODO 3: Apply condition operator (equals, in, contains, etc.)

    // TODO 4: Use AttributeValue wrapper for type-safe comparisons

    // TODO 5: Return comparison result

    //

    // Hint: Use NewAttributeValue(context.Attributes[condition.Attribute])

    // Hint: Each operator has different type requirements

}

```

**Milestone Checkpoint:** After implementing the evaluation engine, you should be able to:

1. **Run unit tests:** `go test ./internal/evaluation/...` should show all tests passing
2. **Test basic evaluation:** Create a simple flag with percentage rollout, verify users get consistent assignments
3. **Test targeting rules:** Create rules with different operators, verify correct matching behavior

4. **Test hash distribution:** Generate 1000 user evaluations, verify roughly even distribution across variants
5. **Test edge cases:** Verify graceful handling of missing flags, invalid context, malformed rules

#### Expected behavior:

- Same user + flag always returns same variant (consistency test)
- Rule precedence works correctly (first matching rule wins)
- Percentage rollouts distribute users evenly across variants
- Missing attributes cause conditions to fail gracefully
- Detailed reason codes explain evaluation decisions

#### Common debugging steps:

- Check hash input format if users get inconsistent assignments
- Verify rule order if wrong targeting rules match
- Examine bucket calculations if percentage distributions look wrong
- Review attribute types if conditions evaluate unexpectedly

## Real-time Flag Updates

**Milestone(s):** This section covers Milestone 2 (Real-time Flag Updates) by implementing streaming protocols for instant flag propagation, connection management with automatic recovery, and intelligent caching strategies to ensure consistent flag state across all client SDKs.

Real-time flag updates transform a feature flag system from a static configuration tool into a dynamic control plane for live applications. Think of this system as a **broadcast network for air traffic control** — when weather conditions change or new safety protocols are established, every pilot needs to receive these updates instantly and reliably. Similarly, when a feature flag is modified to disable a problematic feature or adjust a rollout percentage, all connected applications must receive this change within seconds to maintain consistent user experiences and operational safety.

The challenge lies in maintaining reliable, low-latency communication with potentially thousands of client SDKs while handling network interruptions, connection failures, and varying client capabilities. Unlike traditional request-response APIs, real-time updates require persistent connections, careful state management, and sophisticated fallback strategies to ensure flags remain available even during infrastructure failures.

This system must balance several competing requirements: minimizing latency for critical flag changes, avoiding overwhelming the infrastructure during mass reconnections, ensuring eventual consistency across all clients, and providing graceful degradation when real-time updates are unavailable. The architecture decisions we make here directly impact both the reliability of feature rollouts and the operational overhead of maintaining the flag system.

## Streaming Protocol Design

Modern feature flag systems require sub-second flag propagation to support rapid incident response and real-time experimentation. The choice of streaming protocol fundamentally shapes the system's scalability, reliability, and operational characteristics. This decision affects everything from server resource usage to client SDK complexity and failure recovery strategies.

### Mental Model: Television Broadcasting vs. Telephone Networks

Think of streaming protocols as the difference between television broadcasting and telephone networks. Server-Sent Events (SSE) work like television broadcasts — the server transmits updates to all connected clients simultaneously through a one-way channel, with clients passively receiving updates. WebSockets function like telephone networks, establishing bidirectional communication channels where both parties can initiate conversations. For feature flags, we typically need the broadcasting model since flag changes flow primarily from server to clients, but the choice impacts connection management, resource usage, and failure scenarios.

#### Decision: Server-Sent Events for Flag Updates

- **Context:** Feature flag systems need efficient one-way communication from server to multiple clients with standard HTTP infrastructure compatibility and simple client implementation.
- **Options Considered:** Server-Sent Events (SSE), WebSockets, HTTP Long Polling
- **Decision:** Server-Sent Events (SSE) as the primary streaming protocol with HTTP polling fallback
- **Rationale:** SSE provides automatic reconnection, works through HTTP proxies and load balancers, has lower server resource overhead than WebSockets for one-way communication, and offers built-in event ordering and message replay. WebSockets add unnecessary bidirectional complexity for predominantly unidirectional flag updates.
- **Consequences:** Enables simple client implementation with browser compatibility, reduces server connection overhead, but limits real-time client feedback and requires polling fallback for SSE-incompatible environments.

The following table compares streaming protocol options for feature flag updates:

Protocol Option	Latency	Resource Usage	HTTP Compatibility	Complexity	Reconnection
Server-Sent Events	100-500ms	Low memory/CPU	Full proxy support	Simple client	Automatic with Last-Event-ID
WebSockets	50-100ms	Higher memory	Proxy configuration required	Complex handshake	Manual reconnection logic
HTTP Long Polling	200-1000ms	High connection overhead	Full compatibility	Simple	Built-in with timeouts

Server-Sent Events provide the optimal balance for feature flag systems. The protocol naturally handles connection drops with automatic reconnection, includes built-in event identification for message replay, and works seamlessly with existing HTTP infrastructure. The slight latency increase compared to WebSockets (typically 50-200ms) is acceptable for feature flag use cases where sub-second updates are sufficient.

## SSE Message Format Design

Feature flag updates require structured messages that convey not just the flag changes but also metadata necessary for proper client state management. Each SSE message must include enough information for clients to apply updates correctly, detect message loss, and maintain consistency during reconnection scenarios.

The core message structure contains several essential components:

Message Component	Type	Purpose	Example
Event Type	String	Categorizes update type for client routing	<code>flag_updated</code> , <code>flag_created</code> , <code>flag_deleted</code>
Event ID	String	Unique sequence identifier for replay	<code>seq_12345_1640995200</code>
Timestamp	ISO 8601	Update occurrence time for ordering	<code>2024-01-15T10:30:00Z</code>
Payload	JSON Object	Flag change details and metadata	Flag definition or change summary

## Complete Flag Updates vs. Delta Updates

The system supports two update strategies depending on the nature of changes:

- Complete Flag Definition Updates:** When targeting rules, variants, or core flag properties change, the server sends the entire updated `FlagDefinition` to ensure clients have complete context for

evaluation. This approach eliminates potential inconsistencies from partial updates but increases message size.

2. **Delta Updates:** For simple changes like enabling/disabling flags or adjusting rollout percentages, the server sends only the changed fields. This reduces bandwidth usage but requires more complex client-side merging logic.

The message format accommodates both strategies through a `change_type` field that instructs clients how to process the update:

Change Type	Description	Client Action	Message Size
<code>full_update</code>	Complete flag definition	Replace existing flag entirely	Large (1-5KB)
<code>delta_update</code>	Specific field changes	Merge changes into existing flag	Small (100-500B)
<code>flag_deleted</code>	Flag removal	Remove from local cache	Minimal (50B)

## Event Ordering and Replay

SSE's built-in event identification system enables reliable message ordering and replay during reconnection. Each event receives a unique, monotonically increasing identifier that clients include in reconnection requests via the `Last-Event-ID` header. The server maintains a bounded buffer of recent events (typically 1000-5000 events covering 1-24 hours) to support client reconnection without requiring full flag synchronization.

Event identifiers follow a structured format: `seq_{sequence_number}_{timestamp}`. This format enables both chronological ordering and efficient buffer management on the server side. When clients reconnect after network interruption, they automatically provide their last received event ID, and the server replays all subsequent events to restore consistency.

**Critical Design Insight:** Event replay must be bounded to prevent memory exhaustion from maintaining infinite event history. The replay buffer size represents a trade-off between supporting long-disconnected clients and server resource usage. Clients disconnected longer than the replay buffer require full flag synchronization via the REST API.

## Connection Management and Recovery

Robust connection management distinguishes production-ready feature flag systems from simple implementations. Client SDKs face numerous connectivity challenges: network interruptions, server restarts, load balancer changes, and varying network conditions across different environments. The connection management strategy must handle these scenarios gracefully while avoiding common pitfalls that can overwhelm infrastructure or create inconsistent flag states.

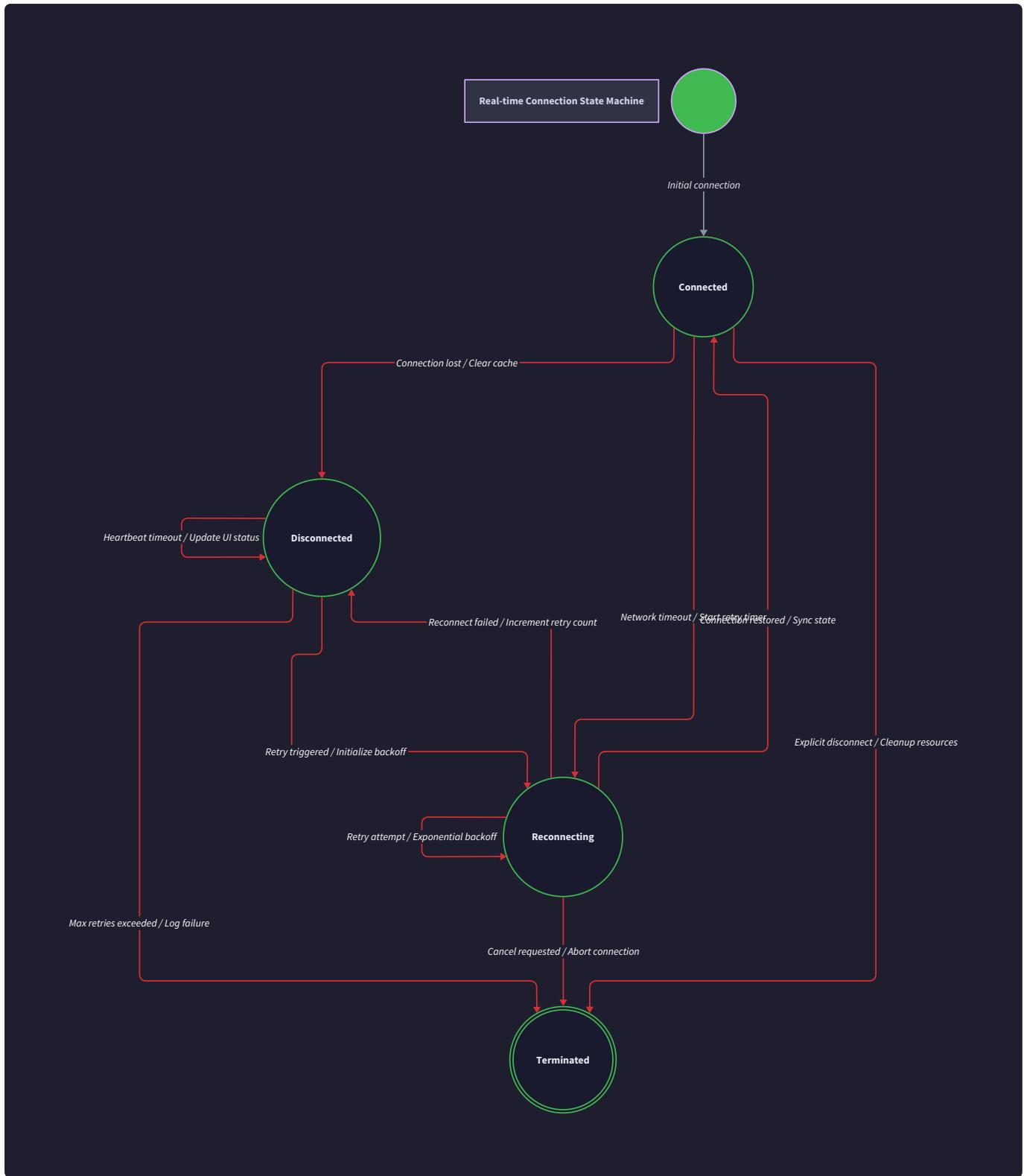
### Mental Model: Cellular Network Roaming

Connection management resembles cellular network roaming systems. When your phone moves between cell towers or loses signal temporarily, it doesn't immediately give up — it systematically searches for connectivity using increasingly patient strategies. Similarly, feature flag clients must gracefully handle connection loss with intelligent retry patterns, maintaining local functionality while persistently attempting to restore real-time updates.

## Connection State Machine

The client SDK connection lifecycle follows a well-defined state machine that governs transitions between connectivity states and defines appropriate actions for each state:

Current State	Event	Next State	Actions Taken
Disconnected	<code>connect()</code> called	Connecting	Initiate SSE connection request
Connecting	Connection established	Connected	Start receiving events, clear retry count
Connecting	Connection failed	Reconnecting	Start exponential backoff timer
Connected	Connection lost	Reconnecting	Begin reconnection attempt with backoff
Connected	Explicit disconnect	Disconnected	Close connection cleanly
Reconnecting	Retry timer expires	Connecting	Attempt reconnection with Last-Event-ID
Reconnecting	Max retries exceeded	Disconnected	Enter degraded mode, log error



Each state defines specific behaviors and responsibilities:

**Connected State:** The client actively receives SSE events, applies flag updates to the local cache, and maintains a heartbeat mechanism to detect connection health. The client tracks the last received event ID for potential replay scenarios and monitors message timestamps to detect clock skew or processing delays.

**Reconnecting State:** The client implements exponential backoff with jitter to avoid thundering herd problems during mass reconnections. Base retry intervals start at 1 second and increase exponentially (2s, 4s, 8s, 16s)

up to a maximum of 300 seconds. Random jitter ( $\pm 25\%$  of the interval) prevents synchronized reconnection attempts from multiple clients.

**Disconnected State:** The client operates in degraded mode, serving flags from local cache while optionally falling back to periodic HTTP polling for critical updates. This state indicates either intentional disconnection or exhausted retry attempts.

## Exponential Backoff with Jitter

Reconnection timing follows a carefully tuned exponential backoff algorithm that balances rapid recovery with infrastructure protection. The algorithm prevents thundering herd scenarios where thousands of clients simultaneously reconnect after a server restart or network partition.

The backoff calculation incorporates several factors:

1. **Base Delay:** Starts at 1 second for the first retry attempt
2. **Exponential Multiplier:** Doubles the delay after each failed attempt (2x multiplier)
3. **Maximum Delay:** Caps at 300 seconds (5 minutes) to prevent indefinite delays
4. **Jitter Factor:** Adds  $\pm 25\%$  randomization to prevent synchronized retries
5. **Success Reset:** Resets to base delay after successful connection

### Backoff Implementation Algorithm:

1. Initialize retry count to zero and base delay to 1 second
2. When connection attempt fails, increment retry count
3. Calculate raw delay as `base_delay * (2 ^ retry_count)`
4. Apply maximum delay cap: `min(raw_delay, max_delay)`
5. Apply jitter: `final_delay = capped_delay * (0.75 + random(0.5))`
6. Wait for final delay before next attempt
7. Reset retry count to zero on successful connection

This approach distributes reconnection attempts over time while ensuring clients don't wait excessively long during brief network interruptions.

## State Synchronization and Recovery

When clients reconnect after extended disconnection, they must efficiently synchronize their local flag state with the current server state. The synchronization strategy depends on the disconnection duration and available event replay history.

### Replay-Based Recovery (preferred for short disconnections):

1. Client includes `Last-Event-ID` header in SSE reconnection request
2. Server validates event ID exists in replay buffer
3. Server replays all events since the specified ID

4. Client applies events sequentially to update local cache
5. Client resumes normal operation after replay completion

#### **Full Synchronization Recovery** (required for long disconnections):

1. Server responds with HTTP 204 (No Replay Available) if event ID is too old
2. Client initiates full flag synchronization via REST API
3. Client retrieves all active flags using `ListFlags(true)`
4. Client replaces entire local cache with fresh flag definitions
5. Client establishes new SSE connection without Last-Event-ID

**Critical Implementation Note:** Clients must handle partial replay scenarios where some events in the replay buffer are corrupted or out of order. The synchronization process should validate event sequence numbers and detect gaps that require full synchronization fallback.

The state synchronization process includes several validation checkpoints:

Checkpoint	Validation	Recovery Action
Event Sequence	Verify sequential event IDs	Request full sync if gaps detected
Flag Versions	Compare flag modification timestamps	Apply updates for newer versions only
Cache Consistency	Validate flag references and dependencies	Rebuild dependency graph
Targeting Rules	Ensure rule conditions are parseable	Mark invalid flags as disabled

## **Cache Invalidation Strategy**

Intelligent caching enables feature flag clients to continue operating during network interruptions while maintaining reasonable freshness guarantees. The caching strategy must balance several competing requirements: minimizing evaluation latency, supporting offline scenarios, ensuring eventual consistency, and avoiding excessive memory usage. Cache invalidation represents the critical mechanism for maintaining correctness when flag definitions change.

#### **Mental Model: Library Book System**

Think of flag caching like a library book checkout system. Each client SDK has checked out copies of flag definitions to use locally (cache). When the library updates a book (flag change), it needs to notify everyone with checked-out copies to return them and get the new version (cache invalidation). However, if someone is reading offline, they can continue using their copy temporarily (graceful degradation) until they reconnect and synchronize.

## Multi-Level Cache Architecture

The caching system employs a hierarchical approach with different cache levels optimized for specific access patterns and consistency requirements:

Cache Level	Location	TTL	Update Mechanism	Purpose
<b>Memory Cache</b>	Client SDK process	No TTL (event-driven)	SSE updates + invalidation	Ultra-fast flag evaluation
<b>Local Storage Cache</b>	Client filesystem/database	24 hours	Periodic refresh + SSE	Offline support and persistence
<b>CDN Cache</b>	Edge locations	60 seconds	Server-side invalidation	Global distribution and load reduction

**Memory Cache:** Provides microsecond flag evaluation latency by keeping all active flag definitions in process memory. Updates occur immediately upon receiving SSE events, ensuring memory cache reflects the most current server state. This cache has no TTL since real-time updates maintain freshness.

**Local Storage Cache:** Supports offline scenarios by persisting flag definitions to local storage (filesystem, embedded database, or browser localStorage). This cache includes timestamps and ETags to support conditional requests during synchronization. Local storage cache serves as the fallback when memory cache is empty or during SDK initialization.

**CDN Cache:** Reduces server load by caching flag responses at edge locations close to client SDKs. CDN cache enables geographic distribution of flag data while supporting rapid invalidation for critical updates.

## Event-Driven Invalidation

Real-time flag updates trigger targeted cache invalidation events that maintain cache consistency without requiring full cache flushes. The invalidation strategy varies based on the type and scope of flag changes:

**Individual Flag Updates:** When a single flag changes, the system sends targeted invalidation events containing the specific `FlagKey`. Clients remove or update only the affected flag, preserving the remainder of their cache. This approach minimizes cache rebuild overhead and maintains evaluation performance for unchanged flags.

**Bulk Flag Updates:** Administrative operations that affect multiple flags (such as environment switches or emergency rollbacks) trigger batch invalidation events. These events contain arrays of affected flag keys or invalidation patterns (such as "all flags for environment X").

**Dependency-Driven Invalidation:** When prerequisite flags change, the system must also invalidate any dependent flags to prevent evaluation inconsistencies. The server maintains a dependency graph and includes downstream flag keys in invalidation events.

The invalidation message structure provides clients with sufficient information to make intelligent caching decisions:

Invalidation Field	Type	Purpose	Example
<code>invalidation_type</code>	String	Scope of invalidation	<code>single_flag</code> , <code>bulk_flags</code> , <code>full_cache</code>
<code>affected_flags</code>	Array	List of flag keys requiring invalidation	<code>["feature_x", "experiment_abc"]</code>
<code>reason</code>	String	Cause of invalidation for debugging	<code>flag_updated</code> , <code>dependency_changed</code> , <code>emergency_rollback</code>
<code>timestamp</code>	ISO 8601	When invalidation was triggered	<code>2024-01-15T10:30:00Z</code>

## Graceful Degradation Patterns

When real-time updates become unavailable, the cache system must provide predictable degradation behavior that maintains application functionality while indicating reduced freshness guarantees. The degradation strategy follows a priority-based approach that preserves the most critical functionality while clearly communicating system state.

### Degradation Priority Levels:

- Memory Cache Only:** Real-time updates unavailable, but memory cache remains valid. Evaluation continues normally with potentially stale data. This level supports minutes to hours of operation depending on flag change frequency.
- Local Storage Fallback:** Memory cache becomes unreliable, system falls back to persisted local storage cache. Freshness guarantees reduce to hours or days, but core functionality remains available.
- Default Values Only:** All cache levels are unavailable or corrupted, system serves only default values specified in flag definitions. This level maintains application stability but disables all dynamic flag behavior.
- Fail-Safe Mode:** Even default values are unavailable, system returns conservative defaults (typically feature flags disabled, experiments off). This level prevents application crashes but may impact user experience.

**Operational Insight:** Each degradation level should emit distinct telemetry and logging to enable operations teams to understand system health and take appropriate remedial action. Clear degradation indicators help distinguish between minor connectivity issues and serious infrastructure failures.

The degradation strategy includes automatic recovery mechanisms:

**Recovery Detection:** Clients periodically attempt to restore higher-priority cache levels by testing connectivity, validating cache consistency, and requesting fresh data. Recovery attempts follow exponential backoff similar to connection retry logic.

**Cache Warming:** When recovering from degraded states, clients prioritize loading the most frequently evaluated flags first, ensuring critical application paths resume normal operation quickly. Less critical flags load in the background.

**Consistency Verification:** During recovery, clients validate cache consistency by comparing local flag versions with server versions, ensuring degraded operation didn't introduce inconsistencies that could affect user experience.

## Common Pitfalls

**⚠ Pitfall: Thundering Herd on Reconnection** Mass client reconnection after server restart can overwhelm infrastructure and cause cascading failures. This occurs when all clients lose connectivity simultaneously and attempt immediate reconnection without coordination. The result is thousands of concurrent connection attempts that exceed server capacity and connection limits.

**Fix:** Implement exponential backoff with jitter for all reconnection attempts. Add connection rate limiting on the server side and consider connection queuing during startup periods. Monitor connection establishment rates and alert on unusual spikes.

**⚠ Pitfall: Infinite Event Replay Buffer** Maintaining unlimited event history for client replay consumes unbounded memory and can cause server crashes during high-throughput periods. Some implementations attempt to store every flag change indefinitely to support arbitrary client reconnection scenarios.

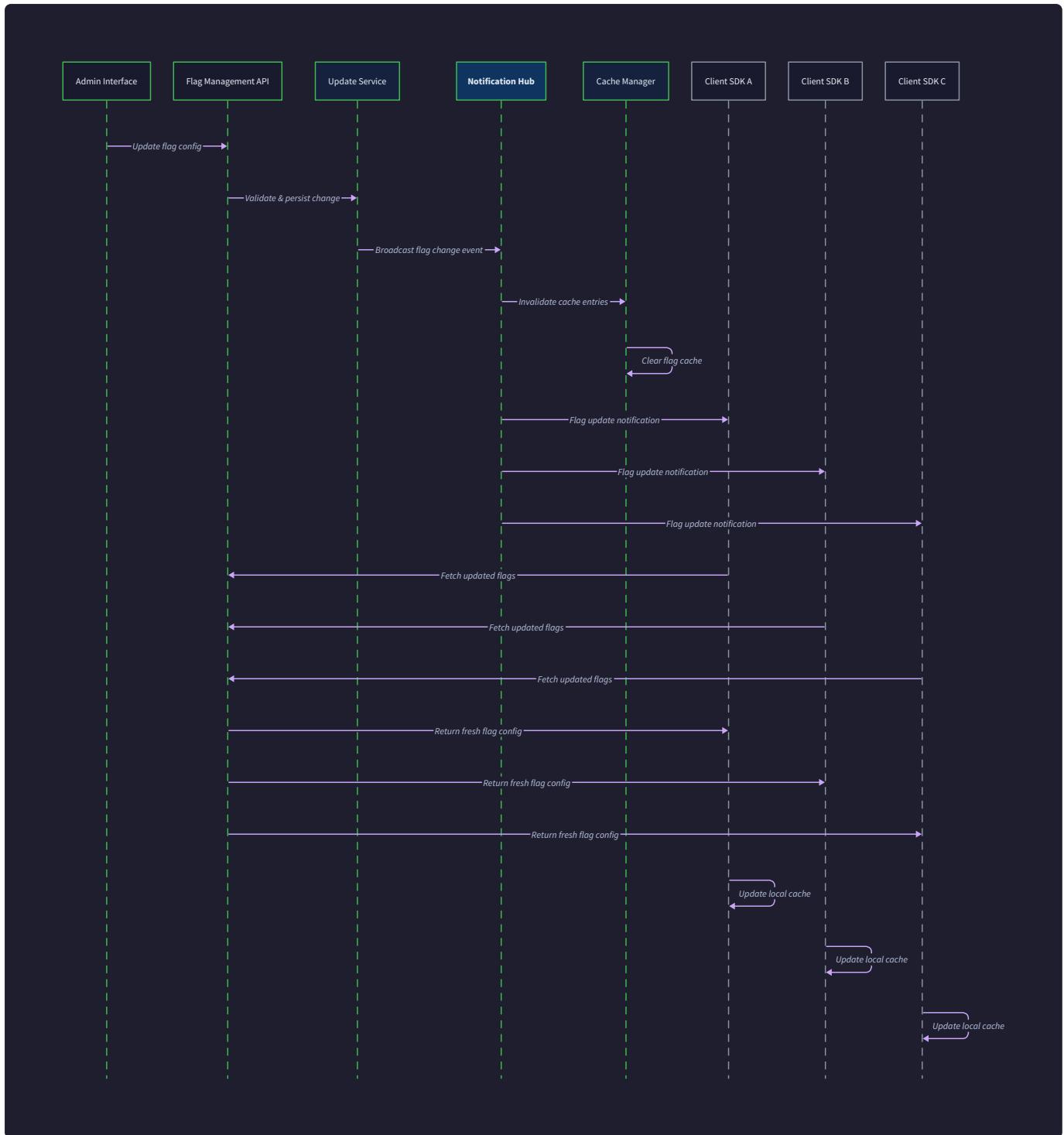
**Fix:** Implement bounded replay buffers with configurable size limits (recommend 1000-5000 events). Clients requesting replay beyond buffer limits must perform full synchronization. Monitor buffer usage and tune size based on typical client disconnection patterns.

**⚠ Pitfall: Cache Corruption During Partial Updates** Delta updates can corrupt client cache if applied out of order or when base flag state is inconsistent. This manifests as clients serving incorrect flag values that don't match server state, leading to inconsistent user experiences.

**Fix:** Include flag version numbers in all updates and validate version compatibility before applying changes. Implement cache consistency checksums and fallback to full synchronization when corruption is detected. Prefer complete flag updates over delta updates for complex changes.

**⚠ Pitfall: Blocking Evaluation During Cache Updates** Applying cache updates synchronously during flag evaluation can cause evaluation latency spikes and thread contention in multi-threaded applications. This particularly impacts high-throughput applications where flag evaluation occurs on critical request paths.

**Fix:** Implement lock-free cache updates using atomic operations or copy-on-write semantics. Stage cache updates in background threads and atomically swap complete cache instances. Never block flag evaluation for cache maintenance operations.



## Implementation Guidance

This section provides concrete implementation guidance for building real-time flag updates in Go, focusing on production-ready patterns that handle the complexities of streaming connections, intelligent caching, and graceful degradation.

## Technology Recommendations

Component	Simple Option	Advanced Option
SSE Server	<code>net/http</code> with <code>http.Flusher</code>	<code>gin-gonic/gin</code> with streaming middleware
Client HTTP	<code>net/http</code> with EventSource pattern	<code>gorilla/websocket</code> for WebSocket fallback
Local Cache	<code>sync.Map</code> with TTL cleanup	<code>patrickmn/go-cache</code> or <code>allegro/bigcache</code>
Persistent Storage	JSON files with <code>os</code> package	<code>bolt/bbolt</code> embedded database
Backoff Logic	Custom exponential calculation	<code>cenkalti/backoff</code> library
Event Serialization	<code>encoding/json</code>	<code>vmitihailenco/msgpack</code> for efficiency

## Recommended File Structure

```
internal/realtim/
  server.go          ← SSE server implementation
  client.go          ← Client SDK with connection management
  cache.go           ← Multi-level cache implementation
  backoff.go         ← Exponential backoff utilities
  events.go          ← Event message definitions
  connection_manager.go ← Connection state machine
  server_test.go     ← SSE server tests
  client_test.go     ← Client connection tests
  cache_test.go      ← Cache invalidation tests
internal/storage/
  flag_store.go      ← Flag storage interface
  memory_store.go    ← In-memory flag storage
pkg/sdk/
  client.go          ← Public SDK interface
  evaluation.go       ← Flag evaluation with caching
examples/
  sse_server/main.go ← Complete SSE server example
  flag_client/main.go← Example client usage
```

## Infrastructure Starter Code

### Complete SSE Server Implementation:

GO

```
// internal/realtime/server.go

package realtime

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

// FlagUpdateEvent represents a flag change notification

type FlagUpdateEvent struct {
    EventType     string      `json:"event_type"`
    EventID       string      `json:"event_id"`
    Timestamp     time.Time   `json:"timestamp"`
    FlagKey       string      `json:"flag_key"`
    Payload       interface{} `json:"payload"`
    ChangeType    string      `json:"change_type"`
}

