Multi-Channel Notification Service: Design Document

Overview

A unified notification system that abstracts multiple delivery channels (email, SMS, push, in-app) behind a common interface, enabling intelligent routing, fallback handling, and user preference management. The key architectural challenge is designing a pluggable channel abstraction that handles the vastly different APIs, failure modes, and delivery semantics of each notification provider.

> 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):** Foundation for all milestones — establishes the core notification system requirements and architectural approach

Modern software applications need to communicate with users across multiple channels: sending password reset emails, SMS verification codes, mobile push notifications for breaking news, and in-app messages for feature announcements. However, building and maintaining these notification capabilities presents significant engineering challenges that mirror the complexity of operating a global postal service.

The Postal Service Analogy

Think of our notification system as a modern postal service that handles different types of mail delivery. Just as a postal service must route letters, packages, and priority mail through different delivery networks while respecting customer preferences and tracking delivery status, our notification system must intelligently route messages through email providers, SMS gateways, and push notification services.

The Address Book and Delivery Preferences

In the postal world, customers can specify delivery preferences: "No junk mail," "Hold packages when traveling," or "Deliver priority mail only between 9 AM and 6 PM." Our notification system needs similar granular controls. A user might want marketing emails but no marketing SMS messages, or they might want transaction confirmations via push notifications but security alerts via both email and SMS for redundancy.

The postal service maintains detailed address books with forwarding rules, delivery instructions, and accessibility requirements. Similarly, our notification system maintains user preference profiles that specify which channels are acceptable for different message categories, quiet hours when non-urgent messages should be suppressed, and fallback channels when primary delivery methods fail.

Multiple Delivery Networks and Failover

A postal service doesn't own every delivery truck and airplane. Instead, it contracts with FedEx for overnight packages, DHL for international shipping, and local carriers for last-mile delivery. Each partner has different capabilities, pricing models, and reliability characteristics. When FedEx experiences weather delays, the postal service automatically reroutes priority packages through UPS.

Our notification system faces identical challenges. Email delivery might use SendGrid for transactional messages and Mailchimp for marketing campaigns. SMS delivery could route through Twilio for US numbers and local providers for international markets. Push notifications require separate integrations with Apple Push Notification service (APNs) for iOS devices and Firebase Cloud Messaging (FCM) for Android. When SendGrid experiences an outage, the system must seamlessly failover to a backup email provider without losing messages.

Tracking and Analytics

Modern postal services provide detailed tracking: when a package was picked up, which distribution centers it passed through, when it was delivered, and whether delivery required a signature. Customers expect this visibility, and postal services use this data to optimize routes and identify delivery issues.

Notification tracking requires similar sophistication. We need to know when an email was sent, whether it was delivered to the recipient's inbox or filtered as spam, whether the recipient opened it, and which links they clicked. SMS delivery status helps identify invalid phone numbers. Push notification analytics reveal which devices have uninstalled the app. This tracking data serves two purposes: providing delivery confirmation to the business and generating analytics to optimize future notification campaigns.

Message Templates and Localization

A postal service standardizes shipping labels, customs forms, and delivery notifications into templates that work across different countries and languages. The core information remains consistent, but the format adapts to local regulations and customs.

Notification templates serve the same purpose. A password reset notification contains the same essential information (user name, reset link, expiration time) whether sent via email, SMS, or push notification, but the formatting must adapt to each channel's constraints. Email templates can include rich HTML formatting and branding, SMS templates must fit within 160-character limits, and push notifications require short titles with brief body text. Multi-language support adds another dimension: the same template must render correctly in English, Spanish, and Japanese while respecting cultural norms for each locale.

Existing Notification Approaches

Understanding the current landscape of notification systems helps clarify why a unified approach provides significant advantages over existing alternatives.

> **Decision: Unified vs. Point-to-Point Integration Strategy**
>
> - **Context**: Teams typically choose between building direct integrations with each notification provider, using notification-as-a-service platforms, or creating unified internal systems
> - **Options Considered**: Direct provider integrations, third-party platforms (OneSignal, Pusher), unified internal system
> - **Decision**: Build a unified internal notification system with pluggable channel adapters
> - **Rationale**: Provides complete control over routing logic, user preferences, and delivery tracking while avoiding vendor lock-in and external service dependencies
> - **Consequences**: Higher initial development effort but greater long-term flexibility and cost control

Approach	Implementation Complexity	Vendor Lock-In Risk	Feature Control	Long-Term Cost
Direct Integrations	Low initially, High at scale	High per provider	Limited to provider features	Variable, hard to predict
Third-Party Platform	Low	Very High	Limited customization	High recurring fees
Unified Internal System	High initially	Low	Complete control	Predictable infrastructure

Direct Provider Integrations

The most common starting approach involves building separate integrations with each notification provider. The application directly calls SendGrid's API for emails, Twilio's API for SMS, and FCM's API for push notifications. Each integration handles authentication, message formatting, and error handling independently.

This approach appears simple initially but creates significant maintenance overhead as the system scales. Each provider has different authentication mechanisms: SendGrid uses API keys, Twilio requires account SID and auth tokens, and FCM needs service account credentials. Error handling varies dramatically: SendGrid returns detailed bounce classifications, Twilio provides carrier-specific error codes, and FCM handles device token invalidation through complex callback mechanisms.

User preference management becomes particularly challenging with direct integrations. The application must implement separate opt-out mechanisms for each channel, track delivery failures across multiple systems, and coordinate fallback logic when providers experience outages. Adding a new notification provider requires duplicating all this logic, increasing the risk of bugs and inconsistent behavior.

Notification-as-a-Service Platforms

Third-party platforms like OneSignal, Pusher Beams, and Amazon SNS promise to abstract away provider complexity behind unified APIs. These services handle provider integrations, maintain fallback configurations, and provide consolidated analytics dashboards.

However, notification platforms introduce their own limitations. Most platforms focus primarily on push notifications with limited email and SMS capabilities. Customization options are constrained to what the platform supports: you cannot implement complex routing rules, custom preference hierarchies, or specialized template logic that doesn't fit their standard feature set.

Vendor lock-in represents a significant long-term risk. Migrating away from a notification platform requires rebuilding all integration logic, migrating user preferences, and potentially losing historical analytics data. Pricing models often include per-message fees that become expensive at scale, and platform outages can disable all notification channels simultaneously.

Unified Internal Systems

Building a unified internal notification system requires more initial engineering effort but provides complete control over routing logic, user preferences, and delivery tracking. The system abstracts provider differences behind consistent internal interfaces while maintaining the flexibility to implement complex business logic.

This approach enables sophisticated features that external platforms typically cannot support: routing notifications through multiple providers based on cost optimization, implementing custom user preference hierarchies that span multiple applications, and integrating deeply with internal analytics systems for advanced attribution tracking.

The trade-off involves higher initial development complexity and ongoing maintenance responsibility. The engineering team must understand the nuances of each notification provider, implement reliable retry logic, and build monitoring systems to detect delivery issues. However, this investment pays dividends as notification requirements become more sophisticated and message volumes increase.

> The key insight driving our unified approach is that notifications represent a core business capability, not a commodity service. The ability to reliably reach users through their preferred channels with personalized, contextual messages directly impacts user engagement, retention, and business metrics. External platforms optimize for their own business models, not yours.

Provider Abstraction Challenges

Creating effective abstractions across notification providers requires understanding their fundamental differences in delivery semantics, error handling, and capabilities. Email providers distinguish between transactional and marketing messages with different sending limits and deliverability optimization. SMS providers have complex international routing with varying costs and delivery guarantees. Push notification services handle device token management, silent notifications, and rich media attachments differently.

The abstraction layer must accommodate these differences without creating a lowest-common-denominator interface that limits functionality. This requires careful interface design that exposes provider-specific capabilities when needed while maintaining consistent behavior for common operations.

Compliance and Regulation Considerations

Different notification channels are subject to different regulatory requirements. Email marketing must comply with CAN-SPAM Act provisions for unsubscribe mechanisms and sender identification. SMS messaging falls under TCPA regulations requiring explicit consent for marketing messages. Push notifications must respect platform-specific guidelines for notification frequency and content.

A unified notification system must implement these compliance requirements consistently while adapting to the specific regulations for each channel and geographic region. This includes maintaining audit trails for consent management, implementing required opt-out mechanisms, and ensuring message content meets regulatory standards.

Implementation Guidance

The foundation for implementing a unified notification system involves selecting appropriate technologies and establishing the basic project structure that will support all four milestones.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Web Framework	`net/http` with `gorilla/mux`	`gin-gonic/gin` or `fiber/fiber`
Message Queue	`RabbitMQ` with `rabbitmq/amqp091-go`	`Apache Kafka` with `segmentio/kafka-go`
Database	`PostgreSQL` with `lib/pq`	`PostgreSQL` with `gorm` ORM
Template Engine	`text/template` and `html/template`	`Masterminds/sprig` with template helpers
Configuration	`yaml` with `gopkg.in/yaml.v3`	`viper` for multi-format config
Logging	`log/slog` (Go 1.21+)	`sirupsen/logrus` or `uber-go/zap`
HTTP Client	`net/http` with custom retry logic	`hashicorp/go-retryablehttp`

B. Recommended Project Structure

```
notification-service/
├── cmd/
│   ├── server/main.go          ← HTTP API server entry point
│   ├── worker/main.go          ← Background notification processor
│   └── migrate/main.go         ← Database migration utility
├── internal/
│   ├── api/                    ← HTTP handlers and routing
│   │   ├── handlers.go         ← Notification submission endpoints
│   │   ├── webhook.go          ← Provider webhook receivers
│   │   └── preferences.go      ← User preference management
│   ├── channels/               ← Channel implementations (Milestone 1)
│   │   ├── interface.go        ← Common channel interface
│   │   ├── email/              ← Email channel implementation
│   │   ├── sms/                ← SMS channel implementation
│   │   └── push/               ← Push notification implementation
│   ├── routing/                ← Message routing and fallback logic
│   │   ├── router.go           ← Main routing engine
│   │   ├── rules.go            ← Routing rule evaluation
│   │   └── fallback.go         ← Fallback and circuit breaker
│   ├── templates/              ← Template system (Milestone 2)
│   │   ├── engine.go           ← Template rendering engine
│   │   ├── localization.go     ← Multi-language support
│   │   └── storage.go          ← Template storage and versioning
│   ├── preferences/            ← User preference system (Milestone 3)
│   │   ├── manager.go          ← Preference management logic
│   │   ├── unsubscribe.go      ← Unsubscribe token handling
│   │   └── quiet_hours.go      ← Do-not-disturb implementation
│   ├── tracking/               ← Delivery tracking (Milestone 4)
│   │   ├── tracker.go          ← Delivery status tracking
│   │   ├── analytics.go        ← Open/click analytics
│   │   └── bounces.go          ← Bounce and failure handling
│   ├── queue/                  ← Message queue abstraction
│   │   ├── publisher.go        ← Message publishing
│   │   └── consumer.go         ← Message consumption
│   └── storage/                ← Database models and queries
│       ├── models.go           ← Core data structures
│       ├── notifications.go    ← Notification persistence
│       ├── users.go            ← User and preference storage
│       └── templates.go        ← Template storage operations
├── pkg/                        ← Shared utilities and external interface
│   ├── config/                 ← Configuration loading and validation
│   ├── logger/                 ← Structured logging setup
│   └── errors/                 ← Common error types and handling
├── migrations/                 ← Database schema migrations
├── templates/                  ← Notification template files
│   ├── email/                  ← Email template variants
│   ├── sms/                    ← SMS template variants
│   └── push/                   ← Push notification templates
├── config/
│   ├── config.yaml             ← Application configuration
│   └── providers.yaml          ← Channel provider configuration
├── docker-compose.yml          ← Development environment setup
├── Makefile                    ← Build and development commands
└── README.md                   ← Setup and usage documentation
```

C. Core Configuration Structure

```go
// pkg/config/config.go

package config

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

// Config represents the complete application configuration
type Config struct {
    Server     ServerConfig     `yaml:"server"`
    Database   DatabaseConfig   `yaml:"database"`
    Queue      QueueConfig      `yaml:"queue"`
    Channels   ChannelConfigs   `yaml:"channels"`
    Templates  TemplateConfig   `yaml:"templates"`
    Tracking   TrackingConfig   `yaml:"tracking"`
}

// ServerConfig configures the HTTP API server
type ServerConfig struct {
    Host         string        `yaml:"host"`
    Port         int           `yaml:"port"`
    ReadTimeout  time.Duration `yaml:"read_timeout"`
    WriteTimeout time.Duration `yaml:"write_timeout"`
    IdleTimeout  time.Duration `yaml:"idle_timeout"`
}

// DatabaseConfig configures PostgreSQL connection
type DatabaseConfig struct {
    Host         string `yaml:"host"`
    Port         int    `yaml:"port"`
    Database     string `yaml:"database"`
    Username     string `yaml:"username"`
    Password     string `yaml:"password"`
    SSLMode      string `yaml:"ssl_mode"`
    MaxOpenConns int    `yaml:"max_open_conns"`
    MaxIdleConns int    `yaml:"max_idle_conns"`
```

```
}


// QueueConfig configures message queue connection
type QueueConfig struct {
    URL              string        `yaml:"url"`
    Exchange         string        `yaml:"exchange"`
    NotificationQueue string        `yaml:"notification_queue"`
    WebhookQueue     string        `yaml:"webhook_queue"`
    RetryLimit       int           `yaml:"retry_limit"`
    RetryDelay       time.Duration `yaml:"retry_delay"`
}


// LoadConfig reads configuration from file and environment variables
func LoadConfig(path string) (*Config, error) {
    // TODO: Read configuration file
    // TODO: Override with environment variables
    // TODO: Validate required fields
    // TODO: Set reasonable defaults
    // Hint: Use os.Getenv() to override YAML values with environment variables
}
```

D. Message Queue Infrastructure

GO

```go
// internal/queue/publisher.go

package queue

import (
    "context"
    "encoding/json"
    "github.com/rabbitmq/amqp091-go"
    "time"
)

// Publisher handles message publishing to notification queues
type Publisher struct {
    conn     *amqp091.Connection
    channel  *amqp091.Channel
    exchange string
}

// NotificationMessage represents a queued notification for processing
type NotificationMessage struct {
    ID          string                 `json:"id"`
    UserID      string                 `json:"user_id"`
    Type        string                 `json:"type"`
    Category    string                 `json:"category"`
    Priority    Priority               `json:"priority"`
    Channels    []string               `json:"channels"`
    TemplateID  string                 `json:"template_id"`
    Variables   map[string]interface{} `json:"variables"`
    ScheduledAt *time.Time             `json:"scheduled_at,omitempty"`
    CreatedAt   time.Time              `json:"created_at"`
}

type Priority int

const (
    PriorityLow Priority = iota
    PriorityNormal
    PriorityHigh
    PriorityUrgent
)
```

```go
// NewPublisher creates a new message queue publisher
func NewPublisher(amqpURL, exchange string) (*Publisher, error) {
    // TODO: Connect to RabbitMQ using amqpURL
    // TODO: Open a channel for publishing
    // TODO: Declare the exchange with type "topic"
    // TODO: Return configured Publisher instance
    // Hint: Use amqp091.Dial() to connect, then conn.Channel() to get a channel
}

// PublishNotification queues a notification for processing
func (p *Publisher) PublishNotification(ctx context.Context, msg *NotificationMessage) error {
    // TODO: Serialize message to JSON
    // TODO: Determine routing key based on priority and category
    // TODO: Publish message to exchange with appropriate routing key
    // TODO: Handle context cancellation and timeouts
    // Hint: Use json.Marshal() for serialization, channel.PublishWithContext() for publishing
}

// Close shuts down the publisher and releases resources
func (p *Publisher) Close() error {
    // TODO: Close channel and connection gracefully
    // TODO: Return any closing errors
}
```

E. Database Models Foundation

GO

```go
// internal/storage/models.go

package storage

import (
    "database/sql/driver"
    "time"
)

// User represents a notification recipient with preferences
type User struct {
    ID          string    `db:"id" json:"id"`
    Email       string    `db:"email" json:"email"`
    PhoneNumber string    `db:"phone_number" json:"phone_number"`
    PushTokens  []string  `db:"push_tokens" json:"push_tokens"`
    Locale      string    `db:"locale" json:"locale"`
    Timezone    string    `db:"timezone" json:"timezone"`
    CreatedAt   time.Time `db:"created_at" json:"created_at"`
    UpdatedAt   time.Time `db:"updated_at" json:"updated_at"`
}

// Notification represents a notification instance with delivery tracking
type Notification struct {
    ID            string                 `db:"id" json:"id"`
    UserID        string                 `db:"user_id" json:"user_id"`
    Type          string                 `db:"type" json:"type"`
    Category      string                 `db:"category" json:"category"`
    Priority      Priority               `db:"priority" json:"priority"`
    TemplateID    string                 `db:"template_id" json:"template_id"`
    Variables     map[string]interface{} `db:"variables" json:"variables"`
    Status        NotificationStatus     `db:"status" json:"status"`
    ScheduledAt   *time.Time            `db:"scheduled_at" json:"scheduled_at"`
    SentAt        *time.Time            `db:"sent_at" json:"sent_at"`
    DeliveredAt   *time.Time            `db:"delivered_at" json:"delivered_at"`
    FailedAt      *time.Time            `db:"failed_at" json:"failed_at"`
    FailureReason string                `db:"failure_reason" json:"failure_reason"`
    CreatedAt     time.Time             `db:"created_at" json:"created_at"`
    UpdatedAt     time.Time             `db:"updated_at" json:"updated_at"`
}
```

```go
type NotificationStatus string

const (
    StatusQueued    NotificationStatus = "queued"
    StatusSending   NotificationStatus = "sending"
    StatusSent      NotificationStatus = "sent"
    StatusDelivered NotificationStatus = "delivered"
    StatusFailed    NotificationStatus = "failed"
    StatusBounced   NotificationStatus = "bounced"
)

// Template represents a notification template with localization support
type Template struct {
    ID          string            `db:"id" json:"id"`
    Name        string            `db:"name" json:"name"`
    Version     int               `db:"version" json:"version"`
    Channel     string            `db:"channel" json:"channel"`
    Locale      string            `db:"locale" json:"locale"`
    Subject     string            `db:"subject" json:"subject"`
    Body        string            `db:"body" json:"body"`
    Variables   []string          `db:"variables" json:"variables"`
    Metadata    map[string]string `db:"metadata" json:"metadata"`
    IsActive    bool              `db:"is_active" json:"is_active"`
    CreatedAt   time.Time         `db:"created_at" json:"created_at"`
    UpdatedAt   time.Time         `db:"updated_at" json:"updated_at"`
}

// UserPreference represents a user's notification channel preferences
type UserPreference struct {
    ID        string    `db:"id" json:"id"`
    UserID    string    `db:"user_id" json:"user_id"`
    Category  string    `db:"category" json:"category"`
    Channel   string    `db:"channel" json:"channel"`
    Enabled   bool      `db:"enabled" json:"enabled"`
    CreatedAt time.Time `db:"created_at" json:"created_at"`
    UpdatedAt time.Time `db:"updated_at" json:"updated_at"`
}
```

F. Language-Specific Implementation Hints

- **Context Handling**: Use `context.Context` throughout for request scoping, cancellation, and timeout handling. Pass context as the first parameter to all service methods.
- **Error Wrapping**: Use `fmt.Errorf("operation failed: %w", err)` to wrap errors with context. Consider implementing custom error types for different failure categories.
- **Configuration**: Use struct tags for YAML unmarshaling and field validation. Implement a validation method that checks required fields and reasonable defaults.
- **Database Connections**: Use `database/sql` with `lib/pq` driver for PostgreSQL. Implement connection pooling with reasonable limits (max 25 open connections for most applications).
- **JSON Handling**: Use `json.Marshal` and `json.Unmarshal` for message serialization. Implement custom `MarshalJSON` methods for complex types like priority enums.
- **Concurrency**: Use `sync.WaitGroup` for coordinating goroutines when sending notifications to multiple channels. Implement proper timeout handling with `context.WithTimeout`.
- **HTTP Clients**: Create a shared `http.Client` with reasonable timeouts (10-30 seconds) and retry logic. Use `http.Client.Do()` with context for all external API calls.

G. Milestone Checkpoint: Foundation Setup

After implementing the basic project structure and configuration system:

1. **Build Verification**: Run `go mod tidy && go build ./cmd/server` — should compile without errors
2. **Configuration Loading**: Test configuration loading with `go run cmd/server/main.go -config config/config.yaml` — should start server and connect to database
3. **Database Migration**: Run initial schema migration — should create tables for users, notifications, templates, and preferences
4. **Message Queue Connection**: Verify RabbitMQ connectivity — should connect and declare exchanges without errors
5. **Health Check Endpoint**: Test `curl http://localhost:8080/health` — should return JSON status with database and queue connectivity

Signs of Implementation Issues:

Symptom	Likely Cause	Diagnosis	Fix
Import cycle errors	Circular dependencies between packages	Run `go list -f '{{.ImportPath}} {{.Deps}}' ./...`	Reorganize imports, move shared types to separate package
Database connection failures	Incorrect connection string or missing database	Check PostgreSQL logs, verify credentials	Update connection string, ensure database exists
RabbitMQ connection timeout	RabbitMQ not running or incorrect URL	Test connection with `amqp://guest:guest@localhost:5672/`	Start RabbitMQ service, verify connection URL
Configuration not loading	YAML syntax errors or missing file	Validate YAML syntax with online parser	Fix YAML indentation, verify file path

Goals and Non-Goals

> **Milestone(s):** Foundation for all milestones — establishes clear boundaries and success criteria for the notification service

Think of defining goals and non-goals like drawing a property line around your house. You need to know exactly which problems you're responsible for solving (your yard) and which ones belong to your neighbors (their yards). Without clear boundaries, you'll either try to solve everything and never finish, or you'll miss critical requirements that users actually need.

The notification service sits at the intersection of multiple complex domains: message delivery, user experience, compliance regulations, and system reliability. Each of these domains could consume years of development effort if approached without constraints. By explicitly defining what we will and won't build, we create a focused product that delivers real value while maintaining reasonable scope.

Functional Goals

The notification service aims to solve four core problems that plague modern application notification systems: **channel fragmentation**, **delivery unreliability**, **poor user experience**, and **lack of visibility**. Each goal directly addresses one of these fundamental pain points.

Multi-Channel Delivery Abstraction

Our primary functional goal is to create a **channel abstraction** that unifies disparate notification providers behind a common interface. Today's applications typically integrate directly with each provider's API — SendGrid for email, Twilio for SMS, Firebase Cloud Messaging for push notifications, and custom WebSocket connections for in-app messages. This creates a maintenance nightmare where each provider requires different authentication, error handling, rate limiting, and retry logic.

The notification service will provide a single API endpoint that accepts notification requests and intelligently routes them to the appropriate delivery channel. A client application should be able to send a notification by specifying the recipient, message content, and notification type, without knowing or caring whether it gets delivered via email, SMS, or push notification.

Channel Type	Provider Examples	Service Interface	Channel-Specific Handling
Email	SendGrid, Mailgun, Amazon SES	`send(recipient, subject, body, priority)`	HTML/plain text rendering, attachment support, bounce handling
SMS	Twilio, Nexmo, AWS SNS	`send(phoneNumber, message, priority)`	Character limit enforcement, cost optimization, delivery receipts
Push	FCM, APNs, OneSignal	`send(deviceToken, title, body, payload)`	Platform-specific formatting, token refresh handling, rich media
In-App	WebSocket, Server-Sent Events	`send(userSession, title, body, actions)`	Real-time delivery, connection management, offline queueing

The abstraction must handle the drastically different capabilities and constraints of each channel. Email supports rich HTML content and attachments but has delayed delivery. SMS has strict character limits and costs money per message but delivers almost instantly. Push notifications can include interactive buttons and rich media but require valid device tokens that expire unpredictably.

> **Decision: Channel Interface Design**
>
> - **Context**: Different notification providers have incompatible APIs, authentication methods, and failure modes
> - **Options Considered**: Direct provider integration, adapter pattern with common interface, message transformation pipeline
> - **Decision**: Implement adapter pattern with standardized `Channel` interface
> - **Rationale**: Adapter pattern allows us to add new providers without changing core routing logic, while the common interface ensures consistent error handling and retry behavior across all channels
> - **Consequences**: Initial development requires more abstraction work, but adding new providers becomes trivial and existing code doesn't break when providers change APIs

Intelligent Routing and Fallback

The second functional goal is implementing a **routing engine** that selects the optimal delivery channel based on message urgency, user preferences, provider availability, and business rules. Simple notification systems send everything via email by default, but this creates poor user experiences and high costs.

The routing engine will evaluate multiple factors to determine the best delivery path:

1. **Message Priority**: `PriorityUrgent` notifications (password reset, security alerts) should prefer immediate channels like SMS or push, while `PriorityLow` notifications (newsletters, marketing) can use cost-effective email delivery
2. **User Preferences**: Recipients can specify preferred channels for different notification categories
3. **Channel Availability**: Circuit breakers track provider health and route around failing services
4. **Cost Optimization**: SMS delivery costs significantly more than email, so non-urgent messages should prefer cheaper channels
5. **Delivery Speed Requirements**: Security alerts need immediate delivery, while monthly reports can wait in queue

When the primary channel fails, the **fallback strategy** should automatically attempt delivery via backup channels. However, fallback logic must be intelligent — we shouldn't send a low-priority marketing message via expensive SMS just because email is temporarily down.

Scenario	Primary Channel	Fallback Chain	Rationale
Password reset (urgent)	Push notification	SMS → Email	Security-critical, needs immediate attention
Marketing newsletter (low priority)	Email	None	Cost-sensitive, not urgent enough for SMS fallback
Order confirmation (normal)	Email	SMS (if email bounces)	Important but not security-critical
System alert (urgent)	SMS	Push → Email	Must reach recipient immediately via any channel

The routing engine must also respect **quiet hours** and **do-not-disturb** settings. A newsletter sent at 2 AM shouldn't wake users with a push notification, even if that's their preferred channel during business hours.

Template System with Localization

The third functional goal is providing a **template engine** that enables content creators to design notification messages without writing code. Marketing teams and customer success managers need to update notification content frequently, but they shouldn't require engineering help for simple copy changes.

The template system will support variable substitution using mustache-style syntax, allowing dynamic personalization of notification content. A template might look like: "Hello {{user.firstName}}, your order {{order.id}} has shipped and will arrive by {{delivery.estimatedDate}}."

Template Feature	Description	Example
Variable Substitution	Replace placeholders with user/context data	`{{user.firstName}}` → "John"
Conditional Content	Show/hide sections based on data	`{{#user.isPremium}}Premium perks...{{/user.isPremium}}`
Localization	Multiple language variants per template	`subject_en: "Welcome!"`, `subject_es: "¡Bienvenido!"`
Channel Formatting	Adapt content for different delivery channels	HTML for email, plain text for SMS
Content Safety	Escape variables to prevent XSS in HTML emails	`{{user.input}}` → HTML-escaped output

Localization support is critical for global applications. The template system must detect the recipient's preferred locale and select the appropriate translation. When a specific translation isn't available, it should fall back gracefully through a locale chain (e.g., `es-MX` → `es` → `en`).

Template versioning allows content creators to update messages while providing rollback capability if new versions contain errors. Version control also enables A/B testing by sending different template versions to different user segments.

> **Decision: Template Engine Choice**
>
> - **Context**: Need balance between template expressiveness and security/performance
> - **Options Considered**: Full programming language (PHP/JavaScript), logic-less templates (Mustache), restricted template language (Go templates with constraints)
> - **Decision**: Mustache-style logic-less templates with custom helpers for localization
> - **Rationale**: Logic-less templates prevent security vulnerabilities from user-generated content while remaining simple enough for non-technical users. Custom helpers provide necessary functionality like pluralization and date formatting.
> - **Consequences**: Some advanced template features require custom helper functions rather than inline logic, but this constraint improves security and maintainability

User Preference Management

The fourth functional goal is comprehensive **user preference management** that gives recipients granular control over which notifications they receive and how they receive them. Poor preference controls are the primary cause of user frustration with notification systems — users either get overwhelmed with unwanted messages or miss important communications.

The preference system will operate on multiple dimensions:

1. **Channel Preferences**: Users can disable specific delivery channels entirely ("never send me SMS")
2. **Category Preferences**: Different notification types can be controlled independently ("send me order updates but not marketing emails")
3. **Timing Preferences**: Quiet hours prevent non-urgent notifications during sleep or focus time
4. **Frequency Limits**: Rate limiting prevents notification spam even when individual messages are wanted

Preference Type	Granularity	Example Configuration
Global Opt-out	All notifications	"Unsubscribe from everything"
Channel Opt-out	Per delivery method	"No SMS notifications"
Category Opt-out	Per notification type	"No marketing emails, yes to transactional"
Quiet Hours	Time-based suppression	"No notifications 10 PM - 8 AM Pacific"
Frequency Limits	Rate limiting per category	"Max 1 marketing email per week"

The preference system must handle **legal compliance** requirements, particularly around marketing communications. CAN-SPAM Act and GDPR regulations require specific unsubscribe mechanisms and response timeframes. Marketing emails must include one-click unsubscribe links that work immediately, while transactional emails (receipts, password resets, security alerts) typically cannot be unsubscribed from.

Preference inheritance creates a logical hierarchy where more specific settings override general ones. A user might allow marketing notifications in general but disable SMS marketing while keeping email marketing enabled.

The unsubscribe system will use **HMAC-signed tokens** in unsubscribe URLs to prevent abuse. Without cryptographic protection, malicious actors could unsubscribe arbitrary users by guessing email addresses or user IDs.

Delivery Tracking and Analytics

The fifth functional goal is comprehensive **delivery tracking** that monitors notification lifecycle from submission through final delivery or failure. Most notification systems are "fire and forget" — they submit messages to providers and hope for the best. This creates blind spots when notifications fail to deliver, leading to poor user experiences and missed business opportunities.

The tracking system will monitor notifications through multiple states:

Status	Description	Trigger Event	Next Possible States
`StatusQueued`	Notification accepted and queued for processing	API submission	`StatusSending`, `StatusFailed`
`StatusSending`	Notification being processed by channel handler	Processing start	`StatusSent`, `StatusFailed`
`StatusSent`	Notification submitted to external provider	Provider API success	`StatusDelivered`, `StatusFailed`, `StatusBounced`
`StatusDelivered`	Notification delivered to recipient device/inbox	Provider webhook	`StatusOpened`, `StatusClicked`
`StatusOpened`	Recipient opened/viewed the notification	Tracking pixel or app event	`StatusClicked`
`StatusClicked`	Recipient clicked a link in the notification	Click tracking redirect	None (terminal state)
`StatusFailed`	Delivery failed due to provider or network error	Provider error response	None (terminal state)
`StatusBounced`	Message bounced due to invalid recipient address	Provider bounce webhook	None (terminal state)

Open and click tracking requires different mechanisms for each channel. Email uses transparent 1×1 pixel images for open tracking and redirect URLs for click tracking. Push notifications report opens through platform APIs when users tap them. SMS and in-app messages have limited tracking capabilities.

Analytics dashboards will aggregate delivery metrics to identify trends and problems:

- **Delivery Rate**: Percentage of notifications successfully delivered vs. sent
- **Bounce Rate**: Percentage of notifications that bounce due to invalid addresses
- **Open Rate**: Percentage of delivered notifications that recipients open
- **Click Rate**: Percentage of opened notifications where recipients click links
- **Channel Performance**: Comparative metrics across email, SMS, push, and in-app channels

Real-time alerting will notify operations teams when delivery rates drop below acceptable thresholds, indicating provider outages or configuration problems.

Non-Goals

Defining what we **won't** build is as important as defining what we will build. The notification service operates within a larger application ecosystem, and it should integrate cleanly with existing systems rather than duplicating their functionality.

Authentication and Authorization

The notification service will **not** implement user authentication or authorization systems. It assumes that client applications have already verified user identity and permission to send notifications. The service trusts that incoming notification requests are authorized and focuses exclusively on reliable delivery.

This boundary keeps the notification service stateless and simple. Authentication systems require user credential storage, session management, password reset flows, and integration with identity providers — all complex domains that are orthogonal to notification delivery.

Client applications remain responsible for:

- Verifying that users can send notifications to specific recipients
- Implementing API key or OAuth-based service authentication
- Managing user sessions and login state
- Enforcing business rules about notification permissions

The notification service will accept notification requests that include recipient identifiers (user ID, email address, phone number) and trust that the calling application has verified the sender's authorization to contact those recipients.

User Management and Profile Storage

The notification service will **not** maintain comprehensive user profiles or serve as a user management system. It will store only the minimal user data required for notification delivery: contact information (email, phone, device tokens), locale preferences, and notification preferences.

User management systems handle complex requirements like profile creation, data validation, GDPR compliance for personal data, user lifecycle management, and integration with CRM systems. These responsibilities belong to dedicated user management services or customer data platforms.

Data Type	Notification Service Responsibility	External System Responsibility
Contact Info	Store email/phone for delivery	Validate format, verify ownership, handle updates
Locale Preference	Use for template localization	Detect user's language, manage locale data
Notification Preferences	Control delivery channels and timing	Provide preference UI, handle consent workflows
User Profiles	None — accept user ID references	Store names, demographics, purchase history, etc.

The service will accept user data updates through API calls but won't implement user registration, profile editing interfaces, or profile data validation beyond what's required for delivery.

Campaign Scheduling and Management

The notification service will **not** implement marketing campaign management, advanced scheduling, or bulk notification orchestration. It processes individual notification requests in near real-time but doesn't provide campaign planning, audience segmentation, or scheduled send functionality.

Campaign management systems require sophisticated features like:

- Audience segmentation and targeting rules
- Send time optimization based on user behavior
- A/B testing with statistical significance tracking
- Campaign performance analytics and reporting
- Drag-and-drop email builders and content management
- Automated drip campaigns and behavioral triggers

These features represent a separate product domain focused on marketing automation rather than reliable message delivery. The notification service should integrate with existing campaign management tools (Mailchimp, HubSpot, Marketo) rather than competing with them.

Campaign systems can use the notification service as a delivery backend by making individual API calls for each recipient, but they handle the logic of determining who should receive notifications and when.

A/B Testing and Experimentation

The notification service will **not** provide A/B testing frameworks or experimentation platforms. While it supports template versioning that could enable A/B tests, it won't implement experiment design, statistical analysis, or automated winner selection.

A/B testing requires careful statistical design to ensure valid results:

- Sample size calculations to achieve statistical power
- Randomization algorithms that ensure balanced treatment groups
- Statistical significance testing to determine experiment winners
- Holdout group management for incrementality testing
- Integration with analytics platforms for conversion tracking

These capabilities belong in dedicated experimentation platforms (Optimizely, LaunchDarkly, internal A/B testing frameworks) that specialize in experiment design and analysis.

The notification service supports A/B testing by allowing external systems to specify which template version to use for each notification, but the experiment logic lives elsewhere.

Advanced Personalization and Machine Learning

The notification service will **not** implement machine learning-driven personalization, send time optimization, or content recommendation systems. It provides basic template variable substitution but doesn't use recipient behavior data to automatically customize message content or timing.

ML-driven personalization requires:

- Large-scale behavioral data collection and storage
- Machine learning model training and deployment infrastructure
- Real-time inference capabilities for content selection
- Feedback loops to measure personalization effectiveness
- Privacy controls and data governance for ML training data

These capabilities require specialized ML infrastructure and data science expertise that goes far beyond notification delivery. Organizations that need advanced personalization should integrate dedicated personalization engines (Dynamic Yield, Adobe Target) with the notification service.

The service provides template variables that personalization systems can populate, but it doesn't implement the intelligence to determine what personalized content to show each user.

> **Decision: Service Boundary Definition**
>
> - **Context**: Notification systems could expand into adjacent domains like user management, campaign management, and personalization
> - **Options Considered**: Build comprehensive notification platform, focus solely on reliable delivery, hybrid approach with optional advanced features
> - **Decision**: Focus exclusively on multi-channel delivery, templating, preferences, and tracking
> - **Rationale**: Reliable message delivery is complex enough to warrant dedicated focus. Integration with existing best-of-breed tools in adjacent domains provides more value than building inferior replacements.
> - **Consequences**: Requires clean API design for integration with external systems, but enables organizations to choose specialized tools for each domain while maintaining notification delivery reliability

Success Metrics

The notification service's success will be measured against specific, measurable criteria that align with the functional goals:

Goal Area	Primary Metric	Target	Secondary Metrics
Multi-Channel Delivery	Channel abstraction adoption	100% of notifications use unified API	Provider onboarding time < 2 days
Intelligent Routing	Delivery success rate	>99% for urgent notifications	Fallback activation rate, cost per notification
Template System	Template usage rate	>80% of notifications use templates	Template update frequency, localization coverage
User Preferences	Preference compliance rate	100% respect for opt-out preferences	Unsubscribe response time, preference update frequency
Delivery Tracking	Tracking coverage	100% of notifications tracked end-to-end	Alert response time, analytics dashboard usage

These metrics provide concrete validation that the notification service delivers value to both application developers (simplified integration) and end users (better notification experiences).

Implementation Guidance

The goals and non-goals translate into specific technology choices and implementation approaches that guide development decisions throughout the project.

Technology Recommendations

Component	Simple Option	Advanced Option
API Framework	HTTP REST with `net/http` and `gorilla/mux`	gRPC with Protocol Buffers for internal services
Message Queue	RabbitMQ with `amqp` library	Apache Kafka with `shopify/sarama` for high throughput
Template Engine	`mustache` library for logic-less templates	Custom template engine with security sandboxing
Database	PostgreSQL with `database/sql` and `lib/pq`	PostgreSQL with `gorm` ORM for rapid development
Configuration	YAML files with `gopkg.in/yaml.v3`	Consul or etcd for dynamic configuration

Recommended Module Structure

The project should organize code around the functional goal areas to maintain clear separation of concerns:

```
notification-service/
  cmd/
    notification-server/main.go     ← HTTP API server entry point
    worker/main.go                  ← Background notification processor
  internal/
    api/                            ← REST API handlers and routing
      handlers.go
      middleware.go
      router.go
    channels/                       ← Channel abstraction (Milestone 1)
      channel.go                    ← Channel interface definition
      email/sendgrid.go             ← Email channel implementation
      sms/twilio.go                 ← SMS channel implementation
      push/fcm.go                   ← Push notification implementation
      inapp/websocket.go            ← In-app notification implementation
    routing/                        ← Intelligent routing engine
      router.go
      fallback.go
      circuit_breaker.go
    templates/                      ← Template system (Milestone 2)
      engine.go
      localization.go
      versioning.go
    preferences/                    ← User preference management (Milestone 3)
      store.go
      unsubscribe.go
      quiet_hours.go
    tracking/                       ← Delivery tracking (Milestone 4)
      tracker.go
      analytics.go
      webhooks.go
    queue/                          ← Message queue abstraction
      publisher.go
      consumer.go
  pkg/
    models/                         ← Shared data structures
      notification.go
      user.go
      template.go
  configs/
    development.yaml                ← Development environment configuration
    production.yaml                 ← Production environment configuration
```

This structure maps directly to the functional goals and milestones, making it easy to locate code related to specific features.

Core Data Structures

Define the primary data types that support the functional goals:

```go
// NotificationMessage represents a notification request submitted to the service
type NotificationMessage struct {
    ID          string            `json:"id"`
    RecipientID string            `json:"recipient_id"`
    TemplateID  string            `json:"template_id"`
    Priority    Priority          `json:"priority"`
    Variables   map[string]string `json:"variables"`
    Category    string            `json:"category"`
    CreatedAt   time.Time         `json:"created_at"`
}

// User represents a notification recipient with contact information and preferences
type User struct {
    ID               string                `json:"id"`
    Email            string                `json:"email,omitempty"`
    PhoneNumber      string                `json:"phone_number,omitempty"`
    DeviceTokens     []string              `json:"device_tokens,omitempty"`
    Locale           string                `json:"locale"`
    Preferences      UserPreference        `json:"preferences"`
    QuietHoursStart  string                `json:"quiet_hours_start"`
    QuietHoursEnd    string                `json:"quiet_hours_end"`
    Timezone         string                `json:"timezone"`
}

// Template represents a notification template with localized content
type Template struct {
    ID          string                     `json:"id"`
    Version     int                        `json:"version"`
    Name        string                     `json:"name"`
    Category    string                     `json:"category"`
    Content     map[string]TemplateContent `json:"content"` // locale -> content
    CreatedAt   time.Time                  `json:"created_at"`
    UpdatedAt   time.Time                  `json:"updated_at"`
}
```

Milestone Checkpoints

After implementing each milestone, validate the functional goals through specific tests:

Milestone 1 Checkpoint (Channel Abstraction):

```bash
# Test unified channel interface
curl -X POST localhost:8080/api/notifications \
  -H "Content-Type: application/json" \
  -d '{
    "recipient_id": "user123",
    "template_id": "welcome",
    "priority": "normal",
    "variables": {"name": "John"}
  }'

# Verify: Notification routes to appropriate channel based on user preferences
# Expected: Response includes notification ID and selected channel
# Expected: Background worker processes notification through correct provider
```

Milestone 2 Checkpoint (Templates):

```bash
# Test template rendering with localization
curl -X POST localhost:8080/api/templates/welcome/preview \
  -H "Content-Type: application/json" \
  -d '{
    "locale": "es",
    "variables": {"name": "María"},
    "channel": "email"
  }'

# Verify: Template renders with Spanish content and proper variable substitution
# Expected: HTML and plain text versions with locale-specific formatting
```

Milestone 3 Checkpoint (User Preferences):

```bash
# Test preference enforcement
curl -X PUT localhost:8080/api/users/user123/preferences \
  -H "Content-Type: application/json" \
  -d '{
    "channels": {"sms": false, "email": true},
    "categories": {"marketing": false, "transactional": true}
  }'

# Verify: Marketing notifications skip SMS and email for this user
# Verify: Transactional notifications still deliver via email
```

Milestone 4 Checkpoint (Delivery Tracking):

```bash
# Test delivery status tracking
curl -X GET localhost:8080/api/notifications/notif123/status

# Verify: Status progression from queued → sent → delivered
# Verify: Webhook processing updates status automatically
# Expected: Analytics dashboard shows delivery metrics
```

Common Implementation Pitfalls

⚠️ **Pitfall: Synchronous Provider API Calls** Calling provider APIs (SendGrid, Twilio) synchronously in HTTP request handlers will cause request timeouts and poor user experience. Provider APIs can take seconds to respond or fail entirely.

Fix: Use message queues for asynchronous processing. HTTP API accepts notifications and returns immediately after queueing. Background workers handle provider integration.

⚠️ **Pitfall: Ignoring Provider Rate Limits** Each notification provider has different rate limits (SendGrid: 600 emails/minute, Twilio SMS: 1 message/second). Exceeding limits causes API errors and delayed delivery.

Fix: Implement per-provider rate limiting with token bucket algorithms. Queue notifications that exceed rate limits for later processing.

⚠️ **Pitfall: Unsafe Template Variable Substitution** Directly interpolating user data into HTML email templates creates XSS vulnerabilities. Malicious users could inject JavaScript that executes when recipients open emails.

Fix: Use template engines with automatic HTML escaping. Validate and sanitize all template variables before substitution.

⚠️ **Pitfall: Blocking on User Preference Lookups** Looking up user preferences synchronously for each notification creates database bottlenecks and increases latency.

Fix: Cache user preferences in Redis with TTL expiration. Accept eventual consistency for preference updates to maintain performance.

The goals and non-goals provide the foundation for making consistent decisions throughout implementation. When faced with feature requests or scope creep, refer back to these boundaries to maintain focus on delivering reliable multi-channel notification delivery.

High-Level Architecture

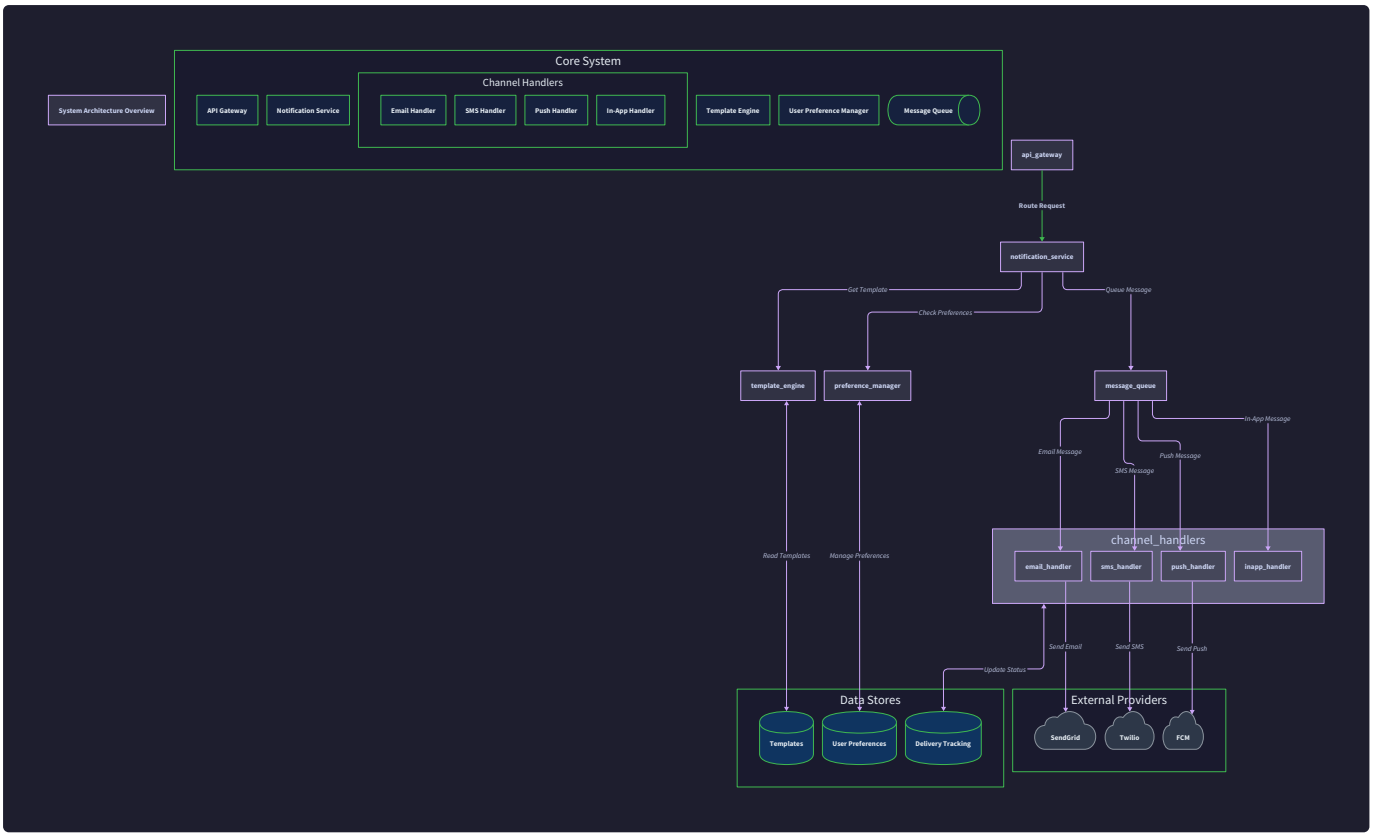
> **Milestone(s):** Foundation for all milestones — establishes the core system structure and component boundaries that support channel abstraction, template processing, user preferences, and delivery tracking

Think of our notification system like a modern postal service. The postal service doesn't just deliver mail — it has sorting facilities that route mail based on addresses and service levels, customer service centers that handle preferences and forwarding requests, printing facilities that format different types of mail, and tracking systems that monitor every package from pickup to delivery. Similarly, our notification system needs multiple specialized components working together: a routing engine that decides which channel to use, preference managers that respect user choices, template engines that format content appropriately for each channel, and tracking systems that monitor delivery status.

The key architectural challenge is designing a system where each component has clear responsibilities but can work together seamlessly. A poorly designed notification system often results in tightly coupled code where adding a new channel requires touching every component, or where user preferences are scattered across multiple systems making them impossible to manage consistently.

Component Overview

Our notification system consists of six primary components, each with distinct responsibilities and clear interfaces. The architecture follows a layered approach where higher-level components orchestrate business logic while lower-level components handle specific technical concerns like message formatting or external API communication.



The **Notification Gateway** serves as the system's front door and primary orchestrator. Think of it as the postal service's main reception desk where all mail gets processed and routed. When a client submits a notification request, the gateway validates the request structure, enriches it with additional metadata like priority levels and timestamps, and then coordinates with other components to process the notification. The gateway maintains the high-level workflow state and ensures that each notification follows the complete processing pipeline from submission to delivery tracking.

Responsibility	Description	Key Interfaces
Request Validation	Verify notification requests have required fields and valid formats	`ValidateNotification(msg *NotificationMessage) error`
Workflow Orchestration	Coordinate template rendering, routing, and delivery tracking	`ProcessNotification(ctx context.Context, msg *NotificationMessage) error`
Priority Management	Apply priority-based routing and rate limiting rules	`ApplyPriorityRules(msg *NotificationMessage, user *User) Priority`
Error Coordination	Handle failures and coordinate retry strategies across components	`HandleFailure(notificationID string, err error) error`

The **Template Engine** functions as the content formatting and localization center. Like a printing press that can produce the same document in multiple formats and languages, the template engine takes raw notification data and transforms it into properly formatted content for each delivery channel. It handles variable substitution, localization based on user locale preferences, and channel-specific formatting requirements. For instance, the same promotional notification might become an HTML email with rich formatting, a plain text SMS message under 160 characters, and a push notification with a title and body structure.

Component Function	Input	Output	Error Conditions
Template Loading	Template ID, version, locale	`Template` struct with content	Template not found, version mismatch
Variable Substitution	Template content, variable map	Rendered content string	Missing variables, malformed template
Localization	User locale, template variants	Localized template content	Unsupported locale, fallback chain exhausted
Channel Formatting	Rendered content, target channel	Channel-specific formatted message	Content too long, invalid HTML, XSS detected

The **User Preference Manager** acts as the system's customer service center, maintaining all user preferences and consent decisions. This component understands the complex hierarchy of user choices: global opt-outs override everything, category preferences filter by notification types, channel preferences control delivery methods, and quiet hours provide time-based filtering. The preference manager also handles compliance requirements like CAN-SPAM unsubscribe rules and GDPR consent management.

Preference Type	Scope	Priority Level	Override Rules
Global Opt-out	All notifications	Highest	Blocks everything except legally required transactional messages
Category Preferences	Marketing, Transactional, Alerts	High	Applied after global opt-out check
Channel Preferences	Email, SMS, Push, In-app	Medium	Applied per-channel after category filtering
Quiet Hours	Time-based suppression	Low	Only blocks non-urgent notifications during specified hours

The **Channel Abstraction Layer** provides a unified interface for communicating with diverse external notification providers. Think of this as a universal translator that speaks the native language of each provider while presenting a consistent interface to the rest of the system. Each channel implementation knows how to authenticate with its provider, format messages according to provider requirements, handle provider-specific errors, and parse delivery status responses.

Channel Type	Provider Examples	Message Format	Key Capabilities
Email	SendGrid, Mailgun, AWS SES	MIME with headers, HTML/text body	Rich content, attachments, tracking pixels
SMS	Twilio, AWS SNS, Nexmo	Plain text under 160 chars	High deliverability, immediate delivery
Push	FCM, APNs, OneSignal	JSON with title/body/data	Real-time delivery, rich notifications
In-app	WebSocket, Server-sent events	Custom JSON payload	Immediate visibility, interactive content

The **Routing Engine** makes intelligent decisions about which channels to use for each notification. Like a postal service routing system that considers delivery speed, cost, and reliability, our routing engine evaluates user preferences, channel availability, rate limits, and fallback strategies. It implements circuit breaker patterns to detect provider outages and automatically switch to backup channels when primary channels fail.

Routing Decision Factor	Weight	Evaluation Criteria	Impact on Channel Selection
User Preferences	Highest	Explicit user channel selections	Can completely block channels
Notification Priority	High	Urgent vs normal vs low priority	Affects channel order and fallback strategy
Channel Health	High	Circuit breaker state, recent failure rates	Routes around unhealthy providers
Rate Limits	Medium	Per-channel quotas and throttling	Delays or reroutes high-volume notifications
Cost Optimization	Low	Provider pricing, message size	Prefers cheaper channels when appropriate

The **Delivery Tracking System** monitors the complete lifecycle of each notification from submission to final delivery status. This system processes webhooks from external providers, updates delivery status records, calculates analytics metrics, and triggers alerts when delivery rates fall below acceptable thresholds. It handles the complexity of different provider webhook formats and timing while presenting a unified view of delivery status to the rest of the system.

Status Tracking Phase	Data Sources	Metrics Calculated	Alerting Triggers
Submission	Internal request logs	Submission rate, validation errors	Unusual spike in invalid requests
Routing	Routing engine decisions	Channel distribution, fallback usage	High fallback rate indicating primary channel issues
Provider Delivery	External webhooks	Delivery rate, bounce rate, latency	Delivery rate below threshold, high bounce rate
Engagement	Open/click tracking	Open rate, click-through rate	Sudden drop in engagement metrics

> **Design Principle: Separation of Concerns** Each component owns a single domain of responsibility. The template engine never makes routing decisions, the routing engine never formats content, and the preference manager never tracks delivery status. This separation makes the system easier to test, debug, and extend. When you need to add a new notification channel, you only touch the channel abstraction layer. When you need to add a new template variable, you only modify the template engine.

Message Flow Architecture: The components communicate through a combination of synchronous API calls for real-time operations and asynchronous message queues for potentially long-running or high-volume operations. Notification submission and routing happen synchronously to provide immediate feedback to clients, while template rendering, actual delivery, and status tracking happen asynchronously to handle provider latency and temporary failures gracefully.

> **Decision: Hybrid Synchronous/Asynchronous Processing**
>
> - **Context**: Notification processing involves both real-time user-facing operations (validation, routing decisions) and potentially slow external operations (provider API calls, webhook processing)
> - **Options Considered**: Fully synchronous processing, fully asynchronous processing, hybrid approach
> - **Decision**: Hybrid approach with synchronous request validation and routing, asynchronous delivery and tracking
> - **Rationale**: Provides immediate feedback to clients while handling provider latency and failures gracefully through queue-based retry mechanisms
> - **Consequences**: Enables better user experience and system resilience, but requires more complex error handling and status communication

Recommended Module Structure

The codebase organization reflects the component architecture while following Go's conventional project layout. Each major component becomes its own internal package with clear dependencies and interfaces. This structure supports both development team organization and deployment flexibility.

```
notification-service/
├── cmd/
│   ├── server/
│   │   └── main.go                          ← HTTP server entry point
│   ├── worker/
│   │   └── main.go                          ← Background worker entry point
│   └── cli/
│       └── main.go                          ← Administrative CLI tools
├── internal/
│   ├── config/
│   │   ├── config.go                        ← Config struct and loading
│   │   ├── database.go                      ← DatabaseConfig
│   │   ├── queue.go                         ← QueueConfig
│   │   └── server.go                        ← ServerConfig
│   ├── gateway/
│   │   ├── gateway.go                       ← NotificationGateway implementation
│   │   ├── validator.go                     ← Request validation logic
│   │   ├── orchestrator.go                  ← Workflow orchestration
│   │   └── gateway_test.go
│   ├── channels/
│   │   ├── interface.go                     ← Channel interface definition
│   │   ├── email/
│   │   │   ├── sendgrid.go                  ← SendGrid implementation
│   │   │   ├── smtp.go                      ← Generic SMTP implementation
│   │   │   └── email_test.go
│   │   ├── sms/
│   │   │   ├── twilio.go                    ← Twilio SMS implementation
│   │   │   └── sms_test.go
│   │   ├── push/
│   │   │   ├── fcm.go                       ← Firebase Cloud Messaging
│   │   │   ├── apns.go                      ← Apple Push Notification Service
│   │   │   └── push_test.go
│   │   ├── inapp/
│   │   │   ├── websocket.go                 ← WebSocket-based in-app notifications
│   │   │   └── inapp_test.go
│   │   └── channels_test.go                 ← Integration tests across channels
│   ├── routing/
│   │   ├── engine.go                        ← RoutingEngine implementation
│   │   ├── rules.go                         ← Routing rule evaluation
│   │   ├── circuit_breaker.go               ← Circuit breaker for provider failures
│   │   ├── fallback.go                      ← Fallback strategy implementation
│   │   └── routing_test.go
│   ├── templates/
│   │   ├── engine.go                        ← TemplateEngine implementation
│   │   ├── renderer.go                      ← Variable substitution and rendering
│   │   ├── localization.go                  ← Multi-language support
│   │   ├── versioning.go                    ← Template version management
│   │   ├── sanitizer.go                     ← XSS protection and content sanitization
│   │   └── templates_test.go
│   ├── preferences/
│   │   ├── manager.go                       ← UserPreferenceManager implementation
│   │   ├── storage.go                       ← Preference persistence layer
│   │   ├── unsubscribe.go                   ← Unsubscribe token generation and validation
│   │   ├── quiet_hours.go                   ← Time-based preference logic
│   │   └── preferences_test.go
│   ├── tracking/
│   │   ├── tracker.go                       ← DeliveryTracker implementation
│   │   ├── webhook_handler.go               ← Provider webhook processing
│   │   ├── analytics.go                     ← Metrics calculation and aggregation
│   │   ├── pixel_tracker.go                 ← Email open tracking implementation
│   │   └── tracking_test.go
│   ├── queue/
│   │   ├── publisher.go                     ← Publisher implementation
│   │   ├── consumer.go                      ← Message consumer for workers
│   │   ├── retry.go                         ← Retry logic and dead letter queues
│   │   └── queue_test.go
│   ├── storage/
│   │   ├── postgres/
│   │   │   ├── notifications.go             ← Notification persistence
│   │   │   ├── users.go                     ← User data persistence
│   │   │   ├── templates.go                 ← Template storage
│   │   │   ├── preferences.go               ← User preference storage
│   │   │   ├── delivery_records.go          ← Delivery tracking storage
```

```
│   │   │   └── migrations/               ← Database schema migrations
│   │   └── redis/
│   │       ├── cache.go                  ← Caching layer for templates and preferences
│   │       └── session.go                ← Session storage for rate limiting
│   ├── api/
│   │   ├── handlers/
│   │   │   ├── notifications.go          ← REST API handlers for notification submission
│   │   │   ├── preferences.go            ← User preference management endpoints
│   │   │   ├── templates.go              ← Template management endpoints
│   │   │   ├── webhooks.go               ← Provider webhook endpoints
│   │   │   └── health.go                 ← Health check and status endpoints
│   │   ├── middleware/
│   │   │   ├── auth.go                   ← Authentication middleware
│   │   │   ├── rate_limit.go             ← API rate limiting
│   │   │   ├── logging.go                ← Request/response logging
│   │   │   └── cors.go                   ← CORS handling
│   │   └── routes.go                     ← HTTP route configuration
│   └── types/
│       ├── notification.go               ← NotificationMessage, Notification types
│       ├── user.go                       ← User, UserPreference types
│       ├── template.go                   ← Template, TemplateContent types
│       ├── enums.go                      ← Priority, NotificationStatus enums
│       └── errors.go                     ← Custom error types
├── pkg/                                  ← Public packages (if needed for external integrations)
├── web/                                  ← Static web assets for admin dashboard
│   ├── static/
│   └── templates/
├── docs/                                 ← API documentation
│   ├── api.yaml                          ← OpenAPI specification
│   └── examples/                         ← Usage examples
├── scripts/
│   ├── migrate.sh                        ← Database migration script
│   ├── seed.sh                           ← Test data seeding
│   └── docker/
│       ├── Dockerfile
│       └── docker-compose.yml
├── go.mod
├── go.sum
├── Makefile
└── README.md
```

Package Dependency Rules: The module structure enforces clear dependency relationships to prevent circular imports and maintain architectural boundaries. The gateway package coordinates between other components but doesn't implement business logic specific to templates, routing, or tracking. Channel implementations depend only on the channel interface and external provider SDKs, never on other internal components.

Package	Can Import	Cannot Import	Rationale
`gateway`	`routing`, `templates`, `preferences`, `tracking`, `queue`	`channels`, `storage`	Orchestrates but doesn't implement specific business logic
`channels`	`types`	All other internal packages	Maintains clean channel abstraction
`routing`	`preferences`, `channels`, `types`	`templates`, `tracking`, `storage`	Makes routing decisions based on preferences and channel health
`templates`	`storage`, `types`	`channels`, `routing`, `preferences`	Focuses purely on content rendering and localization
`preferences`	`storage`, `types`	`channels`, `routing`, `templates`	Manages user preferences without knowing about other business logic
`tracking`	`storage`, `types`	`channels`, `routing`, `templates`	Tracks delivery status independently of other components

Configuration Management Strategy: Each component's configuration lives in its own sub-package under `internal/config`, but the main `Config` struct aggregates all component configurations. This allows components to own their configuration schema while supporting both environment-based and file-based configuration loading.

Configuration Area	Environment Variables	Config File Section	Override Priority
Server Settings	`PORT`, `HOST`, `TLS_CERT_PATH`	`server.port`, `server.host`	Environment > File > Defaults
Database Connection	`DATABASE_URL`, `DB_POOL_SIZE`	`database.url`, `database.pool_size`	Environment > File > Defaults
Queue Configuration	`RABBITMQ_URL`, `QUEUE_EXCHANGE`	`queue.url`, `queue.exchange`	Environment > File > Defaults
Provider Credentials	`SENDGRID_API_KEY`, `TWILIO_TOKEN`	Never stored in files (security)	Environment only

Testing Strategy Integration: The module structure supports multiple levels of testing. Unit tests live alongside their implementation files using Go's conventional `_test.go` suffix. Integration tests that span multiple components live in dedicated test packages, while end-to-end tests use the `cmd/` entry points to test complete workflows.

Test Level	Location	Purpose	External Dependencies
Unit Tests	`internal/*/package_test.go`	Test individual component logic	Mocked or minimal
Integration Tests	`tests/integration/`	Test component interactions	Test database, test queues
End-to-End Tests	`tests/e2e/`	Test complete notification flows	All real dependencies
Performance Tests	`tests/performance/`	Load testing and benchmarks	Production-like environment

Deployment Flexibility: The structure supports multiple deployment patterns. The `cmd/server` and `cmd/worker` entry points can be deployed as separate services for horizontal scaling, or combined into a single binary for simpler deployments. The clear component boundaries make it easy to extract individual components into microservices later if needed.

Implementation Guidance

A. Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Server	`net/http` with `gorilla/mux` for routing	`gin-gonic/gin` with middleware ecosystem
Message Queue	RabbitMQ with `streadway/amqp`	Apache Kafka with `segmentio/kafka-go`
Database	PostgreSQL with `lib/pq`	PostgreSQL with `jackc/pgx` connection pooling
Caching	Redis with `go-redis/redis`	Redis Cluster with `go-redis/redis` cluster support
Configuration	`spf13/viper` for file/env loading	`spf13/viper` + `spf13/cobra` for CLI
Logging	`sirupsen/logrus` structured logging	`uber-go/zap` high-performance logging
Testing	Built-in `testing` + `testify/assert`	`testify/suite` + `gomega` for BDD-style

B. Core Infrastructure Starter Code:

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

package config

import (
    "fmt"
    "time"

    "github.com/spf13/viper"
)

type Config struct {
    Server   ServerConfig   `mapstructure:"server"`
    Database DatabaseConfig `mapstructure:"database"`
    Queue    QueueConfig    `mapstructure:"queue"`
    Redis    RedisConfig    `mapstructure:"redis"`
}

type ServerConfig struct {
    Host         string        `mapstructure:"host"`
    Port         int           `mapstructure:"port"`
    ReadTimeout  time.Duration `mapstructure:"read_timeout"`
    WriteTimeout time.Duration `mapstructure:"write_timeout"`
    TLSCertPath  string        `mapstructure:"tls_cert_path"`
    TLSKeyPath   string        `mapstructure:"tls_key_path"`
}

type DatabaseConfig struct {
    URL             string        `mapstructure:"url"`
    MaxOpenConns    int           `mapstructure:"max_open_conns"`
    MaxIdleConns    int           `mapstructure:"max_idle_conns"`
    ConnMaxLifetime time.Duration `mapstructure:"conn_max_lifetime"`
}

type QueueConfig struct {
    URL         string `mapstructure:"url"`
    Exchange    string `mapstructure:"exchange"`
    RoutingKey  string `mapstructure:"routing_key"`
    Durable     bool   `mapstructure:"durable"`
    AutoDelete  bool   `mapstructure:"auto_delete"`
```

```go
}

type RedisConfig struct {
    Address     string        `mapstructure:"address"`
    Password    string        `mapstructure:"password"`
    DB          int           `mapstructure:"db"`
    DialTimeout time.Duration `mapstructure:"dial_timeout"`
}

func LoadConfig(path string) (*Config, error) {
    config := &Config{}

    viper.SetConfigFile(path)
    viper.SetEnvPrefix("NOTIF")
    viper.AutomaticEnv()

    // Set defaults
    viper.SetDefault("server.host", "localhost")
    viper.SetDefault("server.port", 8080)
    viper.SetDefault("server.read_timeout", "30s")
    viper.SetDefault("server.write_timeout", "30s")
    viper.SetDefault("database.max_open_conns", 25)
    viper.SetDefault("database.max_idle_conns", 5)
    viper.SetDefault("database.conn_max_lifetime", "5m")
    viper.SetDefault("queue.exchange", "notifications")
    viper.SetDefault("queue.routing_key", "notification.send")
    viper.SetDefault("queue.durable", true)
    viper.SetDefault("redis.db", 0)
    viper.SetDefault("redis.dial_timeout", "5s")

    if err := viper.ReadInConfig(); err != nil {
        return nil, fmt.Errorf("failed to read config file: %w", err)
    }

    if err := viper.Unmarshal(config); err != nil {
        return nil, fmt.Errorf("failed to unmarshal config: %w", err)
    }
```

```
    return config, nil

}
```

```go
// internal/queue/publisher.go

package queue

import (
    "context"
    "encoding/json"
    "fmt"
    "time"

    "github.com/streadway/amqp"
)

type Publisher struct {
    conn     *amqp.Connection
    channel  *amqp.Channel
    exchange string
}

func NewPublisher(url, exchange string) (*Publisher, error) {
    conn, err := amqp.Dial(url)
    if err != nil {
        return nil, fmt.Errorf("failed to connect to RabbitMQ: %w", err)
    }

    ch, err := conn.Channel()
    if err != nil {
        conn.Close()
        return nil, fmt.Errorf("failed to open channel: %w", err)
    }

    err = ch.ExchangeDeclare(
        exchange,  // name
        "topic",   // type
        true,      // durable
        false,     // auto-deleted
        false,     // internal
        false,     // no-wait
        nil,       // arguments
```

```
    )
    if err != nil {
        ch.Close()
        conn.Close()
        return nil, fmt.Errorf("failed to declare exchange: %w", err)
    }

    return &Publisher{
        conn:     conn,
        channel:  ch,
        exchange: exchange,
    }, nil
}

func (p *Publisher) PublishNotification(ctx context.Context, msg interface{}) error {
    body, err := json.Marshal(msg)
    if err != nil {
        return fmt.Errorf("failed to marshal message: %w", err)
    }

    return p.channel.Publish(
        p.exchange,             // exchange
        "notification.send",    // routing key
        false,                  // mandatory
        false,                  // immediate
        amqp.Publishing{
            ContentType:  "application/json",
            Body:         body,
            Timestamp:    time.Now(),
            DeliveryMode: amqp.Persistent, // persist messages
        },
    )
}

func (p *Publisher) Close() error {
    if p.channel != nil {
        p.channel.Close()
    }
```

```
    if p.conn != nil {
        return p.conn.Close()
    }
    return nil
}
```

C. Core Component Interface Skeletons:

```go
// internal/types/notification.go

package types

import (
    "time"
)

type Priority int

const (
    PriorityLow Priority = iota
    PriorityNormal
    PriorityHigh
    PriorityUrgent
)

type NotificationStatus int

const (
    StatusQueued NotificationStatus = iota
    StatusSending
    StatusSent
    StatusDelivered
    StatusFailed
    StatusBounced
    StatusOpened
    StatusClicked
)

type NotificationMessage struct {
    ID          string            `json:"id"`
    RecipientID string            `json:"recipient_id"`
    TemplateID  string            `json:"template_id"`
    Priority    Priority          `json:"priority"`
    Variables   map[string]string `json:"variables"`
    Category    string            `json:"category"`
    CreatedAt   time.Time         `json:"created_at"`
}

type User struct {
```

```
    ID              string         `json:"id"`
    Email           string         `json:"email"`
    PhoneNumber     string         `json:"phone_number"`
    DeviceTokens    []string       `json:"device_tokens"`
    Locale          string         `json:"locale"`
    Preferences     UserPreference `json:"preferences"`
    QuietHoursStart string         `json:"quiet_hours_start"` // "22:00"
    QuietHoursEnd   string         `json:"quiet_hours_end"`   // "08:00"
    Timezone        string         `json:"timezone"`          // "America/New_York"
}

type UserPreference struct {
    GlobalOptOut     bool                   `json:"global_opt_out"`
    EmailEnabled     bool                   `json:"email_enabled"`
    SMSEnabled       bool                   `json:"sms_enabled"`
    PushEnabled      bool                   `json:"push_enabled"`
    InAppEnabled     bool                   `json:"in_app_enabled"`
    Categories       map[string]bool        `json:"categories"` // category -> enabled
}
```

```go
// internal/gateway/gateway.go

package gateway

import (
    "context"


    "github.com/yourorg/notification-service/internal/types"
)

type NotificationGateway struct {
    // Dependencies will be injected
}

// ProcessNotification coordinates the complete notification workflow.
// This is the main entry point that orchestrates validation, routing,
// template rendering, and delivery tracking.
func (ng *NotificationGateway) ProcessNotification(ctx context.Context, msg *types.NotificationMessage) error {
    // TODO 1: Validate the notification message structure and required fields
    // TODO 2: Load user details and preferences from storage
    // TODO 3: Check if user preferences allow this notification (global opt-out, category, etc.)
    // TODO 4: Apply quiet hours filtering based on user timezone
    // TODO 5: Route to appropriate channels based on user preferences and notification priority
    // TODO 6: Queue the notification for asynchronous processing
    // TODO 7: Return immediate response while processing continues in background


    panic("implement me")
}
```

D. Milestone Checkpoints:

After implementing the high-level architecture:

1. **Verify Configuration Loading**: Run `go run cmd/server/main.go` with a config file - should start without errors
2. **Test Component Interfaces**: Run `go test ./internal/types/...` - all type definitions should compile
3. **Validate Module Structure**: Check that imports work correctly across packages without circular dependencies
4. **Queue Connection**: Verify `Publisher` can connect to RabbitMQ and declare exchanges
5. **HTTP Server Startup**: Server should bind to configured port and respond to health checks

Expected output when starting the server:

```
2024/01/01 12:00:00 INFO Configuration loaded successfully file=config.yaml
2024/01/01 12:00:00 INFO Connected to database host=localhost database=notifications
2024/01/01 12:00:00 INFO Connected to message queue exchange=notifications
2024/01/01 12:00:00 INFO HTTP server starting host=localhost port=8080
```

E. Language-Specific Recommendations:

- Use `context.Context` for all long-running operations and request scoping
- Leverage Go's interface composition - small interfaces are easier to test and mock
- Use `embed` package for static assets like email templates in the binary
- Apply Go's error wrapping with `fmt.Errorf("context: %w", err)` for better error traces
- Use `sync.Pool` for frequently allocated objects like template rendering contexts
- Implement `io.Closer` interface for components that need cleanup (database connections, queues)

F. Common Implementation Pitfalls:

⚠️ **Pitfall: Circular Package Dependencies** When implementing component interactions, avoid importing packages that import back to your package. For example, don't let the `channels` package import `routing` if `routing` already imports `channels`. Use dependency injection and interfaces to break cycles.

⚠️ **Pitfall: Missing Context Cancellation** Always check `ctx.Done()` in long-running operations and pass context through all function calls. Without proper context handling, the service won't shut down gracefully and may leave goroutines running.

⚠️ **Pitfall: Forgetting to Close Resources** Database connections, message queue channels, and HTTP client connections must be properly closed. Implement the `io.Closer` interface on components that manage resources and ensure `defer Close()` is called.

Data Model

> **Milestone(s):** Foundation for all milestones — establishes core data structures that support channel abstraction (Milestone 1), template processing (Milestone 2), user preferences (Milestone 3), and delivery tracking (Milestone 4)

Think of the data model as the **blueprint for a modern logistics company**. Just as FedEx needs to track packages, customers, delivery routes, and shipping preferences in their system, our notification service needs to model notifications, users, templates, and preferences. The data model defines the "nouns" of our system — the entities that exist and how they relate to each other. Without a solid data model, we'd be like a shipping company trying to deliver packages without knowing what a package is, who the customers are, or where anything should go.

The data model serves as the foundation that all four milestones build upon. Channel abstraction needs to know what notification channels users prefer. Template processing requires structured template definitions with localization support. User preferences need granular controls over channels and categories. Delivery tracking must record status changes throughout the notification lifecycle. Each of these capabilities depends on having well-designed data structures that capture the essential attributes and relationships.


Data Model Relationships
NotificationMessage
- "id string"
- "user_id string"
- "template_id string"
- "channel Channel"
- "priority Priority"
- "status Status"
- "created_at timestamp"
- "scheduled_at timestamp"
+ "send() void"
+ "get_status() Status"
uses
instantiated as
creates
Template
- "id string"
- "name string"
- "category string"
- "subject string"
- "body_template string"
- "supported_channels Channel list"
- "variables string list"
- "created_at timestamp"
+ "render(variables Map): string"
+ "validate() boolean"
belongs to
DeliveryRecord
- "id string"
- "notification_id string"
- "user_id string"
- "channel Channel"
- "status DeliveryStatus"
- "provider string"
- "external_id string"
- "attempts int"
- "delivered_at timestamp"
- "error_message string"
+ "update_status(status DeliveryStatus): void"
+ "retry() void"
delivered to
User
- "id string"
- "email string"
- "phone string"
- "timezone string"
- "locale string"
- "created_at timestamp"
+ "get_preferences() UserPreference list"
+ "update_profile(data UserData): void"

Our notification service data model centers around five core entity types, each serving a specific purpose in the overall system architecture:

Core Entities and Their Roles:

Entity	Primary Responsibility	Key Relationships
`NotificationMessage`	Represents a notification request submitted to the system	Links to User (recipient) and Template (content structure)
`User`	Stores recipient information and delivery preferences	Contains UserPreference and links to DeliveryRecord
`Template`	Defines reusable notification content with localization	Used by NotificationMessage, versioned for safe updates
`UserPreference`	Controls which channels and categories a user receives	Embedded within User, consulted during routing decisions
`DeliveryRecord`	Tracks notification status through the delivery pipeline	Links back to NotificationMessage for status updates

The relationships between these entities form a coherent system where notifications flow from templates through user preferences to delivery tracking. This design supports both the immediate needs of sending notifications and the long-term requirements of analytics, compliance, and user experience management.

Notification and User Types

The notification and user types form the core operational entities of our system. Think of `NotificationMessage` as a **shipping order** — it contains all the information needed to process and deliver a notification, including who should receive it, what template to use, and any custom data for personalization. The `User` type acts like a **customer profile** in our logistics analogy, storing not just contact information but also delivery preferences and service options.

> **Decision: Separate NotificationMessage from Delivery Tracking**
>
> - **Context**: We need to distinguish between the notification request (what the client wants to send) and the delivery process (how our system handles it)
> - **Options Considered**:
>   1. Single notification entity with embedded delivery status
>   2. Separate NotificationMessage (request) and DeliveryRecord (tracking)
>   3. Event-sourced approach with notification events
> - **Decision**: Separate NotificationMessage from DeliveryRecord entities
> - **Rationale**: Clean separation allows the message to be immutable while delivery status evolves. Supports scenarios where one message generates multiple delivery attempts or channels. Simplifies querying and analytics.
> - **Consequences**: Requires foreign key relationships and joins, but provides better data integrity and clearer domain modeling

NotificationMessage Structure:

The `NotificationMessage` represents an individual notification request submitted to our system. This is the primary input entity that triggers the entire notification processing pipeline.

Field	Type	Description
ID	string	Unique identifier for this notification request, used for idempotency and tracking
RecipientID	string	Foreign key reference to the User who should receive this notification
TemplateID	string	Identifier for the template to use, combined with template version for content lookup
Priority	Priority	Enumeration controlling delivery urgency and routing behavior (PriorityLow, PriorityNormal, PriorityHigh, PriorityUrgent)
Variables	map[string]string	Key-value pairs for template variable substitution, such as user names or transaction amounts
Category	string	Classification for preference filtering (marketing, transactional, alerts, security)
CreatedAt	time.Time	Timestamp when the notification was submitted to the system

The `Priority` field drives both routing decisions and queue processing order. `PriorityUrgent` notifications bypass quiet hours and rate limits, ensuring critical messages like password resets always reach users. `PriorityLow` messages may be batched or delayed during high-traffic periods. The `Category` field enables users to opt out of marketing messages while still receiving transactional notifications, supporting compliance with email marketing regulations.

User Structure:

The `User` entity stores recipient information and delivery preferences. This is our customer profile that the routing engine consults when deciding how and when to deliver notifications.

Field	Type	Description
ID	string	Unique user identifier, typically matching your application's user ID
Email	string	Primary email address for email channel delivery
PhoneNumber	string	Mobile phone number for SMS channel delivery (E.164 format recommended)
DeviceTokens	[]string	Array of push notification tokens (FCM, APNs) for mobile app notifications
Locale	string	User's preferred language (RFC 5646 format, e.g., "en-US", "es-MX")
Preferences	UserPreference	Embedded preference settings controlling channel and category delivery
QuietHoursStart	string	Time when non-urgent notifications should be suppressed (24-hour format: "22:00")
QuietHoursEnd	string	Time when normal notification delivery resumes (24-hour format: "08:00")
Timezone	string	User's timezone for quiet hours calculation (IANA timezone identifier like "America/New_York")

The `DeviceTokens` array supports users with multiple devices — a phone, tablet, and desktop app can all receive push notifications for the same user. Tokens are managed by the client applications and updated through our preference management API when devices are added or removed.

> **Critical Design Insight**: The combination of `QuietHoursStart`, `QuietHoursEnd`, and `Timezone` enables respectful notification delivery. A user in Tokyo with quiet hours from 22:00 to 08:00 JST won't receive marketing emails at 3 AM local time, even if our servers are running in UTC. Only `PriorityUrgent` notifications bypass these settings.

Priority Enumeration:

The `Priority` type controls how urgently notifications should be delivered and which restrictions they can bypass.

Priority Level	Numeric Value	Behavior Description
PriorityLow	1	May be batched, delayed during high traffic, subject to aggressive rate limits
PriorityNormal	2	Standard delivery speed, respects all user preferences and quiet hours
PriorityHigh	3	Faster processing, may use premium provider routes, still respects opt-outs
PriorityUrgent	4	Immediate delivery, bypasses quiet hours and rate limits, cannot be opted out

Priority affects both message queue processing (urgent messages jump to the front) and channel selection (high priority might prefer faster but more expensive SMS over email). The routing engine uses priority to determine acceptable delivery delays and fallback strategies.

Concrete Walk-through Example:

Consider a password reset notification processing flow. A `NotificationMessage` is created with:

- `RecipientID`: "user_12345"
- `TemplateID`: "password_reset_v2"
- `Priority`: `PriorityUrgent`
- `Variables`: `{"reset_link": "https://app.com/reset?token=abc123", "user_name": "Alice"}`
- `Category`: "security"

The system looks up the User record for "user_12345" and finds their email is "alice@example.com", locale is "en-US", and they have quiet hours from 23:00 to 07:00 EST. Even though it's currently 2 AM EST, the `PriorityUrgent` designation bypasses quiet hours. The template engine loads the English version of the password reset template, substitutes the variables, and delivers via email immediately.

Template and Preference Types

Templates and preferences work together to control what content gets delivered and how users receive it. Think of templates as **form letters** that can be customized with specific information and translated into different languages. User preferences act like **subscription settings** and **delivery instructions** that determine which notifications a user wants and how they want to receive them.

> **Decision: Template Versioning Strategy**
>
> - **Context**: Templates need updates for content changes, localization, and bug fixes, but changing a template shouldn't break in-flight notifications
> - **Options Considered**:
>   1. Immutable templates with version numbers
>   2. Git-style template history with commit hashes
>   3. Simple template replacement with rollback capability
> - **Decision**: Immutable templates with integer version numbers
> - **Rationale**: Version numbers are simple to understand and increment. Immutability ensures notifications sent with template v3 always look the same, even after v4 is released. Supports gradual rollouts and instant rollbacks.
> - **Consequences**: Storage grows over time (old versions retained), but provides audit trail and consistency guarantees

Template Structure:

The `Template` entity defines reusable notification content that can be personalized with variables and localized for different languages.

Field	Type	Description
ID	string	Template identifier used in NotificationMessage.TemplateID (e.g., "welcome_email")
Version	int	Integer version number for this template revision (1, 2, 3...)
Name	string	Human-readable template name for management interfaces
Category	string	Template classification matching notification categories (marketing, transactional, alerts)
Content	map[string]TemplateContent	Localized content variants keyed by language code ("en", "es", "fr")
CreatedAt	time.Time	Timestamp when this template version was created
UpdatedAt	time.Time	Timestamp when this template version was last modified (for draft templates)

The `Content` map enables multi-language support where each language code maps to a `TemplateContent` structure containing the actual notification text. The template system supports fallback chains — if a user's locale is "es-MX" (Mexican Spanish) but only "es" (generic Spanish) and "en" (English) variants exist, the system will use "es" as the closest match.

TemplateContent Structure:

Each language variant of a template contains channel-specific content optimized for different delivery methods.

Field	Type	Description
Subject	string	Email subject line or SMS preview text with variable placeholders
HTMLBody	string	Rich HTML content for email delivery with mustache-style variables
TextBody	string	Plain text version for email clients that don't support HTML
SMSBody	string	Concise text for SMS delivery, must fit within character limits
PushTitle	string	Push notification title displayed in the notification shade
PushBody	string	Push notification body text with variable substitution

Different channels have different content requirements — emails can be rich HTML with images and styling, while SMS messages must be concise and fit within 160-character segments to avoid extra charges. Push notifications need both a title and body that render well in mobile notification interfaces.

Variable Substitution Format:

Templates use mustache-style syntax for variable placeholders that get replaced with actual values from `NotificationMessage.Variables`. For example:

- Template content: `"Hello {{user_name}}, your order {{order_id}} has shipped!"`
- Variables: `{"user_name": "Alice", "order_id": "ORD-12345"}`
- Rendered result: `"Hello Alice, your order ORD-12345 has shipped!"`

The template engine performs variable substitution after locale selection but before channel formatting. This ensures personalized content is properly escaped for each delivery channel (HTML entities for email, URL encoding for SMS links).

UserPreference Structure:

The `UserPreference` type controls which notifications a user receives and through which channels. This embedded structure within the User entity supports granular opt-out controls required for compliance with email marketing laws.

Field	Type	Description
GlobalOptOut	bool	Master switch that blocks ALL non-urgent notifications across all channels
ChannelPreferences	map[string]bool	Per-channel opt-in status keyed by channel name ("email", "sms", "push", "in_app")
CategoryPreferences	map[string]bool	Per-category opt-in status keyed by category name ("marketing", "alerts", "transactional")
LastUpdated	time.Time	Timestamp of most recent preference change for audit compliance

The preference resolution follows a hierarchy: `GlobalOptOut` trumps everything, then category preferences, then channel preferences. A user could disable all "marketing" notifications but still receive "transactional" messages via email and SMS.

Preference Resolution Logic:

Global Opt-Out	Category Enabled	Channel Enabled	Final Decision	Reasoning
true	any	any	Block	Global opt-out overrides everything except PriorityUrgent
false	false	any	Block	Category disabled blocks regardless of channel preference
false	true	false	Block	Channel disabled even though category allowed
false	true	true	Allow	All preferences align to allow delivery
any	"transactional"	any	Allow*	Transactional messages bypass most opt-outs per CAN-SPAM

*Note: Even transactional messages respect channel availability — you can't send SMS to a user without a phone number.

> **Compliance Consideration**: The distinction between transactional and marketing notifications is legally significant. Transactional messages (receipts, password resets, account notifications) generally cannot be opted out of under CAN-SPAM and GDPR, while marketing messages require explicit consent and must honor unsubscribe requests immediately.

Concrete Template Processing Example:

Consider a welcome email template processing flow:

1. Template "welcome_email" version 3 exists with content for "en", "es", and "fr"
2. User has locale "es-MX" (Mexican Spanish)
3. Template engine looks for exact match "es-MX" — not found
4. Falls back to "es" (generic Spanish) — found
5. Loads Spanish TemplateContent: subject "¡Bienvenido {{user_name}}!", HTMLBody with Spanish welcome text
6. Substitutes variables: `{"user_name": "Carlos"}` becomes "¡Bienvenido Carlos!"
7. Channel formatter applies HTML sanitization and inline CSS for email delivery
8. Result: Personalized Spanish welcome email ready for delivery

Common Pitfalls:

⚠️ **Pitfall: SMS Template Length**
SMS templates must fit within 160 characters after variable substitution, or they'll be split into multiple segments that cost extra to send. Always test SMS templates with realistic variable values (long names, large numbers) to ensure they stay within limits.

⚠️ **Pitfall: HTML Email XSS**
Template variables inserted into HTML email content must be properly escaped to prevent XSS attacks if user data contains malicious scripts. Use HTML entity encoding for all variables in HTMLBody content.

⚠️ **Pitfall: Transactional vs Marketing Classification**
Misclassifying marketing emails as "transactional" to bypass opt-outs is illegal under CAN-SPAM. Only genuine transactional messages (receipts, password resets, account changes) should use the transactional category.

⚠️ **Pitfall: Locale Fallback Chains**
Not implementing proper locale fallback can result in users receiving notifications in completely wrong languages. Always have English ("en") as the ultimate fallback, and test with edge case locales like "zh-Hans-CN".

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Data Storage	PostgreSQL with GORM	PostgreSQL with custom queries + connection pooling
Schema Migration	GORM AutoMigrate	golang-migrate with version control
JSON Handling	encoding/json with struct tags	jsoniter for high performance
Time Handling	time.Time with UTC storage	time.Time + github.com/rickar/cal for timezone math
Validation	Manual validation in handlers	github.com/go-playground/validator for struct validation

Recommended File Structure:

```
internal/
  models/
    notification.go          ← NotificationMessage, Priority, NotificationStatus types
    user.go                  ← User, UserPreference types
    template.go              ← Template, TemplateContent types
    delivery.go              ← DeliveryRecord type (used in Milestone 4)
    models_test.go           ← Unit tests for all model types
  repository/
    notification_repo.go     ← Database operations for notifications
    user_repo.go             ← Database operations for users
    template_repo.go         ← Database operations for templates
    repository_test.go       ← Integration tests with test database
  database/
    migrations/              ← SQL migration files
      001_initial_schema.sql
      002_add_indexes.sql
    connection.go            ← Database connection and configuration
```

Core Model Definitions:

Here are the complete Go struct definitions that implement our data model design:

```go
// internal/models/notification.go

package models

import (
    "time"
)

// Priority controls notification urgency and delivery behavior
type Priority int

const (
    PriorityLow Priority = iota + 1
    PriorityNormal
    PriorityHigh
    PriorityUrgent
)

// NotificationStatus tracks delivery state throughout processing pipeline
type NotificationStatus int

const (
    StatusQueued NotificationStatus = iota + 1
    StatusSending
    StatusSent
    StatusDelivered
    StatusFailed
    StatusBounced
)

// NotificationMessage represents a notification request submitted to the system
type NotificationMessage struct {
    ID          string            `json:"id" db:"id"`
    RecipientID string            `json:"recipient_id" db:"recipient_id"`
    TemplateID  string            `json:"template_id" db:"template_id"`
    Priority    Priority          `json:"priority" db:"priority"`
    Variables   map[string]string `json:"variables" db:"variables"`
    Category    string            `json:"category" db:"category"`
    CreatedAt   time.Time         `json:"created_at" db:"created_at"`
}
```

```go
// Validate checks that all required fields are present and valid
func (nm *NotificationMessage) Validate() error {
    // TODO: Implement validation logic
    // - Check ID is not empty
    // - Check RecipientID exists
    // - Check TemplateID is valid format
    // - Check Priority is within valid range
    // - Check Category is non-empty
    // - Validate Variables map for common XSS patterns
    return nil
}
```

```go
// internal/models/user.go

package models

import (
    "time"
)

// User stores recipient information and delivery preferences
type User struct {
    ID              string         `json:"id" db:"id"`
    Email           string         `json:"email" db:"email"`
    PhoneNumber     string         `json:"phone_number" db:"phone_number"`
    DeviceTokens    []string       `json:"device_tokens" db:"device_tokens"`
    Locale          string         `json:"locale" db:"locale"`
    Preferences     UserPreference `json:"preferences" db:"preferences"`
    QuietHoursStart string         `json:"quiet_hours_start" db:"quiet_hours_start"`
    QuietHoursEnd   string         `json:"quiet_hours_end" db:"quiet_hours_end"`
    Timezone        string         `json:"timezone" db:"timezone"`
}

// UserPreference controls notification delivery at channel and category level
type UserPreference struct {
    GlobalOptOut       bool              `json:"global_opt_out" db:"global_opt_out"`
    ChannelPreferences  map[string]bool   `json:"channel_preferences" db:"channel_preferences"`
    CategoryPreferences map[string]bool   `json:"category_preferences" db:"category_preferences"`
    LastUpdated        time.Time         `json:"last_updated" db:"last_updated"`
}

// CanReceive determines if this user should receive a notification
func (u *User) CanReceive(category string, channel string, priority Priority) bool {
    // TODO: Implement preference resolution logic
    // 1. Check GlobalOptOut (unless PriorityUrgent)
    // 2. Check CategoryPreferences for this category
    // 3. Check ChannelPreferences for this channel
    // 4. Check quiet hours based on user timezone (unless PriorityUrgent)
    // 5. Return final allow/block decision
    return false
}
```

```go
// IsInQuietHours checks if current time falls within user's quiet hours
func (u *User) IsInQuietHours() bool {
    // TODO: Implement quiet hours calculation
    // 1. Parse user timezone
    // 2. Convert current UTC time to user local time
    // 3. Parse QuietHoursStart and QuietHoursEnd as times
    // 4. Check if current local time falls between start and end
    // 5. Handle edge case where quiet hours span midnight
    return false
}
```

GO

```go
// internal/models/template.go

package models

import (
    "time"
)

// Template defines reusable notification content with localization support
type Template struct {
    ID        string                    `json:"id" db:"id"`
    Version   int                       `json:"version" db:"version"`
    Name      string                    `json:"name" db:"name"`
    Category  string                    `json:"category" db:"category"`
    Content   map[string]TemplateContent `json:"content" db:"content"`
    CreatedAt time.Time                 `json:"created_at" db:"created_at"`
    UpdatedAt time.Time                 `json:"updated_at" db:"updated_at"`
}

// TemplateContent contains channel-specific content for one language
type TemplateContent struct {
    Subject   string `json:"subject" db:"subject"`
    HTMLBody  string `json:"html_body" db:"html_body"`
    TextBody  string `json:"text_body" db:"text_body"`
    SMSBody   string `json:"sms_body" db:"sms_body"`
    PushTitle string `json:"push_title" db:"push_title"`
    PushBody  string `json:"push_body" db:"push_body"`
}

// GetContentForLocale returns the best matching content for a user's locale
func (t *Template) GetContentForLocale(locale string) *TemplateContent {
    // TODO: Implement locale resolution with fallback chain
    // 1. Try exact match (e.g., "es-MX")
    // 2. Try language code only (e.g., "es" from "es-MX")
    // 3. Fall back to "en" as default
    // 4. Return nil if no suitable content found
    return nil
}

// RenderContent substitutes variables into template content
```

```go
func (tc *TemplateContent) RenderContent(variables map[string]string, channel string) (string, error) {
    // TODO: Implement mustache-style variable substitution
    // 1. Select appropriate content field based on channel (HTMLBody for email, SMSBody for sms, etc.)
    // 2. Find all {{variable_name}} placeholders in content
    // 3. Replace with corresponding values from variables map
    // 4. Apply channel-specific escaping (HTML entities for email, etc.)
    // 5. Return rendered content or error if required variables missing
    return "", nil
}
```

Database Schema Migration:

```sql
-- migrations/001_initial_schema.sql

CREATE TABLE users (
    id VARCHAR(255) PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    phone_number VARCHAR(50),
    device_tokens JSONB DEFAULT '[]',
    locale VARCHAR(10) DEFAULT 'en',
    preferences JSONB NOT NULL DEFAULT '{}',
    quiet_hours_start TIME DEFAULT '23:00',
    quiet_hours_end TIME DEFAULT '08:00',
    timezone VARCHAR(50) DEFAULT 'UTC',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE templates (
    id VARCHAR(255) NOT NULL,
    version INTEGER NOT NULL,
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100) NOT NULL,
    content JSONB NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    PRIMARY KEY (id, version)
);

CREATE TABLE notification_messages (
    id VARCHAR(255) PRIMARY KEY,
    recipient_id VARCHAR(255) NOT NULL REFERENCES users(id),
    template_id VARCHAR(255) NOT NULL,
    priority INTEGER NOT NULL DEFAULT 2,
    variables JSONB DEFAULT '{}',
    category VARCHAR(100) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Indexes for common query patterns
CREATE INDEX idx_users_email ON users(email);
```

```
CREATE INDEX idx_notification_messages_recipient ON notification_messages(recipient_id);

CREATE INDEX idx_notification_messages_created ON notification_messages(created_at DESC);

CREATE INDEX idx_templates_category ON templates(category);
```

Language-Specific Hints for Go:

- Use `time.Time` consistently and store all timestamps in UTC. Convert to user timezone only for display and quiet hours calculations.
- JSONB columns in PostgreSQL map well to Go `map[string]interface{}` or specific struct types with json tags.
- The `database/sql` package requires custom scanning for JSONB fields — consider using GORM or pgx for easier JSON handling.
- Use `github.com/google/uuid` for generating notification IDs to ensure uniqueness across distributed deployments.
- Validate phone numbers with `github.com/nyaruka/phonenumbers` to ensure SMS delivery compatibility.
- Load templates at startup and cache them in memory — templates don't change frequently and you'll need them for every notification.

Milestone Checkpoint:

After implementing the data model:

1. **Run the tests**: `go test ./internal/models/...` should pass all validation and rendering tests
2. **Database verification**: Connect to your Postgres database and verify tables were created with: `\d users`, `\d templates`, `\d notification_messages`
3. **Manual testing**: Create a test user and template, then verify the `CanReceive` method correctly evaluates preferences
4. **Template rendering**: Test variable substitution with a sample template containing multiple variables and verify HTML escaping works

Common Implementation Issues:

Symptom	Likely Cause	How to Diagnose	Fix
"template not found" errors	Template ID/version mismatch	Check database for exact ID and version	Ensure template creation includes version 1 as default
Variables not substituting	Incorrect mustache syntax	Print template content before substitution	Use `{{var_name}}` format, not `{var_name}` or `$var_name$`
Quiet hours not working	Timezone conversion issues	Log both UTC and user local times	Use `time.LoadLocation()` with IANA timezone names
JSON marshaling errors	Missing struct tags	Check Go struct definitions	Add `json:"field_name"` tags to all exported fields
Preference resolution wrong	Incorrect logic precedence	Test each preference combination	GlobalOptOut should override everything except PriorityUrgent

Channel Abstraction and Routing

> **Milestone(s):** Milestone 1 - Channel Abstraction & Routing — implements pluggable channels with intelligent routing, fallback handling, and circuit breaker patterns

Think of the channel abstraction layer like a postal service sorting facility. When a letter arrives, the postal workers don't need to know whether it's going by truck, plane, or ship — they just need to know the destination and urgency level. The sorting facility has a standard process for every piece of mail: validate the address, determine the best delivery method based on destination and speed requirements, and if the primary route is unavailable (maybe the bridge is out), automatically route it through an alternative path. Our notification system works the same way — we create a unified interface that can handle any type of notification channel, with intelligent routing that considers user preferences, channel availability, and fallback strategies when things go wrong.

The key architectural insight is that while email, SMS, push notifications, and in-app messages have completely different APIs, delivery semantics, and failure modes, they all share the same fundamental operations: validate that we can reach the recipient, format the message appropriately for the channel, and send it with proper error handling. By abstracting these common operations behind a standard interface, we can build routing and fallback logic once and have it work across all channels, rather than duplicating this logic for each provider integration.

Channel Interface Design

The **channel abstraction** serves as the foundation that allows our notification system to treat all delivery methods uniformly while still respecting their unique characteristics. Every notification channel — whether it's SMTP email, Twilio SMS, Firebase Cloud Messaging for push notifications, or WebSocket connections for in-app messages — must implement the same core contract, enabling the routing engine to make decisions without understanding the underlying provider details.

> **Decision: Unified Channel Interface with Provider-Specific Implementations**
>
> - **Context**: Each notification provider (SendGrid, Twilio, FCM, APNs) has different APIs, authentication methods, payload formats, and error responses. We need to integrate multiple providers per channel type for redundancy.
> - **Options Considered**:
>   1. Direct provider integration in business logic
>   2. Provider-specific service classes with no common interface
>   3. Unified channel interface with adapter pattern implementations
> - **Decision**: Implement a unified `Channel` interface with provider-specific adapters
> - **Rationale**: The adapter pattern allows us to build routing, fallback, and preference logic once while maintaining the flexibility to swap providers or add new ones. It also enables testing with mock implementations and simplifies the routing engine by eliminating provider-specific conditional logic.
> - **Consequences**: Enables provider-agnostic routing logic and simplified testing, but requires additional abstraction layer and careful interface design to accommodate all provider capabilities.

The channel interface defines three core operations that every notification delivery method must support:

Method Name	Parameters	Returns	Description
`Validate`	`recipient User`, `message NotificationMessage`	`error`	Verifies that the channel can deliver to this recipient (valid email, phone number, device token, etc.)
`Format`	`content TemplateContent`, `variables map[string]string`	`FormattedMessage`, `error`	Transforms template content into channel-specific format (HTML email, 160-char SMS, JSON push payload)
`Send`	`recipient User`, `message FormattedMessage`, `priority Priority`	`DeliveryResult`, `error`	Delivers the formatted message via the provider API and returns tracking information

The `Validate` method serves as a pre-flight check that prevents expensive API calls to providers when we know delivery will fail. For email channels, this means checking that the recipient has a valid email address and hasn't previously hard-bounced. For SMS channels, it verifies the phone number format and that the user hasn't opted out of SMS delivery. For push notification channels, it confirms that we have a current device token that hasn't been invalidated. This validation step is crucial because provider APIs typically charge per attempt, regardless of whether the message is ultimately deliverable.

The `Format` method handles the complex task of transforming our generic template content into the specific format required by each channel. Email channels must decide between HTML and plain text based on the template content, properly encode headers, and inline CSS styles for maximum client compatibility. SMS channels must truncate content to fit within 160-character limits (or properly segment longer messages), remove HTML formatting, and handle Unicode characters correctly. Push notification channels must construct JSON payloads with the correct structure for iOS APNs versus Android FCM, including appropriate badge counts, sound settings, and custom data fields.

> The critical design principle here is that the channel interface operates on our internal data types (`User`, `NotificationMessage`, `TemplateContent`) rather than provider-specific formats. This keeps the abstraction clean and prevents provider details from leaking into the routing logic.

The `Send` method encapsulates all provider-specific communication, including authentication, rate limiting, retry logic, and error handling. Each implementation must translate our internal priority levels into provider-specific settings — for example, mapping `PriorityUrgent` to immediate delivery for push notifications or high-priority queues for email. The method returns a `DeliveryResult` that contains provider-specific tracking identifiers, delivery timestamps, and any metadata needed for subsequent delivery status updates.

Here's how different channel types implement the interface contract:

Channel Type	Validation Logic	Format Transformation	Send Implementation
Email (SMTP)	Valid email format, not in bounce list, MX record exists	HTML/plain text selection, header encoding, CSS inlining	SMTP authentication, TLS connection, proper envelope handling
Email (SendGrid)	Valid email format, SendGrid suppression list check	SendGrid template format, personalization tags	SendGrid API authentication, webhook setup, unsubscribe handling
SMS (Twilio)	Valid phone format, country code normalization, opt-out check	160-character segmentation, Unicode handling, URL shortening	Twilio API authentication, webhook configuration, cost optimization
Push (FCM)	Valid FCM token, app registration check	Android notification payload, data fields, priority mapping	FCM API v1 authentication, topic vs token delivery, batch optimization
Push (APNs)	Valid device token, app certificate validation	iOS notification payload, badge logic, sound configuration	APNs HTTP/2 connection, JWT authentication, feedback service integration
In-App (WebSocket)	Active connection exists, user session valid	JSON message format, real-time delivery requirements	WebSocket connection management, connection pooling, offline queuing

Each channel implementation also handles provider-specific error conditions and translates them into standard error types that the routing engine can understand. For example, both email providers might return different error codes for invalid email addresses, but the channel implementations translate these into a standard `ErrInvalidRecipient` that the routing engine can handle consistently.

Decision: Provider-Agnostic Error Classification

- **Context**: Different providers return different error codes and formats for similar failure conditions (invalid recipient, rate limiting, temporary failures)
- **Options Considered**:
  1. Expose raw provider errors to routing logic
  2. Create comprehensive error mapping for each provider
  3. Define standard error categories with provider-specific details
- **Decision**: Standard error categories (`ErrInvalidRecipient` , `ErrRateLimited` , `ErrProviderDown` , `ErrTemporaryFailure`) with provider details preserved in error metadata
- **Rationale**: Enables consistent retry and fallback logic while preserving debugging information. The routing engine can make decisions based on error category without understanding provider-specific error codes.
- **Consequences**: Simplifies routing logic and enables provider swapping, but requires careful error categorization and potential loss of provider-specific error nuances.

Routing Engine

The **routing engine** acts as the intelligent decision-making component that determines how each notification should be delivered based on the message characteristics, user preferences, channel availability, and current system conditions. Think of it like a sophisticated traffic management system that doesn't just route cars to their destination, but considers current road conditions, driver preferences (some people avoid highways), vehicle capabilities (motorcycles can't use carpool lanes), and real-time traffic data to choose the optimal path.

The routing decision process involves multiple layers of evaluation, each potentially eliminating certain channels or influencing the final delivery strategy. The engine must balance user preferences, system constraints, cost considerations, and delivery reliability to make optimal routing decisions for each notification.

> **Decision: Multi-Stage Routing Pipeline with Preference Hierarchy**
>
> - **Context**: Routing decisions depend on multiple factors: user preferences, notification urgency, channel availability, rate limits, cost constraints, and provider reliability
> - **Options Considered**:
>   1. Simple preference-based routing (user chooses email or SMS)
>   2. Rule-based system with hard-coded decision logic
>   3. Multi-stage pipeline with configurable routing rules
> - **Decision**: Multi-stage routing pipeline where each stage can filter or rank available channels
> - **Rationale**: Provides flexibility to handle complex routing scenarios while keeping each stage simple and testable. New routing criteria can be added as pipeline stages without modifying existing logic.
> - **Consequences**: Enables sophisticated routing strategies and easy testing, but adds complexity and potential performance overhead from multiple evaluation stages.

The routing pipeline processes each notification through the following stages:

1. **Channel Eligibility Filtering**: Determines which channels are theoretically available for this notification type and recipient
2. **Preference Evaluation**: Applies user preferences to eliminate channels the recipient has disabled
3. **Quiet Hours Assessment**: Suppresses non-urgent notifications during the user's do-not-disturb window
4. **Provider Health Checking**: Removes channels with currently unhealthy providers (circuit breaker state)
5. **Rate Limit Evaluation**: Defers channels that would exceed rate limits for this user or globally
6. **Cost Optimization**: Ranks remaining channels by cost efficiency while respecting urgency requirements
7. **Fallback Chain Construction**: Builds an ordered list of channels to attempt, with retry and backoff strategies

Let's walk through a concrete example to illustrate how the routing engine processes a notification. Consider a password reset notification for a user named Alice:

The notification arrives with `Priority` set to `PriorityHigh` because password resets are security-sensitive and time-critical. Alice's user profile shows she has both email and SMS configured, with device tokens for push notifications on her mobile app. Her preferences indicate she wants to receive security notifications via both email and SMS, but has disabled marketing notifications on SMS to avoid charges.

In the **Channel Eligibility** stage, the engine identifies that password reset notifications are classified as transactional security notifications. The system configuration specifies that security notifications can use email, SMS, and push channels, but not in-app notifications (since the user might be locked out of their account). This eliminates the in-app channel from consideration.

During **Preference Evaluation**, the engine checks Alice's notification preferences. Her security notification preferences show email enabled, SMS enabled for high-priority notifications, and push notifications enabled. Since this is a `PriorityHigh` notification, all three remaining channels pass this stage.

The **Quiet Hours Assessment** stage checks whether Alice is currently in her configured quiet hours (10 PM to 7 AM in her timezone). It's currently 2 AM in Alice's timezone, but since password reset is classified as a security notification with `PriorityHigh`, it bypasses quiet hours restrictions. Lower priority notifications would be deferred until 7 AM.

Provider Health Checking examines the circuit breaker state for each channel's providers. The email channel using SendGrid shows healthy status, the SMS channel using Twilio shows healthy status, but the push notification channel using FCM shows a degraded state due to recent API timeouts. The engine marks the push channel as available but lower priority due to provider health concerns.

During **Rate Limit Evaluation**, the engine checks that Alice hasn't exceeded per-user rate limits (maximum 5 SMS messages per hour, 50 email messages per hour) and that the system hasn't hit global provider rate limits. All channels pass this check since this is Alice's first notification today.

The **Cost Optimization** stage ranks the remaining channels by delivery cost and reliability. Email via SendGrid costs approximately $0.0001 per message, SMS via Twilio costs approximately $0.0075 per message, and push notifications via FCM are essentially free but have lower reliability due to current provider health issues. The engine ranks them: email (primary), push notifications (secondary), SMS (tertiary for cost reasons).

Finally, **Fallback Chain Construction** builds the delivery strategy: attempt email delivery first, if that fails within 30 seconds, attempt push notification, if that also fails, fall back to SMS delivery. Each attempt will have exponential backoff retry logic, and the total delivery window should not exceed 5 minutes for high-priority security notifications.

The routing engine also handles special cases and edge conditions that commonly occur in production notification systems:

Scenario	Routing Logic	Fallback Strategy
All preferred channels unavailable	Override user preferences for urgent notifications	Use any available channel, log preference override
User has no contact methods	Queue notification for later delivery	Send to admin dashboard for manual intervention
Notification sent outside business hours	Apply quiet hours logic by priority	Defer low priority, allow high/urgent priority
Provider experiencing outages	Circuit breaker eliminates affected channels	Route to backup providers or alternative channels
Rate limits exceeded	Queue for later delivery or use alternative channel	Implement backoff and retry with exponential delays
Cost budget exceeded	Prefer free channels (email, push) over paid (SMS)	Queue expensive notifications for manual approval
User timezone unknown	Use server timezone with conservative quiet hours	Default to 9 PM - 8 AM UTC quiet hours window
Conflicting preferences	Apply most restrictive preference	Log conflicts for preference cleanup

> The key insight for routing engine design is that the decision process must be deterministic and auditable. Every routing decision should be logged with sufficient detail to understand why particular channels were chosen or rejected, enabling debugging and compliance reporting.

The routing engine maintains routing metrics and performance data to continuously optimize decision-making:

Metric	Purpose	Usage
Channel success rates	Track delivery reliability by channel and provider	Influence fallback chain ordering
Average delivery latency	Measure end-to-end delivery time	Optimize channel selection for time-sensitive notifications
Cost per successful delivery	Calculate total cost including retries and fallbacks	Inform cost optimization decisions
User preference override frequency	Monitor how often system overrides user preferences	Identify preference configuration issues
Provider health trends	Track provider reliability over time	Adjust circuit breaker thresholds
Geographic delivery patterns	Analyze delivery success by user location	Optimize provider selection by region

Fallback and Circuit Breaker Strategy

When notification delivery fails, the system must gracefully handle the failure and attempt alternative delivery methods without overwhelming failing providers or creating cascading failures across the entire notification infrastructure. Think of this like a robust postal service that, when the primary delivery truck breaks down, has backup trucks ready to go, but also knows when to stop sending trucks down a flooded road that's clearly impassable.

The **fallback strategy** defines how the system responds to delivery failures by attempting alternative channels in a predetermined order, while the **circuit breaker pattern** protects the system from repeatedly attempting to use failing providers that are unlikely to recover quickly.

> **Decision: Hierarchical Fallback with Circuit Breaker Protection**
>
> - **Context**: Notification providers fail in various ways: temporary API issues, rate limiting, permanent outages, network problems, and authentication failures. Failed notifications must be retried through alternative channels without overwhelming failing systems.
> - **Options Considered**:
>   1. Simple retry with exponential backoff on the same channel
>   2. Immediate fallback to all alternative channels simultaneously
>   3. Sequential fallback through alternative channels with circuit breaker protection
> - **Decision**: Sequential fallback through alternative channels with per-provider circuit breakers
> - **Rationale**: Sequential fallback prevents duplicate notifications while ensuring delivery, circuit breakers prevent wasting resources on consistently failing providers, and per-provider isolation prevents one failing provider from affecting others.
> - **Consequences**: Provides reliable delivery with efficient resource usage, but adds complexity in state management and requires careful tuning of circuit breaker thresholds.

The fallback system operates on three distinct levels, each handling different types of failures:

Provider-Level Fallbacks handle failures within the same channel type by switching to alternative providers. For email delivery, if the primary SendGrid integration fails, the system automatically attempts delivery through a backup SMTP provider. For SMS delivery, if Twilio returns rate limiting errors, the system switches to an alternative SMS provider like AWS SNS. This level of fallback is invisible to the user and maintains the same channel characteristics (email stays email, SMS stays SMS).

Channel-Level Fallbacks switch to completely different notification channels when an entire channel type becomes unavailable. If email delivery fails across all email providers due to recipient issues (invalid email address, hard bounce), the system falls back to SMS delivery if the user has SMS enabled. This type of fallback may require content reformatting since different channels have different message format requirements.

Temporal Fallbacks defer notification delivery when immediate delivery isn't possible due to system-wide issues or user preferences. If all immediate delivery channels are unavailable, the system queues the notification for delivery during the next maintenance window or when providers recover. For non-urgent notifications sent during quiet hours, temporal fallback simply schedules delivery for after the quiet hours window ends.

The circuit breaker implementation tracks provider health independently for each notification channel and provider combination. Here's how the circuit breaker state machine operates:

Current State	Failure Condition	Success Condition	Next State	Actions Taken
`Closed` (Normal)	5 failures in 60 seconds	N/A - already healthy	`Open`	Mark provider unavailable, start recovery timer
`Open` (Failing)	N/A - not accepting requests	Recovery timer expires (300 seconds)	`HalfOpen`	Allow limited test requests
`HalfOpen` (Testing)	Any failure during test period	3 consecutive successes	`Open` / `Closed`	Return to Open or promote to Closed
`HalfOpen` (Testing)	10 successful test requests	10 consecutive successes	`Closed`	Mark provider healthy, resume normal traffic

The circuit breaker maintains separate failure counters for different types of errors because not all failures indicate provider health issues:

Error Type	Counts Toward Circuit Breaker	Rationale	Fallback Action
Invalid recipient address	No	User data issue, not provider issue	Try alternative channel if available
Authentication failure	Yes	Provider configuration or service issue	Switch to backup provider immediately
Rate limiting (429)	No	Expected behavior, not a failure	Defer request with backoff, don't fallback
Timeout (>30 seconds)	Yes	Provider performance issue	Count as failure, may indicate overload
Server error (5xx)	Yes	Provider service issue	Count as failure, likely temporary outage
Network connectivity	Yes	Infrastructure issue	Count as failure, may affect all providers
Invalid API response format	Yes	Provider API change or corruption	Count as failure, indicates service problem

When a circuit breaker opens (indicating a provider is unhealthy), the routing engine must dynamically recalculate fallback chains for in-flight notifications. This requires careful coordination to prevent notification duplication:

1. **In-Flight Request Handling**: Requests currently being processed by the failing provider are allowed to complete their retry cycles, but new requests are immediately routed to fallback providers.
2. **Notification State Tracking**: Each notification attempt is tagged with a provider identifier and attempt timestamp. If a provider circuit breaker opens while a notification is in progress, the system can identify which notifications need re-routing.
3. **Duplicate Prevention**: Before attempting delivery through a fallback provider, the system checks the notification's delivery history to ensure the same channel hasn't already succeeded through a different provider.
4. **User Communication**: For critical notifications (password resets, security alerts), if all fallback channels fail, the system may need to escalate to manual intervention or administrative notification.

Let's trace through a complex fallback scenario to illustrate how all these pieces work together:

A high-priority security alert needs to be sent to 10,000 users at 3 PM EST. The primary email provider (SendGrid) is experiencing an outage, the primary SMS provider (Twilio) is rate-limiting due to high volume, and the push notification provider (FCM) is operating normally.

The routing engine initially selects email as the primary channel for all 10,000 notifications based on user preferences and cost optimization. As SendGrid requests begin failing with timeout errors, the circuit breaker starts counting failures. After 5 failures in 60 seconds, the email circuit breaker opens.

For the remaining 9,950 notifications still queued for email delivery, the routing engine immediately re-evaluates the fallback strategy. Since email is unavailable, it attempts the secondary email provider (SMTP backup). If this provider is also experiencing issues due to the same underlying infrastructure problems, its circuit breaker may also open quickly.

With email channels exhausted, the routing engine falls back to push notifications for users who have mobile apps installed and push notifications enabled. This succeeds for approximately 60% of users who have active device tokens and recent app usage.

For the remaining 40% of users who either don't have push notifications enabled or have invalid device tokens, the system attempts SMS delivery. However, Twilio is returning 429 rate limiting responses. Since rate limiting doesn't trigger the circuit breaker (it's expected behavior), the system applies exponential backoff and queues these SMS notifications for delivery over the next 30 minutes.

For users who have disabled both push notifications and SMS (approximately 5% of the user base), the system must make a decision about preference overrides. Since this is a high-priority security notification, the system logs the preference override and attempts SMS delivery anyway, accepting the risk that some users may receive unwanted SMS messages rather than missing critical security information.

> ⚠️ **Pitfall: Cascading Failures During Provider Outages** A common mistake is not implementing proper backpressure when fallback providers become overwhelmed by traffic from failed primary providers. If your primary email provider fails and you immediately route 100,000 notifications to your backup provider, you may overwhelm the backup and cause it to fail too. Always implement gradual traffic shifting and monitor backup provider performance during failovers.

The fallback system must also handle edge cases that commonly occur during real-world provider outages:

Edge Case	Problem	Solution
Partial Provider Outage	Provider works for some users but not others	Implement per-user circuit breaker state, not just global
Geographic Provider Issues	Provider works in some regions but not others	Circuit breaker tracks failure rates by user location
Provider Recovery Flapping	Provider alternates between working and failing	Implement minimum circuit breaker open time (5 minutes)
Fallback Provider Cost Explosion	Backup SMS provider has much higher costs	Implement cost-based throttling during fallback scenarios
Duplicate Notification Risk	User receives notification via multiple channels	Track successful deliveries across all providers and channels
Notification Ordering	Critical notifications arrive out of order due to fallbacks	Implement sequence numbers and delivery ordering constraints

Implementation Guidance

The channel abstraction and routing system requires careful attention to concurrent processing, provider API integration, and stateful circuit breaker management. Here's how to structure the implementation effectively.

Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Client	Go net/http with timeout configuration	Resilient HTTP client with retry logic (hashicorp/go-retryablehttp)
Circuit Breaker	Simple counter-based implementation	Sony gobreaker or hystrix-go for production features
Provider APIs	Direct REST API integration	Official SDKs (sendgrid-go, twilio-go, firebase-admin-go)
State Storage	In-memory maps with sync.RWMutex	Redis for distributed circuit breaker state
Configuration	JSON config files	Consul/etcd for dynamic configuration updates
Metrics	Simple counters and gauges	Prometheus metrics with Grafana dashboards

Recommended File Structure:

```
internal/
├── channels/
│   ├── channel.go              ← Channel interface definition
│   ├── email/
│   │   ├── sendgrid.go         ← SendGrid implementation
│   │   ├── smtp.go             ← Generic SMTP implementation
│   │   └── email_test.go       ← Email channel tests
│   ├── sms/
│   │   ├── twilio.go           ← Twilio implementation
│   │   └── sms_test.go         ← SMS channel tests
│   ├── push/
│   │   ├── fcm.go              ← Firebase Cloud Messaging
│   │   ├── apns.go             ← Apple Push Notification Service
│   │   └── push_test.go        ← Push notification tests
│   └── inapp/
│       ├── websocket.go        ← WebSocket in-app notifications
│       └── inapp_test.go       ← In-app channel tests
├── routing/
│   ├── engine.go               ← Main routing engine
│   ├── fallback.go             ← Fallback strategy implementation
│   ├── circuit_breaker.go      ← Circuit breaker implementation
│   └── routing_test.go         ← Routing logic tests
└── delivery/
    ├── processor.go            ← Main notification processor
    ├── tracker.go              ← Delivery status tracking
    └── delivery_test.go        ← End-to-end delivery tests
```

Core Channel Interface (Complete):

```go
package channels

import (
    "context"
    "time"
)

// Channel defines the interface that all notification delivery channels must implement.
// This abstraction allows the routing engine to work with any provider without
// understanding provider-specific details.
type Channel interface {
    // Validate checks if this channel can deliver to the specified recipient.
    // Returns an error if the recipient data is invalid or delivery is not possible.
    Validate(ctx context.Context, recipient User, message NotificationMessage) error

    // Format transforms template content into the channel-specific format required
    // by the provider API (HTML email, SMS text, JSON push payload, etc.).
    Format(ctx context.Context, content TemplateContent, variables map[string]string) (FormattedMessage, error)

    // Send delivers the formatted message via the provider API and returns tracking
    // information for delivery status monitoring.
    Send(ctx context.Context, recipient User, message FormattedMessage, priority Priority) (DeliveryResult, error)

    // HealthCheck verifies that the channel provider is currently accessible.
    // Used by circuit breakers to determine provider availability.
    HealthCheck(ctx context.Context) error

    // Name returns a human-readable identifier for this channel implementation.
    Name() string
}

// FormattedMessage contains channel-specific formatted content ready for delivery.
type FormattedMessage struct {
    Subject     string            // Email subject, SMS preview, push title
    Body        string            // Main content formatted for channel
    ContentType string            // "text/plain", "text/html", "application/json"
    Headers     map[string]string // Channel-specific headers or metadata
    Attachments []Attachment      // File attachments (email only)
```

```
}

// DeliveryResult contains provider response data for delivery tracking.
type DeliveryResult struct {
    ProviderMessageID string    // Provider's tracking ID for this message
    DeliveryStatus    string    // Initial status from provider
    Timestamp         time.Time // When the message was accepted by provider
    ProviderMetadata  map[string]interface{} // Provider-specific response data
}

// Attachment represents a file attachment for email notifications.
type Attachment struct {
    Filename    string
    ContentType string
    Data        []byte
}
```

Circuit Breaker Implementation (Complete):

GO

```go
package routing

import (
    "sync"
    "time"
)

// CircuitBreaker implements the circuit breaker pattern to protect against
// repeatedly calling failing services. Each provider gets its own circuit breaker.
type CircuitBreaker struct {
    mu               sync.RWMutex
    state            CircuitState
    failures         int
    lastFailureTime  time.Time
    nextRetryTime    time.Time

    // Configuration
    failureThreshold  int           // Number of failures before opening
    recoveryTimeout   time.Duration // How long to wait before testing recovery
    monitoringWindow  time.Duration // Time window for failure counting
}

type CircuitState int

const (
    StateClosed CircuitState = iota   // Normal operation
    StateOpen                         // Failing, rejecting requests
    StateHalfOpen                     // Testing recovery
)

// NewCircuitBreaker creates a circuit breaker with the specified thresholds.
func NewCircuitBreaker(failureThreshold int, recoveryTimeout time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
        state:            StateClosed,
        failureThreshold: failureThreshold,
        recoveryTimeout:  recoveryTimeout,
        monitoringWindow: 60 * time.Second,
    }
}
```

```go
// CanExecute checks if requests should be allowed through the circuit breaker.
func (cb *CircuitBreaker) CanExecute() bool {
    // TODO: Implement state checking logic
    // TODO: Handle automatic state transitions based on time
    // TODO: Return true for Closed/HalfOpen, false for Open
    return true // Placeholder
}

// RecordSuccess notifies the circuit breaker of a successful operation.
func (cb *CircuitBreaker) RecordSuccess() {
    // TODO: Reset failure counter if in Closed state
    // TODO: Transition from HalfOpen to Closed after enough successes
    // TODO: Update last success timestamp
}

// RecordFailure notifies the circuit breaker of a failed operation.
func (cb *CircuitBreaker) RecordFailure(err error) {
    // TODO: Increment failure counter
    // TODO: Check if failure count exceeds threshold
    // TODO: Transition to Open state if threshold exceeded
    // TODO: Set next retry time based on recovery timeout
}
```

Routing Engine Core Logic (Skeleton):

```go
package routing


// RoutingEngine determines the best channel and provider for each notification

// based on user preferences, provider health, and system constraints.

type RoutingEngine struct {

    channels        map[string]Channel

    circuitBreakers map[string]*CircuitBreaker

    config          *RoutingConfig

}


// RouteNotification determines the delivery strategy for a notification.

func (re *RoutingEngine) RouteNotification(ctx context.Context, user User, message NotificationMessage) (*RoutingPlan, error)
{

    // TODO 1: Filter channels based on notification type eligibility

    // TODO 2: Apply user preferences to eliminate disabled channels

    // TODO 3: Check quiet hours and defer non-urgent notifications

    // TODO 4: Remove channels with unhealthy providers (circuit breaker check)

    // TODO 5: Apply rate limiting constraints

    // TODO 6: Rank remaining channels by cost and reliability

    // TODO 7: Build fallback chain with retry strategies

    // TODO 8: Return complete routing plan with primary and fallback channels



    return &RoutingPlan{}, nil // Placeholder

}


// RoutingPlan contains the complete delivery strategy for a notification.

type RoutingPlan struct {

    PrimaryChannel   string

    FallbackChannels []string

    RetryPolicy      RetryPolicy

    DeferUntil       *time.Time // For quiet hours handling

}
```

Provider Integration Examples:

For **SendGrid Email Channel**:

```go
package email


import (

    "github.com/sendgrid/sendgrid-go"

    "github.com/sendgrid/sendgrid-go/helpers/mail"

)


type SendGridChannel struct {

    client *sendgrid.Client

    fromEmail string

    fromName string

}


func NewSendGridChannel(apiKey, fromEmail, fromName string) *SendGridChannel {

    return &SendGridChannel{

        client: sendgrid.NewSendClient(apiKey),

        fromEmail: fromEmail,

        fromName: fromName,

    }

}


func (sg *SendGridChannel) Send(ctx context.Context, recipient User, message FormattedMessage, priority Priority)
(DeliveryResult, error) {

    // TODO: Create SendGrid mail object from FormattedMessage

    // TODO: Set recipient, subject, and content

    // TODO: Apply priority-specific settings (send time, tracking)

    // TODO: Make API call to SendGrid

    // TODO: Parse response and extract tracking ID

    // TODO: Return DeliveryResult with provider tracking information



    return DeliveryResult{}, nil // Placeholder

}
```

For **Twilio SMS Channel**:

```go
package sms

import (
    "github.com/twilio/twilio-go"
    api "github.com/twilio/twilio-go/rest/api/v2010"
)

type TwilioChannel struct {
    client *twilio.RestClient
    fromNumber string
}

func (tw *TwilioChannel) Format(ctx context.Context, content TemplateContent, variables map[string]string) (FormattedMessage, error) {
    // TODO: Render template content with variable substitution
    // TODO: Strip HTML tags and format as plain text
    // TODO: Truncate to 160 characters or calculate SMS segments
    // TODO: Handle Unicode characters properly
    // TODO: Return FormattedMessage with SMS-appropriate content

    return FormattedMessage{}, nil // Placeholder
}
```

Milestone Checkpoint: After implementing the channel abstraction:

1. **Unit Test Coverage**: Run `go test ./internal/channels/...` - you should see tests passing for each channel's Validate, Format, and Send methods
2. **Integration Test**: Start the notification service and send a test notification via curl - verify it routes to the correct channel based on user preferences
3. **Circuit Breaker Test**: Simulate a provider outage by returning errors from a channel implementation - verify the circuit breaker opens and routes to fallback channels
4. **Routing Logic Test**: Send notifications with different priorities during quiet hours - verify urgent notifications bypass quiet hours while normal priority notifications are deferred

Debugging Common Issues:

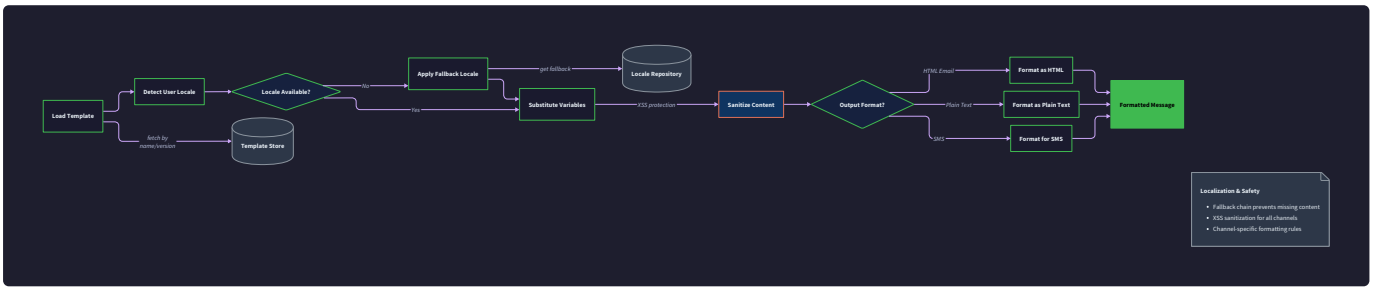
Symptom	Likely Cause	How to Diagnose	Fix
All notifications fail with "no available channels"	Circuit breakers opened for all providers	Check circuit breaker state logs, test provider health endpoints	Verify provider API keys and network connectivity
Notifications sent to wrong channel	Routing engine preference logic error	Log routing decisions at each stage	Review user preference data and routing rule implementation
Duplicate notifications received	Fallback logic not checking delivery success	Check delivery tracking logs for duplicate attempts	Add delivery status checking before fallback attempts
High provider API costs	Fallback routing to expensive channels unnecessarily	Monitor channel usage and cost metrics	Tune circuit breaker thresholds and fallback ordering
Notifications delayed indefinitely	Quiet hours logic preventing delivery	Check user timezone and quiet hours configuration	Implement priority override for urgent notifications

Template System and Localization

> **Milestone(s):** Milestone 2 - Template System — implements notification templates with localization, personalization, variable substitution, XSS protection, and template versioning

Think of the template system as a sophisticated printing press for the digital age. Just as a traditional printing press uses master plates with variable slots that can be filled with different content to produce customized documents, our template engine maintains master templates with placeholder variables that get substituted with personalized data for each recipient. The key insight is that we need to support not just variable substitution, but also multiple languages (localization), multiple output formats (HTML email vs plain text SMS), and safe content rendering that prevents malicious script injection.

The template system serves as the content generation layer between the raw notification data and the formatted messages that channels deliver to recipients. When a notification flows through the system, the template engine transforms generic notification requests into personalized, localized, and channel-appropriate content. This transformation must handle complex requirements: a single logical notification (like "password reset") might need to render as an HTML email in French, a plain text SMS in English, or a push notification with character limits in Spanish.



The template system architecture consists of three core subsystems working in concert. The **template engine** handles variable substitution and content rendering using mustache-style syntax with security safeguards. The **localization system** manages multiple language variants with intelligent fallback chains when exact locale matches aren't available. The **versioning system** tracks template changes over time, enabling safe deployments, rollbacks, and A/B testing of template content.

Template Engine

The template engine transforms static template content into personalized messages by substituting variables with actual data values. Think of it as a mail merge system, but one that must handle multiple output formats, prevent security vulnerabilities, and gracefully handle missing or malformed data.

> **Decision: Mustache-Style Template Syntax**
>
> - **Context**: Need a template syntax that's simple for non-technical users to edit, secure against injection attacks, and supports multiple output formats
> - **Options Considered**: Handlebars (full JavaScript expressions), Mustache (logic-less), Custom syntax
> - **Decision**: Mustache-style syntax with custom extensions for pluralization and formatting
> - **Rationale**: Logic-less templates prevent injection attacks, familiar syntax reduces learning curve, can be safely edited by marketing teams without developer oversight
> - **Consequences**: Enables safe template editing by non-developers, limits template complexity (requires business logic in code), requires custom functions for complex formatting

The template engine processes templates through a multi-stage pipeline that ensures security and correctness at each step. First, it parses the template content to identify variable placeholders using double-brace syntax like `{{user.firstName}}` and `{{order.total}}`. Second, it validates that all required variables are present in the substitution data, applying default values or error handling for missing variables. Third, it performs the actual substitution, escaping values appropriately for the target format (HTML escaping for email, URL encoding for links, plain text for SMS). Finally, it applies channel-specific formatting transformations to ensure the content fits the channel's constraints and conventions.

Template Processing Stage	Input	Output	Purpose
Parse	Raw template string	AST with variable nodes	Identify substitution points and validate syntax
Validate	AST + variable data	Validation result	Ensure all required variables are available
Substitute	AST + validated data	Rendered string	Replace placeholders with actual values
Format	Rendered content	Channel-specific message	Apply channel constraints and formatting
Sanitize	Formatted message	Safe message	Remove/escape potentially dangerous content

The variable substitution mechanism supports nested object access using dot notation, allowing templates to reference complex data structures like `{{user.preferences.language}}` or `{{order.items.0.name}}`. The engine handles missing properties gracefully by providing configurable default values or empty strings, preventing template rendering failures that would block notification delivery. For numeric values, the engine includes built-in formatters for currencies, dates, and percentages that respect the recipient's locale settings.

> The critical security insight is that different output formats require different escaping strategies. HTML email content must escape `<script>` tags and HTML entities, while SMS content can contain raw text, and JSON push payloads must escape quotes and control characters. A single variable value like `user input with <script>alert('xss')</script>` must be transformed differently for each channel.

Security protection operates at multiple levels to prevent both accidental and malicious content injection. The template parser rejects templates containing potentially dangerous patterns like `{{#eval}}` or `{{{raw_html}}}` that could execute code. The variable substitution engine applies context-aware escaping based on where the variable appears in the template—variables inside `href` attributes get URL encoding, while variables in HTML content get HTML entity encoding. For SMS and plain text channels, the engine strips HTML tags and decodes entities to produce clean text output.

Escaping Context	Pattern	Example	Applied Escaping
HTML Content	`<p>{{message}}</p>`	`<p>Hello <script></p>`	HTML entity encoding
HTML Attribute	`<a href="{{url}}">`	`<a href="https%3A//example.com">`	URL encoding
JSON Value	`{"text": "{{message}}"}`	`{"text": "He said \"hello\""}`	JSON string escaping
Plain Text	`{{message}}`	`Hello there`	No escaping needed
CSS Value	`style="color: {{color}}"`	`style="color: #ff0000"`	CSS value validation

Template content validation ensures that templates produce valid output for their target channels before they're used in production. The validation engine checks that HTML email templates produce well-formed markup with proper DOCTYPE declarations and inline CSS for maximum email client compatibility. For SMS templates, it verifies that the rendered content fits within the 160-character limit for single-segment messages, warning template authors when variables might cause overflow. Push notification templates are validated against platform-specific payload size limits and required field structures for iOS and Android.

⚠️ **Pitfall: SMS Character Counting** Many developers count characters in SMS templates without considering that variable substitution can dramatically increase the final message length. A template like `Hi {{firstName}}, your order {{orderNumber}} is ready!` might fit in 160 characters with short test data, but real user names like "Christopher Alexander" and order numbers like "ORD-2023-11-15-AB12345" can push the message over multiple SMS segments, increasing costs significantly. Always validate SMS templates with realistic maximum-length variable values, not just test data.

Localization and Fallback

The localization system manages multiple language variants of templates and intelligently selects the best available content for each recipient's language preferences. Think of it as a librarian who maintains the same book in multiple languages and always finds the closest match when someone requests a specific edition that might not be available.

The locale detection process begins by examining the recipient's profile for their preferred language setting, represented as standard locale codes like `en-US`, `fr-CA`, or `pt-BR`. When an exact match isn't available, the system applies a cascading fallback strategy that tries progressively broader locale matches. For example, if a user prefers `zh-HK` (Chinese as used in Hong Kong) but only `zh-CN` (simplified Chinese) and `en-US` templates exist, the system selects `zh-CN` as a closer cultural match than falling back to English.

Decision: ICU Locale Fallback Chain Strategy

- **Context**: Users specify locale preferences like `pt-BR` but templates might only exist in `pt` or `en`
- **Options Considered**: Exact match only (fail if not found), Language-only fallback (`pt-BR` → `pt` → `en`), Custom priority lists per user
- **Decision**: ICU-standard fallback chain with configurable default locale
- **Rationale**: ICU standard is widely understood, handles regional variations correctly, balances cultural appropriateness with practical availability
- **Consequences**: Reduces template maintenance burden, provides predictable fallback behavior, may serve less-preferred language rather than failing

The fallback chain resolution follows a structured algorithm that maximizes content relevance while ensuring notification delivery always succeeds. The system first attempts an exact locale match (e.g., `fr-CA`), then tries the language without region (e.g., `fr`), then falls back to the configured default locale (typically `en-US`). If even the default locale is missing, the system selects the first available locale variant and logs an error for template administrators to address.

Fallback Stage	Example Request	Available Templates	Selected Template	Reasoning
Exact Match	`de-AT`	`de-AT`, `en-US`	`de-AT`	Perfect match found
Language Match	`de-CH`	`de-DE`, `en-US`	`de-DE`	Same language, different region
Default Locale	`ja-JP`	`fr-FR`, `en-US`	`en-US`	Fall back to system default
First Available	`ko-KR`	`es-ES`	`es-ES`	No good match, use any available
Template Missing	`it-IT`	*(none)*	Error	Cannot render notification

Template content organization supports both shared and region-specific customizations within language families. A template might define general French content in the `fr` locale while providing Quebec-specific variants in `fr-CA` that use different currency formats, legal disclaimers, or cultural references. The template storage system handles this hierarchy efficiently by storing only the differences between regional variants and their parent language templates, reducing storage overhead and simplifying maintenance.

The localization system extends beyond simple text translation to handle culture-specific formatting requirements. Date formats, currency symbols, number separations, and address formats all vary by locale and must be applied consistently within templates. The system integrates with internationalization libraries to format variables like `{{order.date}}` as "November 15, 2023" for English users and "15 novembre 2023" for French users, using the same underlying data but culturally appropriate presentations.

> The key insight for locale fallback is that it's better to show a user content in a related language than to fail to deliver an important notification. A Portuguese speaker can usually understand Spanish content well enough to reset their password, but they can't take action on a notification that never arrives because no exact template match existed.

Pluralization rules add another layer of complexity that varies dramatically between languages. English has relatively simple plural rules (one item vs. multiple items), but languages like Russian have complex rules based on the specific numeric value, and languages like Arabic have different plural forms for zero, one, two, few, many, and other quantities. The template system integrates with ICU MessageFormat to handle these rules correctly, allowing templates to specify patterns like `{itemCount, plural, =0 {no items} =1 {one item} other {# items}}` that render appropriately for the user's language.

Language	Plural Categories	Example Numbers	Template Complexity
English	one, other	1 → one, 2+ → other	Simple
French	one, other	0,1 → one, 2+ → other	Simple
Russian	one, few, many, other	1,21 → one, 2-4 → few, 5+ → many	Complex
Arabic	zero, one, two, few, many, other	0 → zero, 1 → one, 2 → two...	Very Complex
Chinese	other	All numbers → other	Minimal

⚠️ **Pitfall: Hardcoded Date/Time Formatting** A common mistake is embedding date formatting directly in templates like `Order placed on {{order.month}}/{{order.day}}/{{order.year}}` instead of using proper locale-aware formatting. This approach breaks for international users

who expect DD/MM/YYYY format instead of MM/DD/YYYY, and completely fails for languages that don't use Arabic numerals. Always use locale-aware date formatting functions that handle the cultural presentation within the template engine, not in the template content itself.

Template Versioning

The template versioning system manages template evolution over time, enabling safe deployments, rollbacks, and gradual rollouts of template changes. Think of it as version control for content, where each template change creates a new immutable version while preserving the complete history for auditing and recovery purposes.

Template versions are immutable once created, meaning that any change to template content, metadata, or localization generates a new version number rather than overwriting the existing template. This immutability provides crucial safety properties: notifications that are queued or in-flight always use the template version that existed when they were created, preventing inconsistent rendering if templates change during processing. It also enables confident rollbacks since previous template versions remain exactly as they were when last used in production.

> **Decision: Semantic Versioning for Templates**
>
> - **Context**: Need to track template changes and coordinate between marketing teams making content changes and engineering teams deploying template code
> - **Options Considered**: Sequential integers, Timestamp-based versions, Semantic versioning (major.minor.patch)
> - **Decision**: Semantic versioning with automated minor version increments for content changes
> - **Rationale**: Semantic versioning communicates change impact clearly, integrates well with deployment processes, allows marketing teams to understand compatibility
> - **Consequences**: Enables coordinated rollouts between content and code, requires training for non-technical users, provides clear rollback points

The version numbering scheme follows semantic versioning principles adapted for template content. Major version increments (1.0.0 → 2.0.0) indicate breaking changes like removing required variables or changing fundamental template structure. Minor version increments (1.1.0 → 1.2.0) represent backward-compatible additions like new localization variants or optional variables. Patch version increments (1.1.0 → 1.1.1) cover content updates like fixing typos or updating styling that don't affect the template's interface contract with the notification system.

Version Change Type	Example	Impact	Rollback Risk
Major (Breaking)	Remove required variable	Code changes needed	High - may break active notifications
Minor (Addition)	Add new locale variant	No code changes	Low - purely additive
Patch (Content)	Fix typo in subject line	No code changes	Minimal - content only
Hotfix	Fix critical security issue	Urgent deployment	Medium - needs validation

Template deployment follows a staged approach that validates changes before they reach production traffic. New template versions begin in a "draft" state where they can be previewed with sample data but aren't used for live notifications. The preview system renders templates with realistic test data across all supported locales and channels, allowing content creators to verify formatting, character limits, and visual appearance before activation. Once validated, templates move to "active" status and become available for new notifications, while the previous version remains available for any in-flight notifications.

The template storage system maintains complete audit trails for all template modifications, recording who made changes, when they were made, and what specific content changed between versions. This audit trail proves essential for compliance requirements, debugging notification issues, and understanding the evolution of communication strategies over time. The system stores both the rendered differences between versions and the complete template content for each version, enabling both quick change summaries and complete historical reconstruction.

Template preview functionality allows stakeholders to see exactly how templates will render across different scenarios before deployment. The preview system accepts sample variable data and generates rendered output for all supported channels and locales, highlighting potential issues like SMS character overruns, HTML rendering problems, or missing translations. Advanced preview features include rendering templates with edge-case data (very long names, special characters, missing optional variables) to identify potential formatting problems before they affect real users.

Preview Scenario	Test Data	Purpose	Common Issues Found
Happy Path	Typical user data	Verify normal rendering	Layout problems, styling issues
Edge Cases	Maximum-length values	Test character limits	SMS overflow, HTML wrapping
Missing Data	Sparse variable set	Test graceful degradation	Template errors, ugly gaps
Special Characters	Unicode, emojis, symbols	Test encoding handling	Mojibake, broken rendering
All Locales	Same data, all languages	Test translation quality	Missing translations, formatting

> The key insight for template versioning is that content and code have different release cycles. Marketing teams need to update email copy frequently, while engineering teams deploy template code changes less often. The versioning system must decouple these cycles while maintaining consistency and preventing conflicts.

Template rollback capabilities provide safety nets when template changes cause unexpected issues in production. The system supports both automatic and manual rollback triggers. Automatic rollbacks activate when template rendering error rates exceed configurable thresholds, indicating that the new template version has systematic problems. Manual rollbacks allow administrators to immediately revert to previous template versions when issues are detected through monitoring or user reports.

The rollback process preserves ongoing notification delivery by switching the "active" template version pointer rather than disrupting in-flight processing. Notifications that were already queued with the problematic template version continue processing with that version to maintain consistency, while new notifications immediately begin using the rolled-back template version. This approach prevents notification rendering inconsistencies that could confuse users who might receive multiple related messages with different formatting.

⚠️ **Pitfall: Template Version Drift** A subtle but serious issue occurs when different notification types or channels end up using different template versions unintentionally, creating inconsistent user experiences. For example, password reset emails might use template version 1.2.0 while password reset SMS messages use version 1.1.0, resulting in different formatting, variable handling, or localization. Always deploy template changes atomically across all channels and notification types, or users will receive confusingly inconsistent messages from the same logical operation.

Template analytics track the performance and effectiveness of different template versions, providing data-driven insights for content optimization. The system measures metrics like delivery rates, open rates, click-through rates, and conversion rates for each template version, enabling A/B testing of template changes. These metrics feed back into the template development process, helping content creators understand which changes improve user engagement and which changes should be reverted or refined.

Implementation Guidance

The template system requires careful balance between flexibility and security, supporting rich content creation while preventing injection attacks and ensuring reliable delivery across all channels.

Technology Recommendations:

Component	Simple Option	Advanced Option
Template Engine	`text/template` with custom functions	`mustache` library with security extensions
Localization	JSON files with fallback logic	ICU MessageFormat with plural rules
Template Storage	File-based with Git versioning	Database with audit trails and preview
Content Validation	Basic syntax checking	Full rendering validation per channel
Security Scanning	Pattern matching for dangerous content	AST-based content analysis

Recommended Module Structure:

```
internal/
  template/
    engine.go              ← Core template rendering and variable substitution
    localization.go        ← Locale detection and fallback logic
    versioning.go          ← Template version management and storage
    validation.go          ← Content validation and security checking
    preview.go             ← Template preview and testing functionality
    types.go               ← Template, TemplateContent, and related types
    engine_test.go         ← Unit tests for rendering and security
    testdata/              ← Sample templates for testing
      email_welcome.json
      sms_verification.json
  storage/
    template_store.go      ← Template persistence and retrieval
    audit_log.go           ← Change tracking and audit trails
```

Core Template Engine Infrastructure:

```go
package template

import (
    "bytes"
    "context"
    "fmt"
    "html/template"
    "strings"
    "time"
)

// TemplateEngine handles template rendering with security and localization
type TemplateEngine struct {
    store          TemplateStore
    validator      ContentValidator
    localizer      LocalizationManager
    securityPolicy SecurityPolicy
}

// NewTemplateEngine creates a template engine with security policies
func NewTemplateEngine(store TemplateStore, validator ContentValidator, localizer LocalizationManager) *TemplateEngine {
    return &TemplateEngine{
        store:          store,
        validator:      validator,
        localizer:      localizer,
        securityPolicy: DefaultSecurityPolicy(),
    }
}

// SecurityPolicy defines content security rules for template rendering
type SecurityPolicy struct {
    AllowHTML       bool
    AllowJavaScript bool
    AllowEmbeds     bool
    MaxVariableSize int
    ForbiddenTags   []string
}

// DefaultSecurityPolicy returns conservative security settings
```

```go
func DefaultSecurityPolicy() SecurityPolicy {
    return SecurityPolicy{
        AllowHTML:       true,  // Email channels need HTML
        AllowJavaScript: false, // Never allow JavaScript in templates
        AllowEmbeds:     false, // No iframes or objects
        MaxVariableSize: 10000, // Prevent massive variable injection
        ForbiddenTags:   []string{"script", "iframe", "object", "embed", "form"},
    }
}

// ContentValidator checks template content for security and correctness
type ContentValidator interface {
    ValidateTemplate(template *Template, channel string) error
    ValidateVariables(variables map[string]string, template *Template) error
    SanitizeContent(content string, contentType string) (string, error)
}

// LocalizationManager handles locale detection and template selection
type LocalizationManager interface {
    DetectLocale(user *User) string
    GetBestTemplate(templateID string, locale string) (*TemplateContent, error)
    FormatVariable(value string, varType string, locale string) (string, error)
}
```

Template Storage and Versioning Infrastructure:

GO

```go
// TemplateStore manages template persistence and versioning
type TemplateStore interface {
    GetTemplate(ctx context.Context, templateID string, version int) (*Template, error)
    GetActiveTemplate(ctx context.Context, templateID string) (*Template, error)
    StoreTemplate(ctx context.Context, template *Template) error
    ListVersions(ctx context.Context, templateID string) ([]TemplateVersion, error)
    SetActiveVersion(ctx context.Context, templateID string, version int) error
    GetAuditLog(ctx context.Context, templateID string) ([]AuditEntry, error)
}

// TemplateVersion represents a version entry in the template history
type TemplateVersion struct {
    Version   int       `json:"version"`
    CreatedAt time.Time `json:"created_at"`
    CreatedBy string    `json:"created_by"`
    Summary   string    `json:"summary"`
    IsActive  bool      `json:"is_active"`
}

// AuditEntry tracks changes to templates for compliance and debugging
type AuditEntry struct {
    TemplateID string                 `json:"template_id"`
    Version    int                    `json:"version"`
    Action     string                 `json:"action"` // "created", "activated", "rolled_back"
    ChangedBy  string                 `json:"changed_by"`
    ChangedAt  time.Time              `json:"changed_at"`
    Changes    map[string]interface{} `json:"changes"`
    Reason     string                 `json:"reason"`
}

// PreviewRequest contains data for template preview generation
type PreviewRequest struct {
    TemplateID string                 `json:"template_id"`
    Version    *int                   `json:"version,omitempty"` // nil = active version
    Variables  map[string]string      `json:"variables"`
    Locale     string                 `json:"locale"`
    Channel    string                 `json:"channel"`
    UserData   map[string]interface{} `json:"user_data,omitempty"`
```

```
}


// PreviewResponse contains rendered template output for review
type PreviewResponse struct {
    Subject     string            `json:"subject"`
    Body        string            `json:"body"`
    ContentType string            `json:"content_type"`
    Warnings    []string          `json:"warnings"`
    Metadata    map[string]string `json:"metadata"`
    CharCount   int               `json:"char_count"` // Important for SMS
}
```

Core Template Rendering Logic (Skeleton for Implementation):

```go
// RenderTemplate transforms a template into channel-specific content
func (e *TemplateEngine) RenderTemplate(ctx context.Context, templateID string, variables map[string]string, user *User,
channel string) (*FormattedMessage, error) {
    // TODO 1: Get active template version from store
    // TODO 2: Detect user's preferred locale using localizer
    // TODO 3: Get best matching template content for user's locale
    // TODO 4: Validate variables against template requirements
    // TODO 5: Apply security policy to sanitize variable values
    // TODO 6: Render template content with variable substitution
    // TODO 7: Apply channel-specific formatting (HTML/plain text/SMS limits)
    // TODO 8: Validate final output meets channel constraints
    // TODO 9: Return FormattedMessage with subject, body, content type
    // Hint: Use html/template for HTML channels, text/template for plain text
    // Hint: Check SMS character limits and warn if message will be multi-segment
    // Hint: Apply different escaping for HTML vs plain text vs JSON channels
}

// GetContentForLocale selects the best template content for a user's locale
func (t *Template) GetContentForLocale(locale string) *TemplateContent {
    // TODO 1: Try exact locale match (e.g., "fr-CA")
    // TODO 2: Try language-only match (e.g., "fr")
    // TODO 3: Try configured default locale (usually "en-US")
    // TODO 4: Fall back to first available locale if nothing else matches
    // TODO 5: Log warning if falling back beyond language level
    // Hint: Store locale fallback chain in order of preference
    // Hint: Use consistent fallback rules across all template operations
}

// ValidateTemplateContent checks template security and correctness
func (v *DefaultContentValidator) ValidateTemplate(template *Template, channel string) error {
    // TODO 1: Parse template content to identify all variable placeholders
    // TODO 2: Check for forbidden HTML tags based on security policy
    // TODO 3: Verify template syntax is valid for template engine
    // TODO 4: Check channel-specific constraints (SMS character limits, etc.)
    // TODO 5: Validate that all required variables are documented
    // TODO 6: Check for potential XSS vectors in template structure
    // TODO 7: Return validation error with specific issues found
    // Hint: Use html/template.Parse() to validate template syntax
```

```
    // Hint: Channel validation rules differ - SMS has char limits, email allows HTML
}
```

Localization and Fallback Logic (Skeleton for Implementation):

```go
// DetectUserLocale determines the best locale for a user
func (l *DefaultLocalizationManager) DetectLocale(user *User) string {
    // TODO 1: Check user's explicit locale preference in profile
    // TODO 2: Parse locale string to extract language and region
    // TODO 3: Apply locale normalization (case, separator standards)
    // TODO 4: Fall back to system default if user locale is invalid
    // TODO 5: Log locale detection decision for debugging
    // Hint: Use language.Parse() from golang.org/x/text/language for parsing
    // Hint: Store default locale in configuration, typically "en-US"
    // Hint: Validate locale format before accepting user preferences
}

// FormatVariableForLocale applies culture-specific formatting to values
func (l *DefaultLocalizationManager) FormatVariable(value string, varType string, locale string) (string, error) {
    // TODO 1: Parse value according to variable type (date, currency, number)
    // TODO 2: Load locale-specific formatting rules
    // TODO 3: Apply appropriate formatting (date format, currency symbol, number separators)
    // TODO 4: Handle parsing errors gracefully with fallback formatting
    // TODO 5: Return formatted string suitable for template substitution
    // Hint: Use golang.org/x/text packages for number and date formatting
    // Hint: Cache formatters per locale to avoid repeated parsing
    // Hint: Define standard variable types: "date", "currency", "number", "text"
}
```

Template Versioning and Deployment (Skeleton for Implementation):

```go
// CreateTemplateVersion creates a new version of an existing template
func (s *DatabaseTemplateStore) CreateTemplateVersion(ctx context.Context, templateID string, content
map[string]TemplateContent, changedBy string, summary string) (*Template, error) {
    // TODO 1: Get current active template to determine next version number
    // TODO 2: Validate new template content using content validator
    // TODO 3: Create new Template with incremented version number
    // TODO 4: Store new template version in database with audit trail
    // TODO 5: Keep previous version as active until explicitly changed
    // TODO 6: Create audit log entry recording the version creation
    // TODO 7: Return new template version for preview/testing
    // Hint: Use database transactions to ensure version consistency
    // Hint: Validate content before incrementing version number
    // Hint: Store complete template content, not just diffs, for reliability
}

// ActivateTemplateVersion makes a specific version active for new notifications
func (s *DatabaseTemplateStore) ActivateTemplateVersion(ctx context.Context, templateID string, version int, changedBy string)
error {
    // TODO 1: Verify the requested template version exists
    // TODO 2: Update active_version pointer in database
    // TODO 3: Create audit log entry for activation
    // TODO 4: Clear any caches containing the old active version
    // TODO 5: Emit activation event for monitoring systems
    // Hint: Use database constraints to ensure only one version is active
    // Hint: Consider graceful cache invalidation to avoid disrupting in-flight notifications
    // Hint: Log activation events for compliance and troubleshooting
}

// RollbackTemplate reverts to the previous active template version
func (s *DatabaseTemplateStore) RollbackTemplate(ctx context.Context, templateID string, changedBy string, reason string)
error {
    // TODO 1: Get current active version from database
    // TODO 2: Find previous active version from audit log
    // TODO 3: Validate that rollback target version still exists
    // TODO 4: Update active_version to point to previous version
    // TODO 5: Create audit log entry explaining rollback reason
    // TODO 6: Emit rollback alert for monitoring systems
    // Hint: Track version activation history to enable intelligent rollbacks
    // Hint: Validate rollback safety - don't roll back if template structure changed
```

```
    // Hint: Consider automatic rollback triggers based on error rates
}
```

Milestone Checkpoint:

After implementing the template system, verify the following functionality:

1. **Template Rendering Test:**