// SSEServer manages Server-Sent Events for flag updates

type SSEServer struct {
    clients      map[string]*SSEClient
    eventBuffer  []FlagUpdateEvent
}
```

```
    bufferSize int

    mu sync.RWMutex

    eventCounter int64

}

// SSEClient represents a connected client

type SSEClient struct {

    ID string

    ResponseWriter http.ResponseWriter

    Flusher http.Flusher

    LastEventID string

    Connected bool

    mu sync.Mutex

}

// NewSSEServer creates a new SSE server with bounded event buffer

func NewSSEServer(bufferSize int) *SSEServer {

    return &SSEServer{

        clients: make(map[string]*SSEClient),

        eventBuffer: make([]FlagUpdateEvent, 0, bufferSize),

        bufferSize: bufferSize,

    }

}

// ServeHTTP handles SSE connection requests

func (s *SSEServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    // Set SSE headers

    w.Header().Set("Content-Type", "text/event-stream")
```

```
w.Header().Set("Cache-Control", "no-cache")

w.Header().Set("Connection", "keep-alive")

w.Header().Set("Access-Control-Allow-Origin", "*")

flusher, ok := w.(http.Flusher)

if !ok {

    http.Error(w, "Streaming unsupported", http.StatusInternalServerError)

    return

}

clientID := r.Header.Get("X-Client-ID")

if clientID == "" {

    clientID = fmt.Sprintf("client_%d", time.Now().UnixNano())

}

client := &SSEClient{

    ID:           clientID,

    ResponseWriter: w,

    Flusher:       flusher,

    LastEventID:   r.Header.Get("Last-Event-ID"),

    Connected:     true,

}

s.mu.Lock()

s.clients[clientID] = client

s.mu.Unlock()

defer func() {

    s.mu.Lock()
```

```
    delete(s.clients, clientID)

    s.mu.Unlock()

}()

// Send replay events if requested

if client.LastEventID != "" {

    s.sendReplayEvents(client)

}

// Keep connection alive and detect disconnection

ctx := r.Context()

ticker := time.NewTicker(30 * time.Second)

defer ticker.Stop()

for {

    select {

    case <-ctx.Done():

        return

    case <-ticker.C:

        if err := s.sendHeartbeat(client); err != nil {

            log.Printf("Client %s disconnected: %v", clientID, err)

            return

        }

    }

}

}

// sendReplayEvents replays buffered events to reconnecting clients

func (s *SSEServer) sendReplayEvents(client *SSEClient) {
```

```
s.mu.RLock()

defer s.mu.RUnlock()

lastEventNum := s.parseEventID(client.LastEventID)

if lastEventNum == -1 {

    // Invalid event ID, client needs full sync

    s.sendEvent(client, FlagUpdateEvent{

        EventType: "sync_required",

        EventID:    s.generateEventID(),

        Timestamp: time.Now(),

    })

    return
}

// Find starting position in buffer

startIndex := -1

for i, event := range s.eventBuffer {

    if s.parseEventID(event.EventID) > lastEventNum {

        startIndex = i

        break
    }
}

if startIndex == -1 {

    // No events to replay or event too old

    return
}

// Send all events since last received
```

```
for i := startIndex; i < len(s.eventBuffer); i++ {
    s.sendEvent(client, s.eventBuffer[i])
}
}

// sendHeartbeat sends keep-alive ping to client

func (s *SSEServer) sendHeartbeat(client *SSEClient) error {
    client.mu.Lock()
    defer client.mu.Unlock()

    if !client.Connected {
        return fmt.Errorf("client disconnected")
    }

    _, err := fmt.Fprintf(client.ResponseWriter, ": heartbeat\n\n")
    if err != nil {
        client.Connected = false
        return err
    }

    client.Flusher.Flush()
    return nil
}

// BroadcastFlagUpdate sends flag change to all connected clients

func (s *SSEServer) BroadcastFlagUpdate(flagKey string, payload interface{}, changeType string) {
    event := FlagUpdateEvent{
        EventType: "flag_updated",
        EventID:   s.generateEventID(),
    }
}
```

```
    Timestamp: time.Now(),  
  
    FlagKey: flagKey,  
  
    Payload: payload,  
  
    ChangeType: changeType,  
}  
  
// Add to buffer for replay  
  
s.mu.Lock()  
  
s.eventBuffer = append(s.eventBuffer, event)  
  
if len(s.eventBuffer) > s.bufferSize {  
  
    // Remove oldest event to maintain buffer size  
  
    s.eventBuffer = s.eventBuffer[1:]  
}  
  
clients := make([]SSEClient, 0, len(s.clients))  
  
for _, client := range s.clients {  
  
    clients = append(clients, client)  
}  
  
s.mu.Unlock()  
  
// Broadcast to all clients  
  
for _, client := range clients {  
  
    if err := s.sendEvent(client, event); err != nil {  
  
        log.Printf("Failed to send event to client %s: %v", client.ID, err)  
    }  
}  
  
}  
  
// sendEvent writes SSE event to client connection
```

```
func (s *SSEServer) sendEvent(client *SSEClient, event FlagUpdateEvent) error {

    client.mu.Lock()

    defer client.mu.Unlock()

    if !client.Connected {

        return fmt.Errorf("client disconnected")

    }

    data, err := json.Marshal(event)

    if err != nil {

        return err

    }

    _, err = fmt.Fprintf(client.ResponseWriter, "id: %s\nevent: %s\ndata: %s\n\n",
        event.EventID, event.EventType, string(data))

    if err != nil {

        client.Connected = false

        return err

    }

    client.Flusher.Flush()

    return nil

}

// generateEventID creates unique sequential event identifier

func (s *SSEServer) generateEventID() string {

    s.eventCounter++

    return fmt.Sprintf("seq_%d_%d", s.eventCounter, time.Now().Unix())

}
```

```
// parseEventID extracts sequence number from event ID

func (s *SSEServer) parseEventID(eventID string) int64 {

    if eventID == "" {

        return -1

    }

    var seq int64

    var timestamp int64

    if _, err := fmt.Sscanf(eventID, "seq_%d_%d", &seq, &timestamp); err != nil {

        return -1

    }

    return seq

}
```

**Complete Connection Manager with State Machine:**

GO

```
// internal/realtime/connection_manager.go

package realtime

import (
    "bufio"
    "context"
    "fmt"
    "math"
    "math/rand"
    "net/http"
    "sync"
    "time"
)

// ConnectionState represents the current connection status

type ConnectionState int

const (
    StateDisconnected ConnectionState = iota
    StateConnecting
    StateConnected
    StateReconnecting
)

// ConnectionManager handles SSE connection lifecycle

type ConnectionManager struct {

    serverURL      string
    clientID       string
    lastEventID    string
}
```

```
state          ConnectionState

retryCount     int

maxRetries     int

baseDelay      time.Duration

maxDelay       time.Duration

eventHandler   func(FlagUpdateEvent)

errorHandler   func(error)

cancelFunc     context.CancelFunc

mu             sync.RWMutex

stateChan      chan ConnectionState

}

// NewConnectionManager creates a new connection manager

func NewConnectionManager(serverURL, clientID string) *ConnectionManager {

    return &ConnectionManager{

        serverURL:    serverURL,

        clientID:     clientID,

        state:        StateDisconnected,

        maxRetries:   10,

        baseDelay:    time.Second,

        maxDelay:     5 * time.Minute,

        stateChan:    make(chan ConnectionState, 10),

    }

}

// Connect initiates SSE connection with automatic retry
```

```
func (cm *ConnectionManager) Connect(ctx context.Context) error {
    cm.mu.Lock()
    defer cm.mu.Unlock()

    if cm.state != StateDisconnected {
        return fmt.Errorf("connection already active")
    }

    connectCtx, cancel := context.WithCancel(ctx)
    cm.cancelFunc = cancel
    cm.setState(StateConnecting)

    go cm.connectionLoop(connectCtx)

    return nil
}

// connectionLoop manages the connection lifecycle with retry logic

func (cm *ConnectionManager) connectionLoop(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            cm.setState(StateDisconnected)
            return
        default:
            if err := cm.attemptConnection(ctx); err != nil {
                if cm.shouldRetry() {
                    cm.setState(StateReconnecting)
                    delay := cm.calculateBackoffDelay()
                }
            }
        }
    }
}
```

```
        select {

            case <-ctx.Done():

                cm.setState(StateDisconnected)

                return

            case <-time.After(delay):

                cm.retryCount++

                continue

        }

    } else {

        cm.setState(StateDisconnected)

        if cm.errorHandler != nil {

            cm.errorHandler(fmt.Errorf("max retries exceeded"))

        }

        return

    }

} else {

    // Connection successful, reset retry count

    cm.retryCount = 0

}

}

}

}

// attemptConnection tries to establish SSE connection

func (cm *ConnectionManager) attemptConnection(ctx context.Context) error {

    req, err := http.NewRequestWithContext(ctx, "GET", cm.serverURL, nil)

    if err != nil {


```

```
    return err

}

// Set SSE headers

req.Header.Set("Accept", "text/event-stream")

req.Header.Set("Cache-Control", "no-cache")

req.Header.Set("X-Client-ID", cm.clientID)

cm.mu.RLock()

if cm.lastEventID != "" {

    req.Header.Set("Last-Event-ID", cm.lastEventID)

}

cm.mu.RUnlock()

client := &http.Client{

    Timeout: 0, // No timeout for streaming connection

}

resp, err := client.Do(req)

if err != nil {

    return err

}

defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {

    return fmt.Errorf("SSE connection failed: %s", resp.Status)

}

cm.setState(StateConnected)

return cm.processEventStream(ctx, resp)
```

```
}

// processEventStream reads and processes SSE events

func (cm *ConnectionManager) processEventStream(ctx context.Context, resp *http.Response) error {

    scanner := bufio.NewScanner(resp.Body)

    var currentEvent FlagUpdateEvent

    var eventID, eventType, eventData string

    for scanner.Scan() {

        line := scanner.Text()

        select {

        case <-ctx.Done():

            return ctx.Err()

        default:

        }

        if line == "" {

            // Empty line indicates end of event

            if eventType != "" && eventData != "" {

                // Parse and handle event

                if err := cm.handleSSEEvent(eventID, eventType, eventData); err != nil {

                    if cm.errorHandler != nil {

                        cm.errorHandler(err)

                    }

                }

                // Reset for next event

                eventID, eventType, eventData = "", "", ""

            }

        }

    }

}
```

```
        }

        continue

    }

    if len(line) > 5 && line[:3] == "id:" {

        eventID = line[4:]

    } else if len(line) > 7 && line[:6] == "event:" {

        eventType = line[7:]

    } else if len(line) > 6 && line[:5] == "data:" {

        eventData = line[6:]

    }

}

return scanner.Err()

}

// handleSSEEvent processes individual SSE events

func (cm *ConnectionManager) handleSSEEvent(eventID, eventType, eventData string) error {

    // Update last event ID for replay

    cm.mu.Lock()

    cm.lastEventID = eventID

    cm.mu.Unlock()

    // Skip heartbeat events

    if eventData == "" {

        return nil

    }

    // TODO: Parse eventData JSON into FlagUpdateEvent
```

```
// TODO: Call eventHandler with parsed event

// This is where learners implement the event parsing logic


return nil
}

// calculateBackoffDelay computes exponential backoff with jitter

func (cm *ConnectionManager) calculateBackoffDelay() time.Duration {

    // Exponential backoff: baseDelay * (2 ^ retryCount)

    delay := cm.baseDelay * time.Duration(math.Pow(2, float64(cm.retryCount)))

    // Apply maximum delay cap

    if delay > cm.maxDelay {

        delay = cm.maxDelay
    }

    // Add jitter ( $\pm 25\%$ )

    jitter := delay / 4

    jitterAmount := time.Duration(rand.Int63n(int64(jitter*2))) - jitter

    return delay + jitterAmount
}

// shouldRetry determines if connection should be retried

func (cm *ConnectionManager) shouldRetry() bool {

    return cm.retryCount < cm.maxRetries
}
```

```
// setState updates connection state and notifies listeners

func (cm *ConnectionManager) setState(newState ConnectionState) {

    cm.state = newState

    select {

        case cm.stateChan <- newState:

        default:

            // Channel full, skip notification

    }

}

// GetState returns current connection state

func (cm *ConnectionManager) GetState() ConnectionState {

    cm.mu.RLock()

    defer cm.mu.RUnlock()

    return cm.state

}

// Disconnect cleanly closes the connection

func (cm *ConnectionManager) Disconnect() {

    cm.mu.Lock()

    defer cm.mu.Unlock()

    if cm.cancelFunc != nil {

        cm.cancelFunc()

        cm.cancelFunc = nil

    }

    cm.setState(StateDisconnected)

}
```

```
}
```

## Core Logic Skeleton Code

### Flag Cache Implementation with Invalidation:

GO

```
// internal/realtime/cache.go

package realtime

import (
    "encoding/json"
    "sync"
    "time"
)

// FlagCache provides multi-level caching with real-time invalidation

type FlagCache struct {
    memoryCache    map[string]*CachedFlag
    localStorage   LocalStorage
    mu             sync.RWMutex
    stats          CacheStats
}

// CachedFlag wraps flag definition with metadata

type CachedFlag struct {
    Definition    *FlagDefinition
    Version       string
    Timestamp     time.Time
    Source        string // "memory", "local", "default"
}

// CacheStats tracks cache performance metrics

type CacheStats struct {
    Hits          int64
    Misses        int64
}
```

```
    Invalidations int64

    mu sync.RWMutex
}

// LocalStorage interface for persistent flag storage

type LocalStorage interface {

    Store(key string, flag *FlagDefinition) error

    Load(key string) (*FlagDefinition, error)

    Delete(key string) error

    List() ([]string, error)
}

// NewFlagCache creates a new multi-level cache

func NewFlagCache(localStorage LocalStorage) *FlagCache {

    return &FlagCache{

        memoryCache: make(map[string]*CachedFlag),

        localStorage: localStorage,
    }
}

// GetFlag retrieves flag from cache with fallback hierarchy

func (c *FlagCache) GetFlag(flagKey string) (*FlagDefinition, error) {

    // TODO 1: Check memory cache first for fastest access

    // TODO 2: If not in memory, try local storage cache

    // TODO 3: If not in local storage, return error for cache miss

    // TODO 4: Update cache statistics (hits/misses)

    // TODO 5: When loading from local storage, populate memory cache

    // Hint: Use read lock for cache access, upgrade to write lock only when updating
}
```

```
    return nil, nil

}

// InvalidateFlag removes flag from all cache levels

func (c *FlagCache) InvalidateFlag(flagKey string) error {

    // TODO 1: Remove from memory cache using write lock

    // TODO 2: Remove from local storage cache

    // TODO 3: Update invalidation statistics

    // TODO 4: Log invalidation event for debugging

    // Hint: Always update statistics even if flag wasn't cached


    return nil

}

// UpdateFlag stores new flag version in cache

func (c *FlagCache) UpdateFlag(flagKey string, flag *FlagDefinition, source string) error {

    // TODO 1: Create CachedFlag wrapper with metadata (version, timestamp, source)

    // TODO 2: Store in memory cache with write lock

    // TODO 3: Persist to local storage for offline support

    // TODO 4: Validate flag structure before caching

    // TODO 5: Handle storage errors gracefully (memory cache still updates)

    // Hint: Generate version string from flag content hash or timestamp


    return nil

}

// ApplyFlagUpdate processes real-time flag change events
```

```
func (c *FlagCache) ApplyFlagUpdate(event FlagUpdateEvent) error {

    // TODO 1: Parse event payload based on change_type

    // TODO 2: For "full_update": replace entire flag definition

    // TODO 3: For "delta_update": merge changes into existing flag

    // TODO 4: For "flag_deleted": remove flag from all cache levels

    // TODO 5: Validate update consistency (version numbers, timestamps)

    // TODO 6: Handle update conflicts and version mismatches

    // Hint: Delta updates should validate base flag exists and versions match


    return nil
}

// GetCacheStats returns current cache performance metrics

func (c *FlagCache) GetCacheStats() CacheStats {

    c.stats.mu.RLock()

    defer c.stats.mu.RUnlock()

    return CacheStats{
        Hits:           c.stats.Hits,
        Misses:         c.stats.Misses,
        Invalidations: c.stats.Invalidations,
    }
}

// WarmCache preloads frequently used flags

func (c *FlagCache) WarmCache(flagKeys []string) error {

    // TODO 1: Load specified flags from local storage

    // TODO 2: Populate memory cache with loaded flags
```

```
// TODO 3: Handle missing flags gracefully

// TODO 4: Load flags in priority order (most critical first)

// TODO 5: Report warming progress and errors

// Hint: Use goroutines for concurrent warming but limit concurrency

return nil

}

// ValidateConsistency checks cache integrity

func (c *FlagCache) ValidateConsistency() []string {

    // TODO 1: Compare memory cache versions with local storage

    // TODO 2: Detect corrupted or unparseable flag definitions

    // TODO 3: Find flags with invalid dependency references

    // TODO 4: Return list of inconsistent flag keys

    // TODO 5: Log detailed consistency issues for debugging

    // Hint: This method helps detect cache corruption during degraded operation

    return nil
}
```

## Milestone Checkpoints

### Checkpoint 1: SSE Server Basic Functionality

```
# Start the SSE server  
  
go run examples/sse_server/main.go  
  
# Test connection with curl  
  
curl -N -H "Accept: text/event-stream" http://localhost:8080/stream  
  
# Expected: Heartbeat messages every 30 seconds  
  
# Expected: Connection remains open indefinitely  
  
# Expected: Server logs show client connection and heartbeat events
```

BASH

## Checkpoint 2: Event Replay and Reconnection

```
# Connect client, then kill connection and reconnect  
  
# Client should automatically reconnect with exponential backoff  
  
# Server should replay missed events based on Last-Event-ID header  
  
# Test event replay  
  
curl -N -H "Accept: text/event-stream" -H "Last-Event-ID: seq_100_1640995200" \  
http://localhost:8080/stream  
  
# Expected: Server replays events newer than specified ID  
  
# Expected: If ID too old, server sends sync_required event
```

BASH

## Checkpoint 3: Cache Invalidation Integration

```
# Update a flag via management API  
  
curl -X PUT http://localhost:8080/flags/test_flag \  
-d '{"enabled": true, "variants": [{"key": "on", "value": true}]}'  
  
# Verify SSE clients receive flag_updated event within 1 second  
  
# Verify client caches invalidate and serve new flag values  
  
# Test with multiple connected clients for broadcast verification
```

BASH

## Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
<b>Clients never reconnect</b>	Missing exponential backoff or infinite retry loop	Check client logs for retry attempts and delays	Implement bounded retry with proper delay calculation
<b>Thundering herd on server restart</b>	All clients reconnecting simultaneously	Monitor connection rate and server resource usage	Add jitter to backoff delays and connection rate limiting
<b>Events delivered out of order</b>	Missing event sequencing or buffer overflow	Compare event IDs between server and client logs	Implement proper event ordering and sequence validation
<b>Cache serves stale data</b>	Invalidation events not processed or lost	Check SSE connection status and event processing logs	Verify event handling and add cache TTL as backup
<b>Memory leak in event buffer</b>	Unbounded event history storage	Monitor server memory usage over time	Implement fixed-size circular buffer with proper cleanup
<b>Clients stuck in reconnecting state</b>	Network issues or server unavailability	Test network connectivity and server health endpoints	Add connection timeout and fallback to polling mode

## Flag Analytics and A/B Testing

**Milestone(s):** This section covers Milestone 3 (Flag Analytics & Experiments) by implementing flag exposure tracking, A/B testing framework, and statistical significance calculation for data-driven feature decisions.

Think of **flag analytics** as the flight data recorder for your feature flags. Just as aviation authorities need detailed logs to understand what happened during a flight and whether aircraft systems performed correctly, product teams need comprehensive exposure tracking to understand how features are performing in the wild. The analytics system captures every flag evaluation event, much like how a black box records every instrument reading and pilot action throughout a flight.

The **A/B testing framework** extends this analogy to become your experimental flight test program. When aircraft manufacturers test new designs, they don't just throw them into commercial service—they run controlled experiments with careful measurement protocols, statistical analysis, and safety monitoring.

Similarly, your A/B testing framework provides the rigorous experimental methodology needed to validate that new features actually improve user outcomes rather than just looking good in demos.

This analytics foundation transforms feature flags from simple configuration toggles into a sophisticated experimentation platform that enables evidence-based product development. Without proper analytics, feature flags become "fire and forget" deployments where teams never know whether their features succeeded or failed.



## Flag Exposure Tracking

Flag exposure tracking forms the observational backbone of the entire analytics system. Think of **exposure events** as the breadcrumbs that users leave behind as they navigate through your feature variations. Each time the evaluation engine determines which variant a user should see, the system records this decision along with the complete context that influenced it.

The core insight behind exposure tracking is that simply deploying a feature flag isn't enough—you need to know which users actually encountered each variant under what circumstances. This detailed record enables you to answer critical questions: Did the targeting rules work as expected? Are certain user segments seeing variants at different rates? Which environmental factors correlate with feature performance?

### Decision: Event-Based Exposure Recording

- **Context:** Need to capture flag evaluations for analytics without impacting evaluation performance
- **Options Considered:** Synchronous database writes, asynchronous event queues, in-memory buffers with batch writes
- **Decision:** Asynchronous event queues with batch persistence
- **Rationale:** Removes database I/O from the critical evaluation path while ensuring reliable event capture
- **Consequences:** Enables high-throughput evaluation with complete audit trails but requires queue failure handling

The `FlagExposure` data structure captures every essential piece of information about a flag evaluation event:

Field	Type	Description
EventID	string	Unique identifier for this specific exposure event
FlagKey	FlagKey	The flag that was evaluated
UserID	UserID	The user who received the variant
Variant	string	The variant key that was assigned
Value	interface{}	The actual value returned to the application
UserContext	UserContext	Complete user attributes and segments at evaluation time
Timestamp	time.Time	Precise moment when evaluation occurred
Source	string	Evaluation source (cache, database, default)
Reason	string	Why this variant was chosen (targeting rule, percentage, fallback)
ExperimentID	string	Experiment identifier if flag is part of A/B test
SessionID	string	User session to group related exposures
RequestID	string	Request context for debugging and tracing

The exposure tracking pipeline operates through several coordinated stages. When `EvaluateFlag` completes its variant assignment, it immediately calls `RecordExposure` with the complete evaluation result. This function packages the exposure data into a `FlagExposure` event and submits it to an asynchronous processing queue.

The queue consumer operates as a separate goroutine that batches exposure events and writes them to persistent storage in configurable intervals. This design prevents individual flag evaluations from blocking on database writes while ensuring that exposure data is captured reliably even under high load.

Here's how the exposure tracking flow operates:

1. The evaluation engine completes a flag evaluation and produces an `EvaluationResult`
2. `RecordExposure` transforms the evaluation result into a `FlagExposure` event
3. The event is submitted to the exposure queue with a unique event ID and timestamp
4. The queue consumer batches events (typically 100-1000 events per batch)
5. Batched events are written to the analytics database with error retry logic
6. Successfully persisted events are removed from the queue buffer

The critical insight here is that exposure tracking must be completely decoupled from flag evaluation performance. A slow analytics database cannot be allowed to make feature flag evaluations slow, as this would defeat the entire purpose of having a responsive flagging system.

The system maintains exposure event ordering through sequential event IDs that enable gap detection and replay capabilities. If the analytics pipeline falls behind or experiences failures, operators can identify missing event ranges and trigger replay from the persistent queue storage.

### Aggregation and Real-time Processing

Raw exposure events provide the granular audit trail, but most analytics queries require aggregated views. The system maintains several real-time aggregation tables that are updated as exposure events flow through the pipeline:

Aggregation Type	Granularity	Purpose
Flag Summary	Flag + Day	Daily exposure counts per variant
User Journey	User + Session	User's variant assignments across flags
Segment Analysis	Segment + Flag + Hour	How targeting rules distribute users
Experiment Metrics	Experiment + Variant + Hour	A/B test exposure tracking

These aggregations enable fast dashboard queries without requiring full table scans of the raw exposure events. The aggregation logic runs as stream processing jobs that consume the same exposure event queue, ensuring consistency between raw events and summary tables.

### A/B Testing Framework

The A/B testing framework transforms feature flags from simple configuration switches into rigorous experimental instruments. Think of **experiments** as controlled clinical trials for your product features. Just as medical researchers follow strict protocols to ensure their findings are statistically valid and not due to chance, the A/B testing framework provides the methodological rigor needed to distinguish genuine feature improvements from random variation.

An **experiment** represents a formal hypothesis test where you compare user behavior across different feature variants to determine which approach produces better outcomes. The framework handles the complex statistical machinery behind the scenes while presenting a clean interface for defining experiments, tracking metrics, and interpreting results.

## Decision: Experiment-Centric Design

- **Context:** Need to support both simple feature flags and rigorous A/B testing
- **Options Considered:** Flag-first with experiment metadata, experiment-first with flag integration, separate systems
- **Decision:** Experiment-first design with automatic flag generation
- **Rationale:** Ensures experimental rigor by making statistical considerations primary rather than afterthoughts
- **Consequences:** More complex setup for simple flags but proper experimental design for A/B tests

The experiment data model extends the flag system with additional statistical metadata:

Field	Type	Description
ExperimentID	string	Unique identifier for the experiment
Name	string	Human-readable experiment name
Hypothesis	string	Specific hypothesis being tested
FlagKey	FlagKey	Associated feature flag for variant assignment
Variants	[]ExperimentVariant	Available variants with allocation percentages
PrimaryMetric	string	Key performance indicator for success measurement
SecondaryMetrics	[]string	Additional metrics to track for insights
StartDate	time.Time	When experiment began collecting data
EndDate	time.Time	When experiment stopped (nil if running)
MinSampleSize	int	Minimum users per variant for statistical power
SignificanceLevel	float64	Required confidence level (typically 0.05)
PowerLevel	float64	Statistical power target (typically 0.8)
TargetingRules	[]TargetingRule	Conditions for experiment eligibility
Status	string	Current state (draft, running, paused, completed)

Each `ExperimentVariant` defines a specific treatment condition within the experiment:

Field	Type	Description
Key	string	Variant identifier (control, treatment_a, treatment_b)
Name	string	Descriptive variant name
Description	string	What this variant implements
Allocation	float64	Percentage of users assigned to this variant
Configuration	interface{}	Variant-specific feature configuration
IsControl	bool	Whether this represents the baseline condition

## Experiment Lifecycle Management

Experiments progress through a carefully managed lifecycle that ensures statistical validity and prevents common experimental errors:

1. **Draft Phase:** Experiment definition with power analysis and sample size calculation
2. **Review Phase:** Statistical review of experimental design and success metrics
3. **Launch Phase:** Begin user assignment and exposure tracking
4. **Monitoring Phase:** Real-time tracking of sample ratio mismatch and metric collection
5. **Analysis Phase:** Statistical significance testing and confidence interval calculation
6. **Decision Phase:** Determine winning variant and plan rollout or rollback

The framework automatically enforces statistical best practices throughout this lifecycle. For example, it prevents "peeking" at results before reaching minimum sample sizes and warns when sample ratio mismatches indicate assignment problems.

The fundamental principle of rigorous A/B testing is that experimental parameters must be locked before data collection begins. Changing success metrics, variant allocations, or targeting rules after seeing preliminary results invalidates the statistical guarantees.

## Sample Size and Power Analysis

Before launching any experiment, the framework performs power analysis to determine the minimum sample size needed for reliable results. This calculation depends on several statistical parameters:

Parameter	Typical Value	Purpose
Significance Level ( $\alpha$ )	0.05	Maximum false positive rate
Statistical Power ( $1-\beta$ )	0.80	Minimum true positive detection rate
Effect Size	Varies	Minimum improvement worth detecting
Baseline Conversion Rate	Historical	Current metric performance

The power calculation ensures that experiments run long enough to detect meaningful differences while avoiding the costs of over-powered tests. Experiments that launch without proper power analysis often either run forever without reaching conclusions or declare false victories based on insufficient data.

### Assignment Consistency and Tracking

The experiment framework leverages the same consistent hashing mechanism used for percentage rollouts to ensure stable user assignment. However, experiments require additional tracking to maintain assignment consistency across the entire experiment duration.

When a user first encounters an experiment flag, the system records their variant assignment in the experiment assignment table:

Field	Type	Description
ExperimentID	string	Which experiment this assignment belongs to
UserID	UserID	The assigned user
Variant	string	Assigned variant key
AssignmentTime	time.Time	When assignment first occurred
FirstExposure	time.Time	When user first saw the variant
LastExposure	time.Time	Most recent variant exposure
ExposureCount	int	Total number of exposures

This assignment persistence ensures that users see the same variant throughout the experiment duration, even if flag configurations change or targeting rules are modified. The assignment record also enables intent-to-treat analysis, which measures the effect of being assigned to a variant regardless of whether the user actually experienced it.

### Statistical Significance Calculation

Statistical significance calculation transforms raw exposure and conversion data into actionable insights about which variants actually perform better. Think of **significance testing** as the mathematical microscope that

helps you distinguish genuine improvements from random noise. Just as a microscope reveals whether an apparent difference is real cellular structure or just optical artifacts, significance testing reveals whether an apparent performance difference represents true variant superiority or just random variation.

The statistical engine provides the mathematical rigor that prevents teams from making costly product decisions based on misleading data patterns. Without proper significance testing, teams often fall victim to false positives (celebrating improvements that don't exist) or false negatives (missing genuine improvements because they don't look dramatic enough).

### Decision: Frequentist Statistical Framework

- **Context:** Need reliable statistical testing for experiment result interpretation
- **Options Considered:** Frequentist hypothesis testing, Bayesian analysis, non-parametric methods
- **Decision:** Classical frequentist testing with t-tests and chi-square tests
- **Rationale:** Well-understood methodology with clear decision criteria and wide industry acceptance
- **Consequences:** Provides definitive go/no-go decisions but requires careful interpretation of p-values

The `SignificanceResult` data structure encapsulates the complete statistical analysis for an experiment:

Field	Type	Description
<code>ExperimentID</code>	<code>string</code>	Experiment being analyzed
<code>MetricName</code>	<code>string</code>	Performance metric being tested
<code>AnalysisTime</code>	<code>time.Time</code>	When this analysis was computed
<code>VariantResults</code>	<code>[]VariantStatistics</code>	Statistical summary for each variant
<code>Comparisons</code>	<code>[]PairwiseComparison</code>	Significance tests between variant pairs
<code>OverallSignificant</code>	<code>bool</code>	Whether any variant shows significant difference
<code>RecommendedAction</code>	<code>string</code>	Statistical recommendation (continue, stop, extend)
<code>SampleSizeAdequate</code>	<code>bool</code>	Whether minimum sample requirements are met
<code>SampleRatioMismatch</code>	<code>bool</code>	Whether variant assignments deviate from expected ratios
<code>ConfidenceLevel</code>	<code>float64</code>	Statistical confidence level used
<code>AnalysisMethod</code>	<code>string</code>	Statistical test applied (t-test, chi-square, etc.)

Each `VariantStatistics` entry provides detailed statistical measures for a single variant:

Field	Type	Description
Variant	string	Variant identifier
SampleSize	int64	Number of users assigned to this variant
ConversionCount	int64	Number of users who performed the target action
ConversionRate	float64	Conversion rate (conversions / sample size)
StandardError	float64	Standard error of the conversion rate
ConfidenceInterval	ConfidenceInterval	Upper and lower bounds for conversion rate
ZScore	float64	Standardized test statistic
RelativeImprovement	float64	Percentage change from control variant

The `PairwiseComparison` structure captures the statistical test results between any two variants:

Field	Type	Description
VariantA	string	First variant in comparison
VariantB	string	Second variant in comparison
PValue	float64	Probability of observing difference by chance
Significant	bool	Whether difference exceeds significance threshold
TestStatistic	float64	Calculated test statistic value
DegreesOfFreedom	int	Degrees of freedom for the test
EffectSize	float64	Magnitude of the observed difference
PowerAchieved	float64	Statistical power achieved with current sample sizes

## Statistical Test Selection and Execution

The significance calculation engine automatically selects appropriate statistical tests based on the metric type and data distribution characteristics:

Metric Type	Data Distribution	Statistical Test	Purpose
Conversion Rate	Binomial	Two-proportion z-test	Compare success rates between variants
Continuous Metric	Normal	Welch's t-test	Compare means with unequal variances
Count Data	Poisson	Chi-square goodness of fit	Compare event frequencies
Time-to-Event	Exponential	Log-rank test	Compare survival curves

The statistical engine executes these tests through a standardized procedure:

1. **Data Validation:** Verify sample sizes meet minimum requirements and check for data quality issues
2. **Assumption Testing:** Validate statistical assumptions (normality, independence, equal variance)
3. **Test Selection:** Choose appropriate statistical test based on metric type and data characteristics
4. **Calculation:** Compute test statistics, p-values, and confidence intervals
5. **Multiple Comparison Correction:** Apply Bonferroni or FDR correction for multiple variant tests
6. **Interpretation:** Generate actionable recommendations based on statistical results

The critical insight in significance testing is that statistical significance doesn't automatically imply practical significance. A tiny improvement might be statistically significant with enough data but not worth implementing given engineering costs and opportunity costs.

## Sequential Testing and Early Stopping

Traditional fixed-horizon testing requires pre-determining experiment duration and sample sizes, then waiting until completion before analyzing results. However, many business contexts require the ability to stop experiments early when results become conclusive or when variants show concerning negative effects.

The framework implements sequential testing procedures that allow valid statistical inference at multiple time points:

Method	Use Case	Advantages	Limitations
Group Sequential	Pre-planned interim analyses	Maintains Type I error control	Requires predetermined stopping boundaries
Alpha Spending	Flexible interim monitoring	Allows unplanned analyses	More complex boundary calculations
Bayesian Sequential	Continuous monitoring	Natural probability interpretation	Requires prior specification

Sequential testing maintains statistical validity while providing the operational flexibility that product teams require. However, these methods require careful implementation to avoid the statistical pitfalls of repeated testing.

### Sample Ratio Mismatch Detection

One of the most common sources of invalid A/B test results is sample ratio mismatch (SRM), where the observed ratio of users across variants deviates significantly from the intended allocation ratios. SRM typically indicates problems with the randomization mechanism, bot traffic, or technical implementation issues.

The significance calculation engine automatically performs SRM detection using chi-square goodness-of-fit tests:

- 1. Calculate Expected Counts:** Multiply total sample size by intended variant allocations
- 2. Compute Chi-square Statistic:** Sum of  $(\text{observed} - \text{expected})^2 / \text{expected}$  across all variants
- 3. Determine P-value:** Compare test statistic to chi-square distribution
- 4. Flag SRM:** Report mismatch if p-value falls below threshold (typically 0.001)

When SRM is detected, the system automatically flags the experiment results as potentially invalid and recommends investigation before making product decisions based on the data.

### Common Pitfalls

**⚠️ Pitfall: Peeking at Results Early** Teams frequently check experiment results before reaching statistical significance, then stop experiments when they see favorable trends. This "peeking" problem dramatically inflates false positive rates because teams essentially perform multiple statistical tests without correcting for multiple comparisons. **Solution:** Use sequential testing methods with proper stopping boundaries, or commit to fixed experiment durations with single final analysis.

**⚠️ Pitfall: Ignoring Sample Ratio Mismatch** When variant assignment ratios don't match the intended allocation (e.g., 45%/55% instead of 50%/50%), teams often proceed with analysis anyway, assuming the mismatch is minor. However, SRM usually indicates systematic bias in user assignment that invalidates statistical assumptions. **Solution:** Always check for SRM using chi-square tests and investigate the root cause before trusting experiment results.

**⚠ Pitfall: Confusing Statistical and Practical Significance** Large sample sizes can make tiny differences statistically significant even when they're not practically meaningful. A 0.01% conversion rate improvement might have  $p < 0.05$  but generate less revenue than the engineering cost to implement. **Solution:** Always define minimum detectable effect sizes during experiment design and consider business impact alongside statistical significance.

**⚠ Pitfall: Multiple Comparison Problems** When testing multiple metrics or multiple variants simultaneously, the probability of false positives increases dramatically. Testing 20 metrics at  $\alpha = 0.05$  gives about a 64% chance of at least one false positive. **Solution:** Apply multiple comparison corrections (Bonferroni, FDR) or designate a single primary metric for decision-making with secondary metrics for insights only.

**⚠ Pitfall: Survivorship Bias in Analysis** Analyzing only users who remained active throughout the entire experiment duration can bias results toward variants that retain users better. This survivorship bias can make harmful variants appear beneficial if they selectively retain certain user types. **Solution:** Use intent-to-treat analysis that includes all assigned users in the final calculation, regardless of their subsequent engagement.

**⚠ Pitfall: Inconsistent Assignment Tracking** If user variant assignments change during an experiment (due to hash function changes, targeting rule modifications, or cache inconsistencies), the statistical analysis becomes meaningless. Users who experience multiple variants contaminate both variant groups. **Solution:** Persist user assignments at first exposure and enforce assignment consistency throughout the experiment duration.

## Implementation Guidance

The analytics and experimentation system requires careful integration of high-throughput event processing, statistical computation, and data persistence. This implementation guidance provides the foundational infrastructure needed to support production-scale A/B testing.

## Technology Recommendations

Component	Simple Option	Advanced Option
Event Queue	Go channels with file persistence	Apache Kafka or AWS Kinesis
Analytics Database	PostgreSQL with time-series tables	ClickHouse or Apache Druid
Statistical Computing	Go math/stats with custom functions	R integration via Rserve
Real-time Aggregation	In-memory maps with periodic persistence	Apache Flink or Spark Streaming
Dashboards	Static HTML with JSON API	Grafana or custom React frontend

## Recommended File Structure

```
internal/analytics/
  exposure/
    tracker.go           ← FlagExposure recording and queuing
    tracker_test.go      ← Exposure tracking tests
    aggregator.go        ← Real-time aggregation pipeline
  experiments/
    experiment.go        ← Experiment lifecycle management
    experiment_test.go   ← Experiment framework tests
    assignment.go         ← User assignment persistence
  statistics/
    significance.go      ← Statistical significance calculation
    significance_test.go ← Statistics computation tests
    power.go              ← Sample size and power analysis
  storage/
    schema.sql            ← Database schema for analytics tables
  migrations/             ← Database migration scripts
```

## Exposure Tracking Infrastructure

GO

```
// ExposureTracker manages the asynchronous recording of flag evaluation events

type ExposureTracker struct {

    queue        chan FlagExposure

    batchSize    int

    flushPeriod  time.Duration

    storage      ExposureStorage

    aggregator   *RealTimeAggregator

    shutdown     chan struct{}

    wg           sync.WaitGroup

}

// FlagExposure represents a single flag evaluation event for analytics

type FlagExposure struct {

    EventID      string      `json:"event_id"`

    FlagKey      FlagKey     `json:"flag_key"`

    UserID       UserID      `json:"user_id"`

    Variant      string      `json:"variant"`

    Value        interface{} `json:"value"`

    UserContext  UserContext `json:"user_context"`

    Timestamp    time.Time   `json:"timestamp"`

    Source       string      `json:"source"`

    Reason       string      `json:"reason"`

    ExperimentID string      `json:"experiment_id,omitempty"`

    SessionID    string      `json:"session_id,omitempty"`

    RequestID    string      `json:"request_id,omitempty"`

}

// RecordExposure submits a flag evaluation event to the analytics pipeline
```

```
func (et *ExposureTracker) RecordExposure(exposure FlagExposure) error {

    // TODO 1: Generate unique event ID and timestamp if not provided

    // TODO 2: Validate exposure data (required fields, valid flag key, etc.)

    // TODO 3: Submit exposure to queue channel (non-blocking with timeout)

    // TODO 4: Return error if queue is full or tracker is shutting down

    // Hint: Use select with default case to avoid blocking on full queue

}

// StartProcessing begins the background exposure processing pipeline

func (et *ExposureTracker) StartProcessing() {

    // TODO 1: Start goroutine for batch processing from queue

    // TODO 2: Collect exposures into batches based on count and time

    // TODO 3: Write batches to persistent storage with retry logic

    // TODO 4: Update real-time aggregations for dashboard queries

    // TODO 5: Handle graceful shutdown when shutdown channel is closed

}
```

## Experiment Framework Core

GO

```
// ExperimentManager handles A/B test lifecycle and statistical analysis

type ExperimentManager struct {

    storage      ExperimentStorage

    assignments  AssignmentStorage

    calculator   *SignificanceCalculator

    mu          sync.RWMutex

}

// Experiment represents a controlled A/B test with statistical parameters

type Experiment struct {

    ExperimentID      string           `json:"experiment_id"`

    Name              string           `json:"name"`

    Hypothesis        string           `json:"hypothesis"`

    FlagKey           FlagKey         `json:"flag_key"`

    Variants          []ExperimentVariant `json:"variants"`

    PrimaryMetric     string           `json:"primary_metric"`

    SecondaryMetrics  []string         `json:"secondary_metrics"`

    StartDate         time.Time        `json:"start_date"`

    EndDate           *time.Time       `json:"end_date,omitempty"`

    MinSampleSize     int              `json:"min_sample_size"`

    SignificanceLevel float64          `json:"significance_level"`

    PowerLevel        float64          `json:"power_level"`

    TargetingRules    []TargetingRule `json:"targeting_rules"`

    Status            string           `json:"status"`

}

// ExperimentVariant defines a treatment condition within an A/B test

type ExperimentVariant struct {
```

```

Key          string      `json:"key"`

Name         string      `json:"name"`

Description  string      `json:"description"`

Allocation   float64     `json:"allocation"`

Configuration interface{} `json:"configuration"`

IsControl    bool        `json:"is_control"`

}

// CreateExperiment initializes a new A/B test with statistical validation

func (em *ExperimentManager) CreateExperiment(experiment Experiment) error {

    // TODO 1: Validate experiment configuration (allocation sum = 1.0, etc.)

    // TODO 2: Perform power analysis to determine required sample sizes

    // TODO 3: Check that associated flag exists and is compatible

    // TODO 4: Create experiment record with "draft" status

    // TODO 5: Generate experiment assignment hash seed for consistency

}

// AssignUserToVariant determines which variant a user should see

func (em *ExperimentManager) AssignUserToVariant(experimentID string, userID UserID,
context UserContext) (string, error) {

    // TODO 1: Check if user has existing assignment for this experiment

    // TODO 2: If no existing assignment, check experiment eligibility rules

    // TODO 3: Use consistent hashing to assign user to variant

    // TODO 4: Persist assignment record for consistency

    // TODO 5: Return assigned variant key

}

```

## Statistical Significance Calculator

GO

```
// SignificanceCalculator performs statistical analysis of experiment results

type SignificanceCalculator struct {
    storage ExperimentStorage
}

// SignificanceResult contains complete statistical analysis of an experiment

type SignificanceResult struct {
    ExperimentID      string            `json:"experiment_id"`
    MetricName        string            `json:"metric_name"`
    AnalysisTime      time.Time        `json:"analysis_time"`
    VariantResults    []VariantStatistics `json:"variant_results"`
    Comparisons       []PairwiseComparison `json:"comparisons"`
    OverallSignificant bool             `json:"overall_significant"`
    RecommendedAction string           `json:"recommended_action"`
    SampleSizeAdequate bool            `json:"sample_size_adequate"`
    SampleRatioMismatch bool           `json:"sample_ratio_mismatch"`
    ConfidenceLevel   float64          `json:"confidence_level"`
    AnalysisMethod    string           `json:"analysis_method"`
}

// VariantStatistics provides detailed statistical measures for a single variant

type VariantStatistics struct {
    Variant      string            `json:"variant"`
    SampleSize   int64             `json:"sample_size"`
    ConversionCount int64          `json:"conversion_count"`
    ConversionRate float64         `json:"conversion_rate"`
    StandardError float64          `json:"standard_error"`
    ConfidenceInterval ConfidenceInterval `json:"confidence_interval"`
}
```

```

    ZScore           float64      `json:"z_score"`
    RelativeImprovement float64      `json:"relative_improvement"`

}

// CalculateSignificance performs complete statistical analysis of experiment results

func (sc *SignificanceCalculator) CalculateSignificance(experimentID string)
(SignificanceResult, error) {

    // TODO 1: Retrieve experiment configuration and current sample sizes

    // TODO 2: Check for sample ratio mismatch using chi-square test

    // TODO 3: Calculate conversion rates and confidence intervals per variant

    // TODO 4: Perform pairwise significance tests between variants

    // TODO 5: Apply multiple comparison corrections if needed

    // TODO 6: Generate actionable recommendations based on results

    // Hint: Use z-test for conversion rate comparisons

}

// DetectSampleRatioMismatch checks if variant assignments match intended ratios

func (sc *SignificanceCalculator) DetectSampleRatioMismatch(experimentID string) (bool,
float64, error) {

    // TODO 1: Get intended allocation ratios from experiment configuration

    // TODO 2: Count actual user assignments per variant

    // TODO 3: Calculate expected counts based on total sample and ratios

    // TODO 4: Compute chi-square test statistic

    // TODO 5: Return mismatch flag and p-value

    // Hint: Chi-square = Σ((observed - expected)^2 / expected)

}

```

## Database Schema

SQL

```
-- Flag exposure events for detailed analytics

CREATE TABLE flag_exposures (
    event_id VARCHAR(36) PRIMARY KEY,
    flag_key VARCHAR(255) NOT NULL,
    user_id VARCHAR(255) NOT NULL,
    variant VARCHAR(255) NOT NULL,
    value JSONB,
    user_context JSONB NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE NOT NULL,
    source VARCHAR(50) NOT NULL,
    reason VARCHAR(255) NOT NULL,
    experiment_id VARCHAR(36),
    session_id VARCHAR(255),
    request_id VARCHAR(255),
    INDEX idx_flag_exposures_flag_time (flag_key, timestamp),
    INDEX idx_flag_exposures_user_time (user_id, timestamp),
    INDEX idx_flag_exposures_experiment (experiment_id, timestamp)
);

-- Experiment definitions and configuration

CREATE TABLE experiments (
    experiment_id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    hypothesis TEXT,
    flag_key VARCHAR(255) NOT NULL,
    variants JSONB NOT NULL,
    primary_metric VARCHAR(255) NOT NULL,
```

```

secondary_metrics JSONB,
start_date TIMESTAMP WITH TIME ZONE NOT NULL,
end_date TIMESTAMP WITH TIME ZONE,
min_sample_size INTEGER NOT NULL,
significance_level DECIMAL(4,3) NOT NULL DEFAULT 0.05,
power_level DECIMAL(4,3) NOT NULL DEFAULT 0.8,
targeting_rules JSONB,
status VARCHAR(50) NOT NULL DEFAULT 'draft',
created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Persistent user variant assignments for experiments