```bash
curl -X POST http://localhost:8080/templates/preview \
  -H "Content-Type: application/json" \
  -d '{
    "template_id": "welcome_email",
    "variables": {"firstName": "Alice", "loginCount": "5"},
    "locale": "fr-CA",
    "channel": "email"
  }'
```

Expected: Returns rendered French email with substituted variables and proper HTML formatting.

2. **Locale Fallback Test:** Request a template in `pt-BR` when only `pt` and `en-US` versions exist. Should return Portuguese content, not English.
3. **Security Validation Test:** Submit template content containing `<script>alert('xss')</script>` in variables. Should be escaped in HTML output, stripped in plain text output.
4. **SMS Character Limit Test:** Render SMS template with long variable values. Should warn when approaching 160-character limit.
5. **Version Management Test:** Create new template version, activate it, then roll back to previous version. Verify audit trail records all changes.

Debugging Tips:

Symptom	Likely Cause	Diagnosis	Fix
Template renders as literal `{{variable}}`	Variable not found in substitution data	Check template variables vs. provided data	Add missing variables or update template
HTML email shows escaped tags `<b>`	Wrong escaping applied for content type	Check channel-specific formatting logic	Use html/template for HTML channels
SMS messages cost more than expected	Characters exceed single segment limit	Count characters with realistic variable data	Add length validation to SMS templates
Users receive English despite locale preference	Locale fallback chain incorrect	Check locale detection and fallback logic	Fix locale parsing and matching logic
Template changes don't appear in notifications	Version not activated after creation	Check active version pointer in database	Activate new version explicitly
Security scanner flags XSS in templates	Variables not properly escaped	Test with malicious variable content	Add proper context-aware escaping

User Preferences and Unsubscribe

> **Milestone(s):** Milestone 3 - User Preferences & Unsubscribe — implements user notification preferences with granular controls and compliance-friendly unsubscribe

Think of user preferences as a sophisticated filtering system, like setting up mail delivery instructions at your local post office. Just as you might tell the postal service "no junk mail on weekdays" or "hold all packages if I'm not home between 10 PM and 8 AM," notification preferences allow users to

control when, where, and what types of messages they receive. The key challenge is building a preference resolution system that elegantly handles multiple layers of control — global preferences, category-specific settings, channel-specific options, and time-based rules — while ensuring regulatory compliance for unsubscribe requests.

The preference system serves as the final gatekeeper before notification delivery. After the routing engine selects a channel and the template engine renders content, the preference resolution system makes the ultimate decision: should this specific notification reach this specific user through this specific channel at this specific time? This decision involves evaluating a complex hierarchy of user choices, from broad "never send me marketing emails" rules down to specific "no SMS between 9 PM and 7 AM" constraints.

Preference Storage Model

The **preference storage model** acts like a personal assistant who knows exactly how you want to be contacted in every situation. Rather than a simple on/off switch, the system maintains a rich hierarchy of preferences that captures the nuanced ways users want to control their notification experience. The challenge lies in designing a data model that is both flexible enough to handle diverse preference types and performant enough for real-time decision making during notification processing.

The foundation of the preference model is the `UserPreference` struct embedded within each `User` record. This design choice reflects the tight coupling between user identity and communication preferences — every user has exactly one preference configuration, and preferences cannot exist without a user. The preference data structure captures four distinct layers of control that work together to determine notification delivery eligibility.

Field Name	Type	Description
`GlobalOptOut`	`bool`	Master switch that blocks ALL notifications when true, except legally required transactional messages
`CategoryPreferences`	`map[string]CategoryPreference`	Per-category settings for marketing, transactional, alerts, social, and security notifications
`ChannelPreferences`	`map[string]ChannelPreference`	Per-channel settings for email, SMS, push, and in-app delivery methods
`QuietHoursEnabled`	`bool`	Whether to suppress non-urgent notifications during user's quiet hours window
`LastUpdated`	`time.Time`	Timestamp for preference change tracking and audit compliance
`ConsentSource`	`string`	How the user provided consent (signup_form, api_call, imported_list) for GDPR compliance
`UnsubscribeTokens`	`map[string]UnsubscribeToken`	Active unsubscribe tokens for one-click email unsubscribe links

The `CategoryPreference` structure allows users to control broad classes of notifications while still receiving critical communications. This granularity addresses the common user frustration of having to choose between "all notifications" or "no notifications" when they actually want "no marketing, but yes to security alerts."

Field Name	Type	Description
`Enabled`	`bool`	Whether this category of notifications is allowed for this user
`AllowedChannels`	`[]string`	Specific channels permitted for this category (e.g., security alerts only via SMS and push)
`MinimumPriority`	`Priority`	Lowest priority level that will be delivered for this category
`RateLimitPerDay`	`int`	Maximum number of notifications in this category per 24-hour period

The `ChannelPreference` structure captures channel-specific constraints that reflect the unique characteristics of each delivery method. SMS preferences might include cost-conscious settings, while email preferences might focus on frequency management.

Field Name	Type	Description
`Enabled`	`bool`	Whether this delivery channel is available for the user
`AllowedCategories`	`[]string`	Categories permitted on this channel (e.g., only transactional emails)
`QuietHoursRespected`	`bool`	Whether quiet hours apply to this channel (emergency alerts might override)
`MaxDailyCount`	`int`	Maximum notifications per day on this channel
`LastNotificationSent`	`time.Time`	For rate limiting calculations
`VerificationStatus`	`VerificationStatus`	Whether the channel endpoint is verified (email confirmed, phone number verified)

Decision: Hierarchical Preference Model

- **Context**: Users need granular control over notifications, but the system must efficiently evaluate preferences during high-volume notification processing
- **Options Considered**: Flat boolean flags per category/channel combination, rule-based preference engine, hierarchical layered model
- **Decision**: Hierarchical layered model with global → category → channel → time-based evaluation
- **Rationale**: Provides maximum user flexibility while maintaining predictable evaluation order and cache-friendly data access patterns
- **Consequences**: More complex preference resolution logic, but enables sophisticated user control and regulatory compliance

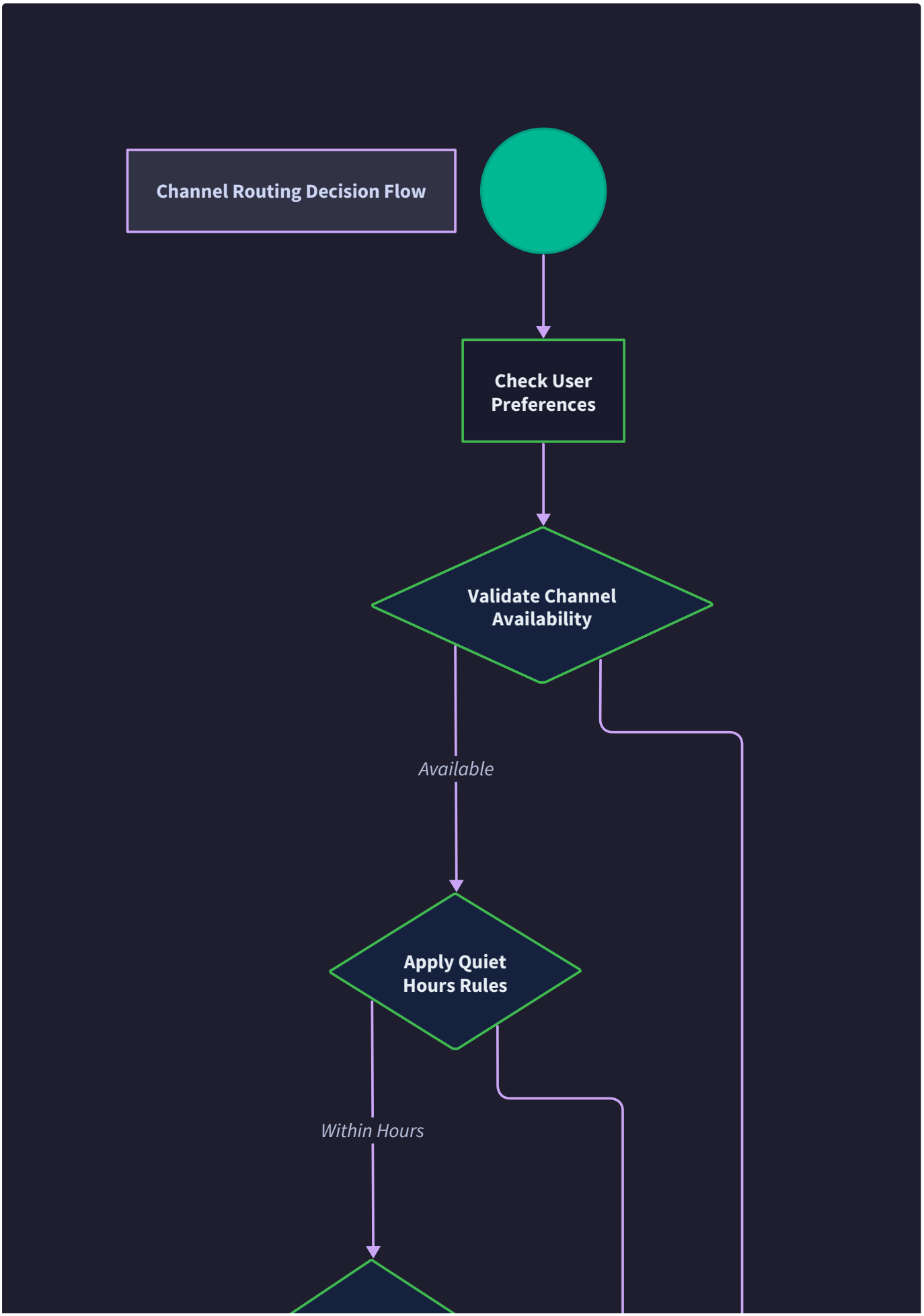
The preference resolution algorithm follows a cascading evaluation pattern that mirrors how users typically think about notification control. The system starts with the most restrictive settings and progressively applies more specific rules, ensuring that explicit user choices always take precedence over default behaviors.

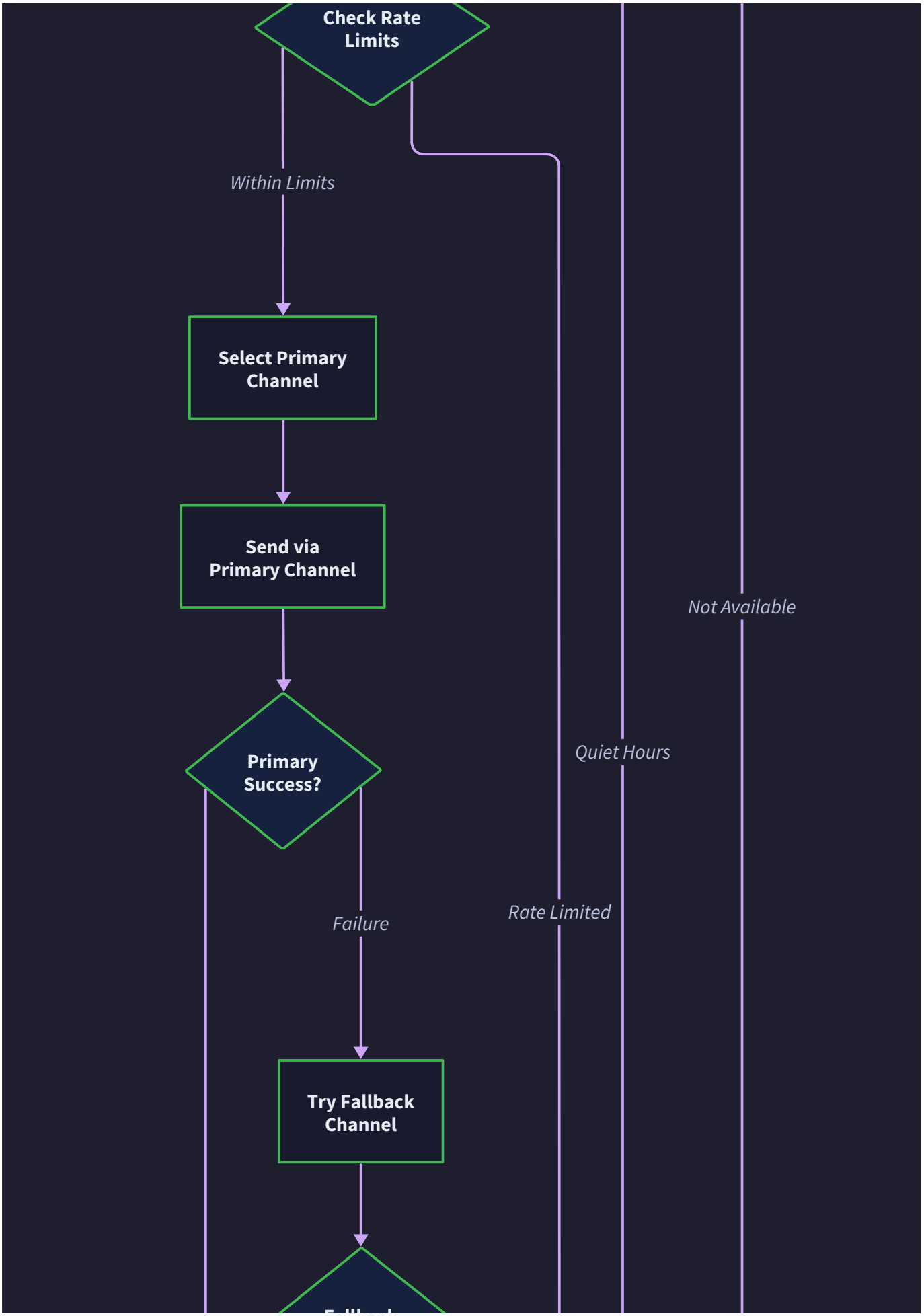
Preference Resolution Algorithm:

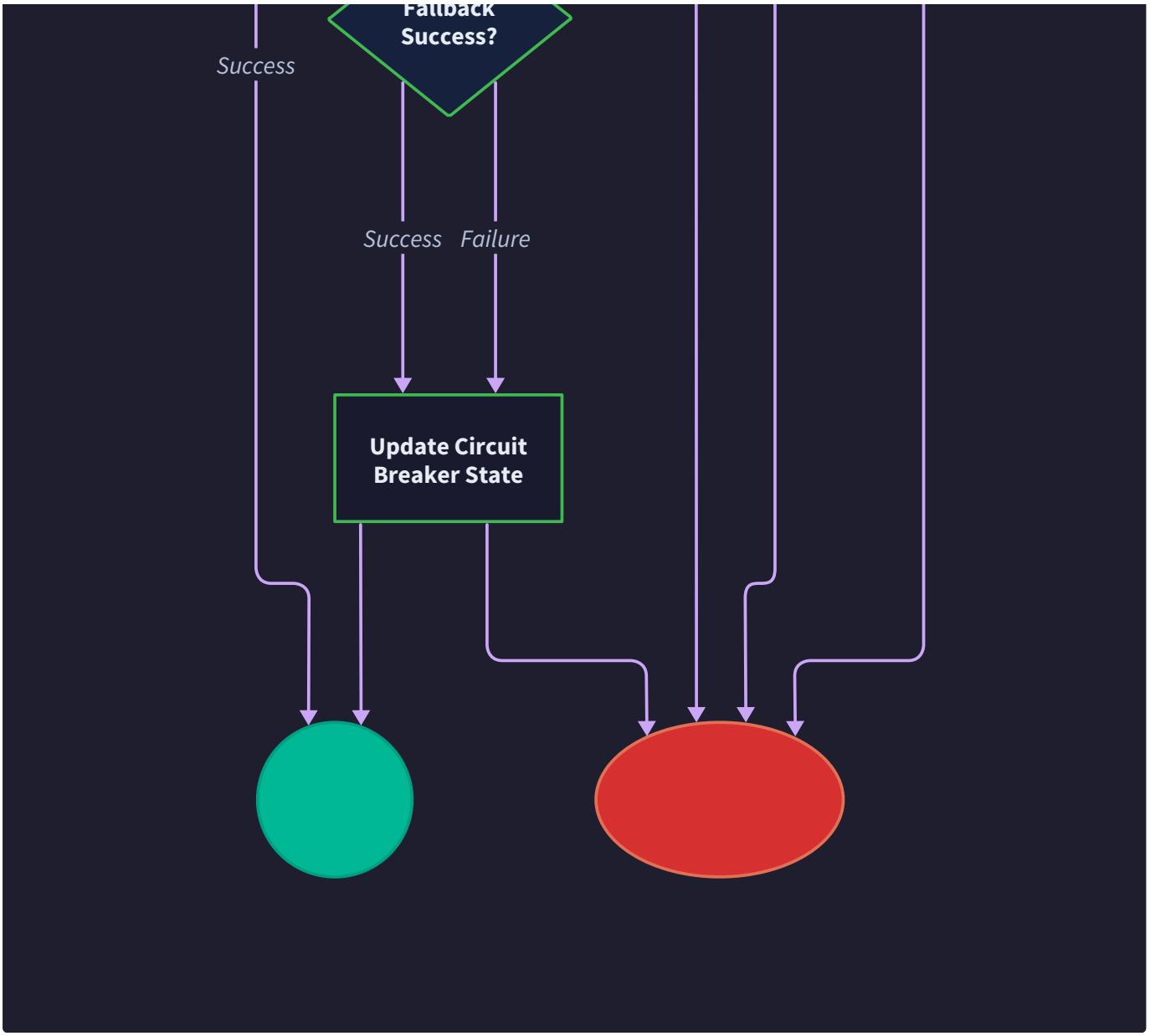
1. **Global Opt-Out Check**: If `GlobalOptOut` is true, immediately reject the notification unless it's marked as legally required transactional content (password resets, payment confirmations)
2. **Category Evaluation**: Check if the notification's category is enabled in `CategoryPreferences` and meets the minimum priority threshold
3. **Channel Compatibility**: Verify that the target channel is listed in the category's `AllowedChannels` and that the category is permitted in the channel's `AllowedCategories`
4. **Rate Limit Enforcement**: Count recent notifications in this category/channel combination against the user's daily limits
5. **Verification Requirements**: Ensure the target channel endpoint is verified if the category requires verified delivery
6. **Quiet Hours Assessment**: If the notification is non-urgent and quiet hours are enabled, check whether delivery should be deferred
7. **Final Authorization**: Generate a `RoutingPlan` with the approved channel or defer/reject decision

The system maintains preference change audit logs to support regulatory compliance and user transparency. Every preference modification generates an `AuditEntry` record that captures what changed, when, how, and optionally why (if the user provided a reason).

Field Name	Type	Description
`UserID`	`string`	User who made the preference change
`ChangeType`	`PreferenceChangeType`	Category of change (opt_out, channel_disable, quiet_hours, etc.)
`OldValue`	`string`	JSON representation of the previous preference state
`NewValue`	`string`	JSON representation of the updated preference state
`ChangeSource`	`string`	How the change was made (user_portal, api_call, unsubscribe_link, admin_action)
`Timestamp`	`time.Time`	When the change occurred
`IPAddress`	`string`	Source IP address for the change request
`UserAgent`	`string`	Browser/client information for web-based changes


Channel Routing Decision Flow
Check User Preferences
Validate Channel Availability
Available
Apply Quiet Hours Rules
Within Hours


Check Rate Limits
Within Limits
Select Primary Channel
Send via Primary Channel
Primary Success?
Failure
Try Fallback Channel
Rate Limited
Quiet Hours
Not Available



This comprehensive preference model enables sophisticated user control while maintaining the performance characteristics needed for real-time notification routing. The hierarchical structure ensures that user intent is clearly captured and consistently applied across all notification scenarios.

Unsubscribe System

The **unsubscribe system** functions like a diplomatic immunity mechanism — it provides users with immediate, unconditional protection from unwanted communications while ensuring that the organization meets all legal obligations for consent management. The system must balance user empowerment with regulatory compliance, providing multiple unsubscribe pathways while preventing abuse and maintaining audit trails for legal protection.

The core challenge in unsubscribe implementation lies in the tension between user experience and security. Users want one-click unsubscribe that works instantly, while the system needs to prevent malicious actors from unsubscribing users without consent and ensure that legitimate business communications continue to flow when legally permitted.

Unsubscribe Token Architecture:

The `UnsubscribeToken` structure provides cryptographically secure, time-limited unsubscribe capabilities that prevent enumeration attacks while enabling genuine users to opt out with minimal friction.

Field Name	Type	Description
`TokenID`	`string`	Unique identifier for this unsubscribe token
`UserID`	`string`	User this token belongs to (for verification)
`Category`	`string`	Notification category this token controls (marketing, alerts, social)
`Channel`	`string`	Specific channel to unsubscribe from (email, SMS, push, or "all")
`Scope`	`UnsubscribeScope`	Whether this unsubscribes from one campaign, category, or all notifications
`CreatedAt`	`time.Time`	When the token was generated
`ExpiresAt`	`time.Time`	Token expiration (typically 90 days)
`UsedAt`	`*time.Time`	When the token was used (nil if unused)
`Signature`	`string`	HMAC signature to prevent tampering

The unsubscribe token generation process uses HMAC signing to ensure tokens cannot be forged or modified. The signature includes the user ID, category, channel, and expiration timestamp, making it cryptographically impossible for attackers to generate valid unsubscribe tokens for arbitrary users.

Token Generation Algorithm:

1. **Generate Token ID**: Create a cryptographically random 32-character token identifier
2. **Set Expiration**: Calculate expiration timestamp (90 days from creation for marketing, 1 year for transactional)
3. **Create Payload**: Combine UserID + Category + Channel + ExpiresAt into a canonical string representation
4. **Generate Signature**: Compute HMAC-SHA256 of the payload using the system's secret key
5. **Store Token**: Persist the complete token record in the user's `UnsubscribeTokens` map
6. **Return URL**: Generate the unsubscribe URL containing the token ID and signature for email embedding

> **Decision: HMAC-Signed Unsubscribe Tokens**
>
> - **Context**: Need to prevent malicious unsubscribe attacks while providing one-click unsubscribe compliance with CAN-SPAM Act
> - **Options Considered**: Signed JWTs, database-only tokens with random IDs, HMAC-signed structured tokens
> - **Decision**: HMAC-signed structured tokens with database backing
> - **Rationale**: Provides cryptographic security against forgery while maintaining fast validation and audit trail capabilities
> - **Consequences**: Requires secure key management but prevents enumeration attacks and provides regulatory compliance

The unsubscribe URL generation creates RFC 8058 compliant one-click unsubscribe links that work across email clients while maintaining security. The URL structure embeds both the token ID and verification parameters needed for secure processing.

```
https://notifications.example.com/unsubscribe?token=abc123...&sig=def456...&category=marketing&quick=true
```

Unsubscribe Processing Workflow:

The unsubscribe request processing follows a defensive validation approach that assumes all incoming requests might be malicious while providing genuine users with immediate relief from unwanted notifications.

1. **Token Extraction**: Parse the token ID and signature from the URL parameters
2. **Token Lookup**: Retrieve the stored token record from the database using the token ID
3. **Signature Validation**: Recompute the HMAC signature and compare with the provided signature to detect tampering
4. **Expiration Check**: Verify the token hasn't expired (reject expired tokens for security)
5. **Usage Verification**: Ensure the token hasn't been used before (prevent replay attacks)
6. **Scope Application**: Apply the unsubscribe action according to the token's scope (category, channel, or global)
7. **Audit Logging**: Record the unsubscribe action with timestamp, IP address, and user agent for compliance
8. **Confirmation Response**: Show a confirmation page and optionally send a confirmation email (if still permitted)

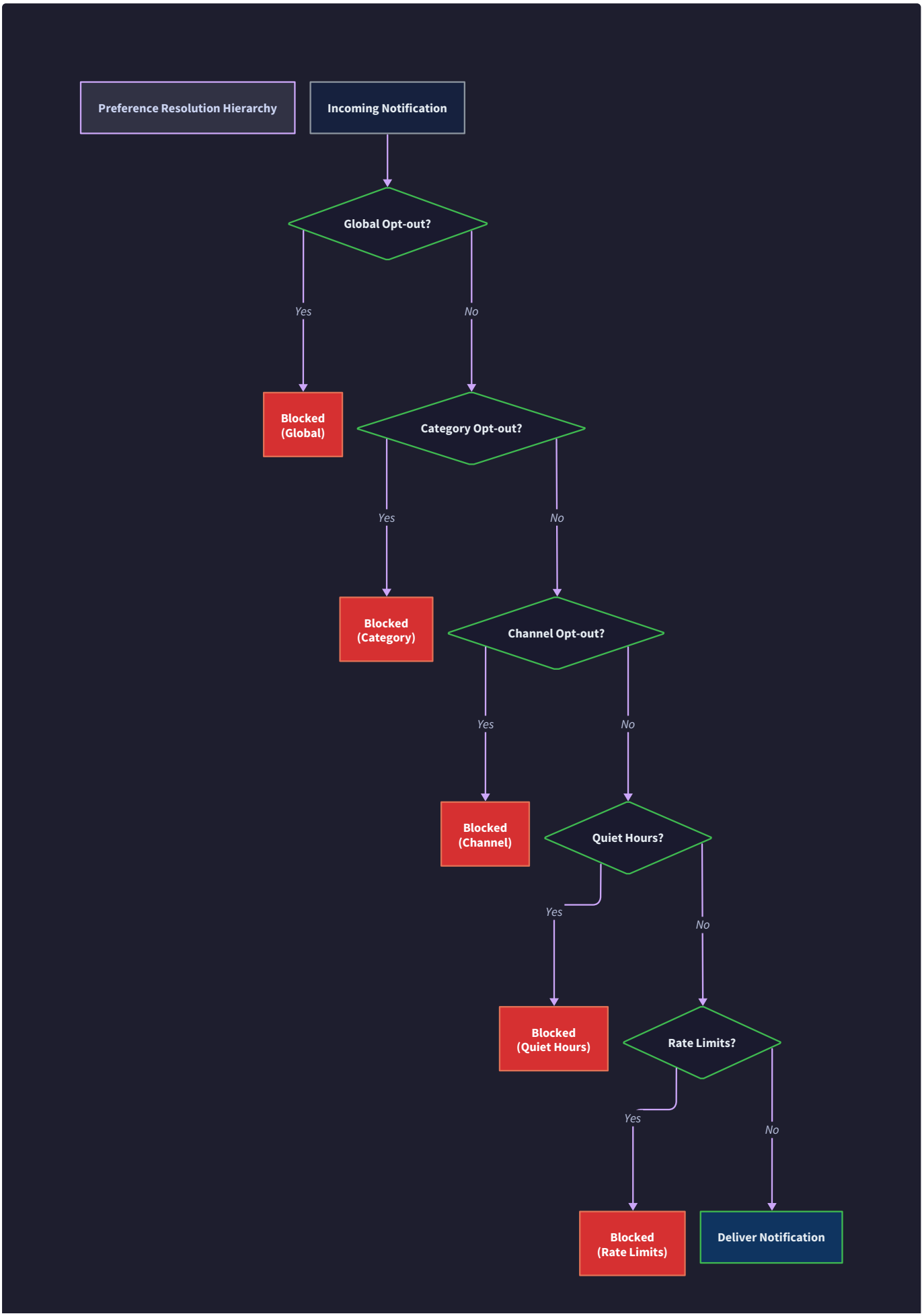
The system supports multiple unsubscribe scopes to handle different regulatory requirements and user expectations:

Scope	Description	Preference Impact	Legal Considerations
`Campaign`	Unsubscribe from specific email campaign	Adds campaign ID to blocked list	Minimal — user still opted in to category
`Category`	Unsubscribe from notification category	Sets category `Enabled` to false	Must honor for marketing, optional for transactional
`Channel`	Unsubscribe from delivery channel	Sets channel `Enabled` to false	Channel-specific — SMS unsubscribe doesn't affect email
`Global`	Unsubscribe from all communications	Sets `GlobalOptOut` to true	Must still allow legally required transactional emails

Regulatory Compliance Features:

The unsubscribe system implements specific compliance features for major email marketing regulations:

- **CAN-SPAM Compliance**: Processes unsubscribe requests within 10 business days (system processes immediately), includes physical mailing address in unsubscribe confirmations, maintains suppression list for minimum 30 days
- **GDPR Article 21**: Honors "right to object" for marketing communications, maintains consent withdrawal records, provides confirmation of processing within required timeframes
- **CASL Compliance** (Canada): Provides clear identification of sender, honors unsubscribe requests immediately, maintains records of consent withdrawal


Preference Resolution Hierarchy
Incoming Notification
Global Opt-out?
Yes
No
Blocked (Global)
Category Opt-out?
Yes
No
Blocked (Category)
Channel Opt-out?
Yes
No
Blocked (Channel)
Quiet Hours?
Yes
No
Blocked (Quiet Hours)
Rate Limits?
Yes
No
Blocked (Rate Limits)
Deliver Notification

The unsubscribe confirmation process balances user experience with legal requirements. The system immediately applies the unsubscribe preference change (providing instant relief) while generating appropriate confirmation communications that respect the user's newly expressed preferences.

Unsubscribe Confirmation Logic:

1. **Immediate Application**: Update user preferences immediately (no delay)
2. **Scope Assessment**: Determine what confirmations are still permitted under new preferences
3. **Confirmation Generation**: If email channel still allowed for transactional category, send confirmation email
4. **Alternative Confirmation**: If email blocked, show web confirmation page with download option for records
5. **Suppression List Update**: Add email/phone to category-specific suppression lists to prevent re-addition

Quiet Hours and Do-Not-Disturb

The **quiet hours system** operates like a sophisticated answering service that understands the difference between urgent and routine communications. Just as a good assistant knows to wake you for emergencies but hold non-critical calls until morning, the quiet hours system evaluates notification urgency against user-defined time windows to ensure important messages get through while respecting users' need for uninterrupted time.

The complexity of quiet hours implementation stems from the intersection of timezone handling, priority evaluation, and channel-specific behaviors. A user in Tokyo shouldn't receive marketing emails at 3 AM local time, but they should receive security alerts regardless of the hour. The system must make these nuanced decisions in real-time while handling users across multiple timezones.

Quiet Hours Data Model:

The quiet hours configuration is embedded within each `User` record to ensure timezone and preference data stay synchronized. This design choice reflects the personal nature of quiet hours — they're deeply tied to individual user schedules and cannot be meaningfully separated from user identity.

Field Name	Type	Description
`QuietHoursStart`	`string`	Start time in HH:MM format (24-hour, local timezone)
`QuietHoursEnd`	`string`	End time in HH:MM format (24-hour, local timezone)
`Timezone`	`string`	IANA timezone identifier (e.g., "America/New_York", "Asia/Tokyo")
`QuietHoursDays`	`[]time.Weekday`	Days of week when quiet hours apply (allows weekend-only quiet hours)
`EmergencyOverride`	`map[Priority]bool`	Priority levels that override quiet hours (typically PriorityUrgent)
`ChannelExceptions`	`map[string]bool`	Channels exempt from quiet hours (e.g., SMS for emergencies)

The quiet hours evaluation algorithm must handle complex timezone scenarios, including daylight saving time transitions, timezone changes when users travel, and edge cases like quiet hours that span midnight.

Quiet Hours Evaluation Algorithm:

1. **Current Time Calculation**: Get current UTC time and convert to user's local timezone using their `Timezone` field
2. **Day-of-Week Check**: Verify that quiet hours apply on the current day of the week
3. **Time Range Evaluation**: Check if current local time falls within the quiet hours window, handling midnight-spanning ranges
4. **Priority Override Assessment**: Check if the notification's priority level is configured to override quiet hours
5. **Channel Exception Evaluation**: Determine if the target delivery channel is exempt from quiet hours restrictions
6. **Urgency Classification**: Apply business rules to classify notification urgency beyond simple priority levels
7. **Defer or Deliver Decision**: Either schedule for post-quiet-hours delivery or proceed with immediate delivery

> **Decision: User-Timezone Based Quiet Hours**
>
> - **Context**: Users across global timezones need quiet hours respected based on their local time, not server time
> - **Options Considered**: Server timezone only, user timezone with fallback, user timezone required
> - **Decision**: User timezone required with system default fallback
> - **Rationale**: Quiet hours are meaningless without proper timezone handling — 10 PM in Tokyo is very different from 10 PM in New York
> - **Consequences**: Requires timezone data management and DST handling, but provides meaningful user experience

The system handles timezone edge cases through careful temporal arithmetic and fallback mechanisms:

Timezone Edge Case Handling:

Scenario	Detection	Resolution
Invalid Timezone	Timezone parsing fails	Fall back to system default timezone, log warning for user profile update
Daylight Saving Transition	Local time calculation encounters DST boundary	Use timezone library's DST-aware conversion, prefer standard time during ambiguous periods
Midnight-Spanning Quiet Hours	Start time > End time (e.g., 23:00 to 07:00)	Split into two ranges: start-to-midnight and midnight-to-end
User Timezone Change	User profile update changes timezone	Recompute all scheduled notifications in user's queue with new timezone
Leap Second/Leap Day	Date/time calculations encounter leap adjustments	Use standard library timezone handling, which manages leap adjustments automatically

The **notification deferral system** queues notifications that arrive during quiet hours for delivery at the next appropriate time. This system must balance timely delivery with respect for user preferences, ensuring that deferred notifications don't become stale or irrelevant.

Deferral Queue Management:

Field Name	Type	Description
`NotificationID`	`string`	Unique identifier for the deferred notification
`UserID`	`string`	Recipient user for timezone and preference lookup
`OriginalScheduledTime`	`time.Time`	When the notification was originally supposed to be delivered
`DeferredUntil`	`time.Time`	Earliest time the notification can be delivered (end of quiet hours)
`ExpirationTime`	`time.Time`	Latest time the notification is still relevant (prevents stale delivery)
`DeferralReason`	`string`	Why the notification was deferred (quiet_hours, rate_limit, channel_unavailable)
`RetryCount`	`int`	Number of delivery attempts made so far
`Priority`	`Priority`	Original notification priority for re-evaluation during delivery

The deferral queue processing runs continuously, evaluating deferred notifications for delivery eligibility. The system uses a priority queue ordered by `DeferredUntil` time to efficiently process notifications as they become deliverable.

Deferral Queue Processing Algorithm:

1. **Queue Scan**: Retrieve all notifications with `DeferredUntil` <= current time
2. **User State Refresh**: Reload user preferences and timezone settings (they might have changed)
3. **Re-evaluation**: Apply current preference resolution algorithm (user might have changed settings)
4. **Expiration Check**: Remove notifications that have passed their `ExpirationTime`
5. **Batch Delivery**: Group deliverable notifications by channel for efficient batch processing
6. **Failure Handling**: Re-defer notifications that fail delivery, with exponential backoff

The system provides users with transparency and control over their deferred notifications through a user portal that shows pending notifications and allows immediate delivery or cancellation.

User Deferral Control Interface:

Action	Description	Implementation
View Pending	Show all notifications deferred due to quiet hours	Query deferral queue for user's notifications with status and estimated delivery time
Deliver Now	Override quiet hours for specific deferred notification	Update `DeferredUntil` to current time and trigger immediate processing
Cancel Notification	Remove deferred notification without delivery	Mark notification as canceled and remove from queue
Adjust Quiet Hours	Modify quiet hours settings with immediate effect	Update user preferences and re-evaluate all deferred notifications
Emergency Mode	Temporarily disable quiet hours for urgent situations	Set temporary override flag that bypasses quiet hours for configurable duration

Common Pitfalls

⚠️ **Pitfall: Timezone Data Staleness** User timezone preferences can become outdated when users travel or relocate, leading to quiet hours being applied at incorrect times. The symptom is users complaining about receiving notifications during their local nighttime hours despite having quiet hours configured. To fix this, implement timezone validation during preference updates and provide clear timezone selection interfaces that show current local time for verification.

⚠️ **Pitfall: Transactional Email Unsubscribe** Allowing users to unsubscribe from legally required transactional emails (password resets, payment confirmations, security alerts) violates CAN-SPAM regulations and breaks critical user workflows. The symptom is users who can't receive password reset emails because they previously unsubscribed from "all emails." To fix this, classify notification categories as transactional vs. promotional and prevent unsubscribe for transactional categories while clearly explaining this to users.

⚠️ **Pitfall: Unsubscribe Token Enumeration** Using sequential or predictable unsubscribe token IDs allows attackers to unsubscribe arbitrary users by guessing token values. The symptom is legitimate users reporting they never unsubscribed but their preferences show opt-out actions. To fix this, use cryptographically random token IDs and HMAC signatures, and implement rate limiting on unsubscribe endpoints to prevent automated enumeration attacks.

⚠️ **Pitfall: Midnight-Spanning Quiet Hours Logic** Incorrectly handling quiet hours that span midnight (e.g., 11 PM to 7 AM) causes notifications to be delivered during restricted hours. The symptom is users receiving notifications between 11 PM and midnight when their quiet hours start at 11 PM. To fix this, split midnight-spanning ranges into two separate time windows and handle date rollover correctly in the evaluation logic.

⚠️ **Pitfall: Preference Inheritance Confusion** Complex preference hierarchies where category preferences, channel preferences, and quiet hours interact can create unexpected behavior where users think they've disabled notifications but still receive them through unexpected pathways. The symptom is user complaints about receiving notifications despite believing they opted out. To fix this, implement clear preference resolution documentation, provide preference preview functionality, and use explicit rather than inherited defaults.

Implementation Guidance

The user preference system requires careful attention to data consistency, regulatory compliance, and real-time performance. The following implementation provides production-ready components that handle the complex preference resolution logic while maintaining audit trails for compliance.

Technology Recommendations:

Component	Simple Option	Advanced Option
Preference Storage	PostgreSQL JSONB columns	PostgreSQL with separate preference tables
Timezone Handling	Go standard `time` package	`github.com/tkuchiki/go-timezone` library
HMAC Signing	Go `crypto/hmac` package	Hardware Security Module (HSM) integration
Audit Logging	Database table with JSON metadata	Elasticsearch with structured audit events
Deferred Queue	PostgreSQL with scheduled job processing	Redis with sorted sets for time-based queues

Recommended File Structure:

```
internal/preferences/
  preference_manager.go     ← main preference resolution logic
  preference_manager_test.go
  unsubscribe.go            ← unsubscribe token generation and processing
  unsubscribe_test.go
  quiet_hours.go            ← quiet hours evaluation and deferral
  quiet_hours_test.go
  audit.go                  ← preference change audit logging
  models.go                 ← data structures for preferences
  storage/
    postgres_store.go       ← PostgreSQL preference persistence
    postgres_store_test.go
```

Complete Preference Manager Implementation:

```go
package preferences

import (
    "context"
    "crypto/hmac"
    "crypto/rand"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "time"
)

// PreferenceManager handles user notification preferences and compliance
type PreferenceManager struct {
    store PreferenceStore
    hmacKey []byte
    auditLogger AuditLogger
    timeProvider TimeProvider // For testable time handling
}

// PreferenceStore defines persistence operations for user preferences
type PreferenceStore interface {
    GetUserPreferences(ctx context.Context, userID string) (*UserPreference, error)
    UpdateUserPreferences(ctx context.Context, userID string, prefs *UserPreference) error
    CreateUnsubscribeToken(ctx context.Context, token *UnsubscribeToken) error
    GetUnsubscribeToken(ctx context.Context, tokenID string) (*UnsubscribeToken, error)
    MarkTokenUsed(ctx context.Context, tokenID string, usedAt time.Time) error
}

// AuditLogger records preference changes for compliance
type AuditLogger interface {
    LogPreferenceChange(ctx context.Context, entry *AuditEntry) error
}

// TimeProvider enables testable time handling
type TimeProvider interface {
    Now() time.Time
}

// NewPreferenceManager creates a preference manager with required dependencies
```

```go
func NewPreferenceManager(store PreferenceStore, hmacKey []byte, auditLogger AuditLogger) *PreferenceManager {

    return &PreferenceManager{

        store: store,

        hmacKey: hmacKey,

        auditLogger: auditLogger,

        timeProvider: &RealTimeProvider{},

    }

}


// CanReceive evaluates whether a user can receive a specific notification

func (pm *PreferenceManager) CanReceive(ctx context.Context, userID string, notification *NotificationMessage, channel string)
(*DeliveryDecision, error) {

    // TODO 1: Load user preferences from store

    // TODO 2: Check global opt-out status - if true, only allow transactional notifications

    // TODO 3: Evaluate category preferences - check if notification category is enabled

    // TODO 4: Evaluate channel preferences - check if channel is enabled and category is allowed

    // TODO 5: Check rate limiting - count recent notifications against user limits

    // TODO 6: Evaluate quiet hours - defer non-urgent notifications if in quiet hours window

    // TODO 7: Return DeliveryDecision with Allow/Defer/Reject and reasoning



    panic("TODO: Implement preference evaluation logic")

}


// GenerateUnsubscribeToken creates a cryptographically secure unsubscribe token

func (pm *PreferenceManager) GenerateUnsubscribeToken(ctx context.Context, userID, category, channel string, scope
UnsubscribeScope) (*UnsubscribeToken, error) {

    // TODO 1: Generate cryptographically random token ID (32 characters)

    // TODO 2: Calculate expiration time based on scope (90 days for marketing, 1 year for transactional)

    // TODO 3: Create canonical payload string: userID + category + channel + expiration

    // TODO 4: Generate HMAC-SHA256 signature of payload using system secret key

    // TODO 5: Create UnsubscribeToken struct with all fields populated

    // TODO 6: Store token in database for later validation

    // TODO 7: Return complete token for URL generation



    panic("TODO: Implement token generation")

}


// ProcessUnsubscribe handles unsubscribe requests with security validation

func (pm *PreferenceManager) ProcessUnsubscribe(ctx context.Context, tokenID, signature string, clientIP, userAgent string)
(*UnsubscribeResult, error) {
```

```
    // TODO 1: Retrieve stored token from database using tokenID
    // TODO 2: Verify token hasn't expired or been used before
    // TODO 3: Regenerate HMAC signature and compare with provided signature
    // TODO 4: Apply unsubscribe action based on token scope (category/channel/global)
    // TODO 5: Mark token as used with current timestamp
    // TODO 6: Log audit entry with change details and client information
    // TODO 7: Return result indicating success/failure and next steps

    panic("TODO: Implement unsubscribe processing")
}

// IsInQuietHours determines if current time falls within user's quiet hours
func (pm *PreferenceManager) IsInQuietHours(ctx context.Context, user *User) (bool, error) {
    // TODO 1: Get current time in UTC
    // TODO 2: Parse user's timezone and convert current time to user's local timezone
    // TODO 3: Check if quiet hours are enabled and today is in configured days
    // TODO 4: Parse start and end times, handling midnight-spanning ranges
    // TODO 5: Determine if current local time falls within quiet hours window
    // TODO 6: Return boolean result with any parsing errors

    panic("TODO: Implement quiet hours evaluation")
}
```

Core Data Structures:

```go
// UnsubscribeScope defines the breadth of an unsubscribe action
type UnsubscribeScope string

const (
    ScopeCampaign  UnsubscribeScope = "campaign"
    ScopeCategory  UnsubscribeScope = "category"
    ScopeChannel   UnsubscribeScope = "channel"
    ScopeGlobal    UnsubscribeScope = "global"
)

// DeliveryDecision represents the outcome of preference evaluation
type DeliveryDecision struct {
    Action    DeliveryAction    `json:"action"`
    Reason    string           `json:"reason"`
    DeferUntil *time.Time      `json:"defer_until,omitempty"`
    RetryAfter *time.Duration  `json:"retry_after,omitempty"`
}

type DeliveryAction string

const (
    ActionAllow  DeliveryAction = "allow"
    ActionDefer  DeliveryAction = "defer"
    ActionReject DeliveryAction = "reject"
)

// UnsubscribeResult contains the outcome of processing an unsubscribe request
type UnsubscribeResult struct {
    Success       bool      `json:"success"`
    Message       string    `json:"message"`
    PreferencesUpdated map[string]interface{} `json:"preferences_updated"`
    ConfirmationSent bool    `json:"confirmation_sent"`
}
```

Quiet Hours Processing:

```go
package preferences

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

// QuietHoursEvaluator handles time-based notification suppression
type QuietHoursEvaluator struct {
    timeProvider TimeProvider
}

// EvaluateQuietHours determines if notification should be deferred due to quiet hours
func (qhe *QuietHoursEvaluator) EvaluateQuietHours(user *User, priority Priority) (*QuietHoursDecision, error) {
    // TODO 1: Check if quiet hours are enabled for this user
    // TODO 2: Load user's timezone and get current local time
    // TODO 3: Check if current day is in user's quiet hours days
    // TODO 4: Parse start and end times from HH:MM format
    // TODO 5: Handle midnight-spanning quiet hours (start > end)
    // TODO 6: Check if current local time falls in quiet hours window
    // TODO 7: Apply priority overrides and channel exceptions
    // TODO 8: Calculate defer until time if notification should be deferred

    panic("TODO: Implement quiet hours evaluation")
}

// ParseTimeInTimezone converts HH:MM time string to time.Time in specified timezone
func (qhe *QuietHoursEvaluator) ParseTimeInTimezone(timeStr, timezone string, referenceDate time.Time) (time.Time, error) {
    // TODO 1: Parse HH:MM format into hours and minutes integers
    // TODO 2: Load timezone location using time.LoadLocation
    // TODO 3: Create time.Time using reference date with parsed hours/minutes in user timezone
    // TODO 4: Handle parsing errors and invalid timezone names

    panic("TODO: Implement timezone-aware time parsing")
}

// QuietHoursDecision represents the result of quiet hours evaluation
```

```
type QuietHoursDecision struct {
    InQuietHours bool       `json:"in_quiet_hours"`
    ShouldDefer  bool       `json:"should_defer"`
    DeferUntil   *time.Time `json:"defer_until,omitempty"`
    Reason       string     `json:"reason"`
}
```

Milestone Checkpoint:

After implementing the preference system, verify the following behaviors:

1. **Preference Resolution**: Create a test user with mixed preferences (email enabled, SMS disabled, marketing category disabled). Send notifications and verify they're correctly filtered.
2. **Unsubscribe Flow**: Generate an unsubscribe token, create a test unsubscribe URL, and verify that clicking it properly updates preferences and shows confirmation.
3. **Quiet Hours**: Set a user's quiet hours to current time +/- 1 hour, send notifications of different priorities, and verify that non-urgent notifications are deferred until quiet hours end.
4. **Token Security**: Attempt to modify an unsubscribe token's signature or payload and verify that tampered tokens are rejected with appropriate error messages.

Run integration tests with: `go test ./internal/preferences/...`

Expected output should show all preference evaluation scenarios passing, including edge cases for timezone handling and midnight-spanning quiet hours.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Notifications delivered during quiet hours	Timezone calculation error or priority override	Log user's local time vs UTC time, check priority override settings	Verify timezone parsing and DST handling, review priority classification
Valid unsubscribe links showing "invalid token"	HMAC key mismatch or token expiration	Compare generated vs stored signatures, check token timestamps	Verify HMAC key consistency across services, extend token expiration if needed
Users can't unsubscribe from transactional emails	Incorrect category classification	Review notification category assignments and unsubscribe scope logic	Reclassify as promotional or add clear explanation why transactional emails can't be disabled
Preference changes not taking effect	Cache staleness or database replication lag	Check if preference updates are immediately visible in database	Add cache invalidation after preference updates, verify database write consistency

Delivery Tracking and Analytics

> **Milestone(s):** Milestone 4 - Delivery Tracking & Analytics — implements delivery status tracking, open/click tracking, bounce handling, and analytics dashboards

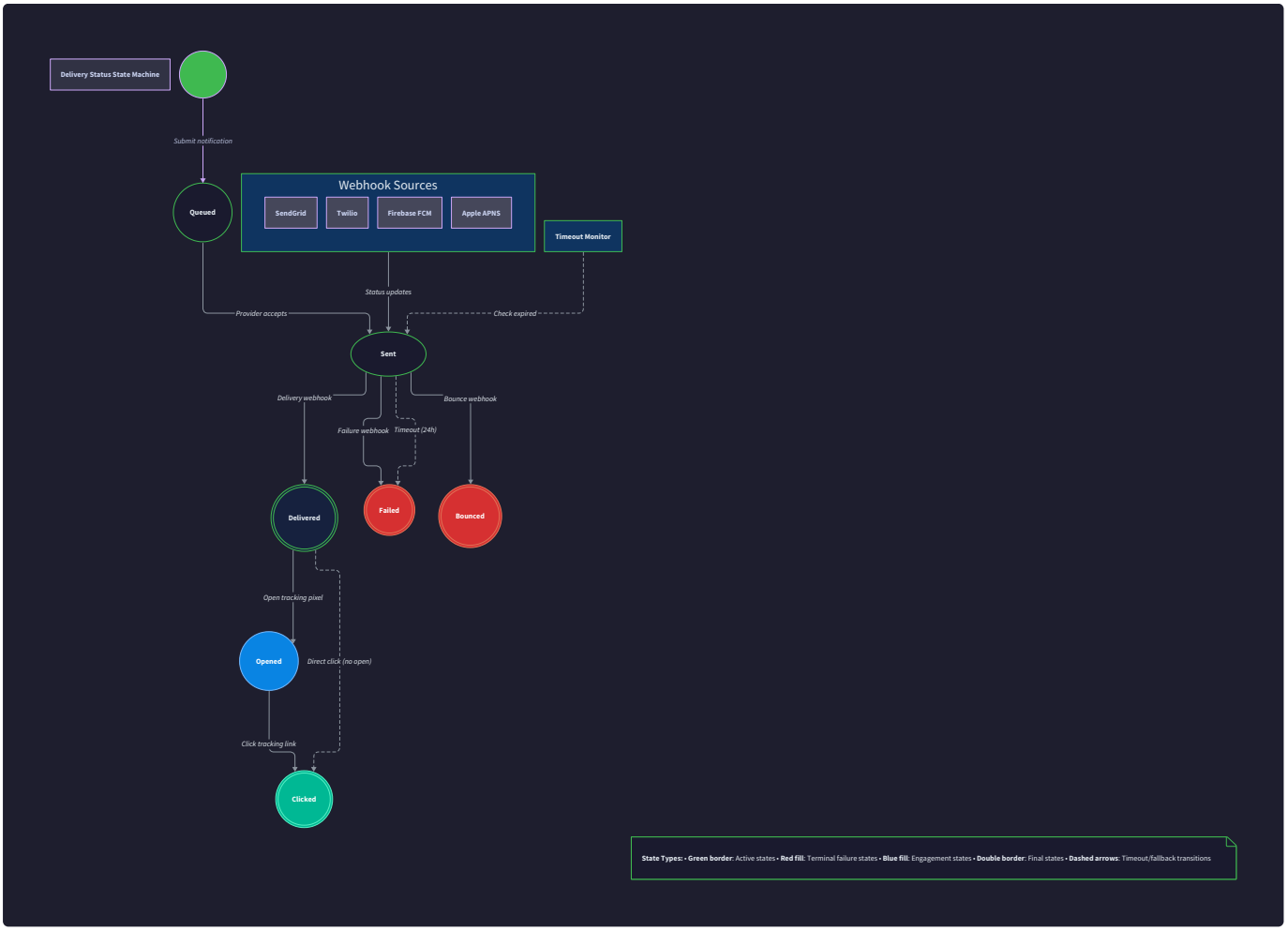
Think of delivery tracking like a package courier service that provides real-time updates on your shipment. When you send a package, you receive a tracking number that lets you monitor its journey: "picked up," "in transit," "out for delivery," "delivered," and sometimes "delivery failed" or "returned to sender." Similarly, a notification service must track each message through its complete lifecycle, from the moment it's submitted until it reaches the recipient (or definitively fails). Unlike package tracking, which relies on physical scans at distribution centers, notification delivery tracking depends on webhooks from external providers, pixel-based open detection, and URL click tracking to reconstruct the recipient's interaction with the message.

The challenge in notification delivery tracking lies in the asynchronous, distributed nature of the system. When your notification service hands off a message to an email provider like SendGrid, SMS provider like Twilio, or push notification service like Firebase Cloud Messaging, the actual delivery happens entirely outside your control. These providers use webhooks to report status updates back to your system, but webhook delivery itself can fail,

arrive out of order, or be delayed by hours. Meanwhile, your analytics dashboard needs to provide real-time insights into delivery rates, engagement metrics, and system health. This creates a complex orchestration problem where you must maintain consistent state across multiple external systems while handling network failures, provider outages, and malicious webhook attempts.

Delivery Status Tracking

The foundation of delivery tracking is a **state machine** that models the lifecycle of every notification from submission to completion. Think of this state machine like a flight departure board at an airport — each notification progresses through predictable stages, but external factors (weather delays, mechanical issues) can cause unexpected state transitions or prolonged stays in certain states.



The notification lifecycle begins when a `NotificationMessage` is submitted to the system and continues until the message reaches a terminal state. Unlike simple boolean success/failure tracking, this state machine captures the nuanced reality of notification delivery, where messages can be "sent" by your system but "undelivered" by the provider, or "delivered" to the recipient's device but never "opened" by the user.

Core Delivery States:

State	Description	Entry Condition	Exit Condition	Duration
`StatusQueued`	Message waiting for processing	Initial state when notification submitted	Routing engine selects channel	Seconds to minutes
`StatusSending`	Message being processed by channel handler	Channel begins formatting/sending	Provider API returns response	1-30 seconds
`StatusSent`	Message accepted by external provider	Provider returns 2xx HTTP response	Webhook reports delivery status	Minutes to hours
`StatusDelivered`	Message reached recipient's device/inbox	Provider webhook confirms delivery	User opens message (optional)	Immediate to days
`StatusOpened`	Recipient opened the message	Open tracking pixel loaded	User clicks link (optional)	Never to indefinite
`StatusClicked`	Recipient clicked embedded link	Click tracking URL accessed	Terminal state	Terminal
`StatusFailed`	Delivery failed permanently	Provider rejects message or timeout	Terminal state	Terminal
`StatusBounced`	Message bounced back to sender	Provider reports bounce	Terminal state	Terminal

The state machine handles both **synchronous transitions** (immediate responses from provider APIs) and **asynchronous transitions** (webhook-driven updates that arrive later). For example, when your email channel calls the SendGrid API to send a message, you immediately transition from `StatusQueued` to `StatusSent` if the API returns HTTP 202. However, the transition from `StatusSent` to `StatusDelivered` only occurs when SendGrid later sends a webhook to report successful delivery to the recipient's mailbox.

> **Decision: Separate Tracking Records from Core Notifications**
>
> - **Context**: Need to track delivery status without cluttering the core `Notification` entity with provider-specific details
> - **Options Considered**:
>   1. Add status fields directly to `Notification` struct
>   2. Create separate `DeliveryRecord` entities linked to notifications
>   3. Use event sourcing with status events
> - **Decision**: Separate `DeliveryRecord` entities with foreign key to `Notification`
> - **Rationale**: Allows multiple delivery attempts per notification (fallback channels), keeps core notification clean, enables detailed provider metadata storage without schema changes
> - **Consequences**: Requires join queries for status lookups, but provides flexibility for complex delivery scenarios and detailed analytics

DeliveryRecord Data Structure:

Field Name	Type	Description
`ID`	`string`	Unique identifier for this delivery attempt
`NotificationID`	`string`	Foreign key to parent notification
`Channel`	`string`	Channel used for delivery (email, sms, push)
`Provider`	`string`	External provider (sendgrid, twilio, fcm)
`ProviderMessageID`	`string`	Provider's tracking ID for this message
`Status`	`NotificationStatus`	Current delivery status
`AttemptNumber`	`int`	Delivery attempt sequence (1 for primary, 2+ for retries)
`SentAt`	`time.Time`	Timestamp when message sent to provider
`DeliveredAt`	`*time.Time`	Timestamp when delivered (null if not delivered)
`OpenedAt`	`*time.Time`	Timestamp when opened (null if not opened)
`ClickedAt`	`*time.Time`	Timestamp when first link clicked
`FailedAt`	`*time.Time`	Timestamp when permanently failed
`BouncedAt`	`*time.Time`	Timestamp when bounced
`ErrorMessage`	`string`	Failure reason if status is failed
`ProviderMetadata`	`map[string]interface{}`	Provider-specific delivery details
`CreatedAt`	`time.Time`	Record creation timestamp
`UpdatedAt`	`time.Time`	Last status update timestamp

The `ProviderMetadata` field captures provider-specific information that varies by channel. For email delivery via SendGrid, this might include SMTP response codes, recipient server details, and spam filter results. For SMS delivery via Twilio, it includes carrier information, message segments, and delivery confirmation details. For push notifications via FCM, it contains device registration token validity, app installation status, and notification priority handling results.

Status Update Algorithm:

The delivery tracking system processes status updates through a carefully ordered sequence that maintains data consistency even when webhooks arrive out of order or duplicate webhook calls occur:

1. **Validate webhook authenticity** using provider-specific signature verification (HMAC-SHA256 for most providers)
2. **Parse webhook payload** to extract notification ID, new status, timestamp, and provider metadata
3. **Locate existing delivery record** using provider message ID or notification ID depending on webhook format
4. **Check status transition validity** — prevent invalid backwards transitions (e.g., delivered → sent)
5. **Update delivery record atomically** with new status, timestamp, and metadata in single database transaction
6. **Trigger dependent processes** — update analytics aggregates, send user notifications for important status changes
7. **Acknowledge webhook receipt** with HTTP 200 response to prevent provider retries

> The critical insight here is that webhook processing must be **idempotent** — receiving the same webhook multiple times should not corrupt your delivery state. Providers often retry webhooks aggressively when they don't receive acknowledgment within their timeout window.

Common Pitfalls in Status Tracking:

⚠️ **Pitfall: Accepting Invalid State Transitions** Webhooks can arrive out of order due to network delays or provider retry logic. If you receive a "sent" webhook after you've already processed a "delivered" webhook for the same message, blindly updating the status would incorrectly regress the notification state. Always validate that status transitions move forward in the state machine before applying updates.

⚠️ **Pitfall: Missing Provider Message ID Correlation** Different providers use different correlation strategies in their webhooks. SendGrid includes their message ID in a custom header, Twilio embeds it in the JSON payload, and FCM uses a different field name entirely. Your webhook processor must handle these provider-specific correlation methods or you'll lose the ability to match status updates to the original notifications.

⚠️ **Pitfall: Ignoring Timezone Handling in Timestamps** Provider webhooks include timestamps in various formats and timezones. SendGrid uses Unix timestamps in UTC, Twilio uses ISO 8601 with timezone offsets, and some providers use their local timezone without indication. Always normalize webhook timestamps to UTC before storing them, or your delivery analytics will show impossible timing data.

Open and Click Tracking

Email open and click tracking transforms passive message delivery into measurable engagement metrics. Think of it like a retail store installing foot traffic sensors and heat maps — you're no longer just knowing whether someone received your marketing material, but whether they actually looked at it and took action. However, unlike physical store analytics, digital tracking must work within the constraints of email clients, user privacy settings, and technical limitations that make tracking imperfect and sometimes unreliable.

Open tracking relies on embedding an invisible 1x1 pixel image (called a **tracking pixel**) in HTML emails. When the recipient opens the email, their email client automatically loads images from your server, triggering a request that you can log as an "open" event. Click tracking works by replacing all links in the email with redirect URLs that pass through your tracking server before forwarding the user to their intended destination. This approach provides comprehensive engagement data but requires careful implementation to avoid breaking user experience or triggering spam filters.

> **Decision: Server-Side Link Rewriting vs Client-Side JavaScript**
>
> - **Context**: Need to track link clicks in emails to measure engagement and campaign effectiveness
> - **Options Considered**:
>   1. Rewrite all links to redirect through tracking server
>   2. Use JavaScript click handlers (doesn't work in email)
>   3. Hybrid approach with both server redirect and JS fallback
> - **Decision**: Server-side link rewriting with UTM parameter injection
> - **Rationale**: JavaScript is blocked/stripped by most email clients, server-side redirects work universally, UTM parameters provide attribution even if tracking fails
> - **Consequences**: Adds latency to user clicks, requires maintaining redirect service, but provides reliable cross-client tracking

Open Tracking Implementation:

Open tracking embeds a unique tracking pixel URL in every HTML email that identifies the specific notification and recipient. The pixel is implemented as a transparent GIF image with dimensions 1x1 pixels, making it invisible to recipients while still triggering the necessary HTTP request when the email is opened.

Component	Implementation	Purpose
Pixel URL	`https://track.example.com/open/{trackingID}.gif`	Unique URL per notification-recipient pair
Tracking ID	`base64(notificationID:recipientID:timestamp:hmac)`	Tamper-proof identifier with metadata
HMAC Signature	`SHA256(notificationID:recipientID:secretKey)`	Prevents tracking ID enumeration attacks
Response Handling	HTTP 200 + 1x1 transparent GIF bytes	Satisfies email client image request
Duplicate Detection	Track first open timestamp only	Avoid inflated metrics from multiple opens

The tracking pixel is embedded in the HTML email template as an IMG tag positioned at the end of the message body. The placement at the end ensures that the open is only recorded when the recipient has scrolled through or fully loaded the email content, providing a more accurate measure of actual engagement versus automatic email client prefetching.

Click Tracking Implementation:

Click tracking rewrites every link in the email to redirect through your tracking service before forwarding to the original destination. This provides comprehensive click analytics while maintaining the user's intended navigation experience.

Component	Implementation	Purpose
Redirect URL	`https://track.example.com/click/{linkID}`	Tracking endpoint that logs click and redirects
Link ID	`base64(originalURL:notificationID:linkIndex:hmac)`	Tamper-proof encoded original destination
UTM Parameters	`?utm_source=email&utm_campaign={campaignID}`	Attribution parameters added to destination
Geographic Logging	IP geolocation lookup on click	User location analytics
Device Detection	User-Agent parsing	Device and client analytics

The link rewriting process occurs during template rendering, after variable substitution but before final formatting. The system identifies all anchor tags (`<a href="...">`) in the HTML content, extracts the destination URLs, generates tracking URLs with embedded metadata, and replaces the original hrefs with the tracking versions.

Click Tracking Algorithm:

1. **Parse HTML content** using a proper HTML parser (not regex) to identify all anchor tags
2. **Extract destination URLs** from href attributes, handling both absolute and relative URLs
3. **Generate tracking IDs** by encoding original URL, notification ID, and link position with HMAC signature
4. **Create redirect URLs** pointing to tracking service with encoded destination information
5. **Add UTM parameters** to original URLs for attribution tracking in downstream analytics
6. **Replace href attributes** in HTML content with new tracking URLs
7. **Preserve link text and styling** to maintain email appearance and user experience
8. **Handle special cases** — skip tracking for unsubscribe links, mailto links, and tel links that shouldn't be redirected

Tracking Data Structures:

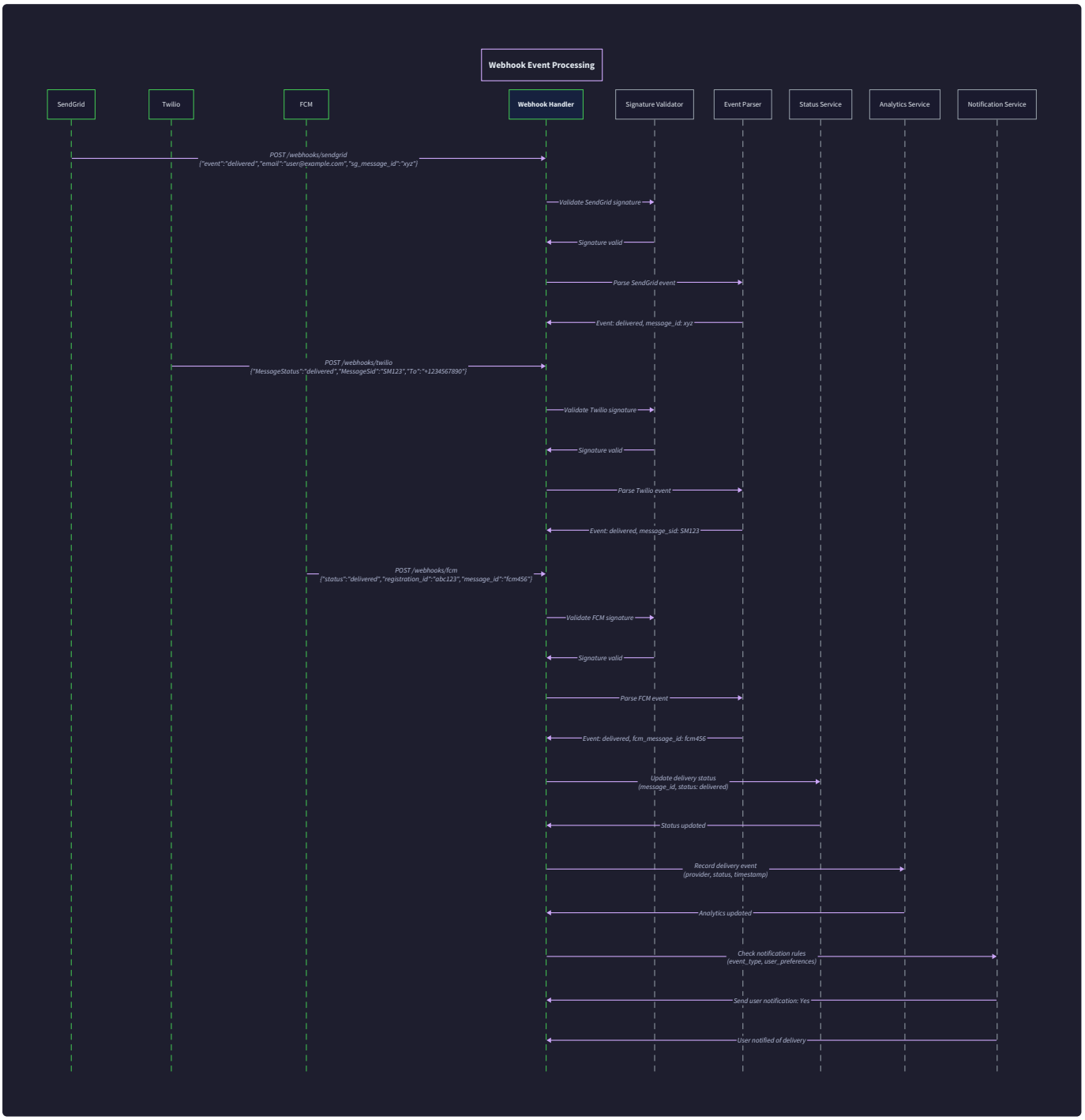
Field Name	Type	Description
`OpenEventID`	`string`	Unique identifier for this open event
`NotificationID`	`string`	Reference to tracked notification
`RecipientID`	`string`	User who opened the message
`OpenedAt`	`time.Time`	Timestamp of first open (ignore duplicates)
`UserAgent`	`string`	Email client information
`IPAddress`	`string`	Source IP address (for geolocation)
`EmailClient`	`string`	Parsed client name (Outlook, Gmail, etc.)

Field Name	Type	Description
`ClickEventID`	`string`	Unique identifier for this click event
`NotificationID`	`string`	Reference to tracked notification
`RecipientID`	`string`	User who clicked the link
`OriginalURL`	`string`	Destination URL before tracking rewrite
`ClickedAt`	`time.Time`	Timestamp when link was clicked
`LinkText`	`string`	Anchor text of the clicked link
`LinkPosition`	`int`	Link index within the email (0-based)
`UserAgent`	`string`	Browser/client information
`IPAddress`	`string`	Source IP address
`ReferrerURL`	`string`	HTTP referrer header

Privacy and Accuracy Considerations:

Modern email clients and privacy regulations significantly impact tracking accuracy. Apple's Mail Privacy Protection (introduced in iOS 15) pre-fetches tracking pixels on mail servers rather than when users actually open emails, inflating open rates by 20-30%. Gmail's image proxy loads tracking pixels through their servers, masking recipient IP addresses and making geolocation unreliable. Many corporate email systems block external images entirely, making open tracking impossible for those recipients.

> The key insight for tracking accuracy is to focus on **relative trends** rather than absolute numbers. A 15% open rate might actually represent 25% real opens due to privacy protection, but if that rate increases to 18% month-over-month, you can confidently conclude that engagement is improving.



Common Pitfalls in Open/Click Tracking:

⚠️ **Pitfall: Using Regex for HTML Link Parsing** Email HTML can be malformed, contain nested attributes, or use unusual quoting patterns that break regex-based link extraction. Always use a proper HTML parser library that handles edge cases correctly. Regex parsing will miss links with unusual formatting and potentially corrupt the HTML structure.

⚠️ **Pitfall: Not Handling URL Encoding in Redirect URLs**
When encoding original URLs into tracking redirects, you must properly URL-encode special characters or the redirect will break for URLs containing ampersands, question marks, or Unicode characters. Use your language's URL encoding libraries rather than manual string manipulation.

⚠️ **Pitfall: Tracking Unsubscribe Links** Never redirect unsubscribe links through click tracking, as this violates RFC 8058 requirements for one-click unsubscribe and can trigger spam filter penalties. Parse link destinations and exclude unsubscribe URLs from tracking rewrite.

Bounce and Failure Handling

Email bounces are like returned mail in the postal system — they indicate that a message could not be delivered to the intended recipient, but the specific reason determines whether you should retry delivery or remove the address from future mailings. Unlike postal returns, which provide a physical envelope with a clear reason stamp ("address unknown," "moved, no forwarding address"), email bounces come as structured data in provider webhooks that must be parsed and classified to determine the appropriate response.

The bounce handling system serves three critical functions: **protecting sender reputation** by removing invalid addresses, **maintaining clean recipient lists** to improve deliverability rates, and **providing actionable feedback** to application owners about data quality issues. Mishandling bounces can result in your sending domain being blacklisted by email providers, making it impossible to deliver even legitimate transactional messages.

Bounce Classification System:

Email bounces fall into two primary categories that require different handling strategies:

Bounce Type	Description	Examples	Retry Strategy	List Action
Hard Bounce	Permanent delivery failure	Invalid email address, domain doesn't exist, recipient account closed	Never retry	Remove from all lists immediately
Soft Bounce	Temporary delivery failure	Mailbox full, server temporarily unavailable, message too large	Retry with exponential backoff	Remove after repeated soft bounces
Block Bounce	Content or reputation block	Spam filter rejection, domain blacklisted, content policy violation	Manual investigation required	Suppress until resolved

Hard bounces indicate fundamental addressing problems that will never resolve — attempting to retry delivery to these addresses wastes resources and damages sender reputation. Soft bounces suggest temporary conditions that might resolve within hours or days, making retry attempts reasonable. Block bounces require human investigation to determine whether the issue is with message content, sender reputation, or recipient server configuration.

Bounce Processing Data Structure:

Field Name	Type	Description
`BounceEventID`	`string`	Unique identifier for this bounce event
`NotificationID`	`string`	Original notification that bounced
`RecipientEmail`	`string`	Email address that bounced
`BounceType`	`BounceType`	Hard, soft, or block classification
`BounceSubType`	`string`	Specific reason (mailbox-full, invalid-domain)
`ProviderBounceCode`	`string`	Provider-specific error code
`SMTPResponseCode`	`string`	SMTP server response (5.x.x format)
`BouncedAt`	`time.Time`	When bounce was detected
`DiagnosticInfo`	`string`	Human-readable error description
`RetryCount`	`int`	Number of delivery attempts made
`SuppressedUntil`	`*time.Time`	When to allow retry attempts (soft bounces)

The `BounceSubType` field captures granular bounce reasons that inform handling decisions. For example, a "mailbox-full" soft bounce should retry after a longer delay than a "message-too-large" soft bounce, which requires reducing the message size rather than simply waiting.

Bounce Processing Algorithm:

1. **Parse provider bounce webhook** to extract bounce classification, SMTP codes, and diagnostic information
2. **Classify bounce severity** using provider codes and SMTP response patterns to determine hard/soft/block type
3. **Update recipient reachability status** in user database to prevent future delivery attempts to hard bounce addresses
4. **Calculate retry schedule** for soft bounces using exponential backoff with jitter to avoid thundering herd
5. **Trigger sender reputation monitoring** if bounce rates exceed threshold levels indicating domain or content issues
6. **Log bounce details** for debugging and deliverability analysis, including original message content references
7. **Send internal alerts** when bounce patterns suggest systemic issues (domain blacklisting, configuration problems)

Recipient Suppression System:

The suppression system maintains a global registry of email addresses that should not receive future notifications due to bounces, spam complaints, or unsubscribe requests. This registry operates independently of user preferences to enforce deliverability protection at the infrastructure level.

Suppression Reason	Trigger Condition	Suppression Duration	Override Allowed
Hard Bounce	Single hard bounce event	Permanent	Manual verification only
Repeated Soft Bounce	5+ soft bounces in 30 days	90 days	Automatic after suppression period
Spam Complaint	Recipient marks email as spam	Permanent	Never (legal requirement)
Unsubscribe	User requests removal	Permanent	User re-subscription only
Domain Block	Entire domain blocks your sends	Until resolved	Manual investigation

Provider-Specific Bounce Handling:

Different email providers format bounce information differently, requiring provider-specific parsing logic:

Provider	Bounce Classification Source	SMTP Code Location	Diagnostic Info Format
SendGrid	`event` field in webhook JSON	`smtp-id` header	`reason` field with human description
Amazon SES	`bounceType` and `bounceSubType`	`bouncedRecipients.diagnosticCode`	AWS-specific error categories
Mailgun	`severity` field (temporary/permanent)	`delivery-status.code`	SMTP response in `delivery-status.description`
Postmark	`Type` field (HardBounce/SoftBounce)	`ErrorCode` integer	`Description` with detailed explanation

Spam Complaint Handling:

Spam complaints occur when recipients mark your emails as spam/junk in their email client. These complaints are reported back to your service through **feedback loops** provided by major email providers. Spam complaints are more damaging to sender reputation than bounces and require immediate action.

> **Critical compliance requirement**: When a recipient files a spam complaint, you must immediately suppress that email address from all future mailings. Continuing to send emails to addresses that have complained constitutes a violation of CAN-SPAM regulations and can result in significant legal penalties.

Common Pitfalls in Bounce Handling:

⚠️ **Pitfall: Retrying Hard Bounces** Hard bounces indicate permanent delivery failures that will never succeed, but many systems incorrectly treat them as temporary issues and continue retry attempts. This wastes system resources and rapidly damages your sender reputation. Always classify bounces correctly and never retry hard bounce addresses.

⚠️ **Pitfall: Ignoring SMTP Response Code Nuances** SMTP response codes like 5.1.1 (user unknown) and 5.1.2 (domain unknown) are both hard bounces but indicate different underlying issues. 5.1.1 suggests a data quality problem with specific addresses, while 5.1.2 indicates typos in domain

names. Tracking these distinctions helps identify systemic data collection issues.

⚠️ **Pitfall: Not Handling Provider API Rate Limits During Bounce Processing** When processing large bounce webhooks (especially after system outages), your bounce handler might need to make many API calls to update user records or send internal notifications. Failing to implement rate limiting can exhaust provider API quotas and cause secondary failures.

Analytics and Metrics

Notification analytics transforms raw delivery events into actionable business intelligence. Think of it like a restaurant analyzing not just how many meals were ordered, but how many were delivered hot, how many customers finished their plates, and which menu items drove repeat visits. The analytics system must process thousands of delivery events per minute to provide real-time dashboards while also supporting complex queries for campaign analysis and system optimization.

The analytics challenge lies in balancing **real-time responsiveness** with **historical accuracy**. Marketing teams want to see open rates updating live during campaign launches, while engineering teams need precise delivery failure analysis to debug system issues. This requires a hybrid approach that uses stream processing for real-time metrics and batch processing for accurate historical reporting.

> **Decision: Real-Time Streaming vs Batch Processing for Analytics**
>
> - **Context**: Need both immediate feedback during campaigns and accurate historical reporting for analysis
> - **Options Considered**:
>   1. Pure real-time streaming with eventual consistency
>   2. Pure batch processing with hourly/daily updates
>   3. Hybrid approach with streaming for dashboards, batch for accuracy
> - **Decision**: Hybrid architecture with streaming aggregates and nightly reconciliation
> - **Rationale**: Marketing needs real-time feedback during campaigns, but financial reporting requires exact numbers that streaming can't guarantee due to late-arriving events
> - **Consequences**: More complex architecture, but provides both immediate insights and eventual accuracy

Core Metrics Data Model:

The analytics system tracks metrics at multiple granularity levels to support different use cases — from high-level executive dashboards showing overall system health to detailed campaign analysis breaking down performance by recipient segment.

Metric Dimension	Examples	Aggregation Period	Use Case
System-Wide	Total sent, overall delivery rate	1min, 1hour, 1day	Infrastructure monitoring, SLA tracking
Channel-Specific	Email open rate, SMS delivery rate	5min, 1hour, 1day	Channel optimization, provider comparison
Campaign-Level	Campaign click rate, conversion rate	Real-time, final	Marketing campaign analysis
User Segment	Enterprise vs consumer engagement	1hour, 1day	Product strategy, personalization
Geographic	Delivery rates by country/region	1hour, 1day	Regional performance optimization
Provider-Specific	SendGrid vs Mailgun performance	1hour, 1day	Vendor relationship management

Real-Time Metrics Stream Processing:

The streaming analytics pipeline processes delivery events as they arrive to provide immediate feedback on campaign performance and system health. This pipeline uses approximate algorithms that trade perfect accuracy for low latency.

Stream Processing Stage	Input	Processing	Output
Event Ingestion	Webhook deliveries, open/click events	Parse, validate, enrich with metadata	Normalized event records
Real-Time Aggregation	Event stream	Sliding window counts, rate calculations	Live metric updates
Anomaly Detection	Metric time series	Statistical deviation analysis	Alert triggers
Dashboard Updates	Aggregated metrics	WebSocket broadcast to dashboards	Live UI updates

The streaming system uses **sliding window aggregation** to calculate metrics like "delivery rate over the last 15 minutes" that update continuously as new events arrive. This provides immediate feedback during campaign launches but may miss late-arriving events that affect final accuracy.

Batch Reconciliation Process:

The nightly batch process recalculates all metrics using complete event data to correct any streaming inaccuracies and generate authoritative reports for business analysis.

Batch Processing Stage	Input	Processing	Output
Event Collection	All delivery events for date range	Deduplication, timestamp ordering	Clean event dataset
Metric Recalculation	Clean events + user/campaign metadata	Precise aggregation with joins	Authoritative metrics
Variance Analysis	Streaming vs batch metric comparison	Statistical difference calculation	Accuracy monitoring
Report Generation	Final metrics + business dimensions	Templated report creation	Executive dashboards

Key Performance Indicators (KPIs):

KPI Category	Metric	Calculation	Target Range	Alert Threshold
Delivery Performance	Delivery Rate	Delivered / Sent	>95%	<90%
Engagement	Open Rate	Opened / Delivered	15-25%	<5% or >50%
Engagement	Click Rate	Clicked / Delivered	2-5%	<0.5%
System Health	Bounce Rate	Bounced / Sent	<5%	>10%
System Health	Failure Rate	Failed / Sent	<1%	>5%
Provider Performance	Provider Latency	Avg(sent_time - queued_time)	<30sec	>2min

The alert thresholds are designed to catch both underperformance (low engagement suggesting content issues) and anomalies (extremely high rates suggesting tracking bugs or data quality problems).

Analytics Query Interface:

The analytics system exposes a flexible query interface that supports both real-time dashboard updates and ad-hoc business intelligence queries.

Query Type	API Endpoint	Parameters	Response Format	Use Case
Real-time Metrics	`GET /analytics/realtime`	`metrics[], timespan`	JSON with current values	Live dashboards
Time Series	`GET /analytics/timeseries`	`metric, start, end, granularity`	JSON array of timestamped values	Trend analysis
Segmented Analysis	`GET /analytics/segments`	`dimensions[], filters[], daterange`	JSON with breakdown by segment	Campaign optimization
Funnel Analysis	`GET /analytics/funnel`	`steps[], cohort, timeframe`	JSON with conversion rates	User journey analysis

Performance Optimization Strategies:

Analytics queries must remain fast even when processing millions of delivery events. The system uses several optimization techniques:

Optimization	Implementation	Performance Impact	Trade-offs
Pre-aggregation	Materialized views with common groupings	10-100x faster queries	Storage overhead, update complexity
Time-based Partitioning	Partition tables by date	Faster historical queries	Complex partition management
Approximate Algorithms	HyperLogLog for unique counts	Constant memory usage	1-2% accuracy loss
Query Result Caching	Redis cache for common queries	Sub-second response times	Cache invalidation complexity

Data Retention and Archival:

Analytics data grows rapidly and requires tiered storage strategies to balance query performance with storage costs:

Data Age	Storage Tier	Granularity	Retention Policy	Access Pattern
0-7 days	Hot storage (SSD)	Event-level detail	Full retention	High-frequency queries
1-12 months	Warm storage	Hourly aggregates	Compressed storage	Regular reporting
1-7 years	Cold storage	Daily aggregates	Archived	Compliance, historical analysis
7+ years	Deep archive	Monthly summaries	Legal hold only	Rare access

Common Pitfalls in Analytics Implementation:

⚠️ **Pitfall: Double-Counting Events Due to Webhook Retries** Providers retry webhook deliveries when they don't receive timely acknowledgments, potentially causing the same delivery event to be counted multiple times in your metrics. Always implement idempotency checks using provider message IDs or event timestamps to deduplicate webhook processing.

⚠️ **Pitfall: Not Accounting for Timezone Differences in Time-based Metrics** When calculating daily metrics like "emails sent today," you must decide whether "today" means UTC today, the sender's timezone, or the recipient's timezone. Inconsistent timezone handling leads to confusing metrics where daily totals don't match expectations. Standardize on UTC for internal processing and convert to display timezones only in user interfaces.

⚠️ **Pitfall: Ignoring Late-Arriving Events in Real-Time Metrics** Network issues, provider delays, or system outages can cause delivery events to arrive hours or days after they actually occurred. Real-time metrics that don't account for these late arrivals will show artificially low performance that gradually improves as delayed events trickle in, confusing stakeholders about actual system performance.

Implementation Guidance

The delivery tracking and analytics system requires careful coordination between real-time event processing, persistent storage, and analytical queries. This implementation provides a foundation that handles the core tracking requirements while remaining extensible for advanced analytics features.

Technology Recommendations:

Component	Simple Option	Advanced Option
Event Storage	PostgreSQL with JSON columns	Apache Kafka + ClickHouse
Real-time Processing	Go channels with worker pools	Apache Flink or Apache Spark Streaming
Metrics Storage	PostgreSQL with time-series tables	InfluxDB or TimescaleDB
Analytics Queries	SQL with materialized views	Apache Druid or Amazon Redshift
Caching	Redis for query results	Redis Cluster with consistent hashing
Dashboards	Grafana with PostgreSQL datasource	Custom React dashboards with WebSocket updates

Recommended File Structure:

```
internal/tracking/
├── delivery/
│   ├── tracker.go              ← DeliveryTracker implementation
│   ├── states.go               ← Status transition logic
│   └── delivery_test.go        ← State machine tests
├── webhooks/
│   ├── processor.go            ← Webhook validation and routing
│   ├── providers/
│   │   ├── sendgrid.go         ← SendGrid webhook parser
│   │   ├── twilio.go           ← Twilio webhook parser
│   │   └── fcm.go              ← FCM webhook parser
│   └── webhooks_test.go        ← Webhook processing tests
├── analytics/
│   ├── aggregator.go           ← Real-time metrics aggregation
│   ├── queries.go              ← Analytics query interface
│   ├── reconciler.go           ← Batch reconciliation process
│   └── analytics_test.go       ← Metrics calculation tests
├── events/
│   ├── types.go                ← Event data structures
│   └── store.go                ← Event persistence
└── tracking_test.go            ← Integration tests
```

Core Event Types:

```go
package events

import "time"

// DeliveryEvent represents a status change in notification delivery
type DeliveryEvent struct {
    EventID          string                 `json:"event_id"`
    NotificationID   string                 `json:"notification_id"`
    RecipientID      string                 `json:"recipient_id"`
    EventType        string                 `json:"event_type"` // sent, delivered, opened, clicked, bounced, failed
    Channel          string                 `json:"channel"`
    Provider         string                 `json:"provider"`
    ProviderMessageID string                `json:"provider_message_id"`
    Timestamp        time.Time              `json:"timestamp"`
    Metadata         map[string]interface{} `json:"metadata"`
    Source           string                 `json:"source"` // webhook, api, internal
}

// OpenEvent tracks when recipients open email messages
type OpenEvent struct {
    EventID        string    `json:"event_id"`
    NotificationID string    `json:"notification_id"`
    RecipientID    string    `json:"recipient_id"`
    OpenedAt       time.Time `json:"opened_at"`
    UserAgent      string    `json:"user_agent"`
    IPAddress      string    `json:"ip_address"`
    EmailClient    string    `json:"email_client"`
    IsFirstOpen    bool      `json:"is_first_open"`
}

// ClickEvent tracks when recipients click links in messages
type ClickEvent struct {
    EventID        string    `json:"event_id"`
    NotificationID string    `json:"notification_id"`
    RecipientID    string    `json:"recipient_id"`
    OriginalURL    string    `json:"original_url"`
    ClickedAt      time.Time `json:"clicked_at"`
    LinkText       string    `json:"link_text"`
    LinkPosition   int       `json:"link_position"`
```

```
    UserAgent      string    `json:"user_agent"`
    IPAddress      string    `json:"ip_address"`
    ReferrerURL    string    `json:"referrer_url"`
}

// BounceEvent tracks delivery failures and their classification
type BounceEvent struct {
    EventID           string    `json:"event_id"`
    NotificationID    string    `json:"notification_id"`
    RecipientEmail    string    `json:"recipient_email"`
    BounceType        string    `json:"bounce_type"`        // hard, soft, block
    BounceSubType     string    `json:"bounce_sub_type"`    // invalid-domain, mailbox-full, etc.
    ProviderBounceCode string   `json:"provider_bounce_code"`
    SMTPResponseCode  string    `json:"smtp_response_code"`
    BouncedAt         time.Time `json:"bounced_at"`
    DiagnosticInfo    string    `json:"diagnostic_info"`
    RetryCount        int       `json:"retry_count"`
}
```

Delivery Tracker Core Logic:

```go
package delivery

import (
    "context"
    "time"
    "your-app/internal/tracking/events"
)

// DeliveryTracker manages notification delivery status and state transitions
type DeliveryTracker struct {
    eventStore    events.Store
    webhookSecret map[string]string // provider -> webhook secret
    analytics     *analytics.Aggregator
}

// UpdateDeliveryStatus processes status updates from providers and webhooks
func (dt *DeliveryTracker) UpdateDeliveryStatus(ctx context.Context, event *events.DeliveryEvent) error {
    // TODO 1: Validate event authenticity using provider webhook signatures
    // TODO 2: Check if this is a duplicate event by querying existing delivery records
    // TODO 3: Validate state transition is legal (can't go from delivered back to sent)
    // TODO 4: Update delivery record in database with new status and timestamp
    // TODO 5: If status is terminal (delivered, failed, bounced), update notification record
    // TODO 6: Trigger analytics aggregation for real-time metrics updates
    // TODO 7: Check if bounce rate threshold exceeded and trigger alerts
    // TODO 8: Return success to acknowledge webhook receipt
    // Hint: Use database transactions to ensure consistency between delivery record and analytics
}

// ProcessBounce handles bounce events with classification and suppression logic
func (dt *DeliveryTracker) ProcessBounce(ctx context.Context, bounce *events.BounceEvent) error {
    // TODO 1: Classify bounce type (hard/soft/block) based on SMTP codes and provider data
    // TODO 2: Update recipient suppression list if hard bounce or repeated soft bounce
    // TODO 3: Calculate retry schedule for soft bounces using exponential backoff
    // TODO 4: Log bounce details for debugging and sender reputation monitoring
    // TODO 5: Update delivery record with bounce information and failure reason
    // TODO 6: If bounce rate exceeds threshold, trigger domain reputation alerts
    // TODO 7: Send internal notification for manual investigation if block bounce
    // Hint: Different providers use different bounce classification schemes
}
```

```
// GetDeliveryStatus returns current status and history for a notification
func (dt *DeliveryTracker) GetDeliveryStatus(ctx context.Context, notificationID string) (*DeliveryStatus, error) {
    // TODO 1: Query delivery records for the notification ID
    // TODO 2: Get the most recent status from each delivery attempt
    // TODO 3: Include open/click events if available for email notifications
    // TODO 4: Calculate delivery timeline and provider performance metrics
    // TODO 5: Return structured status with state machine position and event history
}
```

Webhook Processing Infrastructure:

```go
package webhooks

import (
    "crypto/hmac"
    "crypto/sha256"
    "encoding/hex"
    "net/http"
    "your-app/internal/tracking/events"
)

// WebhookProcessor validates and routes provider webhooks to appropriate handlers
type WebhookProcessor struct {
    providers map[string]ProviderHandler
    tracker   *delivery.DeliveryTracker
    secrets   map[string]string // provider -> webhook secret
}

// ProviderHandler defines the interface for provider-specific webhook processing
type ProviderHandler interface {
    ValidateSignature(payload []byte, signature string) bool
    ParseEvents(payload []byte) ([]*events.DeliveryEvent, error)
    GetProviderName() string
}

// ProcessWebhook handles incoming webhook deliveries with signature validation
func (wp *WebhookProcessor) ProcessWebhook(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Identify provider from URL path or headers (SendGrid, Twilio, etc.)
    // TODO 2: Read webhook payload and extract provider signature header
    // TODO 3: Validate webhook signature using provider-specific HMAC verification
    // TODO 4: Parse webhook payload into normalized delivery events using provider handler
    // TODO 5: Process each event through delivery tracker to update status
    // TODO 6: Handle partial failures - some events succeed, others fail validation
    // TODO 7: Return HTTP 200 to acknowledge successful processing
    // TODO 8: Return HTTP 400 for signature validation failures (don't retry)
    // TODO 9: Return HTTP 500 for processing errors (provider should retry)
    // Hint: Always validate signatures before parsing to prevent malicious webhooks
}

// ValidateSignature verifies webhook authenticity using HMAC-SHA256
```

```go
func (wp *WebhookProcessor) ValidateSignature(provider string, payload []byte, signature string) bool {
    // TODO 1: Get webhook secret for the provider from configuration
    // TODO 2: Calculate HMAC-SHA256 of payload using provider secret
    // TODO 3: Compare calculated signature with received signature header
    // TODO 4: Use constant-time comparison to prevent timing attacks
    // TODO 5: Handle different signature formats (hex, base64) per provider
    // Hint: Each provider formats signature headers differently
}
```

Analytics Aggregation System:

```go
package analytics

import (
    "context"
    "sync"
    "time"
    "your-app/internal/tracking/events"
)

// Aggregator processes delivery events to generate real-time metrics
type Aggregator struct {
    metrics    map[string]*MetricAggregate
    metricsMux sync.RWMutex
    storage    MetricsStore
    alerts     AlertManager
}

// MetricAggregate holds real-time counters for a specific metric dimension
type MetricAggregate struct {
    Sent      int64     `json:"sent"`
    Delivered int64     `json:"delivered"`
    Opened    int64     `json:"opened"`
    Clicked   int64     `json:"clicked"`
    Bounced   int64     `json:"bounced"`
    Failed    int64     `json:"failed"`
    LastUpdate time.Time `json:"last_update"`
}

// ProcessEvent updates real-time metrics when delivery events occur
func (a *Aggregator) ProcessEvent(ctx context.Context, event *events.DeliveryEvent) error {
    // TODO 1: Extract metric dimensions from event (channel, provider, campaign)
    // TODO 2: Generate metric keys for different aggregation levels (system, channel, campaign)
    // TODO 3: Update in-memory counters for each relevant metric dimension
    // TODO 4: Use read-write locks to handle concurrent metric updates safely
    // TODO 5: Check if metric thresholds exceeded and trigger alerts
    // TODO 6: Persist metric updates to storage for dashboard queries
    // TODO 7: Broadcast metric updates to connected dashboards via WebSocket
    // Hint: Balance update frequency vs performance - batch updates when possible
}
```

```
// GetMetrics returns current aggregated metrics for specified dimensions
func (a *Aggregator) GetMetrics(ctx context.Context, dimensions []string, timespan time.Duration)
(map[string]*MetricAggregate, error) {
    // TODO 1: Validate requested dimensions and timespan parameters
    // TODO 2: Acquire read lock on metrics map to prevent data races
    // TODO 3: Filter metrics by requested dimensions and time window
    // TODO 4: Calculate derived metrics (rates, percentages) from raw counters
    // TODO 5: Format response with metric metadata and timestamps
    // TODO 6: Return metrics ordered by dimension hierarchy for consistent display
}

// ReconcileMetrics performs nightly batch processing for accurate historical metrics
func (a *Aggregator) ReconcileMetrics(ctx context.Context, date time.Time) error {
    // TODO 1: Query all delivery events for the specified date from persistent storage
    // TODO 2: Recalculate all metrics from raw events with complete data (no streaming approximations)
    // TODO 3: Compare batch-calculated metrics with streaming metrics to measure accuracy
    // TODO 4: Update historical metrics storage with authoritative batch-calculated values
    // TODO 5: Generate daily/weekly/monthly summary reports for business stakeholders
    // TODO 6: Clean up old streaming metric data to prevent unbounded memory growth
    // TODO 7: Schedule next reconciliation run and update processing status
    // Hint: This process corrects any inaccuracies from streaming approximations
}
```

Milestone Checkpoints:

After implementing the delivery tracking system, verify these behaviors:

1. **Status Transition Validation**: Submit a notification and manually trigger webhook calls with different status values. Verify that invalid transitions (delivered → sent) are rejected while valid transitions update the delivery record correctly.
2. **Open/Click Tracking**: Send an HTML email with tracking enabled, open it in a browser, and click embedded links. Check that open events and click events are recorded with correct timestamps and user agent information.
3. **Bounce Classification**: Use webhook testing tools to simulate hard bounce, soft bounce, and spam complaint events. Verify that hard bounces immediately suppress the recipient while soft bounces increment retry counters.
4. **Real-time Analytics**: Launch a campaign with 100+ recipients and watch the delivery metrics update in real-time. Verify that delivery rates, open rates, and click rates reflect actual recipient behavior within 30 seconds of events occurring.
5. **Provider Webhook Security**: Attempt to send webhooks with invalid signatures to your webhook endpoint. Verify that signature validation rejects unauthorized webhooks while accepting properly signed requests from legitimate providers.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Metrics show 0% delivery rate despite sent notifications	Webhook signature validation failing	Check webhook processor logs for signature errors	Verify webhook secrets match provider configuration
Open tracking shows impossibly high rates (>80%)	Apple Mail Privacy Protection prefetching	Analyze user agents in open events	Adjust metrics interpretation, focus on trends
Click tracking redirects fail	URL encoding issues in redirect links	Test tracking URLs manually in browser	Properly encode original URLs before embedding
Bounce handling not working	Provider bounce classification differences	Log raw webhook payloads to analyze structure	Implement provider-specific bounce parsing
Analytics queries timeout	Missing database indexes on large event tables	Examine query execution plans	Add indexes on notification_id, timestamp, channel columns
Real-time metrics lag behind events	Event processing bottleneck in aggregation	Monitor aggregator queue depths and processing times	Scale aggregator workers or implement batching

Interactions and Data Flow

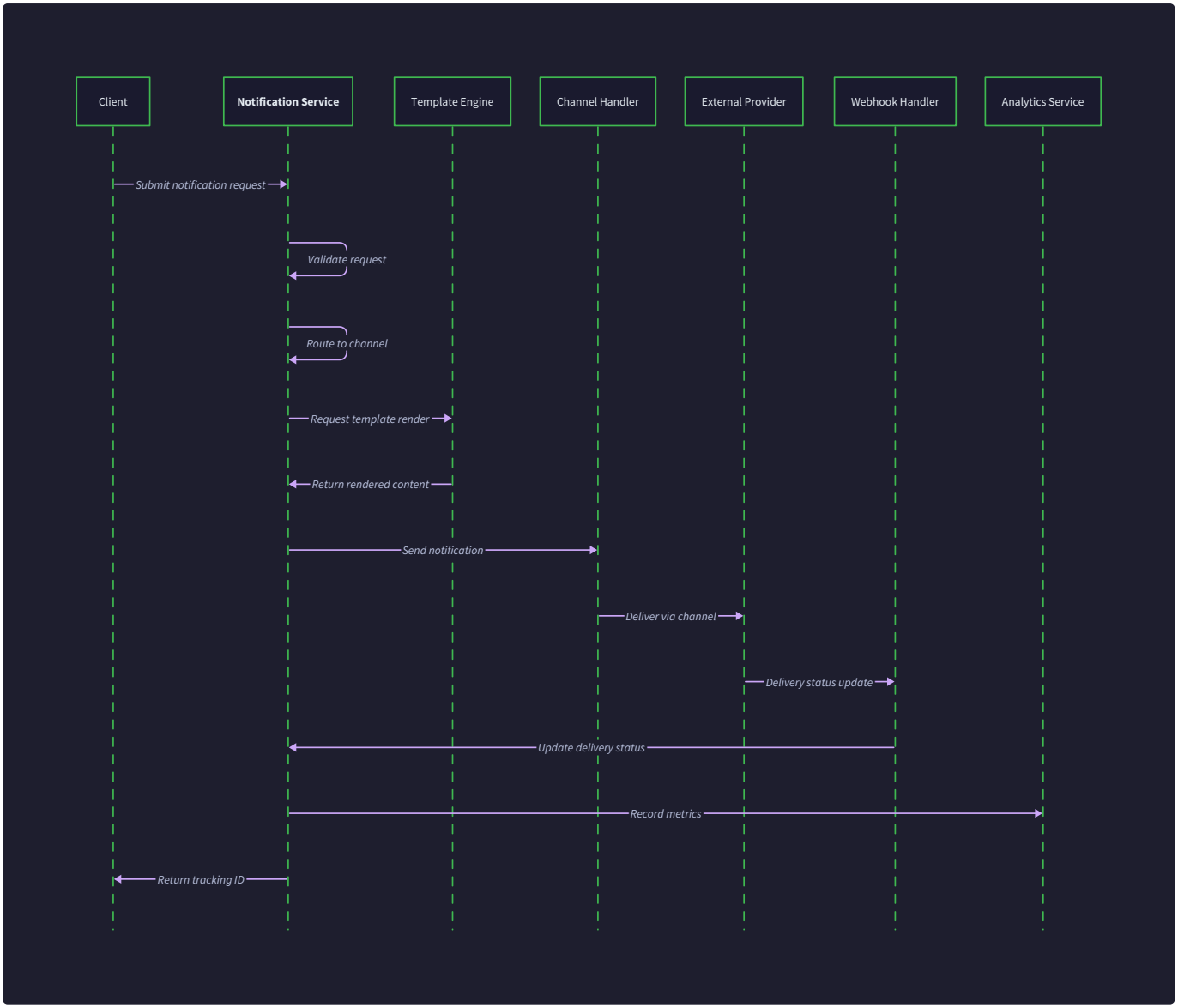
> **Milestone(s):** All milestones — demonstrates how components from channel abstraction (Milestone 1), template system (Milestone 2), user preferences (Milestone 3), and delivery tracking (Milestone 4) interact through message queues and REST APIs

Think of the notification system's data flow like a sophisticated mail processing center. When a package (notification) arrives, it goes through multiple stations: address validation, sorting by destination, packaging for the appropriate delivery method, routing to the correct truck, and finally tracking confirmation when delivered. Each station has its own specialized equipment and procedures, but they all communicate through a common conveyor belt system and status boards. The notification service follows this same pattern, with components communicating through message queues (the conveyor belt) and REST APIs (the status boards), while each component specializes in its particular processing step.

The interaction patterns in this system are designed around two primary communication mechanisms: asynchronous message passing for high-throughput notification processing, and webhook-based callbacks for real-time status updates from external providers. This hybrid approach maximizes throughput while providing timely delivery confirmations and failure notifications.

End-to-End Notification Flow

The complete notification processing flow involves eight distinct phases, each with specific responsibilities and failure handling requirements. Understanding this flow is crucial because it demonstrates how all the components we've designed work together to deliver a reliable notification service.



Phase 1: Notification Submission and Validation

The notification flow begins when a client application submits a notification request through the REST API. This initial phase handles authentication, input validation, and immediate feedback to the calling application. The API gateway serves as the entry point and performs several critical validation steps before accepting the notification for processing.

Think of this phase like a post office counter where a customer submits a package. The clerk checks that the address is readable, the package isn't too large, and the postage is correct before accepting it into the mail system. Similarly, the API gateway validates the notification structure, checks that the recipient exists, verifies the template is valid, and ensures the sender has permission to use the specified template.

The validation process follows a specific sequence designed to fail fast on common errors while minimizing database queries for performance. The system first validates the JSON structure and required fields, then checks template existence and permissions, followed by recipient validation and rate limiting enforcement.

Validation Step	Check Performed	Failure Response	Performance Impact
Schema Validation	JSON structure, required fields, data types	400 Bad Request with field errors	Minimal - no database calls
Template Existence	Template ID exists and is active	404 Template Not Found	Single database query with caching
Permission Check	Sender can use template category	403 Forbidden	Single database query with caching
Recipient Validation	User ID exists and is not globally opted out	404 User Not Found or 409 Opted Out	Single database query
Rate Limiting	Sender hasn't exceeded submission rate	429 Too Many Requests	Redis counter check
Priority Validation	Priority level allowed for template category	400 Invalid Priority	No additional queries

Upon successful validation, the system generates a unique notification ID and returns it immediately to the client. This provides fast feedback while allowing the actual processing to happen asynchronously. The notification is then wrapped in a `NotificationMessage` structure and published to the message queue for background processing.

> **Design Insight**: The notification ID is generated using a time-based UUID to ensure global uniqueness across multiple service instances while allowing approximate time-based sorting. This helps with debugging and provides natural ordering for dashboard displays.

Decision: Immediate ID Generation vs Deferred

- **Context**: Clients need a tracking identifier to reference the notification, but full processing takes time
- **Options Considered**:
  1. Generate ID after successful delivery
  2. Generate ID during queue processing
  3. Generate ID immediately upon submission
- **Decision**: Generate ID immediately upon submission
- **Rationale**: Provides immediate feedback to clients, enables tracking before delivery completes, and simplifies error handling in async processing
- **Consequences**: Requires cleanup of orphaned tracking records if processing fails, but enables better user experience and debugging

Phase 2: Queue Processing and Message Routing

Once the notification enters the message queue, the notification processor retrieves it for handling. This phase involves deserializing the message, loading the associated user and template data, and determining the appropriate processing path based on user preferences and notification characteristics.

The queue processor operates as a pool of worker goroutines, each capable of handling notifications independently. This design ensures high throughput while providing isolation between different notifications. If one notification encounters an error (such as a malformed template), it doesn't affect the processing of other notifications in the queue.

The message routing logic first loads the complete user profile, including preferences, timezone settings, and quiet hours configuration. This data determines whether the notification should be processed immediately, deferred to a later time, or rejected entirely based on user settings.

Routing Decision	Condition	Action Taken	Retry Behavior
Process Immediately	User allows category, outside quiet hours, not rate limited	Continue to template rendering	No retry needed
Defer for Quiet Hours	In user quiet hours window for non-urgent notification	Reschedule based on quiet hours end time	Single retry after quiet hours
Defer for Rate Limiting	User exceeded daily limit for category	Reschedule for next day	Single retry next day
Reject - Opt Out	User opted out of category or channel	Log rejection, update metrics	No retry - permanent rejection
Reject - Global Opt Out	User globally opted out	Log rejection, update metrics	No retry - permanent rejection
Defer - System Overload	Queue depth exceeds threshold	Reschedule with exponential backoff	Multiple retries with backoff

The routing engine also handles temporal aspects of notification delivery. For users in quiet hours, the system calculates the appropriate delivery time based on the user's timezone and quiet hours settings. This prevents notifications from waking users in the middle of the night while ensuring important communications still get delivered.

> **Critical Implementation Detail**: The queue processor must handle timezone calculations correctly by converting the user's quiet hours (stored as local time strings like "22:00" and "08:00") into absolute UTC timestamps for the current day. This prevents edge cases around daylight saving time transitions and ensures consistent behavior across geographic regions.

Phase 3: Template Rendering and Localization

The template rendering phase transforms the generic notification message into channel-specific content customized for the recipient's language and preferences. This involves loading the appropriate template version, detecting the user's locale, performing variable substitution, and applying security sanitization.

Think of this phase like a translation and formatting service at an international shipping company. When a package needs to be sent to France, they need to translate the shipping labels into French, format the address according to French postal standards, and ensure all customs documentation is properly completed. Similarly, the template engine takes the raw notification data and transforms it into properly formatted, localized content for each delivery channel.

The template engine follows a specific processing sequence to handle localization fallbacks gracefully. If the user's preferred locale (e.g., "pt-BR" for Brazilian Portuguese) isn't available, the system tries progressively broader matches ("pt" for Portuguese, then "en" for English as the default). This ensures users always receive understandable content even when perfect localization isn't available.

Template Processing Step	Input	Output	Error Handling
Template Loading	Template ID, version	Template structure with all locale variants	Fail notification if template not found
Locale Detection	User locale preference, Accept-Language header	Best matching locale string	Fall back to "en" if no match found
Content Selection	Template structure, detected locale	TemplateContent for specific locale	Use fallback locale chain if exact match missing
Variable Substitution	Template content, notification variables	Rendered content with variables replaced	Fail if required variables missing
Security Sanitization	Rendered content, output channel	Sanitized content safe for channel	Strip dangerous content, log security events
Channel Formatting	Sanitized content, target channel	FormattedMessage ready for delivery	Apply channel-specific formatting rules

The variable substitution process handles different data types appropriately for the target locale. Dates are formatted according to local conventions, numbers use appropriate decimal separators, and currency values include correct symbols and positioning. This attention to localization detail significantly improves user experience for international audiences.

Security sanitization occurs after variable substitution but before channel formatting to ensure that user-provided data can't introduce security vulnerabilities. For HTML email content, this means sanitizing HTML tags, removing dangerous attributes, and ensuring that URLs are properly encoded. For SMS content, this involves removing control characters and ensuring the content fits within SMS length limits.

Decision: Template Rendering Timing

- **Context**: Templates can be rendered when the notification is submitted, when it's processed from the queue, or when it's sent to each channel
- **Options Considered**:
  1. Render at submission time for immediate validation
  2. Render during queue processing for better error handling
  3. Render per-channel during sending for maximum flexibility
- **Decision**: Render during queue processing with per-channel formatting during sending
- **Rationale**: Balances validation feedback, error handling, and flexibility. Allows different channels to format the same content differently while catching template errors early
- **Consequences**: Requires storing rendered content temporarily, but enables better error messages and channel-specific optimization

Phase 4: Channel Selection and Routing Strategy

Once the content is rendered, the routing engine determines which channels should be used for delivery based on the notification's priority, the user's preferences, and the current health status of each channel provider. This phase implements the sophisticated fallback strategy that ensures important notifications get delivered even when preferred channels are experiencing issues.

The channel selection process operates like an intelligent mail sorting system that considers multiple factors: the recipient's delivery preferences, the current availability of different delivery methods, and the urgency of the message. Just as a shipping service might choose overnight delivery for urgent packages or ground shipping for routine deliveries, the notification system selects channels based on priority and reliability requirements.

The routing engine maintains real-time information about each channel's health through circuit breakers that track success rates and response times. When a channel starts failing frequently, the circuit breaker opens and routes traffic to alternative channels until the primary channel recovers. This prevents cascading failures and ensures high overall delivery rates.

Channel Selection Factor	Weight	Evaluation Criteria	Impact on Routing
User Channel Preference	High	User's preferred channel order for category	Primary determinant of channel priority
Channel Health Status	High	Circuit breaker state, recent failure rate	Skip unhealthy channels, prefer healthy ones
Notification Priority	Medium	Priority level vs minimum channel priority	High priority can override some preferences
Rate Limiting Status	Medium	Current rate limit usage for user/channel	Skip channels approaching rate limits
Channel Availability	Medium	Provider maintenance windows, quota limits	Skip temporarily unavailable channels
Cost Considerations	Low	Relative cost of channel (SMS > email > push)	Prefer cheaper channels when equivalent

The routing plan includes both primary and fallback channels to handle delivery failures gracefully. For high-priority notifications, the system might plan to try push notification first, fall back to SMS if the push fails, and finally attempt email as a last resort. Each channel in the plan includes specific retry parameters and timeout settings appropriate for that channel's characteristics.

The routing engine also considers delivery timing preferences when building the routing plan. If the user has configured quiet hours and the notification isn't urgent, the system includes timing constraints in the routing plan to defer delivery until appropriate hours. This ensures that fallback attempts also respect user preferences rather than bypassing them.

> **Performance Optimization**: The routing engine caches channel health information and user preferences to minimize database queries during high-traffic periods. Circuit breaker states are stored in Redis for fast access, while user preferences are cached with a TTL that balances performance with preference update responsiveness.

Phase 5: Channel-Specific Message Delivery

The actual message delivery phase involves invoking the appropriate channel handler with the formatted content and routing plan. Each channel handler implements the `Channel` interface but has dramatically different implementation details based on the underlying provider's API and delivery characteristics.

Think of this phase like the final mile delivery process where packages leave the central sorting facility and get loaded onto the appropriate delivery vehicles. An overnight envelope goes on an airplane, a local package goes on a delivery truck, and urgent documents might go with a courier. Each delivery method has its own protocols, tracking systems, and confirmation processes, but they all need to report back to the central system about delivery status.

Email delivery involves connecting to the SMTP provider (like SendGrid), constructing proper MIME messages with headers, handling authentication, and managing connection pooling for efficiency. SMS delivery requires interfacing with REST APIs (like Twilio), handling character encoding for international messages, and managing cost controls to prevent unexpected charges. Push notification delivery involves provider-specific protocols (FCM for Android, APNs for iOS) with device token management and payload size limitations.

Channel Type	Delivery Mechanism	Success Confirmation	Typical Latency	Failure Modes
Email (SMTP)	SMTP protocol with TLS encryption	Provider acceptance (not delivery)	1-5 seconds	Authentication failure, rate limiting, content rejection
SMS (REST API)	HTTP POST with JSON payload	Provider acceptance and message ID	2-10 seconds	Invalid phone number, insufficient credits, content filtering
Push (FCM/APNs)	Provider-specific HTTP/2 protocol	Provider acceptance and delivery	1-3 seconds	Invalid token, app uninstalled, payload too large
In-App (WebSocket)	WebSocket message to active sessions	Client acknowledgment	100-500ms	User offline, session expired, connection dropped

Each channel handler implements retry logic appropriate for the channel's characteristics. Email providers typically have strict rate limits but accept retries after short delays. SMS providers often have both rate limits and cost implications, requiring careful retry strategies. Push notification providers may invalidate device tokens, requiring the handler to detect and report invalid tokens for cleanup.

The delivery process includes comprehensive error handling that distinguishes between temporary failures (which should be retried) and permanent failures (which should trigger fallback channels). Temporary failures include network timeouts, rate limiting responses, and temporary provider outages. Permanent failures include invalid recipient addresses, authentication failures, and content policy violations.

Decision: Synchronous vs Asynchronous Channel Delivery

- **Context**: Channel handlers can make provider API calls synchronously or asynchronously to improve throughput
- **Options Considered**:
  1. Synchronous calls with connection pooling
  2. Asynchronous calls with callback handling
  3. Hybrid approach with fast/slow channel differentiation
- **Decision**: Synchronous calls with aggressive timeouts and connection pooling
- **Rationale**: Simpler error handling, easier to implement fallback logic, and sufficient performance with proper connection management. Async complexity not justified for typical notification volumes
- **Consequences**: May limit maximum throughput under extreme load, but provides predictable behavior and simpler debugging

Phase 6: Provider Integration and Response Handling

The provider integration layer handles the specifics of communicating with external notification services, each with unique API characteristics, authentication methods, and response formats. This layer abstracts the complexity of multiple providers behind the common `Channel` interface while handling provider-specific requirements like authentication token refresh, request signing, and response parsing.

The integration layer operates like a diplomatic corps that knows how to communicate with different countries according to their protocols and customs. Each provider has its own "language" (API format), authentication requirements, and response expectations. The integration layer translates between the universal notification format and each provider's specific requirements.

Provider authentication varies significantly across services. SendGrid uses API key headers, Twilio uses HTTP basic authentication with account credentials, and Firebase uses service account JSON Web Tokens with expiration handling. The integration layer manages these authentication mechanisms transparently, including automatic token refresh for JWT-based services.

Provider	Authentication Method	Request Format	Response Parsing	Rate Limit Handling
SendGrid	API key header	JSON POST to REST endpoint	JSON response with message ID	429 status with retry-after header
Twilio SMS	HTTP Basic Auth	Form-encoded POST	JSON response with SID	429 status with account limits
Firebase FCM	JWT Bearer token	JSON POST with auto-refresh	JSON response with success/failure	429 status with exponential backoff
Apple APNs	JWT with key rotation	HTTP/2 binary protocol	Structured response codes	Connection-level rate limiting

Response handling involves parsing provider-specific success and error responses into standardized `DeliveryResult` structures. This normalization allows the routing engine to make consistent decisions about retries and fallbacks regardless of which provider was used. The response parsing also extracts provider-specific metadata that might be useful for debugging or analytics.

Error classification is crucial for proper fallback handling. The integration layer categorizes provider errors into standard types: temporary failures that should be retried (rate limiting, temporary outages), permanent failures that should trigger fallbacks (invalid credentials, account suspended), and recipient-specific failures that should update user records (invalid email address, phone number disconnected).

> **Provider Reliability Consideration**: Different providers have different reliability characteristics and failure modes. Email providers typically have high availability but may delay delivery during high-volume periods. SMS providers can have regional outages affecting specific country codes. Push notification providers may experience device token churn requiring regular cleanup. The integration layer tracks these patterns and adjusts retry strategies accordingly.

The provider integration layer also handles webhook registration and management for providers that support delivery status callbacks. This involves registering webhook URLs, managing webhook authentication secrets, and handling webhook endpoint failures gracefully. Some providers require periodic webhook endpoint verification, which the integration layer handles automatically.

Phase 7: Fallback Channel Processing

When the primary channel fails to deliver a notification, the fallback processing logic activates to attempt delivery through alternative channels. This phase implements the sophisticated fallback strategy designed to maximize delivery success rates while respecting user preferences and cost considerations.

Fallback processing operates like a package delivery service that tries multiple delivery attempts through different methods. If the recipient isn't home for a regular delivery, they might leave a package at a pickup location, send a delivery notification, or schedule a redelivery. Similarly, if an email bounces, the system might try SMS, and if SMS fails, it might try push notification as a final attempt.

The fallback engine evaluates each potential fallback channel against the same criteria used for initial routing: user preferences, channel health, rate limits, and notification priority. However, fallback processing includes additional considerations like delivery urgency (how long since the original attempt), escalation rules (higher priority channels for repeated failures), and cost thresholds (avoiding expensive channels unless necessary).

Fallback Scenario	Primary Channel	Fallback Strategy	Timing Considerations	Success Criteria
Email Hard Bounce	Email	SMS → Push Notification	Immediate for urgent, 15min delay for normal	Any successful delivery
SMS Rate Limited	SMS	Email → Push Notification	Respect quiet hours for email	Successful delivery or all channels exhausted
Push Token Invalid	Push Notification	Email → SMS	Immediate for urgent notifications	Update token status, deliver via fallback
Provider Outage	Any	Next healthy provider → Alternative channel	Exponential backoff for same provider	Successful delivery or timeout reached
All Channels Failed	Multiple	Log failure, alert operations	No further attempts	Escalation to manual intervention

The fallback processing includes intelligent timing to avoid overwhelming users with duplicate notifications across multiple channels. If the primary channel fails quickly (within seconds), fallback attempts can proceed immediately. However, if the primary channel fails after a delay (indicating possible network issues), the fallback processing includes delays to prevent rapid-fire delivery attempts that could annoy users.

Fallback attempts respect all the same user preferences and quiet hours settings as primary delivery attempts. This means that if email fails and SMS is the fallback, but the user is in quiet hours, the SMS delivery will be deferred unless the notification has urgent priority that overrides quiet hours settings.

The fallback engine maintains detailed logging of all attempts and failures to support debugging and optimization. This includes tracking which fallback strategies are most effective for different types of failures, which providers have the most reliable fallback behavior, and which users experience frequent fallback scenarios that might indicate preference or contact information issues.

Decision: Fallback Attempt Limits

- **Context**: Fallback processing could potentially try many channels and providers, leading to excessive retry attempts
- **Options Considered**:
  1. Unlimited fallback attempts until success
  2. Fixed maximum number of attempts across all channels

3. Time-based limits with exponential backoff

- **Decision**: Maximum 3 total delivery attempts across all channels with time-based limits
- **Rationale**: Prevents infinite retry loops while allowing reasonable fallback attempts. Three attempts cover primary + two fallbacks, which handles most failure scenarios without becoming excessive
- **Consequences**: Some notifications may fail to deliver in extreme scenarios, but prevents system overload and user annoyance from excessive retry attempts

Phase 8: Delivery Confirmation and Status Updates

The final phase of notification processing involves confirming successful delivery and updating all relevant tracking systems with the final status. This phase ensures that the notification system maintains accurate records of all delivery attempts and provides reliable status information to clients and analytics systems.

Delivery confirmation operates like a package tracking system that updates the status at each stage of delivery. When a package is picked up, sorted, shipped, and delivered, each step generates a tracking update that customers can view. Similarly, the notification system updates the delivery status as the notification progresses through sending, delivery confirmation, and recipient engagement.

The confirmation process involves updating multiple data stores consistently to maintain system integrity. The primary delivery record gets updated with the final status, timing information, and any provider-specific metadata. The analytics system receives event updates for real-time metrics calculation. The user's notification history is updated for preference management and debugging purposes.

Status Update Target	Update Type	Timing	Consistency Requirements
Delivery Records Database	Final status, provider metadata	Immediately after delivery attempt	Strong consistency - must not lose delivery status
Analytics Event Stream	Delivery event with dimensions	Asynchronously via message queue	Eventual consistency - can tolerate brief delays
User Notification History	Status summary for user dashboard	Asynchronously via background job	Eventual consistency - user-facing only
Circuit Breaker State	Success/failure statistics	Immediately after attempt	Strong consistency - affects routing decisions
Rate Limiting Counters	Delivery count increment	Immediately after attempt	Strong consistency - affects rate limiting

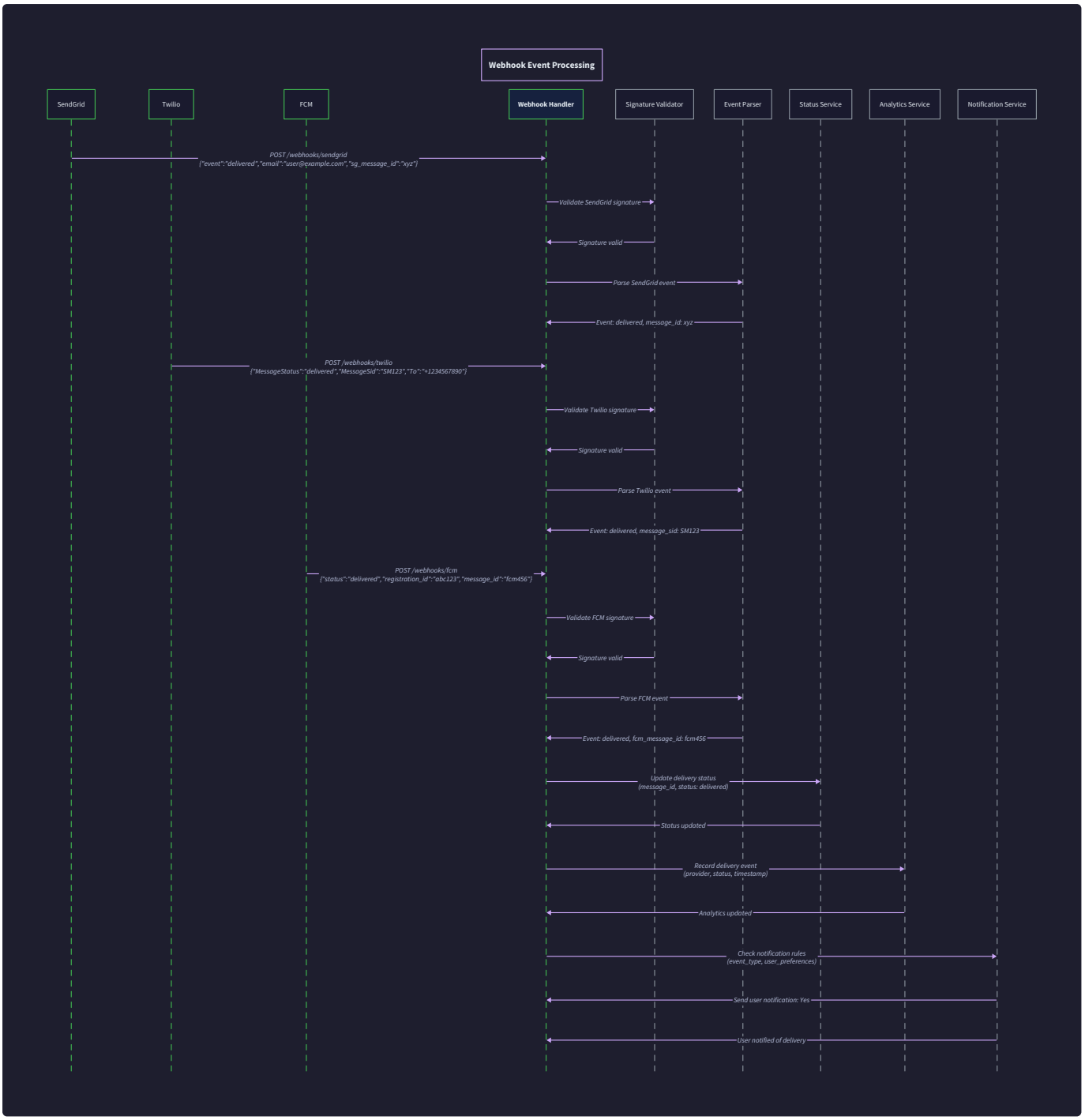
The delivery confirmation process includes handling of partial delivery scenarios that can occur with multi-recipient notifications or rich content delivery. For example, an email might be accepted by the provider but then bounce due to an invalid recipient address. The system tracks these nuanced delivery states and updates the status appropriately when webhook information becomes available.

Error handling during the confirmation phase is critical because delivery may have succeeded even if status updates fail. The system prioritizes delivering the notification over maintaining perfect tracking records, but implements retry mechanisms to ensure tracking updates eventually complete. Failed status updates are queued for retry processing to maintain data integrity without blocking notification delivery.

> **Data Consistency Insight**: The delivery confirmation process uses an eventually consistent model where immediate delivery success is recorded synchronously, but detailed analytics and user-facing status updates can be processed asynchronously. This ensures that critical routing decisions (like circuit breaker state) are based on current information while allowing non-critical updates to be processed more efficiently.

Webhook Processing

External notification providers use webhooks to report delivery status updates, bounce notifications, and engagement events (opens, clicks) back to the notification service. This asynchronous communication pattern allows the system to track the complete lifecycle of notifications without polling provider APIs continuously.



Think of webhook processing like a dispatch system at a delivery company that receives status reports from drivers in the field. When a driver successfully delivers a package, encounters a problem, or needs to report customer feedback, they radio back to dispatch with an update. The dispatch center needs to verify that the report is legitimate (from a real driver), understand the message format each driver uses, and update the appropriate tracking systems based on the report.

Webhook processing presents unique challenges because each provider uses different authentication methods, payload formats, and delivery guarantees. Some providers guarantee delivery with retry logic, while others use best-effort delivery that might miss events. Some providers send detailed event information, while others provide minimal status updates. The webhook processing system must handle this variability while maintaining consistent internal event records.

Webhook Security and Authentication

Webhook security is paramount because these endpoints are publicly accessible and can be targeted by malicious actors attempting to inject false delivery events or probe system vulnerabilities. Each provider implements different security mechanisms, and the webhook processor must validate authenticity rigorously before processing any events.

The security validation process operates like a secure communication system that verifies message authenticity through cryptographic signatures. Just as military communications use authentication codes to verify that messages come from legitimate sources, webhook processors use HMAC signatures to verify that webhook payloads come from the claimed provider and haven't been tampered with during transmission.

Most providers use HMAC-SHA256 signatures calculated from the raw request body using a shared secret key. However, the signature format, header name, and verification process varies significantly between providers. SendGrid includes the signature in the `X-Twilio-Email-Event-Webhook-Signature` header with a timestamp-based verification process. Twilio uses the `X-Twilio-Signature` header with URL and parameter verification. Firebase uses JWT tokens with public key verification.

Provider	Authentication Method	Signature Header	Verification Process	Security Considerations
SendGrid	HMAC-SHA256 with timestamp	X-Twilio-Email-Event-Webhook-Signature	Verify signature and timestamp freshness	Prevent replay attacks with timestamp validation
Twilio	HMAC-SHA1 with URL	X-Twilio-Signature	Include full URL in signature calculation	Validate against exact webhook URL
Firebase	JWT with public key	Authorization Bearer	Verify JWT signature and expiration	Handle key rotation and token refresh
Custom SMTP	Basic Auth or API key	Authorization	Provider-specific authentication	Implement per-provider security requirements

The webhook processor implements strict timing validation to prevent replay attacks where attackers capture legitimate webhooks and replay them later. Most providers include timestamp information in their signatures, and the processor rejects webhooks that are more than a few minutes old. This prevents attackers from using captured webhooks to inject false events.

Rate limiting on webhook endpoints protects against both accidental and malicious abuse. The system implements both per-provider and global rate limits to handle scenarios where a provider experiences issues and starts sending excessive webhook traffic. The rate limiting uses sliding window algorithms to provide smooth limiting without sudden traffic drops.

> **Security Best Practice**: Webhook endpoints should be dedicated URLs that don't serve other application traffic, should use HTTPS exclusively, and should implement comprehensive request logging for security monitoring. The webhook processor logs all requests (including invalid ones) with sufficient detail for security analysis without logging sensitive payload data.

Decision: Webhook Authentication Strategy

- **Context**: Different providers use incompatible authentication methods, requiring flexible authentication handling
- **Options Considered**:
  1. Require all providers to use the same authentication method
  2. Implement provider-specific authentication handlers
  3. Use a plugin-based authentication system
- **Decision**: Implement provider-specific authentication handlers with a common interface
- **Rationale**: Maximizes provider compatibility while maintaining security. Plugin approach adds unnecessary complexity for a finite set of providers
- **Consequences**: Requires implementing multiple authentication methods, but enables integration with any provider's authentication scheme

Event Parsing and Normalization

Once webhook authenticity is verified, the payload must be parsed and normalized into standard internal event structures. This process handles the significant variations in payload format, field naming, and event type classification across different providers.

Event parsing operates like a universal translator that converts different languages into a common format for processing. Each provider speaks its own "language" with unique vocabulary, grammar rules, and cultural conventions. The parsing system understands these different languages and produces standardized event records that the rest of the system can process uniformly.

Provider payload formats range from simple flat JSON structures to complex nested objects with extensive metadata. SendGrid sends arrays of event objects with flat structure and specific field names. Twilio sends individual event objects with nested status information. Firebase sends structured event objects with device-specific metadata. The parser must handle these format variations while extracting the essential information needed for delivery tracking.

Provider	Payload Format	Key Fields	Event Types Supported	Parsing Complexity
SendGrid	JSON array of flat event objects	email, timestamp, event, sg_event_id	processed, delivered, open, click, bounce, dropped	Medium - array handling, timestamp conversion
Twilio SMS	JSON object with nested status	MessageSid, MessageStatus, To, From	sent, delivered, failed, undelivered	Low - simple object structure
Firebase FCM	JSON object with registration info	registration_id, message_id, error	delivered, failed, canonical_id	Medium - error code interpretation
Apple APNs	HTTP/2 response codes	apns-id, apns-expiration	delivered, failed, invalid_token	High - binary protocol, response codes

Event type normalization is crucial because providers use different terminology for equivalent states. SendGrid uses "processed" to mean sent, "delivered" for successful delivery, and "bounce" for failures. Twilio uses "sent" for successful submission, "delivered" for confirmed delivery, and "failed" for permanent failures. Firebase uses delivery receipts for success and error codes for failures. The parser maps these provider-specific terms to standardized `NotificationStatus` values.

The parsing process includes comprehensive validation to ensure that parsed events contain all required information and that field values are reasonable. This includes validating timestamp formats, checking that message IDs match expected patterns, and verifying that recipient information matches system records. Invalid events are logged and rejected rather than processed with incomplete information.

Duplicate event detection is essential because some providers may send the same event multiple times, especially during network issues or provider-side retry scenarios. The parser uses provider-specific event IDs combined with message IDs to detect and ignore duplicate events. This prevents double-counting in analytics and avoids unnecessary status update processing.

> **Performance Optimization**: Event parsing includes extensive caching of template lookups and user information to minimize database queries during high-volume webhook processing. Provider-specific parsers are initialized once at startup rather than per-request to reduce parsing overhead.

Event Processing and Status Updates

Parsed and validated webhook events trigger comprehensive status updates across multiple system components. This processing ensures that delivery tracking, analytics, user preferences, and circuit breaker systems all reflect the current state based on provider feedback.

Event processing operates like a newsroom that receives reports from field correspondents and then updates all relevant stories, databases, and broadcasts with the new information. When a correspondent reports a development, editors update the main story, notify other departments that might be affected, update any related stories, and ensure the information flows to all appropriate distribution channels.

The event processing pipeline handles events in a specific order to maintain data consistency and avoid race conditions. Delivery status updates occur first to ensure that tracking queries return accurate information immediately. Analytics updates follow to maintain real-time dashboard accuracy. Finally, preference and suppression updates occur to affect future notifications based on delivery feedback.

Processing Step	Purpose	Timing Requirements	Failure Handling
Delivery Record Update	Update notification status and timing	Immediate - affects user-visible tracking	Retry with exponential backoff
Analytics Event Recording	Feed real-time metrics and dashboards	Near real-time - slight delays acceptable	Queue for batch processing if needed
Circuit Breaker Update	Adjust channel health based on delivery results	Immediate - affects routing decisions	Must succeed to prevent routing issues
Suppression List Update	Add hard bounces and complaints to suppression	Immediate - affects future deliveries	Critical - must not lose suppression data
User Preference Update	Handle unsubscribe links and preference changes	Immediate - regulatory compliance	Critical - must honor unsubscribe requests

Bounce event processing requires special handling because bounces indicate delivery issues that affect future notifications. Hard bounces (permanent delivery failures) trigger immediate addition to suppression lists to prevent future delivery attempts to invalid addresses. Soft bounces (temporary delivery failures) increment failure counters and may trigger suppression after repeated failures.

Engagement event processing (opens and clicks) provides valuable feedback about notification effectiveness and user engagement patterns. Open events update delivery records and feed analytics systems for engagement tracking. Click events provide detailed information about user interaction with notification content, including which links were clicked and when.

The event processing system implements comprehensive error handling to ensure that webhook processing failures don't lose important delivery information. Failed processing attempts are queued for retry with exponential backoff. Critical events like bounces and unsubscribes receive special retry handling to ensure compliance with email regulations.

Decision: Event Processing Guarantees

- **Context**: Webhook events contain critical delivery information that affects compliance and user experience
- **Options Considered**:
  1. Best-effort processing with logging of failures
  2. At-least-once processing with idempotency handling
  3. Exactly-once processing with complex coordination
- **Decision**: At-least-once processing with idempotency keys for critical events
- **Rationale**: Provides strong guarantees for critical events (bounces, unsubscribes) while allowing efficient processing of routine events (delivery confirmations). Exactly-once complexity not justified for most event types
- **Consequences**: Requires idempotency handling in downstream systems, but ensures critical events are never lost due to processing failures

Common Pitfalls

⚠️ **Pitfall: Webhook Endpoint Timeout Issues** Webhook processing can appear to work during testing but fail in production when providers implement aggressive timeout requirements. SendGrid expects webhook responses within 10 seconds, while Twilio allows only 15 seconds. If event processing involves slow database operations or external API calls, the provider will timeout and retry the webhook, leading to duplicate event processing.

Why it's wrong: Provider timeouts cause webhook retries that can overwhelm the system and create duplicate events in analytics and tracking systems.

How to fix: Implement a two-phase webhook processing pattern: acknowledge the webhook immediately with a 200 response, then queue the actual event processing asynchronously. Store the raw webhook payload in a processing queue and return success immediately, then process events from the queue with appropriate retry logic.

⚠️ **Pitfall: Message Queue Poison Messages** When notification processing encounters malformed data or corrupted messages, the default behavior is often to retry processing indefinitely. This creates "poison messages" that consume worker threads and prevent other notifications from processing. Common causes include malformed JSON in notification payloads, invalid user IDs that don't exist in the database, or template IDs that reference deleted templates.

Why it's wrong: Poison messages can deadlock entire notification processing pipelines, causing significant delivery delays for all notifications.

How to fix: Implement dead letter queue handling with maximum retry limits. After a configurable number of retry attempts (typically 3-5), move failed messages to a dead letter queue for manual investigation. Include comprehensive validation in message processing to detect and reject obviously invalid messages early.

⚠️ **Pitfall: Circuit Breaker State Synchronization** When running multiple notification service instances, circuit breaker state must be synchronized across instances to be effective. If each instance maintains independent circuit breaker state, a failing provider might appear healthy to some instances while failing for others, leading to inconsistent routing behavior and continued failures.

Why it's wrong: Inconsistent circuit breaker state defeats the purpose of failure detection and causes some instances to continue routing to failing providers while others correctly avoid them.

How to fix: Store circuit breaker state in a shared store like Redis with appropriate TTL and update patterns. Implement a distributed circuit breaker that aggregates failure information across all instances and makes routing decisions based on global provider health rather than instance-local information.

⚠️ **Pitfall: Timezone Handling in Quiet Hours** Calculating quiet hours requires careful timezone handling to avoid edge cases around daylight saving time transitions, timezone changes, and leap seconds. A common mistake is converting user quiet hours to server timezone rather than handling them in the user's local timezone, leading to notifications delivered at inappropriate times.

Why it's wrong: Incorrect timezone calculations can wake users with notifications during their configured quiet hours, violating user preferences and potentially triggering unsubscribes.

How to fix: Always calculate quiet hours in the user's configured timezone using proper timezone libraries that handle DST transitions. Store quiet hours as local time strings ("22:00", "08:00") and convert them to UTC for the current date in the user's timezone when making delivery decisions.

⚠️ **Pitfall: Webhook Signature Verification Implementation** Implementing webhook signature verification incorrectly is a common security vulnerability. Mistakes include using string comparison instead of constant-time comparison (vulnerable to timing attacks), including the wrong parts of the request in signature calculation, or not validating signature format before verification.

Why it's wrong: Incorrect signature verification can allow malicious actors to inject false delivery events, manipulating analytics and potentially triggering inappropriate system responses.

How to fix: Use cryptographically secure signature verification libraries that implement constant-time comparison. Carefully follow each provider's signature calculation specification, including exactly which headers and payload components are included. Validate signature format before attempting verification to prevent parsing errors.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Message Queue	RabbitMQ with AMQP (github.com/streadway/amqp)	Apache Kafka with Confluent client
HTTP Client	Standard library net/http with connection pooling	Fasthttp for high-performance requirements
Webhook Server	Standard library net/http with middleware	Gin or Echo for advanced routing features
Background Processing	Worker pool with goroutines and channels	Machinery or Asynq for distributed task processing
Circuit Breaker	Custom implementation with sync primitives	Hystrix-go or go-kit circuit breaker
Event Storage	PostgreSQL with JSONB for event metadata	ClickHouse for high-volume analytics workloads

Recommended File Structure