CREATE TABLE experiment_assignments (
experiment_id VARCHAR(36) NOT NULL,
user_id VARCHAR(255) NOT NULL,
variant VARCHAR(255) NOT NULL,
assignment_time TIMESTAMP WITH TIME ZONE NOT NULL,
first_exposure TIMESTAMP WITH TIME ZONE,
last_exposure TIMESTAMP WITH TIME ZONE,
exposure_count INTEGER DEFAULT 0,
PRIMARY KEY (experiment_id, user_id),
INDEX idx_assignments_experiment_variant (experiment_id, variant),
INDEX idx_assignments_user (user_id, assignment_time)
);

```

## Milestone Checkpoint

After implementing the analytics and A/B testing framework:

## 1. Test Exposure Tracking:

- Run flag evaluations and verify exposure events appear in the database
- Check that exposure events contain complete context information
- Confirm that high evaluation throughput doesn't block on analytics

## 2. Validate Experiment Assignment:

- Create an experiment with 50/50 allocation between two variants
- Assign 1000 test users and verify the allocation ratio is approximately correct
- Confirm that users receive the same variant on repeat assignments

## 3. Verify Statistical Calculations:

- Generate synthetic experiment data with known effect sizes
- Run significance calculations and verify they detect the expected differences
- Test sample ratio mismatch detection with intentionally skewed allocations

Expected behavior: The system should handle thousands of exposure events per second while maintaining assignment consistency and producing accurate statistical analysis. Dashboard queries should complete in under 500ms even with millions of exposure events.

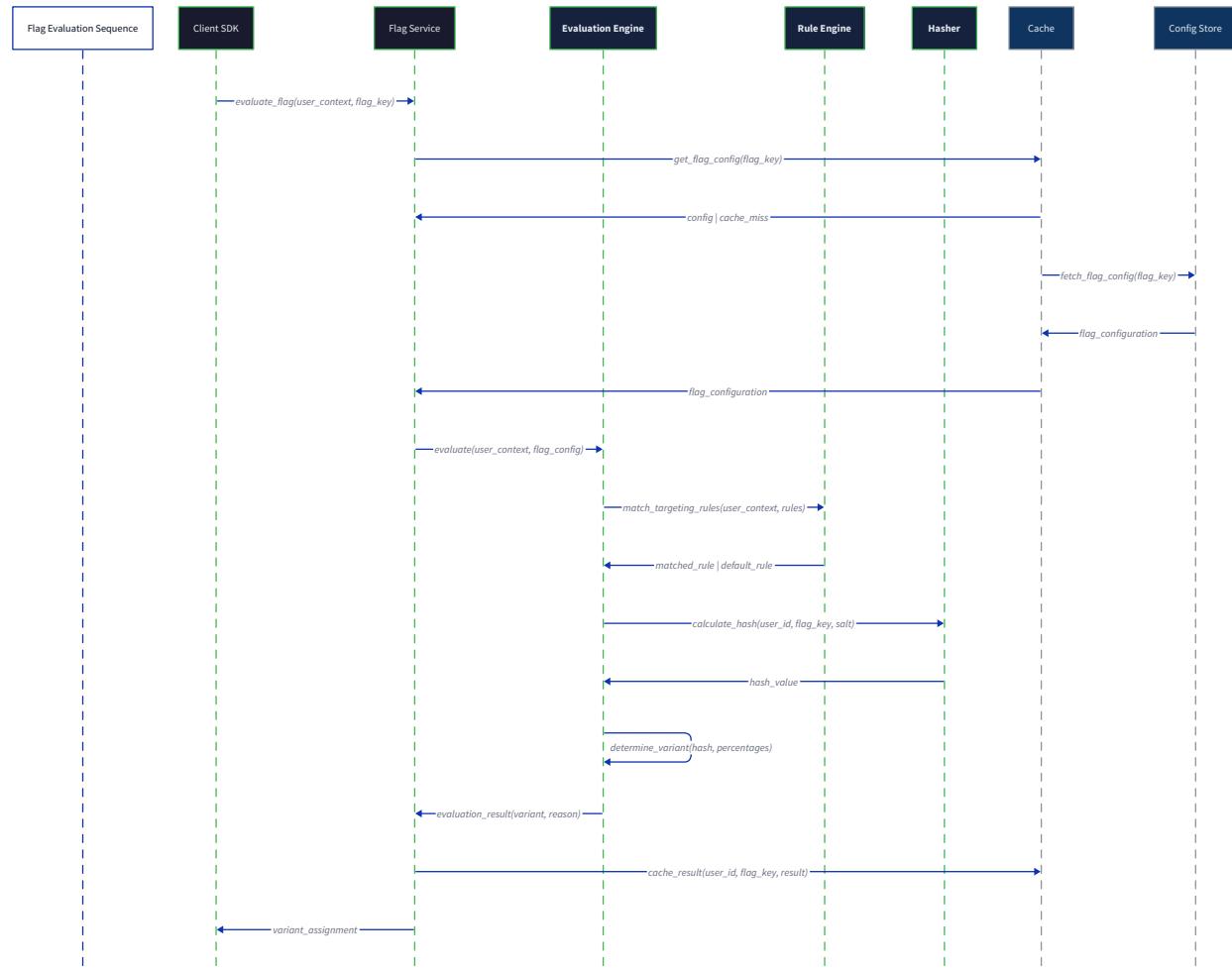
## Interactions and Data Flow

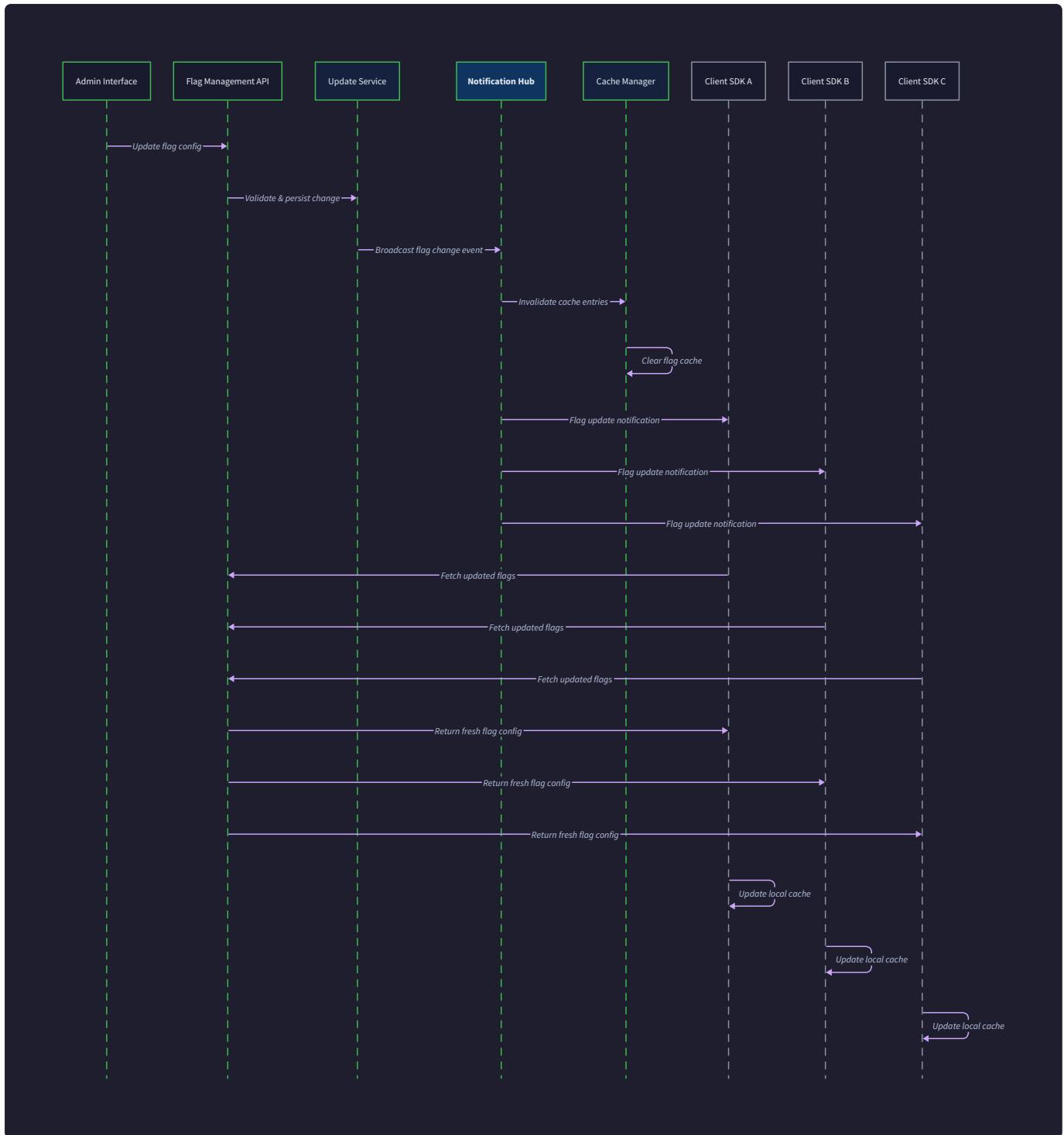
**Milestone(s):** This section spans all three milestones by describing the communication patterns and data flows that connect the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3) into a cohesive system.

Think of the feature flag system as **a sophisticated air traffic control tower** managing multiple types of aircraft movements simultaneously. The evaluation flow is like guiding incoming flights through a series of checkpoints—checking flight credentials, runway availability, and weather conditions—before assigning them to specific gates. The update propagation flow resembles broadcasting critical weather updates or runway changes to all aircraft in the system, ensuring they adjust their flight plans in real-time. Just as air traffic control maintains constant communication between the control tower, radar systems, and aircraft, our feature flag system orchestrates seamless interactions between the management interface, evaluation engine, real-time update service, and distributed client SDKs.

The interactions and data flow represent the **nervous system** of the feature flag platform, connecting decision-making components with data sources and ensuring consistent behavior across all system boundaries. Understanding these flows is crucial because feature flags must provide both **immediate consistency** for flag evaluations and **eventual consistency** for configuration updates, while maintaining high availability and low latency under varying load conditions.

This section examines two critical interaction patterns that define the system's operational behavior: the **flag evaluation flow** that processes individual user requests and returns targeted variants, and the **update propagation flow** that distributes configuration changes across all connected clients. These flows represent the primary data pathways through which the system delivers its core value proposition of controlled feature rollouts and real-time configuration management.





## Flag Evaluation Flow

The flag evaluation flow represents the **critical path** through which user requests are processed and appropriate feature variants are assigned. This flow must operate with **sub-millisecond latency** while maintaining consistency guarantees and supporting complex targeting logic. Think of this process as a **sophisticated sorting machine** that examines each user request through multiple lenses—user attributes, segment membership, percentage allocations, and experimental assignments—before directing them to the appropriate feature experience.

The evaluation flow begins when a client SDK receives a request to evaluate a specific flag for a given user context. This triggers a carefully orchestrated sequence of operations that balance **performance requirements** with **correctness guarantees**. The flow must handle scenarios ranging from simple boolean toggles to complex multi-variate experiments with intricate targeting rules, all while ensuring that users receive consistent assignments across multiple evaluations.

Understanding the evaluation flow is essential for implementing proper **caching strategies**, **fallback mechanisms**, and **performance optimizations**. The flow also establishes the foundation for flag analytics by generating exposure events that track which users see which variants under what conditions. Every step in this flow represents a potential **decision point** where the system must choose between different variants, making the logic both critically important and inherently complex.

## Step-by-Step Evaluation Process

The flag evaluation process follows a deterministic sequence that ensures consistent results while accommodating various edge cases and error conditions. Each step in this process serves a specific purpose in the overall targeting and assignment logic:

- 1. Request Validation and Context Preparation:** The client SDK receives an evaluation request containing a `FlagKey` and `UserContext`. The SDK validates that the user context contains required fields (`UserID` is non-empty, `Attributes` map is properly formatted) and enriches the context with any cached segment memberships or computed attributes. Invalid contexts trigger immediate fallback value returns with appropriate error reasons.
- 2. Cache Lookup and Freshness Verification:** The evaluation engine attempts to retrieve the flag definition from its local cache using `GetFlag(flagKey)`. The cache lookup includes timestamp verification to ensure the cached definition hasn't exceeded its time-to-live threshold. Cache hits proceed directly to rule evaluation, while cache misses trigger background refresh operations but continue with stale data if available to maintain low latency.
- 3. Flag Existence and Status Verification:** If the flag definition is found, the engine verifies that the flag is in an active state and hasn't been archived or deleted. Inactive flags return their configured default values with an appropriate reason code. This step also checks for any flag prerequisites or dependencies that must be satisfied before evaluation can proceed.
- 4. User Context Enrichment and Segment Resolution:** The engine enhances the provided user context with any missing segment memberships by evaluating segment rules against user attributes. This step is **computationally expensive** but can be optimized through pre-computed segment caches or background segment resolution processes. The enriched context becomes the input for subsequent targeting rule evaluation.
- 5. Targeting Rule Evaluation with Precedence Ordering:** The engine processes targeting rules in priority order using `evaluateTargetingRules(rules, context)`. Each rule consists of conditions that are evaluated using `evaluateRuleConditions(conditions, operator, context)` with proper AND/OR

logic. The first rule that matches determines the variant assignment, making rule ordering critically important for correct behavior.

6. **Percentage Rollout Processing with Consistent Hashing:** If no targeting rules match, the engine falls back to percentage rollout logic. It calculates the user's bucket assignment using `calculateUserBucket(userID, flagKey)` and determines which variant allocation range contains that bucket. This ensures that users receive consistent assignments across multiple evaluations while respecting the configured percentage distributions.
7. **Variant Value Resolution and Type Coercion:** Once the target variant is determined, the engine resolves the actual variant value from the flag definition. This may involve type coercion (converting stored JSON values to appropriate types), template variable substitution, or other value transformation logic depending on the flag type and configuration.
8. **Exposure Event Generation and Analytics Tracking:** The successful evaluation triggers the creation of a `FlagExposure` event that captures all relevant context about the evaluation decision. This event is queued for asynchronous processing using `RecordExposure(exposure)` to avoid impacting evaluation latency while ensuring comprehensive analytics tracking.
9. **Result Packaging and Return:** The engine constructs an `EvaluationResult` containing the resolved value, selected variant key, evaluation reason (which rule matched or percentage allocation), and source attribution. This result is returned to the client SDK and may be cached locally for subsequent evaluations of the same flag-user combination.

**Design Insight:** The evaluation flow prioritizes **deterministic behavior** over performance optimization. Each step produces predictable outputs for given inputs, ensuring that debugging and troubleshooting can trace exactly why a user received a specific variant. This determinism is crucial for maintaining trust in feature rollouts and experiment validity.

## Caching and Performance Optimization

The evaluation flow incorporates **multi-level caching** to achieve target latency requirements while maintaining data consistency. The caching strategy must balance fresh data with performance, particularly for high-traffic applications where flag evaluations occur thousands of times per second.

**Flag Definition Caching** represents the first level of optimization, where complete flag configurations are cached in memory after retrieval from the primary data store. These caches use **time-based expiration** combined with **event-driven invalidation** to ensure reasonable freshness while avoiding excessive database queries. The cache maintains version timestamps that enable **optimistic locking** when updates occur.

**User Context Caching** provides the second optimization layer by storing computed segment memberships and derived attributes for frequently seen users. Since segment evaluation can be computationally expensive (requiring multiple attribute checks and rule processing), caching these results significantly improves evaluation performance. The cache uses **LRU eviction** to manage memory consumption while prioritizing recently active users.

**Evaluation Result Caching** represents an optional third layer where complete evaluation results are cached for specific flag-user combinations. This optimization is most effective for **low-variance flags** (where most users receive the same result) but must be carefully managed to avoid serving stale results when flag configurations change. Result caches require **immediate invalidation** when flag definitions are updated.

The caching architecture includes **cache warming** mechanisms that pre-populate frequently accessed flags and user contexts during application startup or low-traffic periods. This warming process reduces **cold start penalties** and ensures consistent performance during traffic spikes.

## Error Handling and Fallback Logic

The evaluation flow must handle various error conditions gracefully while maintaining system availability and providing meaningful feedback for debugging purposes. The error handling strategy emphasizes **fail-safe behavior** where evaluation errors result in sensible default values rather than complete request failures.

**Network and Service Failures** are handled through **graceful degradation** where the evaluation engine continues operating with cached data even when the primary flag management service becomes unavailable. The engine maintains **staleness indicators** that help clients understand the freshness of cached data and make appropriate decisions about feature behavior.

**Malformed Flag Definitions** trigger **validation errors** that prevent the flag from being cached or evaluated, but don't impact other flags in the system. The validation process includes **schema checking**, **circular dependency detection**, and **rule consistency verification** to catch configuration errors before they affect user experiences.

**User Context Errors** (missing required attributes, invalid data types, or malformed segment references) are handled through **context sanitization** and **default value substitution**. The evaluation engine attempts to proceed with available context information while logging detailed error information for debugging purposes.

**Targeting Rule Failures** that occur during evaluation (such as division by zero in percentage calculations or invalid regular expressions in condition matching) cause the specific rule to be skipped and evaluation to continue with subsequent rules. This **rule-level isolation** prevents individual misconfigured rules from breaking entire flag evaluations.

## Common Pitfalls in Evaluation Flow

**⚠ Pitfall: Inconsistent User Bucket Calculations** Many implementations suffer from **bucket assignment drift** where users flip between different variants due to inconsistent hashing algorithms or input normalization. This occurs when the bucketing logic uses floating-point arithmetic with rounding errors, includes unstable user attributes in the hash input, or fails to normalize string attributes consistently. To avoid this issue, use **integer-based consistent hashing** with stable, normalized inputs that include only the user ID and flag key. Implement comprehensive tests that verify bucket assignments remain stable across multiple evaluation calls.

**⚠ Pitfall: Cache Invalidation Race Conditions** Race conditions in cache invalidation can cause **temporary inconsistencies** where some evaluation requests use old flag definitions while others use updated definitions. This typically happens when cache invalidation events arrive out of order or when multiple cache

layers aren't updated atomically. Implement **versioned caching** with **optimistic concurrency control** where each cache entry includes a version timestamp, and invalidation events specify the minimum version that should be evicted.

**⚠ Pitfall: Context Pollution from Previous Evaluations** Reusing `UserContext` objects across multiple evaluations can lead to **context pollution** where computed values (like resolved segment memberships) from previous evaluations influence subsequent ones. This creates subtle bugs where flag evaluations appear to depend on the order of previous evaluations. Always **deep copy** or **reset** user context objects between evaluations, and clearly separate provided attributes from computed values.

## Update Propagation Flow

The update propagation flow represents the **distribution backbone** that ensures flag configuration changes reach all connected clients promptly and consistently. Think of this flow as a **sophisticated broadcast network** similar to how emergency alert systems simultaneously notify all receivers of critical updates—the system must deliver messages quickly, handle network partitions gracefully, and ensure no client is left with outdated information that could impact user experiences.

The update propagation flow begins when flag configurations are modified through the management interface and concludes when all connected client SDKs have applied the new configuration and invalidated relevant caches. This flow must handle scenarios ranging from simple flag toggles that need immediate deployment to complex targeting rule changes that require careful coordination across distributed client populations.

The propagation flow serves as the **consistency mechanism** that prevents the system from exhibiting different behaviors across different clients or geographic regions. Without proper update propagation, users could experience **inconsistent feature experiences** where the same action produces different results depending on which server handles their request or which SDK version they're using. This consistency is particularly critical for experiment integrity and user experience quality.

Understanding the update propagation flow is essential for implementing proper **change management**, **rollback capabilities**, and **deployment strategies**. The flow establishes the foundation for **operational safety** by ensuring that configuration changes can be applied, monitored, and reverted in a controlled manner across the entire distributed system.

## Propagation Sequence and Timing

The update propagation process follows a carefully orchestrated sequence designed to minimize **inconsistency windows** while maintaining system performance and reliability. Each stage in the propagation process serves specific purposes in the overall change management strategy:

- 1. Change Validation and Staging:** When a flag modification is submitted through the management API using `UpdateFlag(flag)`, the system performs comprehensive validation including schema verification, rule consistency checking, and dependency analysis. The validated change is staged in a **pending state** before being committed to the primary data store, allowing for additional approval workflows or automated safety checks.

2. **Primary Storage Update with Versioning:** The validated flag configuration is committed to the primary data store with an incremented version number and change metadata. This update is performed atomically to ensure that partial updates don't create inconsistent states. The storage layer generates a **change event** that includes the flag key, change type (create, update, delete), and version information.
3. **Change Event Broadcasting via Real-time Service:** The real-time update service detects the storage change and generates a `FlagUpdateEvent` that is broadcast to all connected clients using `BroadcastFlagUpdate(flagKey, payload, changeType)`. The broadcast uses **Server-Sent Events** or **WebSocket connections** to deliver the change notification with minimal latency, typically within 100-500 milliseconds of the original modification.
4. **Client-side Change Reception and Validation:** Connected client SDKs receive the update notification through their persistent connections and validate the change event format and version information. Clients verify that the received change is newer than their current cached version to prevent **regression issues** where older changes overwrite newer ones due to network delays or message reordering.
5. **Local Cache Invalidation and Refresh:** Each client SDK applies the received update using `ApplyFlagUpdate(event)`, which invalidates the locally cached flag definition and either applies the new configuration immediately or marks it for **lazy loading** on the next evaluation request. The invalidation process must handle **concurrent evaluations** that might be using the old cached data.
6. **Background Refresh and Verification:** Clients that don't have persistent connections (such as mobile applications or batch processing systems) discover changes through **background polling** or **SDK initialization** processes. These clients compare their cached flag versions with the current server versions and fetch updated configurations as needed, ensuring eventual consistency even without real-time connections.
7. **Propagation Confirmation and Monitoring:** The update service tracks which clients have acknowledged receipt of change notifications and maintains **propagation metrics** that indicate how quickly changes are spreading through the client population. This monitoring enables **rollback decisions** if propagation failures or adverse effects are detected during the rollout process.
8. **Stale Cache Detection and Cleanup:** The system includes **background processes** that identify clients serving stale flag configurations (perhaps due to persistent connection failures or implementation bugs) and takes corrective action such as forcing cache invalidation or alerting operations teams about potentially inconsistent deployments.

**Design Insight:** The propagation flow implements **optimistic consistency** where changes are applied immediately to the primary store and distributed asynchronously to clients. This approach minimizes **write latency** for flag modifications while accepting brief **read inconsistency** windows during propagation. The trade-off favors operational agility over perfect consistency, which aligns with feature flag use cases where brief inconsistencies are generally acceptable.

## Connection Management and Message Delivery

The update propagation system must maintain **persistent connections** with potentially thousands of client SDKs while handling network instabilities, client failures, and varying connection qualities. The connection management strategy balances **resource efficiency** with **delivery reliability** to ensure that critical flag changes reach all intended recipients.

**Connection Lifecycle Management** begins when client SDKs establish connections to the real-time update service using `Connect(ctx)` with **exponential backoff** retry logic. Each connection is assigned a unique client identifier and tracked in the service's connection registry. The service monitors connection health through **heartbeat mechanisms** and **activity tracking** to detect failed or stale connections that should be cleaned up.

**Message Buffering and Replay** ensures that clients reconnecting after network interruptions receive any flag changes that occurred during their absence. The update service maintains a **sliding window buffer** of recent change events using `sendReplayEvents(client)` to deliver missed updates when clients reconnect. The buffer size and retention period are configurable based on system requirements and client reconnection patterns.

**Delivery Confirmation and Retry Logic** provides **at-least-once delivery** guarantees for critical flag changes. Clients send **acknowledgment messages** when they successfully apply flag updates, allowing the service to track delivery success rates and identify problematic connections or clients. Failed deliveries trigger **retry attempts** with appropriate backoff delays to avoid overwhelming struggling clients.

**Load Balancing and Scaling** accommodates varying client populations by distributing connections across multiple update service instances. The system uses **consistent hashing** or **connection affinity** strategies to ensure that clients maintain connections with specific service instances while supporting **graceful failover** when instances become unavailable.

**Message Ordering and Deduplication** prevents **out-of-order updates** that could cause clients to apply older flag configurations after newer ones. Each change event includes **sequence numbers** and **version timestamps** that clients use to detect and discard outdated messages. The system also implements **deduplication logic** to prevent clients from processing the same change multiple times due to retry mechanisms.

## Network Partition and Recovery Handling

The update propagation system must continue operating effectively during **network partitions**, **service outages**, and **client isolation events** that prevent normal real-time communication. The recovery handling strategy ensures that system consistency is eventually restored without requiring manual intervention or complex reconciliation procedures.

**Partition Detection and Isolation** occurs when the update service loses connectivity with client populations due to network infrastructure failures or geographic connectivity issues. The service detects these partitions through **connection monitoring**, **heartbeat timeouts**, and **message delivery failure patterns**. Isolated client populations continue operating with their cached flag configurations while the partition persists.

**Autonomous Client Operation** enables client SDKs to continue providing flag evaluation services using their **local flag caches** even when disconnected from the update service. Clients maintain **cache timestamps** and **staleness indicators** that help application code make informed decisions about whether cached flag data is still reliable for business-critical features.

**Reconciliation After Recovery** involves **state synchronization** processes that execute when network connectivity is restored after partition events. Clients compare their cached flag versions with current server versions using **bulk version check** APIs and download updated configurations for any flags that changed during the partition. This reconciliation process uses **batching** and **rate limiting** to avoid overwhelming the service during mass reconnection events.

**Conflict Resolution and Consistency Repair** addresses scenarios where **configuration conflicts** or **inconsistent states** are detected during recovery processes. The system implements **server-authoritative** conflict resolution where server-side flag configurations always take precedence over client-side cached data, ensuring that inconsistencies are resolved in favor of the most recent management decisions.

**Progressive Rollout During Recovery** provides **controlled recovery** mechanisms that gradually restore normal operation rather than immediately resuming full-scale update propagation. The service can implement **traffic shaping** and **connection throttling** to prevent **thundering herd** effects when large client populations attempt to reconnect simultaneously after widespread network issues.

## Change Impact Analysis and Safety Mechanisms

The update propagation flow includes **safety mechanisms** and **impact analysis** capabilities that help prevent problematic flag changes from causing widespread system issues or user experience degradation. These mechanisms provide **operational safety nets** that enable confident flag management even in complex, high-stakes environments.

**Pre-propagation Impact Assessment** analyzes proposed flag changes to estimate their **blast radius** and potential system impact before changes are propagated to client populations. This analysis considers factors such as the number of users affected, the scope of feature changes, and dependencies on other system components to help operations teams make informed rollout decisions.

**Gradual Propagation and Circuit Breakers** enable **controlled rollouts** where flag changes are initially propagated to small subsets of the client population before broader deployment. The system monitors **error rates**, **performance metrics**, and **user experience indicators** during initial propagation phases and can **halt** or **rollback** the propagation process if adverse effects are detected.

**Automatic Rollback Triggers** provide **fail-safe mechanisms** that automatically revert flag changes when predefined **safety thresholds** are exceeded. These triggers might include error rate spikes, performance degradation beyond acceptable limits, or explicit user experience metrics that indicate negative impact from the flag change.

**Change Approval and Gating** implements **workflow controls** that require appropriate approvals before high-impact flag changes are propagated to production client populations. The approval process can include

**automated testing, peer review, stakeholder sign-off, and deployment scheduling** to ensure that changes are applied safely and with appropriate coordination.

**Propagation Monitoring and Alerting** provides **real-time visibility** into the change distribution process, enabling operations teams to detect and respond to propagation issues quickly. The monitoring includes **delivery success rates, client acknowledgment patterns, error distributions, and consistency metrics** that indicate the health and effectiveness of the update propagation system.

## Common Pitfalls in Update Propagation

**⚠ Pitfall: Thundering Herd During Mass Reconnection** When many clients lose connectivity and then reconnect simultaneously (such as during network outages or service restarts), they can overwhelm the update service with **concurrent connection attempts** and **bulk update requests**. This creates a **thundering herd effect** that can cause the service to become unavailable, preventing successful reconnection and creating a cascade failure. Implement **exponential backoff with jitter** using `calculateBackoffDelay()` where each client waits a randomized delay before attempting reconnection, and use **connection rate limiting** on the server side to control the maximum concurrent connection establishment rate.

**⚠ Pitfall: Message Ordering Violations During High Load** Under high load or during network congestion, flag update messages can arrive **out of order** at client SDKs, causing them to apply older flag configurations after newer ones. This creates **temporal inconsistency** where clients regress to previous flag states unpredictably. Implement **sequence number validation** where each update message includes a monotonically increasing sequence number using `generateEventID()`, and clients discard any messages with sequence numbers lower than the most recently processed message.

**⚠ Pitfall: Infinite Propagation Loops from Circular Dependencies** When flag changes trigger other flag changes (through computed dependencies or automated reactions), the system can enter **infinite propagation loops** that overwhelm the update infrastructure and prevent normal operation. This often occurs when **A/B testing flags** automatically trigger **feature flags** that in turn influence **configuration flags** in circular patterns. Implement **propagation depth limits** and **change cycle detection** that tracks the **causal chain** of flag changes and prevents updates that would create circular dependencies or exceed reasonable propagation depths.

## Implementation Guidance

The interactions and data flow implementation requires careful coordination between multiple system components while maintaining performance and reliability guarantees. The following guidance provides practical approaches for implementing robust evaluation and propagation flows.

## Technology Recommendations

Component	Simple Option	Advanced Option
Evaluation Engine	In-memory map with RWMutex (sync.RWMutex)	Distributed cache with Redis Cluster
Real-time Updates	Server-Sent Events with net/http	WebSocket with gorilla/websocket + message queuing
Message Serialization	JSON with encoding/json	Protocol Buffers with gogo/protobuf
Connection Management	Basic HTTP keepalive	Connection pooling with circuit breakers
Event Streaming	Simple broadcast to all clients	Pub/sub with message persistence (NATS, Apache Kafka)
Cache Layer	Local in-memory cache	Multi-tier with Redis + local LRU
Metrics Collection	Basic counters with expvar	Prometheus metrics with custom collectors
Background Processing	Simple goroutines	Worker pool with <a href="https://github.com/gammazero/workerpool">github.com/gammazero/workerpool</a>

## Recommended File Structure

```
internal/
  evaluation/
    engine.go          ← Core evaluation logic
    engine_test.go     ← Evaluation flow tests
    cache.go           ← Multi-level caching implementation
    context.go          ← User context handling
    bucket.go          ← Consistent hashing logic
  realtime/
    server.go          ← SSE/WebSocket server
    client.go          ← Client connection management
    broadcast.go        ← Update propagation logic
    reconnect.go        ← Recovery and retry logic
  analytics/
    tracker.go          ← Exposure event tracking
    aggregator.go        ← Real-time metrics aggregation
  flows/
    evaluation_flow.go   ← Complete evaluation orchestration
    propagation_flow.go  ← Update distribution coordination
    flows_test.go         ← Integration tests for complete flows
```

## Evaluation Flow Infrastructure

```
package flows

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

    "github.com/your-org/flagsystem/internal/evaluation"
    "github.com/your-org/flagsystem/internal/analytics"
)

// EvaluationOrchestrator coordinates the complete flag evaluation flow

type EvaluationOrchestrator struct {

    engine      *evaluation.Engine
    cache       *evaluation.FlagCache
    tracker     *analytics.ExposureTracker
    validator   *evaluation.ContextValidator
    mu          sync.RWMutex
    metrics     *EvaluationMetrics
}

// EvaluationMetrics tracks performance and behavior of evaluation flows

type EvaluationMetrics struct {

    TotalEvaluations    int64
    CacheHits           int64
    CacheMisses         int64
}
```

GO

```
EvaluationLatency time.Duration
ErrorCount int64
mu sync.RWMutex
}

// NewEvaluationOrchestrator creates a complete evaluation flow handler

func NewEvaluationOrchestrator(engine *evaluation.Engine, cache *evaluation.FlagCache,
tracker *analytics.ExposureTracker) *EvaluationOrchestrator {
    return &EvaluationOrchestrator{
        engine: engine,
        cache: cache,
        tracker: tracker,
        validator: evaluation.NewContextValidator(),
        metrics: &EvaluationMetrics{},
    }
}

// ProcessEvaluationRequest handles the complete evaluation flow from request to response

func (e *EvaluationOrchestrator) ProcessEvaluationRequest(ctx context.Context, flagKey FlagKey, userCtx UserContext) (*EvaluationResult, error) {
    startTime := time.Now()
    defer func() {
        e.metrics.mu.Lock()
        e.metrics.TotalEvaluations++
        e.metrics.EvaluationLatency = time.Since(startTime)
        e.metrics.mu.Unlock()
    }()
    // TODO 1: Validate user context using e.validator.ValidateUserContext(userCtx)
}
```

```
// TODO 2: Attempt cache lookup using e.cache.GetFlag(flagKey)

// TODO 3: If cache miss, fetch from primary store and update cache

// TODO 4: Enrich user context with segment memberships

// TODO 5: Execute core evaluation using e.engine.EvaluateFlag(flagKey, enrichedCtx)

// TODO 6: Generate exposure event and queue using e.tracker.RecordExposure()

// TODO 7: Update evaluation metrics and return result

// Placeholder return - implement the actual flow

return &EvaluationResult{

    FlagKey: flagKey,
    Value:   false,
    Variant: "default",
    Reason:  "not_implemented",
    Source:  "cache",
}, nil

}
```

## Real-time Update Infrastructure

```
package realtime

import (
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "time"
)

// UpdatePropagationService manages the complete update distribution flow

type UpdatePropagationService struct {

    sseServer      *SSEServer
    clients        map[string]*ClientConnection
    eventBuffer    []FlagUpdateEvent
    bufferSize     int
    mu             sync.RWMutex
    metrics        *PropagationMetrics
}

// PropagationMetrics tracks update distribution performance

type PropagationMetrics struct {

    ActiveConnections   int64
    MessagesSent        int64
    DeliveryFailures   int64
    ReconnectionCount   int64
    PropagationLatency time.Duration
}
```

GO

```
}

// ClientConnection represents a connected SDK client

type ClientConnection struct {

    ID          string
    LastSeen    time.Time
    Connection   *SSEClient
    PendingAcks map[string]time.Time
    mu          sync.Mutex
}

// NewUpdatePropagationService creates the complete propagation flow handler

func NewUpdatePropagationService(bufferSize int) *UpdatePropagationService {
    return &UpdatePropagationService{
        sseServer:  NewSSEServer(),
        clients:    make(map[string]*ClientConnection),
        eventBuffer: make([]FlagUpdateEvent, 0, bufferSize),
        bufferSize:  bufferSize,
        metrics:    &PropagationMetrics{},
    }
}

// PropagateChange handles the complete flow from flag change to client delivery

func (u *UpdatePropagationService) PropagateChange(flagKey string, changeType string,
    payload interface{}) error {
    startTime := time.Now()

    // TODO 1: Create FlagUpdateEvent with unique ID and timestamp

    // TODO 2: Add event to buffer for replay to reconnecting clients
```

```
// TODO 3: Broadcast to all active connections using u.sseServer.BroadcastFlagUpdate()

// TODO 4: Track delivery confirmations and retry failed deliveries

// TODO 5: Update propagation metrics with timing and success rates

// TODO 6: Handle buffer overflow by removing oldest events

event := FlagUpdateEvent{

    EventType:    "flag_update",
    EventID:     u.generateEventID(),
    Timestamp:   time.Now(),
    FlagKey:     flagKey,
    Payload:     payload,
    ChangeType:  changeType,
}

// Placeholder implementation - complete the propagation logic

u.mu.Lock()
defer u.mu.Unlock()

u.eventBuffer = append(u.eventBuffer, event)

u.metrics.PropagationLatency = time.Since(startTime)

return nil
}

// HandleClientConnection manages the complete connection lifecycle

func (u *UpdatePropagationService) HandleClientConnection(w http.ResponseWriter, r *http.Request) {

    clientID := r.Header.Get("Client-ID")
```

```
if clientID == "" {  
  
    http.Error(w, "Client-ID header required", http.StatusBadRequest)  
  
    return  
  
}  
  
  
// TODO 1: Establish SSE connection with proper headers  
  
// TODO 2: Register client in connection registry  
  
// TODO 3: Send buffered events for any missed updates  
  
// TODO 4: Start heartbeat monitoring for connection health  
  
// TODO 5: Handle connection cleanup on client disconnect  
  
  
// Placeholder - implement complete connection handling  
  
fmt.Fprintf(w, "data: {"type": "connected", "client_id": "%s"}\n\n", clientID)  
}  
  
  
func (u *UpdatePropagationService) generateEventID() string {  
  
    // TODO: Implement monotonic event ID generation  
  
    return fmt.Sprintf("%d", time.Now().UnixNano())  
}
```

## Core Evaluation Logic Skeleton

```
package evaluation GO

// Engine implements the core flag evaluation logic with caching and consistency

type Engine struct {

    storage    FlagStorage
    cache      *FlagCache
    hasher     *ConsistentHasher
    validator  *ContextValidator
}

// EvaluateFlag processes the complete evaluation flow for a single flag

func (e *Engine) EvaluateFlag(flagKey FlagKey, context UserContext) EvaluationResult {
    // TODO 1: Retrieve flag definition from cache or storage

    // TODO 2: Validate flag is active and not archived

    // TODO 3: Check for prerequisite flag dependencies

    // TODO 4: Enrich context with computed segment memberships

    // TODO 5: Evaluate targeting rules in priority order

    // TODO 6: Fall back to percentage rollout if no rules match

    // TODO 7: Resolve variant value and perform type coercion

    // TODO 8: Return comprehensive evaluation result with reason

    return EvaluationResult{
        FlagKey: flagKey,
        Value:   nil,
        Variant: "default",
        Reason:  "not_implemented",
        Source:  "engine",
    }
}
```

```

    }

}

// evaluateTargetingRules processes rules with proper precedence and logic

func (e *Engine) evaluateTargetingRules(rules []TargetingRule, context UserContext)
(string, interface{}, string, bool) {

    // TODO 1: Sort rules by priority (highest first)

    // TODO 2: Iterate through rules and evaluate conditions

    // TODO 3: For matching rule, return its variant and value

    // TODO 4: Handle AND/OR logic properly within rule conditions

    // TODO 5: Return detailed reason for debugging

    return "", nil, "no_matching_rules", false
}

// calculateUserBucket provides consistent user assignment to percentage buckets

func (e *Engine) calculateUserBucket(userID UserID, flagKey FlagKey) int {

    // TODO 1: Create hash input from userID and flagKey

    // TODO 2: Apply consistent hashing algorithm

    // TODO 3: Convert hash to bucket number (0-99)

    // TODO 4: Ensure deterministic results for same inputs

    return 0 // Placeholder
}

```

## Milestone Checkpoint: Evaluation Flow

After implementing the evaluation flow, verify the following behavior:

**Test Command:** `go test ./internal/flows/ -v -run TestEvaluationFlow`

**Expected Behavior:**

- Flag evaluations complete within 5ms for cached flags
- User bucket assignments remain stable across multiple evaluations
- Targeting rules are evaluated in correct priority order
- Cache hits/misses are properly tracked and reported
- Exposure events are generated for all successful evaluations

#### **Manual Verification:**

```
# Start the flag service
go run cmd/server/main.go

# Test flag evaluation endpoint
curl -X POST http://localhost:8080/api/v1/evaluate \
-H "Content-Type: application/json" \
-d '{
  "flag_key": "test-flag",
  "user_context": {
    "user_id": "user123",
    "attributes": {"beta_user": true}
  }
}'