```
internal/
  flow/
    coordinator.go           ← main notification processing coordinator
    coordinator_test.go      ← end-to-end flow tests
    phases.go               ← individual processing phase implementations
  webhooks/
    processor.go            ← webhook processing coordinator
    handlers/
      sendgrid.go           ← SendGrid-specific webhook handler
      twilio.go             ← Twilio-specific webhook handler
      firebase.go           ← Firebase-specific webhook handler
    validator.go            ← webhook signature validation
    events.go               ← event parsing and normalization
  queue/
    publisher.go            ← message queue publishing
    consumer.go             ← message queue consumption
    messages.go             ← message type definitions
  routing/
    engine.go               ← routing decision logic
    fallback.go             ← fallback strategy implementation
    circuitbreaker.go       ← circuit breaker implementation
```

Infrastructure Starter Code

Complete message queue publisher implementation:

```go
package queue

import (
    "context"
    "encoding/json"
    "fmt"
    "time"

    "github.com/streadway/amqp"
)

// Publisher handles message queue publishing with connection management
type Publisher struct {
    conn     *amqp.Connection
    channel  *amqp.Channel
    exchange string
    routingKey string
}

// NewPublisher creates a new message queue publisher with connection pooling
func NewPublisher(url, exchange string) (*Publisher, error) {
    conn, err := amqp.Dial(url)
    if err != nil {
        return nil, fmt.Errorf("failed to connect to message queue: %w", err)
    }

    channel, err := conn.Channel()
    if err != nil {
        conn.Close()
        return nil, fmt.Errorf("failed to create channel: %w", err)
    }

    // Declare exchange as durable topic exchange
    err = channel.ExchangeDeclare(
        exchange, // name
        "topic",  // type
        true,     // durable
        false,    // auto-deleted
        false,    // internal
        false,    // no-wait
```

```
        nil,      // arguments
    )
    if err != nil {
        channel.Close()
        conn.Close()
        return nil, fmt.Errorf("failed to declare exchange: %w", err)
    }

    return &Publisher{
        conn:       conn,
        channel:    channel,
        exchange:   exchange,
        routingKey: "notification.process",
    }, nil
}

// PublishNotification queues a notification for background processing
func (p *Publisher) PublishNotification(ctx context.Context, msg *NotificationMessage) error {
    body, err := json.Marshal(msg)
    if err != nil {
        return fmt.Errorf("failed to marshal notification: %w", err)
    }

    return p.channel.Publish(
        p.exchange,   // exchange
        p.routingKey, // routing key
        false,        // mandatory
        false,        // immediate
        amqp.Publishing{
            ContentType:  "application/json",
            DeliveryMode: amqp.Persistent, // make message persistent
            Timestamp:    time.Now(),
            MessageId:    msg.ID,
            Body:         body,
            Priority:     uint8(msg.Priority), // map Priority enum to AMQP priority
        },
    )
}
```

```
// Close releases publisher resources
func (p *Publisher) Close() error {
    if p.channel != nil {
        p.channel.Close()
    }
    if p.conn != nil {
        p.conn.Close()
    }
    return nil
}
```

Complete webhook signature validator:

```go
package webhooks

import (
    "crypto/hmac"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "strconv"
    "strings"
    "time"
)

// WebhookValidator handles provider-specific webhook authentication
type WebhookValidator struct {
    secrets map[string]string // provider name -> webhook secret
}

// NewWebhookValidator creates a validator with provider secrets
func NewWebhookValidator(secrets map[string]string) *WebhookValidator {
    return &WebhookValidator{
        secrets: secrets,
    }
}

// ValidateSendGridSignature verifies SendGrid webhook signature with timestamp validation
func (v *WebhookValidator) ValidateSendGridSignature(payload []byte, signature, timestamp string) bool {
    secret, exists := v.secrets["sendgrid"]
    if !exists {
        return false
    }

    // Parse timestamp and verify recency (prevent replay attacks)
    ts, err := strconv.ParseInt(timestamp, 10, 64)
    if err != nil {
        return false
    }

    if time.Since(time.Unix(ts, 0)) > 10*time.Minute {
        return false // reject old webhooks
```

```
    }

    // Calculate expected signature: HMAC-SHA256(timestamp + payload)
    h := hmac.New(sha256.New, []byte(secret))
    h.Write([]byte(timestamp))
    h.Write(payload)
    expected := hex.EncodeToString(h.Sum(nil))

    // Use constant-time comparison to prevent timing attacks
    return hmac.Equal([]byte(signature), []byte(expected))
}

// ValidateTwilioSignature verifies Twilio webhook signature including URL
func (v *WebhookValidator) ValidateTwilioSignature(url string, params map[string]string, signature string) bool {
    secret, exists := v.secrets["twilio"]
    if !exists {
        return false
    }

    // Build string to sign: URL + sorted parameters
    var parts []string
    parts = append(parts, url)

    // Add parameters in sorted order
    for key, value := range params {
        parts = append(parts, key+value)
    }

    data := strings.Join(parts, "")

    // Calculate HMAC-SHA1 signature
    h := hmac.New(sha256.New, []byte(secret))
    h.Write([]byte(data))
    expected := hex.EncodeToString(h.Sum(nil))

    return hmac.Equal([]byte(signature), []byte(expected))
}
```

Core Logic Skeleton

Notification processing coordinator with detailed TODO steps:

```go
package flow

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

// NotificationCoordinator orchestrates the complete notification processing flow
type NotificationCoordinator struct {
    templateEngine   *template.Engine
    routingEngine    *routing.Engine
    channelManager   *channels.Manager
    preferenceManager *preferences.Manager
    deliveryTracker  *tracking.DeliveryTracker
}

// ProcessNotification coordinates the complete notification workflow from queue to delivery
func (c *NotificationCoordinator) ProcessNotification(ctx context.Context, msg *NotificationMessage) error {
    // TODO 1: Validate the notification message structure and required fields
    // Check that RecipientID, TemplateID are non-empty, Variables is not nil
    // Return validation error immediately if required fields missing

    // TODO 2: Load user profile including preferences, timezone, and contact info
    // Query user table by RecipientID, include preferences and quiet hours
    // Return user not found error if recipient doesn't exist

    // TODO 3: Check if user can receive this notification category/channel
    // Call preferenceManager.CanReceive() with notification details
    // If action is ActionReject, update delivery record and return early
    // If action is ActionDefer, reschedule notification and return early

    // TODO 4: Load and render template for user's locale and all required channels
    // Call templateEngine.RenderTemplate() for user's detected locale
    // Handle fallback locales if exact locale not available
    // Return template error if rendering fails completely

    // TODO 5: Determine routing plan including primary and fallback channels
```

```
    // Call routingEngine.RouteNotification() based on user preferences and channel health
    // Verify at least one healthy channel is available in routing plan

    // TODO 6: Attempt delivery through primary channel with timeout
    // Call channel.Send() with formatted message and delivery timeout
    // Handle immediate failures vs temporary failures differently

    // TODO 7: On primary failure, attempt fallback channels in order
    // Iterate through fallback channels in routing plan
    // For each channel: check health, format message, attempt delivery
    // Stop on first success or when all channels exhausted

    // TODO 8: Update delivery tracking with final status and provider metadata
    // Call deliveryTracker.UpdateDeliveryStatus() with delivery result
    // Include timing information, provider message IDs, and error details
    // Ensure status update succeeds even if delivery failed

    return nil
}
```

Webhook processing coordinator:

```go
package webhooks

import (
    "encoding/json"
    "net/http"
)

// WebhookProcessor handles incoming webhook deliveries from all providers
type WebhookProcessor struct {
    providers map[string]ProviderHandler
    tracker   *delivery.DeliveryTracker
    validator *WebhookValidator
}

// ProcessWebhook handles incoming webhook deliveries with signature validation
func (w *WebhookProcessor) ProcessWebhook(rw http.ResponseWriter, r *http.Request) {
    // TODO 1: Identify provider from URL path or headers
    // Extract provider name from request path (/webhooks/sendgrid, /webhooks/twilio)
    // Look up provider handler in providers map
    // Return 404 if provider not supported

    // TODO 2: Read and validate request payload size and content type
    // Read request body with size limits to prevent memory exhaustion
    // Validate Content-Type header matches provider expectations
    // Return 400 for oversized or malformed requests

    // TODO 3: Validate webhook signature using provider-specific method
    // Call provider.ValidateSignature() with payload and headers
    // Return 401 immediately if signature validation fails
    // Log security violation for monitoring

    // TODO 4: Parse webhook payload into standardized event structures
    // Call provider.ParseEvents() to convert provider format to internal events
    // Handle multiple events in single webhook (SendGrid sends arrays)
    // Return 400 for unparseable payloads

    // TODO 5: Process each event through delivery tracking system
    // For each parsed event, call tracker.ProcessEvent()
```

```
    // Update delivery records, analytics, and suppression lists
    // Continue processing other events if one event fails


    // TODO 6: Return success response immediately to prevent provider retries
    // Return 200 OK response regardless of individual event processing results
    // Include any required response headers (some providers expect specific format)
    // Log processing summary for monitoring


    rw.WriteHeader(http.StatusOK)
}
```

Milestone Checkpoints

Milestone 1 Checkpoint - Basic Message Flow:

- Run `go test ./internal/flow/...` - all notification processing tests pass
- Start notification service with `go run cmd/server/main.go`
- Submit test notification via `curl -X POST localhost:8080/notifications -d '{"recipientId":"test-user","templateId":"welcome","variables":{"name":"Test User"}}'`
- Verify notification appears in message queue with `rabbitmqctl list_queues`
- Check that notification processes through queue and appears in delivery tracking table

Milestone 2 Checkpoint - Template Integration:

- Run template rendering test with `go test ./internal/template/... -run TestEndToEnd`
- Submit notification with template variables and verify proper substitution in delivery record
- Test localization by submitting notifications for users with different locale settings
- Verify template fallback works by requesting unavailable locale and checking fallback chain

Milestone 3 Checkpoint - Preference Handling:

- Test quiet hours by submitting notification during user quiet hours and verifying deferral
- Test category opt-out by disabling category and verifying notification rejection
- Test unsubscribe by generating token and processing unsubscribe request
- Verify preference inheritance by testing global, category, and channel preference combinations

Milestone 4 Checkpoint - Webhook Processing:

- Set up webhook endpoint with `ngrok http 8080` for testing with real providers
- Send test webhook from SendGrid/Twilio dashboard and verify signature validation
- Check delivery record updates by sending notification and processing delivery webhook
- Test bounce handling by sending to invalid email and processing bounce webhook

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Notifications stuck in queue	Worker process crashed or deadlocked	Check queue depth with `rabbitmqctl list_queues`, examine worker process logs	Restart workers, check for poison messages in dead letter queue
Webhook signature validation failing	Incorrect secret or signature calculation	Log raw signature and calculated signature for comparison	Verify webhook secret matches provider dashboard, check signature algorithm
Circuit breaker stuck open	Failure threshold too low or provider still failing	Check circuit breaker state in Redis, test provider API directly	Adjust failure threshold or manually reset circuit breaker state
Template rendering errors	Missing variables or locale issues	Check template content and variable substitution logs	Verify all required variables provided, test template rendering with sample data
Delivery tracking inconsistent	Webhook processing failures or race conditions	Check webhook processing logs and delivery record timestamps	Implement webhook retry logic, add database constraints to prevent duplicate updates

Error Handling and Edge Cases

> **Milestone(s):** All milestones — robust error handling is critical for channel abstraction (Milestone 1), template processing (Milestone 2), user preferences (Milestone 3), and delivery tracking (Milestone 4)

Think of error handling in a notification system like operating an emergency response center during a natural disaster. When multiple systems fail simultaneously — cell towers go down, power grids fail, internet connections drop — you need predefined protocols for switching to backup communication channels, managing resource constraints, and ensuring critical messages still reach their destinations. Your notification service faces similar challenges: email providers may have outages, SMS gateways may hit rate limits, push notification tokens may expire, and malformed templates may cause rendering failures. Just as emergency responders have cascading fallback plans and resource management protocols, your notification system needs sophisticated failure detection, recovery strategies, and graceful degradation mechanisms.

The complexity of error handling in a multi-channel notification system stems from the heterogeneous nature of the underlying providers. Each channel — email, SMS, push notifications — has distinct failure modes, rate limiting behavior, and recovery characteristics. SendGrid might return HTTP 429 for rate limiting while Twilio returns HTTP 429 with different semantics and retry guidance. Firebase Cloud Messaging might invalidate device tokens silently, while Apple Push Notification service returns explicit error codes. Your error handling strategy must account for these provider-specific behaviors while maintaining a consistent experience for the application layer.

Beyond provider-specific failures, the notification system must handle cascading failures that can occur when multiple components interact. A database connection failure might prevent user preference lookups, causing notifications to be deferred. A template rendering failure might occur only for specific locales, affecting some users but not others. A webhook processing failure might cause delivery status updates to be lost, leading to incorrect analytics. These failure scenarios require careful analysis and robust recovery mechanisms to maintain system reliability and data consistency.

Failure Mode Analysis

Understanding failure modes in a notification system requires examining each component and its potential failure scenarios. The key insight is that failures are not just binary success/failure states — they exist on a spectrum from temporary hiccups to permanent degradation, each requiring different handling strategies.

Provider Outages represent complete unavailability of external notification services. Unlike simple network timeouts, outages can last from minutes to hours and affect entire regions or customer segments. The challenge is distinguishing between temporary connectivity issues and genuine provider outages that require fallback to alternative channels.

Failure Type	Detection Method	Duration	Impact	Recovery Strategy
Complete Provider Outage	HTTP 5xx responses, connection timeouts	Minutes to hours	All channels of that type affected	Circuit breaker activation, fallback to alternative providers
Regional Provider Degradation	Increased latency, partial failures	Hours to days	Geographic subset of users	Geographic routing to healthy regions
Provider Maintenance Window	HTTP 503 responses, provider notifications	Planned 1-8 hours	Predictable downtime	Pre-emptive channel switching
Provider API Deprecation	HTTP 410 responses, sunset headers	Months (warning period)	Future functionality loss	Migration to new API endpoints

The detection of provider outages requires more sophistication than simple HTTP status code checking. A 500 status code might indicate a temporary server hiccup, while a pattern of 500s across multiple requests suggests a broader outage. Your system needs to implement sliding window analysis to distinguish between isolated failures and systematic problems.

Decision: Multi-Signal Outage Detection

- **Context**: Single HTTP status codes are insufficient to distinguish temporary failures from genuine outages
- **Options Considered**: Simple status code checking, sliding window failure rate analysis, provider status page integration
- **Decision**: Hybrid approach combining failure rate analysis with external status monitoring
- **Rationale**: Failure rate analysis provides real-time detection while status page integration gives authoritative outage confirmation
- **Consequences**: Enables faster failover than waiting for manual confirmation but requires maintaining multiple detection mechanisms

Rate Limiting occurs when notification volume exceeds provider-imposed quotas. Unlike outages, rate limiting is often predictable based on your sending patterns, but the specific limits and retry guidance vary significantly between providers. Some providers return precise retry-after headers, while others provide only generic "try again later" responses.

Provider	Rate Limit Type	Limit Scope	Retry Guidance	Burst Handling
SendGrid	Requests per second	Per API key	Retry-After header	Short burst allowance
Twilio SMS	Messages per day	Per account/phone number	Rate limit headers	No burst capability
Firebase FCM	Messages per minute	Per project/topic	Exponential backoff recommended	Burst up to 1000 messages
Apple APNs	Connections per hour	Per certificate	Connection throttling	Stream multiplexing

The complexity of rate limiting extends beyond simple quota management. Some providers implement dynamic rate limiting based on sender reputation, meaning your limits can change based on bounce rates, spam complaints, or sending patterns. Additionally, rate limits often have multiple dimensions — you might have separate limits for total messages per day, messages per recipient, and API requests per second.

Invalid Tokens and Credentials represent authentication and authorization failures that can occur at multiple levels. Device tokens for push notifications naturally expire or become invalid when users uninstall apps. API credentials might be rotated for security reasons. Email addresses might be deactivated or forwarded. Each type of credential failure requires different handling strategies.

Credential Type	Failure Scenarios	Detection Method	Persistence	Recovery Action
Push Device Token	App uninstalled, token refresh	Provider error codes (InvalidToken)	Immediate suppression	Request fresh token from client
API Key/Secret	Rotation, account suspension	HTTP 401/403 responses	Immediate failure	Alert operations team
Email Address	Account deactivation, forwarding	SMTP bounce codes	Permanent/temporary classification	Suppress or retry based on bounce type
Phone Number	Carrier porting, deactivation	SMS delivery failures	Provider-specific classification	Mark as unreachable

The challenge with credential failures is determining whether they represent temporary issues or permanent changes. A push notification token might be temporarily invalid due to app updates but become valid again after the user restarts the application. An email address might be temporarily full but accept messages after the user clears their inbox.

Malformed Templates occur when template content violates channel-specific constraints or contains invalid variable substitutions. These failures are particularly challenging because they might affect only specific combinations of templates, user data, and target channels. A template might render correctly for most users but fail when a user has null values in required fields.

Template Error Type	Detection Phase	Scope	Root Cause	Prevention Strategy
Invalid Variable Reference	Rendering time	Single notification	Typo in variable name	Template validation on save
Missing Required Fields	Rendering time	User segment	Incomplete user data	Schema validation and default values
Content Length Violation	Post-rendering	Channel-specific	SMS exceeding 160 characters	Character count validation
HTML Security Violation	Rendering time	Email channel	Unsafe HTML content	Content Security Policy enforcement
Encoding Issues	Rendering time	Locale-specific	Unicode handling errors	Character encoding normalization

⚠️ **Pitfall: Silent Template Failures** Many developers assume template rendering either succeeds completely or fails completely. In reality, templates can partially render — succeeding for some variables while failing for others. This leads to notifications being sent with placeholder text like `{{undefined_variable}}` visible to users. Always implement strict variable validation that fails the entire rendering process if any required variable is missing or invalid.

Recovery and Retry Strategies

Recovery strategies for notification systems must balance several competing concerns: delivering notifications promptly, avoiding overwhelming failing systems, maintaining cost efficiency, and preserving user experience. The key insight is that different types of failures require fundamentally different recovery approaches — you cannot apply a one-size-fits-all retry strategy.

Exponential Backoff with Jitter forms the foundation of retry strategies, but its implementation requires careful consideration of failure types and provider characteristics. The basic exponential backoff increases delay between retries exponentially, while jitter adds randomization to prevent thundering herd problems when multiple instances retry simultaneously.

The mathematical formula for exponential backoff with jitter is:

```
delay = min(base_delay * 2^attempt, max_delay) + random(0, jitter_range)
```

However, the parameters must be tuned based on failure type and provider guidance:

Failure Type	Base Delay	Max Delay	Max Attempts	Jitter Range	Rationale
Rate Limiting	Provider Retry-After or 60 seconds	15 minutes	5	±25%	Respect provider guidance, limit total wait time
Temporary Provider Error	1 second	5 minutes	8	±50%	Quick recovery for transient issues
Network Connectivity	500ms	30 seconds	12	±75%	Network issues often resolve quickly
Invalid Credentials	No retry	N/A	1	N/A	Authentication issues require manual intervention
Template Rendering	No retry	N/A	1	N/A	Code bugs don't resolve with time

Decision: Failure-Type-Specific Retry Policies

- **Context**: Different failure types have fundamentally different recovery characteristics and optimal retry patterns
- **Options Considered**: Universal exponential backoff, provider-specific policies, failure-type-specific policies
- **Decision**: Failure-type-specific retry policies with provider customization overlays
- **Rationale**: Maximizes delivery success rate while minimizing resource waste and provider relationship impact
- **Consequences**: Requires more complex retry logic but provides significantly better recovery characteristics

Dead Letter Queues serve as the safety net for notifications that exhaust all retry attempts. Rather than simply discarding failed notifications, dead letter queues preserve them for manual analysis and potential reprocessing. The design of your dead letter queue strategy affects both operational overhead and compliance requirements.

Dead letter queue processing involves several stages:

1. **Immediate Quarantine**: Messages that fail all retry attempts are moved to the dead letter queue with full context about the failure progression
2. **Root Cause Analysis**: Operations teams analyze patterns in dead letter messages to identify systematic issues
3. **Selective Reprocessing**: Once underlying issues are resolved, messages can be requeued for delivery
4. **Permanent Suppression**: Messages that cannot be delivered due to permanent recipient issues are marked as permanently failed

The data structure for dead letter queue entries must capture sufficient context for effective debugging:

Field	Type	Purpose	Example
OriginalMessage	NotificationMessage	Complete original notification	User signup welcome email
FailureHistory	[]FailureAttempt	All retry attempts with errors	Attempt 1: Rate limited, Attempt 2: Invalid token
QuarantineReason	string	Final failure classification	PERMANENT_BOUNCE
QuarantineTime	time.Time	When message entered dead letter queue	2024-01-15T10:30:00Z
ReprocessingEligible	bool	Whether message can be retried	false (permanent bounce)
OperationalNotes	string	Manual annotations from debugging	User account was deleted

⚠️ **Pitfall: Dead Letter Queue Accumulation** Without proper dead letter queue management, these queues can grow indefinitely, consuming storage and making analysis increasingly difficult. Implement automated policies to purge old dead letter messages and generate regular reports on dead letter queue patterns to identify systematic issues early.

Circuit Breaker Pattern provides automatic failover when providers become systematically unreliable. Unlike simple retry logic, circuit breakers track success/failure patterns over time and can proactively prevent requests to failing services, reducing latency and resource consumption.

The circuit breaker state machine operates through three states:

State	Behavior	Transition Trigger	Request Handling
StateClosed	Normal operation, all requests allowed	Failure rate exceeds threshold	Forward all requests to provider
StateOpen	Failing state, all requests rejected immediately	Timeout period expires	Return failure immediately, attempt fallback
StateHalfOpen	Testing recovery, limited requests allowed	Success rate indicates recovery or additional failures	Allow single test request per interval

The implementation of circuit breaker thresholds requires careful tuning based on provider characteristics and business requirements:

```
Circuit Breaker Configuration Examples:

Email Provider (High Volume, Low Latency Requirements):
- Failure Threshold: 50% failure rate over 100 requests
- Open Timeout: 60 seconds
- Half-Open Test Interval: 10 seconds
- Rationale: Email can tolerate brief delays, prefer stability

SMS Provider (Lower Volume, Cost Sensitive):
- Failure Threshold: 25% failure rate over 20 requests
- Open Timeout: 300 seconds
- Half-Open Test Interval: 30 seconds
- Rationale: SMS costs money, avoid unnecessary charges on failing provider

Push Notifications (High Volume, Real-time Requirements):
- Failure Threshold: 75% failure rate over 50 requests
- Open Timeout: 30 seconds
- Half-Open Test Interval: 5 seconds
- Rationale: Push notifications often recoverable, minimize delivery delays
```

Manual Intervention Triggers recognize that some failure scenarios require human judgment and cannot be resolved through automated retry logic. The system must detect these scenarios and escalate appropriately while continuing to handle automated recoverable failures.

Manual intervention scenarios include:

Scenario	Detection Method	Escalation Timeline	Required Action
Provider API Deprecated	HTTP 410 responses, sunset headers	30 days before sunset	Update integration to new API version
Account Suspension	HTTP 403 with account-related error codes	Immediate	Contact provider support for account review
Compliance Violation	Bounce codes indicating spam/abuse	Immediate	Review sending practices and content
Infrastructure Misconfiguration	Systematic authentication failures	15 minutes	Review credentials and network configuration
Template Security Issue	Multiple XSS/injection detection events	Immediate	Review and sanitize template content

The escalation system must provide sufficient context for effective manual intervention while avoiding alert fatigue from transient issues:

Alert Context Requirements:

1. **Failure Scope**: How many notifications/users are affected
2. **Business Impact**: Which types of notifications are failing (transactional vs. marketing)
3. **Timeline**: How long the issue has persisted and trend analysis
4. **Recovery Options**: What automated recovery has been attempted
5. **Provider Context**: Any known provider issues or maintenance windows

> **Decision: Tiered Manual Intervention**
>
> - **Context**: Different failure scenarios require different levels of urgency and expertise for resolution
> - **Options Considered**: All-hands immediate escalation, tiered escalation based on impact, delayed escalation with automated recovery attempts
> - **Decision**: Tiered escalation with business impact assessment and automated recovery parallel track
> - **Rationale**: Balances rapid response for critical issues with efficient resource utilization for manageable problems
> - **Consequences**: Requires sophisticated impact assessment but reduces operational overhead and alert fatigue

The integration of these recovery strategies creates a comprehensive failure handling system that can maintain service reliability even when individual components or providers experience issues. The key principle is graceful degradation — when primary channels fail, the system automatically shifts to backup channels while preserving as much functionality as possible.

Consider a concrete example of cascading failure and recovery:

1. **Initial State**: Primary email provider (SendGrid) operating normally, SMS provider (Twilio) as backup
2. **Failure Event**: SendGrid experiences regional outage affecting 40% of email delivery attempts
3. **Detection**: Circuit breaker detects 60% failure rate over 5-minute window, transitions to StateOpen
4. **Immediate Recovery**: New email notifications automatically route to backup email provider (Amazon SES)
5. **Retry Processing**: Failed notifications enter exponential backoff retry queue with 15-minute max delay
6. **Escalation**: After 30 minutes of outage, operations team receives alert with context and recovery options
7. **Recovery Detection**: Circuit breaker transitions to StateHalfOpen, tests single request every 30 seconds
8. **Full Recovery**: After 3 consecutive successful tests, circuit breaker returns to StateClosed, normal routing resumes

This layered approach ensures that notification delivery continues even during complex failure scenarios, while providing operations teams with the information and tools needed for effective incident response.

Implementation Guidance

This error handling implementation provides the robust failure detection, retry mechanisms, and recovery strategies needed to maintain notification service reliability across diverse provider ecosystems and failure scenarios.

Technology Recommendations:

Component	Simple Option	Advanced Option
Circuit Breaker	Manual state tracking with counters	Netflix Hystrix-inspired library (github.com/sony/gobreaker)
Retry Logic	Basic exponential backoff with time.Sleep	Queue-based retry with RabbitMQ delayed messages
Dead Letter Queue	Database table with JSON error storage	Dedicated message queue (RabbitMQ DLX, AWS SQS DLQ)
Error Classification	String-based error types	Structured error codes with hierarchy
Metrics and Alerting	Simple log-based monitoring	Prometheus metrics with Grafana dashboards
Provider Health Monitoring	HTTP status code tracking	Integration with provider status pages via APIs

Recommended File/Module Structure:

```
internal/
  errors/
    types.go                    ← error classification and types
    circuit_breaker.go          ← circuit breaker implementation
    retry.go                    ← retry policy and execution
    dead_letter.go              ← dead letter queue management
    recovery.go                 ← recovery strategy coordination
  monitoring/
    health_checker.go           ← provider health monitoring
    metrics.go                  ← error metrics collection
    alerting.go                 ← manual intervention triggers
  channels/
    email/
      errors.go                 ← email-specific error handling
    sms/
      errors.go                 ← SMS-specific error handling
    push/
      errors.go                 ← push notification error handling
```

Infrastructure Starter Code:

```go
// File: internal/errors/types.go

package errors

import (
    "fmt"
    "time"
)

// ErrorCategory classifies errors for appropriate handling strategy
type ErrorCategory string

const (
    ErrorCategoryTransient   ErrorCategory = "transient"    // Retry with backoff
    ErrorCategoryRate        ErrorCategory = "rate_limit"   // Respect rate limits
    ErrorCategoryAuth        ErrorCategory = "auth"         // Manual intervention
    ErrorCategoryPermanent   ErrorCategory = "permanent"    // Dead letter queue
    ErrorCategoryTemplate    ErrorCategory = "template"     // Code fix required
)

// NotificationError provides structured error information for recovery decisions
type NotificationError struct {
    Category        ErrorCategory          `json:"category"`
    Provider        string                 `json:"provider"`
    Channel         string                 `json:"channel"`
    ErrorCode       string                 `json:"error_code"`
    Message         string                 `json:"message"`
    RetryAfter      *time.Duration         `json:"retry_after,omitempty"`
    Metadata        map[string]interface{} `json:"metadata,omitempty"`
    Timestamp       time.Time              `json:"timestamp"`
    RecoveryHint    string                 `json:"recovery_hint,omitempty"`
}

func (e *NotificationError) Error() string {
    return fmt.Sprintf("[%s:%s] %s: %s", e.Provider, e.Channel, e.ErrorCode, e.Message)
}

// IsRetryable returns whether this error should trigger retry logic
func (e *NotificationError) IsRetryable() bool {
    return e.Category == ErrorCategoryTransient || e.Category == ErrorCategoryRate
}
```

```go

// RequiresManualIntervention returns whether this error needs human attention

func (e *NotificationError) RequiresManualIntervention() bool {

    return e.Category == ErrorCategoryAuth

}


// FailureAttempt tracks a single retry attempt for dead letter queue context

type FailureAttempt struct {

    AttemptNumber int                  `json:"attempt_number"`

    Timestamp     time.Time            `json:"timestamp"`

    Error         *NotificationError   `json:"error"`

    RetryAfter    time.Duration        `json:"retry_after"`

}


// DeadLetterEntry represents a notification that exhausted all retry attempts

type DeadLetterEntry struct {

    ID                   string                `json:"id"`

    OriginalMessage      *NotificationMessage  `json:"original_message"`

    FailureHistory       []FailureAttempt      `json:"failure_history"`

    QuarantineReason     ErrorCategory         `json:"quarantine_reason"`

    QuarantineTime       time.Time             `json:"quarantine_time"`

    ReprocessingEligible bool                  `json:"reprocessing_eligible"`

    OperationalNotes     string                `json:"operational_notes,omitempty"`

    ExpiresAt            time.Time             `json:"expires_at"`

}
```

```go
// File: internal/errors/circuit_breaker.go

package errors

import (
    "sync"
    "time"
)

// CircuitBreakerConfig defines thresholds and timeouts for circuit breaker behavior
type CircuitBreakerConfig struct {
    FailureThreshold    int           `json:"failure_threshold"`     // Number of failures to trigger open
    SuccessThreshold    int           `json:"success_threshold"`     // Number of successes to close from half-open
    OpenTimeout         time.Duration `json:"open_timeout"`          // How long to stay open before testing
    HalfOpenTestInterval time.Duration `json:"half_open_test_interval"` // How often to test in half-open
}

// CircuitBreakerStats provides visibility into circuit breaker state
type CircuitBreakerStats struct {
    State              CircuitState `json:"state"`
    ConsecutiveFailures int         `json:"consecutive_failures"`
    ConsecutiveSuccesses int        `json:"consecutive_successes"`
    LastFailureTime    time.Time    `json:"last_failure_time"`
    LastSuccessTime    time.Time    `json:"last_success_time"`
    NextRetryTime      time.Time    `json:"next_retry_time"`
    TotalRequests      int64        `json:"total_requests"`
    TotalFailures      int64        `json:"total_failures"`
}

// CircuitBreakerImpl manages failure detection and automatic failover
type CircuitBreakerImpl struct {
    config   CircuitBreakerConfig
    stats    CircuitBreakerStats
    mutex    sync.RWMutex
    onStateChange func(from, to CircuitState)
}

// NewCircuitBreaker creates a circuit breaker with specified thresholds
func NewCircuitBreaker(config CircuitBreakerConfig) *CircuitBreakerImpl {
    return &CircuitBreakerImpl{
```

```
        config: config,
        stats: CircuitBreakerStats{
            State: StateClosed,
        },
    }
}

// CanExecute checks if circuit breaker allows request execution
func (cb *CircuitBreakerImpl) CanExecute() bool {
    cb.mutex.RLock()
    defer cb.mutex.RUnlock()

    switch cb.stats.State {
    case StateClosed:
        return true
    case StateOpen:
        return time.Now().After(cb.stats.NextRetryTime)
    case StateHalfOpen:
        return time.Now().After(cb.stats.NextRetryTime)
    default:
        return false
    }
}

// RecordSuccess notifies circuit breaker of successful operation
func (cb *CircuitBreakerImpl) RecordSuccess() {
    // TODO 1: Acquire write lock for state modification
    // TODO 2: Increment total requests and consecutive successes
    // TODO 3: Reset consecutive failures to 0
    // TODO 4: Update last success time to now
    // TODO 5: If state is StateHalfOpen and consecutive successes >= success threshold, transition to StateClosed
    // TODO 6: If transitioning states, call onStateChange callback if configured
    // TODO 7: If state is StateHalfOpen, set next retry time for next test interval
}

// RecordFailure notifies circuit breaker of failed operation
func (cb *CircuitBreakerImpl) RecordFailure(err error) {
    // TODO 1: Acquire write lock for state modification
```

```
    // TODO 2: Increment total requests, total failures, and consecutive failures
    // TODO 3: Reset consecutive successes to 0
    // TODO 4: Update last failure time to now
    // TODO 5: If state is StateClosed and consecutive failures >= failure threshold, transition to StateOpen
    // TODO 6: If transitioning to StateOpen, calculate next retry time (now + open timeout)
    // TODO 7: If state is StateHalfOpen, transition back to StateOpen and reset timeout
    // TODO 8: If transitioning states, call onStateChange callback if configured
}

// GetStats returns current circuit breaker statistics for monitoring
func (cb *CircuitBreakerImpl) GetStats() CircuitBreakerStats {
    cb.mutex.RLock()
    defer cb.mutex.RUnlock()
    return cb.stats
}
```

Core Logic Skeleton Code:

```go
// File: internal/errors/retry.go

package errors

import (
    "context"
    "math"
    "math/rand"
    "time"
)

// RetryPolicy defines retry behavior for different error categories
type RetryPolicy struct {
    MaxAttempts    int           `json:"max_attempts"`
    BaseDelay      time.Duration `json:"base_delay"`
    MaxDelay       time.Duration `json:"max_delay"`
    JitterPercent  float64       `json:"jitter_percent"`
    BackoffFactor  float64       `json:"backoff_factor"`
}

// RetryExecutor handles exponential backoff retry logic with jitter
type RetryExecutor struct {
    policies map[ErrorCategory]RetryPolicy
    random   *rand.Rand
}

// NewRetryExecutor creates retry executor with category-specific policies
func NewRetryExecutor() *RetryExecutor {
    return &RetryExecutor{
        policies: map[ErrorCategory]RetryPolicy{
            ErrorCategoryTransient: {
                MaxAttempts:   8,
                BaseDelay:     1 * time.Second,
                MaxDelay:      5 * time.Minute,
                JitterPercent: 0.5,
                BackoffFactor: 2.0,
            },
            ErrorCategoryRate: {
                MaxAttempts:   5,
                BaseDelay:     60 * time.Second,
```

```
                MaxDelay:      15 * time.Minute,
                JitterPercent: 0.25,
                BackoffFactor: 2.0,
            },
        },
        random: rand.New(rand.NewSource(time.Now().UnixNano())),
    }
}

// ExecuteWithRetry executes operation with appropriate retry policy based on error category
func (re *RetryExecutor) ExecuteWithRetry(ctx context.Context, category ErrorCategory, operation func() error) error {
    // TODO 1: Look up retry policy for error category, return error if no policy exists
    // TODO 2: Initialize attempt counter starting at 1
    // TODO 3: Create loop for retry attempts up to policy.MaxAttempts
    // TODO 4: Execute operation and check for success (nil error)
    // TODO 5: If operation succeeded, return nil
    // TODO 6: If error is not retryable (check error.IsRetryable), return error immediately
    // TODO 7: If this was the last attempt, return the error
    // TODO 8: Calculate delay using exponential backoff with jitter formula
    // TODO 9: Check if context was cancelled, return context error if so
    // TODO 10: Sleep for calculated delay, checking context cancellation periodically
    // TODO 11: Increment attempt counter and continue loop
    // Hint: Use math.Min to cap delay at policy.MaxDelay
    // Hint: Add jitter as random percentage of calculated delay
}

// CalculateDelay computes exponential backoff delay with jitter for specific attempt
func (re *RetryExecutor) CalculateDelay(policy RetryPolicy, attempt int) time.Duration {
    // TODO 1: Calculate base exponential delay: baseDelay * (backoffFactor ^ (attempt - 1))
    // TODO 2: Apply maximum delay cap using math.Min
    // TODO 3: Calculate jitter range: delay * jitterPercent
    // TODO 4: Generate random jitter between -jitterRange and +jitterRange
    // TODO 5: Add jitter to delay and ensure result is non-negative
    // TODO 6: Return final delay duration
    // Hint: Use math.Pow for exponentiation
    // Hint: Use re.random.Float64() for jitter randomization
}
```

```go
// File: internal/errors/dead_letter.go

package errors

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

// DeadLetterQueue manages notifications that exhausted all retry attempts
type DeadLetterQueue struct {
    store    DeadLetterStore
    config   DeadLetterConfig
    metrics  DeadLetterMetrics
}

// DeadLetterStore defines persistence interface for dead letter entries
type DeadLetterStore interface {
    Store(ctx context.Context, entry *DeadLetterEntry) error
    Get(ctx context.Context, id string) (*DeadLetterEntry, error)
    List(ctx context.Context, filters DeadLetterFilters) ([]*DeadLetterEntry, error)
    Update(ctx context.Context, entry *DeadLetterEntry) error
    Delete(ctx context.Context, id string) error
}

// DeadLetterConfig controls dead letter queue behavior and retention
type DeadLetterConfig struct {
    RetentionPeriod     time.Duration `json:"retention_period"`
    MaxEntriesPerUser   int           `json:"max_entries_per_user"`
    ReprocessingEnabled bool          `json:"reprocessing_enabled"`
    AlertThreshold      int           `json:"alert_threshold"`
}

// QuarantineNotification moves notification to dead letter queue with full failure context
func (dlq *DeadLetterQueue) QuarantineNotification(ctx context.Context, msg *NotificationMessage, failures []FailureAttempt)
error {
    // TODO 1: Generate unique ID for dead letter entry
    // TODO 2: Determine quarantine reason from final failure in failures slice
    // TODO 3: Assess whether notification is eligible for reprocessing based on failure type
```

```
    // TODO 4: Calculate expiration time based on retention period
    // TODO 5: Create DeadLetterEntry with all context information
    // TODO 6: Store entry using dead letter store interface
    // TODO 7: Update metrics counters for quarantine reason
    // TODO 8: Check if alert threshold exceeded and trigger notification if so
    // Hint: Use last failure's error category to determine quarantine reason
    // Hint: Only auth and permanent errors are not eligible for reprocessing
}