# Expected response includes variant, value, and evaluation reason
# Check logs for exposure event generation

```

BASH

#### **Milestone Checkpoint: Update Propagation**

After implementing the update propagation flow, verify the following behavior:

**Test Command:** `go test ./internal/realtimel/ -v -run TestPropagation`

#### **Expected Behavior:**

- Flag changes propagate to connected clients within 500ms
- Clients reconnect automatically after network interruptions
- Buffered events are replayed to reconnecting clients

- Connection counts and delivery metrics are accurate
- No message duplication or ordering violations occur

### Manual Verification:

```
# Start update service and connect test client
go run cmd/server/main.go &

go run examples/test-client/main.go

# Update a flag through management API
curl -X PUT http://localhost:8080/api/v1/flags/test-flag \
-H "Content-Type: application/json" \
-d '{"enabled": true, "variants": [...]}'

# Verify client receives update within 500ms
# Test reconnection by restarting client
```

BASH

### Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Users flip between variants	Inconsistent bucket calculation	Check hash inputs and normalization	Use stable, normalized user ID and flag key only
Flag updates take minutes to propagate	Connection failures or buffering issues	Check SSE connection health and event buffer	Implement connection monitoring and retry logic
Evaluation errors during flag updates	Cache invalidation race conditions	Check concurrent access to cache during updates	Use versioned caching with proper locking
High evaluation latency	Cache misses or expensive segment resolution	Monitor cache hit rates and segment computation time	Implement cache warming and segment pre-computation
Missing exposure events	Async queue overflows or processing failures	Check exposure queue depths and error logs	Implement backpressure and queue monitoring

# Error Handling and Edge Cases

**Milestone(s):** This section applies to all three milestones by ensuring robust error handling and graceful degradation across the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3).

Think of error handling in feature flag systems like building a suspension bridge with multiple safety systems. Just as bridges have primary cables, backup cables, and emergency procedures for various failure scenarios, feature flag systems require layered safety mechanisms. When the primary evaluation service fails, the system should seamlessly fall back to cached data. When the real-time update connection drops, clients should continue operating with their last known state while attempting reconnection. When analytics services are unavailable, flag evaluations should continue uninterrupted. The key insight is that feature flags control critical business functionality, so the system must be designed to fail gracefully rather than catastrophically.

Building a resilient feature flag system requires anticipating every possible failure mode and designing appropriate responses. Unlike traditional applications where failures might cause user-facing errors, feature flag failures can silently corrupt A/B test results, cause inconsistent user experiences, or even break critical business features. The challenge is maintaining system reliability while preserving the performance characteristics that make feature flags practical for high-traffic applications.

## Fallback and Degradation Strategies

The foundation of resilient feature flag systems lies in **graceful degradation** - the ability to provide reduced but functional service when components fail. This requires establishing clear fallback hierarchies and ensuring that each layer can operate independently when upstream dependencies become unavailable.

### Decision: Multi-Layer Fallback Architecture

- **Context:** Feature flag evaluations must continue even when various system components fail, but maintaining consistency and performance during degradation is challenging
- **Options Considered:** Single cache fallback, distributed cache with local backup, hierarchical fallback with offline capability
- **Decision:** Implement hierarchical fallback with memory cache, local storage, and hardcoded defaults
- **Rationale:** Provides maximum availability while maintaining predictable performance characteristics and allowing for offline operation
- **Consequences:** Increases complexity but ensures evaluation requests never fail completely, though may serve stale data during extended outages

The fallback hierarchy follows a clear precedence order that balances freshness with availability:

Fallback Level	Data Source	Freshness	Availability	Latency
Primary	Flag Management API	Real-time	Network dependent	50-200ms
Secondary	Memory Cache	Minutes old	High	<1ms
Tertiary	Local Storage	Hours old	Very High	1-5ms
Ultimate	Default Values	Static	Guaranteed	<1ms

## Memory Cache Fallback Strategy

When the primary Flag Management API becomes unavailable, the evaluation engine immediately switches to its in-memory cache. This cache maintains a complete copy of all flag definitions with their targeting rules and variant configurations. The cache includes metadata about when each flag was last updated, allowing the system to make informed decisions about data freshness.

The memory cache fallback operates transparently to client applications. Flag evaluations continue with identical response formats and performance characteristics. However, the `EvaluationResult` includes a `Source` field that indicates "cache" rather than "api", allowing downstream systems to understand they may be receiving stale data.

Cache invalidation during fallback mode presents unique challenges. Since the system cannot receive real-time updates from the management API, it must implement intelligent cache refresh strategies. The system attempts periodic background refreshes with exponential backoff, but never blocks evaluation requests on these refresh attempts.

## Local Storage Persistence

For extended outages that exceed memory cache retention policies, the system falls back to persistent local storage. This storage layer maintains a complete snapshot of flag definitions along with evaluation metadata. The local storage uses a simple key-value structure where flag keys map to serialized `FlagDefinition` objects.

Local storage fallback occurs automatically when memory cache misses indicate that cached data has been evicted. The system logs these fallback events with appropriate severity levels to alert operations teams about degraded service quality. Despite using local storage, evaluation performance remains acceptable because modern storage systems provide sub-millisecond read access for small objects.

The local storage layer requires careful consideration of disk space management. Flag definitions can accumulate over time, especially for systems managing thousands of flags across multiple environments. The system implements a least-recently-used eviction policy that maintains the most actively evaluated flags while removing obsolete definitions.

## Default Value Resolution

The ultimate fallback mechanism relies on default values embedded directly in flag definitions. These defaults are hardcoded values that require no external dependencies and guarantee that evaluation requests never fail.

completely. Default values represent the "safe" configuration - typically the existing behavior before a feature flag was introduced.

Default value fallback must be carefully designed to avoid breaking critical business functionality. For boolean flags controlling new features, the default should typically be `false` to maintain existing behavior. For configuration flags that modify system parameters, defaults should represent well-tested values that maintain system stability.

The system provides clear visibility into default value usage through comprehensive logging and metrics. When evaluations fall back to default values, this indicates a significant service degradation that requires immediate attention. Operations teams should treat extended default value usage as a high-priority incident.

## Fallback Decision Logic

The fallback decision process follows a deterministic algorithm that ensures consistent behavior across all system components:

1. **Primary Evaluation Attempt:** The system first attempts to retrieve the flag definition from the Flag Management API with a short timeout (typically 100-200ms)
2. **Connection Health Check:** If the API request fails, the system checks connection health metrics to distinguish between temporary network issues and service outages
3. **Cache Consultation:** For failed API requests, the system immediately checks the memory cache for a cached flag definition
4. **Freshness Assessment:** If cached data exists, the system evaluates whether the data is fresh enough based on configured staleness tolerance
5. **Local Storage Query:** For cache misses or excessively stale data, the system queries persistent local storage
6. **Default Value Selection:** If no stored data is available, the system returns the hardcoded default value from the flag definition
7. **Circuit Breaker Update:** The system updates circuit breaker state based on the success or failure of each fallback attempt

## Performance During Degradation

Maintaining acceptable performance during degraded operation requires careful attention to timeout configuration and resource management. The system uses aggressive timeouts for fallback decisions to prevent cascade failures where slow upstream services cause downstream performance degradation.

Circuit breakers play a crucial role in performance preservation. Once a circuit breaker opens due to repeated API failures, the system immediately routes evaluation requests to cached data without attempting API calls. This prevents the accumulated latency of failed network requests from affecting user-facing performance.

Resource isolation ensures that fallback mechanisms don't compete with primary evaluation paths for system resources. Memory caches use separate allocation pools from real-time processing, and local storage operations use dedicated I/O queues that don't interfere with network communication.

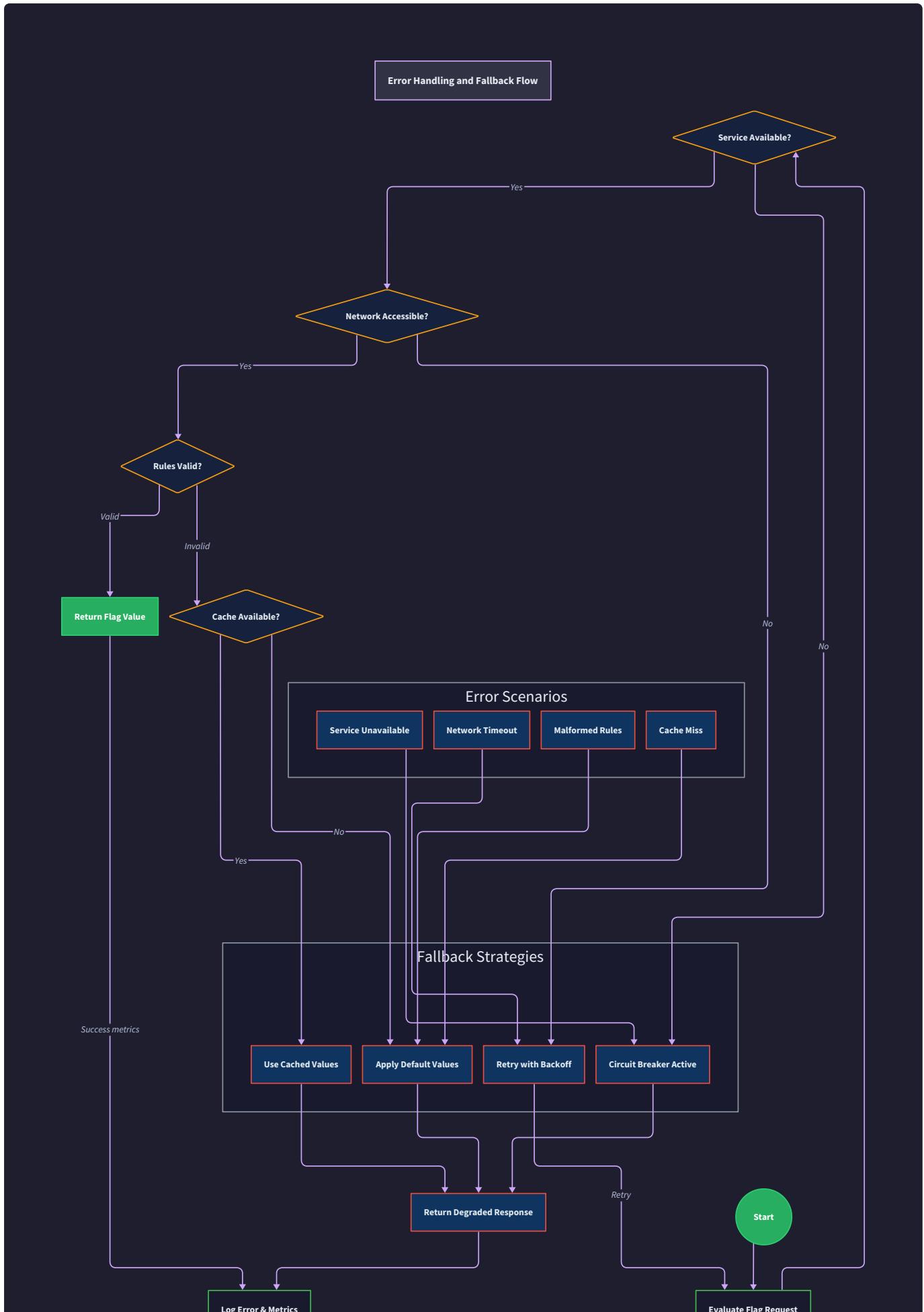
## **Consistency Guarantees During Fallback**

While fallback mechanisms prioritize availability over consistency, the system provides clear guarantees about what level of consistency clients can expect during degraded operation. These guarantees help application developers make appropriate decisions about feature flag usage in critical code paths.

During memory cache fallback, the system guarantees that all evaluation requests for the same user context will return identical results within the cache retention window. This ensures that user experiences remain consistent even during service outages.

Local storage fallback provides weaker consistency guarantees because data may be hours or days old. However, the system guarantees that it will return the same result for identical evaluation requests until fresh data becomes available. This prevents situations where users experience inconsistent behavior due to non-deterministic fallback logic.

Default value fallback provides the strongest consistency guarantee - all evaluation requests for a given flag will return identical results regardless of system state. This predictability allows applications to implement critical business logic with confidence that behavior will remain consistent during outages.



## Edge Cases and Corner Scenarios

Feature flag systems encounter numerous edge cases that can cause subtle bugs or system instability if not handled properly. These scenarios often arise from the complex interactions between targeting rules, user contexts, and system constraints under unusual operating conditions.

### Malformed Targeting Rules

Invalid or malformed targeting rules represent one of the most common edge cases in feature flag systems. These can result from manual configuration errors, automated rule generation bugs, or data corruption during storage or transmission.

The system must validate all rule components during flag definition creation and updates. Rule validation includes checking operator compatibility with attribute types, ensuring percentage allocations sum to valid ranges, and verifying that referenced user segments exist in the system.

Validation Type	Check Description	Error Response	Recovery Action
Operator Compatibility	Ensure operators match attribute types	Return validation error	Reject flag update
Percentage Range	Verify allocations are 0-100%	Return validation error	Apply automatic normalization
Segment References	Check referenced segments exist	Return validation error	Remove invalid references
Circular Dependencies	Detect prerequisite cycles	Return validation error	Reject flag update
Rule Complexity	Limit nested condition depth	Return validation error	Flatten rule structure

When malformed rules are detected during evaluation rather than creation, the system applies safe fallback behavior. Rather than throwing exceptions that could break application functionality, the evaluation engine logs detailed error information and returns default values with clear indication of the evaluation failure in the response metadata.

### Circular Flag Dependencies

Feature flags can reference other flags as prerequisites, creating dependency relationships that must be evaluated in proper order. Circular dependencies occur when flags reference each other either directly (A depends on B, B depends on A) or indirectly through longer chains ( $A \rightarrow B \rightarrow C \rightarrow A$ ).

Circular dependency detection uses depth-first search traversal of the flag dependency graph during flag creation and update operations. The system maintains an in-memory representation of all flag dependencies and validates that new dependency additions don't create cycles.

The detection algorithm maintains a visited set and a recursion stack during graph traversal:

1. **Initialize Traversal:** Start from the flag being updated and mark it as visited
2. **Follow Dependencies:** For each prerequisite flag, recursively traverse its dependencies
3. **Cycle Detection:** If a flag is encountered that's already in the current recursion stack, a cycle exists
4. **Path Recording:** Maintain the dependency path to provide detailed error messages about cycle location
5. **Validation Result:** Return success only if no cycles are detected in the complete dependency graph

When cycles are detected during flag updates, the system rejects the update with detailed error information including the complete cycle path. This helps flag administrators understand the dependency chain that would be violated and make appropriate corrections.

## Extreme Load Conditions

High-traffic applications can generate evaluation request volumes that exceed normal system capacity, especially during traffic spikes or when new flags are introduced that affect large user populations. The system must maintain functionality and prevent cascade failures under these extreme conditions.

Load shedding mechanisms activate when evaluation request rates exceed configured thresholds. Rather than accepting all requests and potentially causing system-wide failures, the system selectively drops requests using consistent criteria that minimize impact on critical functionality.

Load Level	Threshold	Response Strategy	Impact
Normal	<1000 req/sec	Standard evaluation	No impact
Elevated	1000-5000 req/sec	Enable additional caching	Slight staleness increase
High	5000-10000 req/sec	Aggressive cache utilization	Moderate staleness
Critical	>10000 req/sec	Load shedding activation	Some requests dropped

Request prioritization during load shedding considers multiple factors including flag importance, user segment membership, and evaluation complexity. Critical flags that control core business functionality receive higher priority than experimental flags used for A/B testing. Authenticated users may receive priority over anonymous users for personalized features.

## Inconsistent User Context Data

User context objects can contain malformed, missing, or inconsistent attribute data that complicates targeting rule evaluation. These issues often arise from client-side data collection errors, integration bugs, or schema evolution without proper migration.

Type coercion handles cases where attribute values don't match expected types in targeting rules. The system attempts intelligent conversion between compatible types (string to number, boolean to string) while maintaining evaluation consistency.

Missing attribute handling requires careful consideration of rule semantics. When a targeting rule references an attribute that's not present in the user context, the system must decide whether this should cause rule evaluation to fail or continue with default assumptions.

Missing Attribute Scenario	Rule Type	Behavior	Rationale
Required attribute absent	Exact match (equals)	Rule evaluates to false	Cannot match without value
Optional attribute absent	Range check (greater_than)	Rule evaluates to false	Cannot compare without value
Attribute type mismatch	Contains check	Attempt coercion, then false	Graceful degradation
Empty string/array	Membership test (in)	Rule evaluates to false	Empty collections match nothing

## Database Connection Failures

Persistent storage failures can occur due to database outages, network partitions, or resource exhaustion. The system must handle these failures gracefully while maintaining evaluation capability and preserving data consistency when connectivity is restored.

Connection pool management becomes critical during database failures. The system uses circuit breakers to detect database unavailability and prevent connection pool exhaustion from repeated failed connection attempts. Once a circuit opens, database operations immediately return cached or default data without attempting database access.

Recovery procedures activate when database connectivity is restored. The system must carefully validate data consistency, replay any missed updates, and gradually restore normal operation without overwhelming the recovered database with reconnection attempts from all system components simultaneously.

## Memory and Resource Exhaustion

Feature flag systems with large numbers of flags or complex targeting rules can encounter memory pressure, especially when maintaining comprehensive caches or processing high evaluation volumes. Resource exhaustion must be detected and handled before it affects system stability.

Memory management includes both proactive monitoring and reactive cleanup mechanisms. The system monitors heap usage, cache sizes, and evaluation queue lengths to detect approaching resource limits. When thresholds are exceeded, the system activates resource reclamation procedures.

Cache eviction policies become critical during memory pressure. The system uses intelligent eviction that considers flag evaluation frequency, data freshness, and flag importance. Rarely-used experimental flags are evicted before frequently-accessed production flags.

## Network Partitions and Split-Brain Scenarios

Distributed feature flag systems can experience network partitions that isolate different components from each other. During partitions, different parts of the system may have inconsistent views of flag configurations, leading to different evaluation results for identical contexts.

Partition detection relies on heartbeat mechanisms and consensus protocols where appropriate. Components that detect they're isolated from the majority of the system should enter a degraded mode that prioritizes consistency over availability.

Split-brain prevention ensures that conflicting flag updates don't occur simultaneously in different partitions. The system uses techniques like majority quorums or designated master selection to ensure only one partition can make authoritative flag changes during network partitions.

## Data Corruption and Consistency Violations

Storage corruption can cause flag definitions to contain invalid data that passes initial validation but causes evaluation errors. This can result from hardware failures, software bugs, or concurrent modification issues.

Checksum validation protects against storage-level corruption by maintaining hash values for all flag definitions. Before evaluating a flag, the system verifies that the definition's checksum matches its stored content and falls back to cached or default data if corruption is detected.

Consistency repair mechanisms activate when corruption is detected. The system can rebuild flag definitions from backup data, request fresh copies from authoritative sources, or temporarily disable corrupted flags until manual intervention restores proper data.

**⚠ Pitfall: Insufficient Error Context** Many implementations provide generic error messages like "evaluation failed" without sufficient context for debugging. This makes production issues extremely difficult to diagnose. Always include flag key, user context summary, error type, and fallback action taken in error messages and logs.

**⚠ Pitfall: Synchronous Fallback Operations** Attempting to perform expensive fallback operations (like local storage queries) synchronously during evaluation requests can cause timeout cascades. Implement asynchronous background processes for fallback data preparation and use in-memory structures for fast fallback decisions during request handling.

**⚠ Pitfall: Inconsistent Error Handling** Different system components handling the same error conditions in different ways can cause confusing behavior and difficult debugging. Establish consistent error handling patterns and ensure all components follow the same fallback hierarchies and timeout behaviors.

## Implementation Guidance

The error handling and fallback mechanisms require careful coordination between all system components. This implementation guidance provides the infrastructure and patterns needed to build robust error handling that maintains system reliability under various failure conditions.

## Technology Recommendations

Component	Simple Option	Advanced Option
Circuit Breaker	Manual state tracking with counters	go-kit/kit circuit breaker with metrics
Error Tracking	Standard Go error handling with logging	Sentry or similar error aggregation service
Fallback Storage	Local file-based JSON storage	Embedded database like BadgerDB or SQLite
Health Monitoring	HTTP health endpoints with basic checks	Comprehensive metrics with Prometheus
Configuration Validation	JSON schema validation	Custom validator with business rule checks

## Recommended File Structure

```
internal/
  errors/
    errors.go      ← Custom error types and handling utilities
    fallback.go    ← Fallback hierarchy implementation
    circuit_breaker.go ← Circuit breaker for service dependencies
    validation.go   ← Flag and context validation logic
  storage/
    local_cache.go ← Memory cache with LRU eviction
    local_storage.go ← Persistent local storage interface
    file_storage.go ← File-based storage implementation
  health/
    checker.go     ← Health monitoring and metrics
    recovery.go    ← Automatic recovery procedures
```

## Error Type Definitions

```
package errors

import (
    "fmt"
    "time"
)

// FlagError represents all feature flag related errors with context

type FlagError struct {

    Type      Errortype      `json:"type"`
    Flagkey   string         `json:"flag_key,omitempty"`
    Message   string         `json:"message"`
    Cause     error          `json:"-"`
    Context   map[string]interface{} `json:"context,omitempty"`
    Timestamp time.Time      `json:"timestamp"`
    Recoverable bool          `json:"recoverable"`
}

func (e *FlagError) Error() string {
    return fmt.Sprintf("flag error [%s]: %s (flag: %s)", e.Type, e.Message, e.Flagkey)
}

func (e *FlagError) Unwrap() error {
    return e.Cause
}

// Errortype categorizes different failure modes for appropriate handling

type Errortype string

const (
```

GO

```
    ErrorTypeValidation     ErrorType = "validation"  
  
    ErrorTypeNetwork       ErrorType = "network"  
  
    ErrorTypeStorage       ErrorType = "storage"  
  
    ErrorTypeEvaluation    ErrorType = "evaluation"  
  
    ErrorTypeCircuit        ErrorType = "circuit_breaker"  
  
    ErrorTypeDependency     ErrorType = "dependency"  
  
    ErrorTypeResource       ErrorType = "resource_exhaustion"  
  
)
```

## Circuit Breaker Implementation

```
package errors

import (
    "sync"
    "time"
)

// CircuitBreaker prevents cascade failures by detecting repeated errors

type CircuitBreaker struct {

    name          string
    maxFailures   int
    resetTimeout  time.Duration
    state         CircuitState
    failures      int
    lastFailTime  time.Time
    successCount  int
    mu            sync.RWMutex
}

type CircuitState string

const (
    StateClosed   CircuitState = "closed"     // Normal operation
    StateOpen     CircuitState = "open"        // Blocking all requests
    StateHalfOpen CircuitState = "half_open"   // Testing recovery
)

func NewCircuitBreaker(name string, maxFailures int, resetTimeout time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
```

GO

```
        name:           name,
        maxFailures:   maxFailures,
        resetTimeout:  resetTimeout,
        state:         StateClosed,
    }

}

// Allow determines if a request should be processed or rejected

func (cb *CircuitBreaker) Allow() error {

    // TODO: Implement circuit breaker allow logic

    // 1. Check current state under read lock

    // 2. For StateClosed: always allow

    // 3. For StateOpen: check if reset timeout has passed
    //     - If timeout passed, transition to StateHalfOpen and allow
    //     - Otherwise reject with circuit breaker error

    // 4. For StateHalfOpen: allow (testing recovery)

    return nil
}

// RecordResult updates circuit breaker state based on operation outcome

func (cb *CircuitBreaker) RecordResult(err error) {

    // TODO: Update circuit breaker state based on result

    // 1. Acquire write lock for state modification

    // 2. If err != nil: increment failure count and record timestamp
    //     - Check if failures exceed threshold and transition to StateOpen

    // 3. If err == nil and state is StateHalfOpen:
    //     - Increment success count and check if should transition to StateClosed

    // 4. If err == nil and state is StateClosed: reset failure count
}
```

```
}

// State returns current circuit breaker state for monitoring

func (cb *CircuitBreaker) State() CircuitState {

    cb.mu.RLock()

    defer cb.mu.RUnlock()

    return cb.state

}
```

## Fallback Handler Infrastructure

```
package errors

import (
    "context"
    "time"
)

// FallbackHandler manages the complete fallback hierarchy

type FallbackHandler struct {

    primaryAPI     FlagAPI

    memoryCache   *MemoryCache

    localStorage  LocalStorage

    circuitBreaker *CircuitBreaker

    metrics        *FallbackMetrics

}

// FlagAPI interface for primary flag data source

type FlagAPI interface {

    GetFlag(ctx context.Context, flagKey string) (*FlagDefinition, error)
}

// FallbackMetrics tracks fallback usage for monitoring

type FallbackMetrics struct {

    APIAttempts     int64

    CacheHits       int64

    LocalStorageHits int64

    DefaultValueHits int64

    mu              sync.RWMutex

}
```

GO

```
func NewFallbackHandler(api FlagAPI, cache *MemoryCache, storage LocalStorage) *FallbackHandler {
    return &FallbackHandler{
        primaryAPI:      api,
        memoryCache:     cache,
        localStorage:   storage,
        circuitBreaker:  NewCircuitBreaker("flag-api", 5, 30*time.Second),
        metrics:         &FallbackMetrics{},
    }
}

// GetFlagWithFallback implements complete fallback hierarchy

func (fh *FallbackHandler) GetFlagWithFallback(ctx context.Context, flagKey string) (*FlagDefinition, string, error) {
    // TODO: Implement complete fallback sequence

    // 1. Check circuit breaker - if open, skip API and go to cache

    // 2. Try primary API with short timeout context
    //
    //     - If success: update cache, record success, return with source "api"
    //
    //     - If failure: record failure, continue to fallback

    // 3. Try memory cache
    //
    //     - Check staleness tolerance before returning cached data
    //
    //     - If found and fresh: return with source "cache"

    // 4. Try local storage
    //
    //     - Load from persistent storage
    //
    //     - If found: update memory cache, return with source "storage"

    // 5. Return default value from flag definition with source "default"

    // 6. Update metrics for monitoring
}
```

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

## Validation Framework

```
package errors

import (
    "fmt"
    "strings"
)

// ValidationErrors collects multiple validation failures

type ValidationErrors struct {
    Errors []ValidationError `json:"errors"`
}

func (ve *ValidationErrors) Error() string {
    var messages []string
    for _, err := range ve.Errors {
        messages = append(messages, err.Error())
    }
    return fmt.Sprintf("validation failed: %s", strings.Join(messages, "; "))
}

func (ve *ValidationErrors) Add(field, message string) {
    ve.Errors = append(ve.Errors, ValidationError{
        Field:   field,
        Message: message,
    })
}

func (ve *ValidationErrors) HasErrors() bool {
    return len(ve.Errors) > 0
}
```

GO

```
}

// ValidationError represents a single validation failure

type ValidationError struct {

    Field  string `json:"field"`

    Message string `json:"message"`

}

func (ve ValidationError) Error() string {

    return fmt.Sprintf("field '%s': %s", ve.Field, ve.Message)

}

// ValidateFlagDefinition performs comprehensive flag validation

func ValidateFlagDefinition(flag *FlagDefinition) error {

    // TODO: Implement comprehensive flag validation

    // 1. Check flag key format and length

    // 2. Validate default value matches flag type

    // 3. For each targeting rule:

    //     - Validate attribute references

    //     - Check operator compatibility with attribute types

    //     - Verify percentage allocations sum correctly

    //     - Validate segment references exist

    // 4. Check for circular dependencies in prerequisites

    // 5. Validate variant definitions match flag type

    // 6. Ensure at least one variant or default value exists

    return nil

}
```

```
// ValidateUserContext ensures context data meets requirements

func ValidateUserContext(ctx *UserContext) error {

    // TODO: Validate user context structure

    // 1. Check UserID is not empty

    // 2. Validate attribute values are compatible types

    // 3. Check segment names follow naming conventions

    // 4. Verify context size doesn't exceed limits

    return nil
}
```

## Recovery and Health Monitoring

```
package health

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

// HealthChecker monitors system component health and triggers recovery

type HealthChecker struct {
    components map[string]HealthComponent
    status     map[string]ComponentHealth
    mu         sync.RWMutex
    interval   time.Duration
    stopCh     chan struct{}
}

type ComponentHealth struct {

    Status      HealthStatus `json:"status"`
    LastCheck   time.Time    `json:"last_check"`
    LastError   string       `json:"last_error,omitempty"`
    CheckCount  int64        `json:"check_count"`
    ErrorCount  int64        `json:"error_count"`
}

type HealthStatus string

const (
    HealthStatusHealthy  HealthStatus = "healthy"
```

GO

```
    HealthStatusDegraded  HealthStatus = "degraded"

    HealthStatusUnhealthy HealthStatus = "unhealthy"