// ReprocessNotifications attempts to resend notifications from dead letter queue
func (dlq *DeadLetterQueue) ReprocessNotifications(ctx context.Context, filters DeadLetterFilters) (int, error) {
    // TODO 1: Check if reprocessing is enabled in configuration
    // TODO 2: Retrieve dead letter entries matching filters from store
    // TODO 3: Filter entries to only those eligible for reprocessing
    // TODO 4: For each eligible entry, validate that original issue has been resolved
    // TODO 5: Re-queue original notification message for normal processing
    // TODO 6: Update dead letter entry with reprocessing timestamp and notes
    // TODO 7: Count successful reprocessing attempts
    // TODO 8: Return total count of reprocessed notifications
    // Hint: Use notification service's ProcessNotification method for re-queuing
}
```

Language-Specific Hints:

- Use `context.Context` for all retry operations to enable cancellation during long backoff periods
- Implement `sync.RWMutex` in circuit breakers to allow concurrent read access to state while protecting writes
- Use `time.NewTimer` instead of `time.Sleep` in retry logic to enable context cancellation
- Store retry policies in configuration files (JSON/YAML) rather than hard-coding values
- Use `math/rand` with per-executor seed for consistent but distributed jitter patterns
- Implement error wrapping with `fmt.Errorf("retry attempt %d: %w", attempt, err)` for debugging
- Use structured logging with fields like `attempt_number`, `delay_ms`, and `error_category`
- Consider using `golang.org/x/time/rate` package for advanced rate limiting beyond simple delays

Milestone Checkpoint:

After implementing error handling and recovery strategies:

1. **Circuit Breaker Validation**:
   - Run `go test ./internal/errors/circuit_breaker_test.go -v`
   - Expected: All states transition correctly under failure/success scenarios
   - Manual test: Configure low failure threshold, send requests to failing mock provider, verify circuit opens
2. **Retry Logic Testing**:
   - Run `go test ./internal/errors/retry_test.go -v`
   - Expected: Exponential backoff with jitter produces appropriate delays
   - Manual test: Send notification with transient error, verify retry attempts with increasing delays

3. **Dead Letter Queue Processing**:
    - Run `go test ./internal/errors/dead_letter_test.go -v`
    - Expected: Notifications quarantined after max retries, reprocessing works for eligible entries
    - Manual test: Send notification that will permanently fail, verify entry appears in dead letter queue
4. **Integration Testing**:
    - Run `go test ./internal/integration/error_handling_test.go -v`
    - Expected: End-to-end error scenarios handled correctly with proper fallback
    - Manual test: Disable primary email provider, send email notification, verify fallback to secondary provider

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Notifications never retry after failure	Retry policy not configured or error not marked retryable	Check error category classification and retry policy lookup	Ensure error categorization matches retry policy keys
Circuit breaker stuck in open state	Success threshold too high or test requests not executing	Check circuit breaker stats and test interval configuration	Lower success threshold or increase test frequency
Dead letter queue growing rapidly	Systematic issue causing all notifications to fail permanently	Analyze dead letter entries for common failure patterns	Fix root cause (credentials, template, configuration)
Retries happening too frequently	Jitter percentage too low causing thundering herd	Monitor retry timing distribution across multiple instances	Increase jitter percentage to 50% or higher
Manual alerts firing constantly	Alert thresholds too sensitive or systematic provider issues	Review alert frequency and provider status pages	Adjust thresholds or implement provider-specific suppression

Testing Strategy

> **Milestone(s):** All milestones — comprehensive testing is essential for validating channel abstraction (Milestone 1), template processing (Milestone 2), user preferences (Milestone 3), and delivery tracking (Milestone 4)

Think of testing a notification system like quality control in a postal service. Just as the postal service must test that mail sorters correctly route packages, that address labels print clearly, that customers' delivery preferences are respected, and that tracking updates accurately reflect package status, our notification system needs multi-layered testing to ensure each component works correctly in isolation and integrates seamlessly with others. The challenge is that unlike physical mail, our system handles multiple delivery channels with vastly different failure modes, processes templates with complex localization logic, manages user preferences with regulatory implications, and tracks deliveries through asynchronous webhook callbacks.

A robust testing strategy for a multi-channel notification service requires careful attention to both the diverse failure modes of external providers and the complex interactions between system components. Each milestone introduces distinct testing challenges: channel abstraction must validate fallback behavior and circuit breaker logic, template processing needs security validation and localization coverage, user preferences require compliance verification and preference resolution testing, while delivery tracking demands webhook simulation and analytics accuracy verification.

Testing Levels

The notification system requires three distinct testing levels, each targeting different aspects of system correctness and reliability. Unit tests validate individual component logic in isolation, integration tests verify interactions with external providers and internal services, while end-to-end tests confirm complete notification workflows function correctly under real-world conditions.

Unit Testing for Template Engine and Core Logic

Unit testing forms the foundation of our testing strategy, focusing on business logic that can be tested deterministically without external dependencies. The template engine represents the most critical unit testing target because it handles user-provided data, applies security transformations, and must work correctly across multiple locales and channels.

Template rendering unit tests must verify variable substitution accuracy across different data types and locales. Consider testing a template containing `"Hello {{name}}, you have {{count}} {{#pluralize count}}message{{/pluralize}} waiting"`. Unit tests should validate that when

`name="José"` and `count=1`, the output renders as `"Hello José, you have 1 message waiting"`, while `count=3` produces `"Hello José, you have 3 messages waiting"`. Each test case should verify specific variable types including strings, numbers, dates, arrays, and nested objects.

Security validation unit tests must confirm that template content undergoes proper escaping based on the target channel. For HTML email templates, verify that user variables containing `<script>alert('xss')</script>` become `&lt;script&gt;alert('xss')&lt;/script&gt;` in the final output. For plain text channels, ensure that special characters don't break the formatting. SMS templates require length validation to prevent unexpected message segmentation charges.

Test Category	Test Description	Input Example	Expected Output	Failure Mode
Variable Substitution	Basic string replacement	`name="John", template="Hello {{name}}"`	`"Hello John"`	Missing variables render as empty strings
Numeric Formatting	Currency and date formatting	`amount=1234.56, locale="en-US"`	`"$1,234.56"`	Wrong locale produces incorrect format
XSS Protection	HTML escaping for email	`name="<script>alert()</script>"`	`"&lt;script&gt;alert()&lt;/script&gt;"`	Unescaped content enables XSS attacks
SMS Length Validation	Character count enforcement	`message length > 160 chars`	`ValidationError`	Long messages incur unexpected charges
Pluralization Rules	ICU MessageFormat rules	`count=0, count=1, count=5`	Correct plural forms	Wrong plurals look unprofessional
Missing Variables	Undefined placeholder handling	`template="Hi {{missing}}"`	`"Hi "` or validation error	Template rendering fails completely

Localization unit tests must verify fallback behavior when exact locale matches aren't available. Test that a user with locale `"pt-BR"` (Brazilian Portuguese) receives the `"pt"` template when `"pt-BR"` doesn't exist, and falls back to `"en"` (English) when neither Portuguese variant is available. Verify that date formatting, number formatting, and currency display adapt correctly to each locale.

Template versioning unit tests should confirm that version activation and rollback operations maintain consistency. Test that activating template version 3 makes it the default for new notifications while preserving existing notifications that reference specific versions. Verify that rolling back from a problematic template version correctly reverts to the previous stable version.

Channel Integration Testing with Provider Mocking

Integration testing focuses on the interaction between our channel abstraction layer and external notification providers. Since external providers can be unreliable, expensive to test against, or have strict rate limits, effective integration testing relies on provider mocking that accurately simulates real provider behavior including success responses, failure modes, and edge cases.

Each channel integration test must validate the complete request-response cycle through our `Channel` interface. For email channels, verify that the `Format` method correctly transforms our internal `FormattedMessage` structure into the JSON payload expected by SendGrid's API, including proper handling of recipient lists, subject lines, HTML/plain text content, and attachment encoding. Test that the `Send` method correctly interprets SendGrid's response format and maps their message IDs to our `DeliveryResult` structure.

Provider mock implementations should simulate realistic failure scenarios that channels encounter in production. Email provider mocks should simulate authentication failures (invalid API keys), rate limiting responses (429 status codes with retry-after headers), recipient validation failures (invalid email addresses), and content rejection (spam detection, blocked domains). SMS provider mocks should simulate invalid phone number formats, carrier routing failures, and international delivery restrictions.

Channel Type	Success Scenario	Failure Scenarios	Provider Response	Our Handling
Email (SendGrid)	Valid recipient, content	Auth failure, rate limit, spam detection	`{"message_id": "abc123"}` vs `{"errors": [...]}`	Map to `DeliveryResult` vs `NotificationError`
SMS (Twilio)	Valid phone number	Invalid number, carrier block	`{"sid": "SM123", "status": "queued"}`	Extract provider message ID
Push (FCM)	Valid device token	Expired token, invalid topic	`{"name": "projects/.../messages/123"}`	Handle token refresh requirements
In-App	User session active	User offline, device disconnected	WebSocket ACK vs timeout	Queue for retry when user reconnects

Circuit breaker integration tests must verify that repeated failures correctly transition the circuit breaker through its state machine. Test that after the configured failure threshold is reached, the circuit breaker enters the open state and rejects new requests immediately without calling the provider. Verify that after the recovery timeout, the circuit breaker transitions to half-open state and allows a limited number of test requests to determine provider recovery.

Rate limiting integration tests should confirm that channels respect provider-imposed limits and apply appropriate backoff strategies. Test that when a provider returns a 429 status code with a `Retry-After` header, the channel correctly schedules the retry for the specified time rather than immediately retrying. Verify that rate limit handling doesn't interfere with circuit breaker logic.

End-to-End Delivery Testing

End-to-end testing validates complete notification workflows from initial request submission through final delivery confirmation. These tests exercise the entire system including message queuing, template rendering, channel selection, provider integration, webhook processing, and analytics updates. End-to-end tests should use real external providers in a sandbox or test mode to verify actual integration behavior.

Complete workflow testing must simulate realistic notification scenarios that exercise multiple system components simultaneously. Test a marketing email campaign that requires template rendering with user-specific variables, preference checking to ensure the user hasn't opted out, quiet hours validation to defer delivery if needed, channel selection with fallback logic, and delivery tracking through provider webhooks.

Consider an end-to-end test for a password reset notification: submit a notification with template ID `"password-reset"` and recipient user ID, verify that the template engine renders the reset link with a valid token, confirm that the routing engine selects email as the primary channel (password resets shouldn't use SMS for security reasons), validate that the email channel formats the message correctly for the provider, simulate successful delivery and webhook callback, and verify that the delivery status is recorded and analytics are updated.

Test Scenario	Components Exercised	Success Criteria	Failure Points to Test
Password Reset Email	Template engine, routing, email channel, webhooks	Email delivered, status tracked, analytics updated	Template rendering failure, SMTP auth failure, webhook signature invalid
SMS Marketing with Opt-out	Preference checking, template engine, SMS channel	User opted out → notification suppressed	User preference lookup failure, SMS provider rate limit
Push Notification Fallback	Routing engine, push channel, circuit breaker, email fallback	Primary push fails → email backup succeeds	Both channels fail, circuit breaker stuck open
Localized Welcome Series	Template engine localization, channel selection, delivery tracking	Correct language template, successful delivery	Missing translation, wrong locale detection

Webhook processing end-to-end tests must verify that delivery status updates flow correctly from provider callbacks through our webhook handlers to delivery tracking and analytics systems. Set up test scenarios where providers send delivery, bounce, and engagement webhooks, then verify that the webhook processor correctly validates signatures, parses events, updates delivery records, and triggers analytics aggregation.

Performance testing within end-to-end tests should validate that the system handles realistic message volumes without degrading. Test notification bursts that might occur during password reset attacks, marketing campaign launches, or system outage recovery. Verify that message queues handle backpressure correctly and that rate limiting prevents overwhelming downstream providers.

⚠️ Pitfall: Testing Only Happy Path Scenarios

Many notification systems fail in production because testing focused exclusively on successful delivery scenarios. Real-world notification systems encounter provider outages, webhook delivery failures, malformed templates, invalid recipient data, and complex preference combinations that never

appear in happy path tests. Comprehensive testing must deliberately exercise failure scenarios and edge cases that might seem unlikely but inevitably occur at scale.

⚠️ Pitfall: Ignoring Timing and Concurrency Issues

Notification systems are inherently asynchronous with message queues, webhook callbacks, and external provider integrations creating complex timing dependencies. Tests that don't account for eventual consistency, race conditions, and timeout scenarios will miss critical bugs that only appear under production load or unusual timing conditions.

Milestone Validation Checkpoints

Each milestone requires specific validation checkpoints that confirm the implementation meets acceptance criteria and handles the failure scenarios most relevant to that milestone's functionality. These checkpoints combine automated testing with manual verification procedures that simulate real-world usage patterns.

Milestone 1: Channel Abstraction & Routing Validation

The channel abstraction milestone focuses on proving that the pluggable channel interface works correctly across different providers and that routing logic selects appropriate channels based on notification type and user preferences. Validation must confirm that fallback mechanisms activate correctly when primary channels fail and that circuit breakers prevent cascading failures.

Channel interface compliance testing verifies that each channel implementation correctly fulfills the `Channel` interface contract. For each channel (email, SMS, push), validate that the `Validate` method correctly identifies invalid recipients (malformed email addresses, invalid phone numbers, expired device tokens), that the `Format` method transforms message content into the provider-specific format, and that the `Send` method returns appropriate `DeliveryResult` structures with provider message IDs.

Create test scenarios that exercise routing decision logic under different conditions. Test that urgent notifications (password resets, security alerts) bypass quiet hours restrictions, that marketing messages respect user channel preferences, and that transactional notifications (order confirmations, receipt emails) use reliable channels even when users have opted out of marketing communications.

Validation Test	Setup	Expected Behavior	Failure Indicators
Basic Channel Send	Submit notification to each channel type	Message sent, delivery result returned	Channel returns error, no provider message ID
Provider Authentication	Configure invalid API keys	Graceful error, retry not attempted	System crashes, infinite retry loop
Routing with Preferences	User has disabled SMS, notification allows multiple channels	Routes to email instead of SMS	Still attempts SMS, routing decision ignores preferences
Circuit Breaker Activation	Simulate repeated provider failures	Circuit opens, requests rejected immediately	Circuit never opens, requests continue failing slowly
Fallback Channel Success	Primary channel fails, fallback configured	Attempts secondary channel, succeeds	Doesn't try fallback, gives up after primary failure

Circuit breaker validation requires systematic testing of state transitions under failure and recovery scenarios. Configure circuit breakers with low thresholds (5 failures to open, 30-second recovery timeout) for faster testing. Send notifications that will fail due to provider configuration, verify that the circuit breaker opens after the threshold is reached, confirm that subsequent requests are rejected immediately, wait for the recovery timeout, and verify that the circuit breaker allows test requests in half-open state.

Manual testing procedures should verify that channel selection logic handles edge cases correctly. Test notifications sent to users with complex preference configurations (opted out of email but allows transactional, SMS enabled only during business hours, push notifications only for urgent messages). Verify that quiet hours logic correctly interprets timezone settings and that rate limiting doesn't interfere with high-priority notifications.

⚠️ Pitfall: Not Testing Provider-Specific Edge Cases

Each notification provider has unique limitations and edge cases that don't appear in generic channel interface testing. SendGrid handles mailing list unsubscribes differently from individual unsubscribes, Twilio charges differently for international SMS, and FCM token refresh behavior varies across Android versions. Provider-specific testing must exercise these differences to avoid production surprises.

Milestone 2: Template System Validation

Template system validation must confirm that template rendering produces correct output across different locales and channels while preventing security vulnerabilities through proper content escaping. Testing must verify that template versioning works correctly and that localization fallback logic

selects appropriate language variants when exact matches aren't available.

Template rendering accuracy testing involves creating test templates with various complexity levels and variable types. Start with simple string substitution templates, progress to templates with conditional logic and loops, and finish with templates that combine multiple data sources. For each template, test rendering with different variable sets including edge cases like null values, empty arrays, and deeply nested objects.

Security validation testing must verify that user-provided variables undergo appropriate escaping based on the target channel. Create test cases with potentially dangerous content including HTML tags, JavaScript code, SQL injection attempts, and shell commands. Verify that HTML email templates escape these inputs appropriately while plain text channels handle them safely without breaking formatting.

Template Test Category	Test Input	Expected HTML Output	Expected Plain Text Output	Security Risk if Failed
HTML Escaping	`name="<script>alert('xss')</script>"`	`<script>alert('xss')</script>`	`<script>alert('xss')</script>`	XSS attacks in email clients
URL Parameter Injection	`link="http://evil.com"`	Escaped or validated URL	Plain URL (validated)	Click hijacking, phishing
Template Injection	`name="{{#admin}}SECRET{{/admin}}"`	Literal text output	Literal text output	Information disclosure
Newline Injection	`message="Line 1\nLine 2"`	`Line 1 Line 2`	`Line 1\nLine 2`	Email header injection

Localization testing must verify that template selection follows the correct fallback chain when exact locale matches aren't available. Create templates in multiple languages (`en-US` , `en-GB` , `en` , `es-MX` , `es` , `fr-CA` , `fr`) and test users with various locale preferences. Verify that a user with locale `"es-MX"` receives the `"es-MX"` template when available, falls back to `"es"` when the specific variant doesn't exist, and ultimately falls back to `"en"` when no Spanish templates are available.

Template versioning validation involves testing version creation, activation, rollback, and concurrent usage scenarios. Create multiple versions of a template, activate version 2, submit notifications that should use the active version, then rollback to version 1 and verify that new notifications use the rolled-back version while in-flight notifications continue using their originally assigned version.

Manual validation procedures should test template preview functionality with real data from your user database (appropriately anonymized). Verify that preview rendering matches actual notification output and that template validation catches common errors like missing variables, malformed mustache syntax, and content that exceeds SMS length limits.

Milestone 3: User Preferences & Unsubscribe Validation

User preferences validation must confirm that preference checking correctly evaluates the complex hierarchy of global, category, and channel preferences while respecting regulatory requirements for unsubscribe handling. Testing must verify that quiet hours logic correctly handles timezone differences and that unsubscribe tokens are cryptographically secure and properly validated.

Preference resolution testing requires creating users with complex preference combinations and verifying that delivery decisions match expected outcomes. Create test users with scenarios like: globally opted out (should receive only transactional notifications), opted out of marketing category but allows newsletter category, SMS disabled during quiet hours but allows urgent notifications, and email enabled but specific senders blocked.

Test the preference resolution logic with notifications of different types and priorities. A marketing email to a user who has disabled marketing should be rejected, while a password reset email should be allowed even if the user has opted out of all marketing. Urgent security notifications should bypass quiet hours restrictions, while newsletter emails should be deferred until the user's active hours.

User Preference State	Notification Type	Priority	Expected Decision	Reason
Global opt-out: false, Email enabled	Marketing newsletter	Normal	Allow	User preferences permit marketing email
Global opt-out: false, Quiet hours active	Marketing newsletter	Normal	Defer	Respect quiet hours for non-urgent messages
Global opt-out: false, Quiet hours active	Password reset	Urgent	Allow	Security notifications bypass quiet hours
Marketing opt-out: true	Marketing newsletter	Normal	Reject	User opted out of marketing category
Marketing opt-out: true	Order confirmation	Normal	Allow	Transactional notifications bypass marketing opt-out

Unsubscribe token validation must verify that tokens are cryptographically secure and cannot be forged or enumerated by attackers. Generate unsubscribe tokens for different users and scopes, verify that each token can only be used once, confirm that tampered tokens are rejected, and test that expired tokens cannot be used. Verify that unsubscribe processing correctly updates user preferences and triggers confirmation emails when appropriate.

Timezone handling validation requires testing quiet hours logic with users in different timezones and edge cases like daylight saving time transitions. Create test users in various timezones (UTC, EST, PST, JST, IST) with quiet hours configured for local times. Send notifications at different UTC times and verify that quiet hours decisions correctly account for the user's local timezone.

Regulatory compliance testing must verify that the unsubscribe system meets CAN-SPAM, GDPR, and other applicable regulations. Test that marketing emails include proper unsubscribe links, that unsubscribe requests are processed immediately (not "within 10 days"), and that users cannot unsubscribe from transactional notifications like password resets or account security alerts.

⚠️ Pitfall: Not Testing Timezone Edge Cases

Quiet hours timezone handling becomes complex during daylight saving time transitions and for users in timezones with unusual offset rules. Test quiet hours logic during DST changes to ensure that a user's 10 PM quiet hours start time doesn't suddenly shift by an hour, potentially missing time-sensitive notifications or violating user expectations.

Milestone 4: Delivery Tracking & Analytics Validation

Delivery tracking validation must confirm that webhook processing correctly updates notification status through the complete lifecycle from sent through delivered, opened, clicked, bounced, or failed. Testing must verify that analytics aggregation produces accurate metrics and that bounce handling correctly identifies hard bounces that require recipient suppression.

Webhook processing testing requires simulating the various webhook formats sent by different providers and verifying that each webhook correctly updates delivery status and triggers appropriate follow-up actions. Set up test webhook endpoints that can receive callbacks from providers (or simulate them with mock HTTP requests) and verify that webhook signature validation correctly accepts authentic webhooks while rejecting forged requests.

Create test scenarios for each major webhook event type: delivery confirmations that update status from sent to delivered, bounce notifications that identify undeliverable addresses, open tracking events that record engagement timestamps, and click tracking events that capture link interaction data. For each event type, verify that the webhook processor correctly parses the provider-specific format and maps it to our internal event structures.

Webhook Event Type	Provider Format	Internal Event	Analytics Impact	Follow-up Actions
Delivery Confirmation	`{"event": "delivered", "sg_message_id": "abc123"}`	`DeliveryEvent` with status delivered	Increment delivered count	None
Hard Bounce	`{"event": "bounce", "reason": "invalid_email"}`	`BounceEvent` with type hard	Increment bounce count	Suppress recipient address
Email Open	`{"event": "open", "user_agent": "Mail.app"}`	`OpenEvent` with timestamp	Increment open count	None (tracking only)
Click Tracking	`{"event": "click", "url": "https://example.com/..."}`	`ClickEvent` with original URL	Increment click count	Update link popularity metrics

Analytics aggregation testing must verify that real-time metrics calculation produces accurate results and that batch reconciliation processes handle data consistency correctly. Send notifications with known delivery outcomes and verify that delivery rates, bounce rates, and engagement metrics match expected values. Test edge cases like notifications that bounce immediately (should count as sent but not delivered) and bulk campaigns where metrics must be calculated across thousands of individual notifications.

Bounce handling validation requires testing both hard and soft bounce scenarios with appropriate follow-up actions. Hard bounces (permanent delivery failures) should immediately suppress the recipient address from future notifications, while soft bounces (temporary failures) should be retried with exponential backoff. Test that bounce classification correctly distinguishes between these categories and that recipient suppression doesn't interfere with other notification channels (email bounce shouldn't suppress SMS delivery).

Manual validation procedures should include testing the analytics dashboard with real data from your staging environment. Verify that metrics visualizations correctly represent the underlying data, that filtering by time ranges produces accurate results, and that drill-down functionality allows investigating delivery issues for specific campaigns or recipients.

Performance validation must confirm that webhook processing can handle high-volume callback bursts that occur during large campaign deliveries. Simulate webhook bursts with hundreds of concurrent callbacks and verify that processing doesn't fall behind, that duplicate event detection prevents double-counting, and that database performance remains acceptable under load.

⚠️ Pitfall: Not Testing Webhook Signature Validation Thoroughly

Webhook signature validation is critical for security but easy to implement incorrectly. Test signature validation with tampered payloads, expired timestamps, wrong signing algorithms, and edge cases like empty request bodies. Verify that signature validation handles different payload encoding schemes correctly and that timing attacks aren't possible through signature comparison logic.

Implementation Guidance

The testing strategy implementation requires careful attention to test isolation, provider mocking strategies, and validation automation that can reliably detect regressions across multiple milestones. The challenge lies in creating tests that are both comprehensive enough to catch real bugs and maintainable enough to evolve with the system.

Technology Recommendations for Testing

Testing Level	Simple Option	Advanced Option	Trade-offs
Unit Testing	`testing` package + table-driven tests	`testify/suite` + `testify/mock`	Simple approach lacks mocking, advanced provides better organization
Integration Testing	`httptest` for HTTP mocking	`testcontainers-go` for real services	HTTP mocking is faster, containers more realistic
End-to-End Testing	Shell scripts + curl	`ginkgo` + `gomega` BDD framework	Scripts are simple, BDD framework provides better structure
Provider Mocking	Hand-written HTTP handlers	`wiremock` or `mockserver`	Hand-written mocks are simpler, dedicated tools more powerful
Database Testing	`sqlite` in-memory	`testcontainers` PostgreSQL	SQLite is faster, real PostgreSQL catches schema issues

Recommended Testing Structure

Organize testing code to mirror the production structure while keeping test utilities reusable across multiple test suites:

```
project-root/
  internal/
    channels/
      email/
        email.go
        email_test.go           ← unit tests for email channel
        email_integration_test.go ← integration tests with provider mocking
      sms/
        sms.go
        sms_test.go
    template/
      engine.go
      engine_test.go            ← unit tests for template rendering
      localization_test.go      ← localization-specific tests
    preferences/
      manager.go
      manager_test.go
  test/
    fixtures/                   ← test data and templates
      templates/
        welcome-email.html
        password-reset.txt
      users/
        test-users.json
    helpers/                    ← shared testing utilities
      mock_providers.go         ← provider mock implementations
      test_database.go          ← database setup/teardown
      webhook_simulator.go      ← webhook testing tools
    e2e/                       ← end-to-end test suites
      notification_flow_test.go
      webhook_processing_test.go
  cmd/
    test-server/               ← test server for manual validation
      main.go
```

Core Testing Infrastructure

Provider Mock Infrastructure - Complete mock implementations that simulate real provider behavior including success, failure, and edge cases:

```go
// MockEmailProvider simulates SendGrid API behavior for testing
type MockEmailProvider struct {
    // Responses maps request patterns to mock responses
    Responses map[string]MockResponse
    // ReceivedRequests stores all requests for verification
    ReceivedRequests []MockRequest
    // SimulateFailure controls failure simulation
    SimulateFailure bool
    mu sync.RWMutex
}

type MockResponse struct {
    StatusCode int
    Body       string
    Headers    map[string]string
    Delay      time.Duration
}

// NewMockEmailProvider creates a configurable email provider mock
func NewMockEmailProvider() *MockEmailProvider {
    return &MockEmailProvider{
        Responses: map[string]MockResponse{
            // TODO 1: Add default success response for valid email sends
            // TODO 2: Add authentication failure response for invalid API keys
            // TODO 3: Add rate limiting response with retry-after header
            // TODO 4: Add recipient validation failure responses
        },
        ReceivedRequests: make([]MockRequest, 0),
        SimulateFailure: false,
    }
}

// ServeHTTP implements http.Handler for use with httptest.Server
func (m *MockEmailProvider) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    m.mu.Lock()
    defer m.mu.Unlock()

    // TODO 1: Record the incoming request for later verification
```

```
    // TODO 2: Check if we should simulate failure based on SimulateFailure flag
    // TODO 3: Match request path/body against configured responses
    // TODO 4: Write appropriate response with configured delay if specified
    // TODO 5: Update internal counters for circuit breaker testing
}

// GetReceivedRequests returns all requests received by this mock
func (m *MockEmailProvider) GetReceivedRequests() []MockRequest {
    // TODO: Return copy of received requests for test verification
}

// SimulateOutage configures the mock to return 5xx errors
func (m *MockEmailProvider) SimulateOutage(duration time.Duration) {
    // TODO: Set failure mode and schedule recovery after duration
}
```

Test Database Helper - Database setup and cleanup utilities that ensure test isolation:

```go
// TestDB provides database setup/teardown for integration tests
type TestDB struct {
    DB       *sql.DB
    Config   *DatabaseConfig
    cleanup  func()
}

// NewTestDB creates an isolated test database instance
func NewTestDB() (*TestDB, error) {
    // TODO 1: Create unique database name (use timestamp + random)
    // TODO 2: Connect to PostgreSQL and create test database
    // TODO 3: Run database migrations to create schema
    // TODO 4: Set up cleanup function to drop database
    // TODO 5: Return TestDB instance with cleanup function
}

// LoadFixtures loads test data from JSON files
func (db *TestDB) LoadFixtures(fixturePath string) error {
    // TODO 1: Read fixture files from test/fixtures directory
    // TODO 2: Parse JSON into appropriate structs (User, Template, etc.)
    // TODO 3: Insert fixture data into test database
    // TODO 4: Handle foreign key dependencies correctly
}

// Cleanup removes the test database and closes connections
func (db *TestDB) Cleanup() error {
    // TODO: Execute cleanup function to drop database
}
```

Webhook Testing Utilities - Tools for simulating provider webhook deliveries:

```go
// WebhookSimulator sends realistic webhook payloads for testing
type WebhookSimulator struct {
    client      *http.Client
    signatures  map[string]string  // provider -> signing secret
    baseURL     string
}

// NewWebhookSimulator creates webhook testing utility
func NewWebhookSimulator(baseURL string, secrets map[string]string) *WebhookSimulator {
    return &WebhookSimulator{
        client:     &http.Client{Timeout: 5 * time.Second},
        signatures: secrets,
        baseURL:    baseURL,
    }
}

// SendDeliveryWebhook simulates successful delivery callback
func (w *WebhookSimulator) SendDeliveryWebhook(providerMessageID, notificationID string) error {
    // TODO 1: Create provider-specific webhook payload
    // TODO 2: Sign payload with HMAC-SHA256 using provider secret
    // TODO 3: POST to webhook endpoint with proper headers
    // TODO 4: Verify response indicates successful processing
}

// SendBounceWebhook simulates bounce notification
func (w *WebhookSimulator) SendBounceWebhook(email, bounceReason string, isHard bool) error {
    // TODO 1: Create bounce webhook payload with proper classification
    // TODO 2: Include diagnostic information and SMTP response codes
    // TODO 3: Send webhook with correct signature
    // TODO 4: Verify bounce processing occurred correctly
}
```

Testing Workflow Skeletons

Channel Integration Test Pattern - Template for testing each channel implementation:

```go
// TestEmailChannelIntegration validates email channel behavior with provider mocking
func TestEmailChannelIntegration(t *testing.T) {
    // TODO 1: Set up mock email provider with httptest.Server
    // TODO 2: Configure email channel to use mock provider URL
    // TODO 3: Create test user and formatted message
    // TODO 4: Test successful send - verify request format and response handling
    // TODO 5: Test authentication failure - verify error classification
    // TODO 6: Test rate limiting - verify retry-after handling
    // TODO 7: Test recipient validation - verify error responses
    // TODO 8: Verify all expected requests were made to mock provider
}

// TestSMSChannelCircuitBreaker validates circuit breaker behavior
func TestSMSChannelCircuitBreaker(t *testing.T) {
    // TODO 1: Configure circuit breaker with low thresholds for faster testing
    // TODO 2: Set up SMS provider mock to simulate failures
    // TODO 3: Send notifications until circuit breaker opens
    // TODO 4: Verify subsequent requests are rejected immediately
    // TODO 5: Wait for recovery timeout and verify half-open behavior
    // TODO 6: Send successful request to verify circuit closes
}
```

Template Engine Test Pattern - Comprehensive template testing approach:

```go
// TestTemplateRenderingAccuracy validates variable substitution correctness
func TestTemplateRenderingAccuracy(t *testing.T) {
    tests := []struct {
        name      string
        template  string
        variables map[string]interface{}
        locale    string
        expected  string
        wantErr   bool
    }{
        // TODO 1: Add test cases for each variable type (string, number, date, array)
        // TODO 2: Add test cases for conditional logic and loops
        // TODO 3: Add test cases for nested object access
        // TODO 4: Add test cases for missing variables
        // TODO 5: Add test cases for locale-specific formatting
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // TODO 1: Set up template engine with test configuration
            // TODO 2: Render template with provided variables
            // TODO 3: Compare output with expected result
            // TODO 4: Verify error handling matches wantErr
        })
    }
}

// TestTemplateSecurityValidation ensures XSS protection works correctly
func TestTemplateSecurityValidation(t *testing.T) {
    // TODO 1: Create templates with user-provided variables containing HTML/JS
    // TODO 2: Render templates for HTML email channel
    // TODO 3: Verify dangerous content is properly escaped
    // TODO 4: Test plain text channel handling of special characters
    // TODO 5: Verify SMS channel prevents length-based attacks
}
```

End-to-End Test Pattern - Complete workflow validation:

```go
// TestCompleteNotificationWorkflow validates end-to-end delivery
func TestCompleteNotificationWorkflow(t *testing.T) {
    // TODO 1: Set up test database with user and template fixtures
    // TODO 2: Start notification service with test configuration
    // TODO 3: Set up provider mocks for all channels
    // TODO 4: Set up webhook simulator for delivery callbacks
    // TODO 5: Submit notification request via API
    // TODO 6: Verify template rendering occurred correctly
    // TODO 7: Verify channel selection logic chose appropriate channel
    // TODO 8: Verify provider received correctly formatted request
    // TODO 9: Simulate webhook callback indicating delivery
    // TODO 10: Verify delivery status was updated in database
    // TODO 11: Verify analytics were updated correctly
}
```

Milestone Validation Scripts

Milestone 1 Validation Commands:

```bash
# Run channel abstraction tests
go test ./internal/channels/... -v

# Test routing engine with various scenarios
go test ./internal/routing/... -v -run TestRoutingEngine

# Validate circuit breaker behavior
go test ./internal/channels/... -v -run TestCircuitBreaker

# Manual validation: submit test notification
curl -X POST localhost:8080/api/notifications \
  -H "Content-Type: application/json" \
  -d '{"recipient_id": "test-user", "template_id": "welcome", "priority": "normal"}'
```

Expected Milestone 1 Behavior:

- Channel tests pass with both success and failure scenarios
- Routing engine selects appropriate channels based on preferences
- Circuit breaker opens after configured failure threshold
- Manual API test returns notification ID and queues for processing
- Provider mock receives correctly formatted request within 5 seconds

Milestone 2 Validation Commands:

```bash
# Test template rendering accuracy
go test ./internal/template/... -v -run TestRenderingAccuracy

# Validate localization and fallback logic
go test ./internal/template/... -v -run TestLocalization

# Test template security features
go test ./internal/template/... -v -run TestSecurityValidation

# Manual template preview test
curl -X POST localhost:8080/api/templates/preview \
  -H "Content-Type: application/json" \
  -d '{"template_id": "welcome", "variables": {"name": "Test User"}, "locale": "en-US", "channel": "email"}'
```

Expected Milestone 2 Behavior:

- Template rendering produces correct output for all variable types
- Localization selects appropriate template variant with fallback
- Security validation prevents XSS and other content injection
- Manual preview returns properly formatted HTML and plain text versions

Common Testing Issues and Debugging

Symptom	Likely Cause	How to Diagnose	Fix
Tests pass individually but fail in suite	Test isolation issues	Check for shared state between tests	Use test-specific database instances, reset mocks
Provider integration tests are flaky	Network timeouts or race conditions	Add logging to track request timing	Increase timeouts, add retry logic to tests
Template tests fail on different machines	Locale or timezone differences	Check system locale in failing environment	Use explicit locale settings in tests
Webhook tests never receive callbacks	Webhook URL not accessible	Verify test server is listening on correct port	Use `httptest.Server` for reliable webhook testing
Circuit breaker tests are slow	High failure thresholds	Check circuit breaker configuration	Use low thresholds (5 failures) for faster test execution

The testing strategy provides comprehensive validation coverage while remaining maintainable as the system evolves through each milestone. Focus on creating reliable test infrastructure first, then build milestone-specific tests that exercise the unique challenges introduced by each component.

Debugging Guide

> **Milestone(s):** All milestones — comprehensive debugging is essential for troubleshooting channel abstraction (Milestone 1), template processing (Milestone 2), user preferences (Milestone 3), and delivery tracking (Milestone 4) issues in production environments

Think of debugging a notification system like being a detective investigating a missing person case. Just as a detective follows clues through different locations and witnesses, debugging notifications requires tracing messages through multiple components, external providers, and asynchronous processes. The key difference is that in a notification system, your "missing person" might have disappeared into a message queue, gotten lost at an external provider, or been filtered out by user preferences — and you need systematic tools and techniques to follow the trail.

The distributed nature of notification systems creates unique debugging challenges. Unlike debugging a single-threaded application where you can step through code line by line, notifications flow through message queues, external APIs, webhook callbacks, and background processing jobs. A notification

might fail at any of these stages, and the failure might not be immediately visible. This section provides systematic approaches for diagnosing and resolving issues across all components of the notification system.

Effective debugging in notification systems requires understanding both the happy path and the numerous failure modes. The complexity comes from the interaction between multiple systems: your notification service, message queues, template engines, user preference systems, external providers (email, SMS, push), and webhook processing. Each component introduces its own failure modes, and problems often manifest as cascading failures across multiple components.

Common Symptoms and Causes

The most effective way to debug notification systems is to start with observable symptoms and work backwards to root causes. This symptom-driven approach mirrors how production issues typically present themselves: users report they didn't receive notifications, dashboards show delivery rate drops, or monitoring alerts fire about queue backups.

Understanding the relationship between symptoms and causes helps narrow down the investigation scope quickly. In notification systems, similar symptoms can have dramatically different root causes, so systematic diagnosis prevents wasting time on incorrect assumptions. The following analysis covers the most frequent issues encountered across all milestones of the notification system.

Notification Never Sent

Symptom	Observable Signs	Likely Root Cause	Investigation Steps	Resolution
Notification not queued	No record in message queue or database	Request validation failed	Check API logs for 400/422 errors, validate request payload against `NotificationMessage` schema	Fix client request format, improve validation error messages
Queued but never processed	Message visible in queue but status remains `StatusQueued`	Queue consumer down or crashed	Check consumer process status, review consumer logs for startup errors	Restart consumer service, fix configuration issues
Processing failed immediately	Status changed to `StatusFailed` within seconds	Template not found or malformed	Check template store for template ID, review template validation logs	Upload correct template, fix template syntax errors
User preference blocking	Processing succeeds but no channel selected	User opted out or in quiet hours	Check `UserPreference` records, review routing engine decision logs	Verify user preferences are correct, check quiet hours timezone handling

Template Rendering Issues

Symptom	Observable Signs	Likely Root Cause	Investigation Steps	Resolution
Template variable not substituted	Rendered content contains `{{variable}}` placeholders	Variable name mismatch or missing data	Compare variable names in template vs. notification payload, check case sensitivity	Fix variable names, ensure all required variables provided
Content security violation	Template rendering returns validation error	Unsafe HTML or JavaScript in template	Review `ContentValidator` rejection logs, check for `<script>` tags or unsafe attributes	Sanitize template content, use text-only for untrusted content
Localization fallback failing	Content appears in wrong language or empty	Locale detection or fallback chain broken	Check user locale value, review `LocalizationManager` fallback logic	Fix locale format (en-US vs en_US), add missing fallback templates
Template version conflict	Old content appears despite new template	Template versioning cache stale	Check active template version in `TemplateStore`, verify cache invalidation	Clear template cache, fix cache invalidation logic

Channel Delivery Failures

Symptom	Observable Signs	Likely Root Cause	Investigation Steps	Resolution
Provider authentication error	Consistent 401/403 from provider APIs	API key expired or invalid	Test provider credentials manually via cURL, check key rotation schedule	Update API keys, implement automatic credential refresh
Rate limit exceeded	429 errors from provider, delivery delays	Traffic burst or inefficient batching	Check provider rate limits in documentation, review send patterns	Implement exponential backoff, add request queuing
Circuit breaker stuck open	All requests failing with circuit open message	Provider outage not recovered	Check `CircuitBreakerStats`, verify provider health status	Manually reset circuit breaker after confirming provider recovery
Invalid recipient format	Provider rejects requests with format errors	Phone number or email validation insufficient	Review provider error responses, check recipient format requirements	Improve validation regex, normalize phone numbers to E.164 format

Webhook Processing Problems

Symptom	Observable Signs	Likely Root Cause	Investigation Steps	Resolution
Webhooks not processed	External provider shows delivered but internal status unchanged	Signature validation failing	Check webhook secret configuration, review HMAC calculation	Update webhook secrets in provider dashboard and application config
Duplicate status updates	Same notification marked delivered multiple times	Webhook replay without idempotency	Check for duplicate webhook payloads, verify idempotency key handling	Implement webhook deduplication, store processed webhook IDs
Status updates delayed	Delivery status updated hours after actual delivery	Webhook retry backlog	Check webhook processing queue depth, review retry policies	Scale webhook processors, implement priority queues for recent events
Analytics inconsistent	Dashboard metrics don't match provider reports	Event processing pipeline failures	Compare raw webhook logs with processed events, check aggregation logic	Fix event processing bugs, implement reconciliation job

Performance and Scale Issues

Symptom	Observable Signs	Likely Root Cause	Investigation Steps	Resolution
Queue backup growing	Message queue depth increasing, processing lag	Consumer throughput insufficient	Monitor consumer CPU/memory usage, check message processing time	Scale consumer instances, optimize processing logic
Database connection exhaustion	Database timeout errors, connection pool full	Too many concurrent database operations	Check database connection pool configuration, review query performance	Increase connection pool size, add connection pooling timeouts
Template cache misses	High template rendering latency, frequent database queries	Cache eviction or invalidation too aggressive	Monitor cache hit rates, review cache size and TTL settings	Increase cache size, adjust TTL based on template update frequency
Memory leaks in processors	Consumer memory usage growing over time	Goroutine leaks or retained references	Profile memory usage, check for unclosed channels or infinite goroutines	Fix resource leaks, implement proper cleanup in error paths

> **Key Insight**: Most notification system bugs are integration issues, not logic bugs. The complexity comes from coordinating multiple external systems, each with their own failure modes, rate limits, and data formats. Systematic logging and monitoring at integration boundaries is essential for effective debugging.

⚠️ Pitfall: Debugging in Production Without Proper Tooling

A common mistake is trying to debug notification issues by manually querying databases and checking logs across multiple services. This approach is time-consuming and error-prone, especially during production incidents when quick resolution is critical.

Instead, build debugging tools into the system from the beginning. Every notification should have a correlation ID that can be traced through all components. Implement debug endpoints that show the complete lifecycle of a notification, including routing decisions, template rendering results, and delivery attempts. Create dashboards that visualize the notification pipeline in real-time, showing queue depths, processing rates, and error rates at each stage.

The investment in debugging tooling pays dividends during production incidents. When users report missing notifications, you should be able to trace the issue within minutes using correlation IDs and debug endpoints, rather than spending hours correlating logs across multiple services.

Debugging Tools and Techniques

Effective debugging of notification systems requires a multi-layered approach combining structured logging, distributed tracing, interactive debugging tools, and systematic testing techniques. Think of this as building a diagnostic laboratory for your notification system — just as medical diagnostics use different tools for different types of problems, notification debugging requires specialized tools for each component and failure mode.

The key principle is observability at every integration boundary. Since notifications flow through multiple external systems, you need visibility into what happens at each handoff point. This includes the message queue interface, template rendering pipeline, channel selection logic, external provider APIs, and webhook processing. Each boundary should emit structured logs and metrics that can be correlated using unique identifiers.

Correlation ID Strategy

Every notification should carry a unique correlation ID that flows through all system components. This ID enables end-to-end tracing from the initial API request through final delivery status updates. The correlation ID should be included in all log messages, queued messages, database records, and external API calls.

Component	Correlation ID Usage	Log Format Example	Storage Location
API Gateway	Generate correlation ID for incoming requests	`[correlation_id=abc123] Notification request received from user 456`	HTTP headers, request logs
Message Queue	Include correlation ID in message properties	`[correlation_id=abc123] Message queued for processing`	Message metadata
Template Engine	Log rendering decisions with correlation ID	`[correlation_id=abc123] Template 'welcome' rendered for locale 'en-US'`	Processing logs
Channel Handler	Include in provider API calls	`[correlation_id=abc123] Sending email via SendGrid, message_id=789`	Provider request logs
Webhook Processor	Link provider callbacks to original notification	`[correlation_id=abc123] Delivery confirmed by provider, status=delivered`	Webhook processing logs

The correlation ID should be a UUID v4 to ensure uniqueness across distributed deployments. Include it in the `NotificationMessage` structure and ensure it propagates to all downstream components. When debugging issues, the correlation ID becomes your primary investigation tool — you can search logs across all services to reconstruct the complete notification lifecycle.

Structured Logging Configuration

Implement structured logging using JSON format to enable powerful log analysis and filtering. Each log entry should include standard fields plus component-specific context. The structured format allows automated log analysis and integration with monitoring systems.

Standard Fields	Description	Example Values
`timestamp`	ISO 8601 timestamp with timezone	`2024-01-15T10:30:45.123Z`
`level`	Log level (ERROR, WARN, INFO, DEBUG)	`ERROR`
`correlation_id`	Unique request identifier	`550e8400-e29b-41d4-a716-446655440000`
`component`	System component generating log	`template_engine`, `email_channel`, `webhook_processor`
`operation`	High-level operation being performed	`render_template`, `send_notification`, `process_bounce`
`user_id`	Recipient user identifier (when available)	`user_12345`
`notification_id`	Notification identifier	`notif_67890`
`duration_ms`	Operation duration in milliseconds	`245`
`error_code`	Structured error code (when applicable)	`TEMPLATE_NOT_FOUND`, `RATE_LIMIT_EXCEEDED`

⚠️ Pitfall: Logging Sensitive Information

Notification systems handle sensitive user data including email addresses, phone numbers, and personal information in template variables. Never log complete user data or notification content in production logs. Instead, log hashed or truncated identifiers that allow correlation without exposing sensitive information.

For debugging purposes, implement a separate debug mode that logs full content but only activates for specific correlation IDs and automatically expires after a short time window. This allows detailed debugging of specific issues without creating security risks in routine logging.

Message Queue Inspection Tools

Debugging message queue issues requires tools to inspect queue contents, message properties, and processing status without disrupting production traffic. Implement administrative endpoints that provide visibility into queue state and message processing.

Tool	Purpose	Implementation	Usage Example
Queue Depth Monitor	Track message backlog across all queues	Periodic polling of queue management API	Alert when depth exceeds threshold, identify processing bottlenecks
Message Inspector	View messages in queue without consuming	Queue management API with peek functionality	Inspect message format during development, debug serialization issues
Dead Letter Browser	Browse and reprocess failed messages	Web interface for `DeadLetterQueue`	Review failed notifications, bulk reprocessing after fixes
Consumer Health Check	Monitor consumer process status and performance	Health check endpoints with processing statistics	Verify consumers are running, identify slow consumers

The message queue inspection tools should be accessible through administrative APIs with appropriate authentication. Implement read-only access for operations teams and write access (such as reprocessing) for engineering teams. Include rate limiting to prevent inspection tools from impacting production performance.

Provider API Testing Tools

Since notification delivery depends on external providers, implement tools for testing provider connectivity and debugging API issues independently of the full notification pipeline. These tools help isolate provider-specific problems from application logic issues.

Provider	Test Tool	Verification Steps	Expected Results
SendGrid Email	`curl` script with API key	Send test email to verified address	Response 202, receive email within 30 seconds
Twilio SMS	`curl` script with account SID	Send test SMS to verified number	Response 201, receive SMS within 60 seconds
FCM Push	Node.js test script	Send test push to device token	Response 200, notification appears on device
APNs Push	`curl` with certificate auth	Send test push to device token	Response 200, notification appears on device

Create wrapper scripts that use the same authentication and configuration as your production application. This ensures that provider testing uses identical credentials and settings as the actual notification system. Include these test tools in your deployment pipeline to verify provider connectivity after configuration changes.

Interactive Debug Endpoints

Implement HTTP endpoints specifically for debugging individual notifications and system components. These endpoints should be available in development and staging environments, with authentication-protected access in production.

Endpoint	Purpose	Parameters	Response
`GET /debug/notification/{id}`	Complete notification lifecycle view	Notification ID	JSON with all processing steps, decisions, and status changes
`POST /debug/template/render`	Test template rendering	Template ID, variables, locale, channel	Rendered content and any validation warnings
`POST /debug/routing/evaluate`	Test channel routing logic	User ID, notification type, priority	Routing decision with explanations
`GET /debug/preferences/{user_id}`	View resolved user preferences	User ID	Complete preference hierarchy and final decisions
`POST /debug/webhook/simulate`	Simulate provider webhook	Provider name, event type, payload	Processing results and status updates

The debug endpoints should return detailed explanations of decisions, not just final results. For example, the routing endpoint should explain why each channel was selected or rejected, including preference checks, quiet hours evaluation, and circuit breaker status.

Production Monitoring and Alerting

Implement comprehensive monitoring that alerts on both system-level issues (queue backups, high error rates) and business-level issues (delivery rate drops, user complaints). The monitoring system should distinguish between expected fluctuations and actionable problems.

Metric Category	Key Metrics	Alert Conditions	Investigation Triggers
Throughput	Notifications per minute, queue processing rate	50% drop from baseline	Check consumer health, provider rate limits
Delivery Success	Delivery rate by channel, bounce rate	Delivery rate < 95%, bounce rate > 5%	Investigate provider issues, check recipient list quality
Template System	Rendering errors, cache hit rate	Error rate > 1%, cache hit rate < 90%	Check template syntax, review cache configuration
Provider Health	API response times, error rates	Response time > 5s, error rate > 2%	Test provider connectivity, check for service degradation

Configure alerting thresholds based on historical baselines rather than fixed values. Notification volume and delivery rates vary significantly based on time of day, day of week, and business cycles. Use statistical analysis to identify anomalies rather than simple threshold checks.

> **Design Insight**: The most effective debugging approach is to instrument the system for observability from the beginning, rather than adding debugging tools after problems occur. Every significant operation should emit structured logs and metrics that support both real-time monitoring and post-incident analysis.

Distributed Tracing Implementation

For complex notification flows that span multiple services, implement distributed tracing using tools like Jaeger or Zipkin. Each major operation should create a trace span with relevant metadata, allowing visualization of the complete notification journey.

Span Name	Parent Span	Metadata	Duration
`process_notification`	Root span	notification_id, user_id, priority	Complete processing time
`evaluate_preferences`	`process_notification`	user_id, category, final_decision	Preference evaluation time
`render_template`	`process_notification`	template_id, locale, variables_count	Template rendering time
`send_email`	`process_notification`	provider, recipient_hash, message_size	Provider API call time
`process_webhook`	Root span	provider, event_type, notification_id	Webhook processing time

Distributed tracing provides visual representation of notification processing, making it easier to identify bottlenecks and failures. The trace visualization shows parallel operations (like rendering multiple channel formats) and sequential dependencies (like preference evaluation before channel selection).

Implementation Guidance

The debugging infrastructure should be built into the notification system from the beginning, not added as an afterthought. This section provides concrete tools and code structures for implementing comprehensive debugging capabilities.

Technology Recommendations

Component	Simple Option	Advanced Option	Production Recommendation
Structured Logging	Go standard `log/slog` with JSON formatter	Logrus or Zap with custom formatters	Zap for performance, structured fields
Message Queue Inspection	RabbitMQ Management UI	Custom admin panel with queue APIs	Custom panel integrated with monitoring
Distributed Tracing	OpenTelemetry with stdout exporter	Jaeger or Zipkin with persistent storage	Jaeger for complex multi-service flows
Metrics Collection	Prometheus client library	Custom metrics aggregation	Prometheus with Grafana dashboards
Debug Interface	Simple HTTP endpoints with JSON	Full web-based debugging console	Web console with authentication

Recommended File Structure

```
internal/
  debug/
    debug.go              ← debug endpoint handlers
    correlation.go        ← correlation ID middleware
    tracing.go            ← distributed tracing setup
    metrics.go            ← metrics collection utilities

  monitoring/
    alerts.go             ← alert condition definitions
    dashboards.go         ← Grafana dashboard configs
    health.go             ← health check endpoints

  logging/
    logger.go             ← structured logger setup
    fields.go             ← standard log field constants
    context.go            ← context-aware logging utilities

  tools/
    provider_test.go      ← provider connectivity tests
    queue_inspect.go      ← queue inspection utilities
    webhook_sim.go        ← webhook simulation tools
```

Infrastructure Starter Code

Complete implementation of correlation ID middleware and structured logging setup:

```go
// internal/debug/correlation.go

package debug


import (

    "context"

    "crypto/rand"

    "fmt"

    "net/http"

)


type correlationKeyType struct{}


var correlationKey = correlationKeyType{}


// CorrelationMiddleware adds correlation ID to requests and context

func CorrelationMiddleware(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        correlationID := r.Header.Get("X-Correlation-ID")

        if correlationID == "" {

            correlationID = generateCorrelationID()

        }



        ctx := context.WithValue(r.Context(), correlationKey, correlationID)

        w.Header().Set("X-Correlation-ID", correlationID)



        next.ServeHTTP(w, r.WithContext(ctx))

    })

}


// GetCorrelationID extracts correlation ID from context

func GetCorrelationID(ctx context.Context) string {

    if id, ok := ctx.Value(correlationKey).(string); ok {

        return id

    }

    return "unknown"

}


func generateCorrelationID() string {

    bytes := make([]byte, 16)

    rand.Read(bytes)
```

```
    return fmt.Sprintf("%x-%x-%x-%x-%x",
        bytes[0:4], bytes[4:6], bytes[6:8], bytes[8:10], bytes[10:16])
}

// internal/logging/logger.go
package logging

import (
    "context"
    "log/slog"
    "os"
    "time"

    "github.com/yourorg/notification-service/internal/debug"
)

type ContextLogger struct {
    logger *slog.Logger
}

func NewContextLogger() *ContextLogger {
    opts := &slog.HandlerOptions{
        Level: slog.LevelInfo,
        AddSource: true,
    }

    handler := slog.NewJSONHandler(os.Stdout, opts)
    logger := slog.New(handler)

    return &ContextLogger{logger: logger}
}

func (cl *ContextLogger) InfoContext(ctx context.Context, msg string, args ...any) {
    cl.logger.InfoContext(ctx, msg, append(cl.contextArgs(ctx), args...)...)
}

func (cl *ContextLogger) ErrorContext(ctx context.Context, msg string, args ...any) {
    cl.logger.ErrorContext(ctx, msg, append(cl.contextArgs(ctx), args...)...)
}
```

```
func (cl *ContextLogger) contextArgs(ctx context.Context) []any {
    return []any{
        "correlation_id", debug.GetCorrelationID(ctx),
        "timestamp", time.Now().UTC(),
    }
}
```

Debug Endpoint Implementation

Complete implementation of notification debugging endpoints:

```go
// internal/debug/debug.go

package debug

import (
    "encoding/json"
    "fmt"
    "net/http"
    "time"

    "github.com/yourorg/notification-service/internal/delivery"
    "github.com/yourorg/notification-service/internal/preferences"
    "github.com/yourorg/notification-service/internal/templates"
)

type DebugServer struct {
    deliveryTracker   *delivery.DeliveryTracker
    preferenceManager *preferences.PreferenceManager
    templateEngine    *templates.TemplateEngine
}

type NotificationDebugView struct {
    NotificationID    string                 `json:"notification_id"`
    CorrelationID     string                 `json:"correlation_id"`
    Status            string                 `json:"status"`
    ProcessingSteps   []ProcessingStep       `json:"processing_steps"`
    DeliveryAttempts []DeliveryAttempt      `json:"delivery_attempts"`
    UserPreferences   map[string]interface{} `json:"user_preferences"`
    TemplateInfo      TemplateDebugInfo      `json:"template_info"`
}

type ProcessingStep struct {
    Timestamp   time.Time   `json:"timestamp"`
    Component   string      `json:"component"`
    Operation   string      `json:"operation"`
    Status      string      `json:"status"`
    Details     interface{} `json:"details"`
    Duration    string      `json:"duration"`
}
```