}

type HealthComponent interface {

    Name() string

    Check(ctx context.Context) error

    Recover(ctx context.Context) error

}

func NewHealthChecker(interval time.Duration) *HealthChecker {

    return &HealthChecker{

        components: make(map[string]HealthComponent),

        status:     make(map[string]ComponentHealth),

        interval:   interval,

        stopCh:     make(chan struct{}),

    }

}

// StartMonitoring begins periodic health checks and automatic recovery

func (hc *HealthChecker) StartMonitoring(ctx context.Context) {

    // TODO: Implement health monitoring loop

    // 1. Create ticker for periodic checks

    // 2. For each component, run health check

    // 3. Update component status based on check result

    // 4. If component unhealthy, attempt recovery

    // 5. Log health status changes

    // 6. Handle context cancellation for graceful shutdown
}
```

}

## Milestone Checkpoints

### After Milestone 1 - Flag Evaluation Engine:

- Test flag evaluation with invalid rule conditions: `go test -run TestEvaluationWithInvalidRules`
- Verify fallback to default values when rules are malformed
- Check that evaluation errors include proper context information
- Expected: Evaluation succeeds with clear indication of fallback usage

### After Milestone 2 - Real-time Flag Updates:

- Test connection failure scenarios: disconnect network and verify client behavior
- Check exponential backoff during reconnection attempts
- Verify circuit breaker opens after repeated connection failures
- Expected: Clients continue operating with cached data during outages

### After Milestone 3 - Flag Analytics & Experiments:

- Test analytics service failure during flag evaluation
- Verify that exposure tracking failures don't affect flag evaluation
- Check that experiment assignment continues when analytics are unavailable
- Expected: Flag evaluations continue normally even when analytics fail

## Debugging Checklist

Symptom	Likely Cause	How to Diagnose	Fix
Evaluations always return defaults	Circuit breaker stuck open	Check circuit breaker state and error logs	Reset circuit breaker or fix underlying service
Inconsistent evaluation results	Cache consistency issues	Compare cache contents across instances	Implement cache invalidation or restart services
Memory leaks during errors	Error context not being cleaned up	Monitor heap usage during error scenarios	Add proper cleanup in error handling paths
Slow recovery after outages	Thundering herd on reconnection	Check connection attempt timing	Add jittered exponential backoff
Missing error context in logs	Generic error handling	Review error logging patterns	Enhance errors with flag keys and user context

# Testing Strategy

**Milestone(s):** This section applies to all three milestones by establishing comprehensive testing strategies for validating the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3).

Testing a feature flag system requires a multi-layered approach that goes beyond traditional unit tests. Think of testing feature flags like **stress-testing air traffic control systems** — you need to verify that the system works correctly under normal conditions, handles edge cases gracefully, and maintains safety guarantees even when components fail. Just as air traffic controllers run emergency scenarios to ensure pilots can land safely during equipment failures, we must test that our feature flag system maintains consistent user experiences even when services are degraded.

The testing strategy for a feature flag system faces unique challenges that don't exist in typical CRUD applications. Feature flags must maintain **consistent user assignments** across evaluations, ensure **real-time updates propagate correctly** without causing inconsistencies, and track **statistically valid experiments** without introducing bias. A single bug in consistent hashing could cause users to flip between variants, invalidating an entire A/B test. A race condition in update propagation could show users conflicting feature states within the same session.

## Milestone Validation Checkpoints

Each milestone introduces distinct functionality that requires specific validation approaches. The checkpoints serve as **quality gates** that verify the system meets its acceptance criteria before proceeding to the next milestone. Think of these checkpoints like **preflight inspections** — systematically verifying each system component before allowing the next phase to begin.

### Milestone 1: Flag Evaluation Engine Validation

The Flag Evaluation Engine validation focuses on correctness of targeting logic, consistency of user assignments, and performance under load. The core challenge is ensuring that complex targeting rules evaluate deterministically and that percentage rollouts assign users consistently across multiple evaluations.

Test Category	What to Validate	Expected Behavior	Failure Indicators
Basic Flag Evaluation	Simple boolean flags return correct values	<code>EvaluateFlag("simple-toggle", userContext) returns {Value: true, Variant: "enabled", Reason: "targeting_match"}</code>	Wrong boolean value returned, missing variant information
Targeting Rule Logic	Complex AND/OR conditions evaluate correctly	User matching <code>userType=premium AND region=us-east</code> gets correct variant	Rule precedence violations, incorrect boolean logic
Percentage Rollouts	Consistent user assignment to percentage buckets	Same user gets same variant across multiple evaluations	User assignment changes between calls, incorrect bucket distribution
Rule Precedence	Higher priority rules override lower priority ones	Targeting rules evaluated before default percentage rollout	Lower priority rules incorrectly taking precedence
Context Validation	Invalid user contexts are handled gracefully	Missing required attributes cause fallback to default value	System crashes on malformed context, silent failures
Performance Baseline	Evaluation latency meets requirements	Single flag evaluation completes under 1ms, 10k evaluations/sec sustained	High latency, memory leaks, CPU spikes

The **consistent hashing validation** is particularly critical for Milestone 1. Create test scenarios with the same user evaluating the same flag thousands of times — the variant assignment must never change. Additionally, test that percentage allocations distribute users correctly by evaluating flags for large user populations and measuring actual distribution against expected percentages.

```
Validation Command: go test ./internal/evaluation/... -v -race
Expected Output: All tests pass, no race conditions detected
Manual Verification: Start evaluation server, send 1000 requests for same user/flag, verify identical responses
```

## Milestone 2: Real-time Flag Updates Validation

Real-time updates introduce complexity around connection management, message ordering, and cache consistency. The validation must ensure that flag changes propagate to all connected clients within acceptable time bounds and that clients handle connection failures gracefully.

Test Category	What to Validate	Expected Behavior	Failure Indicators
Update Propagation	Flag changes reach all connected clients	Change broadcast within 100ms to all SSE connections	Clients not receiving updates, significant propagation delays
Connection Recovery	Clients reconnect after network interruption	Exponential backoff, replay missed events, resume normal operation	Infinite retry loops, missed updates after reconnection
Message Ordering	Updates arrive in correct sequence	Event sequence numbers increment monotonically, no gaps	Out-of-order delivery, duplicate or missing events
Cache Invalidation	Stale flag data is replaced promptly	Local cache updated within propagation timeout	Clients serving stale data after update notification
Connection Limits	System handles connection scaling gracefully	Support target number of concurrent SSE connections	Memory exhaustion, connection drops under load
Fallback Behavior	Clients function when update service unavailable	Fall back to cached data, continue flag evaluations	Service unavailable causes client failures

The **connection state machine validation** requires simulating various network conditions. Test clients connecting, disconnecting, experiencing intermittent failures, and recovering. Verify that the  `ConnectionState` transitions correctly through  `StateDisconnected` ,  `StateConnecting` ,  `StateConnected` , and  `StateReconnecting` states based on network events.

```
Validation Command: go test ./internal/realtime/... -v -timeout 30s
Expected Output: Connection state transitions verified, no goroutine leaks
Manual Verification: Start server with 100 concurrent clients, toggle flag via API, verify all clients receive update
```

### Milestone 3: Flag Analytics & Experiments Validation

Analytics and experimentation validation focuses on data accuracy, statistical correctness, and performance of the tracking pipeline. The challenge is ensuring that exposure events are captured reliably and that statistical calculations produce valid results.

Test Category	What to Validate	Expected Behavior	Failure Indicators
Exposure Tracking	All flag evaluations are logged accurately	Each <code>EvaluateFlag</code> call generates corresponding <code>FlagExposure</code> event	Missing exposure events, incorrect exposure data
Statistical Calculations	Significance tests produce correct results	Known experiment data yields expected p-values and confidence intervals	Wrong statistical results, confidence intervals don't match
Sample Ratio Detection	Mismatch between intended and actual allocations detected	50/50 split with 60/40 actual triggers <code>SampleRatioMismatch</code> flag	Sample ratio mismatches not detected, false positives
Experiment Assignment	Users consistently assigned to same variant	Same user gets same experiment variant across sessions	User variant assignments change over time
Performance Pipeline	High-volume exposure tracking without data loss	Process 10k exposures/sec without dropping events	Event queue overflow, processing delays, data loss
Analytics Aggregation	Real-time metrics accurately reflect exposure data	Conversion rates match raw exposure data calculations	Aggregation errors, metrics don't match raw data

The **statistical significance validation** requires test datasets with known ground truth. Create synthetic experiment data where variant A has a 5% conversion rate and variant B has a 6% conversion rate, then verify that your `CalculateSignificance` function produces the expected p-values and confidence intervals for different sample sizes.

```
Validation Command: go test ./internal/analytics/... -v -bench=BenchmarkExposureTracking
Expected Output: Statistical calculations match expected values, throughput targets achieved
Manual Verification: Run synthetic A/B test, compare calculated significance to external statistical tool
```

## Cross-Milestone Integration Validation

The final validation step verifies that all three milestones work together correctly. This integration testing simulates realistic usage patterns where clients evaluate flags, receive real-time updates, and generate analytics data simultaneously.

Integration Scenario	Validation Steps	Success Criteria
Flag Update During Active Experiment	Modify flag while experiment running, verify exposure tracking continues	No gaps in exposure data, users maintain variant assignments
High-Load Mixed Operations	Concurrent flag evaluations, updates, and analytics queries	All operations succeed, no degradation in any component
Partial Service Degradation	Disable analytics service, verify flag evaluation continues	Flag evaluation unaffected, analytics queue buffers data

## Property-based Testing

Property-based testing is particularly valuable for feature flag systems because it can discover edge cases that traditional example-based tests miss. Instead of testing specific inputs and outputs, property-based tests verify **mathematical invariants** that should hold regardless of the specific input values. Think of property-based testing like **physics laws** — just as gravity always pulls objects downward regardless of their mass or composition, certain properties of your feature flag system should always hold true regardless of the specific flags, users, or configurations involved.

Property-based testing excels at finding the subtle bugs that can destroy user trust in feature flags. A traditional unit test might verify that user "alice" gets variant "treatment" for flag "checkout-flow", but a property-based test would verify that ANY user gets the SAME variant every time they're evaluated for ANY flag. This approach discovers bugs like hash function inconsistencies, floating-point precision issues in percentage calculations, and race conditions in concurrent access patterns.

### Consistent Hashing Stability Properties

The most critical property to test is **deterministic user assignment** — the same user must always receive the same variant for a given flag configuration, regardless of when or how many times the evaluation occurs. This property is foundational to user experience and experiment validity.

Property	Description	Test Strategy
Assignment Stability	Same user + same flag + same config = same variant	Generate random users and flags, evaluate multiple times, assert all results identical
Percentage Distribution	Large user populations distribute according to percentage allocations	Generate 10k random users, evaluate against 50/50 split, assert 45-55% in each bucket
Independence Across Flags	User's variant for one flag doesn't influence variant for other flags	Generate user + multiple flags, assert variant assignments are uncorrelated
Hash Function Uniformity	User IDs distribute evenly across bucket space	Generate 10k sequential user IDs, assert bucket assignments don't cluster

The **assignment stability property** can be implemented using property-based testing frameworks by generating random `UserID` and `FlagKey` combinations, then evaluating the same combination multiple times and asserting that the `EvaluationResult.Variant` never changes. This test should run with thousands of random inputs to discover edge cases like specific user ID patterns that cause hash collisions.

```
// Property: Same user + same flag = same variant (always) GO

func TestAssignmentStability(t *testing.T) {

    // TODO 1: Generate random UserID and FlagKey

    // TODO 2: Create FlagDefinition with percentage rollout

    // TODO 3: Call EvaluateFlag 100 times with same inputs

    // TODO 4: Assert all results have identical Variant field

    // TODO 5: Repeat with 1000 different random user/flag combinations

}
```

## Rule Evaluation Determinism Properties

Complex targeting rules with AND/OR operators and multiple conditions must evaluate deterministically regardless of the order in which conditions are processed or the internal representation of the rule tree.

Property	Description	Test Strategy
Boolean Logic Correctness	AND/OR operators follow mathematical laws (associativity, commutativity)	Generate random rule combinations, verify against truth tables
Condition Order Independence	Same conditions in different orders produce same result	Generate rule with multiple conditions, test all permutations
Context Attribute Type Safety	Same logical value in different types produces same result	Test string "true" vs boolean true, integer 1 vs string "1"
Rule Precedence Consistency	Higher priority rules always override lower priority ones	Generate rules with random priorities, assert highest priority determines result

## Cache Consistency Properties

Real-time flag updates must maintain consistency between cached and authoritative flag definitions. Cache inconsistencies can cause the same user to see different feature behavior within a short time window, creating poor user experience.

Property	Description	Test Strategy
Read-After-Write Consistency	Flag reads return updated value after write completes	Write flag update, immediately read from cache, assert new value returned
Invalidation Completeness	Cache invalidation removes ALL stale entries	Update flag, trigger invalidation, assert no cache level returns old value
Concurrent Access Safety	Multiple threads reading/writing cache don't corrupt data	Generate concurrent read/write operations, assert no race conditions
TTL Enforcement	Expired cache entries are not served	Set short TTL, wait for expiration, assert cache miss triggers refresh

## Statistical Analysis Properties

A/B test analysis must produce mathematically correct results that match established statistical formulas. Incorrect statistical calculations can lead to false conclusions about feature effectiveness.

Property	Description	Test Strategy
Significance Calculation Accuracy	P-values and confidence intervals match statistical formulas	Generate synthetic data with known ground truth, verify calculations
Sample Size Monotonicity	Larger samples never decrease statistical power	Generate data with increasing sample sizes, assert power never decreases
Alpha Level Consistency	Significance threshold correctly applied across all tests	Test data at boundary conditions, verify alpha=0.05 behaves correctly
Effect Size Calculation	Effect size measures correctly reflect practical significance	Generate data with known effect sizes, verify calculated values

The **significance calculation accuracy** property can be tested by generating synthetic experiment data using known statistical distributions, then verifying that your `CalculateSignificance` function produces results that match calculations from established statistical libraries like R or Python's `scipy.stats`.

## Implementation Strategy for Property-based Tests

Property-based testing frameworks like Go's `gopter` or `testing/quick` generate hundreds or thousands of random test cases automatically. The key is designing property functions that encode the mathematical invariants your system must maintain.

Component	Property Function Signature	Random Generators Needed
Evaluation Engine	<pre>func TestAssignmentStability(user UserID, flag FlagKey) bool</pre>	UserID generator, FlagKey generator, percentage generator
Rule Engine	<pre>func TestRuleDeterminism(rules []TargetingRule, ctx UserContext) bool</pre>	Rule generator, condition generator, context generator
Cache Layer	<pre>func TestCacheConsistency(operations []CacheOperation) bool</pre>	Cache operation generator, flag definition generator
Statistics	<pre>func TestSignificanceAccuracy(experimentData ExperimentData) bool</pre>	Conversion rate generator, sample size generator

Each property function should return `true` if the invariant holds and `false` if it's violated. The testing framework automatically shrinks failing test cases to find minimal examples that violate the property, making debugging much easier than trying to reproduce complex failure scenarios manually.

**Design Insight:** Property-based testing is particularly valuable for feature flag systems because bugs often manifest only under specific combinations of user attributes, flag configurations, or timing conditions that are difficult to anticipate when writing example-based tests.

## Common Property-based Testing Pitfalls

### ⚠️ Pitfall: Testing Implementation Details Instead of Business Properties

Many developers write property-based tests that verify implementation details like "hash function returns 32-bit integer" rather than business properties like "user assignment remains stable". This leads to brittle tests that break when implementation changes but don't catch real bugs that affect users.

**Why it's wrong:** Implementation details can change without affecting system behavior. Testing them creates maintenance burden without providing confidence in system correctness.

**How to fix:** Focus on observable behavior and business invariants. Test that "users get consistent variants" rather than "hash function returns specific values".

### ⚠️ Pitfall: Insufficient Random Data Generation

Property-based tests are only as good as the random data they generate. Tests that generate only simple, well-formed inputs miss edge cases like empty strings, special characters, extreme numeric values, or unusual data combinations.

**Why it's wrong:** Real-world data contains edge cases that simple generators miss. Bugs often lurk in boundary conditions and unusual input combinations.

**How to fix:** Design generators that produce edge cases: empty collections, null values, extreme numbers, special characters, and boundary conditions relevant to your domain.

## Implementation Guidance

### A. Technology Recommendations:

Testing Component	Simple Option	Advanced Option
Property-based Testing	<code>testing/quick</code> (Go stdlib)	<code>gopter</code> with custom generators
Load Testing	<code>go test -bench</code> with table tests	<code>k6</code> with realistic traffic patterns
Integration Testing	<code>httptest</code> with test servers	<code>testcontainers</code> with real dependencies
Statistical Validation	Manual calculation verification	<code>gonum/stat</code> for reference implementations
Test Data Management	JSON fixtures in <code>testdata/</code>	<code>factory_bot</code> style builders with randomization
Milestone Validation	<code>make test</code> with milestone-specific tags	CI pipeline with environment-specific test suites

### B. Recommended Test Structure:

```
internal/
  evaluation/
    evaluation.go
    evaluation_test.go      ← unit tests
    evaluation_property_test.go ← property-based tests
    evaluation_integration_test.go ← integration scenarios
  realtime/
    realtime_test.go
    realtime_load_test.go     ← connection scaling tests
  analytics/
    analytics_test.go
    analytics_statistical_test.go ← statistical accuracy tests
test/
  fixtures/                  ← test data
    flags.json
    users.json
    experiments.json
  integration/              ← cross-component tests
    milestone_1_test.go
    milestone_2_test.go
    milestone_3_test.go
  property/                 ← property-based test generators
    generators.go           ← random data generation
    properties.go           ← invariant definitions
```

### C. Test Data Generators (Complete Implementation):

```
package property

import (
    "math/rand"
    "time"
    "testing/quick"
)

// UserGenerator creates realistic UserID values for property testing

func UserGenerator(rand *rand.Rand, size int) reflect.Value {
    // Generate mix of numeric IDs, UUIDs, and email-like strings

    patterns := []func() string{
        func() string { return fmt.Sprintf("user_%d", rand.Intn(100000)) },
        func() string { return fmt.Sprintf("test+%d@example.com", rand.Intn(10000)) },
        func() string {
            return fmt.Sprintf("%08x-%04x-%04x",
                rand.Uint32(), rand.Uint32()&0xFFFF, rand.Uint32()&0xFFFF)
        },
    }
    pattern := patterns[rand.Intn(len(patterns))]

    return reflect.ValueOf(userID(pattern()))
}

// FlagGenerator creates realistic flag definitions for property testing

func FlagGenerator(rand *rand.Rand, size int) reflect.Value {
    flag := &FlagDefinition{
        Key:      FlagKey(fmt.Sprintf("test_flag_%d", rand.Intn(1000))),
        Name:     fmt.Sprintf("Test Flag %d", rand.Intn(1000)),
    }
}
```

GO

```
Description: "Generated for property testing",

Enabled:      rand.Float32() > 0.1, // 90% enabled

Variants: []Variant{

    {Key: "control", Value: false, Weight: rand.Intn(50) + 25},

    {Key: "treatment", Value: true, Weight: rand.Intn(50) + 25},

},

}

// Add random targeting rules 20% of the time

if rand.Float32() > 0.8 {

    flag.TargetingRules = generateRandomRules(rand, size)

}

return reflect.ValueOf(flag)

}

// ContextGenerator creates diverse user contexts for testing

func ContextGenerator(rand *rand.Rand, size int) reflect.Value {

    attributes := make(map[string]interface{})

    // Add common attribute types

    attributes["user_type"] = []string{"premium", "basic", "trial"}[rand.Intn(3)]

    attributes["region"] = []string{"us-east", "us-west", "eu", "asia"}[rand.Intn(4)]

    attributes["account_age_days"] = rand.Intn(1000)

    attributes["is_beta_user"] = rand.Float32() > 0.7

}

// Add random attributes 30% of the time
```

```
if rand.Float32() > 0.7 {  
  
    randomKey := fmt.Sprintf("custom_%d", rand.Intn(100))  
  
    attributes[randomKey] = fmt.Sprintf("value_%d", rand.Intn(100))  
  
}  
  
  
  
return reflect.ValueOf(UserContext{  
  
    UserID:      UserID(fmt.Sprintf("user_%d", rand.Intn(10000))),  
  
    Attributes: attributes,  
  
    Segments:   generateRandomSegments(rand),  
  
})  
  
}
```

#### D. Property Test Examples (Core Logic Skeletons):

GO

```
// TestAssignmentStabilityProperty verifies consistent user assignment

func TestAssignmentStabilityProperty(t *testing.T) {
    property := func(userID UserID, flag *FlagDefinition) bool {
        // TODO 1: Create evaluation engine with test configuration
        // TODO 2: Build user context with consistent attributes
        // TODO 3: Evaluate flag 100 times in tight loop
        // TODO 4: Verify all evaluations return identical variant
        // TODO 5: Return true if stable, false if any variation detected

        // Hint: Store first result, compare all subsequent results
        // Hint: Use reflect.DeepEqual for EvaluationResult comparison
        return true // Replace with actual stability check
    }

    if err := quick.Check(property, &quick.Config{
        MaxCount: 1000,
        Rand:     rand.New(rand.NewSource(time.Now().UnixNano())),
    }); err != nil {
        t.Errorf("Assignment stability property violated: %v", err)
    }
}

// TestPercentageDistributionProperty verifies rollout percentages

func TestPercentageDistributionProperty(t *testing.T) {
    property := func(percentage int) bool {
        // Clamp percentage to valid range
        if percentage < 5 || percentage > 95 {

```

```
        return true // Skip extreme values
    }

// TODO 1: Create flag with specified percentage allocation

// TODO 2: Generate 10,000 diverse user contexts

// TODO 3: Evaluate flag for each user, count variant assignments

// TODO 4: Calculate actual percentage distribution

// TODO 5: Verify actual percentage within ±5% of expected

// Hint: Use chi-square test for distribution validation

// Hint: Account for small sample variation in acceptance criteria

return true // Replace with actual distribution check
}

if err := quick.Check(property, nil); err != nil {
    t.Errorf("Percentage distribution property violated: %v", err)
}
}

// TestStatisticalAccuracyProperty verifies significance calculations

func TestStatisticalAccuracyProperty(t *testing.T) {
    property := func(controlRate, treatmentRate float64, sampleSize int) bool {
        // Validate input ranges
        if controlRate < 0.01 || controlRate > 0.99 ||
            treatmentRate < 0.01 || treatmentRate > 0.99 ||
            sampleSize < 100 || sampleSize > 10000 {
            return true // Skip invalid inputs
        }
        // ... rest of the function logic ...
    }
    quick.Check(property, nil)
}
}
```

```
}

// TODO 1: Generate synthetic experiment data with known rates

// TODO 2: Call CalculateSignificance with synthetic data

// TODO 3: Calculate expected p-value using reference implementation

// TODO 4: Verify calculated p-value within acceptable tolerance

// TODO 5: Verify confidence intervals contain true effect size

// Hint: Use gonum/stat for reference statistical calculations

// Hint: Allow 1% tolerance for floating-point precision differences

return true // Replace with actual accuracy verification

}

if err := quick.Check(property, &quick.Config{MaxCount: 100}); err != nil {

    t.Errorf("Statistical accuracy property violated: %v", err)

}

}
```

## E. Milestone Checkpoint Implementation:

GO

```
// Milestone1ValidationSuite runs comprehensive validation for flag evaluation

func Milestone1ValidationSuite(t *testing.T) {

    t.Run("BasicFunctionality", func(t *testing.T) {

        // TODO 1: Start evaluation server on test port

        // TODO 2: Create simple boolean flag via API

        // TODO 3: Evaluate flag with test user context

        // TODO 4: Verify response structure and timing

        // TODO 5: Clean up test resources

    })

    t.Run("LoadTesting", func(t *testing.T) {

        // TODO 1: Create flag with complex targeting rules

        // TODO 2: Generate 10,000 concurrent evaluation requests

        // TODO 3: Measure average response time and error rate

        // TODO 4: Verify all responses are deterministic

        // TODO 5: Check for memory leaks and resource cleanup

    })

    t.Run("EdgeCases", func(t *testing.T) {

        // TODO 1: Test evaluation with malformed user context

        // TODO 2: Test evaluation with non-existent flag key

        // TODO 3: Test evaluation with circular flag dependencies

        // TODO 4: Verify graceful degradation in all cases

        // TODO 5: Check error messages are informative

    })

}

}
```

```
// checkpoint command: go test -tags= milestone1 -v -timeout=60s ./test/integration/
// expected: All subtests pass, load test achieves >1000 QPS, <1ms avg latency
```

## F. Debugging Property Test Failures:

Symptom	Likely Cause	Diagnosis Steps	Resolution
Property fails on specific input combination	Edge case in evaluation logic	Run failing case in debugger, examine intermediate values	Add bounds checking, handle edge case explicitly
Property fails intermittently	Race condition or timing dependency	Run with <code>-race</code> flag, add logging to identify timing issues	Add proper synchronization, eliminate timing dependencies
Property fails on large inputs only	Performance degradation or resource exhaustion	Profile memory and CPU usage during large input tests	Optimize algorithms, add resource limits
Statistical properties fail consistently	Mathematical error in calculations	Compare results with external statistical tools (R, Python)	Verify formulas, check for floating-point precision issues

The property-based testing approach ensures your feature flag system maintains its mathematical guarantees under the full range of possible inputs, giving confidence that the system will behave correctly in production environments with diverse user bases and complex flag configurations.

## Debugging Guide

**Milestone(s):** This section applies to all three milestones by providing systematic debugging approaches for common issues encountered while implementing the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3).

Think of debugging feature flags as being a detective investigating a mystery. Every incorrect flag evaluation or missed update is a clue pointing to deeper issues in the system. Unlike traditional debugging where you might see immediate crashes, feature flag bugs are often subtle - users getting the wrong variant, rules not evaluating correctly, or updates arriving out of order. These silent failures can impact user experience without triggering obvious error conditions, making systematic debugging approaches essential.

The complexity of debugging feature flag systems stems from their distributed nature and the multiple layers of caching, networking, and evaluation logic. A single flag evaluation request touches the cache layer, rule evaluation engine, consistent hashing algorithm, and potentially real-time update mechanisms. When

something goes wrong, the root cause could be anywhere in this chain, requiring structured approaches to isolate and identify the problem.

This debugging guide provides systematic methodologies for diagnosing and resolving the most common issues encountered when implementing feature flag systems. Rather than exhaustive troubleshooting for every possible scenario, we focus on the patterns of problems that consistently trip up developers building these systems for the first time.

## Flag Evaluation Issues

Flag evaluation problems manifest as users receiving unexpected variants, rules not behaving as configured, or inconsistent assignments across multiple evaluations. These issues often stem from incorrect rule precedence, flawed consistent hashing implementation, or mishandled user context processing.

**Mental Model: CSI Investigation** Think of debugging flag evaluation like a crime scene investigation. Each evaluation leaves "evidence" in the form of logs, metrics, and cached states. The key is systematically examining this evidence to reconstruct what happened during the evaluation process. Just as a detective follows the timeline of events, you need to trace the evaluation flow from user context input through rule processing to final variant assignment.

The `EvaluationResult` structure includes a `Reason` field specifically designed for debugging - it should contain enough information to understand why a particular variant was chosen. When implementing the evaluation engine, ensure every code path that sets a variant also sets a descriptive reason explaining the decision.

## Rule Evaluation Problems

Rule evaluation issues typically occur when targeting rules don't behave as expected, often due to incorrect condition logic, operator precedence problems, or context attribute handling errors.

Symptom	Likely Root Cause	Investigation Steps	Resolution
Rule never matches despite correct user attributes	Condition operator logic error or AND/OR precedence	Check <code>EvaluationResult.Reason</code> for rule evaluation details, verify condition evaluation order	Implement proper boolean expression parsing with parentheses support
Rule matches wrong users	Context attribute type mismatch or missing null checks	Log user context before evaluation, verify attribute types match condition expectations	Add type validation and safe type conversion in condition evaluation
Inconsistent rule behavior across evaluations	Race condition in rule evaluation or shared state mutation	Test with concurrent evaluations, check for global variable modifications	Ensure rule evaluation is stateless and thread-safe
Rules evaluate in wrong order	Incorrect rule precedence implementation	Verify rule sorting logic, check for stable sort requirements	Implement explicit rule priority with tie-breaking mechanisms
Complex nested conditions fail	Boolean logic parsing errors or stack overflow	Test with progressively simpler conditions, check recursion depth	Implement iterative condition parsing or increase recursion limits

**⚠ Pitfall: Forgetting Operator Precedence** A common mistake is implementing condition evaluation without proper precedence rules. For example, `userAge > 18 AND userCountry = "US" OR userCountry = "CA"` should evaluate as `(userAge > 18 AND userCountry = "US") OR userCountry = "CA"`, but naive left-to-right evaluation produces different results. Always implement proper boolean expression parsing with AND taking precedence over OR, or require explicit parentheses for complex conditions.

The `evaluateRuleConditions` method should handle operator precedence correctly:

**Rule Evaluation Algorithm:**

1. Parse conditions into expression tree respecting precedence
2. Evaluate leaf conditions against user context
3. Apply boolean operators bottom-up through the tree
4. Return final boolean result with evaluation path for debugging

When debugging rule evaluation, examine the `UserContext.Attributes` map to ensure all required attributes are present and have the expected types. Missing attributes should be handled gracefully with configurable default behaviors rather than causing evaluation failures.

**Design Insight:** Rule evaluation debugging becomes much easier when each condition evaluation is logged with its inputs, operator, and result. This creates an audit trail showing exactly how complex boolean expressions were resolved.

## Consistent Hashing Assignment Problems

Consistent hashing issues manifest as users receiving different variants across evaluations, percentage rollouts not matching expected distributions, or assignment instability when flag configurations change.

Symptom	Likely Root Cause	Investigation Steps	Resolution
User gets different variants on repeated evaluations	Non-deterministic hashing or salt inconsistency	Test same user/flag combination multiple times, verify hash input construction	Use stable hash algorithm with consistent salt generation
Percentage rollout doesn't match configured allocation	Bucket calculation error or range assignment logic	Analyze actual assignment distribution, check bucket boundary calculations	Verify bucket ranges sum to 100% and boundaries are exclusive
Assignment changes when unrelated flags modified	Using unstable hash inputs or global state in bucket calculation	Test assignment stability when other flags change, verify hash input dependencies	Ensure hash inputs only include user ID and flag key
Some users never see certain variants	Biased hash function or incorrect bucket range mapping	Check hash output distribution, verify variant allocation math	Use cryptographic hash function and validate bucket range calculations
A/B test assignments skewed heavily toward one variant	Incorrect weight interpretation or cumulative percentage error	Compare intended vs actual allocation percentages, check weight summation	Implement proper cumulative percentage calculation for variant selection

**⚠ Pitfall: Using Unstable Hash Inputs** Including volatile data like timestamps, session IDs, or request IDs in the hash calculation causes users to flip between variants unpredictably. The hash input should only include stable identifiers like `UserID` and `FlagKey`. Even adding seemingly stable data like user email can cause problems if users change their email addresses.

The `calculateUserBucket` function should use a deterministic approach:

### Bucket Calculation Algorithm:

1. Construct hash input from stable user ID and flag key only
2. Apply cryptographic hash function (SHA-256 or similar)
3. Convert hash output to integer in range [0, 99]
4. Map integer to variant based on cumulative weight ranges
5. Log bucket value and selected variant for debugging

Debugging consistent hashing requires examining the actual bucket values assigned to users. Implement logging that shows the hash input, computed bucket, and variant assignment reasoning. This data helps identify patterns in assignment problems and validates that the hashing algorithm produces the expected distribution.

### Architecture Decision: Hash Algorithm Selection

- **Context:** Need consistent user assignment to variants across evaluations
- **Options:** CRC32 checksum, MD5 hash, SHA-256 hash
- **Decision:** SHA-256 with user ID + flag key input
- **Rationale:** Cryptographic hash ensures uniform distribution, flag key inclusion prevents correlation between different flags
- **Consequences:** Higher computation cost but better assignment quality and debuggability

### Context and Segmentation Issues

User context and segment membership problems cause targeting rules to fail when they should match, or match when they shouldn't. These issues often arise from context validation problems, segment calculation errors, or attribute type mismatches.

Symptom	Likely Root Cause	Investigation Steps	Resolution
Segment targeting doesn't work	Segment membership calculation error or stale segment data	Verify segment membership computation, check segment cache freshness	Implement real-time segment recalculation or cache invalidation
Attribute-based targeting fails	Missing attributes in context or type conversion errors	Log complete user context, verify attribute presence and types	Add context validation with helpful error messages
Geographic targeting inconsistent	IP geolocation service issues or timezone problems	Test with known IP addresses, verify geolocation data accuracy	Implement geolocation service fallbacks and result caching
Demographic targeting unreliable	Outdated user profile data or attribute synchronization lag	Check user profile data freshness, verify attribute update mechanisms	Add user profile versioning and change tracking
Multi-attribute rules fail unexpectedly	Context parsing errors or attribute name mismatches	Compare expected vs actual context structure, check attribute naming consistency	Standardize context schema validation and attribute naming

**⚠ Pitfall: Assuming Attribute Availability** Rules often fail because they assume certain user attributes will always be present. For example, a rule checking `userAge > 21` fails if the `UserContext` doesn't contain

an age attribute. Always implement graceful handling for missing attributes with configurable default behaviors.

The `ValidateUserContext` function should verify that all required attributes for active targeting rules are present and properly typed:

```
Context Validation Algorithm:  
1. Extract all attributes referenced by active targeting rules  
2. Verify each required attribute exists in user context  
3. Validate attribute types match rule expectations  
4. Check segment membership data is current and complete  
5. Log any validation warnings or missing data
```

Debugging context issues requires comprehensive logging of the user context data alongside evaluation results. Include the complete `UserContext.Attributes` map and `Segments` list in debug logs to verify that targeting rules have access to the expected data.

## Real-time Update Issues

Real-time update problems manifest as clients not receiving flag changes, receiving updates out of order, or experiencing connection instability. These issues often stem from connection management problems, event ordering bugs, or cache synchronization failures.

**Mental Model: Television Broadcasting** Think of real-time updates like a television broadcast system. The server acts as a broadcast tower sending the same signal to many receivers (client SDKs). When clients don't receive updates, it could be due to poor signal quality (network issues), wrong channel tuning (connection configuration), or interference (caching problems). Just as TV systems have fallback mechanisms like cable when over-the-air fails, flag update systems need robust fallback hierarchies.

The key insight for debugging real-time updates is understanding the distinction between connection-level problems (can't establish or maintain the stream) versus data-level problems (receiving wrong or stale data). These require different diagnostic approaches and solutions.

## Connection Management and Reliability

Connection problems typically involve clients failing to establish SSE connections, frequent disconnections, or exponential backoff not working correctly.

Symptom	Likely Root Cause	Investigation Steps	Resolution
Clients can't establish SSE connection	CORS configuration, proxy blocking SSE, or endpoint not found	Check browser developer tools network tab, verify SSE endpoint accessibility	Configure proper CORS headers and proxy SSE support
Frequent connection drops	Keep-alive timeout mismatch or network infrastructure issues	Monitor connection duration patterns, check load balancer timeout settings	Implement heartbeat mechanism with configurable intervals
Exponential backoff not working	Incorrect backoff calculation or missing jitter implementation	Log actual retry delays, verify backoff formula includes randomization	Add jitter to prevent thundering herd on reconnection
Clients stuck in reconnecting state	Server rejecting connections or client retry logic bug	Check server logs for connection rejections, verify retry state transitions	Implement connection state debugging and maximum retry limits
SSE events not reaching clients	Server not flushing events or client event parsing errors	Verify server-side flush calls, check client event handler registration	Ensure proper SSE message formatting and flushing

**⚠ Pitfall: Forgetting SSE Connection Flushing** Server-Sent Events require explicit flushing after writing each event, or messages may buffer indefinitely. The `SSEServer.BroadcastFlagUpdate` method must call `Flush()` on each client's response writer after sending the event data. Without flushing, clients appear connected but never receive updates.

The connection management debugging process should examine both client and server state:

**Connection Debugging Algorithm:**

1. Verify SSE endpoint responds with correct Content-Type and headers
2. Check client connection state transitions in logs
3. Examine server-side client registry for connected clients
4. Test connection with curl or browser dev tools
5. Monitor network layer for connection resets or timeouts

Implementing comprehensive connection state logging helps diagnose problems. The `ConnectionState` should be logged whenever it changes, along with the triggering event and any error details.

**Design Insight:** Connection problems often cascade - a single network hiccup can cause mass reconnections if exponential backoff isn't implemented correctly. Always include jitter in backoff calculations to prevent thundering herd scenarios.

## Event Ordering and Delivery

Event ordering problems cause clients to process flag updates out of sequence, leading to inconsistent state or missed updates. These issues typically stem from concurrent update broadcasting or improper sequence number handling.

Symptom	Likely Root Cause	Investigation Steps	Resolution
Clients receive events out of order	Concurrent broadcasting without sequence numbers or race conditions	Check event sequence numbers in client logs, verify server broadcasting is serialized	Implement monotonic sequence numbers and client-side ordering
Some flag updates never arrive at clients	Event broadcasting failure or client disconnection during update	Compare server broadcast logs with client received events, check connection status during updates	Add event acknowledgment mechanism and retry logic
Clients process duplicate events	Reconnection replay overlap or at-least-once delivery without deduplication	Monitor for duplicate sequence numbers, check replay boundary calculation	Implement client-side deduplication using sequence numbers
Flag updates arrive but aren't applied	Client update processing errors or cache update failures	Check client-side update processing logs, verify cache invalidation logic	Add update processing error handling and retry mechanisms
Bulk flag updates overwhelm clients	Large payload size or high update frequency without throttling	Monitor event payload sizes and frequency, check client processing performance	Implement update batching and client-side throttling

**⚠ Pitfall: Race Conditions in Event Broadcasting** Broadcasting flag updates concurrently can cause events to arrive out of order at clients. If multiple flag changes occur simultaneously, clients might receive the second change before the first, resulting in inconsistent state. Always serialize flag update broadcasting or use sequence numbers to enable client-side ordering.

The event ordering debugging approach should trace events from creation through delivery:

### Event Ordering Debugging Algorithm:

1. Generate unique sequence numbers for each flag update event
2. Log event creation with sequence number and timestamp
3. Trace event broadcasting to all connected clients
4. Monitor client event reception with sequence verification
5. Check for gaps or duplicates in client event sequences

Debugging event delivery requires correlation between server broadcast logs and client reception logs.

Implement request IDs or trace IDs that span from flag update through client processing to enable end-to-end

tracking.

## Cache Synchronization Problems

Cache synchronization issues cause clients to serve stale flag data despite receiving updates, or fail to invalidate cached values when flags change. These problems often involve cache layer bugs or update processing failures.

Symptom	Likely Root Cause	Investigation Steps	Resolution
Clients serve stale flag values after updates	Cache invalidation not triggered or cache hierarchy inconsistency	Check cache invalidation logs, verify update event processing triggers cache refresh	Ensure update handlers call cache invalidation for affected flags
Cache updates fail silently	Storage errors or validation failures during cache write	Check cache operation error logs, verify storage layer health	Add cache operation error handling and fallback mechanisms
Inconsistent cache state across instances	Race conditions in cache updates or distributed cache synchronization issues	Compare cache contents across client instances, check update ordering	Implement cache versioning and conflict resolution
Memory cache grows unbounded	Missing cache eviction or TTL not enforced	Monitor cache size and memory usage, verify eviction policies	Implement proper cache size limits and LRU eviction
Local storage cache corruption	Disk write errors or concurrent access without locking	Check local storage integrity, verify file locking mechanisms	Add local storage validation and repair mechanisms

**⚠ Pitfall: Ignoring Cache Update Failures** When real-time flag updates fail to update the local cache, clients continue serving stale data without indication of the problem. The `ApplyFlagUpdate` method should handle cache update errors gracefully and potentially trigger a full cache refresh if individual updates fail consistently.

Cache synchronization debugging requires examining the cache hierarchy and update propagation:

### Cache Synchronization Debugging Algorithm:

1. Check memory cache state before and after update events
2. Verify local storage persistence of cache changes
3. Compare cache timestamps with server flag modification times
4. Test cache invalidation triggers from update events
5. Validate cache consistency across multiple SDK instances

Effective cache debugging requires visibility into cache hit rates, invalidation events, and synchronization status. The `CacheStats` structure should be monitored to identify cache performance issues and

synchronization problems.

## Implementation Guidance

This section provides concrete implementation approaches for building debugging capabilities into the feature flag system, focusing on observability, error tracking, and systematic problem resolution.

## Technology Recommendations

Component	Simple Option	Advanced Option
Error Tracking	Log to stdout with structured format	Distributed tracing with OpenTelemetry
Metrics Collection	In-memory counters with periodic dumps	Prometheus metrics with Grafana dashboards
Debug Logging	Standard library logger with levels	Structured logging with correlation IDs
Health Monitoring	HTTP endpoint returning component status	Continuous health checks with alerting
Performance Profiling	Built-in Go profiler endpoints	Continuous profiling with pprof integration

## Recommended File Structure

```
internal/debug/
├── instrumentation.go      ← Debug logging and metrics
├── health_check.go        ← Component health monitoring
├── error_classification.go ← Error type classification
├── evaluation_tracer.go   ← Flag evaluation debugging
├── connection_monitor.go  ← Real-time connection debugging
├── cache_inspector.go    ← Cache state examination
└── debug_test.go          ← Debug functionality tests
```

## Infrastructure Starter Code

Complete error classification system for categorizing flag system errors:

```
package debug
```

```
import (
    "context"
    "fmt"
    "sync"
    "time"
)
```

```
// FlagError represents a categorized error in the flag system
```

```
type FlagError struct {
```

Type	ErrorType	`json:"type"`
FlagKey	string	`json:"flag_key,omitempty"`
Message	string	`json:"message"`
Cause	error	`json:"cause,omitempty"`
Context	map[string]interface{}	`json:"context,omitempty"`
Timestamp	time.Time	`json:"timestamp"`
Recoverable	bool	`json:"recoverable"`

```
}
```

```
type ErrorType string
```

```
const (
```

ErrortypeValidation	ErrorType = "validation"
ErrortypeNetwork	ErrorType = "network"
ErrortypeStorage	ErrorType = "storage"
ErrortypeEvaluation	ErrorType = "evaluation"
ErrortypeCircuit	ErrorType = "circuit"
ErrortypeDependency	ErrorType = "dependency"

GO

```
    ErrorTypeResource    Errortype = "resource"

}

// Error implements the error interface

func (e *FlagError) Error() string {

    if e.FlagKey != "" {

        return fmt.Sprintf("[%s:%s] %s", e.Type, e.FlagKey, e.Message)

    }

    return fmt.Sprintf("[%s] %s", e.Type, e.Message)

}

// Unwrap returns the underlying error for error chain compatibility

func (e *FlagError) Unwrap() error {

    return e.Cause

}

// NewFlagError creates a new categorized flag error

func NewFlagError(errorType Errortype, flagKey, message string, cause error) *FlagError {

    return &FlagError{

        Type:          errorType,

        FlagKey:       flagKey,

        Message:       message,

        Cause:         cause,

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

        Timestamp:     time.Now(),

        Recoverable:   isRecoverable(errorType),

    }

}
```

```
// AddContext adds debugging context to the error

func (e *FlagError) AddContext(key string, value interface{}) *FlagError {
    if e.Context == nil {
        e.Context = make(map[string]interface{})
    }
    e.Context[key] = value
    return e
}

// isRecoverable determines if an error type typically allows recovery

func isRecoverable(errorType ErrorType) bool {
    switch errorType {
        case ErrorTypeNetwork, ErrorTypeResource, ErrorTypeCircuit:
            return true
        case ErrorTypeValidation, ErrorTypeDependency:
            return false
        case ErrorTypeStorage, ErrorTypeEvaluation:
            return true // Usually recoverable with retry or fallback
        default:
            return false
    }
}

// CircuitBreaker provides failure detection and recovery coordination

type CircuitBreaker struct {
    name          string
    maxFailures   int
    resetTimeout  time.Duration
}
```

```
state      CircuitState

failures   int

lastFailTime time.Time

successCount int

mu         sync.RWMutex

}

type CircuitState string

const (

    StateClosed  CircuitState = "closed"

    StateOpen    CircuitState = "open"

    StateHalfOpen CircuitState = "half_open"

)

// Allow checks if the circuit breaker permits a request

func (cb *CircuitBreaker) Allow() error {

    cb.mu.Lock()

    defer cb.mu.Unlock()

    switch cb.state {

    case StateClosed:

        return nil

    case StateOpen:

        if time.Since(cb.lastFailTime) > cb.resetTimeout {

            cb.state = StateHalfOpen

            return nil

        }

    }

}
```

```
        return NewFlagError(ErrorTypeCircuit, "",  
            fmt.Sprintf("circuit breaker %s is open", cb.name), nil)  
  
    case StateHalfOpen:  
  
        return nil  
  
    default:  
  
        return NewFlagError(ErrorTypeCircuit, "",  
            fmt.Sprintf("unknown circuit breaker state: %s", cb.state), nil)  
    }  
}  
  
// RecordResult updates circuit breaker state based on operation result  
  
func (cb *CircuitBreaker) RecordResult(err error) {  
  
    cb.mu.Lock()  
  
    defer cb.mu.Unlock()  
  
  
    if err != nil {  
  
        cb.failures++  
  
        cb.lastFailTime = time.Now()  
  
        cb.successCount = 0  
  
  
  
  
        if cb.failures >= cb.maxFailures {  
  
            cb.state = StateOpen  
  
        }  
    } else {  
  
        cb.failures = 0  
  
        cb.successCount++  
    }  
}
```

```
    if cb.state == StateHalfOpen && cb.successCount >= 3 {

        cb.state = StateClosed

    }

}

}

// State returns the current circuit breaker state

func (cb *CircuitBreaker) State() CircuitState {

    cb.mu.RLock()

    defer cb.mu.RUnlock()

    return cb.state

}

// NewCircuitBreaker creates a circuit breaker with specified parameters

func NewCircuitBreaker(name string, maxFailures int, resetTimeout time.Duration)
*CircuitBreaker {

    return &CircuitBreaker{

        name:           name,

        maxFailures:   maxFailures,

        resetTimeout:  resetTimeout,

        state:         StateClosed,

    }

}
```

Complete health monitoring system for tracking component status:

GO

```
// ComponentHealth tracks the health status of system components

type ComponentHealth struct {

    Status      HealthStatus `json:"status"`

    LastCheck   time.Time   `json:"last_check"`

    LastError   string     `json:"last_error,omitempty"`

    CheckCount int64      `json:"check_count"`

    ErrorCount int64      `json:"error_count"`

}

type HealthStatus string

const (
    HealthStatusHealthy  HealthStatus = "healthy"
    HealthStatusDegraded HealthStatus = "degraded"
    HealthStatusUnhealthy HealthStatus = "unhealthy"
)

// HealthComponent defines interface for health-checkable components

type HealthComponent interface {

    Name() string

    Check(ctx context.Context) error

    Recover(ctx context.Context) error

}

// HealthChecker monitors component health and coordinates recovery

type HealthChecker struct {

    components map[string]HealthComponent

    status     map[string]ComponentHealth

    mu         sync.RWMutex
}
```

```
    interval    time.Duration
    stopCh      chan struct{}`

}

// StartMonitoring begins the health monitoring loop

func (hc *HealthChecker) StartMonitoring(ctx context.Context) {
    ticker := time.NewTicker(hc.interval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            hc.checkAllComponents(ctx)

        case <-hc.stopCh:
            return

        case <-ctx.Done():
            return
        }
    }
}

// checkAllComponents performs health checks on all registered components

func (hc *HealthChecker) checkAllComponents(ctx context.Context) {
    hc.mu.Lock()
    defer hc.mu.Unlock()

    for name, component := range hc.components {
        health := hc.status[name]

```

```
    health.CheckCount++

    health.LastCheck = time.Now()

    if err := component.Check(ctx); err != nil {

        health.ErrorCount++

        health.LastError = err.Error()

        // Determine health status based on error rate

        errorRate := float64(health.ErrorCount) / float64(health.CheckCount)

        if errorRate > 0.5 {

            health.Status = HealthStatusUnhealthy

            // Attempt recovery for unhealthy components

            if recoverErr := component.Recover(ctx); recoverErr != nil {

                health.LastError = fmt.Sprintf("recovery failed: %v", recoverErr)

            }

        } else if errorRate > 0.2 {

            health.Status = HealthStatusDegraded

        }

    } else {

        health.Status = HealthStatusHealthy

        health.LastError = ""

    }

    hc.status[name] = health

}

}
```

## **Core Logic Skeleton Code**

Debug-enhanced flag evaluation that provides comprehensive debugging information:

GO

```
// EvaluateFlag performs flag evaluation with comprehensive debugging support

func (e *Evaluator) EvaluateFlag(flagKey FlagKey, userContext UserContext)
(*EvaluationResult, error) {

    // TODO 1: Start evaluation trace with unique trace ID for debugging

    // TODO 2: Validate user context and flag key, return descriptive errors for invalid
input

    // TODO 3: Retrieve flag definition with cache hit/miss logging

    // TODO 4: Check flag prerequisites and dependency chain with cycle detection

    // TODO 5: Evaluate targeting rules with detailed condition logging

    // TODO 6: Perform consistent hashing with bucket calculation details

    // TODO 7: Select variant based on percentage allocation with weight verification

    // TODO 8: Record evaluation metrics and exposure events

    // TODO 9: Return evaluation result with complete reasoning chain

    // Hint: Use structured logging with trace IDs to correlate debugging information

    // Hint: Include evaluation timing metrics to identify performance bottlenecks

    // Hint: Log cache hit/miss ratio to optimize caching strategy

}

// PropagateChange handles flag update distribution with connection tracking

func (u *UpdateService) PropagateChange(flagKey FlagKey, changeType string, payload
interface{}) error {