```
// GetNotificationDebug returns complete debugging information for a notification
func (ds *DebugServer) GetNotificationDebug(w http.ResponseWriter, r *http.Request) {
    // TODO: Extract notification ID from URL path
    // TODO: Query delivery tracker for notification status and history
    // TODO: Query preference manager for user preferences at send time
    // TODO: Query template engine for template rendering information
    // TODO: Combine all information into NotificationDebugView structure
    // TODO: Return JSON response with complete debugging information
}

// RenderTemplateDebug tests template rendering with provided parameters
func (ds *DebugServer) RenderTemplateDebug(w http.ResponseWriter, r *http.Request) {
    // TODO: Parse JSON request body with template ID, variables, locale, channel
    // TODO: Call template engine to render template with provided parameters
    // TODO: Include validation warnings and security policy results
    // TODO: Return rendered content along with metadata about rendering process
}
```

Queue Inspection Tools

Complete implementation of message queue inspection utilities:

```go
// internal/tools/queue_inspect.go

package tools

import (
    "context"
    "encoding/json"
    "fmt"
    "time"

    "github.com/streadway/amqp"
)

type QueueInspector struct {
    conn    *amqp.Connection
    channel *amqp.Channel
}

type QueueStatus struct {
    Name           string `json:"name"`
    MessageCount   int    `json:"message_count"`
    ConsumerCount  int    `json:"consumer_count"`
    MessagesPerSec float64 `json:"messages_per_sec"`
}

type MessagePreview struct {
    MessageID     string                 `json:"message_id"`
    CorrelationID string                 `json:"correlation_id"`
    Timestamp     time.Time              `json:"timestamp"`
    Size          int                    `json:"size_bytes"`
    Headers       map[string]interface{} `json:"headers"`
    PayloadPreview string                `json:"payload_preview"`
}

// GetQueueStatus returns current status of all notification queues
func (qi *QueueInspector) GetQueueStatus(ctx context.Context) ([]QueueStatus, error) {
    // TODO: Query RabbitMQ management API for queue statistics
    // TODO: Calculate message processing rates based on historical data
    // TODO: Return status for all notification-related queues
    return nil, fmt.Errorf("not implemented")
```

```
}


// PeekMessages returns preview of messages in queue without consuming them
func (qi *QueueInspector) PeekMessages(ctx context.Context, queueName string, limit int) ([]MessagePreview, error) {
    // TODO: Use RabbitMQ management API to peek at messages
    // TODO: Parse message headers and extract correlation IDs
    // TODO: Truncate payload for preview (first 200 characters)
    // TODO: Include message metadata (timestamp, size, headers)
    return nil, fmt.Errorf("not implemented")
}
```

Provider Testing Framework

Complete implementation of provider connectivity testing:

```go
// internal/tools/provider_test.go

package tools

import (
    "context"
    "fmt"
    "time"

    "github.com/yourorg/notification-service/internal/channels"
)

type ProviderTester struct {
    emailChannel channels.Channel
    smsChannel   channels.Channel
    pushChannel  channels.Channel
}

type TestResult struct {
    Provider    string        `json:"provider"`
    Channel     string        `json:"channel"`
    Success     bool          `json:"success"`
    Duration    time.Duration `json:"duration"`
    Error       string        `json:"error,omitempty"`
    Details     interface{}   `json:"details"`
}

// TestAllProviders runs connectivity tests against all configured providers
func (pt *ProviderTester) TestAllProviders(ctx context.Context) ([]TestResult, error) {
    // TODO: Test email provider with known good recipient address
    // TODO: Test SMS provider with verified phone number
    // TODO: Test push notification providers with test device tokens
    // TODO: Measure response times and validate responses
    // TODO: Return structured results for each provider
    return nil, fmt.Errorf("not implemented")
}

// TestSpecificProvider runs detailed test against single provider
func (pt *ProviderTester) TestSpecificProvider(ctx context.Context, providerName string) (*TestResult, error) {
    // TODO: Select appropriate channel based on provider name
```

```
    // TODO: Execute health check and test message delivery
    // TODO: Validate provider response format and error handling
    // TODO: Return detailed test results including timing information
    return nil, fmt.Errorf("not implemented")
}
```

Milestone Checkpoints

After implementing debugging infrastructure, verify functionality with these checkpoints:

Checkpoint 1: Correlation ID Tracing

- Submit a notification via API and capture the correlation ID from response headers
- Search logs across all services using the correlation ID
- Verify you can trace the notification from API request through queue processing to delivery
- Expected: Complete timeline of processing steps with consistent correlation ID

Checkpoint 2: Debug Endpoint Functionality

- Use debug endpoint to inspect a specific notification by ID
- Verify response includes processing steps, delivery attempts, and user preferences
- Test template rendering debug endpoint with sample variables
- Expected: Detailed debugging information accessible via HTTP endpoints

Checkpoint 3: Queue Inspection Tools

- Use queue inspector to view message counts and consumer status
- Peek at messages in notification queue to verify format
- Verify dead letter queue browser shows failed messages
- Expected: Administrative visibility into message queue state without disrupting processing

Debugging Workflow Documentation

Create runbooks for common debugging scenarios that operations teams can follow:

1. **User Reports Missing Notification**: Start with correlation ID lookup, check routing decisions, verify channel delivery attempts
2. **Delivery Rate Drop**: Check provider health, review rate limiting, inspect queue backlogs
3. **Template Rendering Errors**: Validate template syntax, check variable availability, review locale fallback
4. **Webhook Processing Issues**: Verify signature validation, check for duplicate events, review retry policies

Each runbook should include specific commands to run, log patterns to search for, and escalation criteria for when manual investigation is needed.

Future Extensions

> **Milestone(s):** Beyond core milestones — extends the notification service with advanced features like A/B testing, campaign management, ML-driven optimization, multi-region deployment, and horizontal scaling patterns

Think of this section as a roadmap for transforming your notification service from a reliable delivery system into a sophisticated marketing and engagement platform. Just as a postal service might evolve from basic letter delivery to offering tracking, express services, international shipping, and package analytics, our notification service can evolve beyond simple message delivery to include intelligent optimization, campaign orchestration, and global scale operations.

The extensions we'll explore fall into two major categories: **advanced features** that enhance the user experience and marketing capabilities, and **scalability extensions** that enable the system to handle enterprise-scale volumes and global deployments. Each extension builds upon the solid foundation of channel abstraction, template management, user preferences, and delivery tracking we've established in the core milestones.

Advanced Feature Extensions

Advanced features transform the notification service from a utility into a strategic business asset. These capabilities enable marketing teams to run sophisticated campaigns, product managers to optimize engagement, and business analysts to derive insights from communication patterns.

Campaign Management System

Campaign management extends our individual notification model to support coordinated, multi-message communication sequences. Think of it like the difference between sending individual letters and orchestrating a comprehensive direct mail campaign with multiple touchpoints, timing coordination, and performance tracking.

A **campaign** represents a coordinated series of notifications sent to a defined audience over a specified time period. Unlike individual notifications, campaigns have lifecycles, target audiences, performance goals, and business context that spans multiple messages.

> **Decision: Campaign Data Model**
>
> - **Context**: Need to group related notifications for tracking, optimization, and business reporting
> - **Options Considered**:
>   1. Simple notification tagging with campaign ID
>   2. Dedicated campaign entity with lifecycle management
>   3. Event-driven campaign orchestration with state machines
> - **Decision**: Dedicated campaign entity with lifecycle management
> - **Rationale**: Provides structured approach to campaign planning, execution monitoring, and performance analysis while maintaining clear separation between campaign orchestration and individual notification delivery
> - **Consequences**: Enables sophisticated campaign features but increases system complexity and requires new storage and processing components

Field	Type	Description
ID	string	Unique campaign identifier
Name	string	Human-readable campaign name
Description	string	Campaign purpose and context
Type	CampaignType	Transactional, promotional, or lifecycle campaign
Status	CampaignStatus	Draft, scheduled, active, paused, completed, cancelled
TargetAudience	AudienceDefinition	Rules defining who receives the campaign
Schedule	CampaignSchedule	When and how often messages are sent
Messages	[]CampaignMessage	Sequence of notifications in the campaign
Goals	CampaignGoals	Success metrics and target values
CreatedBy	string	User who created the campaign
CreatedAt	time.Time	Campaign creation timestamp
StartedAt	*time.Time	When campaign execution began
CompletedAt	*time.Time	When campaign finished or was cancelled
Performance	CampaignMetrics	Real-time performance statistics

The **Campaign Orchestrator** manages campaign execution by maintaining state machines for each active campaign and triggering individual notifications based on campaign schedules and audience rules. This component bridges the gap between high-level marketing intent and low-level notification delivery.

Method	Parameters	Returns	Description
CreateCampaign	ctx, definition CampaignDefinition, createdBy string	*Campaign, error	Creates new campaign in draft state
ScheduleCampaign	ctx, campaignID string, schedule CampaignSchedule	error	Moves campaign to scheduled state with execution timeline
StartCampaign	ctx, campaignID string, startedBy string	error	Begins campaign execution and audience processing
PauseCampaign	ctx, campaignID string, reason string	error	Temporarily stops campaign execution
ResumeCampaign	ctx, campaignID string	error	Continues paused campaign from current state
CancelCampaign	ctx, campaignID string, reason string	error	Permanently stops campaign and marks as cancelled
GetCampaignStatus	ctx, campaignID string	*CampaignStatus, error	Returns current campaign state and progress
ProcessScheduledCampaigns	ctx	error	Background job that checks for campaigns ready to execute

Campaign execution follows a sophisticated workflow that respects user preferences, quiet hours, and rate limits while maintaining campaign integrity. The orchestrator generates individual notifications based on campaign templates and audience rules, then delegates to the existing notification processing pipeline.

The **audience targeting system** determines which users receive each campaign message. Unlike static mailing lists, audience definitions use dynamic rules that evaluate user attributes, behavioral data, and preference settings at execution time.

Field	Type	Description
IncludeRules	[]AudienceRule	Conditions that include users in the audience
ExcludeRules	[]AudienceRule	Conditions that remove users from the audience
MaxAudienceSize	*int	Upper limit on campaign recipients
RefreshInterval	time.Duration	How often to re-evaluate dynamic audience rules
SamplingRate	*float64	Percentage of matching users to include (for testing)

Audience rules leverage the same expression language used for routing decisions, enabling complex targeting like "users who registered in the last 30 days AND have not opened an email in 7 days AND have mobile app installed".

A/B Testing Framework

A/B testing enables data-driven optimization of notification content, timing, and delivery channels. Think of it as running controlled experiments where different groups of users receive different variations of the same logical message, allowing us to measure which approach produces better engagement.

> **Decision: A/B Testing Architecture**
>
> - **Context**: Need to systematically test notification variants to optimize engagement rates
> - **Options Considered**:
>   1. Client-side randomization in notification processing
>   2. Campaign-level variant assignment with persistent bucketing
>   3. User-level experiment enrollment with cross-campaign consistency
> - **Decision**: User-level experiment enrollment with cross-campaign consistency
> - **Rationale**: Ensures statistical validity by avoiding assignment bias and enables sophisticated experiment designs like multi-armed bandits
> - **Consequences**: Requires experiment management infrastructure but provides reliable, statistically sound testing capability

The **Experiment Framework** manages test design, user assignment, variant selection, and results analysis. Each experiment defines multiple variants (including a control), assignment methodology, success metrics, and statistical confidence requirements.

Field	Type	Description
ID	string	Unique experiment identifier
Name	string	Human-readable experiment name
Hypothesis	string	What we're testing and expected outcome
Status	ExperimentStatus	Draft, active, paused, completed, archived
TrafficAllocation	float64	Percentage of eligible users to include
Variants	[]ExperimentVariant	Different versions being tested
SuccessMetrics	[]SuccessMetric	How to measure experiment success
AssignmentMethod	AssignmentMethod	Random, stratified, or custom assignment
StartDate	time.Time	When experiment begins
EndDate	*time.Time	When experiment stops (or null for ongoing)
MinSampleSize	int	Minimum participants needed for statistical significance
ConfidenceLevel	float64	Required confidence level (e.g., 0.95 for 95%)
Results	*ExperimentResults	Current statistical analysis

User assignment ensures each user receives consistent treatment throughout the experiment duration. The assignment algorithm uses deterministic hashing of user ID and experiment ID, ensuring the same user always gets the same variant while maintaining random distribution across the population.

The assignment process follows these steps:

1. Check if user is eligible for the experiment based on targeting rules
2. Calculate assignment hash using `hash(userID + experimentID + salt)`
3. Map hash to variant based on traffic allocation percentages
4. Store assignment in persistent store to ensure consistency
5. Return variant configuration for use in notification processing

Variant configuration modifies notification processing at multiple levels. A variant can override template content, change delivery channels, adjust send timing, or modify personalization rules. The experiment framework integrates with the existing template engine and routing system to apply these modifications transparently.

Field	Type	Description
Name	string	Variant identifier (e.g., "control", "subject_line_test")
Weight	float64	Probability of assignment (sum must equal 1.0)
TemplateOverrides	map[string]string	Template ID mappings for this variant
ChannelPreferences	[]string	Preferred delivery channels for variant
SendTimeOffset	*time.Duration	Delay from original send time
PersonalizationRules	map[string]interface{}	Custom personalization parameters

Statistical analysis runs continuously to determine when experiments reach significance and which variants are performing best. The analysis engine calculates conversion rates, confidence intervals, and p-values for each success metric.

ML-Driven Send Time Optimization

Machine learning can dramatically improve engagement by predicting the optimal time to send each notification to each user. Think of it as having a personal assistant who knows exactly when each person is most likely to read their messages, based on their historical behavior patterns.

Decision: Send Time Optimization Approach

- **Context**: Different users are active at different times, and sending notifications when users are engaged increases open and click rates
- **Options Considered**:
  1. Simple time zone adjustment with fixed "best hours"
  2. Cohort-based optimization using demographic segments
  3. Individual user behavior modeling with ML prediction
- **Decision**: Individual user behavior modeling with ML prediction
- **Rationale**: Provides highest precision by learning each user's unique patterns while falling back to cohort data for new users
- **Consequences**: Requires ML infrastructure and training data but can significantly improve engagement rates

The **Send Time Optimizer** analyzes user engagement patterns to predict optimal delivery windows. The system learns from historical data including open times, click times, app usage patterns, and time zone information to build personalized engagement profiles.

Field	Type	Description
UserID	string	User this model applies to
TimeZone	string	User's detected time zone
EngagementWindows	[]TimeWindow	Predicted high-engagement periods
Confidence	float64	Model confidence (0.0 to 1.0)
LastTrainingDate	time.Time	When model was last updated
FeatureImportance	map[string]float64	Which factors most influence this user
FallbackCohort	string	Cohort to use if individual data insufficient

The **feature extraction pipeline** processes user interaction data to identify patterns. Features include time-of-day preferences, day-of-week patterns, seasonal variations, device usage correlation, and notification category preferences.

Key features for send time optimization:

Feature Category	Examples	Data Source
Temporal Patterns	Hour of day, day of week, season	Engagement timestamps
Device Correlation	Mobile vs desktop usage times	Device tracking events
Category Preferences	News vs promotional vs transactional	Notification interaction data
External Factors	Local weather, holidays, events	Third-party data sources
Recency Effects	How recent behavior differs from historical	Sliding window analysis

The **prediction engine** uses these features to score potential send times and select optimal delivery windows. For each notification, the system evaluates possible send times within business constraints and selects the time with highest predicted engagement probability.

Model training occurs continuously using incremental learning to adapt to changing user behavior. The system retrains individual user models weekly and updates cohort models daily based on aggregated behavior patterns.

The optimizer integrates with the existing routing engine by modifying the `RoutingPlan` to include optimal send times:

Field	Type	Description
OptimalSendTime	*time.Time	ML-predicted best delivery time
EngagementProbability	float64	Predicted engagement likelihood
ModelConfidence	float64	How confident the prediction is
FallbackTime	time.Time	Default time if optimal time passes
WindowDuration	time.Duration	How long the optimal window lasts

⚠️ **Pitfall: Cold Start Problem** New users have no engagement history, making predictions impossible. The system must gracefully fall back to cohort-based predictions and gradually transition to personalized models as data accumulates. Cohort assignment should consider available user attributes like location, device type, and signup source.

⚠️ **Pitfall: Feedback Loop Bias** If the system only sends notifications at predicted optimal times, it never learns about user behavior at other times. Include controlled randomization to continuously gather data across all time periods and avoid local optimization maxima.

Scalability Extensions

As notification volume grows and global deployment becomes necessary, the system must evolve beyond single-instance operation to distributed, horizontally scalable architecture. These extensions enable handling millions of notifications daily across multiple geographic regions.

Multi-Region Deployment

Global applications require notification delivery from multiple regions to minimize latency, comply with data residency requirements, and provide disaster recovery capabilities. Think of it like a postal service opening regional distribution centers — each center handles local delivery while coordinating with the central system for consistency.

> **Decision: Multi-Region Architecture**
>
> - **Context**: Need to serve global users with low latency while meeting data residency requirements
> - **Options Considered**:
>   1. Single global instance with edge caches
>   2. Regional instances with shared global database
>   3. Fully distributed instances with cross-region replication
> - **Decision**: Regional instances with shared global database for preferences and templates, local databases for delivery tracking
> - **Rationale**: Balances performance, consistency, and operational complexity while meeting data residency requirements
> - **Consequences**: Enables global scale with acceptable complexity but requires sophisticated replication and conflict resolution

Regional deployment topology places notification processing instances in multiple geographic regions, each handling users assigned to that region. User assignment considers data residency requirements, network latency, and provider availability.

Component	Deployment Strategy	Data Locality	Replication Method
API Gateway	Regional instances with global load balancer	Request routing only	Stateless — no replication needed
Notification Service	Regional instances	Local processing queues	Cross-region queue federation
Template Store	Global primary with regional read replicas	Globally consistent	Master-slave replication
User Preferences	Regional primary with cross-region sync	Region-specific with backup	Multi-master with conflict resolution
Delivery Tracking	Regional instances	Local to processing region	Eventually consistent aggregation
Analytics	Regional collection with global aggregation	Local collection, global reporting	Streaming aggregation to global data warehouse

User assignment determines which region processes notifications for each user. The assignment algorithm considers data residency laws (GDPR, CCPA), user location, and regional provider availability.

The **Global Coordinator** manages cross-region consistency for user preferences and template distribution. When a user updates preferences in one region, the coordinator propagates changes to all regions while handling conflicts and ensuring eventual consistency.

Method	Parameters	Returns	Description
AssignUserToRegion	userID, location, residencyRules	regionID, error	Determines optimal region for user
PropagatePreferenceChange	userID, change PreferenceChange, originRegion	error	Replicates preference updates across regions
SynchronizeTemplates	templateID, version, targetRegions	error	Distributes template changes to specified regions
ResolvePreferenceConflict	userID, conflictingChanges	resolvedPreference, error	Handles concurrent preference updates
GetUserRegion	userID	regionID, error	Looks up assigned region for user

Cross-region replication uses eventually consistent synchronization for user preferences and immediate consistency for templates. The replication strategy prioritizes availability over consistency for preferences (users can tolerate slightly stale settings) but ensures consistency for templates (incorrect content is worse than delayed content).

⚠️ **Pitfall: Split-Brain Scenarios** Network partitions can create situations where regions cannot communicate, potentially leading to conflicting preference changes. Implement regional quorum requirements and prefer availability over consistency for non-critical operations during partitions.

Notification Batching

High-volume applications benefit from batching individual notifications into bulk operations, reducing provider API calls, improving throughput, and often reducing costs. Think of it like consolidating individual letters into bulk mail shipments — more efficient for both sender and postal service.

> **Decision: Batching Strategy**
>
> - **Context**: Individual API calls don't scale to millions of notifications daily and most providers offer bulk sending discounts
> - **Options Considered**:
>   1. Simple time-based batching (collect for X seconds)
>   2. Size-based batching (send when batch reaches Y notifications)
>   3. Adaptive batching based on volume and provider capabilities
> - **Decision**: Adaptive batching based on volume and provider capabilities
> - **Rationale**: Optimizes throughput and cost while maintaining acceptable latency for different notification priorities
> - **Consequences**: Improves system efficiency but adds complexity in batch formation and partial failure handling

The **Batch Coordinator** collects individual notifications into optimally-sized batches based on provider capabilities, notification priority, and current system load.

Field	Type	Description
ProviderLimits	map[string]BatchLimits	Maximum batch sizes per provider
BatchWindows	map[Priority]time.Duration	How long to collect notifications by priority
MaxBatchSize	map[string]int	Maximum notifications per batch by channel
FlushTriggers	[]FlushCondition	Conditions that force immediate batch sending

Batch formation balances efficiency with latency requirements. High-priority notifications trigger immediate batch formation, while lower-priority notifications can wait for more optimal batch sizes.

The batching algorithm follows these steps:

1. Notifications arrive and are sorted by priority and channel
2. High-priority notifications immediately trigger batch formation if batch is non-empty
3. Normal-priority notifications accumulate until batch size limit or time window expires
4. Low-priority notifications wait for optimal batch sizes unless maximum window expires
5. Formed batches are sent to appropriate channel handlers
6. Individual notification status is tracked even within batch operations

Partial failure handling becomes critical with batching since some notifications in a batch may succeed while others fail. The system must track individual notification status within batch operations and retry failed notifications appropriately.

Field	Type	Description
BatchID	string	Unique identifier for this batch
NotificationIDs	[]string	Individual notifications in batch
ChannelType	string	Delivery channel for this batch
Provider	string	Which provider will handle the batch
CreatedAt	time.Time	When batch was formed
SentAt	*time.Time	When batch was sent to provider
CompletedAt	*time.Time	When all individual results were processed
SuccessCount	int	Notifications that delivered successfully
FailureCount	int	Notifications that failed delivery
PartialResults	[]NotificationResult	Individual status for each notification

⚠️ **Pitfall: Batch Poisoning** One malformed notification can cause an entire batch to fail. Implement pre-batch validation and isolation for suspicious notifications to prevent cascading failures.

⚠️ **Pitfall: Status Tracking Complexity** Batching makes it harder to provide real-time status updates since individual notification status becomes available only after batch completion. Consider exposing batch-level status while batch is processing, then individual status when available.

Horizontal Scaling Patterns

As notification volume grows beyond single-instance capacity, the system must scale horizontally across multiple processing nodes. This requires careful attention to state management, work distribution, and coordination between instances.

> **Decision: Scaling Architecture**
>
> - **Context**: Single instances cannot handle enterprise notification volumes and manual scaling is operationally unsustainable
> - **Options Considered**:
>   1. Stateless processing with external coordination
>   2. Sharded processing with consistent hashing
>   3. Actor model with location transparency
> - **Decision**: Stateless processing with external coordination via message queues and shared state stores
> - **Rationale**: Provides linear scalability with operational simplicity and integrates well with existing message queue infrastructure
> - **Consequences**: Enables automatic scaling but requires careful design of shared state and coordination protocols

Stateless notification processors handle the core notification workflow without maintaining local state. All persistent state lives in shared databases and message queues, enabling processors to be added or removed dynamically without coordination.

Each processor instance runs the same core workflow:

1. Pull notifications from shared message queue
2. Load user and template data from shared stores
3. Process notification through routing, templating, and channel logic
4. Update delivery status in shared tracking store
5. Publish analytics events to shared event stream

Work distribution uses message queue partitioning to distribute load across processor instances. The partitioning strategy ensures related notifications (same user, same campaign) can be processed in order while maintaining load balance.

Partition Strategy	Benefits	Trade-offs
Round-robin	Perfect load balance	No ordering guarantees
User ID hash	Maintains per-user ordering	Uneven load if user activity varies
Campaign ID hash	Processes campaigns consistently	Poor load balance for large campaigns
Priority-based	High priority gets dedicated resources	Complex partition management

Auto-scaling monitors queue depth, processing latency, and system resource utilization to automatically adjust the number of processor instances. The scaling algorithm uses predictive models to anticipate demand spikes and pre-scale capacity.

Metric	Scale-Up Trigger	Scale-Down Trigger	Stabilization Period
Queue Depth	> 1000 messages for 2 minutes	< 100 messages for 10 minutes	5 minutes
Processing Latency	> 30 seconds average	< 5 seconds average	10 minutes
CPU Utilization	> 70% for 5 minutes	< 30% for 15 minutes	10 minutes
Memory Usage	> 80% for 3 minutes	< 50% for 15 minutes	5 minutes

State coordination between instances handles scenarios where multiple processors might operate on related data. The coordination strategy uses optimistic locking and idempotency keys to prevent conflicts while maintaining high throughput.

Deployment strategies support zero-downtime updates and gradual rollouts. The system uses blue-green deployments with traffic shifting to minimize disruption during updates.

Deployment Phase	Traffic Allocation	Health Checks	Rollback Triggers
Initial Deploy	0% to new version	Basic connectivity	Deployment failure
Canary Phase	5% to new version	Full health suite	Error rate > 1%
Gradual Rollout	25%, 50%, 75% phases	Performance monitoring	Latency increase > 50%
Full Migration	100% to new version	End-to-end validation	Customer complaints

Implementation Guidance

The advanced features and scalability extensions represent significant architectural evolution beyond the core notification service. Each extension requires careful planning, gradual implementation, and thorough testing to avoid disrupting existing functionality.

Technology Recommendations

Component	Simple Option	Advanced Option
Campaign Management	PostgreSQL with JSON fields	Event sourcing with Apache Kafka
A/B Testing	In-memory variant assignment	Dedicated experimentation platform (e.g., Optimizely)
ML Send Time Optimization	Python scikit-learn batch processing	Real-time ML with Apache Spark or TensorFlow Serving
Multi-Region Coordination	Database replication with application logic	Distributed coordination service (e.g., Consul, etcd)
Horizontal Scaling	Kubernetes Horizontal Pod Autoscaler	Custom controller with predictive scaling
Message Queuing	RabbitMQ with clustering	Apache Kafka with multiple partitions
Analytics Storage	PostgreSQL with time-series extensions	Dedicated time-series database (InfluxDB, TimescaleDB)

Recommended Implementation Phases

```
Phase 1: Campaign Management Foundation (2-3 months)
├── campaign-service/
│   ├── models/
│   │   ├── campaign.go           ← Campaign data structures
│   │   ├── audience.go           ← Audience targeting rules
│   │   └── schedule.go           ← Campaign scheduling logic
│   ├── orchestrator/
│   │   ├── orchestrator.go       ← Main campaign execution engine
│   │   ├── audience_builder.go   ← Dynamic audience evaluation
│   │   └── scheduler.go          ← Background scheduling jobs
│   └── storage/
│       ├── postgres.go           ← Campaign persistence
│       └── migrations/           ← Database schema changes

Phase 2: A/B Testing Framework (2-3 months)
├── experiments/
│   ├── assignment/
│   │   ├── bucketing.go          ← User assignment algorithm
│   │   ├── stratification.go     ← Stratified sampling
│   │   └── persistence.go        ← Assignment storage
│   ├── analysis/
│   │   ├── statistics.go         ← Statistical significance testing
│   │   ├── conversion.go         ← Conversion rate calculation
│   │   └── reporting.go          ← Results dashboard
│   └── integration/
│       ├── template_variants.go  ← Template A/B testing
│       ├── channel_variants.go   ← Channel A/B testing
│       └── timing_variants.go    ← Send time A/B testing

Phase 3: ML Send Time Optimization (3-4 months)
├── ml-optimizer/
│   ├── features/
│   │   ├── extraction.go         ← Feature pipeline
│   │   ├── temporal.go           ← Time-based features
│   │   └── behavioral.go         ← User behavior features
│   ├── training/
│   │   ├── pipeline.go           ← ML training pipeline
│   │   ├── validation.go         ← Model validation
│   │   └── deployment.go         ← Model serving
│   └── prediction/
│       ├── optimizer.go          ← Send time prediction
│       ├── cohorts.go            ← Fallback cohort models
│       └── feedback.go           ← Model performance feedback

Phase 4: Scalability Extensions (2-3 months)
├── scaling/
│   ├── coordination/
│   │   ├── global_coordinator.go ← Cross-region coordination
│   │   ├── replication.go        ← Data replication
│   │   └── conflict_resolution.go← Conflict handling
│   ├── batching/
│   │   ├── batch_coordinator.go  ← Batch formation
│   │   ├── adaptive_sizing.go    ← Dynamic batch sizing
│   │   └── partial_failures.go   ← Batch failure handling
│   └── autoscaling/
│       ├── metrics_collector.go  ← Scaling metrics
│       ├── predictor.go          ← Demand prediction
│       └── controller.go         ← Auto-scaling logic
```

Campaign Management Starter Code

```go
// Campaign represents a coordinated series of notifications
type Campaign struct {
    ID              string           `json:"id"`
    Name            string           `json:"name"`
    Description     string           `json:"description"`
    Type            CampaignType     `json:"type"`
    Status          CampaignStatus   `json:"status"`
    TargetAudience  AudienceDefinition `json:"target_audience"`
    Schedule        CampaignSchedule `json:"schedule"`
    Messages        []CampaignMessage `json:"messages"`
    Goals           CampaignGoals    `json:"goals"`
    CreatedBy       string           `json:"created_by"`
    CreatedAt       time.Time        `json:"created_at"`
    StartedAt       *time.Time       `json:"started_at"`
    CompletedAt     *time.Time       `json:"completed_at"`
    Performance     CampaignMetrics  `json:"performance"`
}

type CampaignOrchestrator struct {
    store          CampaignStore
    audienceEngine AudienceEngine
    scheduler      Scheduler
    notificationService NotificationService
    metrics        MetricsCollector
}

// ProcessScheduledCampaigns checks for campaigns ready to execute
func (co *CampaignOrchestrator) ProcessScheduledCampaigns(ctx context.Context) error {
    // TODO 1: Query for campaigns in StatusScheduled with start time <= now
    // TODO 2: For each ready campaign, validate audience and message templates
    // TODO 3: Generate individual notifications for audience members
    // TODO 4: Submit notifications to notification service queue
    // TODO 5: Update campaign status to StatusActive
    // TODO 6: Record campaign execution metrics
    // Hint: Use transactions to ensure campaign state consistency
}
```

```go
// StartCampaign immediately begins campaign execution

func (co *CampaignOrchestrator) StartCampaign(ctx context.Context, campaignID string, startedBy string) error {

    // TODO 1: Load campaign and verify it's in StatusScheduled or StatusDraft

    // TODO 2: Validate all templates exist and are properly configured

    // TODO 3: Evaluate audience rules to build recipient list

    // TODO 4: Check that audience size is within campaign limits

    // TODO 5: Generate notifications for all audience members

    // TODO 6: Update campaign status to StatusActive with timestamp

    // TODO 7: Schedule background job for campaign progress monitoring

}
```

A/B Testing Assignment Algorithm

```go
type ExperimentAssigner struct {
    store       AssignmentStore
    experiments map[string]*Experiment
    hasher      hash.Hash
    mutex       sync.RWMutex
}

// AssignUserToVariant determines which variant a user should receive
func (ea *ExperimentAssigner) AssignUserToVariant(ctx context.Context, userID, experimentID string) (*VariantAssignment, error) {
    // TODO 1: Check if user already has assignment for this experiment
    // TODO 2: If existing assignment found, return it (consistency guarantee)
    // TODO 3: Load experiment configuration and validate it's active
    // TODO 4: Check if user meets experiment targeting criteria
    // TODO 5: Calculate assignment hash using userID + experimentID + salt
    // TODO 6: Map hash value to variant based on traffic allocation percentages
    // TODO 7: Store assignment in persistent store for future consistency
    // TODO 8: Return variant assignment with metadata
    // Hint: Use deterministic hashing to ensure same user gets same variant
}

// CalculateAssignmentHash generates deterministic hash for variant assignment
func (ea *ExperimentAssigner) CalculateAssignmentHash(userID, experimentID, salt string) uint64 {
    // TODO 1: Reset hasher state to ensure clean calculation
    // TODO 2: Write userID, experimentID, and salt to hasher in specific order
    // TODO 3: Calculate hash sum and convert to uint64
    // TODO 4: Return hash value for variant mapping
    // Hint: Use fnv.New64a() for fast, deterministic hashing
}
```

ML Send Time Optimization Framework

```go
type SendTimeOptimizer struct {
    models      map[string]*UserModel
    cohorts     map[string]*CohortModel
    features    FeatureExtractor
    predictor   MLPredictor
    fallback    FallbackStrategy
    modelStore  ModelStore
    mutex       sync.RWMutex
}

// OptimizeSendTime predicts best delivery time for user and notification
func (sto *SendTimeOptimizer) OptimizeSendTime(ctx context.Context, userID string, notification *NotificationMessage)
(*SendTimeRecommendation, error) {
    // TODO 1: Load user's historical engagement data and timezone
    // TODO 2: Extract features for current notification and user context
    // TODO 3: Load personalized model for user (or fallback to cohort model)
    // TODO 4: Generate send time predictions for next 24-48 hours
    // TODO 5: Apply business constraints (quiet hours, business hours)
    // TODO 6: Select optimal time with highest engagement probability
    // TODO 7: Return recommendation with confidence score and fallback time
}

// TrainUserModel updates ML model based on recent engagement data
func (sto *SendTimeOptimizer) TrainUserModel(ctx context.Context, userID string) error {
    // TODO 1: Collect user's engagement events from last 90 days
    // TODO 2: Extract temporal and behavioral features
    // TODO 3: Prepare training dataset with positive/negative examples
    // TODO 4: Train or update user's personalized model
    // TODO 5: Validate model performance against holdout data
    // TODO 6: Store updated model if performance improves
    // TODO 7: Fall back to cohort model if insufficient data
}
```

Multi-Region Coordination

```go
type GlobalCoordinator struct {

    regions         map[string]RegionClient

    replicationLog  ReplicationLog

    conflictResolver ConflictResolver

    consistency     ConsistencyLevel

    topology        RegionTopology

}


// PropagatePreferenceChange replicates user preference updates across regions

func (gc *GlobalCoordinator) PropagatePreferenceChange(ctx context.Context, userID string, change PreferenceChange,
originRegion string) error {

    // TODO 1: Determine which regions need this preference update

    // TODO 2: Create replication log entry with vector clock

    // TODO 3: Send async replication requests to each target region

    // TODO 4: Handle partial failures and retry mechanisms

    // TODO 5: Update replication status when all regions confirm

    // TODO 6: Clean up old replication log entries

}


// ResolvePreferenceConflict handles concurrent updates from different regions

func (gc *GlobalCoordinator) ResolvePreferenceConflict(ctx context.Context, userID string, conflicts []PreferenceConflict)
(*UserPreference, error) {

    // TODO 1: Load current preference state from all regions

    // TODO 2: Compare vector clocks to determine causality

    // TODO 3: Apply conflict resolution rules (last-writer-wins, merge, etc.)

    // TODO 4: Generate resolved preference state

    // TODO 5: Propagate resolution to all regions

    // TODO 6: Log conflict resolution for debugging and analysis

}
```

Horizontal Scaling Auto-Scaler

```go
type AutoScaler struct {
    metricsCollector MetricsCollector
    scalingPolicies  map[string]ScalingPolicy
    kubernetesClient kubernetes.Interface
    predictor        DemandPredictor
    cooldownTracker  map[string]time.Time
}

// EvaluateScaling checks metrics and determines if scaling is needed
func (as *AutoScaler) EvaluateScaling(ctx context.Context) error {
    // TODO 1: Collect current system metrics (queue depth, CPU, memory)
    // TODO 2: Compare metrics against scaling policy thresholds
    // TODO 3: Check cooldown periods to prevent thrashing
    // TODO 4: Use demand predictor to anticipate future load
    // TODO 5: Calculate optimal replica count for each component
    // TODO 6: Execute scaling operations via Kubernetes API
    // TODO 7: Update cooldown timers and record scaling events
}

// PredictDemand uses historical data to forecast notification volume
func (as *AutoScaler) PredictDemand(ctx context.Context, lookAhead time.Duration) (*DemandForecast, error) {
    // TODO 1: Load historical notification volume data
    // TODO 2: Account for seasonal patterns (daily, weekly, holiday)
    // TODO 3: Consider scheduled campaigns that will increase load
    // TODO 4: Apply trend analysis for growth patterns
    // TODO 5: Generate volume forecast with confidence intervals
    // TODO 6: Translate volume forecast to resource requirements
}
```

Milestone Checkpoints

Campaign Management Validation:

```bash
# Start campaign service
go run cmd/campaign-service/main.go

# Create test campaign
curl -X POST localhost:8080/campaigns \
  -d '{"name":"Welcome Series","type":"lifecycle","messages":[...]}'

# Verify campaign execution
curl localhost:8080/campaigns/test-campaign-1/status
# Expected: {"status":"active","progress":{"sent":150,"delivered":142}}
```

A/B Testing Verification:

```bash
# Create experiment
curl -X POST localhost:8080/experiments \
  -d '{"name":"Subject Line Test","variants":[{"name":"control"},{"name":"emoji"}]}'

# Test user assignment consistency
for i in {1..100}; do
  curl localhost:8080/experiments/test-1/assign/user-$i
done
# Verify: Same user always gets same variant
```

ML Optimization Testing:

```bash
# Train models
go run cmd/ml-trainer/main.go --users=1000 --days=30

# Test prediction
curl localhost:8080/optimize/send-time \
  -d '{"user_id":"user123","notification":{"category":"marketing"}}'
# Expected: {"optimal_time":"2024-01-15T14:30:00Z","confidence":0.87}
```

Debugging Advanced Features

Symptom	Likely Cause	Diagnosis	Fix
Campaign stuck in "sending"	Audience query timeout	Check audience evaluation logs	Optimize audience query or add timeout
A/B test showing no difference	Insufficient sample size	Check experiment metrics dashboard	Wait for more data or increase traffic allocation
ML predictions always same time	Model not training	Check training pipeline logs	Verify training data quality and pipeline execution
Cross-region sync delays	Network latency or failures	Monitor replication lag metrics	Add retry logic and fallback regions
Auto-scaler thrashing	Conflicting scaling policies	Review scaling decision logs	Adjust thresholds and cooldown periods

These future extensions transform the notification service from a utility into a comprehensive engagement platform. Each extension builds upon the solid foundation established in the core milestones while adding sophisticated capabilities for campaign management, optimization, and global scale operation.

Glossary

> **Milestone(s):** All milestones — provides comprehensive definitions for all technical terms, acronyms, and domain-specific vocabulary used throughout the notification service design and implementation

Think of a glossary as a dictionary for our specialized domain. When building complex systems like a multi-channel notification service, we develop a rich vocabulary of technical terms, each with precise meanings that may differ from common usage. Just as a legal contract defines key terms to prevent ambiguity, our design document establishes clear definitions to ensure all engineers share the same understanding of concepts, components, and behaviors.

This glossary serves as the authoritative reference for terminology used throughout the notification service design document. It includes technical terms, domain-specific vocabulary, architectural patterns, and implementation concepts. Each definition provides not just the meaning, but also context for how the term applies specifically to our notification system.

Core System Concepts

Channel Abstraction A unified interface that provides a consistent programming model across different notification providers (email, SMS, push notifications, in-app messages). The abstraction allows the notification service to treat all delivery mechanisms uniformly while hiding the complexity of provider-specific APIs, authentication methods, and message formats. This enables features like intelligent routing, fallback handling, and provider switching without affecting the core notification logic.

Routing Engine The decision-making component responsible for selecting the appropriate notification channels based on message type, user preferences, channel availability, and business rules. The routing engine evaluates multiple factors including user opt-out preferences, quiet hours, rate limits, circuit breaker states, and provider health to determine the optimal delivery strategy. It produces a routing plan that specifies primary and fallback channels with retry policies.

Template Engine A system for rendering notification content by substituting variables into predefined templates while applying proper escaping for different output contexts. The template engine handles localization by selecting the appropriate language variant, formatting variables according to cultural conventions, and falling back to default locales when specific translations are unavailable. It enforces security policies to prevent XSS attacks in HTML content.

Delivery Tracking A comprehensive monitoring system that follows notifications through their complete lifecycle, from initial submission through final delivery confirmation or failure. Delivery tracking processes webhooks from external providers, maintains a state machine for each notification, aggregates metrics for analytics, and handles bounce processing. It provides real-time visibility into notification performance and delivery issues.

User Preferences A hierarchical system of controls that allows recipients to specify which types of notifications they want to receive through which channels. Preferences operate at multiple levels (global, category, channel) and include settings for quiet hours, rate limiting, and unsubscribe status. The preference system ensures compliance with regulations like CAN-SPAM and GDPR while providing granular user control.

Channel and Provider Terms

Provider Integration The implementation layer that connects to external notification services like SendGrid for email, Twilio for SMS, or Firebase Cloud Messaging for push notifications. Each provider integration handles authentication, API-specific message formatting, rate limiting, error handling, and webhook processing. Provider integrations are pluggable through the channel abstraction interface.

Fallback Strategy A resilience pattern where backup delivery channels are automatically attempted when the primary channel fails to deliver a notification. The fallback strategy considers channel priorities, user preferences, and message urgency to select alternative delivery methods. For example, if push notification delivery fails, the system might fall back to SMS for urgent messages or email for non-urgent communications.

Circuit Breaker A reliability pattern that prevents repeated calls to failing external services by "opening" when failure rates exceed thresholds and allowing limited test requests during recovery. In the notification service, circuit breakers protect against provider outages by temporarily routing traffic away from failing providers while monitoring for service recovery.

Provider Webhook HTTP callbacks sent by external notification services to report delivery status updates, bounces, opens, clicks, and other events. Webhooks are authenticated using HMAC signatures to ensure they originate from the legitimate provider. The webhook processor parses provider-specific event formats into standardized internal events for delivery tracking.

Template and Localization Terms

Variable Substitution The process of replacing placeholder tokens in notification templates with actual data values at send time. Variable substitution uses mustache-style syntax with double braces ({{variable}}) and applies context-aware escaping to prevent security vulnerabilities. The substitution

engine supports nested objects, conditional logic, and formatting helpers for dates, numbers, and currency.

Locale Fallback A strategy for selecting the best available language variant when the user's exact locale preference is not available. Locale fallback follows a hierarchy from specific to general (e.g., en-US → en → system default) and ensures that users always receive content in a language they can understand, even if their preferred translation is missing.

Content Validation Security and correctness checks applied to notification templates before they can be used in production. Content validation includes XSS protection through HTML sanitization, length limits for SMS content, required field verification, and syntax checking for template variables. Invalid templates are rejected with specific error messages to help developers fix issues.

Template Versioning A system for tracking changes to notification templates over time, enabling rollback to previous versions when issues are discovered. Each template version is immutable and includes metadata about who made the change, when it was created, and what was modified. Template versioning supports A/B testing and gradual rollout of template changes.

ICU MessageFormat An internationalization standard for handling complex text formatting including pluralization, gender selection, and number formatting. ICU MessageFormat is particularly important for notifications that include variable quantities (e.g., "You have 1 message" vs "You have 5 messages") and need to handle different grammatical rules across languages.

User Preference and Compliance Terms

Unsubscribe System A compliance-friendly mechanism that allows users to opt out of notifications through one-click links, preference management interfaces, or reply commands. The unsubscribe system generates cryptographically secure tokens to prevent forgery, immediately honors unsubscribe requests, and maintains audit logs for regulatory compliance.

HMAC-Signed Tokens Cryptographic tokens that prevent tampering and enumeration attacks on unsubscribe links. Each token includes the user ID, scope of unsubscribe (category, channel, or global), expiration time, and a signature generated using a secret key. Only tokens with valid signatures are accepted for unsubscribe processing.

Preference Hierarchy A layered system where user preferences are resolved by combining global settings, category-specific preferences, and channel-specific preferences. More specific preferences override general ones, allowing fine-grained control. For example, a user might opt out of marketing emails globally but still receive transactional emails and all SMS messages.

Regulatory Compliance Adherence to legal requirements for electronic communications, including CAN-SPAM (US), GDPR (EU), and CASL (Canada). Compliance features include mandatory unsubscribe links, sender identification, suppression list management, consent tracking, and data retention policies. Transactional messages have different rules from marketing communications.

Quiet Hours Time windows during which non-urgent notifications are suppressed to respect user preferences for do-not-disturb periods. Quiet hours are defined in the user's local timezone and can be overridden for high-priority notifications like security alerts or emergency communications.

Delivery and Analytics Terms

State Machine A model that defines the possible states of a notification (queued, sent, delivered, opened, clicked, failed, bounced) and the allowed transitions between states. The state machine ensures that delivery status updates follow a logical progression and helps detect invalid webhook events or system inconsistencies.

Open Tracking A technique for detecting when email recipients view a message by embedding a transparent 1x1 pixel image with a unique URL. When the email client loads the image, it triggers a request to the tracking server, which records the open event. Open tracking has limitations due to image blocking and privacy features in modern email clients.

Click Tracking URL rewriting that replaces original links with tracking URLs that redirect through the notification service. When users click links, the tracking system records the click event with metadata like timestamp, link position, and user agent before redirecting to the original destination. Click tracking provides engagement metrics and attribution data.

Bounce Handling Processing of delivery failure notifications from email providers, categorizing them as hard bounces (permanent failures) or soft bounces (temporary failures). Hard bounces trigger immediate suppression to prevent future delivery attempts to invalid addresses, while soft bounces allow retries with backoff delays.

UTM Parameters URL tracking parameters (utm_source, utm_medium, utm_campaign, utm_content, utm_term) that provide attribution data when users click notification links. UTM parameters help track which notifications drive traffic and conversions in analytics platforms like Google Analytics.

Real-time Metrics Streaming aggregation of delivery events to provide immediate insights into notification performance. Real-time metrics are computed as events arrive and provide dashboards with current delivery rates, failure rates, and engagement metrics without waiting for batch processing.

Batch Reconciliation Nightly reprocessing of delivery events to ensure accuracy of historical metrics and correct any discrepancies from real-time processing. Batch reconciliation handles late-arriving webhooks, corrects duplicate events, and provides authoritative metrics for reporting and analytics.

Error Handling and Reliability Terms

Exponential Backoff A retry strategy where the delay between retry attempts increases exponentially (e.g., 1s, 2s, 4s, 8s) to reduce load on failing services while allowing time for recovery. Jitter is often added to prevent thundering herd problems when many clients retry simultaneously.

Dead Letter Queue A storage mechanism for notifications that have exhausted all retry attempts and cannot be delivered. Dead letter queues preserve failed messages for manual investigation, allow reprocessing after fixes are applied, and provide audit trails for delivery failures. Messages in the dead letter queue eventually expire to prevent unbounded growth.

Circuit Breaker States The three operational states of a circuit breaker: Closed (normal operation), Open (failing fast without calling the service), and Half-Open (testing recovery with limited requests). State transitions are based on failure rates, success rates, and timeout configurations.

Failure Mode Analysis Systematic examination of potential failure scenarios including provider outages, rate limiting, authentication errors, template rendering failures, and webhook processing errors. Failure mode analysis identifies detection mechanisms, recovery strategies, and prevention measures for each type of failure.

Cascade Prevention Strategies to prevent failures from propagating through the system and causing total outages. Cascade prevention includes circuit breakers, bulkheading, timeout settings, graceful degradation, and fallback mechanisms that maintain partial functionality when components fail.

Testing and Debugging Terms

Provider Mocking Simulation of external notification services for testing purposes. Mock providers implement the same interface as real providers but return predictable responses, allow failure simulation, and track requests for verification. Provider mocking enables testing without sending real notifications or incurring costs.

Test Isolation Ensuring that automated tests don't interfere with each other by using separate databases, clean fixtures, and independent test data. Test isolation prevents flaky tests and allows parallel test execution for faster feedback cycles.

Correlation ID A unique identifier that follows a request through all system components, enabling distributed tracing and log correlation. Correlation IDs help debug issues by linking related log entries across services and providing end-to-end visibility into request processing.

Structured Logging JSON-formatted log entries with consistent fields for timestamp, level, correlation ID, component, and message. Structured logging enables automated log parsing, filtering, and analysis while providing human-readable output for debugging.

Milestone Validation Systematic verification that implementation meets the acceptance criteria for each project milestone. Milestone validation includes automated tests, manual testing scenarios, performance benchmarks, and code review checkpoints.

Advanced Features and Extensions

A/B Testing Framework Infrastructure for conducting controlled experiments on notification content, delivery timing, or channels. A/B testing includes user assignment algorithms, variant definition, statistical significance testing, and automated experiment conclusion based on performance metrics.

Campaign Management Coordinated sequences of related notifications sent to target audiences over time. Campaigns include audience definition, message scheduling, delivery pacing, performance tracking, and optimization based on engagement metrics.

Send Time Optimization Machine learning-driven prediction of optimal delivery times for individual users based on their historical engagement patterns. Send time optimization considers factors like timezone, device type, app usage patterns, and engagement history to maximize open and click rates.

Multi-region Deployment Global distribution of notification services across geographic regions for reduced latency, compliance with data residency requirements, and improved resilience. Multi-region deployment includes data replication, request routing, and conflict resolution for user preferences.

Auto-scaling Automatic adjustment of computational resources based on notification volume and processing demands. Auto-scaling monitors queue depth, processing latency, and throughput metrics to add or remove instances dynamically while maintaining performance targets.

Acronyms and Technical Terms

API (Application Programming Interface) A contract that defines how software components communicate, specifying request formats, response formats, error codes, and authentication requirements. In the notification service, APIs include REST endpoints for notification submission and webhook endpoints for status updates.

GDPR (General Data Protection Regulation) European Union regulation that governs personal data processing, including notification preferences and consent management. GDPR requires explicit consent for marketing communications, data portability, and the right to be forgotten.

CAN-SPAM (Controlling the Assault of Non-Solicited Pornography and Marketing) United States federal law that sets requirements for commercial email, including sender identification, truthful subject lines, unsubscribe mechanisms, and physical address disclosure.

HMAC (Hash-based Message Authentication Code) A cryptographic technique that ensures message integrity and authenticity using a secret key and hash function. HMAC signatures prevent tampering with unsubscribe tokens and validate webhook authenticity.

SMS (Short Message Service) Text messaging standard with 160-character limit per segment. SMS costs are typically per-message, making careful content optimization and appropriate usage (urgent notifications only) important for cost management.

FCM (Firebase Cloud Messaging) Google's cross-platform messaging service for mobile push notifications. FCM handles device token management, message delivery, and provides analytics for push notification performance.

APNs (Apple Push Notification service) Apple's service for delivering push notifications to iOS, iPadOS, macOS, and watchOS devices. APNs requires certificates for authentication and has specific payload format requirements.

JSON (JavaScript Object Notation) A lightweight data interchange format used for API requests, webhook payloads, configuration files, and structured logging. JSON is human-readable and widely supported across programming languages.

HTTP (HyperText Transfer Protocol) The protocol used for web communication, including REST APIs and webhooks. HTTP status codes (2xx, 4xx, 5xx) provide standardized response classification for error handling and retry logic.

REST (Representational State Transfer) An architectural style for web APIs that uses HTTP methods (GET, POST, PUT, DELETE) and standard status codes. RESTful APIs are stateless, cacheable, and follow consistent URL patterns for resource management.

Data Types and Constants

Priority Levels Enumerated values that determine notification urgency and handling:

- `PriorityLow`: Non-urgent notifications that respect all user preferences and quiet hours
- `PriorityNormal`: Standard notifications with default handling
- `PriorityHigh`: Important notifications that may override some preferences
- `PriorityUrgent`: Critical notifications that bypass quiet hours and rate limits

Notification Status Values Enumerated states in the notification lifecycle:

- `StatusQueued`: Notification accepted and queued for processing
- `StatusSending`: Notification being processed and sent to provider
- `StatusSent`: Notification successfully submitted to provider
- `StatusDelivered`: Provider confirmed delivery to recipient
- `StatusOpened`: Recipient opened the notification (email only)
- `StatusClicked`: Recipient clicked a link in the notification
- `StatusFailed`: Delivery failed with non-recoverable error
- `StatusBounced`: Message bounced back from recipient address

Error Categories Classification of errors for retry and handling logic:

- `ErrorCategoryTransient`: Temporary failures suitable for retry (network timeouts)
- `ErrorCategoryRate`: Rate limiting requiring exponential backoff
- `ErrorCategoryAuth`: Authentication errors requiring manual intervention
- `ErrorCategoryPermanent`: Permanent failures requiring dead letter queue
- `ErrorCategoryTemplate`: Template rendering errors requiring code fixes

Implementation Guidance

This glossary provides the complete vocabulary for implementing and maintaining the multi-channel notification service. When working with the codebase, use these exact terms for consistency and clarity. The following table maps key concepts to their Go implementation patterns:

Concept	Go Implementation Pattern	Key Packages
Channel Abstraction	Interface with concrete structs	`internal/channels/`
Template Engine	Template parsing with `text/template`	`internal/templates/`
User Preferences	Struct with JSON serialization	`internal/preferences/`
Delivery Tracking	Event sourcing with state machine	`internal/tracking/`
Circuit Breaker	State machine with mutex protection	`internal/reliability/`
Webhook Processing	HTTP handler with signature validation	`internal/webhooks/`
Message Queue	Publisher/subscriber with RabbitMQ	`internal/queue/`
Database Models	Structs with SQL tags	`internal/models/`

Recommended Development Workflow:

1. **Start with interfaces** - Define the `Channel`, `TemplateEngine`, and `PreferenceManager` interfaces first
2. **Implement core types** - Create `NotificationMessage`, `User`, `Template`, and `UserPreference` structs
3. **Build channel abstraction** - Implement concrete channels for email, SMS, and push notifications
4. **Add template processing** - Create template engine with variable substitution and localization
5. **Implement preference system** - Build preference storage and evaluation logic
6. **Create delivery tracking** - Add webhook processing and metrics collection
7. **Add error handling** - Implement circuit breakers, retries, and dead letter queues

Testing Approach:

Use the exact type names and method signatures defined in this glossary when writing tests. Create mock implementations for external providers and use the `MockEmailProvider`, `MockSMSProvider`, and `WebhookSimulator` utilities provided in the testing framework.

Common Integration Points:

- **Message Queue**: Use `Publisher.PublishNotification()` to submit notifications
- **Template Rendering**: Call `TemplateEngine.RenderTemplate()` for content generation
- **Preference Checking**: Use `PreferenceManager.CanReceive()` for delivery decisions
- **Status Updates**: Process webhooks through `WebhookProcessor.ProcessWebhook()`
- **Metrics Collection**: Update analytics via `DeliveryTracker.ProcessEvent()`

This glossary serves as both a reference during development and a training resource for new team members. Keep it updated as the system evolves and new concepts are introduced.