    // TODO 1: Validate change payload and generate unique event ID

    // TODO 2: Create flag update event with sequence number for ordering

    // TODO 3: Broadcast to all connected clients with failure tracking

    // TODO 4: Handle broadcasting failures with retry mechanism

    // TODO 5: Update propagation metrics and connection health status

    // TODO 6: Log successful and failed deliveries for debugging

    // Hint: Track per-client delivery status to identify problematic connections

    // Hint: Implement exponential backoff for failed broadcast retries
```

```

    // Hint: Monitor propagation latency to detect performance issues

}

// ApplyFlagUpdate processes incoming flag changes with cache synchronization

func (c *Client) ApplyFlagUpdate(event FlagUpdateEvent) error {

    // TODO 1: Validate event sequence number and detect out-of-order delivery

    // TODO 2: Parse flag update payload with comprehensive error handling

    // TODO 3: Update memory cache with version tracking

    // TODO 4: Persist to local storage with integrity verification

    // TODO 5: Invalidate dependent caches and derived data

    // TODO 6: Record update processing metrics and cache statistics

    // Hint: Implement sequence number gaps detection for missed updates

    // Hint: Add cache consistency verification after updates

    // Hint: Log cache invalidation cascades for debugging dependency issues

}

```

## Language-Specific Debugging Hints

### Go-Specific Debugging Techniques:

- 1. Use `context.Context` for Request Tracing:** Pass context through all evaluation calls to maintain trace IDs and enable request correlation across logs.
- 2. Leverage `sync.RWMutex` for Safe Concurrent Access:** Use read locks for flag evaluation and write locks for cache updates to prevent race conditions.
- 3. Implement `fmt.Stringer` for Complex Types:** Add String() methods to `UserContext`, `EvaluationResult`, and other debug types for readable log output.
- 4. Use `testing.T.Logf()` for Test Debugging:** In unit tests, use `t.Logf()` to output debugging information only when tests fail.
- 5. Add `//go:build debug` Tags:** Create debug-only code paths that provide extra logging and validation in development builds.

## Milestone Checkpoints

### Milestone 1 Validation - Flag Evaluation Engine:

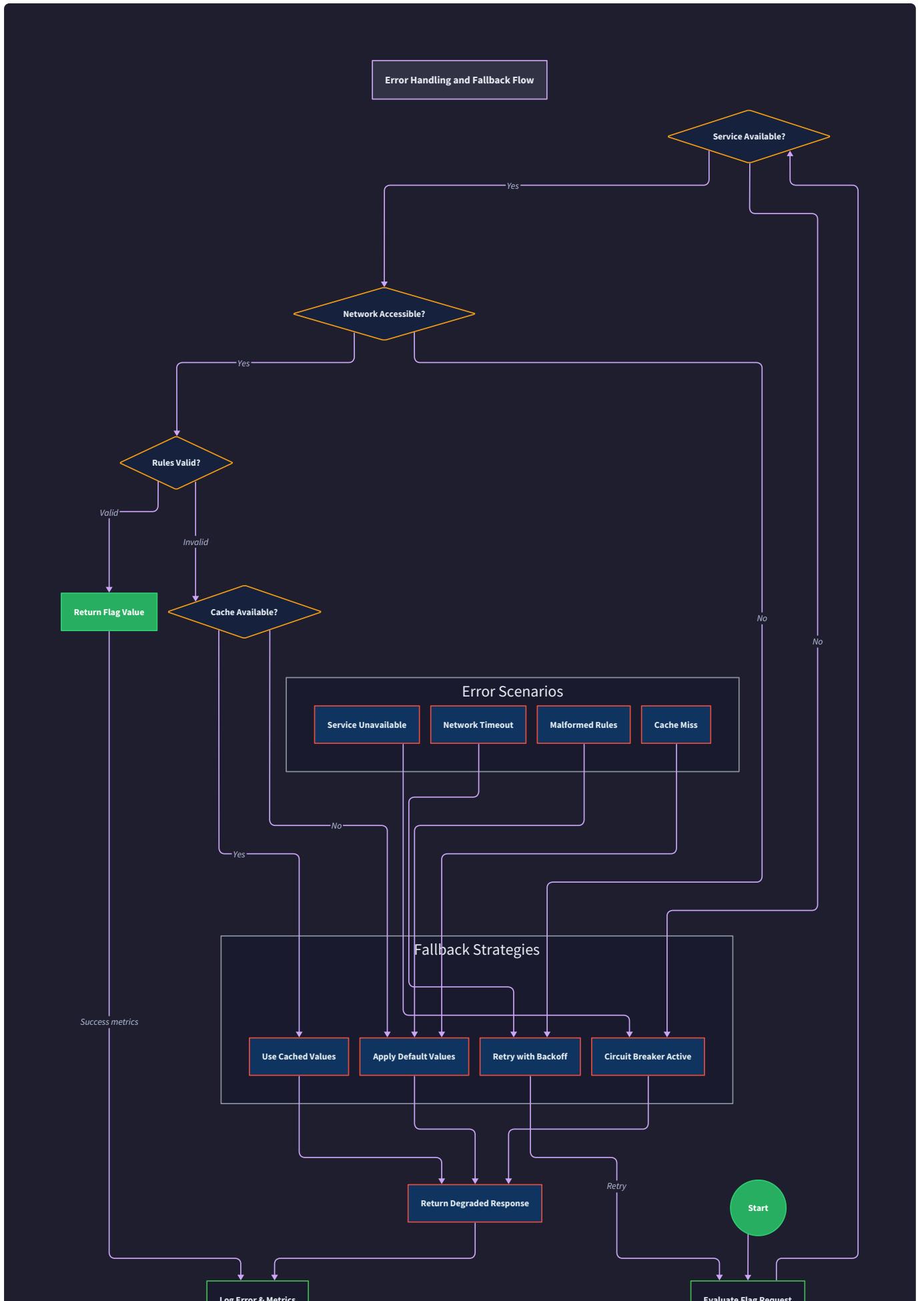
- Run `go test -v ./internal/evaluation/...` - all tests should pass with detailed evaluation logs
- Test flag evaluation with debug logging enabled - should see rule-by-rule evaluation traces
- Verify consistent hashing stability - same user/flag combination should always return same variant
- Expected output: Detailed evaluation logs showing rule matching, bucket calculation, and variant selection reasoning

#### **Milestone 2 Validation - Real-time Flag Updates:**

- Run SSE connection test with `curl -N -H "Accept: text/event-stream"`  
`http://localhost:8080/flags/stream`
- Monitor connection logs during network interruption and recovery
- Verify event ordering with sequence numbers in client logs
- Expected output: SSE events received in correct order with reconnection handling and no duplicate processing

#### **Milestone 3 Validation - Flag Analytics & Experiments:**

- Generate test experiment data and verify statistical calculations
- Check exposure event logging for completeness and accuracy
- Validate significance testing with known statistical outcomes
- Expected output: Accurate statistical significance calculations and comprehensive exposure tracking logs



The error handling flow diagram illustrates the decision points and fallback mechanisms that debugging tools must account for when diagnosing system issues. Understanding these flows helps identify where problems originate and which debugging approaches will be most effective.

## Future Extensions

**Milestone(s):** This section provides forward-looking enhancements that build upon the foundation established by all three milestones, extending the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3) with advanced capabilities for production-scale deployments.

Think of the feature flag system we've built as a **city's traffic infrastructure**. We've constructed the core roads (evaluation engine), traffic lights (real-time updates), and monitoring systems (analytics). Now we're ready to expand our city with highways (multi-environment support), advanced traffic management (sophisticated targeting), and interconnected transportation networks (flag dependencies). Just as a growing city needs these advanced systems to handle increasing complexity and scale, our feature flag system can evolve to support more sophisticated use cases while maintaining the solid foundation we've established.

The design we've implemented follows an **evaluation-first architecture** that prioritizes performance and consistency in the core flag evaluation path. This architectural foundation naturally accommodates the extensions described in this section without requiring fundamental changes to the evaluation engine, data model, or real-time update mechanisms. Each extension builds upon existing components while introducing new capabilities that enhance the system's flexibility and power.

### Advanced Targeting and Segmentation

Modern feature flag systems require increasingly sophisticated targeting capabilities beyond basic user attributes and percentage rollouts. Advanced targeting enables product teams to create nuanced user experiences by combining multiple data sources, temporal conditions, and contextual information.

**Dynamic Segment Computation** represents a significant evolution from the static segment membership we implemented in our user context model. Instead of pre-computed segment assignments stored in the `UserContext`, dynamic segments evaluate membership criteria in real-time during flag evaluation. This approach enables targeting based on live user behavior, current application state, or external system data.

The dynamic segmentation system would extend our existing `TargetingRule` structure with new condition types that support external data lookups and computed values. For example, a segment might target "users who made a purchase in the last 7 days" by querying an external analytics service during flag evaluation.

While this introduces latency considerations, the evaluation engine's caching mechanisms and circuit breaker patterns provide natural resilience.

### Decision: Real-time vs. Pre-computed Segmentation

- **Context:** Advanced targeting requires access to frequently changing user data that may be expensive to pre-compute
- **Options Considered:**
  - Pre-compute all segments periodically and store in user context
  - Evaluate segments dynamically during flag evaluation with caching
  - Hybrid approach with both static and dynamic segments
- **Decision:** Hybrid approach supporting both static segments (in UserContext) and dynamic segments (evaluated on-demand)
- **Rationale:** Maintains backward compatibility while enabling real-time targeting; caching and circuit breakers mitigate performance concerns
- **Consequences:** Increased evaluation latency for dynamic segments, but greater targeting flexibility and real-time accuracy

Segmentation Type	Evaluation Timing	Data Freshness	Performance Impact	Use Cases
Static Segments	Pre-computed	Batch update intervals	Low latency	Demographic targeting, subscription tiers
Dynamic Segments	Real-time	Always current	Higher latency	Recent behavior, live metrics, external data
Cached Dynamic	First access + TTL refresh	TTL-bounded	Moderate latency	Frequently accessed behavioral segments

**Contextual Targeting** extends the `UserContext` model to include environmental and situational information beyond user attributes. This includes device characteristics, network conditions, application version, geographic location precision, time-based conditions, and integration with external systems. The evaluation engine's consistent hashing mechanism ensures that users with identical contexts receive consistent variant assignments even as context information becomes more detailed.

**Multi-dimensional Rollouts** enhance percentage-based targeting by supporting multiple allocation dimensions simultaneously. Instead of a single percentage rollout, flags could allocate users across orthogonal dimensions like geographic region, device type, and user tenure. This enables sophisticated rollout strategies like "10% of mobile users in North America" while maintaining the deterministic assignment properties of our consistent hashing implementation.

## Flag Dependencies and Prerequisites

Complex applications often require coordinated feature rollouts where one feature depends on another being enabled. Flag dependencies create controlled activation sequences and prevent inconsistent feature combinations that could break user experiences.

**Dependency Graph Management** builds upon the cycle detection validation we implemented in the flag evaluation engine. The system would maintain a directed acyclic graph of flag relationships, with each `FlagDefinition` containing a `prerequisites` field listing required flags and their states. The evaluation engine extends its processing to check prerequisite satisfaction before applying targeting rules.

Consider a scenario where a new payment flow (Flag A) depends on updated user authentication (Flag B) and enhanced fraud detection (Flag C). The dependency system ensures Flag A only activates for users who have access to both prerequisites, preventing broken payment experiences.

**Key Design Insight:** Flag dependencies should fail safe by defaulting to the most restrictive state. If any prerequisite fails evaluation, the dependent flag should return its default value rather than attempting partial activation.

Dependency Type	Description	Evaluation Impact	Common Patterns
Hard Prerequisites	Required flags must be enabled	Blocks dependent flag if prerequisites disabled	Infrastructure changes, breaking API updates
Soft Dependencies	Preferred but not required	Logs warnings but allows evaluation	UI enhancements, optional integrations
Mutual Exclusions	Flags cannot be active simultaneously	Disables conflicting flags automatically	A/B test variants, alternative implementations
Version Dependencies	Specific versions of prerequisite flags	Checks version compatibility during evaluation	API versioning, graduated feature rollouts

**Prerequisite Evaluation Order** requires careful consideration to maintain evaluation performance. The evaluation engine processes dependencies in topological order, caching prerequisite results to avoid repeated evaluation within a single request. The `EvaluationResult` extends to include dependency information for debugging complex activation scenarios.

**Circular Dependency Prevention** leverages the validation framework established in our flag management system. The dependency validator performs depth-first traversal during flag updates, rejecting configurations that would create cycles. This validation runs during both flag creation and updates, ensuring the dependency graph remains acyclic.

## Multi-Environment Support

Production feature flag systems must support multiple deployment environments (development, staging, production) with independent configuration management while maintaining operational consistency.

**Environment Isolation** extends the data model with environment-scoped flag definitions. Each `FlagDefinition` includes an environment identifier, and the flag management API enforces environment-based access controls. This isolation prevents accidental cross-environment configuration changes while allowing controlled promotion workflows.

The evaluation engine remains environment-agnostic, receiving flag definitions through the same interfaces established in our architecture. Environment-specific configuration occurs at the data layer, with separate storage namespaces or database schemas for each environment. This design maintains evaluation performance while providing complete configuration isolation.

### Decision: Environment Configuration Strategy

- **Context:** Multiple environments need independent flag configurations with promotion capabilities
- **Options Considered:**
  - Single database with environment field in all records
  - Separate databases/schemas per environment
  - Environment-specific configuration files
- **Decision:** Separate storage namespaces with cross-environment promotion API
- **Rationale:** Complete isolation prevents accidental changes; promotion API enables controlled configuration flow; maintains existing data model
- **Consequences:** Additional operational complexity but maximum safety and flexibility

**Configuration Promotion Workflows** enable controlled advancement of flag configurations through environment stages. The system provides promotion APIs that copy flag definitions between environments while maintaining audit trails and validation checkpoints. This builds upon our flag change tracking mechanisms to provide complete configuration lifecycle management.

**Environment-Specific Analytics** extends the experiment framework to segment results by environment, enabling teams to validate feature performance in staging before production rollouts. The `FlagExposure` model includes environment context, and the significance calculation engine can filter results by environment for isolated analysis.

## Advanced Analytics and Machine Learning Integration

The analytics foundation established in our A/B testing framework naturally extends to support machine learning-driven optimization and predictive analytics capabilities.

**Automated Flag Optimization** uses machine learning models to recommend optimal variant allocations based on observed user behavior and conversion patterns. The system analyzes historical `FlagExposure` data to identify user segments with different variant preferences, automatically suggesting targeting rules that maximize desired outcomes.

This capability builds upon our existing significance calculation framework by extending the `SignificanceCalculator` with predictive models. Instead of only reporting experiment results, the system recommends actions like "increase allocation for mobile users in the high-engagement variant" based on statistical evidence.

**Predictive User Segmentation** applies clustering algorithms to flag exposure and conversion data to identify previously unknown user segments with distinct behavior patterns. These discovered segments become available as targeting criteria in the dynamic segmentation system, creating a feedback loop between analytics insights and flag targeting capabilities.

**Anomaly Detection** monitors flag performance metrics in real-time, automatically alerting teams when conversion rates, error rates, or user engagement metrics deviate significantly from historical patterns. This extends our health monitoring framework to include business-metric monitoring alongside system health checks.

ML Capability	Data Requirements	Algorithmic Approach	Integration Point
Automated Optimization	Flag exposures, conversion events	Multi-armed bandits, Bayesian optimization	Significance calculator extension
Predictive Segmentation	User behavior, attribute data	K-means clustering, collaborative filtering	Dynamic segment engine
Anomaly Detection	Time-series metrics, baseline data	Statistical process control, outlier detection	Health monitoring system
Causal Impact Analysis	Pre/post experiment data	Difference-in-differences, synthetic control	Experiment results framework

## Global Distribution and Edge Computing

Large-scale applications require feature flag evaluation at global scale with minimal latency, necessitating edge computing integration and distributed caching strategies.

**Edge Evaluation Nodes** deploy lightweight versions of the evaluation engine to edge locations worldwide. These nodes maintain cached flag definitions synchronized through our real-time update system, enabling sub-millisecond flag evaluation anywhere in the world. The edge nodes implement the same evaluation algorithms and consistent hashing mechanisms as the central system, ensuring globally consistent user assignments.

The distributed architecture maintains the same API interfaces while adding geographic routing intelligence. Client SDKs automatically discover and connect to the nearest edge node, falling back to central evaluation during edge node failures. This builds upon our existing fallback hierarchy patterns while adding geographic distribution.

**Conflict-Free Replicated Data Types (CRDTs)** provide eventual consistency for flag configurations across distributed edge nodes. Flag updates propagate through the real-time update system while CRDT properties ensure all nodes converge to identical configuration states despite network partitions or update ordering differences.

**Regional Compliance and Data Sovereignty** extends the multi-environment architecture to support regulatory requirements like GDPR data locality. Flag configurations and user context data remain within specified geographic boundaries while maintaining global evaluation consistency through federated architectures.

## Enterprise Integration Capabilities

Production deployments require integration with existing enterprise systems for authentication, authorization, audit logging, and workflow management.

**Single Sign-On (SSO) Integration** extends the flag management API with pluggable authentication providers supporting SAML, OAuth 2.0, and OpenID Connect. The system maintains role-based access controls for flag management while delegating authentication to enterprise identity providers.

**Audit and Compliance Logging** enhances our flag change tracking to meet enterprise audit requirements. Every flag modification, user assignment, and experiment result includes comprehensive audit metadata suitable for compliance reporting. The system supports audit log export in standard formats and integrates with Security Information and Event Management (SIEM) systems.

**Workflow Integration** connects flag lifecycle management with existing development and deployment workflows. This includes integration with source control systems for configuration-as-code, continuous integration pipelines for automated flag validation, and project management tools for feature delivery tracking.

**API Gateway Integration** enables centralized feature flag evaluation within API gateway infrastructure, providing flag-based request routing, rate limiting, and access control. This architectural pattern reduces evaluation latency by embedding flag logic directly in the request processing pipeline.

## Performance Optimization Extensions

As usage scales, the system benefits from advanced performance optimization techniques that maintain the evaluation-first design principles while handling extreme throughput requirements.

**Evaluation Result Caching** extends beyond our current flag definition caching to cache evaluation results for identical user contexts. This optimization particularly benefits scenarios with repeated evaluations for the same users, though it requires careful cache invalidation when flag configurations change.

**Batch Evaluation APIs** support evaluating multiple flags simultaneously for a single user context, reducing network overhead and enabling optimizations like shared rule evaluation across flags. This maintains the deterministic evaluation properties while improving throughput for SDK implementations.

**Asynchronous Evaluation Pipelines** separate flag evaluation from result delivery for use cases where immediate responses aren't required. This enables advanced optimizations like speculative evaluation and result pre-computation based on predicted user behavior patterns.

**Memory-Efficient Data Structures** replace standard hash tables and trees with specialized data structures optimized for flag evaluation workloads. Techniques like compressed tries for rule matching and bloom filters for negative lookups reduce memory usage while maintaining evaluation speed.

## Integration Ecosystem Extensions

The feature flag system becomes more valuable as part of a broader development and operations ecosystem, requiring integration capabilities that extend its utility without compromising core functionality.

**Observability Platform Integration** connects flag evaluation metrics with distributed tracing systems, application performance monitoring, and log aggregation platforms. Every flag evaluation becomes observable within the broader application context, enabling correlation analysis between feature flags and system behavior.

**Feature Management Lifecycle Tools** extend beyond flag evaluation to support the complete feature development lifecycle. This includes feature planning tools that integrate with flag definitions, deployment tracking that correlates flag changes with application releases, and technical debt management that identifies obsolete flags for cleanup.

**Developer Experience Enhancements** improve the day-to-day interaction with the feature flag system through IDE plugins, command-line tools, and local development environment integration. These tools build upon our existing SDK architecture while providing development-time conveniences that encourage proper flag usage.

## Scalability Architecture Enhancements

Supporting massive scale requires architectural enhancements that maintain the system's reliability and performance characteristics while handling orders of magnitude more traffic.

**Horizontal Evaluation Scaling** distributes flag evaluation across multiple server instances using consistent hashing for request distribution. This maintains user assignment consistency while enabling linear scaling of evaluation throughput. The architecture builds upon our existing load balancing and health monitoring capabilities.

**Event Streaming Architecture** replaces direct database interactions with event-driven patterns using systems like Apache Kafka. Flag changes become events in the stream, enabling multiple consumers to process updates for real-time propagation, analytics processing, and audit logging without impacting evaluation performance.

**Polyglot Storage Strategies** optimize data storage by matching storage technologies to access patterns. Flag definitions might live in document stores for flexible schema evolution, while evaluation results use time-series databases optimized for analytics queries. The abstraction layers in our architecture facilitate these storage technology choices without affecting client interfaces.

## Future-Proofing Considerations

The extensions described maintain compatibility with our core architectural decisions while providing natural evolution paths as requirements continue expanding.

**API Versioning Strategy** ensures backward compatibility as new capabilities add to existing interfaces. The REST API design accommodates feature additions through optional parameters and response fields, while SDK interfaces use extensible patterns that don't break existing client code.

**Configuration Schema Evolution** supports adding new flag types, targeting criteria, and evaluation modes without requiring system-wide updates. The validation framework accommodates schema versioning, enabling gradual rollout of new capabilities across distributed deployments.

**Telemetry and Instrumentation** provides comprehensive visibility into system behavior as complexity increases. This builds upon our health monitoring foundation while adding detailed metrics for capacity planning, performance optimization, and operational decision-making.

**Critical Insight:** Each extension maintains the evaluation-first design principle by ensuring core flag evaluation performance doesn't degrade as new capabilities are added. Complex features are implemented as optional enhancements rather than fundamental changes to the evaluation path.

## Implementation Guidance

The extensions described can be implemented incrementally without disrupting existing functionality. The modular architecture established in our original design provides natural extension points for these capabilities.

## Technology Recommendations

Extension Category	Simple Approach	Advanced Approach
Dynamic Segmentation	HTTP calls to external APIs with caching	Event-driven segment computation with stream processing
Edge Distribution	CDN with API caching	Dedicated edge computing platform with custom evaluation nodes
ML Integration	Periodic batch analysis with manual recommendations	Real-time ML inference with automated optimization
Enterprise Integration	Basic RBAC with API keys	Full SSO integration with enterprise workflow systems
Advanced Analytics	Extended PostgreSQL with time-series tables	Dedicated analytics database with specialized query engines

## Extension Implementation Strategy

When implementing these extensions, follow a **capability-driven approach** that adds new functionality without modifying core evaluation logic. Each extension should:

- Maintain Interface Compatibility:** New capabilities extend existing interfaces rather than replacing them, ensuring client SDKs continue working without modification.
- Preserve Performance Characteristics:** Extensions shouldn't degrade evaluation latency or throughput. Performance-impacting features like dynamic segmentation include circuit breakers and caching mechanisms.
- Support Gradual Rollout:** New capabilities can be enabled incrementally, allowing validation in development and staging environments before production deployment.
- Provide Operational Visibility:** Each extension includes monitoring, logging, and health checking capabilities that integrate with existing operational tools.

The architectural foundation we've established provides natural extension points through:

- Plugin interfaces for new evaluation criteria and segment types
- Event streaming capabilities for real-time data integration
- Caching abstractions that support new data sources
- Health monitoring frameworks that accommodate new component types

## Development Priorities

When choosing which extensions to implement first, consider business value and architectural complexity:

**High Value, Low Complexity:** Multi-environment support and basic flag dependencies provide immediate operational benefits with minimal architectural changes.

**High Value, High Complexity:** Advanced targeting and ML-driven optimization deliver significant capabilities but require careful implementation to maintain performance.

**Foundation Building:** Edge distribution and enterprise integration create platforms for future capabilities while providing immediate value for large-scale deployments.

The modular architecture ensures that implementing one extension doesn't preclude others, allowing development teams to prioritize based on immediate business needs while maintaining long-term architectural flexibility.

## Glossary

**Milestone(s):** This section provides comprehensive definitions for terminology used across all three milestones, ensuring consistent understanding of concepts throughout the Flag Evaluation Engine (Milestone 1), Real-time Flag Updates (Milestone 2), and Flag Analytics & Experiments (Milestone 3) implementation.

The feature flag system introduces numerous specialized terms and concepts that span multiple domains including distributed systems, statistical analysis, real-time communication, and software release management. This glossary serves as the authoritative reference for understanding the precise meaning of each term as used within our system's context.

## Core System Concepts

**Feature Flags** are runtime toggles that control feature availability without requiring code deployment. Think of feature flags as smart light switches in a building automation system - they can be controlled remotely, scheduled to activate at specific times, and configured with complex rules about who can access them. Unlike simple boolean toggles, our feature flags support multiple variants, percentage rollouts, and sophisticated targeting rules.

**Air Traffic Control** serves as our mental model for coordinating software releases. Just as air traffic controllers manage multiple aircraft approaching an airport with different priorities, weather conditions, and landing capabilities, feature flags coordinate the rollout of multiple features to different user segments with varying risk tolerances and business requirements. The control tower (flag management system) maintains a real-time view of all active "flights" (feature rollouts) and can adjust their "flight paths" (targeting rules) or "landing sequences" (rollout percentages) based on current conditions.

**Consistent Hashing** is the deterministic algorithm that ensures users receive the same variants across evaluations. This functions like a deterministic seating assignment system - given a user ID and flag key, the

algorithm always assigns the same "seat" (variant) to that user, regardless of when or where the assignment occurs. This prevents users from experiencing jarring transitions between different feature variants.

**Graceful Degradation** describes how the system maintains functionality during partial failures. Like a commercial building's backup power systems, the feature flag system includes multiple fallback layers: memory cache, local storage, and default values. When the primary flag service becomes unavailable, the system automatically switches to increasingly stale but still functional data sources.

**Flag Debt** represents the accumulation of obsolete feature flags in the codebase. Similar to technical debt, flag debt compounds over time as old flags remain in code long after their experiments conclude. Each unused flag increases cognitive load, creates potential security risks through unintended code paths, and complicates system maintenance.

## Evaluation and Targeting

**Evaluation Flow** describes the step-by-step process from client request to variant assignment. The flow begins when a client SDK requests a flag evaluation, continues through rule processing and consistent hashing, and concludes with variant assignment and exposure logging. Each step includes error handling and fallback mechanisms to ensure reliable operation.

**Targeting Rules** are conditions that determine user variant assignment based on user context and environmental factors. These rules function like sophisticated email filters that can examine multiple user attributes, combine conditions with boolean logic, and apply percentage-based allocations. Rules are evaluated in priority order until a match is found.

**Percentage Rollout** enables gradual feature exposure using allocation buckets. Think of this as a controlled water release from a dam - you can gradually open the gates (increase percentage) to allow more water (users) through while monitoring downstream effects (metrics). The bucket ranges are calculated using consistent hashing to ensure stable user assignment.

**User Context** provides comprehensive user information for targeting decisions. This includes stable identifiers, demographic attributes, behavioral data, and environmental context like geographic location or device type. The context serves as the input to targeting rule evaluation and consistent hashing calculations.

**Segment Membership** represents pre-computed group classifications for efficient targeting. Rather than evaluating complex conditions for each flag request, segments allow for efficient "user is in premium\_customers segment" checks. Segments can be static (manually managed) or dynamic (automatically computed based on user attributes).

**Dynamic Segmentation** performs real-time evaluation of user segment membership during flag evaluation. While pre-computed segments optimize performance for stable user groups, dynamic segmentation enables targeting based on real-time context like current session behavior or time-sensitive conditions.

## Data Structures and Types

Term	Description	Key Fields
FlagKey	String identifier uniquely identifying a feature flag across the system	-
UserID	Stable identifier for a user that remains consistent across sessions and evaluations	-
Variant	Represents a possible flag outcome with its configuration and traffic allocation	Key, Value, Weight
UserContext	Comprehensive user information used for targeting and segmentation	UserID, Attributes, Segments
EvaluationResult	Complete outcome of flag evaluation including value, reason, and audit trail	FlagKey, Value, Variant, Reason, Source
FlagDefinition	Complete specification of a feature flag including all rules and variants	FlagKey, Name, Description, Variants, DefaultVariant, Rules, Dependencies
TargetingRule	Individual targeting condition with associated variant allocation	Conditions, Operator, Allocation
AttributeValue	Type-safe wrapper for user attributes supporting multiple data types	-

## Real-time Communication

**Server-Sent Events** provide unidirectional streaming protocol for real-time updates from server to client. SSE functions like a radio broadcast where the server (radio station) transmits updates to all listening clients (radios). Unlike bidirectional protocols like WebSocket, SSE maintains simplicity by focusing solely on server-to-client communication with built-in reconnection handling.

**Cache Invalidation** describes updating cached flag definitions when configurations change. Like a content delivery network pushing updated website assets, the flag system must coordinate cache updates across all client instances to prevent serving stale flag configurations.

**Connection Management** handles the establishment, maintenance, and recovery of streaming connections between clients and the update service. This includes implementing exponential backoff for reconnection attempts, maintaining connection state, and gracefully handling network partitions.

**Thundering Herd** occurs when mass simultaneous connection attempts overwhelm the service. This typically happens when many clients lose connection simultaneously (due to server restart or network issue) and all attempt to reconnect immediately. The system prevents this through randomized backoff delays.

**Exponential Backoff** implements progressively longer retry delays with randomization to prevent overwhelming failed services. Starting with a short delay (e.g., 100ms), each retry doubles the delay (200ms, 400ms, 800ms) while adding jitter to prevent synchronized retries across multiple clients.

## Statistical and Experimental Concepts

**Flag Analytics** encompasses comprehensive tracking and analysis of feature flag usage and performance. This includes exposure tracking, conversion attribution, performance monitoring, and business impact measurement. Analytics enable data-driven decisions about feature rollouts and business value assessment.

**Exposure Events** are recorded instances of users encountering specific flag variants. Each exposure captures the complete context of the evaluation including user attributes, variant assigned, timestamp, and evaluation reason. These events form the foundation for all subsequent analysis.

**A/B Testing Framework** provides systematic approach to controlled feature experimentation. The framework manages experiment lifecycle from setup through analysis, ensuring proper randomization, statistical power, and unbiased result interpretation.

**Statistical Significance** represents mathematical confidence that observed differences between variants are not due to random chance. The system uses established statistical tests (chi-square for categorical outcomes, t-tests for continuous metrics) with appropriate corrections for multiple comparisons.

**Sample Ratio Mismatch** occurs when the actual variant allocation ratios deviate significantly from intended ratios. This can indicate implementation bugs, biased assignment logic, or data collection issues. The system continuously monitors for SRM and alerts when detected.

**Intent-to-treat Analysis** analyzes all assigned users regardless of actual exposure to the feature. This approach prevents bias from users who opt out or experience technical issues that prevent feature loading, providing a more conservative but unbiased estimate of feature impact.

**Sequential Testing** enables valid interim analysis of experiments without inflating false positive rates. Unlike traditional fixed-sample tests, sequential methods allow peeking at results during the experiment using appropriate statistical boundaries.

**Power Analysis** calculates the sample size needed for reliable effect detection given expected effect size, significance level, and desired statistical power. This prevents underpowered experiments that fail to detect meaningful effects.

**Survivorship Bias** represents analytical error from excluding users who dropped out during the experiment. For example, analyzing only users who completed a multi-step flow ignores the impact on users who abandoned earlier steps.

## System Architecture and Operations

**Evaluation-first Design** prioritizes flag evaluation performance above all other system considerations. Like designing a race car where aerodynamics takes precedence over passenger comfort, our architecture

optimizes for sub-millisecond evaluation latency even when this complicates other system aspects.

**Fallback Hierarchy** defines the ordered sequence of data sources used when primary sources become unavailable. The hierarchy typically flows from real-time service → memory cache → local storage → default values, with each layer providing increasingly stale but still functional flag data.

**Circuit Breaker** prevents cascade failures by temporarily stopping requests to failing services. When a dependent service experiences high error rates, the circuit breaker opens, immediately returning cached results instead of continuing to send requests to the failing service.

**Load Shedding** involves selectively dropping requests during high traffic to maintain service availability for critical operations. The system may prioritize flag evaluations over analytics ingestion or defer less critical operations when under extreme load.

**Health Monitoring** systematically checks component operational status to enable proactive issue detection and automated recovery. Each system component reports health metrics including response times, error rates, and resource utilization.

## Error Handling and Resilience

Error Type	Description	Recovery Strategy
ErrorTypeValidation	Invalid flag definitions or user context	Return validation errors with specific field information
ErrorTypeNetwork	Connection failures or timeouts	Retry with exponential backoff, fallback to cache
ErrorTypeStorage	Database or persistence layer failures	Switch to read-only mode, use cached data
ErrorTypeEvaluation	Rule processing errors or dependency cycles	Return default variant with error logging
ErrorTypeCircuit	Circuit breaker preventing requests	Return cached results, attempt periodic recovery
ErrorTypeDependency	Required services unavailable	Activate fallback data sources
ErrorTypeResource	Memory or processing capacity exceeded	Enable load shedding, scale horizontally

**Split-brain Scenario** occurs during network partitions where different parts of the system operate independently with potentially conflicting state. The system prevents this through consistent leadership election and read-only fallback modes during partition events.

**Optimistic Consistency** allows immediate writes with asynchronous distribution to connected clients. While this may create temporary inconsistency across clients, it provides better user experience than waiting for

global consensus on every flag change.

**At-least-once Delivery** guarantees message delivery with possible duplication. The system includes sequence numbers and idempotency checks to handle duplicate flag updates gracefully.

## Development and Testing

**Property-based Testing** validates system invariants using randomly generated inputs. Rather than testing specific scenarios, property tests verify that fundamental system properties (like assignment stability and rule determinism) hold across a wide range of inputs.

**Milestone Validation** provides quality gates between development phases. Each milestone includes specific acceptance criteria, test scenarios, and performance benchmarks that must be met before proceeding to subsequent development phases.

**Assignment Stability** ensures users receive the same variant consistently across evaluations. This property is critical for user experience and experiment validity - users should not flip between variants due to evaluation timing or system state.

**Rule Evaluation Determinism** guarantees that targeting rules evaluate consistently given identical input context. This property ensures reliable system behavior and simplifies debugging when evaluation outcomes don't match expectations.

**Cache Consistency** maintains alignment between cached data and authoritative sources. While temporary inconsistency is acceptable for performance, the system must eventually converge to consistent state across all cache levels.

## Advanced Concepts

**Prerequisite Flags** are required flags that must be enabled before dependent flags activate. This creates a dependency graph that prevents invalid feature combinations - for example, a premium feature flag that depends on the user authentication flag being enabled.

**Multi-environment Support** provides independent flag configurations across development, staging, and production environments. Each environment maintains separate flag state while sharing common flag definitions and targeting rule templates.

**Configuration Promotion** enables controlled advancement of flag settings between environments. Changes typically flow from development → staging → production with appropriate testing and approval gates at each stage.

**Edge Evaluation** performs flag evaluation at geographically distributed nodes to reduce latency for global applications. This requires careful cache synchronization and fallback strategies to maintain consistency across edge locations.

**Contextual Targeting** extends basic user targeting to include environmental and situational factors like time of day, geographic location, device characteristics, or application version. This enables sophisticated targeting

scenarios like "show feature X only to mobile users in Pacific timezone during business hours."

## Performance and Scalability

**Bucket Ranges** represent deterministic allocation intervals for percentage splits. When implementing a 30%/70% split, users hash into buckets 0-29 (first variant) or 30-99 (second variant). The bucket assignment remains stable as percentages change.

**Sequence Numbers** provide monotonic identifiers that prevent out-of-order processing of flag updates. Each update includes a sequence number that clients use to detect and handle updates that arrive out of order due to network conditions.

**Event Ordering** ensures updates are processed in correct sequence even when network delivery occurs out of order. The system buffers out-of-order events and processes them when missing sequence numbers arrive.

**Resource Exhaustion** occurs when the system runs out of memory, file handles, or other critical resources. The system monitors resource usage and implements graceful degradation strategies like reducing cache sizes or limiting concurrent connections.

**Dependency Graph** represents the directed acyclic graph of flag prerequisite relationships. The system validates this graph to prevent circular dependencies and computes evaluation order to ensure prerequisite flags are evaluated before dependent flags.

## Implementation Guidance

The glossary definitions above establish the conceptual foundation needed to implement a robust feature flag system. Each term has been defined with sufficient precision to guide implementation decisions while remaining accessible to developers new to feature flag systems.

When implementing system components, refer to these definitions to ensure consistent terminology and behavior across all modules. The data structure definitions provide the exact field specifications needed for implementation, while the conceptual terms establish the mental models for understanding system behavior.

For debugging and troubleshooting, this glossary serves as a reference for understanding error messages, log entries, and system behavior descriptions. When encountering unfamiliar terms in documentation or code comments, this glossary provides the authoritative definition within the context of our feature flag system.

The progression from basic concepts (feature flags, variants) through advanced topics (statistical significance, edge evaluation) reflects the learning journey developers will experience when implementing and operating the system. Mastering the foundational terms enables understanding of more sophisticated concepts as system requirements evolve.